

# 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, as well as creating a pool of non-daemon threads.
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++) {
                    //Do something
                }
                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(1000000000L, -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 {
    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(10000000000L, -1), runResponseProcessor);
}
}
}
}
}

```

## Output:

```

initialized
Worker0: started 1
[pool-1-thread-2] ERROR org.agilewiki.jactor2.core.reactors.Reactor - message timeout -> reactor close
[Thread-1] WARN org.agilewiki.jactor2.core.reactors.Reactor - Uncaught throwable
[pool-1-thread-2] ERROR org.agilewiki.jactor2.core.reactors.Reactor - hung thread
message=runWorker0, isComplete=true, isOneWay=false, source=org.agilewiki.jactor2.core.impl.mtReactors.IsolationReactorMtImpl@1,
target=org.agilewiki.jactor2.core.impl.mtReactors.IsolationReactorMtImpl@2, this=class
org.agilewiki.jactor2.core.impl.mtRequests.AsyncRequestMtImpl#41de9139
message=run, isComplete=true, isOneWay=true, source=null, target=org.agilewiki.jactor2.core.impl.mtReactors.IsolationReactorMtImpl@1,
this=class org.agilewiki.jactor2.core.impl.mtRequests.AsyncRequestMtImpl#6bb97f10
[pool-1-thread-2] ERROR org.agilewiki.jactor2.core.reactors.Reactor - hung thread -> plant exit
org.agilewiki.jactor2.core.reactors.ReactorClosedException
    at org.agilewiki.jactor2.core.impl.mtRequests.RequestMtImpl.close(RequestMtImpl.java:366)
    at org.agilewiki.jactor2.core.impl.mtRequests.AsyncRequestMtImpl.close(AsyncRequestMtImpl.java:331)
    at org.agilewiki.jactor2.core.impl.mtReactors.ReactorMtImpl.fail(ReactorMtImpl.java:298)
    at org.agilewiki.jactor2.core.impl.mtPlant.Recovery.onMessageTimeout(Recovery.java:46)
    at org.agilewiki.jactor2.core.impl.mtReactors.ReactorMtImpl.reactorPoll(ReactorMtImpl.java:624)
    at org.agilewiki.jactor2.core.impl.mtReactors.ReactorMtImpl.reactorPoll(ReactorMtImpl.java:634)
    at org.agilewiki.jactor2.core.impl.mtPlant.PlantMtImpl$1.run(PlantMtImpl.java:283)
    at java.util.concurrent.Executors$RunnableAdapter.call(Executors.java:511)
    at java.util.concurrent.FutureTask.runAndReset(FutureTask.java:308)
    at java.util.concurrent.ScheduledThreadPoolExecutor$ScheduledFutureTask.access$301(ScheduledThreadPoolExecutor.java:180)
    at java.util.concurrent.ScheduledThreadPoolExecutor$ScheduledFutureTask.run(ScheduledThreadPoolExecutor.java:294)
    at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1142)
    at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:617)
    at java.lang.Thread.run(Thread.java:745)

```

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.

Most messages do not take long to process, so the default timeout is only a few seconds.

## 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(100000000L, -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.

## The Isolation 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 Isolation extends IsolationBladeBase {

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

    private Isolation(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");
                            }
                        }
                    };
                Single single = new Single();
                for (int i = 0; i < _p; i++)
                    _asyncRequestImpl.send(single.run(i, 100000000L, -1),
                        runResponseProcessor);
            }
        }.signal();
    }
}

class Single extends IsolationBladeBase {

    Single() throws Exception {
    }

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

            @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 {
                            System.out.println("runIso finish " + i);
                            _asyncResponseProcessor.processAsyncResponse(null);
                        }
                    };

                System.out.println("runIso start " + i);
                _asyncRequestImpl.send(new Worker(i).run(100000000L, -1), runResponseProcessor);
            }
        };
    }
}
```

```
}  
}
```

## Output:

```
initialized  
runIso start 0  
Worker0: started 1  
Worker0: finished 1  
runIso finish 0  
runIso start 1  
Worker1: started 1  
Worker1: finished 1  
runIso finish 1  
runIso start 2  
Worker2: started 1  
Worker2: finished 1  
runIso finish 2  
runIso start 3  
Worker3: started 1  
Worker3: finished 1  
runIso finish 3  
runIso start 4  
Worker4: started 1  
Worker4: finished 1  
runIso finish 4  
finished
```

The Iso Blade blocks all but one request until that request is complete.

