Porting Lucene to .NET Using Visual J#

Leveraging existing code to support both .NET and Java

Larry Reeve

Visual J# .NET is a Java programming language toolset from Microsoft. Designed to build applications that run on the .NET Framework, Visual J# includes a Java language compiler as well as support for independently developed portions of the Java Development Kit (JDK). Microsoft currently provides support for most of JDK 1.1.4 and portions of the JDK 1.2 java.util package (http://msdn.microsoft.com/vjsharp/). In addition, the Swing API is also available for academic use. J# has many uses, such as porting legacy Visual J++ code to .NET. J# also opens the door to porting existing Java libraries to .NET to extend the reach of existing Java code. Leveraging existing source to support both Java and .NET platforms can be very cost effective.

To determine how practical porting a Java library to J# is, I ported the Lucene search engine to .NET. Lucene is a text-search engine written entirely in Java. As a subproject of the Apache Jakarta Project (http://jakarta.apache.org/lucene/), Lucene is open source and freely downloadable. Originally developed by Doug Cutting, Lucene implements ranked and field searching based on Boolean and phrase queries. For developers, Lucene provides APIs for searching and maintaining document indexes, making Lucene ideal as an embedded component of larger systems.

Larry is a full-time software developer and is pursuing a Ph.D. at Drexel University. He can be contacted at larry-reeve@comcast.net.

There are enough implementation differences between the two current Lucene releases—1.2 Final Release and 1.3 Release Candidate 1. I will examine the porting issues for each.

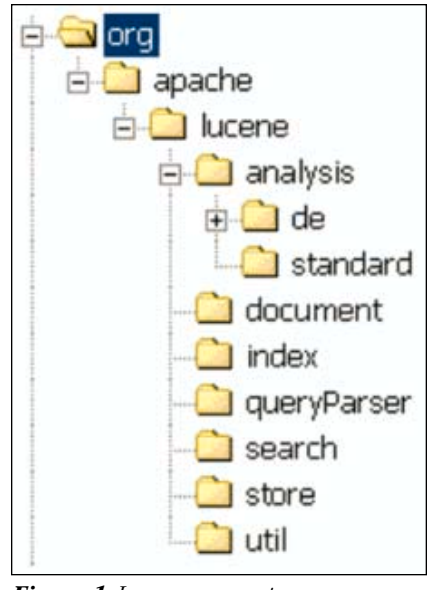

Figure 1: *Lucene source tree organization.*

1.2 Porting Issues

To determine a successful port, I set the following benchmarks:

- The core Lucene library must be compiled into a .NET DLL component.
- The IndexFiles.java and SearchFiles.java demo programs must be compiled using J# and use the Lucene .NET library.
- The Lucene library must be callable from a C# console search application using the index files generated by the IndexFiles application.

To provide data to search, the IndexFiles application is run against all files in the Lucene src directory hierarchy. The C# console search application is a direct conversion of the SearchFiles.java application; see Listing One.

The core Lucene source is decomposed into several directories, with each directory containing a functional area, such as search, data store, and so forth. Figure 1 illustrates the layout of the source area. The Lucene demo files are located in the src\demo\org\apache\lucene\demo path of the Lucene source.

There are three steps to porting Lucene:

1. Running the Java Compiler Compiler (JavaCC) parser generator against *.jj files to generate Java source files.
2. Making minor modifications to Lucene source to enable J# compilation.
3. Implementing parts of the JDK that are not supported or are incomplete under .NET.

The first step in porting Lucene is to generate a series of source files using the JavaCC tool, a Java-based parser generator application requiring the Java Runtime on the development machine (http://javacc.dev.java.net/). There are two areas of Lucene requiring conversion of JavaCC .jj source files into Java source files— analysis and query parser. Analysis provides tokenization of text input, while Query Parser provides support for the search syntax. JavaCC is run against the StandardTokenizer.jj file in the analysis directory and the QueryParser.jj file in the queryParser directory. The output of JavaCC is multiple Java source files in each directory; see Figure 2. The use of JavaCC is the same for both versions of Lucene.

The next porting step requires modifying several Lucene source files that do not compile with J# or cause unexpected run-time results. There are four Java source files out of 108 files in Lucene 1.2 that require modification to complete the port test; see Table 1. Some changes are minor, such as changing an EOF check on a *CharStream* class from –1 to <= 0. Although –1 is the indicator for EOF, during indexing tests several files returned 0 as an EOF condition rather than –1. I only replaced the EOF check in the

(continued from page 54)

FastCharStream class and the IndexFiles demo application ran successfully. Other classes may also have –1 EOF checks requiring change for production use.

Other minor modifications include adding a cast specification of "*L*" to a literal static *int* field in the *document/DateField* class. An interesting compiler failure that, if changed, could help J# work with existing Java code is checking a constant field that ignores source, acting as a sort of runtime preprocessor. In the *store/FSDirectory* class, a method called *obtain()* checks for a predefined Boolean constant called *JAVA_1_1*. If the value is True, a call to method *java.io.File.createNewFile()* is skipped. This lets the source adapt to the JDK it is running on. The J# compiler, in this case, attempted to resolve the call at compile time, which it could not accomplish due to the J# level of JDK support. The solution is to comment out the method call because, with JDK 1.1 support, the method will not be called anyway.

