

Extensions to JavaScript Exception Handling

This document describes TIBCO General Interface extensions to JavaScript exception handling and how to leverage these extensions to write code that is more robust and easier to maintain.

Version 2.0: 02-10-2006



<http://www.tibco.com>

Global Headquarters

3303 Hillview Avenue
Palo Alto, CA 94304
Tel: +1 650-846-1000
Toll Free: 1 800-420-8450
Fax: +1 650-846-1005

© 2006, TIBCO Software Inc. All rights reserved. TIBCO, the TIBCO logo, The Power of Now, and TIBCO Software are trademarks or registered trademarks of TIBCO Software Inc. in the United States and/or other countries. All other product and company names and marks mentioned in this document are the property of their respective owners and are mentioned for identification purposes only.

Table of Contents

Exceptions in JavaScript	3
Exceptions in GI	3
Exceptions and the Call Stack	4
When to Use Exceptions.....	4

Exceptions in JavaScript

Like many other programming languages, JavaScript provides for throwing and catching exceptions. JavaScript supports the standard try...catch...finally statement:

```
try {
    doSomething();
} catch (e) {
    window.alert(e.description);
} finally {
    cleanUp();
}
```

JavaScript also supports the throw statement. Any type of object can be thrown:

```
throw "error!";
throw {name:"anError", description:"an error occurred"};
```

The browser creates and throws errors under certain circumstances. For example, trying to access a field on an undefined object will raise an error. Trying to call a function that does not exist also raises an error. These types of exceptions can also be caught with the try...catch statement.

When an exception is raised and is not caught, the current call stack is unwound and the browser receives an error event. Execution of JavaScript code continues when the next stack is created by a timeout event or by user interaction.

Exceptions in GI

General Interface extends the native JavaScript exception facilities in several ways that make it easier to build and debug large applications.

General Interface defines a class, `jsx3.lang.Exception`, which extends `jsx3.lang.Object` and is the class of which all types of exceptions descend. This base class is stack aware, meaning that it can communicate the call stack at the point where it was created. By using and/or extending `jsx3.lang.Exception`, a developer can take advantage of some features of exceptions that exist in more advanced languages such as Java.

Several methods in the General Interface framework throw exceptions to communicate to the caller that an error has occurred. (Some classes, such as `jsx3.xml.Document` store error information in instance fields rather than throwing exceptions.) If a framework method throws an exception it is documented in the API Documentation. Such methods should be surrounded by a try...catch block to prevent the exception from reaching the top of the call stack. Note that exceptions in JavaScript are not "checked" (as they are in Java) so it is not a compilation error not to surround a method that throws an exception with a try...catch block.

General Interface also defines a subclass of `jsx3.lang.Exception`, `jsx3.lang.NativeError`, which wraps the native browser exception object. `NativeError` is a cross-platform interface onto the exceptions that the browser may raise. Since `NativeError` extends `Exception`, application code can treat all exceptions caught in a `try...catch` block as instances of the `Exception` class. This is accomplished with the following code:

```
try {
    doSomething();
} catch (e) {
    // e may be either a native browser Error object, or an instance of
    // jsx3.lang.Exception thrown by application or framework code
    var ex = jsx3.lang.NativeError.wrap(e);
    window.alert(ex.printStackTrace());
}
```

Finally, in GI Builder and when error trapping is enabled in running GI applications, any error that reaches the top of the stack and the browser is routed to the GI logging system. The exception will be sent to the global logger (`jsx3.util.Logger.GLOBAL`) with severity `ERROR`.

Exceptions and the Call Stack

Any time an instance of `jsx3.lang.Exception` is created, it stores the current call stack. This can be accessed with either the `getStack()` or `printStackTrace()` methods of the class. This feature can be very helpful when diagnosing a problem in an application.

Unfortunately, in Internet Explorer, the native exception class does not store stack information. Therefore, an instance of `jsx3.lang.NativeError` only contains the stack up to the point when it was created, not up to the point where the error it wraps was raised.

The one exception to this is when a native error reaches the top of the call stack. In that particular case the GI logging system does have access to the entire stack up to the function where the error was raised. This is problematic from the point of view of a developer, however, because the choice is between maintaining the stack by catching an error but losing information about how that error was created and seeing the full stack trace of the error but not being able to recover from the error.

One possible development practice to deal with this shortcoming of Internet Explorer is to include fewer `try...catch` statements in the early stages of developing an application and including more as the quality of the code improves.

When to Use Exceptions

Exceptions are a way of communicating an exception or error condition from a method to the caller of the method. They are favored over the older technique of returning a status code from a method (i.e. 0 for success, 1 for error) because they cannot be ignored by the calling code. Without exceptions it is easy to write code that assumes every operation is successful. When an operation doesn't end in success, the error can show up in subsequent operations, and therefore be harder to diagnose.

In most implementations, throwing exceptions is a relatively expensive operation. Therefore, exceptions should be used to communicate truly exceptional conditions rather than conditions that may occur under the normal execution of a program.

The benefit of using exceptions usually increases with the size of the application. Exceptions require the developer to define the success and error conditions of methods and require the calling code to handle both possibilities. Code therefore can have fewer bugs and be more robust and clear than code that uses another error reporting mechanism or

Exceptions in General Interface

ignores error conditions altogether. Additionally, exceptions help with the principle of failing early. In general it is better to fail (raise an exception) at the first sign of trouble when the cause of the failure is well known. Otherwise errors might cascade to other parts of a program where they will be hardy to diagnose.

