**Listing 1 Blocking Calls**

```
Try
{
  myConnection.setAutoCommit(false);
  GetData.addBatch("INSERT into
    Products (1, "Watchband"));
  GetData.addBatch("INSERT into
    Products (1, "Bracelet"));
  GetData.addBatch("INSERT into
    Products (1, "Ring"));
  GetData.executeBatch( );
  myConnection.commit( );
  myConnection.setAutoCommit(true);
}
```

The try block is one way to test your code in Java.

Possibly the most interesting aspect of both of these classes is that you can derive your own error-handling objects from them. Deriving your own objects from the standard error classes has the advantage of enabling you to fine-tune your error-handling routines. The goal in incorporating error handling into JavaServer Pages (JSP) applications isn't to deliver error-free code; it may be unrealistic to do so, and the cost in any event may be prohibitive. Instead, the goal is to never have the user adversely affected by an error. A user should never be frustrated enough to hesitate to use the application. That is a difficult, if not impossible, goal to achieve, but is worthwhile to try.

What kind of information is contained in these objects? In general, you'll get what was being executed at the time the error was encountered—typically, calls to Java methods or other types of actions. You'll usually get a brief description of the error and the routines executed leading up to the error.

## Try and Catch It

Java uses very descriptive terminology in its error-handling activities. Your code attempts a particular operation. If it succeeds, all is well and good. If it fails, an error object is created and passed up the object hierarchy of your application until you catch it with an error handler. Java refers to this as "try-catch."

These keywords are exceptionally easy to use. Simply encapsulate any code in which you want to handle an error with a try block:

```
Try
{
  Insert your application code here
}
```

For example, if you're making a database call, the try block might look something like Listing 1.

You can be reasonably certain that any error encountered within this block has to do with being able to write into the database, and you should be able to create an error handler that can respond gracefully to that error. If an exception is thrown from within a try block, it is handled with code encapsulated in a catch block.

```
Catch(RuntimeException e)
{
  Insert your error handling
  code here
}
```

Because Java is an object-oriented language, you have to be concerned about the scope of your variables. In particular, local variables within the try block won't be available outside of that block, so you can't expect to be able to access information in these variables unless you take steps within your try block to make them available.

A catch block will always process the error that it was designed to catch. In our catch block code snippet, we catch the most general type of error, the RuntimeException, but you can also catch objects derived from this class. This means you can write a catch for each type of error that could occur on the page. The catch can cause a page to display a customized error message, or perform specific actions in code, based on the intent of the user's actions on the page.

## Paging an Error

In the event of an error, most likely you would want to redirect the page request to an error page, either to the user or the system administrator. This can be a part of your catch, or in lieu of a catch. You can redirect to another JSP page, or to an HTML page, or other type of page. When an error occurs on a JSP page, the server stops processing the page and sends an error message to the user's browser. This error message typically contains an error code and a brief technical description of the type of error. In other words, it's not information that is useful to the user. Instead, it makes more sense for the application to send the user automatically to a page that's more useful, or at least more friendly.

To redirect an error to another page, you use the errorPage attribute of the page directive. This attribute lets you submit the URL of your error page, and the Web server will process this page directive whenever it encounters an error on that page:

```
<%@ page errorPage="myerror.jsp" %>
```

To prevent the buffer from being flushed before the page directive is executed, you might also execute the autoFlush attribute of the page directive. Not doing this could cause another error in conjunction with the first one:

```
<%@ page autoFlush="false" %>
```

Your error page can be static or dynamic, although most developers take the conservative approach and keep their error pages in straight HTML. Often Web application developers send users to a page that says the Webmaster has been notified of the error and may also ask the user to fill in a short form on their activities leading up to the error. From the standpoint of the user, I find this method of handling an error to be annoying and unhelpful. I don't know what type of error response is the right one, but it's worth the time and effort for you to find it for your application.

The important fact to remember when planning your error-handling strategy is that not all errors are the fault of your code, and not all errors are fatal or even visible to the user. You have to accept that you could write error-free code and still have to handle errors.

There's more to error handling than simply logging the error and notifying the user. Next month, I'll examine topics such as catching multiple errors and creating a finally block—a section of code that you can use to clean up from any errors. *JP*

**GO ONLINE**
www.javapro.com

Use these Locator+ codes at www.javapro.com to go directly to these related resources.

**JP0202** Download all of the code for this issue of *Java Pro*.

**JP0202WC** Download the code for this article separately.

**JP0202WC_T** Read this article online.