Using the Java Bridge

In the worlds of Mac OS X, Yellow Box for Windows, and WebObjects programming, there are two languages in common use: Java and Objective-C. This document describes the Java bridge, a technology from Apple that makes communication between these two languages possible.

The first section, "Introduction," gives a brief overview of the bridge's capabilities.

For a technical overview of the bridge, see "How the Bridge Works" (page 2).

To learn how to expose your Objective-C code to Java, see "Wrapping Objective-C Frameworks" (page 9).

If you want to write Java code that references Objective-C classes, see "Using Java-Wrapped Objective-C Classes" (page 6). If you are writing Objective-C code that references Java classes, read "Using Java from Objective-C" (page 5).

Introduction

The original OpenStep system developed by NeXT Software contained a number of object-oriented frameworks written in the Objective-C language. Most developers who used these frameworks wrote their code in Objective-C.

In recent years, the number of developers writing Java code has increased dramatically. For the benefit of these programmers, Apple Computer has provided Java APIs for these frameworks: Foundation Kit, AppKit, WebObjects, and Enterprise Objects. They were made possible by using techniques described later in this document. You can use these same techniques to expose your own Objective-C frameworks to Java code.

Java and Objective-C are both object-oriented languages, and they have enough similarities that communication between the two is possible. However, there are some differences between the two languages that you need to be aware of in order to use the bridge effectively. In general, using Objective-C from Java requires a bit more work on the part of the programmer than the reverse process.

Once the proper setup has been done, you can:

- Instantiate and use classes from one language in the other.
- Pass objects as arguments and receive objects as return values from one language to the other.
- Directly subclass Objective-C classes in Java.
- See Objective-C protocols as Java interfaces.

How the Bridge Works

The term *bridge* refers to the fact that there is a connection between two different worlds: the Objective-C world and the Java world. To use the bridge effectively, it's helpful to understand what's happening on each side of the bridge.

The Java bridge allows you to write code in one language that references an object from the other language. For example, you can send a message to a Java object from Objective-C:

```
myInt = [myJavaObject numberOfItems];
```

And you can send a message to an Objective-C object from Java:

```
myInt = myObjCObject.numberOfItems();
```

The bridge uses two basic approaches to allow this type of communication:

- Objects of certain commonly used classes are transparently converted, or *morphed*, into a corresponding object on the other side of the bridge. An example would be using a Java String object in a method call that expects an Objective-C NSString. See "Morphing" for more information.

- For all other objects, the bridge builds a *proxy* object that references the actual object on the other side of the bridge that is being messaged.

Exposing Objective-C classes to Java code requires a bit more work than the converse. Because Java is a strongly typed language, it requires more information about the classes and interfaces it manipulates at compile time. Therefore, before using Objective-C classes as Java ones, a description of them has to be written and compiled. This is called *wrapping* the Objective-C classes. The wrapping process involves creating a *Java to Objective-C Bridging Specification* (`.jobs`) file—either with the WrapIt application or by hand—and processing it with a tool called `bridget`. See "Wrapping Objective-C Frameworks" (page 9) for more information.

Java has a facility called the Java Native Interface, which allows a Java programmer to write methods that are implemented in another language (such as C or Objective-C). The Java bridge makes use of this capability to allow you to send messages from Java to Objective-C:

- It provides a native method for each Objective-C method that is being wrapped.
- It generates stub code that dispatches to the Objective-C method.

Suppose you have an Objective-C framework containing a class called Foo with a method called `ping`, as follows:

```
@interface Foo:NSObject
     - (void) ping;  //the actual Objective-C method
@end
```

To use this method from Java, you specify in your `.jobs` file that you want to expose the Foo class, and that its corresponding Java name should be Foo (you must also specify the package name, for example, `com.yourFirm.whatever.Foo`). You also specify that you want to expose the `ping` method (and its Java name, if different).

The `bridget` tool generates a native method declaration (in `Foo.java`) that looks like this:

```
public class Foo() extends com.apple.yellow.foundation.NSObject {
    public native void ping();  //native method declaration
}
```

The `native` keyword indicates that the code for the method is in a language other than Java.

