# The Java Concurrency API

**Cristian Mateos**[1,2]

[1]ISISTAN-UNICEN-CONICET

[2]Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET)

Computación Paralela y Distribuida

# Agenda

- The Java Concurrency API: Why
- Java Concurrency API: Basics
  - Executors: thread pools and scheduling
  - Futures
  - Synchronizers
  - Atomic variables and concurrent collections
- The ForkJoin framework

# Why concurrency utilities

- Recall that Java has built-in concurrency primitives - *wait()*, *notify()*, and *synchronized*

**... but they all have shortcomings**

- Hard to use correctly; easy to use incorrectly
- Too low level for many applications
- Can lead to poor performance if used incorrectly
- Leave out lots of useful concurrency constructs

# Why concurrency utilities

- Recall that Java has built-in concurrency primitives - *wait()*, *notify()*, and *synchronized*

**... but they all have shortcomings**

- Hard to use correctly; easy to use incorrectly
- Too low level for many applications
- Can lead to poor performance if used incorrectly
- Leave out lots of useful concurrency constructs

# Concurrency utilities: Goals

On one hand, provide efficient, correct and reusable concurrency building blocks. On the other hand, enhance scalability, performance, readability, maintainability, and thread-safety of concurrent Java applications.

# Building blocks

- Executor, ThreadPool, and Future
- Synchronizers: Semaphores and Barriers
- Atomic Variables
  - Low-level compare-and-set operations
- Concurrent collections
  - *BlockingQueue, SynchronousQueue, ConcurrentHashMap*
- Recommended reading: “Java Concurrency in Practice” by Brian Goetz et al.

# Executor

- Standardizes asynchronous invocation
- Separates job submission from execution policy
  - *anExecutor.execute(aRunnable)*
  - not *new Thread(aRunnable).start()*
- Two code styles supported:
  - Actions: *Runnables*
  - Functions: *Callables*
  - Also has lifecycle management: e.g., cancellation, shutdown
- Executor usually created via Executors factory class
  - Implicitly associated to a *ThreadPoolExecutor*
  - Via ExecutorService, customizes shutdown methods, before/after hooks, saturation policies, queuing

# Executor and ExecutorService

- ExecutorService adds lifecycle management to Executor

```
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
}
```

# Executors factory methods

```
public class Executors {
 static ExecutorService newSingleThreadedExecutor();
 static ExecutorService newFixedThreadPool(int n);
 // Beware demand peaks and long-running threads!
 static ExecutorService newCachedThreadPool(int n);
 static ScheduledExecutorService
                        newScheduledThreadPool(int n);
 // additional versions & utility methods
}
```

## Example without Executors: Web Server

- 1 Thread per message received (no limit on thread creation)

```
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);}
   };
   new Thread(r).start();
  }
 }
 public static void handleRequest(Socket socket){...}
}
```

# Example with Executors: Web Server

- Thread pool-based Web 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);
  }
 }
 public static void handleRequest(Socket socket){...}
}
```

## Callable and Future

- *Callable* is functionally analog of *Runnable*

```
interface Callable<V> {
  V call() throws Exception;
}
```

- *Future* represents result of asynchronous computation

```
interface Future<V> {
 V get() throws InterruptedException, ExecutionException;
 V get(long timeout, TimeUnit unit);
 boolean cancel(boolean mayInterruptIfRunning);
 boolean isCancelled();
 boolean isDone();
}
```

# Using Futures

- Client initiates asynchronous computation via one-way message
- Client receives a “handle” to the result: a *Future*
- Client does other work while waiting for result
- When ready, Client requests result from *Future*, blocking if necessary until result is available
- Client uses result
- **Example**: Searching a big file for a given string

# File searcher

```
// I am conciously ignoring exception handling...

public class FutureFileSearcher {
 ExecutorService executor = Executors.newFixedThreadPool(1);
 String searchFile(String keyword, String filePath) {
  Future<String> future = executor.submit(
    new Callable<String>() {
     public String call() {
      return doFileSearch(keyword, filePath);
     }
  });
  // Do some other useful stuff here
  // Check every 0.5 secs
  // (just for the sake of exemplification!)
  while (!future.isDone())
   Thread.sleep(500);
  return future.get();
 }
}
```

# FutureTask

- A cancellable asynchronous computation
- A base implementation of *Future*
- Can wrap a *Callable* or *Runnable*
  - Allows *FutureTask* to be submitted to an *Executor*

```
...
FutureTask<String> futureTask = new FutureTask<String>(
 new Callable<String>(){
  public String call() {
   return doFileSearch(keyword, filePath);
  }
});
executor.execute(futureTask);
...
```

# FutureTask and conventional threads

