AsyncObjects Framework

Developer's Guide

Version: 0.3.2 (110)

by Constantine Plotnikov

Revision History

Date	Version	Changes	Authors
2007-04-28	0.1.0	Initial draft	Constantine Plotnikov
2007-05-07	0.2.0	Updated according to changes introduced in the framework at the version 0.2.0.	Constantine Plotnikov
2007-05-16	0.3.0	Updated according to changes introduced in the framework at the version 0.3.0.	Constantine Plotnikov
2007-05-22	0.3.1	Updated according to changes introduced in the framework at the version 0.3.1.	Constantine Plotnikov
2007-06-04	0.3.2	Updated according to changes introduced in the framework at the version 0.3.2. Minor clarifications in description of asynchronous components and utilities	Constantine Plotnikov

Table of Contents

1 Introduction

The goal of this document is to describe how to use the AsyncObjects framework. The document introduces core classes of framework and their intended usage patterns. For more information consult javadoc and source code of the framework.

All code samples used here are also distributed in the AsyncObjects framework package.

2 Framework Overview

The AsyncObjects framework is created to simplify creation of asynchronous programs in Java. The framework classes are modeled after syntax of E programming language (http://www.erights.org). However the framework features some own utilities that support asynchronous programming.

The framework uses the following basic concepts:

- A *vat* is lightweight container for asynchronous objects. All asynchronous objects live in context of some vat. The vat provides *event queue* used by the objects. The vat can accept events. The events are dispatched and executed by *vat runners*. All events for the vat are executed in order they have been accepted. However ordering of events is not guaranteed within single vat runner. Event in one vat can execute earlier than event in other vat, even if it has been accepted later.
- A *promise* is an object that represents the outcome of an asynchronous operation. A promise is initially created in unresolved state meaning that operation is not yet finished. Eventually a promise becomes either resolved with some value meaning that operation has finished successfully or smashed with some `Throwable` meaning that operation has failed. The promise state is modified using the resolver object associated with promise. The promise also notifies listener resolver objects when it becomes smashed or resolved.
- An *asynchronous object* is a Java object that belongs to some vat and implements some asynchronous methods. An asynchronous object implements one or more *asynchronous interface* that declares *asynchronous methods*. The asynchronous interface must directly or indirectly extend the `AsyncObject` interface. The framework generates a proxy implementation of asynchronous interface that dispatches events to the vat of wrapped asynchronous object. While asynchronous interfaces are implemented directly, the object should not be used directly from other threads, asynchronous proxy interface should be used instead. An implementation class for asynchronous object must directly or indirectly extend the class `AsyncUnicastServer`.
- Asynchronous methods can be either *oneway* or *request-response*. Oneway methods are fire-and-forget methods. Request-response methods have an associated promise.

Basing on these basic quite complex asynchronous services might be built. There are also utility classes that simplify usage of these classes.

Currently framework is focused on creation of asynchronous network services. So its biggest part is network IO API and IO utility classes.

3 Core Concepts

3.1 Vats and Vat Runners

Asynchronous objects can be created and asynchronous operations might be run only in a context of some vat.

However the vat itself is not aware about asynchronous objects. The vat is just a wrapper for some event queue. The events can be posted to the vat. The vat runner associated with the vat reads events from the vat and dispatches them. Each event implements the interface `java.lang.Runnable`. So its method `run()` is executed. Events in the vat are dispatched to objects sequentially in the order they are posted to the vat. Therefore objects that belong to the one vat, are never executed in concurrent threads at the same time. Several activities might be executed in interleaved way. However boundaries of each action that belongs to an activity are explicit.

This makes a vat a unit of concurrency. If some activities should happen in different threads at the same time, they should belong to different vats.

The most vat runners support the property `batchingFactor`. This property specifies how many events from a single vat the runner should execute at one time. Increasing this property will likely improve throughput because, because there more chances that work will be done over the same objects. Decreasing this property will increase fairness, because events from different vats will be interleaved more often. It is recommended to leave this property as is.

There are several ways to create a vat and the vat creation process depends on the vat runner type. It is also possible to create a new type of the vat and vat runners when needed.

3.1.1 `Vat`

The class Vat is an default implementation Vat and it is a base class for all vats. It is recommended to create a Vat using newVat() methods on corresponding runner. The vat is than use this runner to execute its events. The samples below demonstrate how to create vat and attach it to the runner. The subclasses of this class might provide additional services for components deployed in it.

The vat can be attached and detached from runners. However this functionality is currently experimental and it should be used with care.

Note that the vat could be created using constructor or factory method `newVat(String name)` on the runner. The later method creates a vat with implementation that is most suitable for runner. However these vats might have additional restrictions like not being attachable to other runners.

3.1.2 `SingleThreadRunner` and `AsyncAction`

This is most primitive runner. When it is run, it occupies the single thread. When event is posted to a vat that is attached to this runner, the vat is added to the end of the runner's queue. The runner waits until vat appears in the vat queue and dispatches at most `batchingFactor` amount of events in the vat. If vat still has events, it is added to the end of the queue.

There are three ways to create the vat `SingleThreadRunner`. The first two ways involve direct creation of a vat and a runner. In both cases the vat should be created using one of the constructors.

The runner might be created to occupy the current thread. This is particularly useful for starting vat in the main thread of the application. This done using `startInCurrentThread()` method.

```
public class SingleThreadRunnerSample1 {
  public static void main(String[] args) {
    final SingleThreadRunner runner = new SingleThreadRunner();
    Vat vat = new Vat(runner,"my vat");
    Runnable r = new Runnable() {
      public void run() {
        System.out.println("Inside the vat: "
                  +Thread.currentThread().getName());
        runner.stop();
      }
    };
    vat.enqueue(r);
    System.out.println("Before starting runner: "
            +Thread.currentThread().getName());
    runner.startInCurrentThread();
    System.out.println("Runner finished: "
            +Thread.currentThread().getName());
  }
}
```

The sample prints the following (note that all activities happen in the same thread):

```
Before starting runner: main
Inside the vat: main
Runner finished: main
```

The vat could be also started in new thread using the method `startInNewThread()`.

```
public class SingleThreadRunnerSample2 {
  public static void main(String[] args) {
    final SingleThreadRunner runner =
      new SingleThreadRunner("other runner",   true);
    Vat vat = runner.newVat("my vat");
    final Semaphore s = new Semaphore(0);
    Runnable r = new Runnable() {
      public void run() {
        System.out.println("Inside the vat: "
            + Thread.currentThread().getName());
        runner.stop();
        s.release();
      }
    };
    vat.enqueue(r);
    System.out.println("Before runner vat: "
        + Thread.currentThread().getName());
    runner.startInNewThread();
    try {
      s.acquire();
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
    System.out.println("Runner finished: "
        + Thread.currentThread().getName());
  }
}
```

The sample prints the following (note that action is executed in other thread):

```
Before runner vat: main
Inside the vat: other runner
Runner finished: main
```

The third way is a more interesting. There is the utility class `AsyncAction`. It creates an instance of the `SingleThreadRunner` in the method `doInCurrentThread()`.

```
public class SingleThreadRunnerSample3 {
  public static void main(String[] args) {
    System.out.println("Before starting action.");
    try {
      String v = new AsyncAction<String>() {
        @Override
        public Promise<String> run() throws Throwable {
          System.out.println("Inside the vat");
          return new Promise<String>("A value from vat.");
        }
      }.doInCurrentThread();
      System.out
          .println("Action finished successfully with value: "
              + v);
    } catch (InvocationTargetException ex) {
      System.out.println("Exception during action execution.");
      ex.getTargetException().printStackTrace(System.out);
    }
  }
}
```

The sample prints the following:

```
Before starting action.
Inside the vat
Action finished successfully with value: A value from vat.
```

Note that this way allows returning results of asynchronous computation as promise. The vat automatically finishes when promise returned by `run()` is resolved. This value will be returned from `doInCurrentThread()`. If asynchronous operation started by `run()` method fails, an exception `InvocationTargetException` is thrown that wraps the exception with which promise was smashed.

The last way is recommended if there is a need to start the vat in the current thread. Note that it uses promise class that will be explained in other sections.