`Bridget` also generates a file containing *stub* code, which is the actual code that gets called on the Objective-C side. This stub code lives in a file called *packageName_className_* `stubs.m`. By convention, the stub routine for the native method has the name *Java_className_methodName*. The following code illustrates what such a stub routine might look like:

```
void Java_Foo_ping(JAVAHandle object) {
    [BRIDGEJavaHandleToId(object) ping]; //the Objective-C method call
}
```

Note: `JAVAHandle` is the way the bridge points to a Java object. `BRIDGEJavaHandleToId` is a function that converts the Java handle to an Objective-C id. These examples are intended to give a sense of how the bridge works. Normally, you don’t need to understand these internal bridge data types in detail, unless you are writing morphing conversion routines, for example. In that case, you can refer to the bridge header files (located in `/System/Developer/Java/Headers/`) for more information.

To call this method from Java, you’d write code such as the following:

```
myFooObject = new Foo();
myFooObject.ping();
```

When you instantiate a Foo object, the Java bridge creates a proxy object on the Java side and a “real” object on the Objective-C side, which is the receiver of any messages that you send to it.

Morphing

There are some frequently used classes for which the Java bridge supplies automatic conversion functions, so that you don’t need to provide a wrapper. When you use an instance of one of these classes, it is transparently converted (or *morphed*) into the corresponding class on the other side of the bridge.

For example, suppose the `ping` method took an NSString argument:

```
@interface Foo:NSObject
     - (void) ping:(NSString *) theString);
@end
```

When you call this method from Java, you pass in a Java String object:

```
myFooObject.ping(myString);
```

The stub code looks something like this:

```
Java_Foo_ping(JAVAHandle object, JAVAHandle str) {
    [BRIDGEJavaHandleToId(object) ping:JavaStringToNSString(str)];
}
```

The first argument to this function points to the object to which the message is being sent (that is, `myFooObject` on the Java side). The second argument points to the Java String object `myString`. JavaStringToNSString is a function provided by the bridge that converts a Java String to an NSString. Its inverse is NSStringToJavaString. This means that you can use these types on both sides of the bridge, and the conversion takes place automatically.

You may want to provide your own conversion functions that transparently morph classes. For example, you might want to expose a C `struct` as a Java class. See “Editing the Specification File” (page 12) for information on how to specify them.

Other classes that are morphed include Number (NSNumber) and Exception (NSException). In the case of NSException, this means that your Java code can handle exceptions as it normally would, regardless of whether the exception might have arisen in an Objective-C method.

Using Java from Objective-C

In Objective-C, you can use Java classes and interfaces as if they were Objective-C classes and protocols.

To reference class objects, you must use the function `NSClassFromString()`, since the Objective-C compiler can't generate static references to bridged Java classes. For example, to instantiate an object from the `AJavaClass` class, you would write code such as the following:

```
MyJavaClass = NSClassFromString(@"java.util.AJavaClass");
MyJavaObj = [[MyJavaClass alloc] init];
```

You can instantiate Java classes from Objective-C and see them as if they were pure Objective-C objects. You can also pass Objective-C objects as parameters to Java methods.

To invoke a Java method from Objective-C:

- If you've specified an explicit mapping between the selector name and a Java method name in a `.jobs` file, your specified Java method name is used.
- If you haven't mapped the selector name to a Java method name in a `.jobs` file, the bridge automatically derives the Java method name from the Objective-C selector name by removing everything beyond the first colon. Thus, a method such as `sendBar:x foo:y` would map to `sendBar(x,y)`

Exceptions raised within the Java code are caught and transformed into NSExceptions, which can then be handled by your code on the Objective-C side.