```java
FutureTask<Integer> task = new FutureTask<Integer>(()-> {
  System.out.println("Something␣complicated␣is␣computed");
  Thread.sleep(1000);
  return 42;
});
Thread t1 = new Thread(()->{
  try {
    int r = task.get();
    System.out.println("Result␣is␣" + r);
  } catch (InterruptedException | ExecutionException e) {}
});
System.out.println("t1␣wil␣wait␣until␣computation␣is␣ready");
t1.start();
task.run();
```

# Future and FutureTask example: Cache

```
public class Cache<K, V> {
 Map<K, Future<V>> map = new ConcurrentHashMap<K, Future<V>>();
 Executor executor = Executors.newFixedThreadPool(8);
 public V get (final K key) {
  Future<V> f = map.get(key); // null if key not found
  if (f == null) {
   Callable<V> c = new Callable<V>() {
    public V call() {// compute value associated with key}
   };
   f = new FutureTask<V>(c);
   // if key not found put(key,f) & return null
   Future old = map.putIfAbsent(key, f);
   if (old == null) { // otherwise return get(key)
    executor.execute(f);
   } else { f = old; }
  }
  return f.get();
 }
}
```

# ScheduledExecutorService

- For deferred and recurring tasks, can schedule
  - *Callable* or *Runnable* to run once with a fixed delay after submission
  - Schedule a *Runnable* to run periodically at a fixed rate
  - Schedule a *Runnable* to run periodically with a fixed delay between executions
- Submission returns a *ScheduledFutureTask* handle which can be used to cancel the task
- Like *Timer*, but supports more threads, pooling and finer thread control, so it is more versatile

# Synchronizers

- *Semaphore*: Dijkstra counting semaphore, managing some number of permits
- *CountDownLatch*: allows one or more threads to wait for a set of threads to complete an action
- *CyclicBarrier*: allows a set of threads to wait until they all reach a specified barrier point
- *Exchanger*: allows two threads to rendezvous and exchange data, such as exchanging an empty buffer for a full one

# Semaphores

- Maintain a logical set of permits
- *acquire()* blocks until a permit is free, then takes it
- *release()* adds a permit, releasing a blocking acquirer
- Often used to restrict the number of threads that can access some resource
  - But can be used to implement many synchronization kernels, patterns and primitives

# CountDownLatch

- Latching variables are conditions that once set never change
- Often used to start several threads, while having them wait for a signal before continuing

```
Main:
CountDownLatch startSignal = new CountDownLatch(1);
CountDownLatch stopSignal = new CountDownLatch(COUNT);
ExecutorService executor = Executors.newFixedThreadPool(COUNT);
for (int i = 0; i < COUNT; i++){
 executor.execute(new Worker(startSignal, stopSignal, i));
}
executor.shutdown();
// Do other stuff here...
startSignal.countDown();
try {
 stopSignal.await();
} catch (InterruptedException ex) {
 ex.printStackTrace();
}
```

# CountDownLatch (cont.)

```
Worker:
public void run() {
 try {
  // wait until the latch has counted down to zero
  startSignal.await();
 } catch (InterruptedException ex) {
  ex.printStackTrace();
 }
 System.out.println("Running:␣" + name);
 stopSignal.countDown();
}
```

## Cyclic barrier

- Allows threads to wait at a common barrier point
- Useful when a fixed-sized party of threads must occasionally wait for each other
- Can be re-used after threads released
- Can execute a *Runnable* once per barrier point
  - After the last thread arrives, but before any are released
  - Useful for updating shared-state before threads continue

# Cyclic barrier: Example

```
final CyclicBarrier barrier = new CyclicBarrier(3);
ExecutorService executor = Executors.newFixedThreadPool(3);
for (int i = 0; i < 3; i++) {
 executor.execute(new Runnable(){
  public void run(){
   try {
    log("At run()"); barrier.await();
    log("Do work");  barrier.await();
    log("Wait for end"); barrier.await();
    log("Done");
   }
   catch (Exception e) { e.printStackTrace(); }
  }
 });
}
executor.shutdown();
```

# Exchanger

- Synchronization point where two threads exchange objects
- A bidirectional *SynchronousQueue*
- Each thread presents some object on entry to the *exchange()* method, and receives the object presented by the other thread on return
- Example: Two threads filling and empyting two different buffers

# Exchanger (cont.)

- exchange(V x): It waits for another thread to arrive at exchange point and exchange object with that thread.
- exchange(V x, long timeout, TimeUnit unit): It waits for another thread for specific time interval provided in the method and exchange object with that thread.