

# Jactor2 Revisited by Example

by Bill la Forge, 2014

Jactor2 is a robust and high-performance alternative to threads and locks. Jactor2 Revisited focuses on a subset of the API that is easy to learn but reasonably comprehensive.

## The HelloWorld Example

```
package org.agilewiki.jactor2.core.revisited;

import org.agilewiki.jactor2.core.blades.IsolationBladeBase;
import org.agilewiki.jactor2.core.impl.Plant;
import org.agilewiki.jactor2.core.requests.AsyncResponseProcessor;
import org.agilewiki.jactor2.core.requests.impl.AsyncRequestImpl;

public class HelloWorld extends IsolationBladeBase {

    public static void main(final String[] args) throws Exception {
        new Plant();
        new HelloWorld();
        System.out.println("initialized");
    }

    private HelloWorld() throws Exception {
        new ASig("run") {
            @Override
            protected void processAsyncOperation(final AsyncRequestImpl _asyncRequestImpl,
                final AsyncResponseProcessor<Void> _asyncResponseProcessor)
                throws Exception {
                System.out.println("Hello world!");
                Plant.close();
                System.out.println("finished");
            }
        }.signal();
    }
}
```

### Output:

```
initialized
Hello world!
finished
```

The *HelloWorld* class is a *Blade*. It has a *Reactor* that is created when the default constructor of *IsolationBladeBase* is called.

```
public static void main(final String[] args) throws Exception {
    new Plant();
    new HelloWorld();
    System.out.println("initialized");
}
```

The *main* method does three things:

1. An instance of *Plant* is created. This provides the operating environment and configuration for the reactors.
2. An instance of *HelloWorld* is created. And
3. The line *initialized* is printed, as this completes the program initialization.

```
public HelloWorld() throws Exception {
    new ASig("run") {
        @Override
        protected void processAsyncOperation(final AsyncRequestImpl _asyncRequestImpl,
            final AsyncResponseProcessor<Void> _asyncResponseProcessor)
            throws Exception {
            System.out.println("Hello world!");
            Plant.close();
            System.out.println("finished");
        }
    }
}
```

```

        }.signal();
    }
}

```

The constructor creates a *run* signal which is passed to the *HelloWorld Blade* via its *Reactor*. On receipt of this signal, the *Blade* prints the line *Hello world!*, closes the operating environment and then prints the line *finished*.

#### Notes:

1. The *ASig.signal* method can be called from any thread and within any context. In this case the method was called from the main thread.
2. *ASig* (Asynchronous Signal) is a nested class, defined in one of the super classes of *HelloWorld*. This is how the *signal* method accesses the *Reactor* of *HelloWorld*.

## The Worker Blade

```

package org.agilewiki.jactor2.core.revisited;

import org.agilewiki.jactor2.core.blades.IsolationBladeBase;
import org.agilewiki.jactor2.core.requests.AsyncResponseProcessor;
import org.agilewiki.jactor2.core.requests.impl.AsyncRequestImpl;

public class Worker extends IsolationBladeBase {
    public final String id;
    private int count;

    public Worker(final int _id) throws Exception {
        id = "Worker" + _id;
    }

    public int getCount() {
        return count;
    }

    public AReq<Void> run(final long _iterations, final int _timeoutMillis) {
        return new AReq<Void>("run" + id) {

            @Override
            protected void processAsyncOperation(final AsyncRequestImpl _asyncRequestImpl,
                final AsyncResponseProcessor<Void> _asyncResponseProcessor)
                throws Exception {
                _asyncRequestImpl.setMessageTimeoutMillis(_timeoutMillis);
                System.out.println(id + ": started " + ++count);
                for (long i = 0L; i < _iterations; i++);
                System.out.println(id + ": finished " + count);
                _asyncResponseProcessor.processAsyncResponse(null);
            }
        };
    }
}

```

The *Worker* blade is useful for simulating a CPU load and we will use it in a number of examples. It has one operation, *run*, which returns an Asynchronous Request, *AReq*, that can be used to pass the *run* request to *Worker*.

Like *ASig*, *AReq* is defined as a nested *class* in a super class of *Worker*, which again is how it can access the *Reactor* of *Worker*. But unlike *ASig*, *AReq* can not be used to send a signal. (This is the only difference—*AReq* is the super class of *ASig*.)

