GNUstep Java Interface (JIGS) Manual

Written by: Nicola Pero (n.pero@mi.flashnet.it)
Date: February 2001, April 2001

Contents

Chapter 1

What is JIGS

JIGS stands for *Java Interface for GnuStep.* It is a package allowing integration between Java and Objective-C. The main purpose of JIGS is to allow Java programmers to use the GNUstep libraries from Java.

JIGS is more than a set of bindings for GNUstep from Java; JIGS takes advantage of the fact that Objective-C and Java are very similar languages to make it possible to use Objective-C classes from Java using **exactly the same API** (and vice versa). You have to learn the GNUstep API only once, and then you can use it both from Objective-C and from Java in the same way! Moreover, JIGS can generate automatically wrappers for your own Objective-C GNUstep libraries.

JIGS allows you to write optimized classes and components in Objective-C, and make them available to Java developers. In the typical configuration, fast Objective-C components and libraries are used in the backend, and JIGS is used to integrate them smoothly in a Java front-end. Moving from one language to the other is as easy as it can possibly be with two different languages.

JIGS is free software and part of the GNU project, freely available from the Free Software Foundation under the GNU LGPL license.

This manual does not cover getting and installing JIGS; please refer to the GNUstep web page for more information on getting and installing JIGS.

Chapter 2

Quick Overview of JIGS

This chapter is a quick technical overview of the various parts of the JIGS source code distribution.

2.1 Components of JIGS

JIGS is composed by:

1. An integrated Java-to-Objective-C interfacing engine (this is the core of JIGS). The source code is located in the `Source/` and `Java/` directories.

2. An automated wrapping tool for Objective-C libraries; this tool can automatically output the wrapping code needed to expose Objective-C libraries to Java using the JIGS core engine; this tool is called `WrapCreator` [but a name change is planned]. The source code is in the `Tools/WrapCreator/` directory.

3. Wrappers for the GNUstep Base and Gui libraries. You need `WrapCreator` to compile these wrappers. The source code is in the `Wrappers/base/` and `Wrappers/gui/` directories.

4. Examples of libraries wrapped by hand. This is mainly useful to JIGS hackers themselves. Source is in `Examples/`.

5. Examples of libraries wrapped using the WrapCreator tool. Source is in `Tools/WrapCreator/Examples`.

6. Examples of using the GNUstep Base and Gui libraries from Java. The source code is in `Testing/Java/`.

7. Examples of running Java code from Objective-C. The source code is in `Testing/Objc/`.

8. This manual. The source is in the `Documentation/Manual/` directory.

2.2 Which Components are Used for What

2.2.1 Starting a Java VM inside an Objective-C program

To start a Java Virtual Machine and access Java from an Objective-C GNUstep program, you need only the JIGS core engine. It must be said that this will not work out the box. You need to find out what libraries you need to link with in order to start a Java Virtual Machine from native code in your Java Environment, because of a lack of standardization between different Java Environments. It's just a couple of library flags for the compiler, but it can be painful, and non-portable. The example is in `Testing/Objc`. If you are lucky, you might also find the flags for your java platform there.

2.2.2 Using the GNUstep Base and Gui Libraries from Java

To use the GNUstep Base and Gui libraries from Java, you only need the JIGS core engine and the GNUstep Base and Gui libraries wrappers. Once you have installed JIGS, it should just work out of the box. The examples are in `Testing/Java/`.

2.2.3 Using Other GNUstep Libraries from Java

To use a generic GNUstep Objective-C library from Java, you need the JIGS core engine, and wrappers for that library. To generate the wrappers, you need to use the WrapCreator tool. Complete examples of wrapping a library and then accessing it from Java are in `Tools/WrapCreator/Examples/`.

Chapter 3

Programming GNUstep in Java

In this chapter, we will learn how to write GNUstep programs in Java. This is the simplest task. We assume that you have installed all the components of JIGS on your system, by following the installation instructions coming with JIGS.

3.1 Compiling Java Code Using GNUstep Make

3.1.1 Compiling a Single Class

Even without JIGS, the GNUstep make package has builtin support for compilation of generic java projects.

As an example, consider the following java class:

```
import java.lang.*;

class Pisa
{
  public static void printQuote ()
    throws Throwable
  {
    System.out.println ("Ahi Pisa, vituperio de le genti");
  }
}
```

This class just prints out the famous quote by Dante about the Italian city of Pisa; it makes only use of the standard java classes, so there is no need to have JIGS to compile or run it.

Compiling this class can be quite easily done directly, but we want to start learning how to use the GNUstep make package facilities for java, so we write the following `GNUmakefile`:

```
include $(GNUSTEP_MAKEFILES)/common.make

JAVA_PACKAGE_NAME = test
test_JAVA_FILES = Pisa.java

include $(GNUSTEP_MAKEFILES)/java.make
```

The makefile is straightforward: the `JAVA_PACKAGE_NAME` is just a name identifying the package (any name would do); `JAVA_FILES` is a list of java files to be compiled.

You should now try to compile this java program by typing `make`. This – quite simply – compiles the java class. What is more interesting is what happens when you type `make install`: in this case, the java class is installed into `$GNUSTEP_LOCAL_ROOT/Libraries/Java/`, which in my case means that the compiled class is installed into

```
/usr/GNUstep/Local/Libraries/Java/Pisa.class
```

Since the GNUstep initialization script – usually

```
/usr/GNUstep/System/Makefiles/GNUstep.sh
```

– adds `/usr/GNUstep/Local/Libraries/Java` to the `CLASSPATH`, after you have installed `Pisa.class`, you can just access `java Pisa` from any Java program automatically.

This can be unfortunate, because if you install many different classes, you end up with all the classes being installed in the same big flat messy directory. The java approach to this is to use packages, an approach which is fully and simply supported by the GNUstep make package.

3.1.2 Compiling Classes Belonging to a Package

To organize our java classes better, we decide to create a new package, called `misc.quotes`, and to make our class part of it. To do this, we rewrite our program as follows

```
package misc.quotes;

import java.lang.*;

class Pisa
{
  public static void printQuote ()
    throws Throwable
  {
    System.out.println ("Ahi Pisa, vituperio de le genti");
  }
}
```

and we save it into the file

```
misc/quotes/Pisa.java
```

This directory is relative to the position of the GNUmakefile, which is where we will run the compiler. We then rewrite our GNUmakefile to be as follows:

```
include $(GNUSTEP_MAKEFILES)/common.make

JAVA_PACKAGE_NAME = test
test_JAVA_FILES = misc/quotes/Pisa.java

include $(GNUSTEP_MAKEFILES)/java.make
```

Typing `make` will compile the `misc.quotes.Pisa` class as usual.

To include the class in other classes, we need now to always specify the full package name, as in

```
import misc.quotes.Pisa;
```

The GNUstep Make Package knows nothing about java packages; but, when installing java class files, it preserves the relative path of the classes, so that running `make install` will install `Pisa.class` into

```
$GNUSTEP_LOCAL_ROOT/Libraries/Java/misc/quotes/
```

In my case, this means that it is installed into:

```
/usr/GNUstep/Local/Libraries/Java/misc/quotes/Pisa.class
```

Since the GNUstep make package does not really know about java packages, it is perfectly correct to compile and install classes belonging to different java packages with the same `GNUmakefile`, as in the following example:

```
include $(GNUSTEP_MAKEFILES)/common.make

JAVA_PACKAGE_NAME = fun
fun_JAVA_FILES = \
  misc/quotes/Pisa.java      \
  misc/quotes/Farinata.java  \
  misc/quotes/Ulisse.java    \
  misc/jokes/Sailor.java     \
  misc/jokes/Worker.java

include $(GNUSTEP_MAKEFILES)/java.make
```

This will correctly compile and install the classes belonging to the two different packages `misc.quotes` and `misc.jokes`; please don't be confused by the `JAVA_PACKAGE_NAME` variable, which bears no relation with java packages, and it is just a temporary name used internally by the make package.