3.1.3 `ExecutorRunner`

This is a runner for vats that uses thread pool to execute vats. The runner can use thread pools provided by `java.util.concurrent.ThreadPoolExecutor`. However it relies on `java.util.concurrent.Executor` interface, so it is easy to provide own executor implementations. However the runner assumes that events are executed asynchronously and that the

method `execute(Runnable)` returns immediately after scheduling the task. The static factory methods create runners over executors created with `java.util.concurrent.Executors` utility class.

Thus when vats do nothing, they do not occupy the thread permanently. When event arrives to the vat, vat is scheduled for dispatching with executor. The executor gets dispatch action from the action queue and to executes it causing scheduled events to dispatch. After vat finished with executing enqueued requests, it frees the occupied thread. At most `batchingFactor` events will be executed.

This runner is created to help implementation of asynchronous objects over synchronous API. For example, the thread-based implementation of network API over blocking socket API uses it.

```
public class ExecutorRunnerSample {
  public static void main(String[] args) {
    final ExecutorRunner runner = ExecutorRunner
        .newFixedThreadPoolRunner(2, "pool runner thread ",
            true);
    // the semaphore requires two releases before
    // it could be aquired.
    final Semaphore s = new Semaphore(1 - 2);
    Runnable r = new Runnable() {
      public void run() {
        System.out.println("Inside the vat '"
            + Vat.current().getName() + "' in the thread '"
            + Thread.currentThread().getName() + "'");
        s.release();
      }
    };
    System.out.println("Before starting vats: "
        + Thread.currentThread().getName());
    new Vat(runner, "vat 1").enqueue(r);
    new Vat(runner, "vat 2").enqueue(r);
    try {
      s.acquire();
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
    System.out.println("Vats finished: "
        + Thread.currentThread().getName());

  }
}
```

The sample prints the following text:

```
Before starting vats: main
Inside the vat 'vat 1' in the thread 'pool runner thread 0'
Inside the vat 'vat 2' in the thread 'pool runner thread 1'
Vats finished: main
```

The actions are executed in threads allocated by `Executor` instances. Note that threads will continue running even when vat finished using them. However threads are can be run in the daemon

mode, so they do not prevent application from exiting if this is needed.

3.1.4 Vat Sensors

The AsyncObjects framework could also use existing event loops to implement a vat over them. This is done using Vat sensors.

When current vat is being located, firstly a thread local variable is being consulted. If no vat is associated with current thread, registered Vat Sensors are queried if the vat could be provided over event loop that is being run the current thread.

As example, `AWTVatSensor` is provided that provides a Vat over AWT event loop. However, it is not registered by default since AWT API is not part of Foundation Profile 1.1.

3.2 Asynchronous Components

While it is possible to work with the vats directly, it is not very convenient. To automate some common tasks related to events, the framework provide a notion of asynchronous component. The component consists of asynchronous interface and implementation class.

Asynchronous interface is an interface that extends `AsyncObject` interface and provides only asynchronous methods on it.

The following methods are treated as asynchronous:

- Methods that has void return type. They are treated as one way operations. These methods can be called from any thread.
- Methods that return `Promise` type. They are treated as asynchronous request-response operations. These methods must be called from a vat thread.
- Methods that return an asynchronous interface type. They are treated as asynchronous request-response operations. These methods must be called from a vat thread. Such methods are considered as experimental feature. Support for them might be dropped at some later version.

```
public interface ASimpleObject extends AsyncObject {
  void onewayMethod();
  Promise<Integer> requestRespopnseMethod1();
  ASimpleObject requestRespopnseMethod2();
  Promise<Integer> requestRespopnseMethod3();
  ASimpleObject requestRespopnseMethod4();
  Promise<Object> requestRespopnseMethod5();
}
```

The implementation class implements asynchronous interfaces.. However instances of this implementation class should not be used directly. The proxy implementations of asynchronous interfaces are provided by framework and this proxies enqueues events for the asynchronous objects to the vat to which the object belongs, thus allowing to avoid concurrency issues associated with multi-threaded usage of the object.

Since object is invoked only from the thread of own vat, it is efficiently executed in a single thread manner. So the classical race conditions are simply impossible. Several simultaneous multi-turn

asynchronous operations are possible on the single object. Their turns just are interleaved. However the turn boundary is a much more clear and coarse grained boundary than boundary of atomic memory operation in Java concurrency model (for example storing int is atomic in Java, but storing long is potentially not atomic). It is also possible to ensure that turns of different asynchronous operation do not interleave using the `RequestQueue` class described in later sections.

The framework provides two proxy implementations of asynchronous interface:

- Immediate – this implementation directly reference instance of implementation class. Such proxy is created using `export()` operation on an asynchronous object.
- Buffering – this implementation is created over promise and remembers events that are posted to it. When promise is resolved, it forwards all events to it in order in which they have been received. This kind of proxy is returned form asynchronous methods that has an asynchronous interface return type. Or it could be created using `Promise.willBe(Class)` operation.

The method `AsyncObject.dereference(AResolver)` allows to eventually receive immediate implementation from buffering one.

An implementation class must directly or indirectly extend `AsyncUnicastServer` class. The class must declare which asynchronous interfaces they support by implementing them. The class also might supply a type as generic argument to `AsyncUnicastServer` class. This type must be compatible with implemented generic interfaces. Since interface is implemented directly, the complier will report an error if some method is not implemented. The class must be public because its methods are accessed using reflection by the framework, and these methods are sometimes accessed from other class loaders. However the class might be an non-anonymous inner class.

One-way operations do not return a value and runtime exceptions and errors are simply logged and discarded even if thrown.

For request-response operations the value is treated a bit differently:

- If exception was thrown, than operation fails with the exception.
- If value is null, null is considered to be result of operations. Note that methods that has proxy return type, will smash returned buffering proxy with `NullPointerException`.
- If value is Promise, request-response operation will finish with result of returned promise when promise is resolved.
- If value is buffering instance of asynchronous interface, it is treated the same as underlying promise.

```
public class SimpleObject extends
    AsyncUnicastServer<ASimpleObject> implements ASimpleObject {
  public void onewayMethod() {
    // do something
    System.out.println("One way method called.");
  }
  public Promise<Integer> requestRespopnseMethod1() {
    // simple implementation that returns scalar value
    return new Promise<Integer>(2);
  }
  public Promise<ASimpleObject> requestRespopnseMethod2() {
```

```
    // this method returns a reference to self
    return this.promise();
  }
  public Promise<Integer> requestRespopnseMethod3() {
    // this method returns a promise of that
    // is result of asynchronous execution of
    // requestRespopnseMethod1()
    return requestRespopnseMethod2()
      .willBe(ASimpleObject.class).requestRespopnseMethod1();
  }
  public Promise<ASimpleObject> requestRespopnseMethod4() {
    // create new object and return it
    return new SimpleObject().promise();
  }
  public Promise<Object> requestRespopnseMethod5() {
    // failing the method
    throw new IllegalStateException("I'm not going to work.");
  }
}
```

Operations sent to asynchronous object are ordered. They are processed in the order they are enqueued to the vat to which the object belongs. This implies that operations that are sent using the same proxy in the same thread are started in the order in which they are sent.

3.3 `Promise` and `AResolver`

A *promise* is an object that represent the outcome of an asynchronous operation. A promise is initially created in unresolved state meaning that operation is not yet finished. Eventually a promise becomes either resolved with some value meaning that operation has finished successfully or smashed with some `Throwable` meaning that operation has failed.

The promise state is modified using the resolver object associated with promise. The promise implements the single assignment model, so only the first `resolve()` or `smash()` operation is effective. Also a resolver could be got only once from promise. Consequent calls for resolver will cause `IllegalStateException`.

```
public class PromiseSample1 {
  public static void main(String[] args) {
    try {
      String v = new AsyncAction<String>() {
        @Override
        public Promise<String> run() throws Throwable {
          Promise<String> p = new Promise<String>();
          AResolver<String> r = p.resolver();
          // This invocation will resolve promise/
          r.resolve("Hello from vat");
          // This invocation will be ineffictive
          // since promise is already resolved.
          r.smash(new Exception("I want to smash it."));
          return p;
        }
      }.doInCurrentThread();
      System.out.println("Action finished "
          + "successfully with value: " + v);
    } catch (InvocationTargetException ex) {
      System.out.println("Exception during action execution.");
```

```
      ex.getTargetException().printStackTrace(System.out);
    }
  }
}
```

The sample prints the following:

```
16.05.2007 15:23:57 net.sf.asyncobjects.Promise$InternalPromiseResolverImpl
smash
SEVERE: promise net.sf.asyncobjects.Promise@1027b4d is already resolved or
broken
Action finished successfully with value: Hello from vat
```

Note that a error message that promise attempted to be resolved twice is reported. Such situation is usually a sign of some bug in program logic. So it is reported on level SEVERE. The result of the entire asynchronous action the string `"Hello from vat"` because the first call to the resolver were with that value.

