JavaArt Chapter 1. On-the-Fly Dynamic Compilation and Execution

This is the first chapter of a two chapter section on how to convert a Java program into a PNG image which can be compiled and executed like the original source code. This *JavaArt* project may appear a bit frivolous, but actually illustrates a wide range of useful techniques, including Java 2D graphics generation, dynamic compilation, byte code manipulation, class loaders, reflection, and issues with creating drag-and-drop applications.

A typical JavaArt image is shown in Figure 1 (enlarged by a factor of 4 to make it easier to see).

Figure 1. An Example of JavaArt.

The image's original Java program is a small drawing application, which can be executed by dropping the PNG file onto a ExecutePixels desktop icon. ExecutePixels dynamically translates the image back into Java, compiles it to byte codes, before passing it to the JVM for execution.

The translation, compilation, and execution are carried out "on-the-fly" without generating any temporary files on the machine (e.g. Java or class files). This approach is often used when executing scripts coded in domain-specific languages: very large speed-ups can be obtained by compiling a script, but it's often not possible to generate temporary files in the process due to space or security restrictions on the device.

This chapter looks at two ways of implementing on-the-fly dynamic compilation. Initially I utilize Java 6's Compiler API, and then try out the Janino compiler API (http://www.janino.net/), which focusses on run-time compilation tasks.

One of my Compiler API examples uses Apache Jakarta BCEL (Byte Code Engineering Library) to examine byte code output. I also employ Java class loaders and reflection to load compiled code into the JVM and execute it.

The second JavaArt chapter concentrates on the translation process for converting Java source code into an image (and back again), and some of the ways that the ExecutePixels can be utilized as a desktop icon which reacts automatically to an image being dropped on top of it.

JavaArt was partially inspired by the Piet language (http://www.dangermouse.net/esoteric/piet.html), which is based around 'painting' a program using colored blocks. A block may be any shape and have holes of other colors inside it. Program instructions are defined by the color transitions from one block to the next in the image. JavaArt is much more conventional since a

programmer writes an ordinary Java program first, then translates it into an image as a separate step.

1. Java 6's Compiler API

Java's Compiler API originated in JSR 199 (http://jcp.org/en/jsr/detail?id=199), as a standard mechanism for invoking a compiler, and other tools, with the necessary interfaces. The API includes programmatic access to diagnostics output (i.e. to error messages and warnings), and allows the compiler's input and output forms to be customized.

A program that invokes the Java compiler can be written in a few lines:

```
import javax.tools.*;

public class JCompiler0
{
  public static void main(String[] args)
  {
    JavaCompiler compiler = ToolProvider.getSystemJavaCompiler();
       // access the compiler provided
    if (compiler == null)
      System.out.println("Compiler not found");
    else {
      int result = compiler.run(null, null, null, args);
                              // compile the files named in args
      if(result == 0)
        System.out.println("Compilation Succeeded");
      else
        System.out.println("Compilation Failed");
    }
  }  // end of main()

} // end of JCompiler0 class
```

ToolProvider.getSystemJavaCompiler() gives the program access to the compiler, and run() calls it. The first three arguments of run() specify alternatives to System.in, System.out, and System.err, which aren't used in this example -- diagnostics messages are written to the default output and error streams. run()'s fourth argument is an array of filenames (in args) obtained from the command line.

run() returns 0 if the compilation finishes successfully, and non-zero if errors have occurred.

There is, unfortunately, a (mild) shock in store for users of this program. It compiles without problem, but is unable to find a compiler at runtime:

```
> javac JCompiler0.java
> java JCompiler0 Painter.java    // compile the Painter program
Compiler not found
>
```


The catch is that the JRE, where java.exe is located (for most users), does not include a compiler. When the JRE's java.exe calls ToolProvider.getSystemJavaCompiler() to obtain a compiler reference, null is returned instead.

We can give java.exe a helping hand by including the JDK's compiler in the classpath used by JCompiler0:

```
> java -cp "C:\Program Files\Java\jdk1.6.0_01\lib\tools.jar;."
                                          JCompiler0 Painter.java
```

The compiler JAR (tools.jar) is located in <JDK_HOME>\lib.

This approach is simple, but has ramifications for applications that want to employ dynamic compilation. One problem is that the location of tools.jar isn't certain, so the classpath addition isn't a robust solution. The safest thing is to include a copy of tools.jar with the application. Unfortunately, this adds nearly 12 MB to the download. There's also the legal issue of distributing Sun Microsystems software with your application, although Sun is currently open sourcing Java at the OpenJDK site (http://openjdk.java.net/).

For the moment, I'll keep on utilizing the classpath solution, but these issues are the motivation for my use of the Janino API later in this chapter.

2. Advanced Usage of the Compiler API

More complex uses of the Compiler API involve compilation tasks, the processing of diagnostic messages, and different forms of IO, made possible by the CompilationTask, DiagnosticListener, JavaFileManager, and JavaFileObject interfaces. Their relationship to JavaCompiler is illustrated in Figure 2.

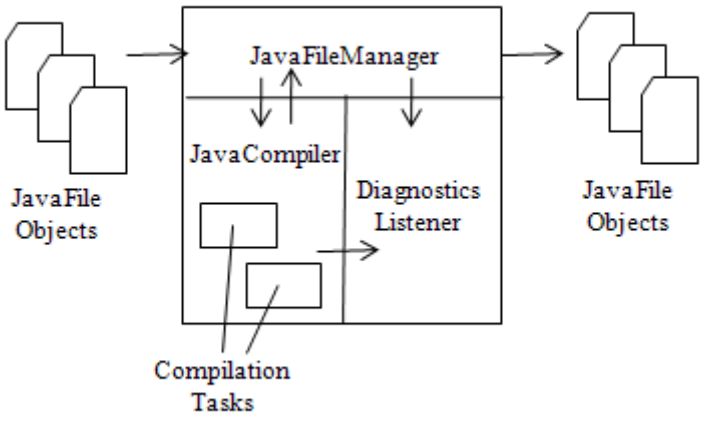

Figure 2. Some of the Compiler API Interfaces.