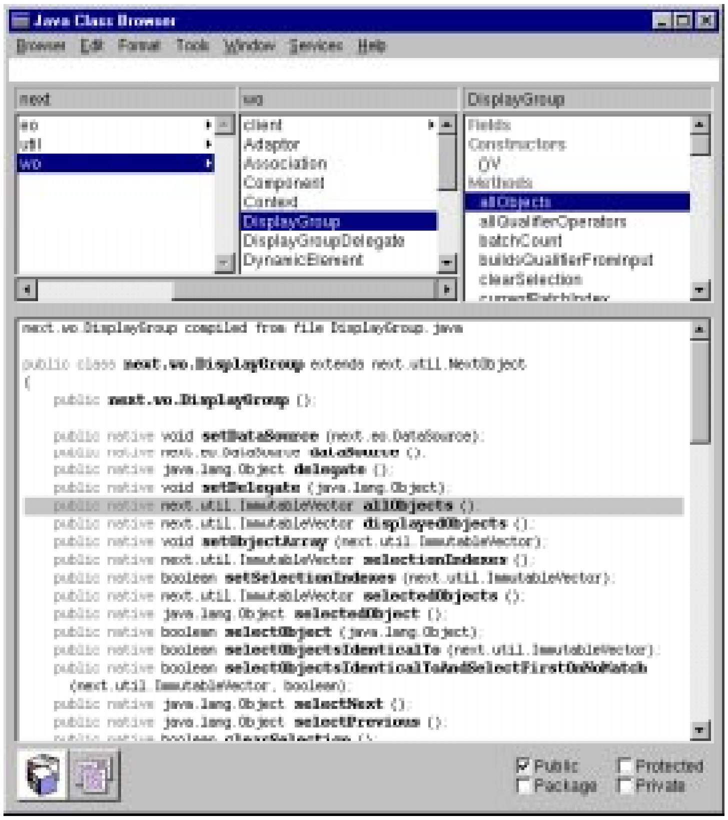
Using Java-Wrapped Objective-C Classes

This section describes how to use wrapped Objective-C classes from Java. The next section, "Wrapping Objective-C Frameworks," explains how to wrap Objective-C frameworks.

To get information about the classes in a wrapped framework, you can't look at header files as you would in Objective-C. Instead, you can:

- Look at documentation provided with wrapped frameworks from Apple, such as Foundation Kit, AppKit, WebObjects, and Enterprise Objects.
- Use the Java Class Browser. This application allows you to browse Java packages found in your `CLASSPATH` (an environment variable that specifies the search path for Java `.class` files. It lists the classes and the syntax of their methods, even for classes for which you don't have the source code.

Figure 1-1 Java Class Browser



```
next.wo.DisplayGroup compiled from file DisplayGroup.java

public class next.wo.DisplayGroup extends next.util.NextObject
{
    public next.wo.DisplayGroup ();

    public native void setDataSource (next.eo.DataSource);
    public native next.eo.DataSource dataSource ();
    public native java.lang.Object delegate ();
    public native void setDelegate (java.lang.Object);
    public native next.util.ImmutableVector allObjects ();
    public native next.util.ImmutableVector displayedObjects ();
    public native void setObjectArray (next.util.ImmutableVector);
    public native next.util.ImmutableVector selectionIndexes ();
    public native boolean setSelectionIndexes (next.util.ImmutableVector);
    public native next.util.ImmutableVector selectedObjects ();
    public native java.lang.Object selectedObject ();
    public native boolean selectObject (java.lang.Object);
    public native boolean selectObjectsIdenticalTo (next.util.ImmutableVector);
    public native boolean selectObjectsIdenticalToAndSelectFirstOnNoMatch
      (next.util.ImmutableVector, boolean);
    public native java.lang.Object selectNext ();
    public native java.lang.Object selectPrevious ();
```

Objective-C methods use keywords for method arguments, while Java uses comma-separated lists (as with C and C++). For example, in Objective-C, the following declares an instance method that takes two parameters:

```
-(void)setObject:(id)anObject forKey:(id)aKey;
```

In Java, the same method is declared as follows:

```
void setObjectForKey(Object anObject, Object aKey);
```

The Java names for methods and classes may or may not be the same as their Objective-C counterparts. It's up to the developer of the wrapped framework to decide how Objective-C names are exposed in Java. "Wrapping Objective-C Frameworks" (page 9) explains the rules and options for mapping names between the two languages.

