The Concurrency Utilities

Java’s original support for multithreading is, it is not

ideal for all applications—especially those that make intensive use of multiple threads. For

example, the original multithreading support does not provide several high-level features,

such as semaphores, thread pools, and execution managers, that facilitate the creation of

intensively concurrent programs.

The term concurrent program refers

to a program that makes extensive, integral use of concurrently executing threads. An example

of such a program is one that uses separate threads to simultaneously compute the partial

results of a larger computation. Another example is a program that coordinates the activities

of several threads, each of which seeks access to information in a database.

To begin to handle the needs of a concurrent program, JDK 5 added the concurrency

utilities, also commonly referred to as the concurrent API. The original set of concurrency

utilities supplied many features that had long been wanted by programmers who develop

concurrent applications. For example, it offered synchronizers (such as the semaphore),

thread pools, execution managers, locks, several concurrent collections, and a streamlined

way to use threads to obtain computational results.

Although the original concurrent API was impressive in its own right, it was significantly expanded by JDK 7. The most important addition was the Fork/Join Framework. The Fork/Join Framework facilitates the creation of programs that make use of multiple processors (such as those found in multicore systems). Thus, it streamlines the development of programs in which two or more pieces execute with true simultaneity (that is, true parallel execution),not just time-slicing.

**The Concurrent API Packages**

The concurrency utilities are contained in the **java.util.concurrent** package and in its two

subpackages: **java.util.concurrent.atomic** and **java.util.concurrent.locks**

java.util.concurrent

java.util.concurrent defines the core features that support alternatives to the built-in

approaches to synchronization and interthread communication. It defines the following

key features:

• Synchronizers

• Executors

• Concurrent collections

• The Fork/Join Framework

Synchronizers offer high-level ways of synchronizing the interactions between multiple

threads. The synchronizer classes defined by java.util.concurrent are

|  |  |
| --- | --- |
| Semaphore | Implements the classic semaphore. |
| CountDownLatch | Waits until a specified number of events have occurred. |
| Exchanger | Exchanges data between two threads. |
| Phaser | Synchronizes threads that advance through multiple phases of an  operation. |

*Executors* manage thread execution. At the top of the executor hierarchy is the Executor

interface, which is used to initiate a thread. **ExecutorService** extends **Executor** and provides

methods that manage execution. There are three implementations of **ExecutorService**:

**ThreadPoolExecutor**, **ScheduledThreadPoolExecutor**, and **ForkJoinPool**. java.util.concurrent

also defines the Executors utility class, which includes a number of static methods that

simplify the creation of various executors.

Related to executors are the **Future** and **Callable** interfaces. A **Future** contains a value

that is returned by a thread after it executes. Thus, its value becomes defined “in the

future,” when the thread terminates. **Callable** defines a thread that returns a value

**java.util.concurrent** defines several concurrent collection classes, including

**ConcurrentHashMap**, **ConcurrentLinkedQueue**, and **CopyOnWriteArrayList**. These offer

concurrent alternatives to their related classes defined by the Collections Framework.

The *Fork/Join Framework* supports parallel programming. Its main classes are **ForkJoinTask**,

**ForkJoinPool**, **RecursiveTask**, and **RecursiveAction**.

**java.util.concurrent.atomic**

**java.util.concurrent.atomic** facilitates the use of variables in a concurrent environment.

It provides a means of efficiently updating the value of a variable without the use of locks.

This is accomplished through the use of classes, such as **AtomicInteger** and **AtomicLong**,

and methods, such as **compareAndSet( )**, **decrementAndGet( )**, and **getAndSet( )**. These

methods execute as a single, non-interruptible operation.

**java.util.concurrent.locks**

**java.util.concurrent.locks** provides an alternative to the use of synchronized methods. At

the core of this alternative is the **Lock** interface, which defines the basic mechanism used

to acquire and relinquish access to an object. The key methods are **lock( )**, **tryLock( )**, and

**unlock( )**. The advantage to using these methods is greater control over synchronization.

The remainder of this chapter takes a closer look at the constituents of the concurrent API.

**Semaphore**

A semaphore controls access to a shared resource

through the use of a counter. If the counter is greater than zero, then access is allowed. If

it is zero, then access is denied. What the counter is counting are *permits* that allow access to

the shared resource. Thus, to access the resource, a thread must be granted a permit from

the semaphore.

In general, to use a semaphore, the thread that wants access to the shared resource tries

to acquire a permit. If the semaphore’s count is greater than zero, then the thread acquires

a permit, which causes the semaphore’s count to be decremented. Otherwise, the thread

will be blocked until a permit can be acquired.

When the thread no longer needs access

to the shared resource, it releases the permit, which causes the semaphore’s count to be

incremented. If there is another thread waiting for a permit, then that thread will acquire

a permit at that time. Java’s **Semaphore** class implements this mechanism.

**Semaphore** has the two constructors shown here:

Semaphore(int *num*)

Semaphore(int *num*, boolean *how*)

Here, *num* specifies the initial permit count. Thus, *num* specifies the number of threads

that can access a shared resource at any one time.

By default, waiting threads are granted a permit in an

undefined order. By setting *how* to **true**, you can ensure that waiting threads are granted a

permit in the order in which they requested access.

To acquire a permit, call the **acquire( )** method, which has these two forms:

void acquire( ) throws InterruptedException

void acquire(int *num*) throws InterruptedException

The first form acquires one permit. The second form acquires *num* permits. Most often, the

first form is used. If the permit cannot be granted at the time of the call, then the invoking

thread suspends until the permit is available.