Note that *count*, which is the number of times a run request has been received, is *private* and is only updated when processing a request, *run*. This means that there will be no race conditions for *count*, as requests are processed strictly one at a time.

We have not yet covered the *AsyncRequestImpl.setMessageTimeoutMillis* method, but we will do that when covering some examples that use *Worker*.

## The Simple Example

```
package org.agilewiki.jactor2.core.revisited;

import org.agilewiki.jactor2.core.blades.IsolationBladeBase;
import org.agilewiki.jactor2.core.impl.Plant;
import org.agilewiki.jactor2.core.requests.AsyncResponseProcessor;
import org.agilewiki.jactor2.core.requests.impl.AsyncRequestImpl;

public class Simple extends IsolationBladeBase {

    public static void main(final String[] args) throws Exception {
        new Plant();
        new Simple();
        System.out.println("initialized");
    }

    private Simple() throws Exception {
        new ASig("run") {
            @Override
            protected void processAsyncOperation(final AsyncRequestImpl _asyncRequestImpl,
                final AsyncResponseProcessor<Void> _asyncResponseProcessor)
                throws Exception {
                AsyncResponseProcessor<Void> runResponseProcessor =
                    new AsyncResponseProcessor<Void>() {
                        @Override
                        public void processAsyncResponse(Void _response) throws Exception {
                            Plant.close();
                            System.out.println("finished");
                        }
                    };
                _asyncRequestImpl.send(new Worker(0).run(100000000L, -1), runResponseProcessor);
            }
        }.signal();
    }
}
```

### Output:

```
initialized
Worker0: started 1
Worker0: finished 1
finished
```

The *Simple Blade* sends a *run* request to a *Worker* and then processes the response message. But note that it is while processing the *run* signal sent to *Simple* that the *AsyncRequestImpl.send* method is called. The *send* method can not be called except while processing a message. The *send* method takes two arguments: the request to be sent and an *AsyncResponseProcessor* object used to process the response message.

## The Timeout Example

```
package org.agilewiki.jactor2.core.revisited;

import org.agilewiki.jactor2.core.blades.IsolationBladeBase;
import org.agilewiki.jactor2.core.impl.Plant;
import org.agilewiki.jactor2.core.requests.AsyncResponseProcessor;
import org.agilewiki.jactor2.core.requests.impl.AsyncRequestImpl;

public class Timeout extends IsolationBladeBase {

    public static void main(final String[] args) throws Exception {
        new Plant();
        new Timeout();
        System.out.println("initialized");
    }

    private Timeout() throws Exception {
        new ASig("run") {
            @Override
            protected void processAsyncOperation(final AsyncRequestImpl _asyncRequestImpl,
                final AsyncResponseProcessor<Void> _asyncResponseProcessor)
                throws Exception {
            }
        }.signal();
    }
}
```

### Output:

In the *Timeout* example, the number of iterations that *Worker* is told to perform has been increased to 10 billion. The result is that the message times out and the *Worker Reactor* is closed because the thread is hung. This unanticipated exception causes the *Timeout Reactor* to *close* as well and the program exits.

## The VerySlow Example

```
package org.agilewiki.jactor2.core.revisited;

import org.agilewiki.jactor2.core.blades.IsolationBladeBase;
import org.agilewiki.jactor2.core.impl.Plant;
import org.agilewiki.jactor2.core.requests.AsyncResponseProcessor;
import org.agilewiki.jactor2.core.requests.impl.AsyncRequestImpl;

public class VerySlow extends IsolationBladeBase {

    public static void main(final String[] args) throws Exception {
        new Plant();
        new VerySlow();
        System.out.println("initialized");
    }

    private VerySlow() throws Exception {
        new ASig("run") {
            @Override
            protected void processAsyncOperation(final AsyncRequestImpl _asyncRequestImpl,
                final AsyncResponseProcessor<Void> _asyncResponseProcessor)
                throws Exception {
                AsyncResponseProcessor<Void> runResponseProcessor =
                    new AsyncResponseProcessor<Void>() {
                        @Override
                        public void processAsyncResponse(Void _response) throws Exception {
                            Plant.close();
                            System.out.println("finished");
                        }
                    }
            }
        }
    }
}
```

```

        }
    };
    _asyncRequestImpl.send(new Worker(0).run(1000000000L, 10000),
        runResponseProcessor);
    }
    }.signal();
}
}
}

```

## Output:

```

initialized
Worker0: started 1
Worker0: finished 1
finished

```

Remember the *AsyncRequestImpl.setMessageTimeoutMillis* method used in the *Worker run* request? Until now it has been passed a value of -1, which indicates that the default timeout should be used. In the *VerySlow* example, a timeout value of 10,000 is used. (10 seconds.) A large timeout value should always be used for messages might take some time to process, to avoid closing reactors needlessly when the system becomes loaded.