The promise also notifies listener resolver objects when it becomes smashed or resolved. They are an asynchronous objects as well, in order to prevent concurrency issues. If promise is already resolved or smashed, listeners are notified as soon as they are registered with promise.

```
public class PromiseSample2 {
  public static class SampleResolver extends
      AsyncUnicastServer<AResolver<String>> implements AResolver<String>{
    final AResolver<String> finalResolver;
    final String name;
    SampleResolver(AResolver<String> finalResolver, String name) {
      this.finalResolver = finalResolver;
      this.name = name;
    }

    public void smash(Throwable ex) {
      System.out.println("Resolver " + name
          + ": promise was smashed: " + ex.toString());
      finalResolver.smash(ex);
    }

    public void resolve(String value) {
      System.out.println("Resolver " + name
          + ": promise was resolved: " + value);
      finalResolver.resolve(name + ": "+ value);
    }
  }

  public static void main(String[] args) {
    try {
      Object v[] = new AsyncAction<Object[]>() {
        @Override
        public Promise<Object[]> run() throws Throwable {
          Promise<String> p = new Promise<String>();
          AResolver<String> r = p.resolver();
          // This invocation will resolve promise
          r.resolve("hello!!");
          Promise<String> r1_finished = new Promise<String>();
```

```
            Promise<String> r2_finished = new Promise<String>();
            AResolver<String> r1 = new SampleResolver(r1_finished
                .resolver(), "r1").export();
            AResolver<String> r2 = new SampleResolver(r2_finished
                .resolver(), "r2").export();
            p.dereference(r1);
            p.dereference(r2);
            return Wait.all(r1_finished, r2_finished);
          }
        }.doInCurrentThread();
        System.out.println("Action finished successfully: " +
            Arrays.asList(v));
      } catch (InvocationTargetException ex) {
        System.out.println("Exception during action execution.");
        ex.getTargetException().printStackTrace(System.out);
      }
    }

}
```

The sample prints the following:

```
Resolver r1: promise was resolved: hello!!
Resolver r2: promise was resolved: hello!!
Action finished successfully: [r1: hello!!, r2: hello!!]
```

When the promise `p` is resolved, it notifies resolvers `r1` and `r2`. Which in their turn print messages and resolve promises `r1_finished` and `r2_finished`. The operation `Wait.all(...)` creates a promise that resolves to an array of values when all promises passed as arguments to the operation are resolved.

The example also demonstrates how to implement a generic asynchronous interfaces. Note that resolver implementation has method resolve that takes `String` as argument rather than `Object`.

Note that while it is possible to create a listener resolver manually, it is recommended to use the `When` utility class instead.

3.4 Utility Classes

Utility classes share the following common principles (`AsyncAction` is an exception that violates almost all of them).

- Each utility classes represent some control construct for asynchronous programming.
- The operation body is executed on some later turn in the context of the same vat unless it is specified otherwise. The condition for executing the body differs for different utility classes. A notable exception from the “later turn” rule is `Any` utility class that starts all operations in its body on the same turn where the instance is created.
- When the operation finishes, a promise returned with `promise()` resolves. The promise from operation should be obtained at most once and on the turn that creates the operation.

3.4.1 When

Creating components that implement the `AResolver` interface each time the promise should be listened is problematic due to the following reasons:

- A resolver must be an asynchronous component in order to avoid concurrency issues.
- An implementation class of asynchronous component must be a public class. So it is not possible to create resolver as anonymous inner class.
- Typically, after doing some work, a resolver needs to produce some value itself. This have to be done by notifying the next resolver in the chain.

The class `When` is an utility class that simplifies listening on promise, by addressing these issues. If the argument promise is resolved, `resolved(Object value)` method is called. If the argument promise is smashed, the method `smashed(Throwable problem)` is called. Note that these methods are called in the same vat where When instance was created. Smashed method could re-throw exception or it could eat exception.

```
public class WhenSample1 {
  public static void main(String[] args) {
    try {
      Object v[] = new AsyncAction<Object[]>() {
        @Override
        public Promise<Object[]> run() throws Throwable {
          Promise<String> sp = new Promise<String>();
          sp.resolver().resolve("hello!!");
          Promise<String> fp = new Promise<String>();
          fp.resolver().smash(new Exception("Smash it!"));
          Promise<String> wp1 = new When<String, String>(sp) {
            @Override
            protected Promise<String> resolved(String value)
                throws Throwable {
              return new Promise<String>("resolved with " + value);
            }
          }.promise();
          Promise<String> wp2 = new When<String, String>(fp) {
            @Override
            protected Promise<String> resolved(String value)
                throws Throwable {
              return new Promise<String>("unexpectedly resolved with " + value);
            }

            @Override
            protected Promise<String> smashed(Throwable problem)
                throws Throwable {
              Promise<String> p = new Promise<String>();
              p.resolver().resolve("Smashed with " + problem);
              return p;
            }
          }.promise();
          return Wait.all(wp1, wp2);
        }
      }.doInCurrentThread();
      System.out.println("Action finished successfully: " +
          Arrays.asList(v));
```

```
    } catch (InvocationTargetException ex) {
      System.out.println("Exception during action execution.");
      ex.getTargetException().printStackTrace(System.out);
    }
  }
}
```

The sample prints the following:

```
Action finished successfully: [resolved with hello!!, Smashed with
java.lang.Exception: Smash it!]
```

The sample above also demonstrate that, the when construct could be also used to create a promise. This promise is resolved or smashed depending on outcome of `resolved(Object value)` or `smashed(Throwable problem)` method correspondingly. Note that promise can be created only on the same turn as the when instance is created. If promise is not created, when operation is executed more efficiently. So by default when is executed assuming that promise is not needed..

If the method `resolved(Object value)` fails, the method `bodySmashed(Throwable problem)` is called. This method by default calls `smashed(Throwable problem)`. The following example demonstrate how to create a promise that succeeds only if the argument promise was smashed with specific exception class and fails otherwise.

```
public class WhenSample2 {
  public static Promise<Throwable> expectFailure(Promise<?> p,
      final Class<?> exceptionClass) {
    return new When<Object, Throwable>(p) {
      @Override
      protected Promise<Throwable> resolved(Object value) throws Throwable {
        throw new RuntimeException(
            "Success value is not expected: " + value);
      }

      @Override
      protected Promise<? extends Throwable> bodySmashed(Throwable problem)
          throws Throwable {
        // note that super is called here that
        // rethrows exception so even if runtime
        // exception is expected, it will not
        // trigger false positive
        return super.smashed(problem);
      }

      @Override
      protected Promise<? extends Throwable> smashed(Throwable problem)
          throws Throwable {
        if (problem.getClass() == exceptionClass) {
          return new Promise<Throwable>(problem);
        }
        return super.smashed(problem);
      }
    }.promise();
  }
  public static void main(String[] args) {
```

```
    try {
      Object v[] = new AsyncAction<Object[]>() {
        @Override
        public Promise<Object[]> run() throws Throwable {
          Promise<?> fp1 = new Promise<Object>();
          fp1.resolver()
              .smash(new RuntimeException("Expected"));
          Promise<?> fp2 = new Promise<Object>();
          fp2.resolver().smash(
              new NullPointerException("Other one!"));
          Promise<String> sp = new Promise<String>();
          sp.resolver().resolve("hello!!");
          Promise<Throwable> r1 = expectFailure(fp1,
              RuntimeException.class);
          class MyWhen extends When<Object, String> {
            MyWhen(Promise<? extends Object> p) {
              super(p);
            }
            @Override
            protected Promise<String> resolved(Object value)
                throws Throwable {
              return new Promise<String>("unexpected resolved: " + value);
            }

            @Override
            protected Promise<String> smashed(Throwable problem)
                throws Throwable {
              return new Promise<String>("expected failure: " + problem);
            }
          }
          Promise<String> r2 = new MyWhen(expectFailure(fp2,
              RuntimeException.class)).promise();
          Promise<String> r3 = new MyWhen(expectFailure(sp,
              RuntimeException.class)).promise();
          return Wait.all(r1, r2, r3);
        }
      }.doInCurrentThread();
      System.out.println("Action finished successfully: " +
          Arrays.asList(v));
    } catch (InvocationTargetException ex) {
      System.out.println("Exception during action execution.");
      ex.getTargetException().printStackTrace(System.out);
    }
  }
}
```

