Adobe

Adobe® Flex™ 2

Flex 2 Compiler API User Guide

Flex™ 2 Compiler API User Guide

Adobe Systems Incorporated, 345 Park Avenue, San Jose, California 95110, USA

Contents

CHAPTER 1

Introduction

1

This topic describes the Adobe® Flex™ compiler API, which is part of the OEM Kit. The Quick Start section contains a full example to get you started.

Contents

About the Flex compiler API

The Flex compiler API is part of the Flex OEM Kit. It lets you compile Flex applications from Java applications. You can also create applications in memory and compile them to SWF files without ever having an MXML file on disk. In addition, the compiler API lets you output linkage reports and other details about your applications.

You can also create project files and libraries with the compiler API. Libraries are SWC files that define a set of components for use in your application, theme files, or Runtime Shared Libraries (RSLs). Projects combine Flex applications and libraries. They let you enforce dependencies between a Flex application and its assets in the compilation process.

Requirements of the compiler API

The Flex compiler API has the following requirements:

License You must have a Flex Data Services 2 license to use any part of the Flex OEM Kit, including the compiler API.

Java JDK The compiler API requires that you have the Java interpreter and the javac compiler from JDK version 1.4.2 or later.

Flex The compiler API is not a stand-alone product. It requires the Flex SDK or the SDK included with Flex Data Services. You can also use it with Flex Builder.

What's included

The compiler API is a ZIP file that includes the following resources:

flex-compiler-oem.jar The flex2.tools.oem.* API for the compiler API. This JAR file is in the /lib directory in the ZIP file. When you expand the ZIP file, you should move this file to your /lib directory.

JavaDocs The API documentation for the public classes and interfaces in the flex2.tools.oem.* package. These files are in the /api directory in the ZIP file.

README.txt The readme file is at the top level of the ZIP file. It contains installation instructions and information on using a temporary license with the compiler API.

In addition, you can download the following separately:

CompilerAPIUserGuide.pdf The Flex 2 Compiler API User Guide is a PDF that provides usage documentation for the compiler API part of the OEM Kit. This file is at the top level of the ZIP file.

Quick Start

The following example describes how to create a simple Java application that compiles a Flex application.

To create and compile a new application with the compiler API:

1. Create a Java application; for example, MyAppCompiler.java:

```
// java/MyAppCompiler.java
import flex2.tools.oem.Application;
import java.io.*;

public class MyAppCompiler {
    public static void main(String[] args) {
        try {
            Application application = new Application(new
                File("../apps/TestApp.mxml"));
            application.setOutput(new
                File("../apps/TestApp.swf"));
            long result = application.build(true);
            if (result > 0) {
                System.out.println("COMPILE OK");
            } else {
                System.out.println("COMPILE FAILED");
            }
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}
```

   In this file, ensure that you do the following:

   a. Create an Application object.

   b. Set an output file for the new Application object.

   c. Call the `Application.build()` method.

   d. Check for a value greater than 0 returned by the `build()` method.

2. Compile the class with your Java compiler; for example:

```
C:\myapps>javac -classpath c:\home\dev\depot\flex\sdk\lib\flex-compiler-
  oem.jar MyAppCompiler.java
```

   Ensure that you add the flex-compiler-oem.jar to your Java classpath.

3. Create an MXML file; for example,TestApp.mxml:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Label text="Hello World"/>
</mx:Application>
```

4. Run the new application compiler with your Java interpreter; for example:

```
C:\myapps>java -classpath c:\home\dev\depot\flex\sdk\lib\flex-compiler-
  oem.jar;. MyAppCompiler
```

Ensure that you add the flex-compiler-oem.jar, and the current directory, to your Java classpath.

The following example sets several configuration options by using the Configuration class's methods:

```
// java/MyConfiguringCompiler.java
import flex2.tools.oem.Application;
import flex2.tools.oem.Configuration;
import java.io.*;

public class MyConfiguringCompiler {
    public static void main(String[] args) {
        String outputRoot = "../apps/";

        try {
            Application application = new Application(new
                File(outputRoot, "ErrorTestApp.mxml"));
            application.setOutput(new
                File(outputRoot, "ErrorTestApp.swf"));

            Configuration config = application.getDefaultConfiguration();

            // Enable ActionScript optimizer.
            config.optimize(true);

            // Disable warnings.
            config.showActionScriptWarnings(false);
            config.showBindingWarnings(false);
            config.showDeprecationWarnings(false);
            config.showUnusedTypeSelectorWarnings(false);