To release a permit, call **release( )**, which has these two forms:

void release( )

void release(int *num*)

The first form releases one permit. The second form releases the number of permits

specified by *num.*

**CountDownLatch**

**CountDownLatch** is

initially created with a count of the number of events that must occur before the latch is

released. Each time an event happens, the count is decremented. When the count reaches

zero, the latch opens.

**CountDownLatch** has the following constructor:

CountDownLatch(int *num*)

Here, *num* specifies the number of events that must occur in order for the latch to open.

To wait on the latch, a thread calls **await( )**, which has the forms shown here:

void await( ) throws InterruptedException

boolean await(long *wait*, TimeUnit *tu*) throws InterruptedException

The first form waits until the count associated with the invoking **CountDownLatch** reaches

zero. The second form waits only for the period of time specified by *wait.* The units

represented by *wait* are specified by *tu,* which is an object the **TimeUnit** enumeration.

To signal an event, call the **countDown( )** method, shown next:

void countDown( )

Each call to **countDown( )** decrements the count associated with the invoking object.

**CyclicBarrier**

A situation not uncommon in concurrent programming occurs when a set of two or more

threads must wait at a predetermined execution point until all threads in the set have

reached that point. To handle such a situation, the concurrent API supplies the **CyclicBarrier**

class. It enables you to define a synchronization object that suspends until the specified

number of threads has reached the barrier point.

**CyclicBarrier** has the following two constructors:

CyclicBarrier(int *numThreads*)

CyclicBarrier(int *numThreads*, Runnable *action*)

Here, *numThreads* specifies the number of threads that must reach the barrier before

execution continues. In the second form, *action* specifies a thread that will be executed

when the barrier is reached.

Here is the general procedure that you will follow to use **CyclicBarrier**. First, create a

**CyclicBarrier** object, specifying the number of threads that you will be waiting for. Next,

when each thread reaches the barrier, have it call **await( )** on that object. This will pause

execution of the thread until all of the other threads also call **await( )**. Once the specified

number of threads has reached the barrier, **await( )** will return and execution will resume.

Also, if you have specified an action, then that thread is executed.

The **await( )** method has the following two forms:

int await( ) throws InterruptedException, BrokenBarrierException

int await(long *wait*, TimeUnit *tu*)

throws InterruptedException, BrokenBarrierException, TimeoutException

The first form waits until all the threads have reached the barrier point. The second form

waits only for the period of time specified by *wait.* The units represented by *wait* are

specified by *tu.* Both forms return a value that indicates the order that the threads arrive

at the barrier point. The first thread returns a value equal to the number of threads waited

upon minus one. The last thread returns zero.

**Exchanger**

Perhaps the most interesting of the synchronization classes is **Exchanger**. It is designed

to simplify the exchange of data between two threads. The operation of an **Exchanger** is

astoundingly simple: it simply waits until two separate threads call its **exchange( )** method.

When that occurs, it exchanges the data supplied by the threads. This mechanism is both

elegant and easy to use. Uses for **Exchanger** are easy to imagine. For example, one thread

might prepare a buffer for receiving information over a network connection. Another

thread might fill that buffer with the information from the connection. The two threads

work together so that each time a new buffer is needed, an exchange is made.

**Exchanger** is a generic class that is declared as shown here:

Exchanger<V>

Here, **V** specifies the type of the data being exchanged.

The only method defined by **Exchanger** is **exchange( )**, which has the two forms

shown here:

V exchange(V *objRef*) throws InterruptedException

V exchange(V *objRef*, long *wait*, TimeUnit *tu*)

throws InterruptedException, TimeoutException

Here, *objRef* is a reference to the data to exchange. The data received from the other thread

is returned. The second form of **exchange( )** allows a time-out period to be specified. The

key point about **exchange( )** is that it won’t succeed until it has been called on the same

**Exchanger** object by two separate threads. Thus, **exchange( )** synchronizes the exchange

of the data.

**Phaser**

Its primary purpose is to enable the

synchronization of threads that represent one or more phases of activity. For example, you

might have a set of threads that implement three phases of an order-processing application.

In the first phase, separate threads are used to validate customer information, check

inventory, and confirm pricing. When that phase is complete, the second phase has two

threads that compute shipping costs and all applicable tax. After that, a final phase confirms

payment and determines estimated shipping time. In the past, to synchronize the multiple

threads that comprise this scenario would require a bit of work on your part. With the

inclusion of **Phaser**, the process is now much easier.

Constructors:

Phaser( )

Phaser(int *numParties*)

The first creates a phaser that has a registration count of zero. The second sets the

registration count to *numParties*. The term *party* is often applied to the objects that register

with a phaser. Although often there is a one-to-correspondence between the number of

registrants and the number of threads being synchronized, this is not required. In both

cases, the current phase is zero. That is, when a **Phaser** is created, it is initially at phase zero.

In general, here is how you use **Phaser**. First, create a new instance of **Phaser**. Next,

register one or more parties with the phaser, either by calling **register( )** or by specifying the

number of parties in the constructor. For each registered party, have the phaser wait until

all registered parties complete a phase. A party signals this by calling one of a variety of

methods supplied by **Phaser**, such as **arrive( )** or **arriveAndAwaitAdvance( )**. After all parties

have arrived, the phase is complete, and the phaser can move on to the next phase (if there

is one), or terminate.

To register parties(thread) after/with a **Phaser** has been constructed, call **register( )**. It is shown here:

int register()

It returns the phase number of the phase to which it is registered.

