C H A P T E R 1

Using the JavaConverter

The JavaConverter is a command line tool that performs automatic conversion of any Objective-C or WebScript source code to pure Java code. This can be done file by file or at the project level for any WebObjects 4.5 application or framework. The JavaConverter runs under Mac OS X Server 1.2 and Windows NT.

JavaConverter Features

The JavaConverter is robust:

- It only works on a copy of your WebObjects project; the original is never touched. (When processing an individual file, the original can get overwritten; always create a copy of your project before processing individual files.)
- It deals with categories either by creating helper classes or by merging them throughout subprojects. You will need to manually correct the calls to these categories after the automatic conversion has been performed.
- It preserves comments and generates clean, easy to read Java code. Any problems encountered by the parser are highlighted with a searchable comment (tagged `JC_ERROR`, `JC_WARNING`, or `JC_INFO`, depending on the severity of the problem).
- To the greatest extent possible, it fixes WebScript weak type references.
- If the parser ever crashes while attempting to process a particular Objective-C construct, an exception is raised indicating at which line the error occurred, after which parsing continues.

- In the event that the parser crashes, you can fix the original source and re-parse the problem files one at a time.
- This release of the JavaConverter has been tested against all of our major Objective-C frameworks (WebObjects, EOControl, EOAccess).

The JavaConverter is extensible simply by dropping `tops` scripts into designated directories (these are all located in the /Library/WebObjects/Java/Conversion directory):

- **PreScriptsObjCWos:** `tops` scripts executed on every `.c`, `.m`, or `.wos` file before their conversion.
- **PostScriptsBridgedJavaToPure:** `tops` scripts executed on every `.java` file after their conversion (unless `-toBridgedJava` is specified).
- **PostScriptsD2WModelToPure:** `tops` scripts executed on every `.d2wmodel` file after their conversion (unless `-toBridgedJava` is specified).
- **PostScriptsWodToPure:** `tops` scripts executed on every `.wod` file after their conversion (unless `-toBridgedJava` is specified).
- **PostScriptsObjCWosToPure:** `tops` scripts executed on every `.c`, `.m`, or `.wos` file after their conversion (unless `-toBridgedJava` is specified).
- **PostScriptsAllToPure:** `tops` scripts executed on every `.c`, `.m`, `.wos`, or `.java` file after their conversion (unless `-toBridgedJava` is specified), after all other scripts.
- **PostScriptsObjCWosToBridged:** `tops` scripts executed on every `.c`, `.m`, or `.wos` file after their conversion (only if `-toBridgedJava` is specified).

Note the following:

- Within each of the above directories, scripts are run in alphabetical order.
- The order in which `replace` commands are executed by `tops` is important. Do not move them around in the scripts provided with JavaConverter.
- The `tops` man pages have more information about `tops` scripts.

It's important to understand that although the JavaConverter does the majority of the work needed to convert your code, it does not—in fact, cannot—perform a complete conversion. Thus, the converted code will not compile until you have reviewed it and made a number of manual adjustments.

Format Hints to the Cross-Compiler

The JavaConverter is savvy enough to recognize some printing methods that take formatting information and at least partially convert them into Java. For instance, if in your original code you had something like

```
fprintf(stderr, “%s has visited this site %d times.”, userName, userHitCount
);
```

the JavaConverter would recognize the fprintf call and would generate

```
System.err.println( userName + “ has visited this site “ + userHitCount + “
times.” );
```

Clearly, some tidying up may still be needed, but the chore of cleaning up the formatting is completed for you. The JavaConverter can only do these conversions if it recognizes the method names (`printf`, `fprintf`, `stringWithFormat`, and `NSLog`, for example). If, however, you implement a method named `superLog`, for example, that takes style formatting arguments like `printf` does, it would be very convenient to have the JavaConverter appropriately convert uses of this method as well. In order to render your methods visible to the JavaConverter (for the purpose of having the style formatting conversions performed), you can have a `tops` script located in the PreScriptsObjCWos directory that has a replace line to convert instances of `superLog` to `superLog_formatted`. The extra "_formatted" is picked up during conversion and the following:

```
superLog_formatted( “%d transactions processed.”, transactionCount );
```

would, after conversion, become

```
superLog( transactionCount + “ transactions processed.” );
```