3.1.3 Installing Java classes where you like

Compiled Java classes are automatically installed into

```
$GNUSTEP_LOCAL_ROOT/Libraries/Java/
```

and this makes sure that they are always in the classpath. This is what you normally want.

But sometimes you might need to modify this behaviour. For example, if you are writing Java servlets, you might want to install them in a special, different place. To do this, you may set the `JAVA_INSTALLATION_DIR` variable in your makefile:

```
include $(GNUSTEP_MAKEFILES)/common.make

JAVA_PACKAGE_NAME = servlet
servlet_JAVA_FILES = MyServlet.java

JAVA_INSTALLATION_DIR = /usr/local/servlets

include $(GNUSTEP_MAKEFILES)/java.make
```

This GNUmakefile will compile your servlet, and install it into `/usr/local/servlets`.

3.1.4 Creating Java Tools

In GNUstep, a tool is a program which is run directly from the command line. The GNUstep make package (versions newer than 1.0.0) supports natively Java tools.

As an example, we now show how to turn our Pisa class into a Java tool. We modify the code to be:

```
import java.lang.*;

class Pisa
{
  public static void main (String[] args)
    throws Throwable
  {
    System.out.println ("Ahi Pisa, vituperio de le genti");
  }
}
```

and we replace the `GNUmakefile` with the following one:

```
include $(GNUSTEP_MAKEFILES)/common.make

JAVA_TOOL_NAME = Pisa
Pisa_JAVA_FILES = Pisa.java
Pisa_PRINCIPAL_CLASS = Pisa

include $(GNUSTEP_MAKEFILES)/java-tool.make
```

There are some differences with the `GNUmakefile` we use before. We include `java-tool.make` rather than `java.make` because we want to create a Java tool rather than a generic Java class. `JAVA_TOOL_NAME` is the name of the tool - this is quite important now, because it is the name we will use to invoke the tool from the command line: when you install the tool, GNUstep make will generate and install a shell script in `$GNUSTEP_LOCAL_ROOT/Tools` with that name (`Pisa` in this case), which runs your java program (setting up the class path and the library path if needed). So, once you have installed the Java tool, just typing `Pisa` at the shell prompt will run your it. `JAVA_FILES` are, as usual, the Java source files for the tool. `PRINCIPAL_CLASS` is the class which implements the `main` method – the GNUstep make package needs this information, so that it can prepare the shell script to run this class.

3.1.5 Documenting Classes Using Javadoc

The GNUstep make package has also some support for generating documentation using `javadoc`; if you want to use this facility, please read on. Otherwise, you may safely skip this section.

Here is an example of the usual `GNUmakefile` with added documentation using javadoc:

```
include $(GNUSTEP_MAKEFILES)/common.make

JAVA_PACKAGE_NAME = fun
fun_JAVA_FILES = \
  misc/quotes/Pisa.java       \
  misc/quotes/Farinata.java   \
  misc/quotes/Ulisse.java     \
```

```
  misc/jokes/Sailor.java      \
  misc/jokes/Worker.java

DOCUMENT_NAME = Reference
Reference_JAVADOC_FILES = misc.quotes misc.jokes
Reference_DOC_INSTALL_DIR = Developer/Fun/

include $(GNUSTEP_MAKEFILES)/java.make
include $(GNUSTEP_MAKEFILES)/documentation.make
```

As you see, you need to choose a document name, list the packages you want to document, choose the installation directory, and include `documentation.make`. We now review each option in more detail.

`DOCUMENT_NAME` is a name describing the documentation to be generated. It has some importance, because the documentation is generated inside a directory called `$(DOCUMENT_NAME)`. In this case, when I run `make`, make creates a directory called `Reference`, and runs `javadoc` telling him to put the documentation into this directory (this is accomplished by using the `-d` option to `javadoc`).

`JAVADOC_FILES` contains a list of packages to be documented. It might also contain standalone java files - which need to be specified with full pathnames then, as in

```
Reference_JAVADOC_FILES = /home/nicola/java/test/test.java
```

You can mix package names and java file names - and use an arbitrary number of them.

`DOC_INSTALL_DIR` specifies where the generated documentation is going to be installed when you run `make install`. In the example, it will end up being installed in

```
$(GNUSTEP_LOCAL_ROOT)/Documentation/Developer/Fun/Reference/
```

which on my system means

```
/usr/GNUstep/Local/Documentation/Developer/Fun/Reference/
```

Additional options you may want to use are:

1. Setting `ADDITIONAL_JAVADOCFLAGS` before including `documentation.make` allows you to pass custom flags to javadoc - a standard example is:

   ```
   ADDITIONAL_JAVADOCFLAGS += -public
   ```

2. By default, the make package will tell javadoc to look for source files in the current directory; if some of them are elsewhere, you can specify additional paths to look for the source files by setting a `JAVADOC_SOURCEPATH`:

   ```
   Reference_JAVADOC_SOURCEPATH=/home/nicola/Java/Nico:/home/nicola/Java/Nicola
   ```

   Before running javadoc, the make package will always add `./` in front of the sourcepath - for example, in this case it would use

   ```
   -sourcepath ./:/home/nicola/Java/Nico:/home/nicola/Java/Nicola
   ```

   when invoking javadoc. Please refer to the javadoc documentation if you want to learn more about the sourcepath.

3.2 Accessing the GNUstep Base Library From Java

3.2.1 A First Example

Accessing the GNUstep Base Library from your Java code is very easy. The GNUstep Base Library classes are made available from Java as part of the `gnu.gnustep.base` package. If you have correctly installed JIGS, this package has been installed as any other Java package by the GNUstep Makefile package in your GNUstep directory tree. For example, on my system the GNUstep Base Library classes are in:

```
/usr/GNUstep/System/Libraries/Java/gnu/gnustep/base/
```

To use the classes, it is enough to import them with the statement:

```
import gnu.gnustep.base.*;
```

As a simple example, here is a java program which reads the host name using GNUstep, and prints it out:

```
import gnu.gnustep.base.*;

class PrintHostName
{
  public static void main (String[] args)
    throws Throwable
  {
    NSProcessInfo process;

    process = NSProcessInfo.processInfo ();
    System.out.println (process.hostName ());
  }
}
```

We will comment in detail the code in the next section.

The GNUmakefile for this program is the usual, trivial, one:

```
include $(GNUSTEP_MAKEFILES)/common.make

JAVA_PACKAGE_NAME = PrintHostName
PrintHostName_JAVA_FILES = PrintHostName.java

include $(GNUSTEP_MAKEFILES)/java.make
```

This is probably a good moment to pause and try out this little program on your system, to get a feeling of how simple and easy the whole thing is.

3.2.2 The General Idea

The previous code example showed the basic idea behind the GNUstep Java Interface: Java and Objective-C are very similar languages, so that Objective-C classes can be exposed as Java classes nearly as they are, with very little (if any) changings in the API.

In this example, the Objective-C class `NSProcessInfo` provided by the GNUstep Base Library is exposed to Java as the Java class

```
gnu.gnustep.base.NSProcessInfo
```

Usually, classes in the same Objective-C library are exposed as classes belonging to the same Java package; for example all classes in the GNUstep Base Library are exposed as part of the `gnu.gnustep.base` package, and all the classes in the GNUstep Gui Library are exposed as part of the `gnu.gnustep.gui` package.

Both in Java and Objective-C there are class methods (called *static methods* in Objective-C) and instance methods. The interface preservers both the name and the role of the Objective-C methods when they are exposed to Java: static methods are made available as class methods with the same name, and instance methods are made available as instance methods with the same name. This is the general plan – but there are exceptions and pitfalls, because the languages differ in some details – we will carefully study these exceptions and details in the next sections.

In the example, the Objective-C static method `processInfo` is accessible in Java as the class method `processInfo`; the Objective-C instance method `hostName` is exposed as a Java method `hostName`.