## The Parallel Example

```

package org.agilewiki.jactor2.core.revisited;

import org.agilewiki.jactor2.core.blades.IsolationBladeBase;
import org.agilewiki.jactor2.core.impl.Plant;
import org.agilewiki.jactor2.core.requests.AsyncResponseProcessor;
import org.agilewiki.jactor2.core.requests.impl.AsyncRequestImpl;

public class Parallel extends IsolationBladeBase {

    public static void main(final String[] args) throws Exception {
        new Plant();
        new Parallel(5);
        System.out.println("initialized");
    }

    private Parallel(final int _p) throws Exception {
        new ASig("run") {
            @Override
            protected void processAsyncOperation(final AsyncRequestImpl _asyncRequestImpl,
                final AsyncResponseProcessor<Void> _asyncResponseProcessor)
                throws Exception {
                AsyncResponseProcessor<Void> runResponseProcessor =
                    new AsyncResponseProcessor<Void>() {
                        @Override
                        public void processAsyncResponse(Void _response) throws Exception {
                            if (_asyncRequestImpl.hasNoPendingResponses()) {
                                Plant.close();
                                System.out.println("finished");
                            }
                        }
                    };
            };
            for (int i = 0; i < _p; i++)
                _asyncRequestImpl.send(new Worker(i).run(1000000000L, -1),
                    runResponseProcessor);
        }
        }.signal();
    }
}

```

## Output:

```

initialized
Worker4: started 1
Worker1: started 1
Worker0: started 1
Worker3: started 1
Worker2: started 1
Worker0: finished 1

```

```
Worker4: finished 1
Worker1: finished 1
Worker3: finished 1
Worker2: finished 1
finished
```

The *Parallel Blade* sends a *run Request* to each of 5 *Worker Blades*. On receiving each response, the *AsynchronousRequestImpl.hasNoPendingResponses* method is called to see if the last response has been received. If so, the *Plant* is closed and *finished* is printed.

## The Sequence Example

```
package org.agilewiki.jactor2.core.revisited;

import org.agilewiki.jactor2.core.blades.IsolationBladeBase;
import org.agilewiki.jactor2.core.impl.Plant;
import org.agilewiki.jactor2.core.requests.AsyncResponseProcessor;
import org.agilewiki.jactor2.core.requests.impl.AsyncRequestImpl;

public class Sequence extends IsolationBladeBase {

    private Worker worker;
    private AsyncResponseProcessor<Void> runResponseProcessor;

    public static void main(final String[] args) throws Exception {
        new Plant();
        new Sequence(5);
        System.out.println("initialized");
    }

    private Sequence(final int maxCount) throws Exception {
        new ASig("run") {

            @Override
            protected void processAsyncOperation(final AsyncRequestImpl _asyncRequestImpl,
                final AsyncResponseProcessor<Void> _asyncResponseProcessor)
                throws Exception {
                worker = new Worker(0);
                runResponseProcessor = new AsyncResponseProcessor<Void>() {
                    @Override
                    public void processAsyncResponse(Void _response) throws Exception {
                        if (worker.getCount() < maxCount) {
                            _asyncRequestImpl.send(worker.run(100000000L, -1),
                                runResponseProcessor);
                        } else {
                            Plant.close();
                            System.out.println("finished");
                        }
                    }
                };
                _asyncRequestImpl.send(worker.run(100000000L, -1), runResponseProcessor);
            }
        }.signal();
    }
}
```

### Output:

```
initialized
Worker0: started 1
Worker0: finished 1
Worker0: started 2
Worker0: finished 2
Worker0: started 3
Worker0: finished 3
Worker0: started 4
Worker0: finished 4
Worker0: started 5
```

Worker0: finished 5  
finished

The *Sequence Blade* sends a series of *run* requests to a *Worker*, sending each request only after receiving the response from the previous request. Everything then is processed in order.