org.jax.mgi.shr.exception Design Document

Author: Mike Walker

Created: March 19, 2003

Last Modified: September 17, 2004 00:16

1 Purpose of Document

This document describes the classes belonging to the org.jax.mgi.shr.exception package and provides source code examples for common usage patterns.

2 Introduction

The purpose of these classes are to provide base class support for defining new exceptions and exception factory classes. The base classes of this package and three distinct usage patterns will be discussed in this document.

These classes were written and tested with java1.4.

3 Overview of Classes

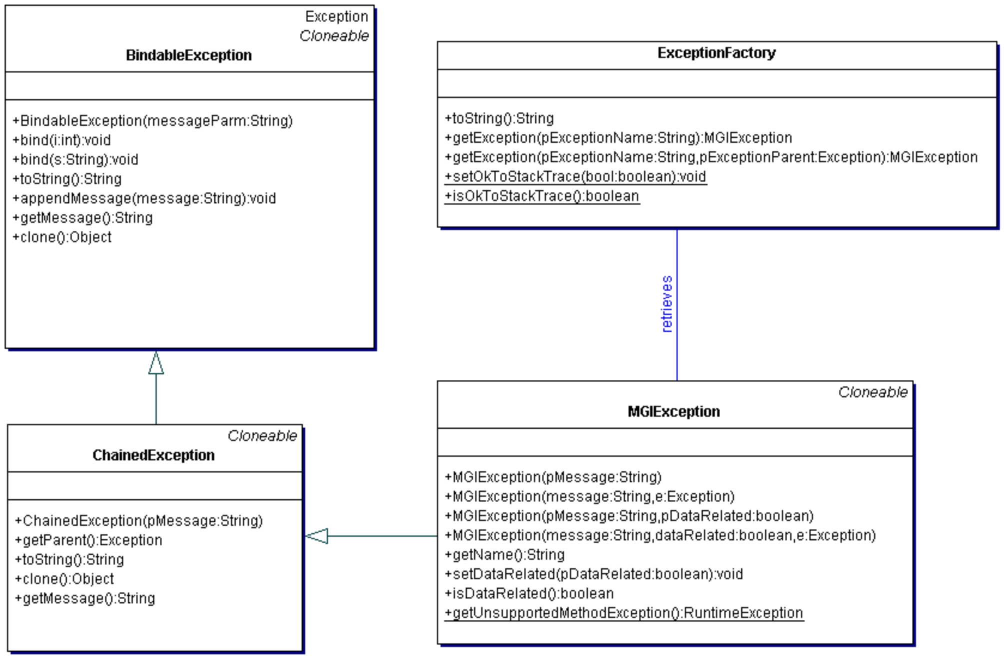
The **BindableException** class extends Exception and stores an error message that may contain a number of variable strings (designated by the '??' character pair) which are substituted at runtime with the actual runtime values.

The **ChainedException** class extends **BindableException** and adds the functionality to chain exceptions together in a hierarchy so that an exception may have a parent exception. The parent exception represents the exception which caused the child exception to be raised and so on down the chain.

The **MGIException** extends **ChainedException** and adds functionality for storing whether or not an exception is data related and for accessing this characteristic. **MGIExceptions** have been named as such because they represent the base class for all exceptions created within java applications for all MGI projects. They are returned from calls to getException() on the **ExceptionFactory** class.

The **ExceptionFactory** class is a base class that provides static storage and lookup methods for named **MGIExceptions**. By storing all exceptions in an **ExceptionFactory**, a single repository can be utilized for standardizing message content and reusing exception messages. The storage is static and common to all subclasses. Each subclass is expected to define its own named exceptions particular to it's domain of interest (derived from **MGIException**) and store them in the base class storage cache using their names as keys.

4 Class Diagram


Exception
Cloneable
BindableException
+BindableException(messageParm:String)
+bind(i:int):void
+bind(s:String):void
+toString():String
+appendMessage(message:String):void
+getMessage():String
+clone():Object
ExceptionFactory
+toString():String
+getException(pExceptionName:String):MGIException
+getException(pExceptionName:String,pExceptionParent:Exception):MGIException
+setOkToStackTrace(bool:boolean):void
+isOkToStackTrace():boolean
retrieves
Cloneable
ChainedException
+ChainedException(pMessage:String)
+getParent():Exception
+toString():String
+clone():Object
+getMessage():String
Cloneable
MGIException
+MGIException(pMessage:String)
+MGIException(message:String,e:Exception)
+MGIException(pMessage:String,pDataRelated:boolean)
+MGIException(message:String,dataRelated:boolean,e:Exception)
+getName():String
+setDataRelated(pDataRelated:boolean):void
+isDataRelated():boolean
+getUnsupportedMethodException():RuntimeException

5 Subclassing MGIException

An **MGIException** is intended to be used for all newly defined exceptions within in-house java applications. It provides functionality for having parameterized message strings which can be used to bind values to them at runtime, and provides the ability to chain exceptions together in hierarchies. Furthermore, by using **ExceptionFactories**, a common static storage area is made available for them.