The only difference is in the syntax: in Objective-C you would write

```
NSProcessInfo *process;

process = [NSProcessInfo processInfo];
```

because this is the Objective-C syntax for method calls; in Java the same code is written as

```
NSProcessInfo process;

process = NSProcessInfo.processInfo ();
```

because this is the Java syntax for method calls.

If you want to know exactly which classes and methods are exposed to Java, with which name, and to which Objective-C classes and methods they correspond, you can consult the quick reference documentation for the GNUstep base library wrappers. This is generated automatically by JIGS when it creates the wrappers; if you have installed JIGS correctly, it should be in:

```
$(GNUSTEP_SYSTEM_ROOT)/Documentation/Developer/Base/Java/Reference/
```

Warning: this is only a quick reference for the java wrapper, listing classes and methods exposed to Java, and does not document the GNUstep Base Library itself.

3.3 How the Wrapping Works

In this section, we review how the various Objective-C entities are exposed to Java.

3.3.1 Classes

Objective-C classes are usually exposed to Java with the same name they have in Objective-C. JIGS provides facilities to allow exposing Objective-C classes with a different name to Java; it's better to avoid it if possible.

3.3.2 Methods

Not all Objective-C methods are exposed to Java; some methods can not be exposed for obvious reasons (such as methods taking `void *` arguments). In general, methods taking pointer arguments (not pointing to objects) or an arbitrary number of arguments (in the way the `printf` function does) are not exposed to Java.

Arguments and return types are automatically converted by the interface.

Methods without arguments are (usually) exposed as they are: for example, the `NSProcessInfo` method

```
- (NSString *) processName;
```

is exposed to Java as

```
 public native String processName ();
```

Sometime, the name is changed. JIGS allows to do this – but the less it is done, the easier for the programmer. Anyway sometimes it is appropriate; for example, the GNUstep method `description` (which can be invoked on any object to return a string describing the object) is exposed to Java with the name `toString ()` (which is the standard method for doing the equivalent job in the Java world).

In Objective-C, methods have their arguments introduced by '`:`', as in the following example:

```
- (void) addObject: (id)anObject;
```

This method is accessible from Java as

```
public native void addObject (Object arg0);
```

The only difference is that in Java the argument is enclosed in brackets.

Things get more complicated if the method has multiple arguments. In Objective-C, each argument is introduced by a little string followed by '`:`', as in the following example:

```
- (BOOL) writeToFile: (NSString *)fileName
          atomically: (BOOL)flag;
```

In this example, `fileName` is the first argument, and `flag` is the second one. The method name is

```
writeToFile:atomically:
```

The method is invoked as follows:

```
/* Objective-C example */
[array writeToFile: @"bar"  atomically: YES];
```

In Java, these methods are (usually) exposed by keeping only the first part of the name (which is what comes before the first '`:`'). For example, `writeToFile:atomically:` is exposed as:

```
public native boolean writeToFile (String arg0, boolean arg1);
```

and can be then invoked as follows:

```
/* Java example */
array.writeToFile ("bar", true);
```

It is possible that different Objective-C methods have a name which begins in the same way, but differs in the next parts. For example, imagine that an hypothetical Objective-C class `LocalizedArray` had both the method

```
- (BOOL) writeToFile: (NSString *)fileName
          atomically: (BOOL)flag;
```

and the method

```
- (BOOL) writeToFile: (NSString *)fileName
          atomically: (BOOL)flag
          inLanguage: (NSString *)language;
```

This second method would translate all the entries of the hypothetical array object into `language`, and then save it.

The methods would be exposed to Java as:

```
public native boolean writeToFile (String arg0, boolean arg1);
public native boolean writeToFile (String arg0, boolean arg1, String arg2);
```

which causes no conflict, because Java allows overloading of methods, and the two methods have a different Java signature. It is worth to remark here that the two methods are different in Objective-C because their names are different, while they are different in Java because their (name + Java signature)s are different. This will be important later in the discussion of selectors.

In rare cases, the conflict can't be escaped – JIGS allows the programmer doing the wrapper to resolve the conflict by exposing one of the two conflicting methods with a different name to Java.

3.3.3 Constructors

In GNUstep, you create objects using `alloc`, followed by some variant of `init`. When you expose your class to Java, a corresponding constructor is created for each (`alloc` + `init`) pair.

In short, for each Objective-C `init` method, the Java class has a constructor.

Let us consider for example the `NSArray` class, which is available in Java as the `gnu.gnustep.base.NSArray` class.

In Objective-C, the class has the following `init` methods:

```
- (id) init;
- (id) initWithArray: (NSArray*)array;
- (id) initWithContentsOfFile: (NSString*)file;
- (id) initWithObjects: firstObj, ...;
- (id) initWithObjects: (id*)objects count: (unsigned)count;
```

The last two methods are not exposed to Java (the first one because it takes an arbitrary number of arguments, which Java does not support; the second one because it takes a pointer argument).

The corresponding Java class has the following constructors:

```
public NSArray ();
public NSArray (NSArray array);
public NSArray (String fileName);
```

For example, to create an array from the contents of the `array.debug` file in my home directory, I would do:

```
/* Java Example */
NSArray array = new NSArray ("/home/nicola/array.debug");
```

When two Objective-C `init` methods have different names but arguments of the same type, it is of course impossible to expose both of them to Java. This case is rare and the only solution is to change or extend the Objective-C API (or expose only one of the two).

3.3.4 Primitive Types

Primitive types are mapped automatically by the interface. For example, an Objective-C method returning a `BOOL` (the Objective-C type for boolean variables) will return a `boolean` in Java.

3.3.5 Strings

Strings are transparently *morphed* through the interface. To the end-user programmer, this simply means that Objective-C methods having as argument (or returning) Objective-C `NSString` objects take as argument (or return) in Java Java `String` objects.

Internally, when the method is called, the interface generates on the fly a GNUstep `NSString` object corresponding to the Java `String` object, and passes it to Objective-C. Viceversa, when an Objective-C method returns a `NSString`, a Java `String` object is generated on the fly from the returned string and returned to Java. This design is very comfortable for the programmer, who always uses the string objects of the environment he is programming in, but it is not very efficient. This will only have an effect on your program's performance if you pass very big strings – and (or) a huge number of them – through the interface, while normally it doesn't make any appreciable difference in terms of performance.

3.3.6 Numbers

Numbers (instances of `java.lang.Number` and subclasses) are transparently morphed through the interface, in the same way as strings are. Again, this simply means that if an Objective-C method takes as argument (or returns) an Objective-C `NSNumber` object, the method in Java will take as argument (or return) a Java `Number` object; the interface will automatically take care of the conversion.

Details of Number Mapping

This section explains the internals of the number morphing; it's quite technical and for most people it's not really useful nor interesting, so you may safely skip it at a first reading.

Internally, the situation is more complicated than in the case of strings, because there are many public subclasses of `java.lang.Number`, which hold considerably different types of numbers each; and `NSNumber`, according to its Objective-C type, can hold very different types of numbers as well.

When the interface has to convert a Java `Number` object to an Objective-C `NSNumber` object, it will check if the `Number` object belongs to one of the standard Java number classes (`Byte`, `Short`, `Int`, `Long`, `Float`, `Double`); in this case, it will extract the value of the number in the appropriate precision (for example, by using `byteValue` for a `Byte`, or by using `longValue` for a `Long`), and create a corresponding `NSNumber` object with an Objective-C type which is big enough to hold the value. If the `Number` object instead is of a non-standard class (such as a class you implemented), the interface will extract the value of the number by using `doubleValue`, and create a `NSNumber` object big enough to hold this double value. This can obviously result in loss of precision or conversion problems if the Java number object is holding a value which can't be represented with a double.

When, viceversa, the interface has to convert an Objective-C `NSNumber` object into a Java `Number` object, it will check the Objective-C type of the Objective-C object (only a finite, documented number of number types are possible); it will extract the value using the appropriate method (such as `longValue` is the number is of type `long`), and create a Java `Number` object of a standard class which can hold the value.

