# Thread Pool

Russell Winder

Version 1.0, 2015-10-01

# Table of Contents

# Concepts

## ThreadPool

On multi-core systems, you can benefit from having some tasks run asynchronously in the background, and so off-load your main thread of execution. The *ThreadPool* class allows you to easily start tasks in the background to be performed asynchronously and collect the results later.

## Usage

### Use of ThreadPool - the Java Executors' Based Concurrent Collection Processor

Closures Enhancements

*A Sample*

```
GParsExecutorsPool.withPool() {
    Closure longLastingCalculation = {calculate()}

    // Create a new closure, which starts the original closure on a thread pool.
    Closure fastCalculation = longLastingCalculation.async()

    // Returns almost immediately.
    Future result=fastCalculation()

    // Do stuff while calculation performs...
    println result.get()
}
```

*Another Sample*

```
GParsExecutorsPool.withPool() {
    /**
     * The callAsync() method is an asynchronous variant of the default call() method
     * to invoke a closure. It will return a Future for the result value.
     */
    assert 6 == {it * 2}.call(3).get()
    assert 6 == {it * 2}.callAsync(3).get()
}
```

### Executor Service Enhancements

*A Sample*

```
GParsExecutorsPool.withPool {ExecutorService executorService ->
    executorService << {println 'Inside parallel task'}
}
```

# Asynchronous Function Processing

*A Sample*

```
GParsExecutorsPool.withPool {

    // Waits for results.
    assert [10, 20] == AsyncInvokerUtil.doInParallel({calculateA()}, {calculateB()})

    // Returns a Future and doesn't wait for results to be calculated.
    assert [10, 20] == AsyncInvokerUtil.executeAsync({calculateA()}, {calculateB()})*.
get()
}
```