The sample prints the following:

```
Action finished successfully: [java.lang.RuntimeException: Expected, expected
failure: java.lang.NullPointerException: Other one!, expected failure:
java.lang.RuntimeException: Success value is not expected: hello!!]
```

The when class also has `finallyDo()` method that allows to execute action right before a result promise is resolved. This method can return a promise, throw an exception, or null. If a promise is returned from finally, the method waits until promise is resolved. If null is returned, it is treated as a

promise resolved with null. If an exception is thrown from the method or returned promise is smashed, the result promise is smashed with that exception. The sample below demonstrates it. It uses `expectFailure` method form previous sample.

```
public class WhenSample3 {
  public static void main(String[] args) {
    try {
      Object v[] = new AsyncAction<Object[]>() {
        @Override
        public Promise<Object[]> run() throws Throwable {
          // success promise
          Promise<Object> fp = new Promise<Object>();
          fp.resolver().smash(
              new NullPointerException("Expected"));
          Promise<Object> sp = new Promise<Object>();
          sp.resolver().resolve("value");
          class FinallyFailingWhen extends When<Object, Object> {
            FinallyFailingWhen(Promise<Object> p) {
              super(p);
            }

            @Override
            protected Promise<Object> resolved(Object value)
                throws Throwable {
              return new Promise<Object>(value);
            }

            @Override
            protected Promise<?> finallyDo() throws Throwable {
              throw new Exception("finally exception");
            }
          }
          Promise<Object> r1 = new FinallyFailingWhen(sp)
              .promise();
          Promise<Object> r2 = new FinallyFailingWhen(fp)
              .promise();
          return Wait.all(WhenSample2.expectFailure(r1,
              Exception.class), WhenSample2.expectFailure(r2,
              Exception.class));
        }
      }.doInCurrentThread();
      System.out.println("Action finished successfully: " +
          Arrays.asList(v));
    } catch (InvocationTargetException ex) {
      System.out.println("Exception during action execution.");
      ex.getTargetException().printStackTrace(System.out);
    }
  }
}
```

The following is printed by the example:

```
Action finished successfully: [java.lang.Exception: finally exception,
java.lang.Exception: finally exception]
```

The `When` class is very useful utility class and it is recommended to use it for listening on results of

asynchronous operations. This class is usually used by creating anonymous inner class, so that resolved and smashed methods has access to definitions from enclosing scope.

3.4.2 `ACloseable` and `Using`

`ACloseable` is an interface that contains a single `close()` operation. Asynchronous objects that need to free resources implement this interface. The interface is similar to `java.lang.Closeable` interface introduced in Java SE 5.

The class `Using` supports automatic closing of the closeable resource after some asynchronous operation finishes. The class uses `finallyDo()` method on `When` construct internally. The example below demonstrate how Using works:

```
public class UsingSample1 {
  public interface AMyCloseable extends ACloseable {
    Promise<String> doSomething();
  }
  public static class MyCloseable extends
      AsyncUnicastServer<AMyCloseable> implements AMyCloseable {
    public Promise<String> doSomething() {
      System.out.println("Doing something!");
      return new Promise<String>("done something");
    }
    public Promise<Void> close() {
      System.out.println("Doing close!");
      Promise<Void> p = new Promise<Void>();
      p.resolver().resolve(null);
      return p;
    }

  }
  public static void main(String[] args) {
    try {
      String v = new AsyncAction<String>() {
        @Override
        public Promise<String> run() throws Throwable {
          return new Using<AMyCloseable, String>(
              new MyCloseable().export()) {
            @Override
            protected Promise<String> run(AMyCloseable service) {
              return service.doSomething();
            }
          }.promise();
        }
      }.doInCurrentThread();
      System.out.println("Action finished successfully: " + v);
    } catch (InvocationTargetException ex) {
      System.out.println("Exception during action execution.");
      ex.getTargetException().printStackTrace(System.out);
    }
  }
}
```

The program prints the following during execution:

```
Doing something!
Doing close!
Action finished successfully: done something
```

As you can see the object's close method is called before process is finished.

3.4.3 `RequestQueue` and `Serialized`

Sometimes it is useful not to allow next operation to proceed until previous one is finished. This might happen when unresolved promise is returned. The messages could dispatched to the asynchronous object while it waits for some external event to happen. The AsyncObjects framework provide two utility classes that helps to handle this situation.

Request queue is such synchronization point for requests. Before starting processing a request code `startRequest()` method is called. The method returns a promise that resolves when all previous requests that are also serialized over this request queue has finished. After method finished its work, it invokes `finishRequest()` method. That causes resolution of promise associated with the next request. It is also possible to start request right away if it is known that there are no active requests. This can be done manually, but there is utility class `Serialized` that simplifies writing such requests.

This the class automatically calls the method `finishRequest()` when Promise returned from `run()` is resolved or smashed (non-promise values or exceptions are treated the same as resolved or smashed promise). Using Serialized class working with request queue is a simpler and less error prone process.

```
public class RequestQueueSample1 {
  public interface AMyService extends AsyncObject {
    Promise<Void> doIt(String id);
  }
  public static class MyServiceNotSerialized extends
      AsyncUnicastServer<AMyService> implements AMyService {
    public Promise<Void> doIt(final String id) {
      System.out.println("Started request " + id);
      return new When<Void, Void>(new Promise<Void>(null)) {
        @Override
        protected Promise<Void> resolved(Void value) throws Throwable {
          System.out.println("Doing request step " + id);
          return new Promise<Void>(null);
        }

        @Override
        protected Promise<?> finallyDo() throws Throwable {
          System.out.println("Finishing request " + id);
          return null;
        }
      }.promise();
    }
  }

  public static class MyServiceSerialized extends
```

```
      MyServiceNotSerialized {
    private final RequestQueue queue = new RequestQueue();
    @Override
    public Promise<Void> doIt(final String id) {
      System.out.println("Accepting request " + id);
      return new Serialized<Void>(queue) {
        @Override
        protected Promise<Void> run() throws Throwable {
          return MyServiceSerialized.super.doIt(id);
        }
      }.start();
    }
  }
  static Promise<Object[]> doRequests(
      MyServiceNotSerialized service) {
    AMyService s = service.export();
    Promise<Void> p1 = s.doIt("R1");
    Promise<Void> p2 = s.doIt("R2");
    Promise<Void> p3 = s.doIt("R3");
    return Wait.all(p1, p2, p3);
  }

  public static void main(String[] args) {
    try {
      new AsyncAction<Object>() {
        @Override
        public Promise<Object> run() throws Throwable {
          return new Seq<Object>() {
            @Override
            protected Object run1() throws Throwable {
              System.out.println("Non serialized execution...");
              return doRequests(new MyServiceNotSerialized());
            }

            @Override
            protected Object run2() throws Throwable {
              System.out.println("Serialized execution...");
              return doRequests(new MyServiceSerialized());
            }
          }.promise();
        }
      }.doInCurrentThread();
      System.out.println("Action finished successfully");
    } catch (InvocationTargetException ex) {
      System.out.println("Exception during action execution.");
      ex.getTargetException().printStackTrace(System.out);
    }
  }
}
```

The program prints the following:

```
Non serialized execution...
Started request R1
Started request R2
Started request R3
Doing request step R1
Doing request step R2
```

```
Doing request step R3
Finishing request R1
Finishing request R2
Finishing request R3
Serialized execution...
Accepting request R1
Started request R1
Accepting request R2
Accepting request R3
Doing request step R1
Finishing request R1
Started request R2
Doing request step R2
Finishing request R2
Started request R3
Doing request step R3
Finishing request R3
Action finished successfully
```

Note that requests for `MyServiceNonSerialized` are executed in interleaved fashion. However requests for `MyServiceSerialized` are executed in one after other, even though they all are accepted in beginning. The sample also demonstrate usage of `Seq` utility class that allow specifying that one asynchronous action should be executed after other one finishes. The result of this class is the value returned from the last action.