For the most part, you can use the classes in a wrapped Objective-C framework just as you would use any Java class. The bridge transparently loads in any needed Objective-C framework whenever a bridged class is used. You can invoke Objective-C instance methods as you'd expect:

```
myInt = myObjCObject.aMethod();
```

Java objects can be passed back to the Objective-C world and manipulated by the code there as if they really were Objective-C classes.

You can invoke a class method by referencing the class name directly:

```
ObjCClassName.classMethod();
```

One restriction is that instance variables in Objective-C classes are not exposed in Java. The class must provide accessor methods that set and get their values.

You can subclass Objective-C classes in Java just like you can any other class:

```
class MySubClass extends MyObjCClass {
     ...
}
```

Again, the original class's instance variables aren't exposed; therefore, they aren't inherited. You can, however, add your own instance variables to the subclass.

As with other classes, you can call `super` from a Java subclass of an Objective-C class to invoke the Objective-C class's implementation.

In Objective-C, you create objects with `alloc` and initialize them with class initialization methods such as `init`. In Java, you create objects with `new` and initialize them with the class's constructor.

For example, to create an instance of the Foo class in Objective-C:

```
someObject = [[Foo alloc] init];
```

In Java:

```
someObject = new Foo();
```

In Objective-C, initialization methods are instance methods, applied to newly created objects. In Java, constructors are not used in the same way as instance methods. Therefore, Objective-C initialization methods must be explicitly specified as constructors when the classes are wrapped. If there is more than one initialization method, they become Java constructors with different argument types. See “Editing the Specification File” (page 12) for specifics on how to do this.

One difference between Java and Objective-C is memory management. In Java, unused objects are garbage-collected. In Objective-C, you must keep track of object references and release objects when they are no longer needed.

For the most part, you don’t need to worry about these differences. One possible occurs if you create a new thread in Java, and that thread manipulates some objects with Objective-C counterparts. In that case, you must explicitly create an autorelease pool at the beginning of the thread, and remove it at the end of the thread.

Wrapping Objective-C Frameworks

The Java bridge allows you to take existing Objective-C code and make it accessible to Java. To do so, follow these steps:

1. Determine which Objective-C entities (types, classes, methods, and protocols) you want to expose in Java.

   These entities must be packaged in a framework (or possibly more than one).

2. Identify a single header (`.h`) file that declares, either directly or indirectly through importing other headers, all the Objective-C entities that you want to expose. You’ll specify this header when you edit the `.jobs` file in step 6 below.

   This header file can be one that already exists in your framework. Alternatively, you can create a header yourself and have it import the other headers you need. In particular, you'll do this when you add extra code to your Objective-C classes.

3. Create a *Java Wrapper project* using Project Builder. See the next section for more information.
4. Add your framework to the project.
5. Optionally, add extra Objective-C code to your project. See “Modifying Your Code” (page 17) for information on when you might need to do this.
6. Edit the `.jobs` file in your project to specify which entities should be exposed, and the Java names that should correspond to them.

   This process is called *mapping* and is described in “Name Mapping” (page 10). You use the `.jobs` file in your project to specify the mapping information. “Editing the Specification File” (page 12) describes the details of working with the `.jobs` file.
7. Build the project.

   The build process invokes the `bridget` tool, which outputs the files needed to use the Objective-C classes in your Java code. See “Building the project” (page 17).

Creating a Java Wrapper Project

In Project Builder:

1. Choose Project-> New.
2. In the New Project panel, select Java Wrapper from the pop-up list.
3. Specify the project path by typing it in the text box or by clicking Browse to navigate to it.
4. Click OK.

The newly created project contains the following files of interest:

- The Other Sources suitcase contains a skeleton specification (`.jobs`) file.
- The Frameworks suitcase contains the Foundation framework by default. You must add the framework you are wrapping to the project.

Name Mapping

This section provides some background information you’ll need in order to properly edit your `.jobs` file.

The Java bridge maintains a table that maps Objective-C selectors to Java method names. By default, the bridge maps Objective-C selectors to Java by using the first keyword only. For example, the method `doThis` would map to `doThis()` in Java. If this is what you want, you don't need to specify an explicit mapping.

