AvalonNoLogging

No Logging

Most server-side APIs define a method for logging. This enables the component - a reusable piece of code - to output log entries in a unified way, and enables the user to view the workings of the system, find original causes for failures, measure system performance and use and so on.

However, we argue that while logging is very useful, there are use cases where a callback interface is preferrable.

Basic Concepts

Why log? Above we list a set of use for logging:

- View the workings of a system
- Find causes for errors
- Measure system performance and use

Of these three uses for logging, the two latter basically boils down to the first one - we want to see what is going on inside this piece of code that we have here. For that, logging is but one of the methods that can be used. We have also:

- Exceptions - whenever a file could not be opened for reading, the constructor throws a FileNotFoundException. We can then test the type of exception and figure out that the constructor failed due to the file not existing.
- Return values - File.lastModified() will return 0 if the file doesn't exist or if an IO error occurs.
- Inspector methods - Thread.isAlive() returns true if the thread is still running.
- Callbacks - see the Observer pattern, exemplified via SAX ErrorHandlers.

Logging for Whom?

You may note that all the above schemes of finding out what is going on inside a piece of code are only available in program code itself. You cannot as a user, sitting in front of a machine, analyze return values or call inspector methods (unless you have connected a debugger, of course).

This brings us to our first assertion: **Logging is primarily for humans**

Sure, there are systems that produce performance monitoring logs, and write-ahead-logging is a well known technique, but as commonly discussed, *logging is all about outputting textual messages for a user to read*.

Log entries are Stringified Callbacks

This brings us to our second assertion: **Callbacks are a superset of logging**

If you have a piece of code, and you have some log statements in it:

```
 FileInputStream input = new FileInputStream( myFile );
getLogger().debug( "Opened " + myFile.getPath() + " successfully.");

... do stuff ...

input.close();
getLogger().debug( "Closed " + myFile.getPath() + ".");
```

You can replace that with a listener that you call at the same points:

```
 FileInputStream input = new FileInputStream( myFile );
listener.onOpen( myFile );

... do stuff ...

input.close();
listener.onClose( myFile );
```

The listener can then output the name of the method being called, and the arguments to it. Usually, instead of outputting

```
 onOpen,/home/user/myfile
```

, it is more user friendly to output

```
Opened /home/user/myfile
```

- as we said, logging is for humans.

When to Log and When not to Log

Well, you say, having callbacks are fine with **one** component that has a callback with few methods. But surely you don't expect me to write one callback for each library of code I use - worse, for each component in those libraries?

Of course not! At some point the system becomes too big, and you should use logging. In that case, you can easily have a callback that outputs log entries. Preferrably, the component should come with its own callback-to-logging listener. The criteria to use when determining whether logging is right or wrong is the size of the component and the environment it will be deployed in. We hold that a component should be fairly large - it should be a "first class citizen" in the environment. By that we mean this, and let us illustrate by example: An EJB session bean is a first class citizen in an EJB container. A servlet is a first class citizen in a servlet container. An Avalon component is a fcc in its container. **An instance of a String class is a first class citizen in a testcase for itself, but not in a servlet container.**

Look at the last statement. What if logging had been implemented in the java.lang.String class. You open up your server logs and see this:

```
 [[DEBUG]] Creating new instance of String class.
["DEBUG"] String.charAt() called with index 0
["DEBUG"] String.charAt() called with index 1
["DEBUG"] String.charAt() called with index 2
["DEBUG"] String.charAt() called with index 3
["DEBUG"] String.charAt() called with index 4
["DEBUG"] String.charAt() called with index 5
["DEBUG"] String.charAt() called with index 6
["DEBUG"] String.charAt() called with index 7
["ERROR"] String.charAt() called with invalid index 8
["DEBUG"] Throwing ArrayIndexOutOfBoundsException
```

Imagine the size of that log file. Big doesn't adequately describe it. However, if you are testing your new String class, you may well find it useful to output debugging information. The issue here is that you have logging at too fine a granularity. This causes not only headaches in terms of log file size, but also of API concerns: How do you provide a logger to a String? Will you have to pass it in a constructor?

```
String myString = new String( "myText", myLogger );
```

Or will all strings get a logger via static accessor methods, and log to the same category?

```
public final class String {

   private final static stringLogger = Logger.getLogger( "java.lang.String" );

   public String( char[] text ) {
       stringLogger.debug( "Creating new instance of String." );
       ...
   }

   ...
}
```

If the latter, are you sure that'll always be the right thing to do?

The question posed at the end of the last section touches upon another reason not to use logging: Unless you know a lot of the environment your code will be used in, you can't really say whether logging is appropriate: **Logging Assumes Knowledge of the Environment**

In particular, it assumes that there is a **human** that find your information **relevant**.

Callbacks Solve This

Instead, we believe that callbacks should be used when you want maximum portability. Use of callbacks enable you to:

- Log the events, if that is what you want. We think including a callback-to-logger adapter is a really great idea.
- No penalty in performance, scalability.
- Easy to integrate and embed - we make the least amount of assumptions on what the deployment environment looks like.
- We can easily provide standard log adapters for common logging toolkits (i.e. nothing is lost).

- It enables a reusable component to communicate with the code it is embedded in in a standard Java way. With logging, you have to parse the log files for the same effect. This is a requirement for embeddable components. If a component lacks features for integration with the container, it is a much worse component than one that has such features.

In short - **this solution gives maximum flexibility, maximum embeddability, for no cost.**

---- Comment by Ceki Gülcü ----

I am intrigued by your article. Would it be possible to show an example of what you have in mind with sample code? Thanking you in advance.