            // Apply the new configuration to the Application.
            application.setConfiguration(config);

            long result = application.build(true);
            if (result > 0) {
                System.out.println("COMPILE OK");
            } else {
                System.out.println("COMPILE FAILED");
            }
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}
```

Configuring the compiler

You can pass configuration options to the compiler when using the compiler API. To do this, you get an instance of the Configuration class using the Application or Library classes's `getDefaultConfiguration()` method, set the options on that instance, and then assign that object to the Application or Library by using the `setConfiguration()` method.

For most compiler options, there is a 1:1 mapping between the flex2.tools.oem.Configuration API methods and the options. For example, to set the value of the `keep-generated-actionscript` compiler option to `true`, you pass `true` to the Configuration class's `keepCompilerGeneratedActionScript()` method:

```
config.keepCompilerGeneratedActionScript(true);
```

Some compiler options, such as `source-path` and `library-path`, can use the += operator to append entries to the source path and library path. For these compiler options, you can replace the entire path using the `setSourcePath()` and `setLibraryPath()` methods, or you can append new entries to the list with the `addSourcePath()` and `addLibraryPath()` methods.

In some cases, the Configuration class does not have a method that maps to a compiler option. You might call a method of a different class, or call multiple methods. For example, there is no method that enables and disables all warnings as per the `warnings` compiler option. Instead, you call the `showActionScriptWarnings()`, `showBindingWarnings()`, `showDeprecationWarnings()`, and `showUnusedTypeSelectorWarnings()` methods.

The following table lists the compiler options that have alternative methods for setting their values:

Compiler option	Equivalent compiler API method or methods
`dump-config`	Call the `Configuration.keepConfigurationReport(true)` method and then the `Report.writeConfigurationReport()` method.
`include-classes`	Call the `Library.addComponent(java.lang.String)` method.
`include-file`	Call the `Library.addArchiveFile()` method.
`include-namespaces`	Call the `Library.addComponent(java.net.URI)` method.
`include-sources`	Call the `Library.addComponent(VirtualLocalFile)` or `Library.addComponent(java.io.File)` method.
`library-path`	Call the `Configuration.setLibraryPath()` method to replace the value of the default source path. Call the `Configuration.addLibraryPath()` method to append new values to the default source path.

Compiler option	Equivalent compiler API method or methods
`link-report`	Call the `Configuration.keepLinkReport(true)` method, then the `Report.writeLinkReport()` method.
`output`	Call the `Application.setOutput()` and `Library.setOutput()/setDirectory()` methods.
`resource-bundle-list`	Call the `Report.getResourceBundleNames()` method.
`source-path`	Call the `Configuration.setSourcePath()` method to replace the value of the default source path. Call the `Configuration.addSourcePath()` method to append new values to the default source path.
`version`	Call the `Report.getCompilerVersion()` method.
`warnings`	Call the `showActionScriptWarnings()`, `showBindingWarnings()`, `showDeprecationWarnings()`, and `showUnusedTypeSelectorWarnings()` methods.

About option and property precedence

Similar to mxmlc and compc, the default configuration that the compiler API uses is the flex-config.xml file. For compiling applications, the compiler API also uses local configuration files. The compiler API then applies the configuration options in flex2.tools.oem.Configuration to create the final set of compiler configuration options.

By default, the compiler invoked by the compiler API populates the default Configuration object with options in the flex-config.xml file. This file's location is relative to the location of the mxmlc.jar file. In most cases, this file is in the frameworks directory, which is at the same level as the JAR file's lib directory. The compiler also uses local configuration files (such as *app_name*-config.xml), or you can point to the location of another configuration file using the Configuration class's `addConfiguration()` method.

Configuration options that you set with the compiler API have the same precedence as a command-line compiler option. This means that they take precedence over all other options (flex-config.xml, alternative configuration file, and local configuration file).

For more information about using configuration files, see Chapter 2, "Using the Flex Compilers," in *Building and Deploying Flex 2 Applications*.

Clearing configurations

You can clear a configuration by calling the `Application.setConguration(null)` method.

Adding assets to applications

You can add assets to the application by using methods of the Configuration class, such as `addExterns()`, `addIncludes()`, and `setTheme()`. Using these methods, you can add external themes, libraries, classes, RSLs, and other types of assets.

The following example uses the Configuration class's `setTheme()` method to add a theme to the resulting application:

```
// java/MyThemeCompiler.java
import flex2.tools.oem.Application;
import flex2.tools.oem.Configuration;
import java.io.*;

public class MyThemeCompiler {
    public static void main(String[] args) {
        String assetRoot = "../assets/";
        String outputRoot = "../apps/";

        try {
            File[] themeFile = new File[]
                {new File(assetRoot, "myTheme.css")};