## The Order 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 Order extends IsolationBladeBase {  
    Other otherX;  
    Other otherY;  
  
    public static void main(final String[] args) throws Exception {  
        new Plant();  
        new Order();  
        System.out.println("initialized");  
    }  
  
    private Order() throws Exception {  
        new ASig("run") {  
            @Override  
            protected void processAsyncOperation(final AsyncRequestImpl _asyncRequestImpl,  
                final AsyncResponseProcessor<Void> _asyncResponseProcessor)  
                throws Exception {  
                otherX = new Other();  
                otherY = new Other();  
                final AsyncResponseProcessor<Void> runResponseProcessor =  
                    new AsyncResponseProcessor<Void>() {  
                        @Override  
                        public void processAsyncResponse(Void _response) throws Exception {  
                            Plant.close();  
                            System.out.println("finished");  
                        }  
                    };  
                final AsyncResponseProcessor<Void> runResponseProcessor3 =  
                    new AsyncResponseProcessor<Void>() {  
                        @Override  
                        public void processAsyncResponse(Void _response) throws Exception {  
                            _asyncRequestImpl.send(otherY.run(otherX, "Y -> X"), runResponseProcessor);  
                        }  
                    };  
                final AsyncResponseProcessor<Void> runResponseProcessor2 =  
                    new AsyncResponseProcessor<Void>() {  
                        @Override  
                        public void processAsyncResponse(Void _response) throws Exception {
```



```

        _asyncRequestImpl.send(otherY.run(otherY, "Y -> Y"), runResponseProcessor3);
    }
};
final AsyncResponseProcessor<Void> runResponseProcessor1 =
    new AsyncResponseProcessor<Void>() {
        @Override
        public void processAsyncResponse(Void _response) throws Exception {
            _asyncRequestImpl.send(otherX.run(otherY, "X -> Y"), runResponseProcessor2);
        }
    };
_asyncRequestImpl.setExceptionHandler(new ExceptionHandler() {
    @Override
    public void processException(Exception e,
        AsyncResponseProcessor _asyncResponseProcessor)
        throws Exception {
        Plant.close();
        System.err.println("caught exception:");
        e.printStackTrace();
    }
});
_asyncRequestImpl.send(otherX.run(otherX, "X -> X"), runResponseProcessor1);
}
}.signal();
}
}

class Other extends IsolationBladeBase {
    Other() throws Exception {}

    AReq<Void> run(final Other _other, final String _i) {
        return new AReq<Void>("RunOther") {
            @Override
            protected void processAsyncOperation(AsyncRequestImpl _asyncRequestImpl,
                AsyncResponseProcessor _asyncResponseProcessor)
                throws Exception {
                _asyncRequestImpl.send(_other.blip(_i), _asyncResponseProcessor);
            }
        };
    }

    AReq<Void> blip(final String _i) {
        return new AReq<Void>("blip") {
            @Override
            protected void processAsyncOperation(AsyncRequestImpl _asyncRequestImpl,
                AsyncResponseProcessor _asyncResponseProcessor)
                throws Exception {
                System.err.println("blip " + _i);
                _asyncResponseProcessor.processAsyncResponse(null);
            }
        };
    }
}

```

## Output:

```

initialized
blip X -> X
blip X -> Y
blip Y -> Y
[Thread-0] ERROR org.agilewiki.jactor2.core.reactors.Reactor - runtime exception -> reactor close
[Thread-1] WARN org.agilewiki.jactor2.core.reactors.Reactor - Uncaught throwable
org.agilewiki.jactor2.core.reactors.ReactorClosedException: java.lang.IllegalStateException: not processing request:
message=blip, isComplete=false, isOneWay=false, source=org.agilewiki.jactor2.core.impl.mtReactors.IsolationReactorMtImpl@3,
target=org.agilewiki.jactor2.core.impl.mtReactors.IsolationReactorMtImpl@2, this=class
org.agilewiki.jactor2.core.impl.mtRequests.AsyncRequestMtImpl#439161ab
message=runOther, isComplete=true, isOneWay=false, source=org.agilewiki.jactor2.core.impl.mtReactors.IsolationReactorMtImpl@1,
target=org.agilewiki.jactor2.core.impl.mtReactors.IsolationReactorMtImpl@3, this=class
org.agilewiki.jactor2.core.impl.mtRequests.AsyncRequestMtImpl#4d5687a9
message=run, isComplete=false, isOneWay=true, source=null,
target=org.agilewiki.jactor2.core.impl.mtReactors.IsolationReactorMtImpl@1, this=class
org.agilewiki.jactor2.core.impl.mtRequests.AsyncRequestMtImpl#49690adb
    at org.agilewiki.jactor2.core.impl.mtRequests.RequestMtImpl.eval(RequestMtImpl.java:402)
    at org.agilewiki.jactor2.core.impl.mtReactors.ReactorMtImpl.processMessage(ReactorMtImpl.java:482)
    at org.agilewiki.jactor2.core.impl.mtReactors.IsolationReactorMtImpl.processMessage(IsolationReactorMtImpl.java:45)
    at org.agilewiki.jactor2.core.impl.mtReactors.ReactorMtImpl.run(ReactorMtImpl.java:571)
    at org.agilewiki.jactor2.core.impl.mtPlant.ReactorThreadPoolThreadManager$1.run(ReactorThreadPoolThreadManager.java:78)
    at java.lang.Thread.run(Thread.java:745)
Caused by: java.lang.IllegalStateException: not processing request:
message=blip, isComplete=false, isOneWay=false, source=org.agilewiki.jactor2.core.impl.mtReactors.IsolationReactorMtImpl@3,
target=org.agilewiki.jactor2.core.impl.mtReactors.IsolationReactorMtImpl@2, this=class
org.agilewiki.jactor2.core.impl.mtRequests.AsyncRequestMtImpl#439161ab
message=runOther, isComplete=true, isOneWay=false, source=org.agilewiki.jactor2.core.impl.mtReactors.IsolationReactorMtImpl@1,
target=org.agilewiki.jactor2.core.impl.mtReactors.IsolationReactorMtImpl@3, this=class
org.agilewiki.jactor2.core.impl.mtRequests.AsyncRequestMtImpl#4d5687a9
message=run, isComplete=false, isOneWay=true, source=null,
target=org.agilewiki.jactor2.core.impl.mtReactors.IsolationReactorMtImpl@1, this=class
org.agilewiki.jactor2.core.impl.mtRequests.AsyncRequestMtImpl#49690adb

```

