Java Concurrency

Sang Shin
www.JPassion.com
"Learn with JPassion!"



Concurrency Utilities: JSR-166

- Enables development of simple yet powerful multithreaded applications
 - > Like Collection provides rich data structure handling capability
- Beat C performance in high-end server applications
- Provide richer set of concurrency building blocks
 - > wait(), notify() and synchronized are too primitive
- Enhance scalability, performance, readability and thread safety of Java applications

Why Use Concurrency Utilities?

- Reduced programming effort
- Increased performance
- Increased reliability
 - Eliminate threading hazards such as deadlock, starvation, race conditions, or excessive context switching are eliminated
- Improved maintainability
- Increased productivity

Concurrency Utilities

- Task Scheduling Framework
- Callable's and Future's
- Synchronizers
- Concurrent Collections
- Atomic Variables
- Locks
- Nanosecond-granularity timing

Task Scheduling Framework

Task Scheduling Framework

- Executor/ExercuteService/Executors framework supports
 - > standardizing submission
 - > scheduling
 - execution
 - control of asynchronous tasks according to a set of execution policies
- Executor is an interface
- ExecutorService extends Executor
- Executors is factory class for creating various kinds of ExercutorService implementations

Executor Interface

- Executor interface provides a way of de-coupling task submission from the execution
 - submission is standardized
 - execution: mechanics of how each task will be run, including details of thread use, scheduling – captured in the implementation class
- Example

```
Executor executor = getSomeKindofExecutor();
executor.execute(new RunnableTask1());
executor.execute(new RunnableTask2());
```

 Many Executor implementations impose some sort of limitation on how and when tasks are scheduled

Executor and ExecutorService ExecutorService adds lifecycle management

Executor and ExecutorService ExecutorService adds task submissionmethods

```
public interface Executor {
  void execute(Runnable command);
public interface ExecutorService extends Executor {
  void shutdown();
  List<Runnable> shutdownNow();
  boolean isShutdown();
  boolean isTerminated();
 boolean awaitTermination(long timeout,
                           TimeUnit unit);
  // other convenience methods for submitting tasks
  <T> Future<T> submit(Callable<T> task);
  Future<?> submit(Runnable task);
  <T> Future<T> submit(Runnable task, T result);
```

Executors: Factory for creating various types of ExecutorService

```
public class Executors {
  static ExecutorService
      newFixedThreadPool(int n);
  static ExecutorService
      newCachedThreadPool(int n);
  static ExecutorService
      newSingleThreadedExecutor();
  static ScheduledExecutorService
      newScheduledThreadPool(int n);
  static ScheduledExecutorService
      newSingleThreadScheduledExecutor();
  // additional versions specifying ThreadFactory
  // additional utility methods
```

newFixedThreadPool(int n)

```
public class Executors {
  // newFixedThreadPool() Creates a thread pool
  // that reuses a fixed number of threads operating
  // off a shared unbounded queue.
  static ExecutorService
      newFixedThreadPool(int n);
// Usage example - Create ExecutorService object
ExecutorService executorService =
    Executors.newFixedThreadPool(NUMBER THREADS);
// Submit a task
future = executorService.submit(new MyTask(i))
// Check if the task is done
String doneStatus = future.isDone()
```

Demo:

Executors_newFixedThreadPool_StandaloneExample 1108_javase5_concurrency.zip



pre-J2SE 5.0 Code Web Server—poor resource management

```
class WebServer {
 public static void main(String[] args)
    ServerSocket socket = new ServerSocket(80);
    while (true) {
      final Socket connection = socket.accept();
      Runnable r = new Runnable() {
        public void run() {
          handleRequest(connection);
      // Don't do this!
      new Thread(r).start();
```

Executors ExampleWeb Server—better resource management

```
class WebServer {
 Executor pool =
    Executors.newFixedThreadPool(7);
  public static void main(String[] args) {
    ServerSocket socket = new ServerSocket(80);
    while (true) {
      final Socket connection = socket.accept();
      Runnable r = new Runnable() {
        public void run()
          handleRequest(connection);
      pool.execute(r);
```