            Application application =
                new Application(new File(outputRoot,
                "TestAppWithAssets.mxml"));
            application.setOutput(new File(outputRoot,
                "TestAppWithAssets.swf"));

            Configuration config = application.getDefaultConfiguration();
            config.setTheme(themeFile);
            application.setConfiguration(config);

            long result = application.build(true);
            if (result > 0) {
                System.out.println("COMPILE OK");
            } else {
                System.out.println("COMPILE FAILED");
            }
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}
```

To add a SWC file that contains components, use the `addLibraryPath()` method. In the following example, the assets directory contains the MyComponents.swc file:

```
// java/MyLibraryPathCompiler.java
import flex2.tools.oem.Application;
import flex2.tools.oem.Configuration;
import java.io.*;

public class MyLibraryPathCompiler {
    public static void main(String[] args) {
        String assetRoot = "../assets/";
        String outputRoot = "../apps/";
        try {
            Application application = new Application(new
                File(outputRoot, "TestAppWithAllAssets.mxml"));
            application.setOutput(new
                File(outputRoot, "TestAppWithAllAssets.swf"));

            Configuration config = application.getDefaultConfiguration();
            File[] libFile = new File[]
                {new File(assetRoot, "MyComponents.swc")};
            config.addLibraryPath(libFile);
            application.setConfiguration(config);

            long result = application.build(true);
            if (result > 0) {
                System.out.println("COMPILE OK");
            } else {
                System.out.println("COMPILE FAILED");
            }
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}
```

To add a directory that contains MXML or ActionScript component files that are not in a SWC file, use the `addSourcePath()` method. In the following example, the assets directory contains several MXML files that are used by the TestSourcePathApp application:

```
// java/MySourcePathAppCompiler.java
import flex2.tools.oem.Application;
import flex2.tools.oem.Configuration;
import java.io.*;

public class MySourcePathAppCompiler {
    public static void main(String[] args) {
        String assetRoot = "../assets/";
        String outputRoot = "../apps/";
        try {
            File[] sourcePath = new File[] {new File(assetRoot)};

            Application application = new Application(new
                File(outputRoot, "TestSourcePathApp.mxml"));
            application.setOutput(new
                File(outputRoot, "TestSourcePathApp.swf"));

            Configuration config = application.getDefaultConfiguration();
            // The source path can be a directory.
            // All MXML and AS files in that directory are added
            // to the source path.
            config.addSourcePath(sourcePath);
            application.setConfiguration(config);

            long result = application.build(true);
            if (result > 0) {
                System.out.println("COMPILE OK");
            } else {
                System.out.println("COMPILE FAILED");
            }
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}
```

You can also create libraries or components with the compiler API. For more information on creating libraries, see “Creating libraries” on page 21. To add libraries and organize their dependencies at run time, use the Project class. For more information, see “Creating projects” on page 22.

CHAPTER 2

Logging and Reports

2

The Adobe Flex compiler API lets you generate reports and provide information such as progress and logs during the compilation process.

Contents

Using custom logging

You can capture error messages with the compiler. To do this, you create a custom logger and assign that logger to the applications that you are compiling.

To use a custom logger:

1. Create a Java class that implements the flex2.tools.oem.Logger interface; for example:

```
// java/SimpleLogger.java
import flex2.tools.oem.Message;
import flex2.tools.oem.Logger;
import java.util.*;

public class SimpleLogger implements Logger {
    SimpleLogger() {
        System.out.println("----------------------------------");
    }

    public void log(Message msg, int errorCode, String source) {
        System.out.println(msg);
        System.out.println("----------------------------------");
    }
}
```

   This class must define the `log()` method, which takes three arguments: the message, the error code, and the source.

2. In your Java application, call the Application instance's `setLogger()` method to assign a logger to that Application; for example:

   ```
   application.setLogger(new SimpleLogger());
   ```

   If you do not call the `setLogger()` method, the compiler logs messages to the standard output.

3. Compile and run your example.

4. To test the logger, add one or more syntax errors to your sample Flex application; for example:

   ```
   <?xml version="1.0"?>
   <!-- apps/ErrorTestApp.mxml -->
   <mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
       <mx:Script>
           // Generates a warning because there is no return type.
           public function doSomethingWrong() {
               // Generates an error because x lacks a type.
               var x;
           }
       </mx:Script>
       <mx:Label text="Hello World"/>
   </mx:Application>
   ```

This example produces output similar to the following:

```
---------------------------------
ERROR: Loading configuration file
  C:\home\dev\depot\flex\sdk\frameworks\flex-config.xml
---------------------------------------------------------------------------
ERROR: return value for function 'doSomethingWrong' has no type
  declaration.
---------------------------------
ERROR: variable 'x' has no type declaration.
---------------------------------
```

You can access other information about the error, such as the error level and the location of the error, by using the Message class's `getLevel()`, `getPath()`, `getLine()`, and `getColumn()` methods. The following example prints more information than the SimpleLogger example:

```
// java/ComplexLogger.java
import flex2.tools.oem.Message;
import flex2.tools.oem.Logger;
import java.util.*;
import java.io.*;

public class ComplexLogger implements Logger {
    ComplexLogger() {
        String LOGFILENAME = "output.txt";
        try {
            System.setOut(new PrintStream(new
                FileOutputStream(LOGFILENAME)));
        } catch (Exception e) {
            System.out.println("There was an error creating
                the log file.");
        }
        System.out.println("Ran at : " + new Date());
        System.out.println("-----------------------------------");
    }

    public void log(Message msg, int errorCode, String source) {
        if (msg.getLevel() == "info") {
            // Suppress info messages.
        } else {
            System.out.println("ERROR  : " + errorCode);
            System.out.println("MESSAGE: " + msg);
            System.out.println("SOURCE : " + source);
            System.out.println("LEVEL  : " + msg.getLevel());
            System.out.println("PATH   : " + msg.getPath());
            System.out.println("LINE   : " + msg.getLine());
            System.out.println("COLUMN : " + msg.getColumn());
            System.out.println("-----------------------------------");
        }
    }
}
```

Not all compiler errors generate valid values for all properties. For example, errors in your MXML code do not typically produce an error code, so the Logger returns a -1. ActionScript errors, however, do generate an error code.

You can also use the Report class to view Message objects when the compiler runs. For more information, see "Using reports to view messages" on page 17.

Creating reports

The compiler API includes the capability to create reports about the application, library, or project that you are compiling. In the reports, you can include details about the application's linkage, assets, dependencies, libraries, and resource bundles, in addition to information about the application, such as the background color, height, and width.

To get a report, you use the Application or Library object's `getReport()` method. This method returns an instance of the Report class, which you then use to output information about the target of the compilation.

Before you generate a report by using the Report class, you must build the Application or Library. This means that you must call the `build()` method before you can call any reporting methods.

The following example prints report information to the standard output:

```
// java/MyReportCompiler.java
import flex2.tools.oem.Application;
import flex2.tools.oem.Report;
import flex2.tools.oem.Configuration;
import flex2.tools.oem.Logger;
import java.io.*;

public class MyReportCompiler {
    public static void main(String[] args) {
        String assetRoot = "../assets/";
        String outputRoot = "../apps/";
        File[] themeFile = new File[]
            {new File(assetRoot, "myTheme.css")};
        File[] libFile = new File[]
            {new File(assetRoot, "MyComponents.swc")};

        try {
            Application application = new
                Application(new File(outputRoot,
                "TestAppWithAllAssets.mxml"));
            application.setOutput(new File(outputRoot,
                "TestAppWithAllAssets.swf"));

            application.setLogger(new ComplexLogger());
            application.setProgressMeter(new MyProgressMeter());

            Configuration config = application.getDefaultConfiguration();
            config.setTheme(themeFile);
            config.addLibraryPath(libFile);
            application.setConfiguration(config);

            application.build(true);

            Report report = application.getReport();

            // Lists the image files that are embedded.
            System.out.println("\n\nEMBEDDED ASSETS: ");
            String[] cnames = report.getAssetNames(Report.COMPILER);
            for (int i=0; i<cnames.length; i++) {
                System.out.println(cnames[i]);
            }

            // Lists the libraries that are used.
            System.out.println("\nLIBRARIES: ");
            String[] libs = report.getLibraryNames(Report.COMPILER);
            for (int i=0; i<libs.length; i++) {
                System.out.println(libs[i]);
            }
```

```
            // Lists source files, their definition names, and dependencies.
            System.out.println("\nSOURCE NAMES: ");
            String[] list = report.getSourceNames(Report.COMPILER);
            for (int i=0; i<list.length; i++) {
                System.out.println(list[i]);
                String[] defs = report.getDefinitionNames(list[i]);

                System.out.println("DEFINITIONS: ");
                for (int j=0; j<defs.length; j++) {
                    System.out.println(defs[j]);