JavaFileObject acts as a file abstraction for Java source input and byte code output (e.g. to class files). It's a sub-interface of FileObject, an abstraction for more general forms of data, such as text files and databases.

JavaFileManager specifies the IO interface for the compiler. Most applications use the StandardJavaFileManager sub-interface, which handles IO based around java.io.File objects.

JavaCompiler can use the run() method to immediately compile input (as in my earlier JCompiler0 example), but greater flexibility is possible by utilizing CompilationTask objects which can have their own file managers and diagnostics listeners.

DiagnosticListener responds to diagnostic messages generated by the compiler or file manager. My examples use an alternative approach, the Compiler API's DiagnosticCollector class, which collects Diagnostic objects in an easily printable list.

3. Compiling the Fancy Way

It's possible to rewrite the JCompiler0 example to use the more advanced Compiler API interfaces and classes. The new JCompiler example is structured as in Figure 3.

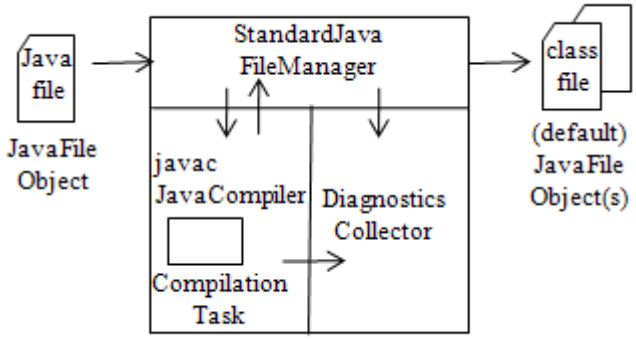

Figure 3. The Components of the JCompiler Example.

A single Java file is input as a JavaFileObject instance, and class files are output. The Compiler API's StandardJavaFileManager is utilized for file management, and DiagnosticsCollector for reporting problems.

JCompiler's main() function creates a single compilation task, executes it with CompilationTask.call(), and prints any diagnostic messages:

```
// global
private static DiagnosticCollector<JavaFileObject> diagnostics;


public static void main(String[] args)
{
  if (args.length < 1) {
    System.out.println("No file name supplied");
    System.exit(1);
  }

  CompilationTask task = makeCompilerTask(args[0]);
            // args[0] is the name of the file to be compiled

  System.out.println("Compiling " + args[0]);
```



```
  if (!task.call())  // carry out the compilation
    System.out.println("Compilation failed");

  for (Diagnostic d : diagnostics.getDiagnostics())   // list probs
    System.out.println(d);
}  // end of main()
```

makeCompilerTask() handles the creation of the input Java file object, the file manager, compiler, and diagnostics collector shown in Figure 3.

```
private static CompilationTask makeCompilerTask(String fileName)
// create a compilation task object for compiling fileName
{
  JavaCompiler compiler = ToolProvider.getSystemJavaCompiler();
  if (compiler == null) {
    System.out.println("Compiler not found");
    System.exit(1);
  }

  // create a collector for diagnostic messages
  diagnostics = new DiagnosticCollector<JavaFileObject>();

  // obtain a standard manager for Java file objects
  StandardJavaFileManager fileMan =
     compiler.getStandardFileManager(diagnostics, null, null);

  /* build an iterable list holding a Java file object
     for the supplied file */
  Iterable<? extends JavaFileObject> fileObjs =
     fileMan.getJavaFileObjectsFromStrings(Arrays.asList(fileName));

  /* create a compilation task using the supplied file
     manager, diagnostic collector, and applied to the
     Java file object */
  return compiler.getTask(null, fileMan, diagnostics,
                                        null, null, fileObjs);
}  // end of makeCompilerTask()
```

The call to JavaCompiler.getStandardFileManager() includes a reference to the diagnostics collector where all non-fatal diagnostics will be sent; fatal errors trigger exceptions.

JavaCompiler.CompilationTask.getTask() expects its input to be an iterable list of JavaFileObjects, which is built in two stages. First the filename string taken from the command line is placed inside a list with Arrays.asList(), then the strings list is converted into a list of JavaFileObjects with StandardJavaFileManager.getJavaFileObjectsFromStrings().

There's no need to specify Java file object output in makeCompilerTask(), since the default behavior of JavaFileManager is to store generated byte codes in a class file in the local directory.

A call to JCompiler to compile Painter.java produces the following output:

```
> java -cp "C:\Program Files\Java\jdk1.6.0_01\lib\tools.jar;."
                                          JCompiler Painter.java
```



```
Compiling Painter.java
Note: Painter.java uses or overrides a deprecated API.
Note: Recompile with -Xlint:deprecation for details.
>
```

Despite the deprecation warning, Painter.java is compiled, and the resulting class files are saved in the current directory. Painter can be called in the usual way:

```
> java Painter
```

3.1. What's been Deprecated?

It would be useful for the compiler to print more information about Painter's deprecated code. This requires the inclusion of the "-Xlint:deprecation" command line option as a JavaCompiler.CompilationTask.getTask() parameter. The changes to makeCompilerTask() in JCompiler.java are:

```
// in makeCompilerTask()
Iterable<String> options =
      Arrays.asList( new String[]{"-Xlint:deprecation"} );
return compiler.getTask(null, fileMan, diagnostics,
                              options, null, fileObjs);
```

Compiler options must be supplied to getTask() as an iterable list of strings.