Please notice that an Objective-C `NSNumber` object can hold a `BOOL` (boolean) value; this type is morphed into a small Java `Number` object (usually a `Short`) holding `0` for `NO`, and `1` for `YES`. It can't be morphed into a Java `Boolean` object, because in Java `Boolean` objects are not numbers, and the whole point of morphing is that a `NSNumber` is morphed exactly into a Java `Number`.

3.3.7 Exceptions

Exception generated during calls to the GNUstep Objective-C code are made available to Java as objects of the `gnu.gnustep.base.NSException` class. This is a subclass of `java.lang.RuntimeException`. GNUstep Objective-C exception usually have a `name` and a `reason`. For example, `NSArray`'s `objectAtIndex ()` method can generate an exception if you ask for an object which lies past the end of the array; in this case the name of the exception is '`NSRangeException`', and the reason is '`Index out of bounds`'. Here is an example:

```
/* Java Example */
NSArray array;
Object object;

array = new NSArray ();
try
{
  /* Raises an exception because array is empty */
  object = array.objectAtIndex (5);
}
catch (NSException e)
{
  System.out.println ("Name is " + e.name ());
  System.out.println ("Reason is " + e.reason ());
  System.out.println ("Full Description is:" + e);
}
```

The example shows how to access the information about the GNUstep exception. On my system, it prints:

```
Name is NSRangeException
Reason is Index out of bounds
Full Description is gnu.gnustep.base.NSException(NSRangeException):Index out of bounds
```

Many GNUstep methods can raise exceptions, even if these exceptions are not declared in the Java method declarations (contrary to the Java convention) because in Objective-C the exceptions raised by methods are declared in the documentation, not in the code – so it's important to understand that, when accessing GNUstep from Java, you could need to use an exception handler for delicate parts of your code even if your Java compiler does not force you to do so.

3.3.8 Base Library Structs

The fundamental structs defined in the base library

```
NSRange, NSPoint, NSSize, NSRect
```

are exposed as the Java classes

```
gnu.gnustep.base.NSRange
gnu.gnustep.base.NSPoint
gnu.gnustep.base.NSSize
gnu.gnustep.base.NSRect
```

The important thing to understand is that you should think of objects of these classes as C struct – that is, very lightweight objects which are not meant to be subclassed (actually, you can't subclass them at all, since they are `final`), and only act as a container for information which you want to access and pass around as fast as possible. This is actually why they are structs and not objects in the GNUstep Objective-C libraries in the first place.

We discuss here the case of `NSPoint`; the other cases are completely similar.

Whenever an Objective-C method takes a `NSPoint` as argument, the corresponding Java method takes a `gnu.gnustep.base.NSPoint` object as argument. When the method is called, JIGS internally creates a struct (allocated on the stack so it's fast) and initializes it with the `x` and `y` coordinates stored in the Java `NSPoint` object; it passes this struct as argument to the method call[1].

Whenever an Objective-C method returns a `NSPoint` (C struct), the corresponding Java methods returns a Java `NSPoint` object. When the method is called, JIGS converts the returned struct by creating a corresponding Java `NSPoint` object, and returning this object to Java.

To create an `NSPoint`, you just use the constructor `NSPoint (float x, float y)`, as follows:

```
/* Java Example */
NSPoint point;

point = new NSPoint (1, 1);
```

Once you have created a `NSPoint`, you can read and set its `x` and `y` coordinates directly, as in the following example:

```
/* Java Example */
point.x = 5;
point.y = 4;
```

What is extremely nice in this approach is that you read and set the coordinates of the point with **exactly** the same code you would use in Objective-C:

[1]Please note that this is completely consistent with C, where `NSPoint` is a struct, which is passed to the called function by value – the Objective-C function/method can not modify the original `NSPoint` argument, because it is only given a local copy of it. This works from Java in the same way: you pass to your method a Java `NSPoint` object as the argument; the method will not modify it. This is an important point if you want to compare JIGS with Apple's Java Bridge. Recent releases of Apple's OPENSTEP-like environment add a flock of new C types to their Foundation library – namely, `NSPointPointer`, `NSSizePointer`, `NSRectPointer`, `NSRangePointer`, `NSPointArray`, `NSSizeArray`, `NSRectArray` and `NSRangeArray` – which are then all mapped in a different way to Java; correspondingly, they have different Java classes (for example they have `NSMutablePoint`, corresponding to their new C type `NSPointPointer`). GNUstep will possibly have these C types for compatibility with Apple, but their use is discouraged, and the same must be said for future implementation of `NSMutablePoint` and similar. JIGS's wrapping of `NSPoint` has been designed to be as easy, intuitive and fast as possible – and in the spirit of the original OPENSTEP design/specification.

```
/* Objective-C Example */
point.x = 5;
point.y = 4;
```

Some functionality which is provided by the GNUstep Base Library in functions has been exposed as methods of the class `gnu.gnustep.base.NSPoint`. This mainly includes:

```
public boolean isEqualToPoint (NSPoint aPoint);
public String toString ();
```

which provide the equivalent of the GNUstep Base Library functions

```
BOOL NSEqualPoints (NSPoint aPoint, NSPoint bPoint);
NSString *NSStringFromPoint (NSPoint aPoint);
```

Since `gnu.gnustep.base.NSPoint` is `final` (ie, it can't be subclassed), your Java compiler should be able to inline all the Java equivalent of these little functions, and they should be reasonably fast.

3.4 Advanced Topics

3.4.1 Weak References to Java Objects

Usually, when you pass a Java object as an argument to an Objective-C method, you do not need to do anything particular:

```
/* Java Example */
NSMutableArray array = new NSMutableArray ();
FileReader nicola = new FileReader ("nicola");

array.addObject (nicola);
```

In this example (assuming the file `nicola` exists), we create a java `FileReader` object, and then put it into a GNUstep `NSMutableArray`.

This is the standard case – all works fine and out-of-the-box; unfortunately, there is an exception. The following methods need special treatment to be used from Java:

```
public void setDelegate (Object delegate);
public void setTarget (Object target);
```

These methods are quite common in the GNUstep API, so it's quite important that you know how to correctly call them. These methods are special, because the Objective-C implementation uses a special memory management for the argument: it saves it as a weak reference rather than a normal reference. This is not a problem from Objective-C, but in Java to use these methods you need to prepare the call by using the method

```
public void retainObject (Object object);
```

provided by the `gnu.gnustep.base.NSObject` class[2], as in the following example:

[2]The Apple Java Bridge has the same memory management problem, but it does not provide this method, so on their platform the Java programmer has to use a hack (an auxiliary array) to deal with it.

```
/* Java Example */
FileReader nicola = new FileReader ("nicola");
NSButton button = new NSButton ();

/* Retain object before calling setTarget */
NSObject.retainObject (nicola);

button.setTarget (nicola);
```

The example (setting a `FileReader` as the target of a `NSButton`) is not necessarily meaningful, but it shows quite clearly how you should call `setTarget`.

In most cases, you use `setTarget` and `setDelegate` only at the beginning, and then keep the target and delegate you have set for the whole life of you application. In these cases, a single call to `retainObject` is enough.

In particular cases, you may want to change the target and delegate during the lifetime of your application, or you may want to destroy the object whose target (or delegate) you have set. In these cases, the single call to `retainObject` is enough to prevent your application from crashing, but you will be leaking a little amount of memory each time you change the target or delegate, or destroy an object having a target or delegate set. To stop this leaking, you need to use `releaseObject` on the old target/delegate after you have relieved it of its duty as target/delegate (or after you have destroyed your object having it as its target/delegate). In other words, you need to compensate each `retainObject` with a `releaseObject` call.