This section discusses the creation of subclasses of **MGIException** objects without the use of **ExceptionFactories** and also the creation of "on-the-fly" **MGIException** objects. The next section discusses how to use **ExceptionFactories**

To create a new exception class, define a class which extends **MGIException** and provide the message string to its constructor. If the message requires runtime binding, then provide methods which will perform the binding. These binding methods provide a public interface that make it obvious what parameters need to be bound. They internally will call the base class bind() methods. See Example 1 which was taken from the org.jax.mgi.shr.ioutils package. It is thrown if a badly formatted record is encountered while passing input records.

EXAMPLE 1. subclassing MGIException

```
public class RecordFormatException extends MGIException {
    public RecordFormatException {
      // define the parameterized message string and indicate that
      // this exception is data related
      super("Unexpected record format for the following record: ??",
             true);
    }
    public void bindRecordString(String s) {
      bind(s);
    }
  }
```

To create and raise an **MGIException** on-the-fly without first creating a new class or without first storing predefined exceptions with an **ExceptionFactory**, then simply use the **MGIException** constructor directly and throw the instance. If the exception is being raised within a catch block, then you may want to call the setParent(Exception) method and provide the caught exception instance to the newly created exception instance.

6 Predefined Exceptions and Exception Factories

Another common use of this library product is to provide a mechanism for creating, storing and accessing predefined exceptions within exception factories. To create a new exception class, define a new class which extends MGIException and provide a constructor which calls the super-class constructor (See Example 2).

EXAMPLE 2. defining a new exception class

```
public class BCPException extends MGIException {

   public BCPException(String message, boolean dataRelated) {
      super(message, dataRelated);
   }
}
```

An **MGIException** takes a message and a boolean for construction parameters. The message is a String which can contain special characters (the '??' character string) for binding values at runtime. Each instance of **MGIException**, or more accurately, each instance of any subclass of **MGIException** has a predefined message associated with it. The bind(String) and bind(int) methods are used for binding runtime values to this message.

The **ExceptionFactory** is used for storing named instances of these exceptions and providing access to them. After creating a new exception class, you would then create an associated **ExceptionFactory**. This class is used to define names for each instance of an exception within the class and store instances within the base class static hashmap (See example 3 which is taken from the org.jax.mgi.shr.dbutils.bcp package).

EXAMPLE 3. defining a new exception factory

```
public class BCPExceptionFactory extends ExceptionFactory {

  public static final String TempFileCreateErr =
     "org.jax.mgi.shr.shrdbutils.TempFileCreateErr";
```

```
    static {
      exceptionsMap.put(TempFileCreateErr, new BCPException(
          "Could not create a temp file for bcp", false));
    }

    public static final String InterruptErr =
        "org.jax.mgi.shr.shrdbutils.InterruptErr";
    static {
      exceptionsMap.put(InterruptErr, new BCPException(
          "BCP process was interrupted while running " +
          "the following bcp command: ??", false));
    }
  }
```

In Example 2 there is a public static final String definition which defines a constant name to an exception instance which can be used as a key for storage. Since all exception factories inherit from ExceptionFactory and that class has the static instance of the hashmap, then all exception factories are using the same storage space. The instance names are fully qualified to prevent name clashing within the storage space. This approach was designed as an alternative to using the database as a store for named exceptions and their message content. The instance names are fully qualified to prevent name clashing within the storage space. Refering to example 3, after the constant name has been defined, a new instance is created and is subsequently put into storage. Applications can then access this instance by calling the getException(String) method on the ExecptionFactory class and passing in the predefined name.

To limit their scope, these exception factory classes are used by a group of related classes, typically belonging to the same java package. Any class which calls the getException() method must have knowledge about these internal exception names and about the bind variables that each message expects. Typically these factories are used by frameworks classes which are interested in predefining all possible exceptions and their messages in one place and then accessing them by name at runtime. Example 4 is taken from the BCPManager class from the org.jax.mgi.shr.dbutils.bcp package.

EXAMPLE 4. using an exception factory

```
// try running the bcp command represented by the variable cmd.
// this is using an instance of the org.jax.mgi.shr.unix.RunCommand
try {
  runner.setCommand(cmd);
  exitCode = runner.run();
}
catch (InterruptedException e) {
  BCPExceptionFactory factory = new BCPExceptionFactory();
  BCPException e2 = (BCPException)
    factory.getException(BCPExceptionFactory.InterruptErr, e);
  e2.bind(cmd);
  throw e2;
}
```

The getException() method within ExceptionFactory has two signatures. One that just accepts the name of the exception and the other which additionally accepts another exception (as shown in Example 4). The use of the latter version is for chaining exceptions into a hierarchy of parent/

child relationships. When an exception is passed into the getException method, it is set as the parent exception. All exception types are possible since the signature accepts the Exception class. When the toString() or the getMessage() is called on an **MGIException**, all the messages up the hierarchy are returned separated by newlines.

ExceptionHandlers are currently not supported at the frameworks level and considered application level objects. See the DLAExceptionHandler from the org.jax.mgi.shr.dla package as an example. Handlers perform exception handling functionality such as logging and system exiting and centralize this logic in one place for the whole application.