The request queue also supports analogs of `Object.wait()` and `Object.notify()` methods. A promise returned from method `awaken()` is resolved when `awake()` method is called on the request queue.

```
public class RequestQueueSample2 {
  public interface AMyService extends AsyncObject {
    Promise<String> doIt(AResolver<Void> resolver, String id);
    void awake();
  }
  public static class MyService extends
      AsyncUnicastServer<AMyService> implements AMyService {
    private final RequestQueue queue = new RequestQueue();
    public Promise<String> doIt(final AResolver<Void> resolver,
        final String id) {
      System.out.println("Accepting request " + id);
      return new Serialized<String>(queue) {
        @Override
        protected Promise<String> run() throws Throwable {
          System.out.println("Sleeping in request " + id);
          resolver.resolve(null);
          return new When<Void, String>(queue.awaken()) {
            @Override
            protected Promise<String> resolved(Void value)
                throws Throwable {
              System.out.println("Awaken request " + id);
              return new Promise<String>(id);
            }

            @Override
            protected Promise<?> finallyDo() throws Throwable {
```

```
              System.out.println("Finishing request " + id);
              return null;
            }
          }.promise();
        }
      }.promise();
    }
    public void awake() {
      queue.awake();
    }
  }
  static Promise<String> doRequest(final AMyService s,
      final String id) {
    Promise<Void> p = new Promise<Void>();
    new When<Void, Void>(p) {
      @Override
      protected Promise<Void> resolved(Void value) throws Throwable {
        System.out.println("Awakening request " + id);
        s.awake();
        return null;
      }
    };
    return s.doIt(p.resolver(), id);
  }
  public static void main(String[] args) {
    try {
      Object v[] = new AsyncAction<Object[]>() {
        @Override
        public Promise<Object[]> run() throws Throwable {
          AMyService s = new MyService().export();
          Promise<String> p1 = doRequest(s, "R1");
          Promise<String> p2 = doRequest(s, "R2");
          Promise<String> p3 = doRequest(s, "R3");
          return Wait.all(p1, p2, p3);
        }
      }.doInCurrentThread();
      System.out.println("Action finished successfully: "
          + Arrays.asList(v));
    } catch (InvocationTargetException ex) {
      System.out.println("Exception during action execution.");
      ex.getTargetException().printStackTrace(System.out);
    }
  }
}
```

The program prints the following:

```
Accepting request R1
Sleeping in request R1
Accepting request R2
Accepting request R3
Awakening request R1
Awaken request R1
Finishing request R1
Sleeping in request R2
Awakening request R2
Awaken request R2
```

```
Finishing request R2
Sleeping in request R3
Awakening request R3
Awaken request R3
Finishing request R3
Action finished successfully: [R1, R2, R3]
```

Note that invocation of these methods is allowed only when some operation is currently active. These `awake()` and `awaken()` methods used of for implementing several interacting classes that share some common resource. In the framework it is used for implementation of `BytePipe` where input and output share the common buffer.

3.4.4 `AsyncProcess`

This class simplifies writing long-running asynchronous operations. The operation is started by calling start method that return promise that resolves when process is terminated.

The operation entry point is the `run()` method. The operation terminates when either `success()` or `failure()` method is called on `AsyncProcess` object. A provided inner class `AsyncProcess.ProcessWhen` behaves just like `When` class, but its smash operation automatically terminates operation by calling the `failure()` method. However success method should be called manually. And care should be taken that success method is eventually called.

```
public class AsyncProcessSample1 {
  static Promise<Integer> countPairs(final int initial) {
    return new AsyncProcess<Integer>() {
      @Override
      protected void run() throws Throwable {
        new ProcessWhen<Integer>(new Promise<Integer>(initial)) {
          @Override
          protected Promise<Void> resolved(Integer value)
              throws Throwable {
            if (initial < 0) {
              // ProcessWhen will catch this exception
              // and smash the the process promise
              throw new IllegalArgumentException(
                  "The value should be non-negative");
            }
            doStep(value.intValue(), 0);
            return null;
          }
        };
      }

      private void doStep(int value, final int rc) {
        if (value == 0) {
          success(new Integer(rc));
          return;
        } else if (value == 1) {
          // explicit usage of failure here
          failure(new RuntimeException(
              "Odd values are not allowed"));
          return;
        } else {
```

```
          new ProcessWhen<Integer>(new Promise<Integer>(
              value - 2)) {
            @Override
            protected Promise<Void> resolved(Integer value)
                throws Throwable {
              doStep(value.intValue(), rc + 1);
              return null;
            }
          };
        }
        ;
      }
    }.promise();
  }
  public static void main(String[] args) {
    try {
      Object v[] = new AsyncAction<Object[]>() {
        @Override
        public Promise<Object[]> run() throws Throwable {
          Promise<Integer> p1 = countPairs(0);
          Promise<Integer> p2 = countPairs(6);
          Promise<Throwable> p3 = WhenSample2.expectFailure(
              countPairs(-1), IllegalArgumentException.class);
          Promise<Throwable> p4 = WhenSample2.expectFailure(
              countPairs(3), RuntimeException.class);
          return Wait.all(p1, p2, p3, p4);
        }
      }.doInCurrentThread();
      System.out.println("Action finished "
          + "successfully with value: " + Arrays.asList(v));
    } catch (InvocationTargetException ex) {
      System.out.println("Exception during action execution.");
      ex.getTargetException().printStackTrace(System.out);
    }
  }
}
```

The sample prints the following:

```
Action finished successfully with value: [0, 3,
java.lang.IllegalArgumentException: The value should be non-negative,
java.lang.RuntimeException: Odd values are not allowed]
```

For more realistic examples look at `IOUtils` class that uses this utility class to implement some utility operations over binary streams.

3.4.5 `AsyncAction`

This class used for many purposes in the framework. It is used to dispatch action to some other vat.

```
public class AsyncActionSample1 {
  public static void main(String[] args) {
    try {
      // create other vat
      final SingleThreadVat other = new SingleThreadVat(
          "other", null, true);
```

```
      other.start();
      // do async action in this thread
      String v = new AsyncAction<String>() {
        @Override
        public Promise<String> run() throws Throwable {
          System.out.println("Current vat: "
              + Vat.current().getName());
          return new AsyncAction<String>() {
            @Override
            public Promise<String> run() throws Throwable {
              System.out.println("Current vat: "
                  + Vat.current().getName());
              return new Promise<String>("Hello!");
            }
          }.doInOtherVat(other);
        }
      }.doInCurrentThread();
      System.out.println("Action finished "
          + "successfully with value: " + v);
    } catch (InvocationTargetException ex) {
      System.out.println("Exception during action execution.");
      ex.getTargetException().printStackTrace(System.out);
    }
  }
}
```

The sample prints the following:

```
Current vat: Action's Vat
Current vat: other
Action finished successfully with value: Hello!
```

It also simplifies running code later in the current vat. The action is appended to end of vat event queue. This can be used to split some computation over several vat turns in order to give other asynchronous operations running in the same vat a chance to do some work in meantime.

```
public class AsyncActionSample2 {
  public static void main(String[] args) {
    try {
      // do async action in this thread
      String v = new AsyncAction<String>() {
        @Override
        public Promise<String> run() throws Throwable {
          return new AsyncAction<String>() {
            @Override
            public Promise<String> run() throws Throwable {
              return new Promise<String>("Hello!");
            }
          }.executeLater();
        }
      }.doInCurrentThread();
      System.out.println("Action finished "
          + "successfully with value: " + v);
    } catch (InvocationTargetException ex) {
      System.out.println("Exception during action execution.");
      ex.getTargetException().printStackTrace(System.out);
    }
```

```
  }
}
```

The sample prints the following:

```
Action finished successfully with value: Hello!
```

It is also used to execute a asynchronous code in the current thread blocking while operation is not yet complete. Such things are sometimes required when interfacing with AsyncObjects components form synchronous code. Both examples in this section, demonstrate such usage of `AsyncAction` for implementing main method of Java application.