```

at org.agilewiki.jactor2.core.impl.mtReactors.IsolationInbox.requestEnd(IsolationInbox.java:138)
at org.agilewiki.jactor2.core.impl.mtReactors.ReactorMtImpl.requestEnd(ReactorMtImpl.java:528)
at org.agilewiki.jactor2.core.impl.mtRequests.RequestMtImpl.setResponse(RequestMtImpl.java:292)
at org.agilewiki.jactor2.core.impl.mtRequests.AsyncRequestMtImpl.setResponse(AsyncRequestMtImpl.java:381)
at org.agilewiki.jactor2.core.impl.mtRequests.RequestMtImpl.processObjectResponse(RequestMtImpl.java:321)
at org.agilewiki.jactor2.core.impl.mtRequests.AsyncRequestMtImpl.processAsyncResponse(AsyncRequestMtImpl.java:203)
at org.agilewiki.jactor2.core.requests.BoundResponseProcessor$1.processSyncOperation(BoundResponseProcessor.java:49)
at org.agilewiki.jactor2.core.requests.BoundResponseProcessor$1.processSyncOperation(BoundResponseProcessor.java:46)
at org.agilewiki.jactor2.core.requests.SOp.doSync(SOp.java:44)
at org.agilewiki.jactor2.core.impl.mtRequests.SyncRequestMtImpl.processRequestMessage(SyncRequestMtImpl.java:48)
at org.agilewiki.jactor2.core.impl.mtRequests.RequestMtImpl.eval(RequestMtImpl.java:396)
... 5 more
java.lang.IllegalStateException: circular resources
at org.agilewiki.jactor2.core.impl.mtReactors.IsolationReactorMtImpl.addResource(IsolationReactorMtImpl.java:61)
at org.agilewiki.jactor2.core.reactors.ReactorBase.addResource(ReactorBase.java:129)
at org.agilewiki.jactor2.core.impl.mtRequests.RequestMtImpl.doSend(RequestMtImpl.java:252)
at org.agilewiki.jactor2.core.impl.mtRequests.AsyncRequestMtImpl.send(AsyncRequestMtImpl.java:264)
at org.agilewiki.jactor2.core.impl.mtRequests.AsyncRequestMtImpl.send(AsyncRequestMtImpl.java:407)
at org.agilewiki.jactor2.core.revisited.Other$1.processAsyncOperation(Other.java:75)
at org.agilewiki.jactor2.core.requests.AOp.doAsync(AOp.java:45)
at org.agilewiki.jactor2.core.impl.mtRequests.AsyncRequestMtImpl.processRequestMessage(AsyncRequestMtImpl.java:232)
at org.agilewiki.jactor2.core.impl.mtRequests.RequestMtImpl.eval(RequestMtImpl.java:396)
at org.agilewiki.jactor2.core.impl.mtReactors.ReactorMtImpl.processMessage(ReactorMtImpl.java:482)
at org.agilewiki.jactor2.core.impl.mtReactors.IsolationReactorMtImpl.processMessage(IsolationReactorMtImpl.java:45)
at org.agilewiki.jactor2.core.impl.mtReactors.ReactorMtImpl.run(ReactorMtImpl.java:571)
at org.agilewiki.jactor2.core.impl.mtPlant.ReactorPoolThreadManager$1.run(ReactorPoolThreadManager.java:78)
at java.lang.Thread.run(Thread.java:745)
[Thread-1] ERROR org.agilewiki.jactor2.core.reactors.Reactor - runtime exception -> reactor close
java.lang.IllegalStateException: not processing request:
message=blip, isComplete=false, isOneWay=false, source=org.agilewiki.jactor2.core.impl.mtReactors.IsolationReactorMtImpl@3,
target=org.agilewiki.jactor2.core.impl.mtReactors.IsolationReactorMtImpl@2, this=class
org.agilewiki.jactor2.core.impl.mtRequests.AsyncRequestMtImpl#439161ab
message=runOther, isComplete=true, isOneWay=false, source=org.agilewiki.jactor2.core.impl.mtReactors.IsolationReactorMtImpl@1,
target=org.agilewiki.jactor2.core.impl.mtReactors.IsolationReactorMtImpl@3, this=class
org.agilewiki.jactor2.core.impl.mtRequests.AsyncRequestMtImpl#4d5687a9
message=run, isComplete=false, isOneWay=true, source=null,
target=org.agilewiki.jactor2.core.impl.mtReactors.IsolationReactorMtImpl@1, this=class
org.agilewiki.jactor2.core.impl.mtRequests.AsyncRequestMtImpl#49690adb
at org.agilewiki.jactor2.core.impl.mtReactors.IsolationInbox.requestEnd(IsolationInbox.java:138)
at org.agilewiki.jactor2.core.impl.mtReactors.ReactorMtImpl.requestEnd(ReactorMtImpl.java:528)
at org.agilewiki.jactor2.core.impl.mtRequests.RequestMtImpl.setResponse(RequestMtImpl.java:292)
at org.agilewiki.jactor2.core.impl.mtRequests.AsyncRequestMtImpl.setResponse(AsyncRequestMtImpl.java:381)
at org.agilewiki.jactor2.core.impl.mtRequests.RequestMtImpl.processObjectResponse(RequestMtImpl.java:321)
at org.agilewiki.jactor2.core.impl.mtRequests.AsyncRequestMtImpl.processAsyncResponse(AsyncRequestMtImpl.java:203)
at org.agilewiki.jactor2.core.requests.BoundResponseProcessor$1.processSyncOperation(BoundResponseProcessor.java:49)
at org.agilewiki.jactor2.core.requests.BoundResponseProcessor$1.processSyncOperation(BoundResponseProcessor.java:46)
at org.agilewiki.jactor2.core.requests.SOp.doSync(SOp.java:44)
at org.agilewiki.jactor2.core.impl.mtRequests.SyncRequestMtImpl.processRequestMessage(SyncRequestMtImpl.java:48)
at org.agilewiki.jactor2.core.impl.mtRequests.RequestMtImpl.eval(RequestMtImpl.java:396)
at org.agilewiki.jactor2.core.impl.mtReactors.ReactorMtImpl.processMessage(ReactorMtImpl.java:482)
