**Synchronizers**

Synchronizers are high-level constructs that coordinate and control thread execution.

The Java Concurrency Utilities framework provides classes that implement semaphore,

cyclic barrier, countdown latch, exchanger, and phaser synchronizers.

**CountDownLatch :**

Scenario: A main thread creates 3 database connections and assigns each of those

connection to 3 different child threads that are spawned from the main thread.

The main thread must wait while all the child threads are completed and

then close all the database connections. So, how will you accomplish this?

Solution: This is where the CountDownLatch comes in handy as you already know that

there are finite (i.e 3) number of threads. CountDownLatch can be used by

the main thread to wait on the child threads.

A CountDownLatch will be created with 3 being the count.

A java.util.concurrent.CountDownLatch is a concurrency construct that allows

one or more threads to wait for a given set of operations to complete.

A CountDownLatch is initialized with a given count. This count is decremented by calls

to the countDown() method. Threads waiting for this count to reach zero can call one of

the await() methods. Calling await() blocks the thread until the count reaches zero.

A CountDownLatch is a versatile synchronization tool and can be used for a number of purposes.

A CountDownLatch initialized with a count of one serves as a simple on/off latch, or gate: i.e.

all threads invoking await wait at the gate until it is opened by a thread invoking countDown().

A CountDownLatch initialized to N can be used to make one thread wait until N threads have completed

some action, or some action has been completed N times.

#### Refer to the Demo :

JavaConcurrency\src\com\seed\concurrent\synchronizers\DemoCountDownLatch.java

**Barriers**

The java.util.concurrent.CyclicBarrier class is a synchronization mechanism that can synchronize threads progressing through some algorithm. In other words, it is a barrier that all threads must wait at, until all threads reach it, before any of the threads can continue. Here is a diagram illustrating that:

|  |
| --- |
| Two threads waiting for each other at CyclicBarriers. |
| **Two threads waiting for each other at CyclicBarriers.** |

The threads wait for each other by calling the await() method on the CyclicBarrier. Once N threads are waiting at the CyclicBarrier, all threads are released and can continue running.

**Creating a CyclicBarrier**

When you create a CyclicBarrier you specify how many threads are to wait at it, before releasing them. Here is how you create a CyclicBarrier:

CyclicBarrier barrier = new CyclicBarrier(2);

**Waiting at a CyclicBarrier**

Here is how a thread waits at a CyclicBarrier:

barrier.await();

You can also specify a timeout for the waiting thread. When the timeout has passed the thread is also released, even if not all N threads are waiting at the CyclicBarrier. Here is how you specify a timeout:

barrier.await(10, TimeUnit.SECONDS);

The waiting threads waits at the CyclicBarrier until either:

* The last thread arrives (calls await() )
* The thread is interrupted by another thread (another thread calls its interrupt() method)
* Another waiting thread is interrupted
* Another waiting thread times out while waiting at the CyclicBarrier
* The CyclicBarrier.reset() method is called by some external thread.

#### Refer to the following Demo :

JavaConcurrency\src\com\seed\concurrent\synchronizers\DemoCyclicBarrier.java

**Phaser**

A *phaser* is a thread-synchronization construct that's similar to a cyclic barrier as it also offers the equivalent of a barrier action. However, a phaser is more flexible.

Unlike a cyclic barrier, which coordinates a fixed number of threads, a phaser can coordinate a variable number of threads, which can register at any time. To implement this capability, a phaser takes advantage of phases and phase numbers.

A *phase* is the phaser's current state, and this state is identified by an integer-based *phase number*. When the last of the registered threads arrives at the *phaser barrier*, a phaser advances to the next phase and increments its phase number by 1.

The java.util.concurrent.Phaser class implements a phaser.

* The Phaser(int threads) constructor creates a phaser that initially coordinates nthreads threads (which have yet to arrive at the phaser barrier) and whose phase number is initially set to 0.
* The int register() method adds a new unarrived thread to this phaser and returns the phase number to which the arrival applies. This number is known as the *arrival phase number*.
* The int arriveAndAwaitAdvance() method records arrival and waits for the phaser to advance (which happens after the other threads have arrived). It returns the phase number to which the arrival applies.

|  |
| --- |
| The int arriveAndDeregister() method arrives at this phaser and deregisters from it without waiting for others to arrive, reducing the number of threads required to advance in future phases.  #### Refer to the following Demo  JavaConcurrency\src\com\seed\concurrent\synchronizers\DemoPhaser.java  **Exchanger :**  An *exchanger* (also known as a *rendezvous i.e. a kind of meeting point*) is a thread-synchronization construct that lets a pair of threads exchange data items. An exchanger is similar to a cyclic barrier whose count is set to 2 but also supports exchange of data when both threads reach the barrier.  Here is an illustration of this mechanism:  Two threads exchanging objects via an Exchanger. |
| **Two threads exchanging objects via an Exchanger.** |

* Exchanging objects is done via one of the two exchange() methods.

### Working with exchangers

Exchanger's Javadoc states that this synchronizer may be useful in genetic algorithms and pipeline designs, where one thread fills a buffer and the other thread empties the buffer. When both threads meet at the exchange point, they swap their buffers.

#### Refer to the following Demo

JavaConcurrency\src\com\seed\concurrent\synchronizers\DemoExchanger.java

Which internally uses MoneyExchanger.java

### Semaphores

A **semaphore** is a thread-synchronization construct for controlling thread access to a common resource. It's often implemented as a protected variable whose value is incremented by an *acquire* operation and decremented by a *release* operation.

The acquire operation either returns control to the invoking thread immediately or causes that thread to block when the semaphore's current value reaches a certain limit. The release operation decreases the current value, which causes a blocked thread to resume.

Semaphores whose current values can be incremented past 1 are known as **counting semaphores**, whereas semaphores whose current values can be only 0 or 1 are known as ***binary semaphores* or mutexes**. In either case, the current value cannot be negative.

The java.lang.concurrent.Semaphore class conceptualizes a semaphore as an object maintaining a set of *permits*. This class provides Semaphore(int permits) and Semaphore(int permits, boolean fair) constructors for specifying the number of permits.

Each call to the Semaphore's void acquire() method takes one of the available permits or blocks the calling thread when one isn't available. Each call to Semaphore's void release() method returns an available permit, potentially releasing a blocking acquirer thread.

### Working with semaphores

Semaphores are often used to restrict the number of threads that can access a resource.

#### Refer to the following Demo

JavaConcurrency\src\com\seed\concurrent\synchronizers\DemoSemaphore.java