The compilation of Painter.java now produces:

```
> java -cp "C:\Program Files\Java\jdk1.6.0_01\lib\tools.jar;."
                                          JCompiler Painter.java
Compiling Painter.java
Painter.java:34: warning: [deprecation] show() in java.awt.Window has
been deprecated
>
```

The offending line is a call to Window.show(), rather than the use of setVisible(true).

4. Compiling a Source String

The previous JCompiler example mimics javac, by reading its source code from a file, and writing the compiled byte codes into a class file. The StringJCompiler example in this section modifies this behavior slightly, to read the code from a string created at runtime, as illustrated by Figure 4.

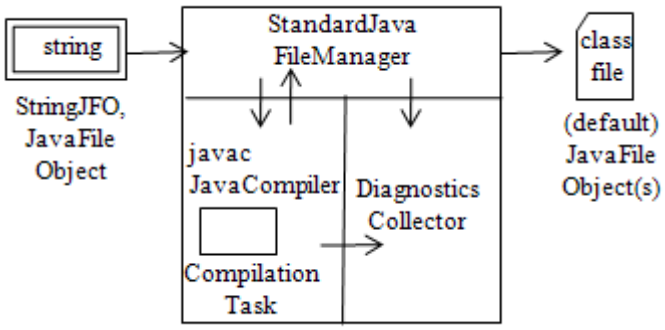

Figure 4. The Components of the StringJCompiler Example.

A comparison with Figure 3 (which gives the components of JCompiler) shows that only the input JavaFileObject instance is different in StringJCompiler. The string is stored in a new subclass of JavaFileObject, called StringJFO, so it can be manipulated by the file manager and compiler.

The main() function of StringJCompiler creates a StringJFO instance, compiles it, and prints any diagnostics.

```
// global -- a collector for diagnostic messages
private static DiagnosticCollector<JavaFileObject> diagnostics;


public static void main(String[] args) throws Exception
{
  if (args.length < 1) {
    System.out.println("No class name supplied");
    System.exit(1);
  }

  StringJFO src = makeCode(args[0]);
     // create a file object using args[0] as the class name

  CompilationTask task = makeCompilerTask(src);

  System.out.println("Compiling " + args[0]);
  if (!task.call())   // carry out the compilation
    System.out.println("Compilation failed");

  for (Diagnostic d : diagnostics.getDiagnostics())
    System.out.println(d);
}  // end of main()
```

The important difference from the previous example is the makeCode() method, which builds a string Java file object for a class, using args[0] as the class name. makeCode() returns a StringJFO object (called src), which is passed to makeCompilerTask() to be used in the compilation task.

The class created in makeCode() contains a main() method which prints “Hello” followed by the input argument:

```
private static StringJFO makeCode(String className)
```



```
/* Convert a string into a string java file object with the
   specified class name. */
{
  String codeStr =
      "public class " + className + " {" +
          "public static void main(String[] args){" +
              "System.out.println(\"Hello \" + args[0]);}}";

  StringJFO src = null;
  try {
    src = new StringJFO(className + ".java", codeStr);
  }
  catch (Exception e)
  {  System.out.println(e);
     System.exit(1);
  }
  return src;
} // end of makeCode()
```

The codeStr string is passed to the StringJFO constructor to build the Java file object. The class name is also supplied as the string's 'filename'. A filename is needed for the class file generated by the compiler to hold the resulting byte codes.

New kinds of Java file objects (such as StringJFO) can be created relatively easily by subclassing the Compiler API's SimpleJavaFileObject class which provides default implementations for most methods in the JavaFileObject interface.

```
public class StringJFO extends SimpleJavaFileObject
{
  private String codeStr = null;

  public StringJFO(String uri, String codeStr) throws Exception
  { super(new URI(uri), Kind.SOURCE);    // store source code
    this.codeStr = codeStr;
  }

  public CharSequence getCharContent(boolean errs) throws IOException
  // called by the Java compiler internally to get the source code
  { return codeStr; }

} // end of StringJFO class
```

SimpleJavaFileObject.getCharContent() is overridden to return the code string, which is passed to the object in the constructor.

4.1. Compiling the String JFO

StringJCompiler's makeCompilerTask() is virtually unchanged from the version in JCompiler, except for the string Java file object input to the compiler.

```
private static CompilationTask makeCompilerTask(StringJFO src)
/* Create a compilation operation object for compiling the
   string Java file object, src.
   The result is saved in a class file. */
{
  JavaCompiler compiler = ToolProvider.getSystemJavaCompiler();
```



```
    // access the compiler provided
  if (compiler == null) {
    System.out.println("Compiler not found");
    System.exit(1);
  }

  // create a collector for diagnostic messages
  diagnostics = new DiagnosticCollector<JavaFileObject>();

  // create a manager for the Java file objects
  StandardJavaFileManager fileMan =
          compiler.getStandardFileManager(diagnostics, null, null);

  /* create a compilation task using the supplied file manager,
    diagnostic collector, and applied to the string Java
    file object (in a list) */
  return compiler.getTask(null, fileMan, diagnostics,
                          null, null, Arrays.asList(src));
}  // end of makeCompilerTask()
```

JavaCompiler.getTask() requires an input list of Java file objects, so the StringJFO instance is added to a list by calling Arrays.asList().

5. Storing the Class Byte Codes in Memory

The next stage in our journey towards on-the-fly dynamic compilation and execution is to store generated byte codes in memory (in a HashMap) rather than in a class file.

The HashMap will hold specialized Java file objects that contain byte code. This approach requires a new subclass of SimpleJavaFileObject called ByteArrayJFO. The creation of ByteArrayJFO objects is managed by a new file manager, called ByteJavaFileManager.

The components of the StringJViewer example, which uses these new features, are shown in Figure 5.