To signal that a party has completed a phase, it must call **arrive( )** or some variation of

**arrive( )**. When the number of arrivals equals the number of registered parties, the phase is

completed and the **Phaser** moves on to the next phase (if there is one). The **arrive( )** method

has this general form:

int arrive( )

This method signals that a party (normally a thread of execution) has completed some

task (or portion of a task). It returns the current phase number. If the phaser has been

terminated, then it returns a negative value.

The **arrive( )** method does not suspend execution of the calling thread. This means that it does not wait for the phase to be completed. This method should be called only by a registered party.

If you want to indicate the completion of a phase and then wait until all other registrants

have also completed that phase, use **arriveAndAwaitAdvance( )**. It is shown here:

**int arriveAndAwaitAdvance( )**

It waits until all parties have arrived. It returns the next phase number or a negative value if

the phaser has been terminated. This method should be called only by a registered party.

A thread can arrive and then deregister itself by calling **arriveAndDeregister( )**. It is

shown here:

**int arriveAndDeregister( )**

It returns the current phase number or a negative value if the phaser has been terminated. It

does not wait until the phase is complete. This method should be called only by a registered

party.

To obtain the current phase number, call **getPhase( )**, which is shown here:

**final int getPhase( )**

When a **Phaser** is created, the first phase will be 0, the second phase 1, the third phase 2,

and so on. A negative value is returned if the invoking **Phaser** has been terminated.

One other point: Although the preceding example used three threads that were all of

the same type, this is not a requirement. Each party that uses a phaser can be unique, with

each performing some separate task.

It is possible to take control of precisely what happens when a phase advance occurs. To

do this, you must override the **onAdvance( )** method. This method is called by the run time

when a **Phaser** advances from one phase to the next. It is shown here:

protected boolean onAdvance(int *phase*, int *numParties*)

Here, *phase* will contain the current phase number prior to being incremented and

*numParties* will contain the number of registered parties. To terminate the phaser,

**onAdvance( )** must return **true**. To keep the phaser alive, **onAdvance( )** must return **false**.

The default version of **onAdvance( )** returns **true** (thus terminating the phaser) when there

are no registered parties. As a general rule, your override should also follow this practice.

One reason to override **onAdvance( )** is to enable a phaser to execute a specific number

of phases and then stop.

**Using an Executor**

The concurrent API supplies a feature called an *executor* that initiates and controls the

execution of threads. As such, an executor offers an alternative to managing threads

through the **Thread** class.

At the core of an executor is the **Executor** interface. It defines the following method:

void execute(Runnable *thread*)

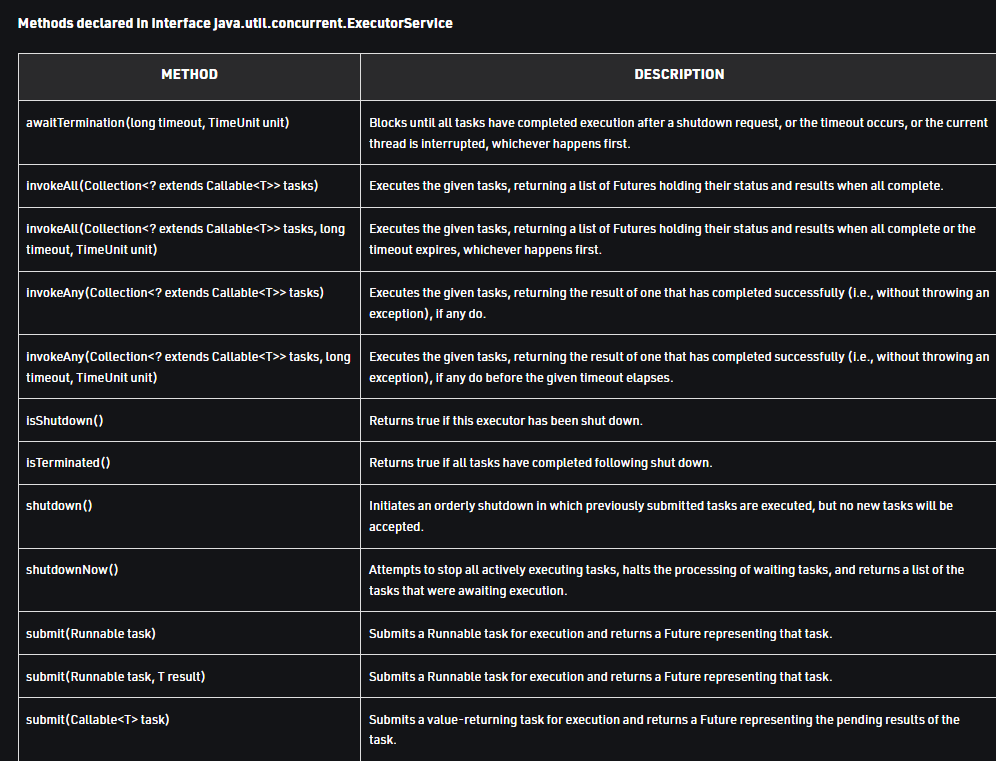
The thread specified by *thread* is executed. Thus, **execute( )** starts the specified thread.