Note that the JavaConverter also removes the "_formatted" part of the method name for you automatically. Additionally, if the streams involved in these calls apply to `stderr` or `stdout`, they will be converted into calls to `System.err.println` and `System.out.println`, respectively. However, if, for example, an `fprintf` stream is attached to something different, you would see the following error message:

```
JC_ERROR - Unable to convert fprintf() as it applies to an unrecognized stream
(neither stdout or stderr).
```

Limitations

Naming Booleans

The JavaConverter may incorrectly translate the Objective-C code

```
BOOL firstTime = YES;
...
if( firstTime )
```

into the Java code

```
boolean firstTime = true;
...
if( firstTime != null ) // “if( !firstTime )” is desired
```

Note that this translation is not the desired one. JavaConverter does not recognize the types of variables and will treat each of them as an Object.

More on Conditionals

In Objective-C, the following makes sense and is something of a time-saver:

```
if( ![value length] )
```

However, the precise meaning of this conditional is somewhat ambiguous and might be

1. `if( [value length] == 0 ) // int, but could be double, float, and so on`
2. `if( [value length] == nil )`
3. `if( [value length] == NULL )`
4. `if( [value length] == FALSE )`

Regardless, a translation cannot be performed correctly for all of the cases, as the type returned by the `length` method cannot simply be assumed. The default behavior for the JavaConverter in this case is to generate the code

```
if( value.length() == null )
```

This correctly covers cases 2 and 3; additionally, case 4 could be handled by naming the methods beginning with “is”, “are”, “was”, “were”, “has”, “have”, “equals”, “does”, “ends”, “starts”, or “bool”. Case 1 cannot be handled properly by the JavaConverter.

Messages to nil

While Objective-C allows for messages sent to `nil` (which are simply ignored), Java’s typing and lack of messaging thoroughly disallow behavior like this: methods called on `null`-valued variables of nonprimitive types always raise NullPointerExceptions in Java. The JavaConverter has no mechanism by which to positively identify when messages in Objective-C code raise NullPointerExceptions after conversion to Java — in any case, frequently this is simply an issue discovered at runtime. Please be aware of this conversion issue and attend to your converted Java code accordingly.

Format Hints to the Cross-Compiler

The JavaConverter has the following limitations:

Due to the fact that certain language conventions available in Objective-C do not translate well (or at all) into Java, there are limitations to the JavaConverter’s abilities. Some of these sticking points can be gotten around or somehow avoided and others cannot. A few issues to keep in mind about the translation process are listed below:

- C code parsing is handled correctly, but this is not a C-to-Java converter. Structures are essentially converted to Object, but pointers to functions and other C constructions that make no sense in Objective-C are ignored.
- C libraries, C functions, UNIX functions, MFC (Microsoft Foundation Class) functions, Objective-C runtime functions, and embedded assembly code in Objective-C are not converted.
- Java is a strongly typed language, and moving from a less strongly typed language to Java is bound to lead to uncertainties. At this point, the converter is not able to do type introspection to resolve unknown types. Some types may be omitted when Java requires them, and some checks for `null` value may be missing in the Java classes after conversion. Specifically, when the JavaConverter converts `if([receiver method:target])` to

`if(receiver.method(target)!=null)`, any method whose name starts with “is”, “are”, “was”, “were”, “has”, “have”, “had”, “equals”, “does”, “ends”, “starts”, or “bool” is assumed to return a boolean.

- The `@asm` and `@defs` constructs aren’t handled. All other @ constructs, however, are.
- The converter does not preserve the original indentation of the code and may move comments from one line to another.
- The converter cannot analyze the code with much depth and thus works best with straightforward code.

Running the JavaConverter

You should only run the JavaConverter on projects or files that successfully compile under WebObjects 4.5.