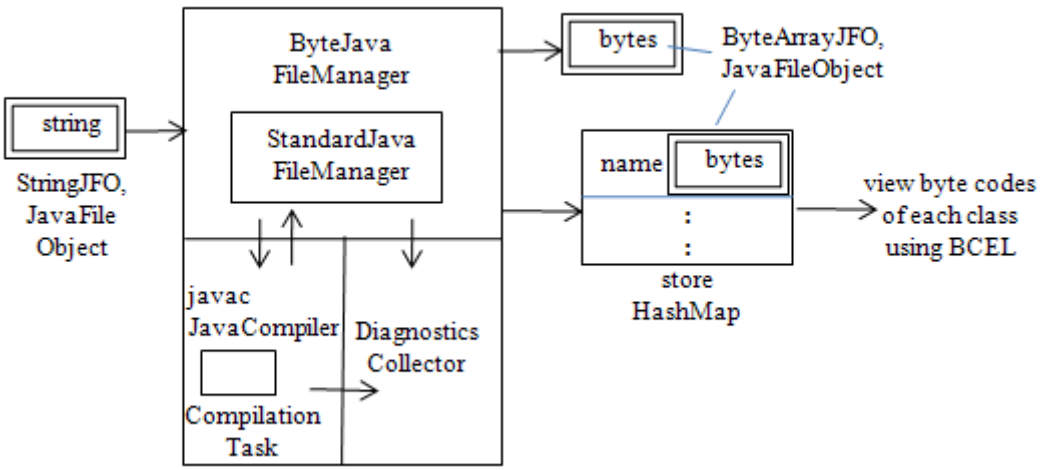

Figure 5. The Components of the StringJViewer Example.

StringJViewer also reports on the byte code contents of the objects in the HashMap, to confirm that the compiler is working correctly. I utilize Apache Jakarta BCEL (Byte Code Engineering Library), available from http://jakarta.apache.org/bcel/.

StringJViewer's main() function carries out three tasks: string Java file object generation, compilation, and byte code viewing using BCEL.

```
// global
private static final String CLASS_NAME = "Foo";
                          // name of the generated class


public static void main(String[] args)
{
  Map<String, JavaFileObject> store =
                           new HashMap<String, JavaFileObject>();
   // maps class names to JFOs containing the classes' byte codes

  StringJFO src = makeCode(CLASS_NAME);
  compileCode(src, store);
  viewStore(store);
}  // end of main()
```

makeCode() is unchanged from the version in StringJCompiler. It creates a StringJFO object for a "Foo" class which prints "Hello" followed by a string supplied as an input argument.

compileCode() creates a compilation task, executes it, and reports the result.

```
// global
private static DiagnosticCollector<JavaFileObject> diagnostics;
       // a collector for diagnostic messages


private static void compileCode(StringJFO src,
                           Map<String, JavaFileObject> store)
// compile using input from src, output to store
{
  CompilationTask task = makeCompilerTask(src, store);

  System.out.println("Compiling...");
  boolean hasCompiled = task.call();   // carry out the compilation

  for (Diagnostic d : diagnostics.getDiagnostics())
    System.out.println(d);

  if (!hasCompiled) {
    System.out.println("Compilation failed");
    System.exit(1);
  }
  else   // list generated class names
    System.out.println("Generated Classes: " + store.keySet());
  }  // end of compileCode()
```

A successful compilation triggers the listing of the keys in the store HashMap. There will be one key for each generated class.

makeCompilerTask() utilizes a new file manager, ByteJavaFileManager. It isn't a subclass of StandardJavaFileManager, since a manager is created by calling the compiler's getStandardFileManager() method, not by invoking a constructor. Instead ByteJavaFileManager employs forwarding (delegation) to contact a StandardJavaFileManager instance.

ByteJavaFileManager lets the StandardJavaFileManager object do most of the work, but handles the storage of compiled classes in the HashMap.

```
private static CompilationTask makeCompilerTask(StringJFO src,
                               Map<String, JavaFileObject> store)
/* Create a compilation operation object for compiling the
   string java file object, src. Use a specialized Java file
   manager.
   The resulting class is saved in store under the class name.
*/
{
  JavaCompiler compiler = ToolProvider.getSystemJavaCompiler();
     // access the compiler provided
  if (compiler == null) {
    System.out.println("Compiler not found");
    System.exit(1);
  }

  // create a collector for diagnostic messages
  diagnostics = new DiagnosticCollector<JavaFileObject>();

  // create a standard manager for the Java file objects
  StandardJavaFileManager fileMan =
            compiler.getStandardFileManager(diagnostics, null, null);

 /* forward most calls to the standard Java file manager
    but put the compiled classes into the store with
    ByteJavaFileManager */
  ByteJavaFileManager jfm =
    new ByteJavaFileManager<StandardJavaFileManager>(fileMan, store);

  /* create a compilation task using the supplied file
     manager, diagnostic collector, and applied to the
     string Java file object (in a list) */
  return compiler.getTask(null, jfm, diagnostics,
                          null, null, Arrays.asList(src));
}  // end of makeCompilerTask()
```

5.1. The Byte Codes Manager

ByteJavaFileManager is a subclass of ForwardingJavaFileManager, which allows it to forward tasks to a given file manager. The ByteJavaFileManager() constructor sets that manager, and also stores a reference to the store HashMap.

```
// global
private Map<String, JavaFileObject> store =
                  new HashMap<String, JavaFileObject>();
  // maps class names to JFOs containing the classes' byte codes
```



```
public ByteJavaFileManager(M fileManager,
                           Map<String, JavaFileObject> str)
{  super(fileManager);
   store = str;
}  // end of ByteJavaFileManager()
```

ByteJavaFileManager overrides ForwardingJavaFileManager.getJavaFileForOutput(), which is called by the compiler to create a new Java file object for holding its output.

```
public JavaFileObject getJavaFileForOutput(Location location,
            String className, Kind kind, FileObject sibling)
                                          throws IOException
{
  try {
    JavaFileObject jfo = new ByteArrayJFO(className, kind);
    store.put(className, jfo);
    return jfo;
  }
  catch(Exception e)
  {  System.out.println(e);
     return null;
  }
} // end of getJavaFileForOutput()
```

getJavaFileForOutput() creates a ByteArrayJFO instance for the compiler, which will fill it with the byte codes for the named class. A reference to the ByteArrayJFO object is also added to the store, so those byte codes can be accessed by other parts of the application.

