

1 Examples

1.1 Planning a Party

In this section we investigate a simple wizard-like flow in Aranea. We first provide it an event-driven solution and then compare it with a blocking implementation.

1.2 Problem

Consider the following case. We have a list of locations and a list of persons. We want to plan a party in one of these locations and invite some of these persons to it. We also give the party a title and a start and end time. Finally we want to send an e-mail with an invitation to each person.

There are the following steps:

1. Setting a title
2. Choosing a location
3. Setting the start and end time
4. Choosing people
5. Sending invitations

1.3 Event-driven Solution

We create a flow container and start a new flow for each step. When starting a flow we also pass the flow container a handler. Each handler processes the returned value and starts the next flow.

The disadvantage is that we couple processing values and starting the following flows together. This complicates making changes to the actual order.

1.4 Classic Implementation

The following `Widget` implements the event-driven approach for our problem:

```
class MethodLevelClassicPartyWidget extends BaseUIWidget {  
    ...  
}
```

```

void planParty() {
    final Party party = new Party();

    final Handler peopleHandler = new Handler() {
        void execute(Object persons) {
            party.setPersons((List<Person>) persons);
            getFlowCtx().start(new InvitationsWidget(party));
        }
    };

    ...

    final Handler titleHandler = new Handler() {
        void execute(Object title) {
            party.setTitle((String) title);
            getFlowCtx().start(new LocationWidget(),
                               locationHandler);
        }
    };

    getFlowCtx().start(new TitleWidget(), titleHandler);
}
}

```

Now, let's describe the `planParty()` method. First we define a local variable `party` that will contain all the information gathered later.

Then we define new handlers as anonymous inner classes assigned to new local variables. Since our local variables are referenced from the method level inner classes, we must define them as `final`. This also enforces us to define them in strict order. As you can see, this order is exactly the opposite of how the program is executed.

Each handler has two actions. Firstly, it processes the returned value by assigning it a field of the outer method local variable `party`. Secondly, it starts the next flow by passing the flow context a new `Widget` and next `Handler` (which must be defined earlier in the outer method). At the end of the `planParty()` the first flow is started exactly the same way.

1.5 Sequential Programming Solution

Instead of passing these event handlers with each new flow it would be easier to just call a method and wait for its return value. This is exactly what blocking calls do.

1.6 Blocking Implementation

The following `Widget` implements the blocking way for our problem:

```

class BlockingPartyWidget extends BaseUIWidget {
    ...
}

```

```

@Resumable
void planParty() {
    try {
        Party party = new Party();

        party.setTitle(chooseTitle());
        party.setLocation(chooseLocation());
        party.setTime(chooseTime());
        party.setPersons(choosePersons());

        sendInvitations(party);
    } catch (FlowCancellationException e) {
        // just cancel the current flow as finishing normally
    }

    getFlowCtx().cancel();
}

String chooseTitle() {
    return (String)
        AraneaBlockingUtil.call(getFlowCtx(),
            new TitleWidget());
}
...
void sendInvitations(Party party) {
    AraneaBlockingUtil.call(getFlowCtx(),
        new InvitationsWidget(party));
}
}

```

Now we have method `planParty()` annotated as `@Resumable`. This enables us to use blocking calls by calling `AraneaBlockingUtil.call()` inside the `planParty()` and all the methods that it calls.

We create a new `Party` object, assign its fields values from the following methods, and send invitations. At last whether we succeeded or the client canceled the flow we finish the current flow returning to the previous page.

Notice that this implementation is shorter and simpler than the event-driven approach. We do not have any inner classes or final variables. Moreover we could easily change the order in which the different flows are called by just switching the corresponding lines in method `planParty()`. In short, we can concentrate on the actual program instead of complicated language structures.

The only disadvantage is that blocking calls in Java need bytecode instrumentation that makes the transformed classes larger and slower.

1.7 Click 10 Times

In this section we examine a simple Swing frame that waits until a button is clicked 10 times and

then exists. We look at an event-driven and a blocking implementation.

1.8 Event Driven Implementation

The following implementation uses the classical event-driven approach:

```
class ClassicSwingExample extends JPanel {

    void init() {
        JButton button = new JButton("Click on me!");
        add(button);

        button.addActionListener(new ActionListener() {
            int counter;
            void actionPerformed(ActionEvent e) {
                if (++counter == 10) {
                    System.exit(0);
                }
            }
        });
    }

    public static void main(String[] args) {
        ClassicSwingExample panel = new ClassicSwingExample();
        JFrame frame = new JFrame();
        frame.add(panel);
        panel.init();
    }
}
```

In method `init()` we add a button with a listener that is called when we click on the button. This listener has its own state containing the number of clicks. On each click, this is increased and when reaching to 10, the program exists.