3.4.6 `Seq`, `All`, and `Any`

These classes simplify organization of asynchronous operations within single process. Utility class `Seq` ensures that the next action is executed only if the previous one is complete. This can be also done using nested `AsyncAction` and `When` utilities. However `Seq` simplifies writing such control flow. The result of the last action is returned. If asynchronous operation started by one of `run()` method fails, other steps are not completed. The method `finallyDo` is executed after last operation finishes and before promise is resolved.

```
public class SeqSample1 {
  public static void main(String[] args) {
    try {
      // do async action in this thread
      Object v = new AsyncAction<String>() {
        @Override
        public Promise<String> run() throws Throwable {
          return new Seq<String>() {
            private Promise<String> doStep(final String id) {
              // This is async operation that
              // spreads across several turns
              System.out.println("Starting step: " + id);
              return new When<Void, String>(new Promise<Void>(
                  null)) {
                @Override
                protected Promise<String> resolved(Void value)
                    throws Throwable {
                  return new Promise<String>(id);
                }

                @Override
                protected Promise<?> finallyDo()
                    throws Throwable {
                  System.out.println("Finishing step: " + id);
                  return null;
                }
              }.promise();
            }

            @Override
            protected Object run1() throws Throwable {
              return doStep("1");
            }

            @Override
```

```
            protected Object run2() throws Throwable {
              return doStep("2");
            }

            @Override
            protected Object run3() throws Throwable {
              return doStep("3");
            }

            @Override
            protected Promise<?> finallyDo() throws Throwable {
              System.out.println("Finished all steps");
              return null;
            }
          }.promise();
        }
      }.doInCurrentThread();
      System.out.println("Action finished "
          + "successfully with value: " + v);
    } catch (InvocationTargetException ex) {
      System.out.println("Exception during action execution.");
      ex.getTargetException().printStackTrace(System.out);
    }
  }
}
```

The sample prints the following:

```
Starting step: 1
Finishing step: 1
Starting step: 2
Finishing step: 2
Starting step: 3
Finishing step: 3
Finished all steps
Action finished successfully with value: 3
```

Note that the next step does not starts until previous one is complete.

There is a very useful utility that allows to wait for outcome of several asynchronous operations. It is `Wait.all(p1, p2, ...)`. This operations returns a promise that resolves to an array of values when all promises are resolved. If at least one of promises is smashed, the result promise is smashed with exception `WaitAllException` that contains results of all operations as two lists: results and failures. The failures list contains list of failures, at position of original promises within argument array. The result lists contains returned results at position of promises. Note that if an argument promise was smashed, the position in failures list is not null and positions in result list is null. `All` utility class also has `finallyDo()` method that is invoked after all branches finished and before promise is resolved. If it fails, its exception overrides `WaitAllException` exception. Below is a samples that demonstrates usage of `All` utility class. Basic `Wait.all()` form were used in other samples.

```
public class AllSample1 {
  public static void main(String[] args) {
    try {
      // do async action in this thread
```

```
      Object[] v = new AsyncAction<Object[]>() {
        @Override
        public Promise<Object[]> run() throws Throwable {
          return new All() {
            private Promise<String> doBranch(final String id) {
              // This is async operation that
              // spreads across several turns
              System.out.println("Starting branch: " + id);
              return new When<Void, String>(new Promise<Void>(
                  null)) {
                @Override
                protected Promise<String> resolved(Void value)
                    throws Throwable {
                  Promise<String> rc = new Promise<String>();
                  rc.resolver().resolve(id);
                  return rc;
                }

                @Override
                protected Promise<?> finallyDo()
                    throws Throwable {
                  System.out.println("Finishing Branch: " + id);
                  return null;
                }
              }.promise();
            }

            @Override
            protected Object run1() throws Throwable {
              return doBranch("1");
            }

            @Override
            protected Object run2() throws Throwable {
              return doBranch("2");
            }

            @Override
            protected Object run3() throws Throwable {
              return doBranch("3");
            }

            @Override
            protected Promise<?> finallyDo() throws Throwable {
              System.out.println("Finished all branches");
              return null;
            }
          }.promise();
        }
      }.doInCurrentThread();
      System.out.println("Action finished "
          + "successfully with value: " + Arrays.asList(v));
    } catch (InvocationTargetException ex) {
      System.out.println("Exception during action execution.");
      ex.getTargetException().printStackTrace(System.out);
    }
  }
```

```
}
```

The sample prints the following:

```
Starting branch: 1
Starting branch: 2
Starting branch: 3
Finishing Branch: 1
Finishing Branch: 2
Finishing Branch: 3
Finished all branches
Action finished successfully with value: [1, 2, 3]
```

`Any` is one of most tricky of the utility classes and it requires very careful use. It returns the first result that has been computed. So it is possible to easily write the code that either opens a connection or fails after timeout. However if open operation failed due timeout, there are two issues. Firstly, the connection might open after we have already returned result. If service is created in the right way, it will be eventually garbage collected to free resources. However until it is garbage collected, it will allocate some resources and could create a connection shortage. Secondly, we might have a mean to interrupt open connection operation, so OS will not try to continue connecting. To handle these situations, the `Compensated` utility class complements the `Any` utility class. It has appropriate callbacks that are called by the `Wait.any()` method that used by the `Any` utility class. There is mode that is specifying providing `ignoreFailures` with value `true` where the class `Any` will treat failures as results with lower priority that normal return values. The sample below illustrate handling of compensated values and failures.

```
public class AnySample1 {
  static abstract class LateCompensated extends Compensated {
    final AResolver<String> finished;
    final String id;
    final Promise<Void> delay = new Promise<Void>();
    public LateCompensated(final AResolver<String> finished,
        final String id) {
      super();
      this.finished = finished;
      this.id = id;
    }
    @Override
    protected void cancel() throws Throwable {
      delay.resolver().resolve(null);
      System.out.println("executing cancel for " + id);
    }
    @Override
    protected void compensateFailure(Throwable value)
        throws Throwable {
      System.out.println("executing compensateFailure " + id
          + ": " + value);
      finished.resolve(id);
    }
    @Override
    protected void compensateValue(Object value)
        throws Throwable {
      System.out.println("executing compensateValue for " + id
          + ": " + value);
      finished.resolve(id);
```

```
    }
    @Override
    protected Object run() throws Throwable {
      System.out.println("Starting " + id + " run()");
      return new When<Void, Object>(delay) {
        @Override
        protected Promise<Object> resolved(Void value) throws Throwable {
          System.out.println("executing " + id + " run()");
          return doRun();
        }
      }.promise();
    }
    protected abstract Promise<Object> doRun();
  }

  static Compensated lateValue(final AResolver<String> finished) {
    return new LateCompensated(finished, "LV") {
      @Override
      protected Promise<Object> doRun() {
        return new Promise<Object>("a value");
      }
    };
  }
  static Compensated lateFailure(
      final AResolver<String> finished) {
    return new LateCompensated(finished, "LF") {
      @Override
      protected Promise<Object> doRun() {
        throw new RuntimeException("a failure");
      }
    };
  }
  public static void main(String[] args) {
    try {
      // do async action in this thread
      Object[] v = new AsyncAction<Object[]>() {
        @Override
        public Promise<Object[]> run() throws Throwable {
          // any ignores failures
          final Promise<String> lf = new Promise<String>();
          final Promise<String> lv = new Promise<String>();
          Promise<Object> a = new Any(true) {
            @Override
            protected Object run1() throws Throwable {
              System.out.println("throwing exception");
              // this failure will be ignored
              throw new RuntimeException();
            }

            @Override
            protected Object run2() throws Throwable {
              System.out.println("returning result");
              // this will be a value of Any
              Promise<String> rc = new Promise<String>();
              rc.resolver().resolve("RESULT");
              return rc;
            }
```

```
            @Override
            protected Object run3() throws Throwable {
              // this failure will arrive too late
              return lateFailure(lf.resolver());
            }

            @Override
            protected Object run4() throws Throwable {
              // this value will arrive too late
              return lateValue(lv.resolver());
            }

            @Override
            protected Promise<?> finallyDo() throws Throwable {
              System.out.println("Got first value");
              return null;
            }
          }.promise();
          return Wait.all(a, lf, lv);
        }
      }.doInCurrentThread();
      System.out.println("Action finished "
          + "successfully with value: " + Arrays.asList(v));
    } catch (InvocationTargetException ex) {
      System.out.println("Exception during action execution.");
      ex.getTargetException().printStackTrace(System.out);
    }
  }
}
```

The sample prints the following:

```
throwing exception
returning result
Starting LF run()
Starting LV run()
executing cancel for LF
executing cancel for LV
executing LF run()
executing LV run()
Got first value
executing compensateFailure LF: java.lang.RuntimeException: a failure
executing compensateValue for LV: a value
Action finished successfully with value: [RESULT, LF, LV]
```

The `Seq`, `Any`, and `All` utility classes can significantly simplify writing asynchronous programs.