at org.agilewiki.jactor2.core.impl.mtReactors.IsolationReactorMtImpl.processMessage(IsolationReactorMtImpl.java:45)
at org.agilewiki.jactor2.core.impl.mtReactors.ReactorMtImpl.run(ReactorMtImpl.java:571)
at org.agilewiki.jactor2.core.impl.mtPlant.ReactorPoolThreadManager$1.run(ReactorPoolThreadManager.java:78)
at java.lang.Thread.run(Thread.java:745)
caught exception:
org.agilewiki.jactor2.core.reactors.ReactorClosedException: java.lang.IllegalStateException: circular resources
at org.agilewiki.jactor2.core.impl.mtRequests.RequestMtImpl.eval(RequestMtImpl.java:402)
at org.agilewiki.jactor2.core.impl.mtReactors.ReactorMtImpl.processMessage(ReactorMtImpl.java:482)
at org.agilewiki.jactor2.core.impl.mtReactors.IsolationReactorMtImpl.processMessage(IsolationReactorMtImpl.java:45)
at org.agilewiki.jactor2.core.impl.mtReactors.ReactorMtImpl.run(ReactorMtImpl.java:571)
at org.agilewiki.jactor2.core.impl.mtPlant.ReactorPoolThreadManager$1.run(ReactorPoolThreadManager.java:78)
at java.lang.Thread.run(Thread.java:745)
Caused by: java.lang.IllegalStateException: circular resources
at org.agilewiki.jactor2.core.impl.mtReactors.IsolationReactorMtImpl.addResource(IsolationReactorMtImpl.java:61)
at org.agilewiki.jactor2.core.reactors.ReactorBase.addResource(ReactorBase.java:129)
at org.agilewiki.jactor2.core.impl.mtRequests.RequestMtImpl.doSend(RequestMtImpl.java:252)
at org.agilewiki.jactor2.core.impl.mtRequests.AsyncRequestMtImpl.send(AsyncRequestMtImpl.java:264)
at org.agilewiki.jactor2.core.impl.mtRequests.AsyncRequestMtImpl.send(AsyncRequestMtImpl.java:407)
at org.agilewiki.jactor2.core.revisited.Other$1.processAsyncOperation(Other.java:75)
at org.agilewiki.jactor2.core.requests.AOp.doAsync(AOp.java:45)
at org.agilewiki.jactor2.core.impl.mtRequests.AsyncRequestMtImpl.processRequestMessage(AsyncRequestMtImpl.java:232)
at org.agilewiki.jactor2.core.impl.mtRequests.RequestMtImpl.eval(RequestMtImpl.java:396)
... 5 more

```

A *Blade* will block requests if it has already started processing another request. So deadlocks can occur unless requests are always passed between *Blades* in the same direction. But the mere possibility of deadlocks needs to be prevented, as deadlocks occur intermittently.

JActor2 tracks the *Blades* which have been sent a request by each *Blade*. It also verifies that requests are always sent in the same direction. So for example, if *Blade X* has sent a request to *Blade Y* and *Blade Y* has sent a request to *Blade Z*, then an attempt by *Blade Z* to send a request to *Blades X* or *Y* raises an *Exception*.

The advantage here is that production errors can be avoided through system testing with reasonable coverage—which is not effective when deadlocks are possible.

Note that in the example given, the first 3 cases succeeded. The fourth case,  $Y \rightarrow X$ , failed only because  $X \rightarrow Y$  had already established the direction.

Also, note the use of an *ExceptionHandler* in the constructor of *Order*. This was used to ensure that

the *Plant* was properly closed. More on that later.

## The Signal 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 Signal extends IsolationBladeBase {

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

    private Signal() 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 Ping(Signal.this).ping(), runResponseProcessor);
            }
        }.signal();
    }

    void blip() {
        new ASig("blip") {
            @Override
            protected void processAsyncOperation(AsyncRequestImpl _asyncRequestImpl,
                AsyncResponseProcessor<Void> _asyncResponseProcessor)
                throws Exception {
                System.out.println("blip");
                _asyncResponseProcessor.processAsyncResponse(null);
            }
        }.signal();
    }
}

class Ping extends IsolationBladeBase {
    private final Signal signal;
    Ping(final Signal _signal) throws Exception {
        signal = _signal;
    }

    AReq<Void> ping() {
        return new AReq<Void>("runPing") {
            @Override
            protected void processAsyncOperation(AsyncRequestImpl _asyncRequestImpl,
                AsyncResponseProcessor<Void> _asyncResponseProcessor)
                throws Exception {
                signal.blip();
                _asyncResponseProcessor.processAsyncResponse(null);
            }
        };
    }
}
```

### Output:

```
initialized
blip
finished
```

One of the advantages of signals is that, unlike requests, they are never blocked. So there are no constraints on the direction a signal message can be passed. Here we see the *Signal Blade* passing a request message to the *Ping Blade*, which in turn passes a signal back to the *Signal Blade*.