The Java bridge also lets you overload Java method names, as long as each one has a unique arguement list, so the methods `doThis:withThat:` and `doThis:withSomethingElse:` can both map to `doThis()` in Java. However, as of press time, the bridge still had limitations with overloading, so in some cases you may need specify different Java names, say, `doThisWithThat()` and `doThisWithSomethingElse()`.. For more information, check the files in `/System/Documentation/Developer/YellowBox/ReleaseNotes/`.

There are some other restrictions on name mapping:

- A single Objective-C selector can't be mapped to different Java method names for different classes or interfaces. For example, If you map `foo:` to `javaFoo()` in a given class, any method named `foo:` in another class must map to `javaFoo()`.
- Different Objective-C selectors can't be mapped to the same Java method name. For example, if you map `foo:` to `javaFoo()` in a given class, you can't map `fooBar:` to `javaFoo()` in another class.
- In Objective-C, it's possible for an instance method to have the same name as a class method, while in Java, class (static) and instance methods share the same name space. Therefore, if an Objective-C class has methods `-foo` and `+foo`, you must map at least one of them to a different name in Java.

Objective-C initialization methods should be exported in Java as constructors, not as instance methods. This allows you to use the `new` operator in Java to allocate and initialize the operator.

You can expose multiple initialization methods as constructors. For example, suppose the Foo class has two initialization methods: `init` and `initWithString:(NSString*)`. The bridge creates the appropriate Java constructors based on the argument types: `init` becomes the constructor `Foo()` in Java, and `initWithString:` becomes `Foo(String s)`.

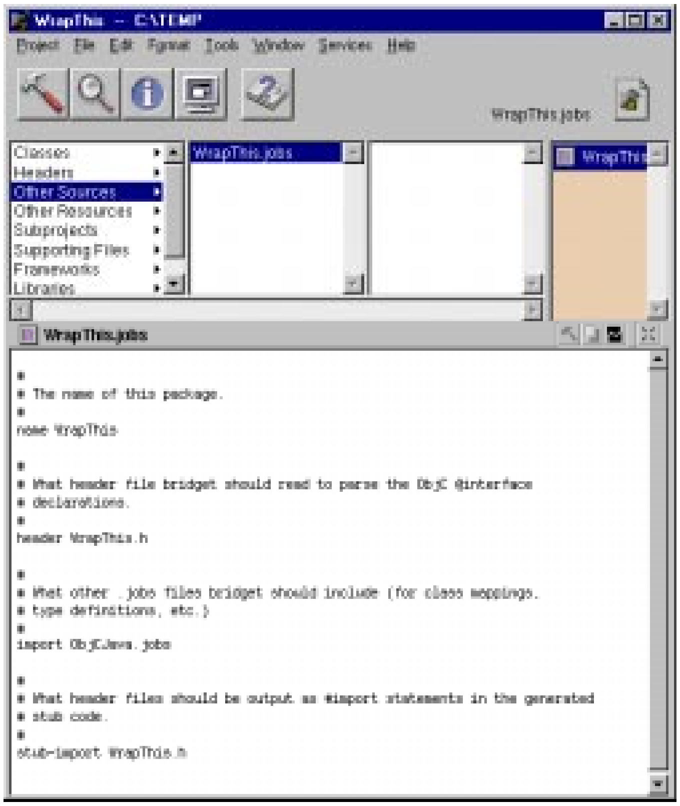
Editing the Specification File

The specification (`.jobs`) file shows how the Objective-C class is exposed in Java. It allows you to choose which Objective-C classes, methods, types, and protocols you want to expose, and the names they should have on the Java side.

The `.jobs` file also allows you to add specific code to a Java class or interface. This can be used to define constants corresponding to enumerated types used in Objective-C, or simply to provide extended functionality in the Java world.