The **ExecutorService** interface extends **Executor** by adding methods that help manage

and control the execution of threads. For example, **ExecutorService** defines **shutdown( )**,

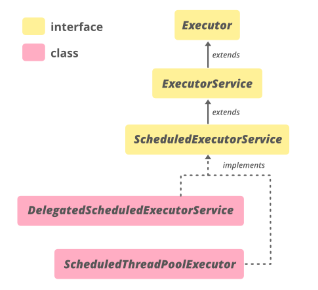
shown here, which stops the invoking **ExecutorService**.

void shutdown( )



**ScheduledExecutorService**, which extends **ExecutorService**

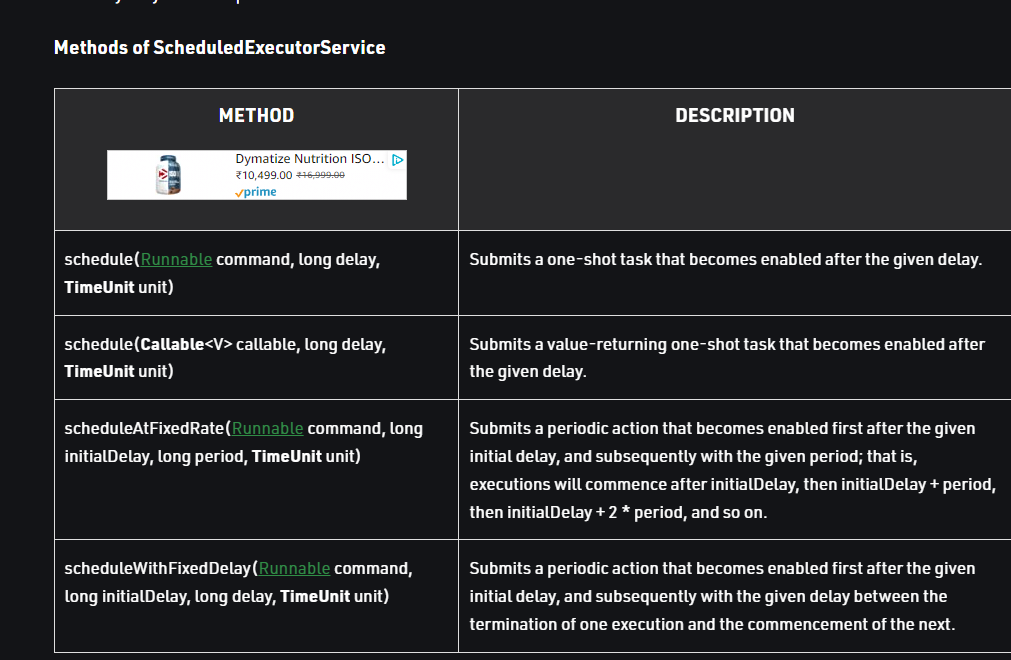
to support the scheduling of threads.



Since ScheduledExecutorService is an interface, so it cannot be instantiated. But the Executors class, defined in java.util.concurrent package, provides some factory methods that return ScheduledExecutorService objects( objects of its implementing classes )

**public static ScheduledExecutorService newScheduledThreadPool(int corePoolSize)** : Creates a new scheduled thread pool with a given core pool size (corePoolSize) and returns a ScheduledExecutorService object which can be downcasted to ScheduledThreadPoolExecutor object. This object can be used to run tasks after a given delay or to execute periodically.

**public static ScheduledExecutorService newScheduledThreadPool(int corePoolSize , ThreadFactory threadFactory)** : Creates a new scheduled thread pool with a given core pool size (corePoolSize) and returns a ScheduledExecutorService object which can be downcasted to ScheduledThreadPoolExecutor object. The second argument is a ThreadFactory object that is used when a new thread is created.



The concurrent API defines three predefined executor classes: **ThreadPoolExecutor**

and **ScheduledThreadPoolExecutor**, and **ForkJoinPool**. **ThreadPoolExecutor** implements

the **Executor** and **ExecutorService** interfaces and provides support for a managed pool of

threads. **ScheduledThreadPoolExecutor** also implements the **ScheduledExecutorService**

interface to allow a pool of threads to be scheduled. **ForkJoinPool** implements the **Executor**

and **ExecutorService** interfaces and is used by the Fork/Join Framework.

A thread pool provides a set of threads that is used to execute various tasks. Instead of

each task using its own thread, the threads in the pool are used. This reduces the overhead

associated with creating many separate threads. Although you can use **ThreadPoolExecutor**

and **ScheduledThreadPoolExecutor** directly, most often you will want to obtain an executor

by calling one of the following static factory methods defined by the **Executors** utility class.

Here are some examples:

static ExecutorService newCachedThreadPool( )

static ExecutorService newFixedThreadPool(int *numThreads*)

static ScheduledExecutorService newScheduledThreadPool(int *numThreads*)

**newCachedThreadPool( )** creates a thread pool that adds threads as needed but reuses

threads if possible. **newFixedThreadPool( )** creates a thread pool that consists of a specified

number of threads. **newScheduledThreadPool( )** creates a thread pool that supports thread

scheduling. Each returns a reference to an **ExecutorService** that can be used to manage

the pool.

**Using Callable and Future**

One of the most interesting features of the concurrent API is the **Callable** interface. This

interface represents a thread that returns a value. An application can use **Callable** objects to

compute results that are then returned to the invoking thread. This is a powerful mechanism

because it facilitates the coding of many types of numerical computations in which partial

results are computed simultaneously. It can also be used to run a thread that returns a

status code that indicates the successful completion of the thread.

**Callable** is a generic interface that is defined like this:

*interface Callable<V>*

Here, **V** indicates the type of data returned by the task. **Callable** defines only one method,

**call( )**, which is shown here:

*V call( ) throws Exception*

Inside **call( )**, you define the task that you want performed. After that task completes, you

return the result. If the result cannot be computed, **call( )** must throw an exception.

