**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

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- Enables a group of threads to wait at a predefined execution point
* Exchanger- Exchanges data between two threads

**Using Synchronization Objects:**

**Semaphore**, **CountDownLatch**, **CyclicBarrier**, and **Exchanger** classes

Semaphore:

* 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)

num-specifies the number of threads that can access a shared resource at any one time

how-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 / /acquires one permit

void acquire(int num) throws InterruptedException // acquires num permits

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( ) // releases one permit

void release(int num) // releases the number of permits specified by num

**CountDownLatch:**

A 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*) // num specifies the number of events that must occur in order for the latch to open.*

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 // waits until the count associated with the invoking

CountDownLatch reaches zero.

void await(long wait, TimeUnit tu) throws InterruptedException // waits only for the period of time specified by tu

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:**

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.

Proceudre:

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

int await( ) throws InterruptedException, BrokenBarrierException

int await(long *wait*, TimeUnit *tu*) throws InterruptedException, BrokenBarrierException, TimeoutException

* The first form waits until the all 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:**

simplify the exchange of data between two threads.

**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 *buffer*) throws InterruptedException

V exchange(V *buffer*, long *wait*, TimeUnit *tu*) throws InterruptedException, TimeoutException

Here, *buffer* 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.

**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

**Using an Executor:**

that initiates and controls the execution of threads

**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( )

**ExecutorService** also defines methods that execute threads that return results, that execute

a set of threads, and that determine the shutdown status.

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:**

An application can use **Callable** objects to compute results that are then returned to the invoking 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

**Locks:**

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( )**

**ReentrantLock:**

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

by the thread that currently holds the lock