Advanced Features

Trenton Computer Festival
April 16th & 17th, 2005

Michael P. Redlich
Senior Research Technician
ExxonMobil Research & Engineering
michael.p.redlich@exxonmobil.com

Table of Contents

1 Introduction

Java offers all of the advantages of object-oriented programming (OOP) by allowing the developer to create user-defined data types for modeling real world situations. However, the real power within Java is contained in its features. Four main topics will be covered in this document:

- Applets and Applications
- JavaBeans
- Exception handling
- Java Database Connectivity API

There will also be an introduction to the Java 2 Collections.

An example Java application was developed to demonstrate the content described in this document and the ***Introduction to Java*** document. The application encapsulates sports data such as team name, wins, losses, etc. The source code can be obtained from **`http://tcf.redlich.net/`**.

2 Applets and Applications

A Java ***applet*** requires the use of a browser, and is invoked within the **`<applet></applet>`** HTML tag pair. The bytecode is executed with the JVM built-in to the browser. The initial point of execution is within a method called **`init()`**.

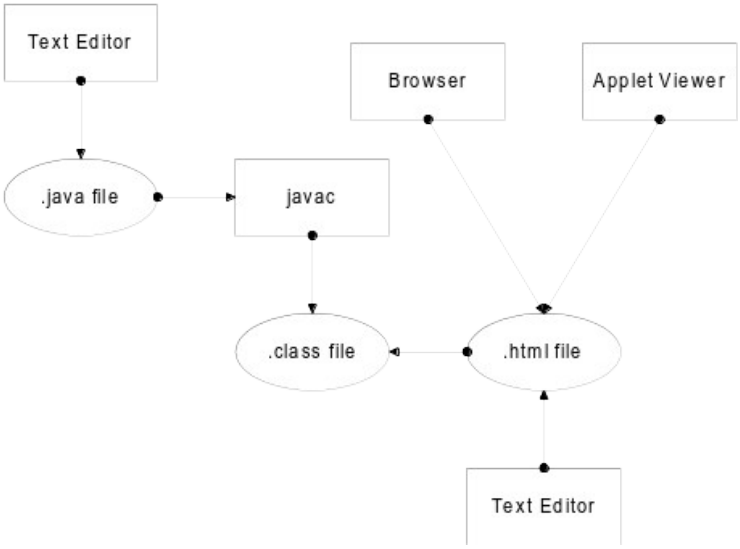
A Java ***application*** is standalone and is normally executed from the command line using the local JVM. The initial point of execution is within a static method called **`main()`**. The method signature is.

```
public static void main(String[] args)
    {
    ...
    }
```

This is, of course, similar to the C/C++ programming languages. However, the parameter list for handling command line arguments can be omitted in C/C++, but the parameter list ***must*** be supplied even if there is no intention of using command line arguments. Also missing from the parameter list is an integer for the number of arguments.

It is possible to write a Java program that can be used as an applet and an application. It must, of course, contain both **`init()`** and **`main()`** methods.

The following diagram describes the process of developing an applet:



The methods required for developing an applet are shown in the general source code example below:

```
// Java applet source code

import java.applet.*;
import java.awt.Graphics;
...

public class MyApplet extends Applet implements Runnable
    {
    public void MyApplet() // construction
        {...}
    public void init() // initialization
        {...}
    public void start() // starting
        {...}
    public void stop() // stopping
        {...}
    public void destroy() // destroying
        {...}
    public void paint(Graphics G) // painting
        {...}
    }

// HTML file that invokes applet

<html>
<head>
    <title>My Applet</title>
</head>

<body>

<hr>

<applet code="MyApplet.class" width="500" height="50">
    <param name="text" value="Welcome to my applet!">
</applet>

<hr>

</body>
</html>
```

3 JavaBeans

JavaBeans (or just Beans) is a method (sic) for developing reusable Java components that can be used in web applications, most notably within Java Server Pages (JSPs). Beans easily store and exchange information. In order for a Java class to be a bean, it must be developed according to the JavaBean specification:

- implements the **`Serializable`** interface
- contains a default constructor (for JSP pages)
- contains getter/setter methods for all the class members

The bean must implement the **`Serializable`** interface so that the bean's current state can be written to disk and recreated between web server restarts.

A default constructor is required when the bean will be used within a JSP page.

The getter and setter methods establish (set) and return (get) the current values of all the class members. These methods must follow a standard naming convention that is relative to each of the class members. The first letter of each class member is in lower case. The corresponding getter and setter methods start with the terms, **`get`** and **`set`**, and are completed with the member name containing an upper case first letter. The term, **`is`**, may be used for a getter method that returns a boolean value. For example:

```
public class SportsBean implements Serializable
    {
    private int win;
    private boolean empty;

    public Sport()
        {}

    public int getWin()
        {
        return win;
        }

    public void setWin(int win)
        {
        this.win = win;
        }

    public boolean isEmpty()
        {
        return empty;
        }

    public void setEmpty(boolean empty)
        {
        this.empty = empty;
        }
    }
```

In this example, the getter and setter methods for the class member, **`win`**, is **`getWin()`** and **`setWin()`** respectively.

4 Exception Handling

Detecting and handling errors within an application has traditionally been implemented using return codes. For example, a function may return zero on success and non-zero on failure. This is, of course, how most of the standard C library functions are defined. However, detecting and handling errors this way can become cumbersome and tedious especially in larger applications. The application's program logic can be obscured as well.