A **Callable** task is executed by an **ExecutorService**, by calling its **submit( )** method. There

are three forms of **submit( )**, but only one is used to execute a **Callable**. It is shown here:

*<T> Future<T> submit(Callable<T> task)*

Here, *task* is the **Callable** object that will be executed in its own thread. The result is

returned through an object of type **Future**.

**Future** is a generic interface that represents the value that will be returned by a **Callable**

object. Because this value is obtained at some future time, the name **Future** is appropriate.

**Future** is defined like this:

*interface Future<V>*

Here, **V** specifies the type of the result.

To obtain the returned value, you will call **Future**’s **get( )** method, which has these two

forms:

*V get( )*

*throws InterruptedException, ExecutionException*

*V get(long wait, TimeUnit tu)*

*throws InterruptedException, ExecutionException, TimeoutException*

The first form waits for the result indefinitely. The second form allows you to specify a

timeout period in *wait.* The units of *wait* are passed in *tu,* which is an object of the **TimeUnit**

enumeration

**The TimeUnit Enumeration**

The concurrent API defines several methods that take an argument of type **TimeUnit**,

which indicates a time-out period. **TimeUnit** is an enumeration that is used to specify the

*granularity* (or resolution) of the timing. **TimeUnit** is defined within **java.util.concurrent**. It

can be one of the following values:

DAYS

HOURS

MINUTES

SECONDS

MICROSECONDS

MILLISECONDS

NANOSECONDS

Although **TimeUnit** lets you specify any of these values in calls to methods that take a

timing argument, there is no guarantee that the system is capable of the specified

resolution.

Here is an example that uses **TimeUnit**. The **CallableDemo** class, shown in the previous

section, is modified as shown next to use the second form of **get( )** that takes a **TimeUnit**

argument.

try {

System.out.println(f.get(10, TimeUnit.MILLISECONDS));

System.out.println(f2.get(10, TimeUnit.MILLISECONDS));

System.out.println(f3.get(10, TimeUnit.MILLISECONDS));

} catch (InterruptedException exc) {

System.out.println(exc);

}

catch (ExecutionException exc) {

System.out.println(exc);

} catch (TimeoutException exc) {

System.out.println(exc);

}

In this version, no call to **get( )** will wait more than 10 milliseconds.

The **TimeUnit** enumeration defines various methods that convert between units. These

are shown here:

long convert(long *tval*, TimeUnit *tu*)

long toMicros(long *tval*)

long toMillis(long *tval*)

long toNanos(long *tval*)

long toSeconds(long *tval*)

long toDays(long *tval*)

long toHours(long *tval*)

long toMinutes(long *tval*)

The **convert( )** method converts *tval* into the specified unit and returns the result. The **to**

methods perform the indicated conversion and return the result.

**TimeUnit** also defines the following timing methods:

void sleep(long *delay*) throws InterruptedExecution

void timedJoin(Thread *thrd*, long *delay*) throws InterruptedExecution

void timedWait(Object *obj*, long *delay*) throws InterruptedExecution

Here, **sleep( )** pauses execution for the specified delay period, which is specified in terms of

the invoking enumeration constant. It translates into a call to **Thread.sleep( )**. The **timedJoin( )**

method is a specialized version of **Thread.join( )** in which *thrd* pauses for the time period

specified by *delay,* which is described in terms of the invoking time unit. The **timedWait( )**

method is a specialized version of **Object.wait( )** in which *obj* is waited on for the period of

time specified by *delay,* which is described in terms of the invoking time unit.

**Locks**

The **java.util.concurrent.locks** package provides support for *locks,* which are objects that

offer an alternative to using **synchronized** to control access to a shared resource. In general,

here is how a lock works. Before accessing a shared resource, the lock that protects that resource is acquired. When access to the resource is complete, the lock is released. If a second thread attempts to acquire the lock when it is in use by another thread, the second thread will suspend until the lock is released. In this way, conflicting access to a shared resource is prevented. Locks are particularly useful when multiple threads need to access the value of shared

data.

In general, to acquire a lock, call **lock( )**. If the lock is unavailable, **lock( )** will wait. To

release a lock, call **unlock( )**. To see if a lock is available, and to acquire it if it is, call **tryLock( )**.

This method will not wait for the lock if it is unavailable. Instead, it returns **true** if the lock

is acquired and **false** otherwise. The **newCondition( )** method returns a **Condition** object

associated with the lock. Using a **Condition**, you gain detailed control of the lock through

methods such as **await( )** and **signal( )**, which provide functionality similar to **Object.wait( )**

and **Object.notify( )**.

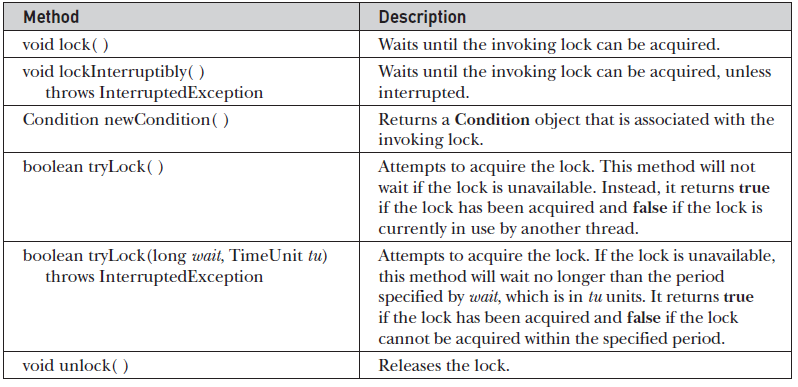
**java.util.concurrent.locks** supplies an implementation of **Lock** called **ReentrantLock**.