5.2. Byte Codes in a Java File Object

ByteArrayJFO creates a Java file object which accepts byte codes on its input stream, and sends them to its output stream. It also allows the output stream to be accessed as a byte array.

```
public class ByteArrayJFO extends SimpleJavaFileObject
{
  private ByteArrayOutputStream baos = null;

  public ByteArrayJFO(String className, Kind kind) throws Exception
  {  super( new URI(className), kind);  }

  public InputStream openInputStream() throws IOException
  // the input stream to the java file object accepts bytes
  { return new ByteArrayInputStream(baos.toByteArray()); }

  public OutputStream openOutputStream() throws IOException
  // the output stream supplies bytes
  {  return baos = new ByteArrayOutputStream();  }

  public byte[] getByteArray()
  // access the byte output stream as an array
  {  return baos.toByteArray(); }

} // end of ByteArrayJFO class
```


openInputStream() and openOutputStream() are overridden SimpleJavaFileObject methods. The new method, getByteArray(), allows the byte codes to be retrieved from a ByteArrayJFO object as a byte[] array.

5.3. Viewing the Byte Codes with BCEL

The Java file objects in the store HashMap are examined by the viewStore() method in StringJViewer:

```
private static void viewStore(Map<String, JavaFileObject> store)
// Extract JFOs from the store, and display their info.
{
  for (String className : store.keySet()) {
    System.out.println(className + " Info");
    JavaFileObject jfo = store.get(className);
    if (jfo == null)
      System.out.println("Class not found in store");
    else {
      byte[] bytes = ((ByteArrayJFO)jfo).getByteArray();
                  // extract the byte codes array
      viewClass(className, bytes);
    }
  }
}  // end of viewStore()
```

BCEL has two main parts: a static API for analyzing existing compiled code, and a dynamic API for creating or transforming class files at runtime. viewClass() utilizes the static part, which represents a compiled Java class as a JavaClass object. viewClass() creates a JavaClass instance by parsing the byte codes in a byte[] array with BCEL's ClassParser:

```
private static void viewClass(String className, byte[] bytes)
{
  ClassParser p = new ClassParser(
           new ByteArrayInputStream(bytes), className);
  JavaClass jc = null;
  try {
    jc = p.parse();
  }
  catch(IOException e)
  {  System.out.println(e);
     return;
  }

  System.out.println(jc);   // print class structure
  // viewMethods(jc);       // view more methods info
  }  // end of viewClass()
```

viewClass() prints the JavaClass object, which contains enough information for my needs (to confirm that the Java code string was compiled correctly). The details printed for the Foo class used in StringJViewer are:

```
public class Foo extends java.lang.Object
filename               Foo
compiled from          Foo.java from StringJFO
```



```
compiler version        50.0
access flags            33
constant pool           40 entries
ACC_SUPER flag          true

Attribute(s):
        SourceFile(Foo.java from StringJFO)

2 methods:
        public void <init>()
        public static void main(String[] arg0)
```

The output shows that Foo comes from a StringJFO object, and contains a default init() method, and a main() function.

It's quite straightforward to access more information about the methods, including their byte code instructions, by utilizing BCEL's Method and Code classes, as in viewMethods():

```
private static void viewMethods(JavaClass jc)
// display method details and their instructions
{
  Method[] methods = jc.getMethods();
  for(int i=0; i < methods.length; i++) {
    System.out.println(methods[i]);  // print method declaration
    Code code = methods[i].getCode();
    if (code != null)
      System.out.println(code);
       // code mnemonics with symbolic refs resolved
  }
}  // end of viewMethods()
```

viewMethods() can be called from viewClass(), but the call is commented out at present, since I don't need that level of detail.

5.4. Getting StringJViewer to Work

BCEL isn't part of Java's standard distribution, and must be obtained from the Apache Jakarta site at http://jakarta.apache.org/bcel/. I downloaded the zipped binary for version 5.2, and unzipped it to a bcel-5.2/ directory on my d: drive. The directory contains the JAR file bcel-5.2.jar, which needs to be added to Java's classpath when StringJViewer is compiled and executed. For example:

```
> javac -cp "D:\bcel-5.2\bcel-5.2.jar;." StringJViewer.java

> java -cp "C:\Program Files\Java\jdk1.6.0_01\lib\tools.jar;
            D:\bcel-5.2\bcel-5.2.jar;." StringJViewer
```

5.5. More Information on BCEL

The BCEL manual is a good place to start for more information (http://jakarta.apache.org/bcel/manual.html): it briefly discusses the Java Virtual Machine and its class file format, introduces the BCEL API, and describes some typical application areas. The BCEL mailing lists, at

http://jakarta.apache.org/bcel/mail-lists.html, are a useful source for code snippets and advice.

The "Analyze Your Classes" article by Vikram Goyal at http://www.onjava.com/pub/a/onjava/2003/10/22/bcel.html is a more leisurely introduction, with examples using both the static and dynamic parts of the API.

"Bytecode engineering with BCEL" by Dennis Sosnoski at http://www.ibm.com/developerworks/java/library/j-dyn0414/ looks at how to use BCEL's dynamic API. He modifies method bodies in classes at runtime to measure the time they take to execute.