On Mac OS X Server 1.2, go to the `JavaConverter.woa` directory and invoke the converter as follows:

```
./JavaConverter [-toBridgedJava] [-header file-containing-headers] [-cppFlags
file-containing-cpp-flags] project-directory | Objective-C-source-file |
bridged-Java-source-file | WebScript-file
```

On Windows NT, go to the `JavaConverter.woa` directory and invoke the converter as follows:

```
./JavaConverter.cmd [-toBridgedJava] [-header file-containing-headers] [-cppFlags
file-containing-cpp-flags] project-directory | Objective-C-source-file |
bridged-Java-source-file| WebScript-file
```

In both of the above, *project-directory* is the path to a WebObjects project directory (the directory that contains the `PB.project` file).

On Mac OS X perform the following steps:

1. Start Project Builder.

   The Project Builder application is located in the `/System/Developer/Applications` directory.

2. Choose File > Import Project.
3. Select Import PB.project in the Project Import Assistant window and click Next.
4. Enter the necessary project information and click Finish.

   Project Builder automatically creates your new project.

The `-toBridgedJava` flag indicates whether you want to stop the conversion at the bridged Java stage. At this stage, the code can still execute on WebObjects 4.5.

The `-header` command line option lets you specify the name of a file that contains extra comments, package specifications, or import information that you want to add to each source file that gets converted. An example file might be:

```
/* Copyright Apple Computer, 2000 - owner fjouaux */
package com.webobjects.javaconverter;
import com.webobjects.foundation.*;
```

The `-cppFlags` command line option lets you specify the name of a file containing extra arguments that the JavaConverter passes to the cpp preprocessor when converting Objective-C source files. Be sure to specify these arguments on a single line in the file. See the man pages for `cpp` for a list of valid preprocessor flags.

When you invoke the JavaConverter on a project, it does all of its work in a copy of the specified project; the original is not touched. When it is first invoked on a WebObjects project, the JavaConverter copies the entire project directory to a parallel directory named after the original with the word "Java" appended. Thus, if you run the JavaConverter on the LocalizedHelloWorld example, the JavaConverter first copies the entire project to "LocalizedHelloWorldJava". Then it

- scans the `LocalizedHelloWorldJava` directory and read the `PB.project` file
- updates the `PB.project` file
- runs the various `tops` scripts in the `PreScriptsObjCWos` directory
- scans each `.h` file, each `.m` file, and each `.wos` file
- generates a set of `.java` files, one for each `.h/.m` pair and one for each `.wos` file
- runs the various `tops` scripts in the `PostScriptsObjCWosToBridged` directory

When the JavaConverter is invoked on a single source file, instead of making a copy of the entire project, it creates a `ppFiles` directory within the project and does its work there. The resulting `.java` file, however, appears in the project alongside the original `.h`/`.m` or `.wos` file. If you are converting a single bridged Java file to pure Java, the converted file appears in a directory called `.pureJava`.

WARNING
The JavaConverter *will* overwrite a preexisting `.java` file if its name corresponds to an existing `.h`/`.m` or `.wos` file. For instance, if your project contains both `Foo.m` and `Foo.java`, when you run the JavaConverter on `Foo.m` it will overwrite `Foo.java`.

WARNING
The JavaConverter may fail silently if it cannot parse the source code. Always make sure that your code compiles under WebObjects 4.5 before you convert it.

Regenerating the Project Makefile

Because the source files have changed (from Objective-C to Java, say), the makefile is no longer applicable. As part of its work when processing an entire WebObjects project, the JavaConverter deletes the makefile in the copy of the project directory being processed. After the JavaConverter is done, you'll need to have ProjectBuilder create a new makefile for the project. Here's how to do this on Mac OS X Server 1.2:

1. Using Project Builder, open the `PB.project` file in the Java project directory.
2. Select Classes. In the classes list, Control-drag one of the `.java` files to a different location in the list (this changes the build order, indicating to ProjectBuilder that the makefile needs to be regenerated).
3. Save the project.
4. Perform a "make clean".