Here is a full example, in which we call `setTarget` to set `oldTarget` as target, and then later we call it again to change the target to be `newTarget`:

```
/* Java Example */

/* Set the old target */
NSObject.retainObject (oldTarget);
button.setTarget (oldTarget);

/* Do something else */

/* Many lines later... */

/* Set the new target */
NSObject.retainObject (newTarget);
button.setTarget (newTarget);

/* Balance the retain you did of the old target */
NSObject.releaseObject (oldTarget);
```

This example is correct and leaks no memory; but hey – in some cases, you don't remember what was the old target! You can still get it from the button before setting the new one:

```
/* Java Example */

Object old;

/* Get the old target */
old = button.target ();
```

```
/* Set the new target */
NSObject.retainObject (newTarget);
button.setTarget (newTarget);

/* Release the old target */
NSObject.releaseObject (old);
```

3.4.2 Selectors

Selectors are a typical feature of the Objective-C language, and are used quite often in the GNUstep API.

For example, in Objective-C you can tell a button that, when it is pressed, it needs to send a message with a certain name to a certain object. The crucial point is that in Objective-C you can dynamically change indipendently both the object which is to receive the message (and consequently its class) and the name of the message you want to be sent to the object.

A selector represents just this – a method with a certain name – leaving unspecified the class the method belongs to. In Objective-C you can then ask (at runtime) to the libraries to know if a class implements a certain selector, or if an object responds to the selector (which happens if and only if its class implements the method), and if it does, invoke it. Selectors are used instead of using directly method names so that the Objective-C runtime can keep selector tables and heavily optimize this kind of operations.

JIGS *morphs* selectors to Java `NSSelector` objects. This means that you can create a selector in Java, as a `NSSelector` object, and then pass it as an argument to methods which in Objective-C would accept a `SEL` (the C type for a selector) argument; and vice versa, methods returning a `SEL` in Objective-C will return an `NSSelector` object in Java.

There is an important difference to learn here: in Objective-C, a method is univocally identified by its name; in Java, different methods can have the same name, and be distinguished only because they have different argument types (overloaded methods). Thus the correct way of interpreting a selector in Java is as a method name and a list of argument types. When you create a Java `NSSelector` object, you must specify both the Java name and the type of the arguments. The JIGS core engine takes care of mapping selectors from Java to Objective-C and vice versa, so that you use selectors on the Java side by specifying both java name and argument types, and on the Objective-C side by only specifying the objective-C name. Where is the trick ? Well, don't forget that Objective-C method names are longer than Java method names – in Objective-C a method name has an additional part for each argument – such as in '`writeToFile:atomically:`', while in Java only the first part of the name is used, such as in '`writeToFile`'. To map the Java method name to the full Objective-C name you need to know the type of arguments – which in this case are a `String` and a `BOOL`. By knowing the argument type, the JIGS engine can find out that the short Java name `writeToFile` has to be expanded when crossing the interface to the long Objective-C name `writeToFile:atomically:`. This is why in Objective-C the name of the method (which has an additional string per parameter) is enough, while in Java, where the name is very short and brief, you need the argument types too.

We now make a complete example to make clear how selectors are used from Java in practice. We want to create a button from java, and have it print out

```
Ahi Pisa, vituperio de le genti
```

when the user pushes it. A full working example which you can compile and run is in

```
Testing/Java/NSButtonTest.java
```

here we only examine the parts related to selectors.

First of all, we implement the callback in a Java object:

```
/* Java Example */
include java.lang.*;

class MyObject
{
  public MyObject () { }

  public void buttonPress (Object sender)
  {
    System.out.println ("Ahi Pisa, vituperio de le genti");
  }
}
```

This is the method we want to be invoked when the button is pressed; it is standard in GNUstep that such callback methods should take a single object argument, which is the object which generated the callback – so that in this case, when the user pushes the button, the method `buttonPress` will be invoked with a single argument: the (object representing the) button which was pressed. If you have more than one button all using the same action, this could be useful to tell which button was pushed.

These callbacks are called *actions* in the GNUstep API. So, we have implemented the action that we want to be invoked when the button is pressed.

Then (somewhere else) we create the button,

```
/* Java Example */
 NSButton button;

 button = new NSButton ();
```

we create an object which implements the method `buttonPress`, and set it as the *target* of the button – when the button is pushed, the button tries to invoke the *action* on the *target* object:

```
/* Java Example */
 MyObject object = new MyObject ();

 NSObject.retain (object);
 button.setTarget (object);
```

(why the call to `retain` is needed was explained in the previous section), then we create a `NSSelector` object representing the method we want to be invoked (more comments below), and set it to be the button action:

```
/* Java Example */
NSSelector selector;

selector = new NSSelector ("buttonPress", new Class[] {Object.class}));
button.setAction (selector);
```

And finally (just to make the example complete), we set the title of the button, and make it of the correct size:

```
/* Java Example */
button.setTitle ("Print Quote");
button.sizeToFit ();
```

As exemplified in the code, `NSSelector` has a single constructor which should be used from Java, and it is:

```
public NSSelector (String aMethodName, Class[] someParameterTypes);
```

The first argument is the name of the method ('`buttonPress`' in our case), the second argument is a Java array of `Class` objects representing the types of the method arguments. In the example, our method has a single argument, which must be an `Object`; and

```
new Class [] {Object.class}
```

is the Java code to create a new array of `Class` objects containing a single `Object.class` object – if this is unclear, please refer to your Java documentation.

FIXME: Explain why in some circumstances selectors for class methods are not safe to use from Java.

3.4.3 Subclassing GNUstep Objects in Java

Classes of GNUstep Libraries exposed to Java are Java classes in all respects, so you can subclass them directly and simply.

`NSObject` implements `finalize ()` to make some important cleanup – so if your `NSObject`'s subclass implements `finalize ()`, you must call `super.finalize ()` at the end of your subclass implementation, as in the following template:

```
public void finalize ()
{
  /* Insert any cleanup specific to this class here */

  super.finalize ();
}
```

WARNING: This does not work yet (as of JIGS 0.9.3) if you pass the object back to Objective-C.

3.4.4 Using Java Threads

You can access GNUstep from multiple Java threads. Make sure that the GNUstep libraries you access are themselves thread-safe before trying to use them in multithreading ! Otherwise you need to properly synchronize access to the library facilities using the standard Java synchronization support.

The Fine Prints About Threads

This section gives you a rough idea of how threads work in JIGS. You may skip it at a first reading.

The first Java thread which accesses GNUstep will be registered as the default Objective-C thread. If this is the only thread you use to access Objective-C, no other thread will be registered with Objective-C, and GNUstep will happily go single-threaded. But as soon as you access

GNUstep from a different thread, GNUstep will detect that you are using a different thread, and switch to multi-threading mode. When multi-threading is enabled, the behaviour of JIGS is also different: each time you call GNUstep, the thread is attached to GNUstep, and detached when the call to GNUstep returns (unless it is the default thread). This is quite expensive – which implies that your default Java thread (the first one you use to access GNUstep) will access GNUstep much faster than other Java threads; and that – if you are aiming at performance – multi-threaded code is not necessarily going to be faster than single-threaded code.

In some rare cases, the default behaviour could not be appropriate for your needs, and you could want to force GNUstep into multithreading mode before you access it from a secondary thread. The standard trick to do this is to start an auxiliary Java thread, and run any GNUstep command inside this thread. The other solution is to call the method

```
static native public void forceMultithreading ();
```

of the `gnu.gnustep.java.JIGS` class. This is a GNUstep extension which simply forces GNUstep into multithreading mode by starting an auxiliary GNUstep thread.

Chapter 4

Wrapping Objective-C Libraries

In this chapter we are going to learn how to use JIGS to expose an existing Objective-C library to Java. Technically, we call this procedure *wrapping* the library.