The `.jobs` file that is created by default with a Java wrapper project looks like this:

Figure 1-2 The .jobs file in a new project



```
#
# The name of this package.
#
name WrapThis

#
# What header file bridget should read to parse the ObjC @interface
# declarations.
#
header WrapThis.h

#
# What other .jobs files bridget should include (for class mappings,
# type definitions, etc.)
#
import ObjCJava.jobs

#
# What header files should be output as #import statements in the generated
# stub code.
#
stub-import WrapThis.h
```

There are two ways to edit a `.jobs` file:

- Use the WrapIt application, which lets you graphically specify the relationships between Objective-C and Java classes. It's in the directory `System/Developer/Applications` and you can find documentation on it in `System/Documentation/Developer/YellowBox/ReleaseNotes`.
- Edit it manually within Project Builder.

Note that if you use the WrapIt application, you should not manually edit the `.jobs` file it creates

The table below lists the keywords that the specification file understands, and shows examples each one.

Syntax of .jobs file

`name`

The name of the package, which comes from the project name (you shouldn't change it). It's also used for the name of the library initialization file, for example, `SimpleWrapper-init.m`.

```
name SimpleWrapper
```

`header`

Specifies the header file that `bridget` should read to parse the Objective-C `@interface` declarations. You can only specify one header file.

```
header MyFramework/Foo.h
```

`import`

Specifies other `.jobs` files that `bridget` should include for class mappings, type definitions, and so on.

```
import FoundationJava.jobs
```

`stub-import`

Specifies header files that you want to output as `#import` statements in the generated stub code.

```
stub-import MyHeader.h
```

`selector`

Specifies any non-default mappings between Objective-C selectors and Java method names. (The default is to use the Objective-C name before the colon as the Java name.) These mappings apply to all classes. **Note:** Put all of the mappings under a single `selector` specification.

```
selector
    -defineClass:withName: = defineClassWithName
    -pathForResource:ofType: = pathForResourceType
```

`protocol`

Exposes an Objective-C protocol as a Java interface. Use one `protocol` directive for each protocol. You must specify all methods within the protocol that you want exposed. **Note:** Constructors are not allowed in Java interfaces. Therefore, don't specify initialization methods in a protocol; you must specify them as constructors in each class that uses the protocol (refer to the `class` keyword for more information).

```
protocol MyProtocol = com.myFirm.whatever.myInterface
-doThis:
-doThat:
-doSomethingElse:
```

`class`

Specifies the classes that should be exposed and the methods in each class. Use one `class` specification for each class. In the example here, the Objective-C class MyObjCClass is exposed as com.myFirm.whatever.myJavaClass. The `implements` directive specifies an Objective-C protocol that the class conforms to; all methods in this protocol are exposed. **Note:** This protocol must also be exposed as a Java interface using `protocol`.

All additional methods you want to expose must be shown explicitly. Objective-C class methods (such as `+myClassMethod`) are mapped to Java static methods. Objective-C instance methods (such as `-myInstanceMethod:`) are mapped to Java instance methods. The `constructor` directive is used to map Objective-C initialization methods to Java constructors.

```
class MyObjCClass = com.MyFirm.Whatever.MyJavaClass
    implements MyProtocol
    -myInstanceMethod:
    -myOtherInstanceMethod:withArguments:
    +myClassMethod
    constructor -init
    constructor -init:withSomething:
```

Within the `class` specification, you can specify additional Java code to be added to the class. In the following example, Java constants are declared, corresponding to enumerated types in Objective-C:

```
@{
    public static final int InnerJoin = 0;
    public static final int FullOuterJoin = 1;
@}
```

You can also specify statements that will appear at the top or bottom of a Java package:

```
@top {
    import com.yourFirm.whatever.*
@}

@end {
    private class notSeenAnywhereElse {
        int secret() { return 2001; }
@}
```

`type`

Used to map Objective-C types to Java basic types or classes.

```
type
    NSTimeInterval = double
    NSComparisonResult = int

    NSMyStruct = com.myFirm.myJavaClass using _f1 _f2 _f3
struct-size 16
```

This last form is designed to map Objective-C structs to Java classes. If you need to do this, you must provide three conversion functions: two that convert the struct to a Java object (by reference and by value), and one that converts the Java object to the Objective-C struct, as in the following prototypes:

```
JAVAHandle _f1(NSMyStruct *n, int *structSize);
JAVAHandle _f2 (NSMyStruct n);
NSMyStruct _f3 (JAVAHandle myJavaObj);
```