                    System.out.println("   DEPENDENCIES: ");
                    String[] deps = report.getDependencies(defs[j]);
                    for (int k=0; k<deps.length; k++) {
                        System.out.println("      " + deps[k]);
                    }
                    System.out.println("   PREREQS: ");
                    String[] prereqs = report.getPrerequisites(defs[j]);
                    for (int k=0; k<prereqs.length; k++) {
                        System.out.println("      " + prereqs[k]);
                    }
                }
            }

            // Get application info.
            System.out.println("\nAPPLICATION INFO: ");
            System.out.println("   Background Color: " +
                "0x" + Integer.toHexString(report.getBackgroundColor()).
                    toUpperCase());
            System.out.println("   Height          : " +
                report.getDefaultHeight() +
                " (" +
                Math.round(100 * report.getHeightPercent()) +
                "%)");
            System.out.println("   Width           : " +
                report.getDefaultWidth() +
                " (" +
                Math.round(100 * report.getWidthPercent()) +
                "%)");
            System.out.println("   Page Title      : " +
                report.getPageTitle());
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}
```

The output might appear as follows:

```
EMBEDDED ASSETS:
C:\home\depot\EN\Docs\Flex\Flex2next\oem_kit\code\assets\bird-gray.gif
```

```
C:\home\depot\EN\Docs\Flex\Flex2next\oem_kit\code\assets\bird-silly.gif
C:\home\depot\EN\Docs\Flex\Flex2next\oem_kit\code\assets\bird.gif

LIBRARIES:
C:\home\depot\EN\Docs\Flex\Flex2next\oem_kit\code\assets\MyComponents.swc
C:\home\dev\depot\flex\sdk\bundles\en_US\charts_rb.swc
C:\home\dev\depot\flex\sdk\bundles\en_US\framework_rb.swc
C:\home\dev\depot\flex\sdk\frameworks\libs\flex.swc
C:\home\dev\depot\flex\sdk\frameworks\libs\framework.swc
C:\home\dev\depot\flex\sdk\frameworks\libs\playerglobal.swc

SOURCE NAMES:
C:\home\depot\EN\Docs\Flex\Flex2next\oem_kit\code\apps\TestAppWithAllAssets
  .mxml

DEFINITIONS:
TestAppWithAllAssets
   DEPENDENCIES:
      AS3
      MyButton
      MyLabel
      mx.controls:Image
      mx.controls:Label
      mx.core:UIComponentDescriptor
      mx.core:mx_internal
      mx.events:PropertyChangeEvent
      mx.styles:CSSStyleDeclaration
      mx.styles:StyleManager
   PREREQS:
      mx.core:Application

APPLICATION INFO:
   Background Color: 0x869CA7
   Height          : 375 (75%)
   Width           : 500 (100%)
   Page Title      : Test App With All Assets
```

Using reports to view messages

You can also send the messages that the compiler outputs during the compilation to the reports. Message levels include info, warning, and error. You access an Array of Message objects by using the `getMessages()` method of the Report class.

The following example gets an array of Message objects and prints them with the Message object's level:

```
// java/MySimpleReportCompiler.java
import flex2.tools.oem.Application;
import flex2.tools.oem.Report;
import flex2.tools.oem.Message;
import java.io.*;

public class MySimpleReportCompiler {
    public static void main(String[] args) {
        try {
            Application application = new Application(new
                File("../apps/ErrorTestApp.mxml"));
            application.setOutput(new File("../apps/ErrorTestApp.swf"));
            application.build(true);

            Report report = application.getReport();
            Message[] m = report.getMessages();
            for (int i=0; i<m.length; i++) {
                System.out.println(m[i].getLevel().toUpperCase() +
                    " MESSAGE " + i + ": " + m[i]);
            }
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}
```

Watching progress

The compiler API includes a progress meter that lets you observe the progress of the compilation.

To use the progress meter:

1. Create a Java class that implements the flex2.tools.oem.ProgressMeter interface; for example:

```
// java/MyProgressMeter.java
import flex2.tools.oem.ProgressMeter;

public class MyProgressMeter implements ProgressMeter {

    long before, after;

    MyProgressMeter() {
    }

    public void start() {
        before = System.currentTimeMillis();
        System.out.print("begin...");
    }

    public void end() {
        after = System.currentTimeMillis();
        System.out.println("done");
        System.out.println("Elapsed Time: " + (after - before) + "ms");
    }

    public void percentDone(int n) {
        System.out.print(n + "...");
    }

}
```

   This class must implement the `start()`, `end()`, and `percentDone()` methods. It adds logic to calculate the elapsed time during compilation.

2. In your Java application that compiles the Flex application, call the Application object's `setProgressMeter()` method to assign a progress meter to that Application; for example:

```
application.setProgressMeter(new MyProgressMeter());
```

3. Compile and run your application. This example produces output similar to the following:

```
begin...1...2...3...4...5...6...7...8...9...10...11...12...13...14...15
...16...17...18...19...20...21...22...23...24...25...26...27...28...29.
..30...31...32...33...34...35...36...37...38...39...40...41...42...43..
.44...45...46...47...48...49...50...51...52...53...54...55...56...57...
58...59...60...61...62...63...64...65...66...67...68...69...70...71...7
2...73...74...75...76...77...78...79...80...81...82...83...84...85...86
...87...88...89...90...91...92...93...94...95...96...97...98...99...100
...done
```

CHAPTER 3

Libraries and Projects

3

You create libraries and projects with the Adobe Flex compiler API.

Contents

Creating libraries

The most common type of library is a SWC file that contains custom components. You create SWC files by using the Flex compiler API. Also, you can create a component library that contains components, resource bundles, or other libraries.

To create a custom component library, you create an instance of the Library class. You then use the `addComponent()` method to add components, resource bundles, and archives to the library. You can add components by source file location, class name, namespace, or by adding dynamically generated component sources. This is the equivalent of using the `include-sources` component compiler option to add classes or directories to the SWC file.

The following example creates a SWC file with two components in it:

```
// java/MyLibraryCreator.java
import flex2.tools.oem.Library;
import java.io.*;

public class MyLibraryCreator {
    public static void main(String[] args) {
        String assetRoot = "../assets/";
        try {
            Library lib=new Library();
            lib.setOutput(new File(assetRoot, "MyComponents.swc"));
          lib.addComponent(new File(assetRoot, "MyButton.mxml"));
           lib.addComponent(new File(assetRoot, "MyLabel.mxml"));

            lib.build(true);
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}
```

You can also use a Configuration object, and call its `addSourcePath()` method to add the assetRoot to the source path. You then call the Library's `addComponent()` method, and pass it just the component class names, as the following example shows:

```
Configuration c = lib.getDefaultConfiguration();
c.addSourcePath(new File(assetRoot));
lib.setConfiguration(c);
lib.addComponent("MyButton");
lib.addComponent("MyLabel");
```

This is the equivalent of using the `include-classes` and `source-path` component compiler options to add classes that are in the source path to the SWC file.

Creating projects

As with the Application class, the Library class implements the Builder interface. So, you can use Logger, Report, and Configuration classes with the Library objects in the same way that you use these classes with an Application object.

To create a project, you define one or more Application and Library objects. You add these to your project by using the `addBuilder()` method.

If you create a library SWC file and use that SWC file in an application that you also create by using a Project class, use the `Project.dependsOn()` method. This method ensures that the compiler finishes creating the Library before it attempts to create the Application. You can use any number of `dependsOn()` methods to ensure the proper dependencies in the project are established before compiling the Application.

When creating projects that include libraries, you generally call the `addLibraryPath()` method to ensure that the compiler can find the library file during compilation.

The following example creates a project that consists of one application file and one library file. This example creates the Library object and ensures that the Library object exists before trying to create the Application object by using the `dependsOn()` method.

```
// java/MyProjectCreator.java
import flex2.tools.oem.Application;
import flex2.tools.oem.Project;
import flex2.tools.oem.Configuration;
import flex2.tools.oem.Library;
import java.io.*;

public class MyProjectCreator {
    public static void main(String[] args) {
        String assetRoot = "../assets/";
        String outputRoot = "../apps/";

        try {
            Project project=new Project();

            // Create the Application.
            Application app = new
                Application(new File(outputRoot,
                "TestAppWithComponents.mxml"));
            app.setOutput(new File(outputRoot,
                "TestAppWithComponents.swf"));
            app.setLogger(new ComplexLogger());

            // Create the Library.
            Library lib=new Library();
            lib.setOutput(new File(assetRoot, "MyComponents.swc"));
            lib.addComponent(new File(assetRoot, "MyButton.mxml"));
          lib.addComponent(new File(assetRoot, "MyLabel.mxml"));
            lib.setLogger(new ComplexLogger());

            // Add the new SWC file to the library-path.
            Configuration config = app.getDefaultConfiguration();
            config.addLibraryPath(new File[]
                {new File(assetRoot, "MyComponents.swc")});
            app.setConfiguration(config);