On Windows NT, the following procedure will cause Project Builder to generate a new makefile:

1. Using Project Builder, open the `PB.project` file in the Java project directory.
2. Select any project file and remove it from the project (only—don't remove it from the disk!).

3. Add the file deleted in the previous step back into the project.
4. Save the project.
5. Perform a “make clean”.

After Conversion

Once the JavaConverter has done its work, you’ll need to inspect the generated Java source code and make manual adjustments in order to get the code to compile. Simply attempting to compile the resulting code is one way to quickly pinpoint many problem areas. It is also important to browse through the generated code and look for comments tagged `JC_ERROR`, `JC_WARNING`, or `JC_INFO`; these messages indicate areas where the JavaConverter may have had difficulty determining exactly what to generate.

Dealing With Errors

In normal operation, the JavaConverter sends messages similar to the following to the terminal window:

```
JavaConverter /tmp/PitSaw
Reading MacOSXServerClassPath.txt ...
Launching JavaConverter ...
java -classpath “/tmp/JavaConverter/Resources/Java/javaconverter.zip:/
System/
Library/Frameworks/JavaVM.framework/Classes/classes.jar” Application
-toBridgedJava /tmp/PitSaw
*********** JavaConverter To Bridged Java Only ***********
JavaConverter Version 0.9:  Copied Project directory to: /tmp/PitSawJava
JavaConverter Version 0.9:  Converting file: /tmp/PitSaw/PB.project
JavaConverter Version 0.9:  Converting file: /tmp/PitSawJava/Application.h
Preprocessed file is ‘/tmp/PitSawJava/ppFiles/Application.h’
There were at least 0 errors during parsing & porting.
```

Occasionally, while running the JavaConverter on a file or project, it may suddenly output a long backtrace in the terminal. This indicates that the parser failed to entirely convert the current file. Although the JavaConverter’s parser has been made as reliable as possible, we cannot anticipate all possible source code constructs which may be in use. Such problem files will be converted only to the point where the failure occurred. All other files will be converted independently of this error (if you are converting a project). You can then fix each problem file and reconvert just those files.

To guide you in identifying the source of conversion problems, should save the output log of your conversion. This can be accomplished by adding an output-redirection command to the end of the converter's launch arguments; on a Unix-based platform this addition would be something like `| tee $HOME/java.log` and on a Windows platform this addition would be like `| tee %HOME%\java.log` (of course, any valid path may be specified). Note that on Windows the `JavaConverter.cmd` executable opens a new window without a scrollbar that closes at the end of the conversion process — redirecting the output log to a file is especially helpful in this case. In the event of an error, if you scroll through this log, before the backtrace there is a statement that ends "Encountered errors during parse". Above this statement is a line that tells you which file was being converted (note that this is not the original file, but a preprocessed file under a '`ppFiles`' directory). For example, here is a portion of a JavaConverter output log:

```
JavaConverter Version 0.9: Converting file:
/tmp/JavaConverter/Test.m
Preprocessed file is
'/tmp/JavaConverter/ppFiles/Test.m'

JavaConverter Version 0.9: Encountered errors during parse.
com.webobjects.javaconverter.ParseException: Encountered "{" at line 60,
column 30.
Was expecting one of:
```

When investigating the source of the error, be sure to check the version of the file in the `ppFiles` directory—the line and column numbers aren't likely to indicate the correct spot in the original source file. When altering your source code in an attempt to work around the error, however, you'll need to make your corrections in the original source file—not the version in the `ppFiles` directory. Once you have made your corrections you can again try to convert the problem files.

Typically, parsing errors come from macros that could not be resolved by the preprocessor. This often occurs when the project being converted would not compile on the machine because the preprocessor does not know where to find the header files needed to resolve the macros. To specify header files, use the `-cppFlags` command line option.

FileNotFoundException

When processing either a single file or an entire WebObjects project, the JavaConverter will raise a FileNotFoundException if either the file being processed or the project directory itself is read-only. When this error occurs during the processing of an entire project, the original read-only source file will also remain in the converted project directory.