The *StandardTokenizer* class generated by JavaCC had the most source changes. The compilation issues center around support for an exception called "*ParseException*." Since the compiler does not allow this exception in method declarations, I modified the source to construct a *ParseException* class and, instead of throwing this exception, throw an *Error* exception instead. The information contained in the *ParseException toString()* method is passed to the *Error* exception as its input. This lets meaningful exception information still be generated if an error is encountered. Since this source file is automatically generated every time JavaCC is run, it must be maintained constantly. For this reason, JavaCC should only be run at the beginning of the port, and not as part of the build process.

TokenMgrError.java
Token.java
CharStream.java
ParseException.java

Figure 2: *JavaCC output files.*

After these porting steps are complete, Lucene 1.2 compiles and runs using J#. In the porting project, I set Lucene to compile into a DLL library callable from other .NET languages. This makes Lucene a valuable toolkit regardless of the language used.

1.3 RC1 Porting Issues

Porting the 1.3 RC1 version of Lucene follows the same steps as 1.2. However, 1.3 RC1 pushes Lucene with new features such as distributed searching, and takes advantage of newer JDK APIs. The dated 1.1.4 JDK that ships with J# is no longer as effective in porting Lucene 1.3 as it was for 1.2. As a result, adding JDK support is the most interesting of all the porting steps. Newer JDK APIs and class methods must be implemented to compile Lucene 1.3 successfully.

Since the Lucene library project uses the J# version of the JDK, the project must reference a library called "VJSLib." VJSLib contains Microsoft's implementation of the JDK APIs. VJSLib also contains the implementation of the com.* packages provided with Visual J++.

J# leaves the door open to replace or extend existing JDK APIs provided by Microsoft with custom versions of the APIs. For example, the *java.util.Vector* class provided by Microsoft is lacking newer method calls such as *toArray()*. To successfully compile and execute Lucene 1.3, you need an updated *Vector* class.

I added additional JDK support by implementing custom versions of the JDK API in a separate library project called "VJSLibEx." This project is then referenced by the Lucene library project. When compiling the Lucene library, this warning is issued:

analysis/standard/FastCharStream.java	EOF check for <= 0
analysis/standard/StandardTokenizer.java	Convert ParseException to Error exception
document/DateField.java	Casting issue with DATE_LEN
store/FSDirectory.java	Remove call to createNewFile()

Table 1: *Lucene 1.2 source files under org/apache/lucene/ requiring modifications for J#.*

All source changes from 1.2, and:	
store/FSDirectory.java	File.setLastModified() API needs implementation
	File.createNewFile() API needs implementation

Table 2: *Lucene 1.3 source files under org/apache/lucene/ requiring modifications for J#.*

```
warning VJS1500: Type 'java.util.Vector' is
found in more than one imported binaries.
Using the one in 'VJSLibEx.dll'
```

This indicates the new JDK functionality provided in VJSLibEx has replaced the Microsoft-provided functionality in VJSLib. Since the intention is to replace the *java.util.Vector* class, the warning is actually good news. VJSLibEx replaces only the *Vector* class, and not the entire *java.util* package provided by Microsoft. This allows for selective replacement of Java classes.

I decided that implementing and testing a complete *java.util.Vector* class on my own was a distraction from the main goal of getting Lucene to run under .NET. Consequently, I looked for an existing implementation of JDK APIs and came across the GNU Classpath project. The Classpath project describes itself as a set of core class libraries for use with Java Virtual Machines and compilers. Classpath has not reached a 1.0 release status yet, so it should still be considered under development. GNU Classpath is licensed under the GNU General Public License Version 2 (http://www.classpath.org/). I downloaded the GNU Classpath source for the *java.util.Vector* class and added it to the VJSLibEx project. It compiled without any issues under J#.

java.util.Vector is not the only class needing replacement for Lucene 1.3. It supports distributed searching and, therefore, requires the Java RMI APIs. The benchmark I set for the porting only requires local searching, and so I was not immediately interested in supporting the distributed search feature of 1.3. In this case, I merely stubbed out the Lucene-required RMI classes, so the code compiles successfully. Use of the stub technique means a successful port of Java code does not necessarily require that all JDK code be complete before functionality is delivered.

Another example of class replacement is the *java.util.Arrays* class. Lucene relies on three static methods in this class: two versions of *fill()* and one *sort()*. I implemented a simple replacement class with only these three methods; see Listing Two.

There are some classes, such as *java.lang.StringBuffer*, that cannot be extended using the library override technique. When a custom version of *StringBuffer* is used, compile errors occur. For example, a *throw* statement with string concatenation will not compile. Fortunately, Lucene does not require use of the newer *StringBuffer* methods such as *substring()* and *replace()*. I have not yet explored how to address this issue should a class not be replaceable.