**ReentrantLock** implements a *reentrant lock,* which is a lock that can be repeatedly entered

by the thread that currently holds the lock. Of course, in the case of a thread reentering a

lock, all calls to **lock( )** must be offset by an equal number of calls to **unlock( )**. Otherwise,

a thread seeking to acquire the lock will suspend until the lock is not in use.



**Atomic Operations**

**java.util.concurrent.atomic** offers an alternative to the other synchronization features when

reading or writing the value of some types of variables. This package offers methods that

get, set, or compare the value of a variable in one uninterruptible (that is, atomic) operation.

This means that no lock or other synchronization mechanism is required.

Atomic operations are accomplished through the use of classes, such as **AtomicInteger**

and **AtomicLong**, and methods such as **get( )**, **set( )**, **compareAndSet( )**, **decrementAndGet( )**,

and **getAndSet( )**, which perform the action indicated by their names.

**Parallel Programming via the Fork/Join Framework**

In recent years, an important new trend has emerged in software development: *parallel*

*programming*. Parallel programming is the name commonly given to the techniques that

take advantage of computers that contain two or more processors (multicore). As most

readers will know, multicore computers are becoming commonplace. The advantage that

multi-processor environments offer is the ability to significantly increase program

performance. As a result, there has been a growing need for a mechanism that gives Java

programmers a simple, yet effective way to make use of multiple processors in a clean,

scalable manner. To answer this need, JDK 7 added several new classes and interfaces that

support parallel programming. They are commonly referred to as the *Fork/Join Framework*.

It is one of the more important additions that has recently been made to the Java class

library. The Fork/Join Framework is defined in the **java.util.concurrent** package.

The Fork/Join Framework enhances multithreaded programming in two important

ways. First, it simplifies the creation and use of multiple threads. Second, it automatically

makes use of multiple processors. In other words, by using the Fork/Join Framework you

enable your applications to automatically scale to make use of the number of available

processors. These two features make the Fork/Join Framework the recommended

approach to multithreading when parallel processing is desired.

Before continuing, it is important to point out the distinction between traditional

multithreading and parallel programming. In the past, most computers had a single CPU

and multithreading was primarily used to take advantage of idle time, such as when a

program is waiting for user input. Using this approach, one thread can execute while

another is waiting. In other words, on a single-CPU system, multithreading is used to allow

two or more tasks to share the CPU. This type of multithreading is typically supported by an

object of type **Thread** .Although this type of multithreading

will always remain quite useful, it was not optimized for situations in which two or more

CPUs are available (multicore computers).

When multiple CPUs are present, a second type of multithreading capability that

supports true parallel execution is required. With two or more CPUs, it is possible to

execute portions of a program simultaneously, with each part executing on its own CPU.

This can be used to significantly speed up the execution of some types of operations, such

as sorting, transforming, or searching a large array. In many cases, these types of operations

can be broken down into smaller pieces (each acting on a portion of the array), and each

piece can be run on its own CPU. As you can imagine, the gain in efficiency can be enormous.

Simply put: Parallel programming will be part of nearly every programmer’s future because

it offers a way to dramatically improve program performance.

The Fork/Join Framework is packaged in **java.util.concurrent**. At the core of the Fork/Join

Framework are the following four classes:

ForkJoinTask<V>: An abstract class that defines a task

ForkJoinPool: Manages the execution of **ForkJoinTask**s

RecursiveAction: A subclass of **ForkJoinTask<V>** for tasks that do not return values

RecursiveTask<V>: A subclass of **ForkJoinTask<V>** for tasks that return values

Here is how they relate. A **ForkJoinPool** manages the execution of **ForkJoinTask**s. **ForkJoinTask**

is an abstract class that is extended by the abstract classes **RecursiveAction** and **RecursiveTask**.

Typically, your code will extend these classes to create a task.

**ForkJoinTask<V>**

**ForkJoinTask<V>** is an abstract class that defines a task that can be managed by a **ForkJoinPool**.

The type parameter **V** specifies the result type of the task. **ForkJoinTask** differs from **Thread**

in that **ForkJoinTask** represents lightweight abstraction of a task, rather than a thread of

execution. **ForkJoinTask**s are executed by threads managed by a thread pool of type

**ForkJoinPool**. This mechanism allows a large number of tasks to be managed by a small

number of actual threads. Thus, **ForkJoinTask**s are very efficient when compared to threads.

**ForkJoinTask** defines many methods. At the core are **fork( )** and **join( )**, shown here:

final ForkJoinTask<V> fork( )

final V join( )

task. This means that the thread that calls **fork( )** continues to run. The **fork( )** method

returns **this** after the task is scheduled for execution. Prior to JDK 8, **fork( )** could be executed

only from within the computational portion of another **ForkJoinTask**, which is running

within a **ForkJoinPool**. However, with the advent of JDK 8, if **fork( )** is not called while executing within

a **ForkJoinPool**, then a common pool is automatically used. The **join( )** method waits until the

task on which it is called terminates. The result of the task is returned. Thus, through the use

of **fork( )** and **join( )**, you can start one or more new tasks and then wait for them to finish.

Another important **ForkJoinTask** method is **invoke( )**. It combines the fork and join

operations into a single call because it begins a task and then waits for it to end. It is shown

here:

**final V invoke( )**

The result of the invoking task is returned.

You can invoke more than one task at a time by using **invokeAll( )**. Two of its forms are

shown here:

**static void invokeAll(ForkJoinTask<?> *taskA*, ForkJoinTask<?> *taskB*)**

**static void invokeAll(ForkJoinTask<?> ... *taskList*)**

In the first case, *taskA* and *taskB* are executed. In the second case, all specified tasks are

executed. In both cases, the calling thread waits until all of the specified tasks have terminated.