Demo:

Executors_newFixedThreadPool_Server,

Executors_Client

1108_javase5_concurrency.zip



newCachedThreadPool()

```
public class Executors {
  // newCachedThreadPool() creates a thread pool that creates
  // new threads as needed, but will reuse previously constructed
  // threads when they are available.
  static ExecutorService
      newCachedThreadPool();
// Usage example - Create ExecutorService object
ExecutorService executorService =
    Executors.newCachedThreadPool();
// Submit a task
future = executorService.submit(new MyTask(i))
// Check if the task is done
String doneStatus = future.isDone()
```

Demo:

Executors_newCachedThreadPool_StandaloneExample 1108_javase5_concurrency.zip



newCachedThreadPool()

```
public class Executors {
  // newScheduledThreadPool(POOL SIZE) creates a thread pool
  // that can schedule commands to run after a given delay, or
  // to execute periodically
  static ScheduledExecutorService
      newScheduledThreadPool(int corePoolSize);
// Usage example - Create ScheduledExecutorService object
ScheduledExecutorService scheduledExecutedService =
                Executors.newScheduledThreadPool(POOL SIZE);
// Schedule a task
ScheduledFuture<?> timeHandle1 =
      scheduledExecutedService.scheduleAtFixedRate(
           new TimePrinterTask1(System.out), // Task to execute
           1, // Initial delay
           3, // the period between successive executions
           SECONDS); // the time unit
```

Demo:

Executors_newScheduledThreadPool_StandaloneExample, Executors_newSingleThreadScheduledExecutor_StandaloneExample 1108_javase5_concurrency_zip



Callables and Futures

Callable's and Future's: Problem (pre-J2SE 5.0)

- If a new thread (callable thread) is started in an application, there is currently no way to return a result from that thread to the thread (calling thread) that started it without the use of a shared variable and appropriate synchronization
 - This is complex and makes code harder to understand and maintain

Callables and Futures

- Callable thread (Callee) implements Callable interface
 - Implement call() method rather than run()
- Calling thread (Caller) submits Callable object to Executor and then moves on
 - > Through submit() not execute()
 - > The submit() returns a Future object
- Calling thread (Caller) then retrieves the result using get() method of Future object
 - > If result is ready, it is returned
 - > If result is not ready, calling thread will block

Build CallableExample(This is Callee)

```
class CallableExample
      implements Callable<String> {
  public String call() {
        Do some work and create a result
                                           * /
    String result = "The work is ended";
    return result;
```

Future Example (Caller)

```
ExecutorService es =
  Executors.newSingleThreadExecutor();
Future<String> f =
  es.submit(new CallableExample());
/* Do some work in parallel */
try {
  String callableResult = f.get();
} catch (InterruptedException ie) {
  /* Handle */
} catch (ExecutionException ee) {
  /* Handle */
```



Semaphores

Semaphores

- Typically used to restrict access to fixed size pool of resources
- New Semaphore object is created with same count as number of resources
- Thread trying to access resource calls aquire()
 - > Returns immediately if semaphore count > 0
 - > Blocks if count is zero until release() is called by different thread
 - > aquire() and release() are thread safe atomic
 operations

Semaphore Example

```
private Semaphore semaphore;
private Resource[] resources;
private boolean[] used;
public Resource(int poolSize) {
  semaphore = new Semaphore(poolSize);
  /* Initialise resource pool */
public Resource getResource() {
  try { semaphore.aquire() } catch (IE) {}
  /* Acquire resource */
public void returnResource(Resource r) {
  /* Return resource to pool */
  semaphore.release();
```

Demo:

SemaphoreExample1,
SemaphoreExample2
1108_javase5_concurrency.zip



Concurrent Collections

BlockingQueue Interface

- Provides thread safe way for multiple threads to manipulate collection
- ArrayBlockingQueue is simplest concrete implementation
- Full set of methods

```
> put()
```

- > offer() [non-blocking]
- > peek()
- > take()
- > poll() [non-blocking and fixed time blocking]