BCEL isn't the only choice for byte code manipulation: a good list of other open source libraries can be found at http://java-source.net/open-source/bytecode-libraries

6. Loading and Executing the Byte Codes

Our gradual build-up towards on-the-fly dynamic compilation and execution using the Compiler API is almost complete. The last things to add are: the loading of byte codes into the JVM with a class loader, and the execution of that code using reflection. Figure 6 extends Figure 5 to include those stages.

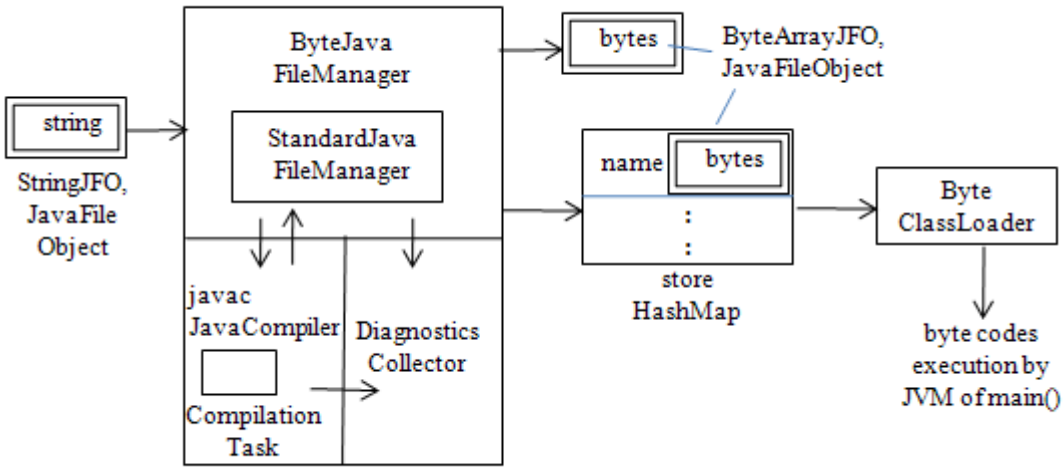

Figure 6. The Components of the StringJExecutor Example.

The StringJExecutor example based on Figure 6 discards the BCEL viewing code from StringJViewer to simplify the program.

The main() function of StringJExecutor is:

```
public static void main(String[] args)
{
  if (args.length < 1) {
    System.out.println("No argument supplied");
    System.exit(1);
  }
```



```
  Map<String, JavaFileObject> store =
                       new HashMap<String, JavaFileObject>();
   // maps class names to JFOs containing the classes' byte codes

  StringJFO src = makeCode(CLASS_NAME);
  compileCode(src, store);
  runCode(CLASS_NAME, args[0], store);  // load and run
}  // end of main()
```

makeCode() and compileCode() are unchanged from StringJViewer so won't be described again. runCode() uses a ByteClassLoader instance to load byte codes, and employs reflection to execute the main() method of the byte code class.

6.1. Loading Byte Codes

A Java class loader finds and loads a class at runtime when it's needed by an application or the JVM. The JRE contains several class loaders, and it is quite easy to add new ones by extending java.lang.ClassLoader.

A class loader must specify its parent loader because when it's asked to load a class, it always starts by delegating the search to its parent loader before attempting to find the class itself.

Most custom class loaders only need to override ClassLoader.findClass(), which is called with the name of the required class to load into the JVM.

The constructor of my ByteClassLoader class sets its parent loader, and maintains a reference to the store HashMap:

```
// global
private Map<String, JavaFileObject> store;

public ByteClassLoader(Map<String, JavaFileObject> str)
  {
  super( ByteClassLoader.class.getClassLoader() );   // set parent
  store = str;
}
```

The overridden findClass() finds byte codes by looking for the named class in the HashMap, and extracting them from the associated file object. The byte codes are passed to the JVM via ClassLoader.defineClass().

```
protected Class<?> findClass(String name)
                          throws ClassNotFoundException
{
  JavaFileObject jfo = store.get(name);    // load java file object
  if (jfo == null)
    throw new ClassNotFoundException(name);

  byte[] bytes = ((ByteArrayJFO)jfo).getByteArray();
                                  // get byte codes array
  Class cl = defineClass(name, bytes, 0, bytes.length);
                                  // send byte codes to the JVM
  if (cl == null)
    throw new ClassNotFoundException(name);
  return cl;
```



```
} // end of findClass()
```

ByteClassLoader can be used in the following way:

```
ByteClassLoader loader = new ByteClassLoader(store);
                                    // create the class loader
Class<?> cl = loader.loadClass(className);
                                    // load the specified class
```

The loader is initialized with the HashMap store, and then a class called className is loaded into the application as the Class object cl. Note that ClassLoader.loadClass() is called, not findClass().

6.2. Reflection

Reflection allows classes which have been loaded into the JVM to be examined and manipulation at runtime: objects can be constructed, fields accessed, and methods called.

The starting point for reflection is a java.lang.Class instance. A Class object holds details on the loaded class' constructors, fields, and methods, which are accessed via the Constructor, Field, and Method classes in the java.lang.reflect package.

I want to build a main() method call for the class loaded by ByteClassLoader. That requires Class.getMethod(), whose prototype is:

```
Method getMethod(String methodName, Class<?>[] parameterTypes);
```

parameterTypes are the method's formal parameter types, which in the case of main() are an array of strings (i.e. the command line arguments). Therefore, a main() method is represented by:

```
Method mainMeth = cl.getMethod("main",
                               new Class[] { String[].class });
```

A Method object is executed using Method.invoke(), whose prototype is:

```
Object invoke(Object obj, Object[] args);
```

The first parameter (obj) refers to the object being manipulated by the method call. For a static method (i.e. for main()), obj is assigned null.

The second invoke() parameter is an array of objects containing the method's actual parameters. If the method has no arguments, then args will be an array of length 0.