3.4.7 Callbacks

The framework provides a number of generic callback interfaces and adapter classes that simplify their creation as inner classes.

- `ARunnable` is just an asynchronous version of `java.lang.Runnable`. Because the method `run()` happens to fit rules for oneway methods, the interface just extends it. This makes it possible to pass proxies for `ARunnable` in places where a `Runnable` instance is

expected.

- `AListener` is a generic listener interface. It has single method `onEvent(Event event)`. It is useful for creation of components that dispatched some events. Because of features of Java Generics, a single object might implement a generic interface only once. So it will not be possible to listen for different events with single object. However, the `ListenerAdpater` class allows crating listeners as anonymous inner classes, so the problem is significantly alleviated.
- `AMapper` is an asynchronous interface for function that maps one value to another. It can be used for situation when listener must produce a value.
- `ACallable` is an asynchronous interface for function that takes no arguments and produces a value. The adapter classes for this interface also support wrapping `Callable` objects from concurrency utilities.

The sample below demonstrate how to create asynchronous objects that implement these callback interfaces using anonymous inner classes.

```
public class CallbacksSample1 {
  public static void main(String[] args) {
    try {
      Object v[] = new AsyncAction<Object[]>() {
        @Override
        public Promise<Object[]> run() throws Throwable {
          System.out.println("Preparing callbacks");
          // prepare callbacks
          final ASemaphore s = new Semaphore(0).export();
          final ARunnable oneway = new RunnableAdapter() {
            protected void run() throws Throwable {
              System.out
                  .println("Oneway notification callback");
              s.release();
            }
          }.export();
          final ACallable<Integer> callable = new CallableAdapter<Integer>() {
            protected Promise<Integer> call() throws Throwable {
              System.out.println("Callable callback");
              return Promise.with(42);
            }
          }.export();
          final AMapper<Integer, Integer> mapper
                       = new MapperAdapter<Integer, Integer>() {
            protected Promise<Integer> map(Integer value)
                throws Throwable {
              System.out.println("Mapper callback with value: "
                  + value);
              return Promise.with(30 + value);
            }
          }.export();
          Promise<Integer> listenerEvent = new Promise<Integer>();
          final AResolver<Integer> resolver = listenerEvent
              .resolver();
          final AListener<Integer> listener = new ListenerAdapter<Integer>() {
            protected void onEvent(Integer event)
                throws Throwable {
```

```
              System.out
                  .println("Listener callback with event: "
                      + event);
              resolver.resolve(event);
            }
          }.export();
          // run callbacks
          System.out.println("Invoking callbacks");
          oneway.run();
          Promise<Integer> p1 = callable.call();
          Promise<Integer> p2 = mapper.map(12);
          listener.onEvent(42);
          return Wait.all(p1, p2, listenerEvent, s.acquire());
        }
      }.doInCurrentThread();
      System.out.println("Action finished successfully: "
          + Arrays.asList(v));
    } catch (InvocationTargetException ex) {
      System.out.println("Exception during action execution.");
      ex.getTargetException().printStackTrace(System.out);
    }
  }
}
```

The sample prints the following:

```
Preparing callbacks
Invoking callbacks
Oneway notification callback
Callable callback
Mapper callback with value: 12
Listener callback with event: 42
Action finished successfully: [42, 42, 42, null]
```

4 IO Library

4.1 Basic IO concepts

4.1.1 Overview

Currently the framework supports only binary IO and character IO. The frameworks uses generics to support the same utilities for both character and binary IO.

4.1.2 `BinaryData` and `TextData`

These are an immutable object that represents sequence of bytes and characters. They supports fast concatenation and subrange operations without copying data. However most important reasons for creating is usability. Because it is immutable, it is very hard to run into concurrency problems with it and it is not required track its state.

The are several ways to create objects of these types. The empty object can be obtained trivially using `empty()` method. Note that there is exactly one empty object for each type. It is also

possible to create object from bytes using `BinaryData.fromBytes()` and `TextData.fromStrings()` methods. It is also possible to concatenate objects together using `concat()` method.

It is also possible to use `BinaryDataBuilder` class to create binary data. The class resembles `StringBuilder` class by its function. It automatically converts different types to binary representation (currently only Big Endian encoding is supported for numbers).

The method `toString()` could be used to get printable representation of the object. For `BinaryData`, it is the string (encoded in hex).

4.1.3 `AInput`, `ATextInput`, and `AByteInput`

These `AInput` interface provides `close()` operation inherited from `AClosable` interface and `read(int)` operation that allows to read data from stream. The limit parameter specify maximum amount of elements that will be read using the request. Read operation returns as soon as some amount is read. It does not waits until limit is reached. To read exactly specified amount of elements, use corresponding method on `IOUtils` class.

The `AInput` interface also provides `pushback(data)` operation. This operation is optional, and streams might throw `UnsupportedOperationException` if it is not supported. This operation returns some already read elements to the stream. This operation is useful when different parts of stream are processed by different components. So even the operation have read some extra bytes, they could be returned back. This is particularly useful for implementing common Internet protocols that generally do not limit size of request line, so either the stream should be read byte by byte or unprocessed bytes have to be somehow passed to the next component.

The interfaces `ATextInput` and `AByteInput` are introduced because the direct usage of generics is problematic in many cases and it is much more cumbersome.

4.1.4 `AOutput`, `ATextOutput`, and `AByteOutput`

These interfaces extends `AClosable` interface so they supports `close()` operation. In addition to it, it supports `write(data)` operation. The promise returned from operation is resolved when stream is ready to accept more data. The client should wait until promise is resolved before sending next portion of data.

The `flush()` operation forces writing data over chains of underlying streams. The returned promise resolves when this process finishes. The semantics of `flush()` operation differs depending on the stream type. For example for sockets it means that all data has been sent to socket layer. For output stream provided by `Pipe`, it means that all data written to pipe has been already read by the client.

4.1.5 `AChannel`, `ATextChannel`, and `AByteChannel`

This is a closeable object that provides byte input and byte output. An example of byte channel is a pipe and socket.

4.1.6 Utilities

Traditional IO utilities are provided for the classes `BufferedByteInput/BufferedTextInput` and `BufferedByteOutput/BufferedTextOutput` provide buffered implementation that is similar in function to `java.io.BufferedInputStream` and `java.io.BufferedOutputStream`. `BufferedInput` also supports `pushback()` operation. `Pipe` provides implementation of `AChannel` that works as pipe (what is written to output, can be read from input).

`IOUtils` class is a bit less standard. It provides asynchronous stream forwarding, comparison, discarding. Most of these operations would have occupied at least one thread if they would have been implemented synchronously. These utilities are good example of implementing asynchronous operations without creating an asynchronous component.

The package `net.sf.asyncobjects.io.util.*` contains some some useful utility streams that simplify debugging, creation of sub-streams, and some other IO operations.

4.2 Network IO

4.2.1 `ASocketFactory`

The interface `ASocketFactory` is used to create sockets. It is does it, it allows creating server and client sockets with operations `makeServerSocket()` and `makeSocket()`. Sockets are created in non-bound and disconnected states.

4.2.2 `ASocket`

Socket is a byte channel that also supports connect operation that connects to remote hosts. It also has methods that allows specifying some socket options.

4.2.3 `AServerSocket`

This a server socket implementation. It allows traditional operations of binding to specific host with `bind()` methods and accepting connections with the `accept()` method.

Note that `close()` operation currently does not interrupt `accept()` operations. This will be fixed in the next version.