4.0.5 Historical Notes

Building java wrappers for an objective-C library in JIGS used to be strongly coupled with building the library itself. Starting with JIGS 1.4.0, the wrapping mechanism has been redesigned to be totally independent from the original library: building the objective-C library and building the java wrappers for it are now totally independent processes. You can now build the objective-C library, install it, and only later on create java wrappers for the library. You can also easily wrap objective-c libraries which you got in binary form (as soon as you have the header files from them).

You can still build wrappers in the old way by including

```
old-java-wrapper.make
```

rather than `java-wrapper.make`. But please - whenever possible - use the new - much better - way, and please update any old wrapper to the new design. In this manual, we only document the new design. The old backward compatibility code will be dropped at some point in the future.

4.1 Compiling ready-to-use wrappers

We start by reviewing the simplest case: you have a library installed on your system, and you download the wrappers for that library. The wrappers actually consist of a `GNUmakefile`, a `.jigs` file, and possibly a few other support files. These files contain all the instructions JIGS needs to wrap your library. You can try the discussion out by using any example in the `Tools/WrapCreator/Examples/` directory (which are slightly more complicated, as the library and the wrapper for the library are kept and compiled in the same directory at the same time).

4.1.1 Creating and compiling the wrappers

When you type `make`, JIGS will automatically create a directory called

```
JavaWrapper
```

and generate inside this directory source code for the wrappers. It will then enter into this directory, and compile the wrapper source code (which has just been generated). If you want more details about the wrapping process (which classes and which methods are being wrapped), you can use

```
make verbose=yes
```

To access your library from Java you need to have compiled and installed both the original library and the wrappers.

Typing `make install` will install the wrappers, so you should be ready to use them from Java.

4.1.2 Structure of the wrappers

It's not strictly necessary for you to know the structure of the wrappers to create or use them, so you may safely skip this section.

In the following discussion, we consider wrappers for an imaginary library called `libexample.so`.

The wrappers for the library are composed of two parts: a set of Java classes, and a native Objective-C library, called `libexample.A.so` in the example. This is called after the original library (`libexample.so`) with a `.A` appended to the name.

If you go into the `JavaWrapper` directory, you can inspect these parts yourself. There are two main subdirectories in the `JavaWrapper` directory, the `Java` and the `Objc` subdirectories.

The Java subdirectory

The `Java` subdirectory contains the Java classes. All methods which are not explicitly marked as `public` in these classes are to be considered internal to the JIGS engine and you should not access them directly (you probably will not be allowed to anyway).

All these classes contain a static initializer of the following form:

```
static
 {
   JIGS.loadLibrary ("example.A");
 }
```

This will load the native library `libexample.A.so` as soon as the first Java class is accessed (`libexample.A.so` will implicitly load `libexample.so`).

Then, most methods are `native`, as in the following example:

```
/**
   * Wraps the Objective-C method
   * <B>start</B>
   */
  public native void start ();
```

The implementation of the Java methods marked as `native` will be found by the JVM inside the native library `libexample.A.so` (see next section).

As it is shown in the example, each class and public method has a tiny minimal javadoc description. When the java classes are compiled, quick reference documentation for the classes will be generated from these comments by using javadoc, and put into the subdirectory `Reference`. This documentation is minimal, and basically consists in the list of all Java wrapper classes and methods which were compiled, and for each one, a comment explaining which Objective-C class or method it wraps. While minimal, this documentation can be quite useful for someone already knowing the Objective-C API (or having the Objective-C documentation to refer to) and wanting to use the library from Java.

The Objc subdirectory

The `Objc` subdirectory contains the Objective-C library `libexample.A.so`, which provides to Java the implementation of all the `native` methods in the Java classes. The implementation of these methods will call the corresponding methods in the (pure Objective-C) `libexample.so` library, by using the JIGS core engine to catch exceptions, convert arguments and return types, etc.

4.2 The GNUmakefile for java wrappers

We are now going to learn how to write the `GNUmakefile` and the other support files to have JIGS wrap an Objective-C library. We begin with the `GNUmakefile`. Here is a simple example:

```
include $(GNUSTEP_MAKEFILES)/common.make

JAVA_WRAPPER_NAME = Name

include $(GNUSTEP_MAKEFILES)/java-wrapper.make
```

You need to create a new directory, and put this `GNUmakefile` into it. The `JAVA_WRAPPER_NAME` must be the same as the name of the library you want to wrap, after removing the `lib` in front of the library name. In the example, we are wrapping a library called `libName`.

To correctly generate the java wrappers, you will actually need to instruct JIGS about what classes and methods you want exposed, and how. This will be discussed in detail in the following sections.

As a last comment, you may want to stop JIGS from automatically trying to compile the java wrapper after creating them, and from installing it automatically after installing your library. I don't see why this could be useful except if you are debugging JIGS itself, anyway if you ever need this, just add

```
BUILD_JAVA_WRAPPER_AUTOMATICALLY=no
```

to your library's GNUmakefile. Then - JIGS will still create the java wrapper source, but will not automatically compile it. When you want to compile it, you have to `cd` into the java wrapper directory, and type `make` there. Similarly, to install the java wrappers `cd` into the java wrapper directory, and type `make install` (you probably need to be root) - JIGS will not install them automatically when you install the library.

4.3 Specifying the main library header

4.4 Library header name

JIGS automatically parses your library headers to determine the exact method declarations of the methods you want to expose.

You need to specify a single header file for your library which includes all the header files for your library. JIGS tries to determine it in the following way: it takes the `JAVA_WRAPPER_NAME`, and adds `.h` to it.

For example, if the GNUmakefile for your java wrapper is the following:

```
include $(GNUSTEP_MAKEFILES)/common.make

JAVA_WRAPPER_NAME = Game

include $(GNUSTEP_MAKEFILES)/java-wrapper.make
```

Then your library name is `libGame`. The header file JIGS will try to parse by default is `Game.h`. This header file should include all the header files which might be need to compile code against your library.

Often, you want to set explicitly the header file name. You can do that simply by setting the

```
XXX_HEADER_FILES
```

variable (where `XXX` is to be replaced with your `JAVA_WRAPPER_NAME`). For example, when wrapping the gnustep-base library, we have in the `GNUmakefile`

```
include $(GNUSTEP_MAKEFILES)/common.make

JAVA_WRAPPER_NAME = gnustep-base
gnustep-base_HEADER_FILES = Foundation/Foundation.h

include $(GNUSTEP_MAKEFILES)/java-wrapper.make
```

Please note that it should be a single header; you can't use multiple headers, despite the variable is called `HEADER_FILES`. The header file name should be in the full form you use for including it into C source code, such as in `Foundation/Foundation.h`. It must be in this form because it **will** end up in an Objective-C source file.

4.5 Library header location

When JIGS has determined the name of the header file to parse, it will need to find it. Usually, JIGS will look into the current directory, and then in the standard GNUstep header directories, that is, it will look (in the order) into:

```
./
$(GNUSTEP_USER_ROOT)/Headers/
$(GNUSTEP_LOCAL_ROOT)/Headers/
$(GNUSTEP_NETWORK_ROOT)/Headers/
$(GNUSTEP_SYSTEM_ROOT)/Headers/
```

For example, if you are using

```
include $(GNUSTEP_MAKEFILES)/common.make

JAVA_WRAPPER_NAME = Game
Game_HEADER_FILES = Game/Game.h

include $(GNUSTEP_MAKEFILES)/java-wrapper.make
```

then JIGS will look for the following files, in the order:

```
./Game/Game.h
$(GNUSTEP_USER_ROOT)/Headers/Game/Game.h
$(GNUSTEP_LOCAL_ROOT)/Headers/Game/Game.h
$(GNUSTEP_NETWORK_ROOT)/Headers/Game/Game.h
$(GNUSTEP_SYSTEM_ROOT)/Headers/Game/Game.h
```

and pick up the first it finds.