Prior to JDK 8, the **invokeAll( )** method could be executed only from within the computational

portion of another **ForkJoinTask**, which is running within a **ForkJoinPool**. JDK 8’s inclusion

of the common pool relaxed this requirement.

**RecursiveAction**

A subclass of **ForkJoinTask** is **RecursiveAction**. This class encapsulates a task that does not

return a result. Typically, your code will extend **RecursiveAction** to create a task that has a

**void** return type. **RecursiveAction** specifies four methods, but only one is usually of interest:

the abstract method called **compute( )**. When you extend **RecursiveAction** to create a concrete

class, you will put the code that defines the task inside **compute( )**. The **compute( )** method

represents the ***computational* portion of the task.**

The **compute( )** method is defined by **RecursiveAction** like this:

protected abstract void compute( )

Notice that **compute( )** is **protected** and **abstract**. This means that it must be implemented

by a subclass

**RecursiveTask<V>**

Another subclass of **ForkJoinTask** is **RecursiveTask<V>**. This class encapsulates a task

that returns a result. The result type is specified by **V**. Typically, your code will extend

**RecursiveTask<V>** to create a task that returns a value. Like **RecursiveAction**, it too specifies

four methods, but often only the abstract **compute( )** method is used, which represents the

computational portion of the task. When you extend **RecursiveTask<V>** to create a concrete

class, put the code that represents the task inside **compute( )**. This code must also return

the result of the task.

The **compute( )** method is defined by **RecursiveTask<V>** like this:

protected abstract V compute( )

Notice that **compute( )** is **protected** and **abstract**. This means that it must be implemented

by a subclass. When implemented, it must return the result of the task.

**ForkJoinPool**

The execution of **ForkJoinTask**s takes place within a **ForkJoinPool**, which also manages the

execution of the tasks. Therefore, in order to execute a **ForkJoinTask**, you must first have a

**ForkJoinPool**. Beginning with JDK 8, there are two ways to acquire a **ForkJoinPool**. First,

you can explicitly create one by using a **ForkJoinPool** constructor. Second, you can use what

is referred to as the *common pool*. The common pool (which was added by JDK 8) is a static

**ForkJoinPool** that is automatically available for your use. Each method is introduced here,

beginning with manually constructing a pool.

**ForkJoinPool** defines several constructors. Here are two commonly used ones:

ForkJoinPool( )

ForkJoinPool(int *pLevel*)

The first creates a default pool that supports a level of parallelism equal to the number of

processors available in the system. The second lets you specify the level of parallelism. Its

value must be greater than zero and not more than the limits of the implementation. The

level of parallelism determines the number of threads that can execute concurrently. As a

result, the level of parallelism effectively determines the number of tasks that can be

executed simultaneously. (Of course, the number of tasks that can execute simultaneously

cannot exceed the number of processors.) It is important to understand that the level of

parallelism *does not,* however, limit the number of tasks that can be managed by the pool.

A **ForkJoinPool** can manage many more tasks than its level of parallelism. Also, the level

of parallelism is only a target. It is not a guarantee.

After you have created an instance of **ForkJoinPool**, you can start a task in a number

of different ways. The first task started is often thought of as the main task. Frequently, the

main task begins subtasks that are also managed by the pool. One common way to begin a

main task is to call **invoke( )** on the **ForkJoinPool**. It is shown here:

<T> T invoke(ForkJoinTask<T> *task*)

This method begins the task specified by *task*, and it returns the result of the task. This

means that the calling code waits until **invoke( )** returns.

To start a task without waiting for its completion, you can use **execute( )**. Here is one

of its forms:

void execute(ForkJoinTask<?> *task*)

In this case, *task* is started, but the calling code does not wait for its completion. Rather, the

calling code continues execution asynchronously.

Beginning with JDK 8, it is not necessary to explicitly construct a **ForkJoinPool** because

a common pool is available for your use. In general, if you are not using a pool that you

explicitly created, then the common pool will automatically be used. Although it won’t always

be necessary, you can obtain a reference to the common pool by calling **commonPool( )**,

which is defined by **ForkJoinPool**. It is shown here:

static ForkJoinPool commonPool( )

A reference to the common pool is returned. The common pool provides a default level of

parallelism. It can be set by use of a system property.

of different ways. The first task started is often thought of as the main task. Frequently, the

main task begins subtasks that are also managed by the pool. One common way to begin a

main task is to call **invoke( )** on the **ForkJoinPool**. It is shown here:

<T> T invoke(ForkJoinTask<T> *task*)

This method begins the task specified by *task*, and it returns the result of the task. This

means that the calling code waits until **invoke( )** returns.

To start a task without waiting for its completion, you can use **execute( )**. Here is one

of its forms:

void execute(ForkJoinTask<?> *task*)

In this case, *task* is started, but the calling code does not wait for its completion. Rather, the

calling code continues execution asynchronously.

Beginning with JDK 8, it is not necessary to explicitly construct a **ForkJoinPool** because

a common pool is available for your use. In general, if you are not using a pool that you

explicitly created, then the common pool will automatically be used. Although it won’t always

be necessary, you can obtain a reference to the common pool by calling **commonPool( )**,

which is defined by **ForkJoinPool**. It is shown here:

static ForkJoinPool commonPool( )

A reference to the common pool is returned. The common pool provides a default level of

parallelism. It can be set by use of a system property.

Typically, the default common pool is a good choice for many applications. Of

course, you can always construct your own pool.