            // Add Application and Library objects to the Project.
            project.addBuilder(app);
            project.addBuilder(lib);

            // Ensure that the Library is created before trying
            // to compile the Application.
            project.dependsOn(app, lib);

            project.build(true);
        } catch (Exception ex) {
```

```
            ex.printStackTrace();
        }
    }
}
```

This example uses the following files and locations:

```
./java/MyProjectCreator.java
./assets/MyButton.mxml
./assets/MyLabel.mxml
./apps/TestAppWithComponents.mxml
```

This example creates the following files:

```
./assets/MyComponents.swc
./apps/TestAppWithComponents.swf
```

CHAPTER 4

Creating Dynamic Applications

4

You can create dynamic applications with the Flex compiler API.

Contents

About the creation of dynamic applications

The compiler API includes the capability to create application and library source files at run time and to compile these files into applications and libraries. To do this, you use the VirtualLocalFile and VirtualLocalFileSystem classes in the flex2.tools.oem package.

The process for creating a Flex application entirely within the Java class is as follows:

1. Create a String that stores the MXML source code. If you are creating a standard Flex application, begin with the XML declaration and opening `<mx:Application>` tag (for example, `<?xml version='1.0'?><mx:Application xmlns:mx='http://www.adobe.com/2006/mxml'>`), and end with a closing `</mx:Application>` tag.
2. Create a VirtualLocalFile by calling the `VirtualLocalFileSystem.create()` method. Pass the following parameters to the `create()` method:

   name Specifies the canonical path name of the target directory, and a virtual filename. The actual name is not usually important, but it must be in the correct location and it must have a .as or .mxml extension. The name is important when you create a custom component that you use in your virtual application.

   text Specifies the String that stores all the MXML source code for application.

   parent Specifies the canonical filename for the target directory.

lastModified Sets the value of the timestamp on the new virtual file. This is usually the current date and time.

3. Create a new Application object, passing the new VirtualLocalFile object as its parameter.
4. Define the output file for the Application object.
5. Call the `Application.build()` method.

The following example creates an application that has a simple Button control:

```
// java/MyVirtualFileCompiler.java
import flex2.tools.oem.*;
import java.io.*;

public class MyVirtualFileCompiler {
    public static void main(String[] args) {
        try {
            File parent = new File(".").getCanonicalFile();

            String src = "
                <?xml version='1.0'?>
                <mx:Application xmlns:mx='http://www.adobe.com/2006/mxml'>
                    <mx:Button label='Click Me'/>
                </mx:Application>";

            VirtualLocalFileSystem fs = new VirtualLocalFileSystem();
            VirtualLocalFile lf =
                fs.create(new File(parent,"VirtualApp.mxml").
                getCanonicalPath(), src, parent,
                    System.currentTimeMillis());

            Application app = new Application(lf);

            app.setOutput(new File("VirtualApp.swf"));

            app.build(true);
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}
```

Using custom components

You can create an application that uses existing custom components. You can also create the custom components dynamically. To do this, you create multiple VirtualLocalFile objects, and pass them as an array of files to the Application object's constructor.

The following example creates two custom components and adds them to the application:

```
// java/MyVirtualFilesCompiler.java
import flex2.tools.oem.*;
import java.io.*;

public class MyVirtualFilesCompiler {
    public static void main(String[] args) {
        try {
            VirtualLocalFileSystem fs = new VirtualLocalFileSystem();
            File parent = new File(".").getCanonicalFile();

            String main = "
                <?xml version='1.0'?>
                <mx:Application xmlns:mx='http://www.adobe.com/2006/mxml'
                    xmlns:comp='*'>
                    <comp:MyCustomLabel/>
                    <mx:Button label='Click Me'/>
                    <comp:MyCustomButton/>
                </mx:Application>";

            String comp1 = "
                <mx:Button
                    xmlns:mx='http://www.adobe.com/2006/mxml'
                    label='Click Me Too'/>";

            String comp2 = "
                <mx:Label
                    xmlns:mx='http://www.adobe.com/2006/mxml'
                    text='Custom Label'/>";

            VirtualLocalFile vlfComp1 = fs.create(new
                File(parent,"MyCustomButton.mxml").getCanonicalPath(),
                comp1, parent, System.currentTimeMillis());
            VirtualLocalFile vlfComp2 = fs.create(new
                File(parent,"MyCustomLabel.mxml").getCanonicalPath(),
                comp2, parent, System.currentTimeMillis());
            VirtualLocalFile vlfMain = fs.create(new
                File(parent,"ComplexVirtualApp.mxml").getCanonicalPath(),
                main, parent, System.currentTimeMillis());