To avoid this problem, make sure that the project directory and all files and directories within it are writable. Note that there is no risk in making the files writable since a copy of the project is made before conversion and all processing takes place only in the copy.

Errors During WebScript Conversion

A "WEBSCRIPT FILE CONVERSION FAILED" message indicates that the JavaConverter doesn't understand the WebScript file's syntax and cannot proceed. To help you to isolate the problem code, the JavaConverter logs the exact command it executed to convert the WebScript file. Cut and paste this command into a terminal window to see the exact reason for the exception. Guided by the information provided, correct the WebScript file and restart the JavaConverter.

Note: As explained in "How the JavaConverter Works" (page 12), WebScript is first converted to "pseudo" Objective-C, which is then converted to bridged Java. This "pseudo" Objective-C isn't sufficient for compilation—it doesn't invoke `autorelease` or `retain`, for instance—and is only intended to be converted to Java.

Failures during WebScript conversion are extremely rare (assuming that your WebScript file is legal). There is currently only one known WebScript construct that can bring about such a failure: an inlined array or dictionary definition. For example, the following will cause the JavaConverter to fail:

```
list = @( { "label" = "Alpha"; "value" = "A";), {...});
```

The addition of two additional '@' characters, as shown here, will clarify things and enable the JavaConverter to correctly interpret the code:

```
list = @( @{ "label" = "Alpha"; "value" = "A";), @{...});
```

How the JavaConverter Works

The JavaConverter contains a Java-based tool, which first duplicates the project being converted and then—in the duplicate—works on each class and header file found in the project and its subprojects.

The Objective-C-to-Java conversion itself is based on javacc, SUN's parser generator. By supplying an Objective-C grammar to this parser, we were able to create what is essentially an Objective-C compiler that generates non-compiled Java classes. Objective-C files are first preprocessed using the gnu preprocessor to resolve macros and headers, then the code is parsed into a graph of objects and regenerated in Java.

The WebScript-to-Java conversion is done in two steps. A separate Objective-C tool based on the MultiScript framework (WebScript's engine) parses scripted files and generates pseudo Objective-C classes. We then feed this intermediate code to the Objective-C-to-Java parser to generate Java classes.

Bridged Java class files are left untouched (except by the `tops` scripts), so the conversion process can be run on WebObjects applications and frameworks that are written in a mixture of Objective-C, WebScript, and bridged Java.

Finally the project is reconstructed, adding each class—and, if, necessary, new helper classes—back to the appropriate project and subprojects.

Conversion Gotchas

Hiding Code from the JavaConverter

In some cases (for instance, if you know beforehand that a conversion may be handled incorrectly), it can be advantageous to prevent the JavaConverter from attempting a conversion on its own. Since comments in the original code are preserved, commenting-out blocks of problematic code can allow you to completely handle the conversion on your own with pre- and post-conversion scripts. Using a custom comment-marker (for instance, perhaps /** and **/) to shield a block from the JavaConverter would leave a simple hook for custom tops scripts to recognize.

EOEditingContext Timestamp Datatypes

In WebObjects 4.5, EOEditingContext's timestamp methods represented times as doubles in units of seconds. For instance

```
// WebObjects 4.5 EOEditingContext timestamp-related method signatures
public static double defaultFetchTimestampLag();
public static void setDefaultFetchTimestampLag(double);
public double fetchTimestamp();
public void setFetchTimestamp(double);
```

However, in WebObjects 5, the analogous methods now represent times as longs in units of milliseconds:

```
// WebObjects 5 EOEditingContext timestamp-related method signatures
public static long defaultFetchTimestampLag();
public static void setDefaultFetchTimestampLag(long);
public long fetchTimestamp();
public void setFetchTimestamp(long);
```

While the use of a double or a long with these methods allows the programmer to discern unambiguously with which version of EOEditingContext one is dealing, the JavaConverter has no mechanism by which to account for the three orders-of-magnitude difference in the units of these timestamps (seconds in WebObjects 4.5 vs. milliseconds in WebObjects 5). Therefore, it is important that you take special care to account for the increased precision of these methods after a conversion.