There are two basic ways to start a task using the common pool. First, you can obtain a

reference to the pool by calling **commonPool( )** and then use that reference to call **invoke( )**

or **execute( )**, as just described. Second, you can call **ForkJoinTask** methods such as **fork( )**

or **invoke( )** on the task from outside its computational portion. In this case, the common

pool will automatically be used. In other words, **fork( )** and **invoke( )** will start a task using

the common pool if the task is not already running within a **ForkJoinPool**.

**ForkJoinPool** manages the execution of its threads using an approach called ***work-stealing***.

Each worker thread maintains a queue of tasks. If one worker thread’s queue is empty, it

will take a task from another worker thread. This adds to overall efficiency and helps

maintain a balanced load. (Because of demands on CPU time by other processes in the system,

even two worker threads with identical tasks in their respective queues may not complete at

the same time.)

**The Divide-and-Conquer Strategy**

As a general rule, users of the Fork/Join Framework will employ a *divide-and-conquer* strategy

that is based on recursion. This is why the two subclasses of **ForkJoinTask** are called

**RecursiveAction** and **RecursiveTask**. It is anticipated that you will extend one of these

classes when creating your own fork/join task.

The divide-and-conquer strategy is based on recursively dividing a task into smaller

subtasks until the size of a subtask is small enough to be handled sequentially. For example,

a task that applies a transform to each element in an array of *N* integers can be broken

down into two subtasks in which each transforms half the elements in the array. That is,

one subtask transforms the elements 0 to *N*/2, and the other transforms the elements *N*/2

to *N*. In turn, each subtask can be reduced to another set of subtasks, each transforming

half of the remaining elements. This process of dividing the array will continue until a

threshold is reached in which a sequential solution is faster than creating another division.

The advantage of the divide-and-conquer strategy is that the processing can occur in

parallel. Therefore, instead of cycling through an entire array using a single thread, pieces

of the array can be processed simultaneously. Of course, the divide-and-conquer approach

works in many cases in which an array (or collection) is not present, but the most common

uses involve some type of array, collection, or grouping of data.

One of the keys to best employing the divide-and-conquer strategy is correctly selecting

the threshold at which sequential processing (rather than further division) is used. Typically,

an optimal threshold is obtained through profiling the execution characteristics. However,

very significant speed-ups will still occur even when a less-than-optimal threshold is used. It

is, however, best to avoid overly large or overly small thresholds. At the time of this writing,

the Java API documentation for **ForkJoinTask<T>** states that, as a rule-of-thumb, a task

should perform somewhere between 100 and 10,000 computational steps.

It is also important to understand that the optimal threshold value is also affected by

how much time the computation takes. If each computational step is fairly long, then

smaller thresholds might be better. Conversely, if each computational step is quite short,

then larger thresholds could yield better results. For applications that are to be run on a

known system, with a known number of processors, you can use the number of processors

to make informed decisions about the threshold value. However, for applications that will

be running on a variety of systems, the capabilities of which are not known in advance, you

can make no assumptions about the execution environment.

One other point: Although multiple processors may be available on a system, other

tasks (and the operating system, itself) will be competing with your application for CPU

time. Thus, it is important not to assume that your program will have unrestricted access to

all CPUs. Furthermore, different runs of the same program may display different run time

characteristics because of varying task loads.

**Executing a Task Asynchronously**

The preceding programs have called **invoke( )** on a **ForkJoinPool** to initiate a task. This

approach is commonly used when the calling thread must wait until the task has completed

(which is often the case) because **invoke( )** does not return until the task has terminated.

However, you can start a task asynchronously. In this approach, the calling thread continues

to execute. Thus, both the calling thread and the task execute simultaneously. To start a

task asynchronously, use **execute( )**, which is also defined by **ForkJoinPool**. It has the two

forms shown here:

void execute(ForkJoinTask<?> *task*)

void execute(Runnable *task*)

In both forms, *task* specifies the task to run. Notice that the second form lets you specify a

**Runnable** rather than a **ForkJoinTask** task. Thus, it forms a bridge between Java’s traditional

approach to multithreading and the new Fork/Join Framework. It is important to remember

that the threads used by a **ForkJoinPool** are daemon. Thus, they will end when the main

thread ends. As a result, you may need to keep the main thread alive until the tasks have

finished.

**Cancelling a Task**

A task can be cancelled by calling **cancel( )**, which is defined by **ForkJoinTask**. It has this

general form:

boolean cancel(boolean *interuptOK*)

It returns **true** if the task on which it was called is cancelled. It returns **false** if the task has

ended or can’t be cancelled. At this time, the *interruptOK* parameter is not used by the

default implementation. In general, **cancel( )** is intended to be called from code outside

the task because a task can easily cancel itself by returning.

You can determine if a task has been cancelled by calling **isCancelled( )**, as shown here:

final boolean isCancelled( )

It returns **true** if the invoking task has been cancelled prior to completion and **false**

otherwise.

**Determining a Task’s Completion Status**

In addition to **isCancelled( )**, which was just described, **ForkJoinTask** includes two other

methods that you can use to determine a task’s completion status. The first is

**isCompletedNormally( )**, which is shown here:

final boolean isCompletedNormally( )

It returns **true** if the invoking task completed normally, that is, if it did not throw an

exception and it was not cancelled via a call to **cancel( )**. It returns **false** otherwise.

The second is **isCompletedAbnormally( )**, which is shown here:

final boolean isCompletedAbnormally( )

It returns **true** if the invoking task completed because it was cancelled or because it threw

an exception. It returns **false** otherwise.