The main() method object is invoked using:

```
String[] methArgs = new String[] { arg };   // one argument
mainMeth.invoke(null, new Object[] { methArgs });
```


6.3. Putting Class Loading and Reflection Together

The StringJExecutor class loads the byte codes as a class and invokes its main() method in runCode():

```
private static void runCode(String className, String arg,
                                  Map<String, JavaFileObject> store)
{
  System.out.println("Running...");
  try {
    ByteClassLoader loader = new ByteClassLoader(store);
                                  // create the class loader
    Class<?> cl = loader.loadClass(className);
                                  // load the specified class

    // Invoke the "public static main(String[])" method
    Method mainMeth = cl.getMethod("main",
                                   new Class[] { String[].class });
    String[] methArgs = new String[] { arg };   // one argument
    mainMeth.invoke(null, new Object[] { methArgs });
  }
  catch(Exception ex)
  {
    if (ex instanceof InvocationTargetException) {
      // ex wraps an exception thrown by the invoked method
      Throwable t =
         ((InvocationTargetException) ex).getTargetException();
      t.printStackTrace();
    }
    else
      ex.printStackTrace();
  }
}  // end of runCode()
```

The arg String passed into runCode() comes from the command line when StringJExecutor is called. It is supplied as an argument to the main() method of the byte code class.

Method.invoke() throws exceptions generated by the invoked code wrapped up as InvocationTargetExceptions, and the actual exceptions can be accessed by calling getTargetException().

6.4. Calling StringJExecutor

StringJExecutor is executed with a single command line argument (e.g. “andrew”):

```
> java -cp "C:\Program Files\Java\jdk1.6.0_01\lib\tools.jar;."
                                          StringJExecutor andrew
Compiling...
Generated Classes: [Foo]
Running...
Hello andrew
```

A class called Foo is dynamically generated, compiled, loaded, and executed, without the creation of any temporary files on the machine. The Foo class prints “Hello”

followed by the command line argument passed to it by StringJExecutor, resulting in "Hello andrew".

7. Janino

The preceding sections have shown that Java 6's Compiler API is quite capable of programming on-the-fly dynamic compilation, but with some drawbacks. The most major is the need to include tools.jar in the application since the JRE doesn't come with a compiler. This adds nearly 12 MB to the application's size, and there may be legal issues with using code from Sun Microsystems' JDK. However, these issue will soon disappear with the release of a stable version of OpenJDK (http://openjdk.java.net/).

A stylistic quibble with the Compiler API is that implementing on-the-fly dynamic compilation requires quite a lot of coding, especially for such a standard kind of compilation task.

For these reasons, I decided to investigate the use of an alternative compilation approach, the Janino API (http://www.janino.net/), developed by Arno Unkrig. The JAR file for the entire API weighs in at a light 440 KB, and is open source. It offers more direct support for on-the-fly dynamic compilation than the Compiler API, to the extent that I don't need to implement any additional support classes and data structures.

Janino is used quite widely, for example in Ant, the Apache Commons JCI (a Java compiler interface), Tomcat, Groovy, and JBoss Rules (formerly Drools).

Perhaps the most significant drawback of Janino is that it's only compatible with Java 1.4, so programs using most Java 5 and 6 features can't be compiled. However, static imports, autoboxing and unboxing, and StringBuilder are available.

I downloaded Janino version 2.5.8 from http://www.janino.net/ as a zipped file containing the janino.jar library, source code, examples, and API documentation. I placed the unzipped directory, janino-2.5.8/ on my d: drive.

7.1. Compiling in One Line

The Janino compiler (the equivalent of javac) is invoked using Compiler.main():

```
public class JaninoCompiler
{
  public static void main(String[] args)
  {  org.codehaus.janino.Compiler.main(args);  }
}
```

The call to Janino's Compiler class has to include its package name to distinguish it from Java's java.lang.Compiler class.

This example is compiled and run like so:


```
> javac -cp "D:\janino-2.5.8\lib\janino.jar;." JaninoCompiler.java
> java -cp "D:\janino-2.5.8\lib\janino.jar;."
                                  JaninoCompiler Painter.java

         // use the Janino compiler to compile Painter.java
> java Painter
```

The classpath refers to the location of the janino.jar JAR file on my machine.

7.2. On-the-Fly Dynamic Compilation and Execution

Where the Janino API really shines in its support for on-the-fly compilation. The following StringExecutor example is a Janino recoding of my final Compiler API example, StringJExecutor, from section 6. It compiles and loads a string representing a class, and call it's main() function with a single argument. There's no need to create Java file objects, a file manager, a HashMap data structure, or a class loader for the byte codes. StringExecutor employs Janino's SimpleCompiler class which compiles a single "compilation unit" (the equivalent of the contents of a Java file), and makes it available via its own class loader.

```
public class StringExecutor
{
  private static final String CLASS_NAME = "Foo";


  public static void main(String[] args) throws Exception
  {
    if (args.length < 1) {
      System.out.println("No argument supplied");
      System.exit(1);
    }

    String codeStr =
      "public class " + CLASS_NAME + " {" +
          "public static void main(String[] args){" +
              "System.out.println(\"Hello \" + args[0]);}}";

    SimpleCompiler compiler = new SimpleCompiler();
    compiler.cook( new StringReader(codeStr) ); // compile the string

    // get the loaded class
    Class<?> cl = compiler.getClassLoader().loadClass(CLASS_NAME);

    // Invoke the "public static main(String[])" method
    Method mainMeth = cl.getMethod("main",
                                   new Class[] { String[].class });
    String[] methArgs = new String[] { args[0] };    // one input
    mainMeth.invoke(null, new Object[] { methArgs });
  }  // end of main()
}  // end of StringExecutor class
```

As in StringJExecutor, java.lang.Class represents the loaded Class object, and I utilize java.lang.reflect.Method to invoke its main() method.