In some cases, that is not enough - for example, you might not yet have installed your library. In this case, you need to help JIGS by telling it where it might find your header, by setting the `XXX_HEADER_FILES_DIR` variable (again, replace `XXX` with your java wrapper name). It might be a relative path (relative to the current directory), such as in

```
Game_HEADER_FILES_DIR = ../GameLibrary/
```

or an absolute one, as in

```
Game_HEADER_FILES_DIR = /opt/games/Source/GameLibrary/
```

4.5.1 Special header flags

If you need any other special flags for reading the header, use the `XXX_ADDITIONAL_CPP_FLAGS` variable, as in

```
Game_ADDITIONAL_CPP_FLAGS = -I/opt/games/Source/Support/Headers
```

Basically, you should specify all flags which you would normally use to build a library or a bundle or an executable linked against the library you want to wrap.

4.6 The wrapper directory

JIGS generates full code for the java wrappers into a specific directory, which we call the wrapper directory. By default, this directory is called `JavaWrapper` but you can change the name of the directory by simply setting the `WRAPPER_DIR_NAME` variable in your `GNUmakefile`. This is not very common.

To determine if the wrappers are to be rebuilt or not, JIGS keeps a stamp file in the wrapper directory.

Everything which is inside the wrapper directory is automatically generated – this means that

1. you can always remove the wrapper directory without loosing anything, since you can always regenerate the wrappers;

2. you should not edit or modify directly the contents of the wrapper directory, because when the wrappers are recreated, the content of the directory is destroyed.

4.7 The .jigs File

4.7.1 Determining which .jigs file to read

After running the preprocessor on the header file, JIGS will read a configuration file to know which wrappers to generate, and how. By default, the file is called `$(JAVA_WRAPPER_NAME).jigs`. In the case described before, this file would be called `Game.jigs`. You can override this choice by setting the `JIGS_FILE` variable in your `GNUmakefile`, but you shouldn't usually need to.

4.7.2 Overview of the .jigs file

The `.jigs` file is the main file from which you can control the way the wrapping is done. In this file, you need to list all the classes of your library which you want to expose, and for each class, which methods. The `.jigs` file is in a format called *property list* – please refer to the GNUstep Base Library tutorials and documentation if you need more help on property lists. The `.jigs` file is a big dictionary. Each entry in the dictionary is to be thought of as of a different section; the key to the section is the section name, the value is an array or a dictionary containing the information referring to that section. For example, we have the `classes` section (containing an array of classes to be wrapped), the `types` section (containing a dictionary of mapping of custom C types to already known C types), the `method name mapping` section (containing a dictionary of mappings from objective-C method names to Java method names, to be used for the whole library). Here is an example:

```
{
  classes = (
    {
      "java name" = "gnu.gnustep.GreatGameLibrary.GGLSprite";
      "objective-c name" = GGLSprite;
      "instance methods" = (speed, "setSpeed:");
      initializers = (init);
     }
   );

  "method name mapping" = {
    speed = getSpeed;
  };
}
```

In this example, we only have the `classes` and `method name mapping` sections. We now examine in detail each section of the `.jigs` file.

4.7.3 The Classes section

Probably the most important section in the file is the `classes` section, where you list all the classes you want to be wrapped, and how. This section is an array, each entry in the array being a different class to be wrapped. Here is an example of this section:

```
classes = (
  {
    "java name" = "gnu.gnustep.GreatGameLibrary.GGLScore";
    "objective-c name" = "GGLScore";
```

```
    "initializers" = ("init");
    "instance methods" = ("setPoints:", "points");
    "class methods" = ("bestScore", "bestTenScores");
  },

  {
    "java name" = "gnu.gnustep.GreatGameLibrary.GGLPlayer";
    "objective-c name" = "GGLPlayer";

    "class methods" = ("computerPlayer");
    "instance methods" = ("name", "score", "addPoints:");
    "initializers" = ("init", "initWithName:");
  }
);
```

In this example, we want to generate wrappers for two classes: `GGLScore` and `GGLPlayer`. Information about each class is represented by a dictionary. Any entry in this dictionary (except the '`java name`' entry) can be omitted, in which case a default value will be used. We briefly now review the use of each entry.

java name

This is a string – the name that is to be used in Java for the class you want to wrap. This is the only compulsory entry for each class. You need to specify the full Java name of the class, such as

```
"java name" = "gnu.gnustep.base.NSArray";
```

You can decide to use a different name in Java than the name the class has in Objective-C. It's generally better to avoid this, but it should work fine if you ever need to do it.

objective-c name

This is a string – the Objective-C name of the class you want to wrap, such as

```
"objective-c name" = "NSArray";
```

If this entry is not present, it is guessed by taking the last part of `java name`. For example, if `java name` is `gnu.gnustep.base.NSArray`, then JIGS will use `NSArray` as Objective-C name if you don't specify an Objective-C name.

initializers

This is an array – the list of `init` methods you want to wrap, such as

```
"initializers" = ("init", "initWithName:", "initWithArray:", "initWithName:array:");
```

Each method in the `initializers` list will be exposed as a constructor. The Objective-C name of each initializer is discarded in Java, so you must make sure that these methods have all different signatures.

class methods

This is an array – the list of class methods you want to wrap, such as

```
"class methods" = (
  "contentRectForFrameRect:styleMask:",
  "frameRectForContentRect:styleMask:",
  "minFrameWidthWithTitle:styleMask:",
  "removeFrameUsingName:");
```

Each of these methods is exposed as a static method in Java. Unless you use a method mapping (explained below), the Java name of each of these methods will be obtained by removing from the Objective-C name everything which comes after the first ‘`:`’. For example, the methods in the example would be wrapped as:

```
public static native NSRect contentRectForFrameRect (NSRect arg0, long arg1);
public static native NSRect frameRectForContentRect (NSRect arg0, long arg1);
public static native NSRect minFrameWidthWithTitle (String arg0, long arg1);
public static native void removeFrameUsingName (String arg0);
```

Please make sure you do not include any methods of the `new` family in the `class methods` list. `new` methods can not be wrapped because they allocate and return non-autoreleased objects – it should not be a problem anyway, because if you need that constructor in Java, you can get it by simply wrapping the corresponding `init` method.

instance methods

This is an array – the list of instance methods you want to wrap, such as

```
"instance methods" = (
  "arguments",
  "environment",
  "hostName",
  "processName",
  "globallyUniqueString",
  "setProcessName:");
```

Each of these methods is exposed as an instance method in Java. The Java names for these methods are obtained in the same way as for class methods.

Make sure you do not include any method of the `copy` and `mutableCopy` family in this list. Methods of these families can not be wrapped like the other ones because they return newly-created non-autoreleased objects. It is possible that in the future JIGS could provide a way to expose these methods; at present, you can not expose them – except the ones exposed by `NSObject`: `NSObject` exposes `copy` (as `clone`) and `mutableCopy` (as `mutableClone`) to Java, so if your Objective-C class implements `copy` and `mutableCopy`, they will automatically be made available to Java by the `NSObject` wrapper without any need to do anything.

method declarations

This is an array – a list of explicit method declarations which you can use to override the declarations found by the wrapper tool in the header file if you think the wrapper tool did not use the correct declaration – this usually happens only if in your library the same method is declared in different classes or protocols with different arguments or return types (this case is infrequent, but it happens sometime) – in this case the wrapper tool is likely not to pick up the correct declaration, because it is designed for the standard case, when the same signature is used across all classes. By adding the correct declaration for your class in the `method declaration` section, you can make sure that the declaration you want is used. For example, in the gui library the following explicit declaration is used when wrapping `NSBox`:

```
"method declarations" = (
   "- (void) setContentView: (NSView*)aView");
```

This is because `NSScrollView` uses a conflicting declaration for this method, taking a `NSClipView *` argument.