1.9 Blocking Implementation

The following implementation uses the blocking approach:

```
class BlockingSwingExample extends JPanel {

    @Resumable
    void init() {
        JButton button = new JButton("Click on me!");
        add(button);

        for (int i = 1; i <= 10; i++) {
            SwingBlockingUtil.waitForOneAction(button);
        }
        System.exit(0);
    }

    public static void main(String[] args) {
        BlockingSwingExample panel = new BlockingSwingExample();
    }
}
```

```
        JFrame frame = new JFrame();
        frame.add(panel);
        panel.init();
    }
}
```

Instead of adding a listener we wait for 10 clicks and then exit. Calling a method `SwingBlockingUtil.waitForOneAction()` waits for one click on the specified button.

Although `SwingBlockingUtil` still uses listeners internally we have the choice whether to use them explicitly or not. Since we could not use a `for` in the first case it demonstrates that blocking calls let us use sequential programming where it was usually impossible.

2 Blocking Calls

2.1 Resumable Methods

A method is called **resumable** if it is suspendable and later resumable **by itself**. Inside a resumable method, one can use **blocking calls** to other methods suspending the first method and resuming it any time.

The effect is similar to partial continuations provided by Javaflow except that the resumable method (a `Runnable` passed to the method `Continuation.startWith()`) has control to resume itself later. This distinction makes it possible to reuse blocking calls and hide the low level API from the developer. Using the Javaflow API, we should write in addition to the each `Runnable` corresponding control logic of resuming it.

A blocking call can be executed, suspended and resumed (possibly returning a value) as follows:

```
ResumableRunnable myRunnable = new ResumableRunnable() {
    void run() {
        IndirectCallable myCallable = new IndirectCallable() {
            void call(ReturnContext returnContext) {
                ...
                returnContext.returnWith("myValue");
            }
        };
        String myVar = (String) BlockingUtil.call(myCallable);
    }
};
BlockingUtil.execute(myRunnable);
```

The actual code is simpler due to the `@Resumable` annotation that allows hiding the calls to `BlockingUtil.execute()`. Also calls to `BlockingUtil.call()` are generally placed into other methods.

In the next section, we cover the blocking calls API in details.

2.2 Blocking Calls Core API

In this section, we describe the core API of blocking calls

2.2.1 Executing Resumable Method

As we are using partial continuations from Commons Javaflow framework we have to distinct the resumable method from other parts of code.

The resumable method is represented by `ResumableRunnable` interface:

```
interface ResumableRunnable extends Serializable {
    void run();
}
```

Because the entire stack of the resumable method is tracked it must be serializable.

NB! Not only the `ResumableRunnable` interface is serializable, but each local variable in method `run()` must be serializable.

To execute a blocking method, one must call

```
ResumableRunnable runnable = new ResumableRunnable() {
    void run() {
        ...
    }
};
BlockingUtil.execute(runnable);
```

Calling `BlockingUtil.execute(ResumableRunnable)` does two things:

1. Executes the resumable method with `Continuation` tracking down its entire stack and enabling the method to suspend.
2. Makes the suspended method resumable.

In Java, the current thread executes the resumable method until it suspends or finishes – technically the method just returns. The resumable method must have a `void` return type since there is no way to return a meaningful value from a suspended method (which we still technically have to do).

2.2.2 Suspending Blocking Method

As mentioned earlier we can also suspend and resume plain Javaflow continuations, but that is not reusable because the resuming logic is contained outside the continuation itself.

By running a blocking call we do not just suspend the resumable method. We also provide a new „runnable” that is executed as the resumable method suspends. This „runnable” has the control to resume the resumable method.

In order to suspend and resume a resumable method, we must do the following:

```
IndirectCallable myCallable = new IndirectCallable() {
    void call(ReturnContext returnContext) {
        ...
        returnContext.returnWith("myValue");
    }
};
```

```
String myVar = (String) BlockingUtil.call(myCallable);
```

`BlockingUtil.call()` does the follows:

1. Suspends the resumable method.
2. Executes the provided `IndirectCallable`.
3. When `returnContext.returnWith("myValue")` is called the resumable method resumes by assigning `"myValue"` to `myVar`.

The `IndirectCallable` and `ReturnContext` have the following methods.

```
interface IndirectCallable {
    void call(ReturnContext returnContext);
}
interface ReturnContext {
    void returnWith(Object result);
    void failWith(Throwable t);
}
```

The `IndirectCallable` is just an abstraction of a usual method that has the explicit control over returning a value or throwing an exception. This is the so-called Continuation-passing style [Ap07].