StringExecutor is compiled and executed like so:


```
> javac -cp "D:\janino-2.5.8\lib\janino.jar;." StringExecutor.java
> java -cp "D:\janino-2.5.8\lib\janino.jar;." StringExecutor andrew
Hello andrew
```

StringExecutor carries out the same tasks as StringJExecutor: it compiles and executes a string class that prints “Hello” followed by the supplied command line argument,

8. Other Uses of Janino

Janino can be used to compile, load, and execute Java code *fragments*, such as expressions, method blocks, and class bodies (class code without a class name). Also, Janino gives the programmer access to the parse tree for a compiled program, which makes it easy to implement code analysis applications. The following two examples give a small taste of these capabilities.

8.1. Compiling a Class Body

The StringBodyExecutor example compiles and loads a string representing a class *body*, and then call it's main() function with a single argument. It employs Janino's ClassBodyEvaluator.

```
public class StringBodyExecutor
{
  public static void main(String[] args) throws Exception
  {
    if (args.length < 1) {
      System.out.println("No argument supplied");
      System.exit(1);
    }

    String codeStr =
            "public static void main(String[] args){" +
                "System.out.println(\"Hello \" + args[0]);}";

    // Compile the class body, and get the loaded class
    Class<?> cl = new ClassBodyEvaluator(codeStr).getClazz();

    // Invoke the "public static main(String[])" method
    Method mainMeth = cl.getMethod("main",
                                   new Class[] { String[].class });
    String[] methArgs = new String[] { args[0] };    // one input
    mainMeth.invoke(null, new Object[] { methArgs });
  }  // end of main()

} // end of StringBodyExecutor class
```

The class body is the same as the one in my StringExecutor example, but there’s no need to invent a dummy class name (in StringExecutor I used “Foo”).

8.2. Walking Over a Parse Tree

The following ShowMethodInfo class is a subclass of org.codehaus.janino.util.Traverser: it parses a Java file whose name is supplied on the command line, and then walks over its parse tree. It prints class, method, and method call information for the file.

The ShowMethodInfo() constructor builds a parse tree as a side effect of creating a Java compilation unit for the input file. It calls Traverser.traverseCompilationUnit() to begin walking over the tree.

```
public ShowMethodInfo(String fnm)
{
  Java.CompilationUnit cu = null;
  try {
    // parse the file
    FileReader fr = new FileReader(fnm);
    Parser p = new Parser( new Scanner(fnm, fr) );   // parse
    cu = p.parseCompilationUnit();  // build compilation unit
    fr.close();
  }
  catch(Exception e)
  {  System.err.println(e);
     System.exit(1);
  }

  // traverse the parse tree
  traverseCompilationUnit(cu);
}  // end of ShowMethodInfo()
```

Traverser contains numerous callback methods corresponding to the different grammatical entities in a parse tree, which can be overridden to carry out specific tasks. In ShowMethodInfo, the traverse methods for class declaration, constructor declarator, method declarator, and method invocation are specialized to print out node information from the tree.

```
public void traverseNamedClassDeclaration(
                            Java.NamedClassDeclaration ncd)
// print class names
{ System.out.println("\n" + ncd);
  super.traverseNamedClassDeclaration(ncd);
}

public void traverseConstructorDeclarator(
                             Java.ConstructorDeclarator cd)
// print constructor prototypes
{ System.out.println("  " + cd);
  super.traverseConstructorDeclarator(cd);
}

public void traverseMethodDeclarator(Java.MethodDeclarator md)
// print method prototypes
{ System.out.println("  " + md);
  super.traverseMethodDeclarator(md);
}

public void traverseMethodInvocation(Java.MethodInvocation mi)
// print method calls
```



```
{ System.out.println("   -> " + mi);
  super.traverseMethodInvocation(mi);
}
```

ShowMethodInfo produces the following output when asked to examine the Painter.java program:

```
Painter
  Painter()
   -> getContentPane()
   -> c.add(new PaintPanel(), "Center")
   -> c.add(new Label("Drag the mouse to draw"), BorderLayout.SOUTH)
   -> setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE)
   -> setSize(300, 150)
   -> show()
  main(String[] args)

PaintPanel
  PaintPanel()
   -> addMouseMotionListener(new MouseMotionAdapter() { ... })
  mouseDragged(MouseEvent e)
   -> e.getX()
   -> e.getY()
   -> repaint()
  paintComponent(Graphics g)
   -> g.fillOval(xCoord, yCoord, 4, 4)
```

This shows that Painter.java contains two classes (Painter and PaintPanel). Painter contains a constructor and main() method, while PaintPanel has a constructor and an implementation of paintComponent().

8.3. More Information on Janino

The Janino Web site describes a series of use cases for the API (http://www.janino.net/use.html), and there's an active mailing list archived at http://www.nabble.com/codehaus---janino-f11887.html and http://archive.janino.codehaus.org/user/.

"Tackling Java Performance Problems with Janino" by Tom Gibara at http://today.java.net/pub/a/today/2007/02/15/tackling-performance-problems-with-janino.html explains how Janino can improve the performance of a Java expressions evaluator, and substantially reduce the coding required to implement it.

9. Where Next?

This chapter looked at implementing on-the-fly dynamic compilation with Java 6's Compiler API, and the Janino compiler API. Class loaders and reflection were employed for dynamic execution, and BCEL was used to view class information.

Due to the small size of the Janino API, its open source license, and its more direct support for dynamic compilation, I'll use Janino from here on.

In the next JavaArt chapter, I'll concentrate on the translation process for converting Java source code into an image (and back again). I'll also examine various ways of making a drag-and-drop application, which allows a user to drop a JavaArt image onto a desktop icon, and have it execute automatically.