The ***exception handling*** mechanism in Java is a more robust method for handling errors than fastidiously checking for error codes. It is a convenient means for returning from deeply nested function calls when an exception is encountered. Unlike C++, exception handling was built-in to the Java programming language from the very beginning. Exception handling is implemented with the keywords **`try`**, **`throw`**, and **`catch`**. An exception is raised with a ***throw-expression*** at a point in the code where an error may occur. The throw-expression has the form:

```
throw T;
```

where **T** can be any data type for which there is an exception handler defined for that type. A ***try-block*** is a section of code containing a throw-expression or a function containing a throw-expression. A ***catch clause*** defined immediately after the try-block, handles exceptions. More than one catch clause can be defined. For example:

```
public class ExceptionTest
    {
    public static void main(String[] args)
        {
        try
            {
            initialize();
            }
        catch(Exception exception)
            {
            exception.printStackTrace();
            }
        }
    public void initialize() throws Exception
        {
        // contains code that may throw an Exception
        // type as specified
        }
    }
```

The Java library contains an extremely exhaustive list of defined exceptions for all types of errors. Most exceptions are ***checked***, i.e., the compiler enforces exception handling. If a certain method call is made without it being placed in a try block, the compiler will flag this as an error. The compiler does not enforce ***unchecked*** exceptions.

Exceptions should be thrown for things that are ***truly*** exceptional. They should not be thrown to indicate special return values.

5 Java Database Connectivity (JDBC)

The Java Database Connectivity (JDBC) API allows the developer to easily connect to, read, and manipulate popular databases (Microsoft Access and SQL Server, Oracle, etc.), spreadsheets, and flat files for use in applications. The JDK supplies a built-in driver for these databases that is used in conjunction with an Open Database Connectivity (ODBC) connection to a particular database. Other, more complex drivers (ones that don't require an ODBC connection, for example) must be obtained from a vendor, and referenced within the application. With only a few lines of code, a connection to a database can be made, a query can be executed, and a result set can be displayed:

```
public class DBTest
    {
    static public void main(String[] args)
        {
        String sql = "SELECT * FROM tblTimeZones;
        try
            {
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            Connection connection =
                DriverManager.getConnection("jdbc:odbc:timezones","","");
            Statement statement = connection.createStatement();
            ResultSet result = statement.executeQuery(sql);
            while(result.next())
                System.out.println(result.getDouble(1)
                    + " " + result.getDouble(2));
            connection.close();
            }
        catch(SQLException exception)
            {}
        }
    }
```

The driver for most databases using an ODBC connection can be accessed via the built-in **`JdbcOdbcDriver`** that is included in the Java Development Kit.

The example above assumes that a database exists with an ODBC connection named **`timezones`**, which is referenced in the statement:

```
Connection connection = DriverManager.getConnection("jdbc:odbc:timezones","","");
```

and returns an instance of type **`Connection`**. The empty strings in the second and third parameters of the **`getConnection()`** method are used for passing the userID and password of the database if they exist. The **`connection`** object is then used to return an instance of type **`Statement`** as in the statement:

```
Statement statement = connection.createStatement();
```

The **`Statement`** object is used to ultimately obtain the desired result set based on a given query. In the example above, the query, **`"SELECT * FROM tblTimeZones"`** is established as a string, and can be passed into the **`Statement`** object's **`executeQuery()`** method. A **`ResultSet`** object is returned upon a successful execution of the query. Examining all the rows of the result set is handled through a simple while loop:

```
while(result.next())
    System.out.println(result.getDouble(1) + " " + result.getDouble(2));
```

The ResultSet object's **`next()`** method returns a boolean to indicate if another row of data is available. The two calls to the **`getDouble()`** method assumes that values of type **`double`** are returned in columns 1 and 2 of the result set. Getter methods for all the built-in data types have been defined in **`ResultSet`**. Other examples include **`getInt()`** and **`getString()`**. The columns of the result set are one-based (as opposed to zero-based), that is the value of first column in each row is retrieved by **`getDouble(1)`** (or **`getString(1)`** or **`getInt(1)`**), and so on.

The only drawback to the above example is that the data types for the columns of the result set must be known in advance. If the schema of the database is changed, the above code can break. Other options are available, such as calling stored procedures and obtaining the result sets meta data.

6 Java 2 Collections

{*this section is under construction…*}

7 References for Further Reading

The references listed below are only a small sampling of resources where further information on Java can be obtained:

- Thinking in Java (*book*)
- Bruce Eckel
- ISBN
- **`http://www.bruceeckel.com/`**

- Java Developer's Journal (*monthly periodical*)
- **`http://www.javadevelopersjournal.com/`**

- Core Java 2, Volume I - Fundamentals (*book*)
- Cay S. Horstmann and Gary Cornell
- ISBN 0-13-081933-6
- **`http://www.sun.com/books/catalog/horstmann6/`**

- Core Java 2, Volume II - Advanced Features (*book*)
- Cay S. Horstmann and Gary Cornell
- ISBN 0-13-081934-4
- **`http://www.sun.com/books/catalog/horstmann7/`**

- The Java Tutorial for the Real World (*book*)
- Yakov Fain
- ISBN 0-9718439-0-2
- **`http://www.smartdataprocessing.com/`**