Java

Java Concurrency Utilities

Java Concurrency Utilities

- The concurrency utilities are contained in *java.util.concurrent*, *java.util.concurrent.atomic*, and *java.util.concurrent.locks* (all in the *java.base*)
- *java.util.concurrent* defines the core features that support alternatives to the built-in approaches to synchronization and interthread communication
  - Synchronizers
  - Executors
  - Concurrent Collections
  - The Fork/Join Framework

Synchronizers

- Synchronizers offer high-level ways of synchronizing the interactions between multiple threads
- Synchronization objects are supported by:
  - **Semaphore**
  - **CountDownLatch**
  - **CyclicBarrier**
  - **Exchanger**
  - **Phaser**
- Collectively, they enable to handle several formerly difficult synchronization situations with ease

Executors

- Executor initiates and controls the execution of threads
  - Executor offers an alternative to managing threads through the Thread class
- At the core of an executor is the Executor interface
  - The ExecutorService interface extends Executor by adding methods that help manage and control the execution of threads
  - Java provides Thread Pool implementation with ExecutorService

Thread Pool

- Thread Pools are useful when you need to limit the number of threads running in your application
  - Performance overhead starting a new thread
  - Each thread is also allocated some memory for its stack
- Instead of starting a new thread for every task to execute concurrently, the task can be passed to a thread pool
  - As soon as the pool has any idle threads the task is assigned to one of them and executed
- Thread pools are often used in multithreaded servers

ExecutorService

```java
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

class MyRunnable implements Runnable {
    public void run() {
        System.out.println("Running task");
        for (int j = 5; j > 0; j--) {
            System.out.println(j);
        }
    }
}

public class ExecutorServiceTest {
    public static void main(String[] args) {
        ExecutorService executorService = Executors.newFixedThreadPool( nThreads: 10);
        for (int i = 0; i < 20; i++) {
            executorService.execute(new MyRunnable());
        }
        executorService.shutdown();
    }
}
```

Callable and Future

- Runnable cannot return a result to the caller
- **Callable** object allows to return values after completion
- Callable task returns a **Future** object to return result
- The result can be obtained using get() that remains blocked until the result is computed
- Check completion by isDone(), cancel by cancel()
- ***Example****: CallableFutures.java*

Concurrent Collections

- java.util.concurrent defines several concurrent collection classes
  - **ConcurrentHashMap**
  - **BlockingQueue**
  - **BlockingQueue** etc.
- **BlockingQueue** can be used to solve the producer-consumer problem
  - No need to use wait(), notify(), notifyAll()
- ***Example****: PCBlockingQueue.java*

TimeUnit

- To better handle thread timing, java.util.concurrent defines the TimeUnit enumeration
  - The concurrent API defines several methods that take TimeUnit as argument, which indicates a time-out period
- TimeUnit is an enumeration that is used to specify the granularity (or resolution) of the timing
- It can be one of the following values:
  - DAYS, HOURS, MINUTES, SECONDS, MICROSECONDS, MILLISECONDS, NANOSECONDS
- **TimeUnit.SECONDS.sleep(1)** is same as **sleep(1000)**

Atomic

- 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 compare the value of a variable in one uninterruptible (atomic) operation
  - No lock or other synchronization mechanism is required
- Atomic operations are accomplished through:
- **Classes**: AtomicInteger, AtomicLong
- **Methods**: get(), set(), compareAndSet(), decrementAndGet(), incrementAndGet(), getAndSet() etc.

Lock

- java.util.concurrent.locks provides support for locks, which are objects that offer an alternative to using synchronized to control access to a shared resource
- The **Lock** interface defines a lock. The methods are:
  - 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, it returns true if acquired and false otherwise

Lock

- **ReentrantLock** is a lock that can be repeatedly entered by the thread that currently holds the lock
- **ReentrantReadWriteLock** is a **ReadWriteLock** that maintains separate locks for read and write access
  - Multiple locks are granted for readers of a resource as long as the resource is not being written
- The advantage to using these methods is greater control over synchronization
- ***Example**: SynchronizationLock.java*

The Fork/Join Framework

- Fork/Join framework supports parallel programming
- It enhances multithreaded programming
  - Simplifies the creation and use of multiple threads
  - Enables applications to automatically scale to make use of the number of available processors
- Recommended approach to multithreading when parallel processing is desired
- **Classes**: ForkJoinTask, ForkJoinPool, RecursiveTask, RecursiveAction
- ***Example****: ForkJoinTest.java*