`map`

Specifies routines for converting (“morphing”) Objective-C classes to Java classes and vice versa.

```
map
    NSNumber = java.lang.Number using _NSNumberToJavaNumber
_JavaNumberToNSNumber
```

`name`

The name of the package. It’s used for the name of the library initialization file as well, for example, `SimpleWrapper.m`.

```
name SimpleWrapper
```

`preinit-callout`

Allows you to specify a function to execute before the bridge's standard initialization code (which sets up the Java-to-Objective-C mappings).

```
preinit-callout your_pre_initialization_function
```

`postinit-callout`

Allows you to specify a function to execute after the bridge's standard initialization code.

```
postinit-callout your_post_initialization_function
```

Building the project

The `bridget` tool uses the `.jobs` file to generate Java classes and interfaces for a given set of Objective-C entities.

When you build the project, the following files are generated:

- A dynamic library containing the Objective-C classes and protocols packaged as Java classes and interfaces.
- A `.java` file for each Objective-C class listed in the `.jobs` file.

  It contains declarations for all of the methods in the class, as well as a static initialization method that insures that the dynamic library is loaded the first time an object of the class is instantiated.
- A C stubs file for each class listed in the `.jobs` file.
- A file (`projectName-init.m`) containing initialization code for the Objective-C classes. This code is run the first time the dynamic library is loaded, and sets up the Java-to-Objective-C mappings.

Modifying Your Code

Occasionally, differences between Objective-C and Java may require you to change your Objective-C framework in order for it to work properly with the bridge. Typically, changes are done by adding categories rather than changing the original classes. You add Objective-C `.m` files to the Classes bucket and `.h` files to the Headers bucket in your project.

The following sections describe some reasons why you might need to modify your code.

Pointers

Java doesn't use pointers, while Objective-C (like C and C++) does. In some cases, you may need to change your code if an Objective-C method you are wrapping returns a pointer or takes a pointer as an argument.

- The bridge automatically converts a pointer to an object (such as an `NSArray*`) to a Java object reference. Therefore, any method using an object pointer will work correctly without modification.
- Pointers such as `int*` or `id*` or `void*` will not work. The Objective-C method must be changed.

In the following example, an Objective-C method uses pointers to return two values "by reference".

```
NSFoo value1:(int*)v1 value2:(int*)v2
```

In Java, values must be returned explicitly by value, so you need to rewrite this method. You can, for example, write separate methods to return each of the values and add them to the NSFoo class via a category, as follows:

```
@interface NSFoo(MyJavaExtensions)
- (int) value1; //returns first argument
- (int) value2; //returns second argument
@end

@implementation NSFoo(MyJavaExtensions)
- (int) value1 {
    int v1, v2;
    [self value1:&v1 value2:&v2]; //invokes original method
    return v1;
};
- (int) value2 {
    int v1, v2;
    [self value1:&v1 value2:&v2];
    return v2;
};
@end
```

Another option is to rewrite `value1:value2:` as a single method whose return type is an NSDictionary that contains both values.

Constructors

Normally, Objective-C initialization methods are mapped to Java constructors based on the number and types of arguments. A special cases arises when an Objective-C class has two initialization methods with the same argument types but different names (say, `initWithString:` and `initWithPathName:`, both taking an NSString). In this case, Java can't distinguish between the two needed constructors.

One solution is to create a "cover" method in Objective-C that takes an extra parameter and calls the correct initialization method depending on the value of the second parameter. This cover method is mapped to the Java constructor.

A similar approach is to write the cover method in Java. You expose the initialization methods as instance methods rather than constructors and add a custom pure Java constructor such as:

```
public Foo (String myString, boolean isPathName) {
    if (isPathName) {
        this.initWithPathName(myString);
    }
    else {
        this.initWithString(myString);
    }
}
```

This code can be added to a Java class in the `.jobs` file. See the description of the `class` keyword in "Editing the Specification File."