This explicitness enables us to use the `ReturnContext` from an arbitrary method, usually an event listener, which is a powerful way to hide the event-based programming. In fact the `ReturnContext` can be made to resume the blocking method more than once, providing new opportunities to our usual programming style.

But the most important thing is that blocking calls can be reused forming new APIs that we cover in the following sections.

2.3 Swing Blocking API

For Swing library (see the tutorial [ST]), for the moment following methods are provided for suspending a resumable method:

```
class SwingBlockingUtil {
    static void waitForAnyAction(AbstractButton button)
    static void waitForOneAction(AbstractButton button)
    static void waitForAnyChange(AbstractButton button)
    static void waitForOneChange(AbstractButton button)
}
```

`waitFor*Action()` and `waitFor*Change()` provide the `Button` with an `ActionListener` or `ChangeListener` respectively.

“Any” corresponds to the fact that the resumable method is resumed from this point each time the respective listener is called. In case of “One” the listener is called only once.

The usual method that abstracts the event listening looks the following:

```
static void waitForAnyAction(AbstractButton button) {
    BlockingUtil.call(new IndirectCallable() {
        void call(ReturnContext returnCtx) {
            ActionListener listener = new ActionListener() {
                void actionPerformed(ActionEvent e) {
                    returnCtx.returnWith(null);
                }
            };
            button.addActionListener(listener);
        }
    });
}
```

The resumable method is suspended by calling `BlockingUtil.call()` and is resumed if the `ActionListener` added to the specified `Button` is reached. The main fact here is that the `ReturnContext` is passed to another method.

There is nothing magical about suspending and new methods can be introduced just as easily.

2.4 Aranea Blocking API

For Aranea Web Framework, currently the following methods are provided for suspending a resumable method.

```
class AraneaBlockingUtil {

    static Object call(FlowContext flowCtx,
                      Widget flow,
                      Configurator configurator) throws FlowCancelException

    static void wait(BaseApplicationWidget widget, String eventId)

    static void waitForClick(ButtonControl control)
    static void waitForChange(DateTimeControl control)
    static void waitForChange(StringArrayRequestControl control)

}
```

These methods do the follows:

- The method `call()` calls the `FlowContext.start()` and returns when the subflow ends execution. If the subflow is cancelled, a `FlowCancelException` is thrown.
- `wait()` adds an `EventListener` to the `Widget` and returns when it is called.
- `waitForClick()` adds an `OnClickEventListener` to the `Control` and returns when it is called.
- `waitForChange()` adds an `OnChangeEventListener` to the `Control` and returns when it is called.

2.5 Blocking Calls Implementation

Blocking calls are based on Commons Javaflow continuations. In addition the implementation is similar to that in two ways.

1. There is only one class that should be accessed from outside (`BlockingUtil`).
2. There is a thread-local class (`BlockingHelper`) for internal use.

`BlockingUtil` just gets a thread local instance of the `BlockingHelper` and proxies the call to method with the same name.

The `BlockingHelper` is based on Javaflow API as follows:

```
class BlockingHelper {
    Continuator continuator;

    void execute(BlockingRunnable runnable) {
        // Create the initial Continuation
        Continuation c = Continuation.startSuspendedWith(runnable);
        // Start it
        continueWith(c, null);
    }

    void continueWith(Continuation initContinuation, Object context)
        // Continue Continuation
        Continuation newContinuation =
            Continuation.continueWith(initContinuation, context);
        // Continuation finished or suspended
        if (newContinuation == null) {
            // Continuation finished
            return;
        }
        // Continuation suspended
        // Start Continuator and provide it with SuspendContext
        this.continuator.run(new Continuator.SuspendContext() {
            void doContinue(Object context) {
                // Continue the current Continuation
                continueWith(newContinuation, context);
            }
        });
    }

    Object call(IndirectCallable callable) {
        // Define continuator
        this.continuator = new Continuator() {
            void run(SuspendContext ctx) {
                callable.call(new ReturnContext() {
                    void returnWith(Object result) {
                        Object wrapper = new Object[]
                            {result};
                        ctx.doContinue(wrapper);
                    }
                    void failWith(Throwable t) {
                        ctx.doContinue(t);
                    }
                });
            }
        });
    }
}
```

```

};
// Let's sleep!
Continuation.suspend();
// Get context!
Object ctx = Continuation.getContext();
if (ctx instanceof Object[]) {
    // Unwrap and return value
    return ((Object[]) ctx)[0];
}
// Uncheck and throw exception
throw ExceptionUtil.uncheck((Throwable) ctx);
}
}