4.2.4 Usage of Thread-based implementation

Thread-based implementation of sockets should be used only when NIO API is not supported by the runtime. It is slower than NIO based implementation and it does not scale well. However it could work for applications that have only few simultaneous connections. For example, network clients deployed on Foundation Profile.

This implementation uses a `ExecutorRunner` to implement socket operations over blocking `java.net.Socket` and `java.net.ServerSocket` API. So each socket has a potential to occupy two threads when reading or writing data.

This API will be likely removed in the future versions of the framework if migration to Java 5 happens.

4.2.5 Usage of NIO

NIO Sockets are used by creating `NIOSelectorVat` and `NIOSelectorRunner` first. `NIOSelectorVat` and `NIOSelectorRunner` are a vat and a runner that is intended to support creation of sockets over non-blocking NIO sockets. And NIO-based implementation of the sockets API could be created only in context of this `NIOSelectorVat`. They use additional services provided by the runner and the vat.

When no events available, this runner waits on NIO `Selector` object for events associated with currently active operations on NIO sockets.

If new event arrives from outside, the `select()` operation on `Selector` object is interrupted with `wakeup()` method. The `wakeup()` method is slow on some architectures, so the vat works best when protocol events are handled within a NIO vat as well.

The `Selector` is also polled every 64 turns for available events. This is done to enforce resuming of IO operations even if there are too much unrelated events queued in this vat.

The components that do not have dependency on NIO also can work freely in this vat. However this vat is usually less efficient than `SingleThreadRunner` since interrupting selector (that happens when event is enqueued to the vat) is relatively expensive operation.

```
public class NIOSelectorRunnerSample1 {
  public static void main(String[] args) {
    final NIOSelectorRunner runner = new NIOSelectorRunner("main", false);
    final NIOSelectorVat vat = runner.newVat("NIO vat") ;
    Runnable r = new Runnable() {
      public void run() {
        System.out.println("Inside the vat");
        runner.stop();
      }
    };
    vat.enqueue(r);
    System.out.println("Before starting runner.");
    runner.startInCurrentThread();
    System.out.println("Runner finished.");
  }
}
```

The example prints the following:

```
Before starting vat.
Inside the vat
Vat finished.
```

As it could be seen from example, it behaves exactly like a `SingleThreadRunner`. Note that `NIOSelectorVat` can be created only in attached state. It also cannot be detached from the runner. This limitation might be removed in the future versions of the framework.

Inside `NIOSelectorVat`, a `NIOSocketFactory` should be created. Than it is possible to work with, like with usual socket factory.

5 JUnit Support

Asynchronous components should be tested just like any other software components. Along with framework a `AsyncTestCase` class is provided that should be extended in order to write own tests. The class is different from normal JUnit `TestCase` that all `test*()` methods return a promise instead of void. A test case is considered finished when promise is resolved.

Currently running tests in parallel is not supported by the framework. This feature is planned for future versions of the framework.

6 Future Plans

Minor changes in API can happen from time to time. The framework is at its alpha phase and no backward compatibility between versions can be assumed. With 0.3.2 release, the kernel can be assumed to be stable. However, utilities and IO library will likely change basing on feedback.

The framework will eventually get IoC container to simplify configuration of the components. Likely it will look like Google's Guice, however other options are under investigation.

There is also work in progress on client and server support for TLS, HTTP, and BEEP. The HTTP and BEEP protocols are being implemented from ground up. The TLS implementation uses Java 5 `SSLEngine`. Because this does not yet work, the code is not included in the current release and exists only in the SVN repository

Possibility of using Tomcat 6 Comet API for asynchronous servlets is also being investigated.

The frameworks needs some optimization. But before this, there should be performance tests created.

However these developments will likely happen soon only if framework will be actually used by some project.

Also Java 7 has two interesting planned features that could affect the framework: Closures and NIO.2 (JSR 203). Closures theoretically could lead to even simpler interfaces with less frameworks clutter. NIO.2 will simplify network IO implementation and will allow creation of asynchronous File IO API.

7 Important Implementation Issues

7.1 Code Verbosity

The code required for the framework usage is quite verbose if created manually. However modern Java IDE-s like Eclipse can greatly simplify creating code through using quick fix and code assists features. Therefore a big portion of this code is created for you automatically by IDE.

7.2 Concurrency

It is important that the vat is a unit of concurrency. Vats can migrate between different threads, however each vat occupies exactly one thread. And events sent to the objects that belong to the vat will be dispatched in the order that they have been received. If objects are used only though proxies,

no two objects from the same vat will have a concurrent method execution.

These principles relieve developers from worrying about synchronization issues in many cases. One should worry about synchronization only when object is passed to other vats. Inside a single vat, objects could be thought as being executed in a single thread.

The framework IO classes mostly uses immutable objects for communication[1]. And when during design of new classes, it is suggested to stick to immutable data types as well.

If it is required to give a mutable object to other vat, there are several approaches possible.

- Object might be treated as a token. The object is given to asynchronous method and it is not used by request sender until a promise returned from asynchronous method is resolved. `RequestQueue` might be used to ensure that no parallel requests that might need this object will start in meantime. The invoked object will stop using the passed object right after it finishes the operation. When a message is sent and dispatched to object, a synchronization over monitor associated with event queue happens. So one does not have use synchronization to ensure that write and read barriers happened.
- Normal Java synchronization can be used. However in this case it should be done very carefully. All operations inside synchronized block should be short in order not to block a progress in the vats.

7.3 Garbage Collection

The framework was created in the way that it does not prevent garbage collection of the classes and class loaders. Classes are mostly weakly referenced from framework.

7.4 Tail Recursion

Without compiler support, proper tail recursion is difficult to implement. It is recommended to avoid long chains of promises if possible since they will be collapsed only when they start to be resolved. However the recursion is the only available way to create asynchronous loop.

A promise shortening mechanism will be considered for the framework later.

It is recommended to organize such loops using `AsyncProcess` class rather than by using promises.

7.5 Using of `Throwable`

The interfaces uses `Throwable` in a lot of places. This happens because otherwise Errors might be thrown from operations and a stray error could interrupt a vat without giving a notice to the user. So errors is dispatched back smashing promises in the invocation chain instead of stopping the current vat.

1 Things are a bit complex in Java. In Java truly immutable objects are impossible. Even java.lang.String has an associated monitor and an object monitor is a mutable state. For example, is relatively easy to pass information form one applet to another using monitors on interned strings even if applets are in different class loaders on the same JVM.

7.6 Interfacing with Synchronous API

Most of currently available Java API are synchronous. And it is required to work with it from asynchronous code.

Since the most of the current API blocks when it needs some data, it should be invoked from vats that can block without affecting behavior of other parts of the application. The recommended way is to use a vat from thread pool vat group for each component that use synchronous operations.

Other problem happens when synchronous component needs to invoke asynchronous component. If synchronous component does not need to wait until completion of asynchronous operation, it is recommended to create a proxy component with oneway methods. Oneway methods can be safely invoked from any thread. If synchronous component need to wait for outcome of asynchronous operation, it is suggested to use `AsyncAction` utility class.

GUI component frameworks (like AWT/Swing and SWT) usually have own event loops. These event loops could be treated as vats if appropriate `VatSensor` is registered with the framework. This allows GUI code that executes in the context GUI event loop to more freely work with asynchronous components.

7.7 Security

Java security model is based on stack inspection. It is incompatible with how framework works since the received message is executed in new stack context and likely in other thread. It might be possible to carry security context with each message and to reinstate it when the message is dispatched, however overheads of this action are expected to be gigantic.

It is not yet decided if the framework will ever reconcile with Java security model. It is likely that some security mechanism based on object-capability security model will be developed instead. Components in the vat will have no permissions to access sensitive resources by default. And they will be able to get it only by using capabilities.

7.8 Class loaders

Class loaders do not provide ability to define class in them for external components by default. Therefore, ASM-generated proxies are created in the shadow class loaders that more or less mirror actual relationships of runtime class loaders.

The mechanism should work for complex class loader configurations like ones present in OSGi or Java EE. However, it has not been tested yet.

8 GNU Free Documentation License

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
- C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
- D. Preserve all the copyright notices of the Document.
- E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- H. Include an unaltered copy of this License.
- I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
- N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
- O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.