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. |
| CyclicBarrier | 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**