Table 2 lists the source changes needed for Lucene 1.3. These changes are in addition to the 1.2 source changes. The 1.3-specific changes are related to the creation

of lock files requiring methods not supported under JDK 1.1. There is code checking the value of a constant to determine the JDK level, but as in the port to 1.2, the J# compiler attempts to resolve the API method calls at compile time, preventing successful compilation. Simply commenting out the API calls lets the 1.3 porting continue without errors.

Building and Testing

Once the source changes to Lucene are made and the Java APIs are added and extended, the Lucene 1.3 library can be successfully built.

I have put together two Visual Studio 2003 solutions (available electronically; see "Resource Center," page 7)— one for Lucene 1.2 and one for Lucene 1.3 RC1. Both solutions contain the individual projects for the Lucene library, the IndexFiles console application, and the J# and C# versions of the SearchFiles console application. The Lucene project contains the full source to the Lucene library and includes the required code changes. Each code change is bracketed with a pair of *// J# Port* comment blocks to make locating and identifying changes easier.

The IndexFiles and SearchFiles applications are provided as part of the Lucene source distribution as demonstration programs. The C# version of SearchFiles is an implementation mimicking the Java version of SearchFiles. It is intended to demonstrate language interoperability by calling the J# Lucene library. All of the executables can be built at once by selecting Build|Rebuild Solution from within Visual Studio.

At the root of each solution there is a Release directory containing the output of the solutions projects, as well as a batch file used to automate testing of indexing and searching. The default test will index the Lucene source and then prompt for a string to search. For example, searching for "array" returns every line where the word "array" is found.

Conclusion

The port described here is a demonstration of getting the base Lucene code to run as a component under .NET. Additional work needs to be done to support the international parser versions, such as German and Russian. Fully implementing the distributed search features in 1.3 requires implementation of the Java RMI APIs. To make Lucene 1.2 and 1.3 fully production ready, extensive tests should be run against the resulting .NET library.

There is a large base of existing open-source and proprietary Java code in use today, and many developers face the issue of integrating the .NET platform with the Java environment. At the same time, many Windows developers are faced with issues already solved by the Java community. For these developers, reusing existing Java code on .NET makes sense, since the .NET platform is, for the most part, language independent. As demonstrated with Lucene, a Java library can be easily used from languages such as C#. With both communities working together, code reuse can be achieved on a large scale. This requires better JDK support, more compatible J# compiler implementation, a conscience effort on the part of Java library developers to recognize the platforms their code will run on, and thorough testing of resulting libraries. The end result will be a proliferation of proven technology, usable regardless of platform.

DDJ

Listing One

```
using System;
using org.apache.lucene.analysis;
using org.apache.lucene.analysis.standard;
using org.apache.lucene.document;
using org.apache.lucene.search;
using org.apache.lucene.queryParser;
namespace CSharpSample
{
    class SearchFiles
    {
        [STAThread]
        static void Main(string[] args)
        {
            try
            {
                Searcher searcher = new IndexSearcher("index");
                Analyzer analyzer = new StandardAnalyzer();
                while (true)
                {
                    Console.Write("Query: ");
                    string line = Console.ReadLine();

                    if (line.Length <= 0)
                        break;
                    Query query = QueryParser.parse(line,"contents",analyzer);
                    Console.WriteLine("Searching for:
                                        " + query.toString("contents"));
                    Hits hits = searcher.search(query);
                    Console.WriteLine(hits.length()
                                        + " total matching documents");
                    const int HITS_PER_PAGE = 10;
                    for (int start = 0; start < hits.length();
                                                start += HITS_PER_PAGE)
                    {
                        int end = Math.Min(hits.length(),
                                                start + HITS_PER_PAGE);
                        for (int i = start; i < end; i++)
                        {
                            Document doc = hits.doc(i);
                            string path = doc.get("path");

                            if (path != null)
                            {
                                Console.WriteLine(i + ". " + path);
                            }
                            else
                            {
                                String url = doc.get("url");

                                if (url != null)
                                {
                                    Console.WriteLine(i + ". " + url);
                                    Console.WriteLine("   - "
                                                + doc.get("title"));
                                }
                                else
                                {
                                    Console.WriteLine(i + ". " +
                                                "No path nor URL for this document");
                                }
                            }
                        }
                        if (hits.length() > end)
                        {
                            Console.Write("more (y/n) ? ");

                            line = Console.ReadLine();

                            if (line.Length == 0 || line.StartsWith("\n"))
                                break;
                        }
                    }
                }
                searcher.close();
            }
            catch (Exception e)
            {
                Console.WriteLine(" caught a " + e.GetType()
                                + "\n with message: " + e.Message);
            }
        }
    }
}
```

Listing Two

```
package java.util;
public class Arrays
extends Object
{
    public static void fill(
        float[] a,
        float val)
    {
        for (int idx=0; idx < a.length; idx++)
            a[idx] = val;
    }
    static void fill(
        Object[] a,
        int fromIndex,
        int toIndex,
        Object val)
    {
        for (int idx=fromIndex; idx < toIndex; idx++)
            a[idx] = val;
    }
    public static void sort(
        int[] a,
        int fromIndex,
        int toIndex)
    {
        System.Array.Sort(a, fromIndex, toIndex - fromIndex);
    }
}
```

DDJ