```

Let's examine each field and method:

- `execute()` is called only the first time resumable method is started. It creates the new `Continuation`, doesn't start it, passes it the `BlockingRunnable` to and calls `continueWith()`.
- `continueWith()` is called as executing a blocking method first time as continuing it later. This can be seen as a wrapper method for `Continuation.continueWith()`. If the latter returns `null`, resumable method has finished and this method returns also. Otherwise `this.continuator` provided by the last call to the method `call()` is run. It is provided with `SuspendContext` that calls the `continueWith()` method recursively if the method `SuspendContext.ReturnContext.doContinue()` is called.
- `call()` suspends an already running resumable method performing a blocking call. This can be seen as a wrapper to `Continuation.suspend()`. Before suspending the continuation, `this.continuator` is provided with a new `Continuator` that wraps the `IndirectCallable` provided to the method `call()`. After the current resumable method is resumed the data is read back and value is returned or an exception is thrown respectively.

Altogether blocking calls improve Commons Javaflow continuations with two features:

1. A continuation can suspend providing a callback that is able to resume this continuation.
2. Suspending the continuation has the syntax of a normal method call (returning values as well as throwing exceptions are supported).

In the following sections we show how the `BlockingUtil.execute` can be hidden also.

2.6 @Resumable Annotation

Instead of the following syntax:

```

void myMethod() {
    BlockingUtil.execute(new ResumableRunnable() {

```

```
        void run() {
            ...
        }
    });
}
```

In Java 5 and later, there is a much simpler approach:

```
@Resumable
void myMethod() {
    ...
}
```

A method that is marked `@Resumable` must have return type `void`. In the next section we show how this annotation works.

2.7 Bytecode Instrumentation

In this section we describe bytecode transformation used in blocking calls.

2.7.1 Blocking Agent

Blocking calls contain a Java instrumentation agent that adds two class file transformers.

1. Commons Javaflow transformer (described on page).
2. Resumable transformer.

These transformers are filtered so that only packages set in the agent options are included. Otherwise the instrumentation would take very long as well as cause validation errors due to different Java versions, incorrect order of class loading etc.

2.7.2 Blocking Transformer

As we described resumable behavior can be achieved by calling `BlockingUtil.execute()`. Resumable transformer enables us to use the `@Resumable` annotation which simplifies the programming letting the code to remain clear of the continuation or blocking-specific API.

The resumable transformer is based on the ASM Java bytecode manipulation framework. In addition to transforming existing classes, it also generates new ones and defines them as described on page .

Blocking transformer does the following:

1. Finds a method marked with the `@Resumable` annotation;
2. Removes the annotation (to not transform any class twice);
3. Examines the method marked `@Resumable` and adds to this class a new public accessor

method for each non-public field and method;

4. Creates a new inner class that implements `ResumableRunnable`. This class has a constructor that takes all arguments of the method marked `@Resumable` as well as the instance of the outer class. All of these arguments are assigned to the corresponding generated instance fields. The `run()` method contains the original resumable method body. Each access to a method argument is transformed to an access to an inner class instance field. Each access to a non-public field or method is transformed to an access to an accessor method;
5. Replaces the blocking method body with bytecode creating a new instance of the generated inner class and passing it to the `BlockingUtil.execute()` call.

As the method marked `@Resumable` must correspond to the

```
void ResumableRunnable.run()
```

it must have a `void` return type.

3 Conclusions

The main goal of this work was to implement blocking calls in Java using an existing library – Commons Javaflow – for continuations in Java.

We implemented the blocking calls core API, provided some Aranea and Swing dependent blocking calls and also implemented a Java bytecode instrumentation agent that enables us to use the `@Resumable` annotation.

In Aranea framework, the blocking calls let us call a flow as calling a normal Java method. The method with a `@Resumable` annotation could suspend and later be continued, so outliving the usual request-response cycle.

We showed an example of combining different flows and how this can be implemented in an event-based way and by a blocking approach. The latter made it possible to uncouple the dependencies that were otherwise hardly avoidable. Therefore, we could easily change the order of flow calls or reuse them. In a word, the code was easier to read and maintain.

We also showed an example of how the blocking approach enables us to use a for-cycle where it was else impossible.

Altogether blocking approach does not replace the event-based programming, but hides the low-level part and lets us express the application logic using higher-level approach. We can still use the event-based way where appropriate. At the same combining event-based and sequential programming together is a very powerful idea because different parts of application logic can be expressed using the most suitable tool.