            // The order of arguments here matters. You must create
            // the components before you can create the application
            // that uses those components.
            Application app = new Application(new
                VirtualLocalFile[] {vlfComp1, vlfComp2, vlfMain});
            app.setLogger(new ComplexLogger());

            app.setOutput(new File("ComplexVirtualApp.swf"));
            app.build(true);
```

```
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}
```

When creating an application or library that uses custom components, keep in mind the following details:

- The name you provide for the name parameter of the `create()` method is important. It must match the name that you use in the application. For example, if you refer to a component as `<comp:MyCustomComponent>`, the name must be "MyCustomComponent", as the following example shows:

  ```
  VirtualLocalFile vlfComp1 = fs.create(new
    File(parent,"MyCustomComponent").getCanonicalPath(), comp1, parent,
    mod);
  ```

- The order of creation for components and applications is important. If your application uses custom components or other files that are created at the same time, you must create those components before you create the application. To ensure this, list the components before the application in the Application object's constructor, as the following example shows:

  ```
  Application app = new Application(new VirtualLocalFile[] {vlfComp1,
    vlfComp2, vlfMain});
  ```

Using incremental compilation

You can use incremental compilation with VirtualLocalFile objects. This makes compilation more efficient because the compiler only recompiles the virtual file that was changed and not all files that are in the application. You typically use incremental compilation in a Java application that builds a Flex application more than once. The first time the Flex application is built, the compiler compiles all parts of the application. On subsequent compilations, if you use incremental compilation, the compiler only compiles the parts of the application that changed.

To trigger the use of incremental compilation, use the VirtualLocalFileSystem class's `update()` method to change a virtual local file. Then call the `build()` method again to compile the application incrementally.

For more information about incremental compilation, see *Building and Deploying Flex 2 Applications*.

The following example calls the `changeComponent()` method and makes a change to one of the VirtualLocalFile objects in it. The application then calls the `fs.update()` method, which updates only that virtual file with the new code. The `build()` method ultimately takes less time to compile the application because only the changed code is recompiled.

```
// java/MyIncrementalCompiler.java
import flex2.tools.oem.*;
import java.io.*;

public class MyIncrementalCompiler {

    static Application app;
    static VirtualLocalFileSystem fs;
    static File parent;

    public static void main(String[] args) {
        try {
            fs = new VirtualLocalFileSystem();
            parent = new File(".").getCanonicalFile();

            String main = "
                <?xml version='1.0'?>
                <mx:Application
                    xmlns:mx='http://www.adobe.com/2006/mxml'
                    xmlns:comp='*'
                >
                    <comp:MyCustomLabel/>
                    <mx:Button label='Click Me'/>
                    <comp:MyCustomButton/>
                </mx:Application>";

            String comp1 = "
                <mx:Button
                    xmlns:mx='http://www.adobe.com/2006/mxml'
                    label='Click Me Too'/>";

            String comp2 = "
                <mx:Label
                    xmlns:mx='http://www.adobe.com/2006/mxml'
                    text='Custom Label'/>";

            VirtualLocalFile vlfComp1 =
                fs.create(new File(parent,"MyCustomButton.mxml").
                getCanonicalPath(), comp1, parent,
                System.currentTimeMillis());

            VirtualLocalFile vlfComp2 =
                fs.create(new File(parent,"MyCustomLabel.mxml").
                getCanonicalPath(), comp2, parent,
                System.currentTimeMillis());
```

```
            VirtualLocalFile vlfMain =
                fs.create(new File(parent,"ComplexVirtualApp.mxml").
                getCanonicalPath(), main, parent,
                System.currentTimeMillis());

            // The order of arguments here matters. You must create
            // the components before you can create the application
            // that uses those components.
            app = new Application(new VirtualLocalFile[] {vlfComp1,
                vlfComp2, vlfMain});
            app.setLogger(new ComplexLogger());

            app.setOutput(new File("IncrementalVirtualApp.swf"));
            app.build(true);

            // Calling changeComponent() updates one of the virtual files,
            // which lets you then use the update() method to use
            // incremental compilation.
            changeComponent();

        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }

    public static void changeComponent() {
        try {
            String newComp = "
                <mx:Button xmlns:mx='http://www.adobe.com/2006/mxml'
                label='Dont Click Me'/>";
            fs.update(new File(parent,"MyCustomButton.mxml").
                getCanonicalPath(), newComp, System.currentTimeMillis());
            app.build(true);
        } catch (Exception ex) {
            ex.printStackTrace();
        }
 }
}
```