Blocking Queue Example 1

```
private BlockingQueue<String> msgQueue;
public Logger(BlockingQueue<String> mq) {
 msgQueue = mq;
public void run() {
  try {
    while (true) {
      String message = msgQueue.take();
      /* Log message */
  } catch (InterruptedException ie) {
    /* Handle */
```

Blocking Queue Example 2

```
private ArrayBlockingQueue messageQueue =
  new ArrayBlockingQueue<String>(10);
Logger logger = new Logger(messageQueue);
public void run() {
  String someMessage;
  try {
    while (true) {
      /* Do some processing
         Blocks if no space available
      messageQueue.put(someMessage);
  } catch (InterruptedException ie) { }
```

Demo:

BlockingQueueExample
1108_javase5_concurrency.zip



Concurrency: Atomic Variables

Atomics

- java.util.concurrent.atomic
 - Small toolkit of classes that support lock-free threadsafe programming on single variables

```
AtomicInteger balance = new AtomicInteger(0);
public int deposit(integer amount) {
  return balance.addAndGet(amount);
}
```

Demo:

AtomicIntegerExample
1108_javase5_concurrency.zip



Concurrency: Locks

Locks

- Lock interface
 - More extensive locking operations than synchronized block
 - No automatic unlocking use try/finally to unlock
 - > Non-blocking access using tryLock()
- ReentrantLock
 - Concrete implementation of Lock
 - Holding thread can call lock () multiple times and not block
 - Useful for recursive code

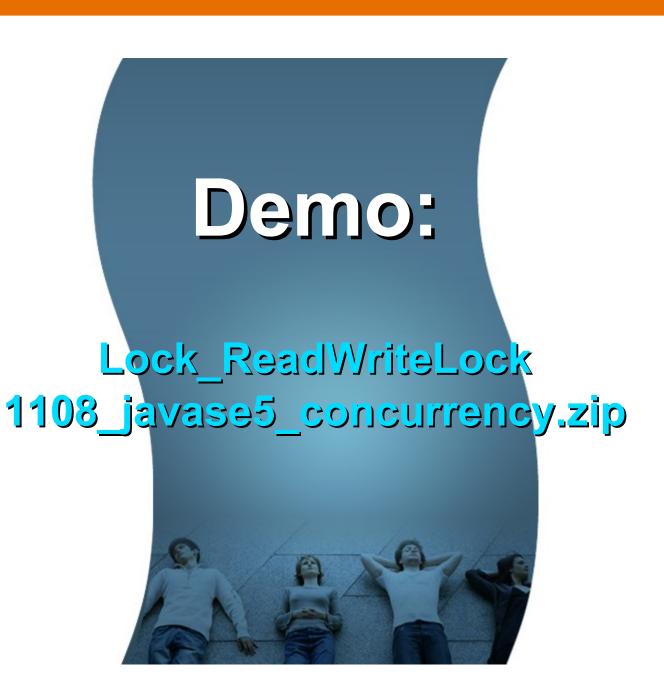
ReadWriteLock

- Has two locks controlling read and write access
 - Multiple threads can acquire the read lock if no threads have a write lock
 - If a thread has a read lock, others can acquire read lock but nobody can acquire write lock
 - If a thread has a write lock, nobody can have read/write lock
 - Methods to access locks

```
rwl.readLock().lock();
rwl.writeLock().lock();
```

ReadWrite Lock Example

```
class ReadWriteMap {
   final Map<String, Data> m = new TreeMap<String, Data>();
   final ReentrantReadWriteLock rwl =
                    new ReentrantReadWriteLock();
   final Lock r = rwl.readLock();
   final Lock w = rwl.writeLock();
   public Data get(String key) {
      r.lock();
      try { return m.get(key) }
       finally { r.unlock(); }
   public Data put(String key, Data value) {
      w.lock();
      try { return m.put(key, value); }
      finally { w.unlock(); }
   public void clear() {
      w.lock();
      try { m.clear(); }
      finally { w.unlock(); }
```



Thank you!

Sang Shin
http://www.JPassion.com
"Learn with Passion!"