Please note that, while in Objective-C these problems are harmless, and usually the worst it can happen is that the compiler warns you at compile time, after you expose your methods to Java these problems become first-rate problems, because in Java casts are dramatic and your program can crash if a method expecting a `NSClipView` argument receives a generic `NSView` instead.

hardcoded constants

This is an array – a list of declaration of constants which will be added as static constants to the Java class, as in the following example:

```
"hardcoded constants" = ("int NotFound = 0x7fffffff");
```

This `hardcoded constants` section will generate the following code in the Java class:

```
public static int NotFound = 0x7fffffff;
```

This example is taken from the wrapping instructions for `NSArray`; it allows the Java programmer to access this constant simply as `NSArray.NotFound`. The value of this constant is of course the value of the C constant `NSNotFound` which is returned by many GNUstep Base Library methods when something is not found.

If your Objective-C library defines some constants, and some methods which you expose to Java return or accept these constants, you may want to make these constants available to Java as well by using the `hardcoded constants` section. Enumerations are exposed using the `enumerations` section, as explained in the next paragraphs.

enumerations

Starting from JIGS 1.5.0, enumeration types are automatically recognized in the header file, and treated by the wrapper tool exactly as if they were simple `ints`. Usually, that is not enough – you want also to have the enumeration constants available from Java as static integer constants of some class. You can use the `hardcoded constants` section to manually add them to the class; this is quite clumsy, because you have to enter all the enumeration constants manually in the jigs file, and if you change the enumeration values in the Objective-C code, you have to manually update the jigs file.

Recent versions of JIGS (1.5.0 and newer) offer a new better option, in the form of the `enumerations` section. For example, the following line inside a class section

```
enumerations = (NSComparisonResult);
```

will cause the following code to be automatically added to the generated Java class code:

```
public static int NSOrderedDescending = 1;
public static int NSOrderedSame = 0;
public static int NSOrderedAscending = -1;
```

Please notice that the wrapper tool has parsed your Objective-C header file, and extracted the constant values from there automatically! This means that you don't have to enter the constants in the first place, and if you ever change the enumeration, the Java wrappers will be immediately and automatically updated as soon as you recreate them.

file to include in java code

This is a string – a filename (relative to the current directory). When WrapCreator generates the Java class, it will read the contents of this file, and add them – without any change – inside the Java class.

For example, to insert a Java method of your own into the generated Java wrapper class, you could use the following entry:

```
"file to include in java code" = "MyClass.java.add";
```

Then, you need to put the Java code you want to add into the file `MyClass.java.add`. Here is a trivial example:

```
  /**
    * Returns the sum of the arguments.
    */
  public int test (int a, int b)
  {
    return a + b;
  }
```

This code would be inserted *inside* the Java class implementation when the Java wrapper is created.

file to include in preamble java code

This is a string – a filename (relative to the current directory). The only difference with `file to include in java code` (described above) is that in this case the code is included *before* the Java class, between the `import` statements and the class implementation, rather than *inside* the class implementation.

file to include in objective-c code

This is a string – a filename (relative to the current directory). When WrapCreator generates the Objective-C code, it will read the contents of this file, and add them – without any change – at the end of the Objective-C file for that class. In this way, you can add arbitrary functions and C or Objective-C code to the wrapper library.

The combination of this feature with the one to include arbitrary java code into the wrapper allows you to add hand-made wrapped methods to the java wrapper. This is a very advanced feature and is not documented here.

4.7.4 Method Name Mapping

This section is a dictionary – mapping Objective-C method names to Java method names, as in the following example:

```
"method name mapping" = {
  "null" = "nullValue";
  "addTimer:forMode:" = "addTimerForMode";
};
```

Try to avoid method mappings, because they are confusing! But sometimes you can't do without, as in the case of `null` which can't be exposed with this name to Java because `null` is a reserved keyword in Java. In this case, you use the method mapping. Please note that JIGS needs to keep a global table of method name mappings while running in order to map selectors between Java and Objective-C; for this reason, you need to use the same mappings in all the classes.

4.7.5 Prerequisite Libraries

Sometimes, the wrappers for your library need the wrappers of some other GNUstep Objective-C library to work correctly. For example, you can't use the wrappers for the GNUstep Gui Library without the wrappers for the GNUstep Base Library. If this is the case for your library, you need to add a `prerequisite libraries` section, containing a list of GNUstep libraries whose wrappers you want to be loaded at run-time when the wrappers for your library are initialized. For example, in `gnustep-gui.jigs` there is

```
"prerequisite libraries" = ("gnustep-base");
```

You will probably usually need at least this line. Please note that the library is specified using the simplest possible name; JIGS at run-time will add the correct prefixes (`lib`) and suffixes (`.A` for the wrappers) to load in the library.

4.7.6 Types

The wrapper tool can parse and manage a certain set of C types. For example, upon finding `BOOL` in the Objective-C headers, the wrapper tool knows how to manage it.

In the `types` section you can specify a list of C types specific to your library, and for each of them, a corresponding (known) C type. The wrapper tool will know it has to convert to/from Java the first C type in the same way as it does with the second. For example, in the `gnustep-base.jigs` file, there is the following line:

```
types = { NSTimeInterval = double; };
```

This simply tells to the wrapper tool that whenever it finds an argument of type `NSTimeInterval` in an Objective-C method declaration, it has to treat it in the same way as if it were a `double`.

Starting from JIGS 1.5.0, the wrapper tool recognizes enumerations when it parses your header file, and it automatically treats them as new types corresponding to `int`. This means that you don't need to do anything special to have JIGS wrap methods taking or returning enumerations – enumerations should be recognized automatically, and treated automatically as `int`s.

4.8 Useful Make Options when Creating Wrappers

At this point, typing `make` should be enough to generate wrappers for your library.

4.8.1 Silencing the Wrapper Generator

By default, wrapper generation is quite verbose – it tells you exactly what it is doing – for example, it lists all the classes and methods which are being wrapped. If you find this annoying, you can turn it off by compiling with the `verbose=no` option, as in:

```
make verbose=no
```

4.8.2 Enabling Debugging

If you are compiling your library with debugging enabled – ie, typing

```
make debug=yes
```

the wrapper library will also be generated with debugging enabled. This means that the GNUmakefiles for the wrapping library will always automatically use `debug=yes` when the wrapping library is compiled.

4.8.3 Turning off the Generation of Quick Reference Doc

To stop JIGS from automatically generating minimal javadoc comments and the quick reference documentation from them, it's enough to compile using `javadoc=no`, as in:

```
make javadoc=no
```

4.9 Including Special Code in the Wrapper GNUmakefiles

This section describes a feature which is usually not needed, so you may skip it at a first reading. As a prerequisite, you probably need to have read the section 4.1.2 to make any use of this section.

It is possible that you may need/want to insert special code in the GNUmakefile.preamble and GNUmakefile.postamble for the wrappers. For example, some special flags required to compile an Objective-C library (the Objective-C wrapper library for your library) against your Objective-C library.

JIGS itself by default does not generate GNUmakefile.preamble and GNUmakefile.postamble at all; but it will look in the library directory for the files

```
GNUmakefile.wrapper.objc.preamble
GNUmakefile.wrapper.objc.postamble
GNUmakefile.wrapper.java.preamble
GNUmakefile.wrapper.java.postamble
```

and, if any of them is present, copy them into the `JavaWrapper` directory, in the following way:

```
GNUmakefile.wrapper.objc.preamble --> JavaWrapper/Objc/GNUmakefile.preamble
GNUmakefile.wrapper.objc.postamble --> JavaWrapper/Objc/GNUmakefile.postamble
GNUmakefile.wrapper.java.preamble --> JavaWrapper/Java/GNUmakefile.preamble
GNUmakefile.wrapper.java.postamble --> JavaWrapper/Java/GNUmakefile.postamble
```

In this way, by adding one of these files to your library's directory, you can have it copied into the generated wrappers in the appropriate place.

4.9.1 Finding the library

A typical usage of this is to wrap libraries which have not yet been installed. Normally, you create java wrappers for libraries which you have already installed. Sometimes though, you need to create java wrappers for a library before it is installed. In this case, you might need to help the compiler to find your actual library when it links the java wrappers.

You would do so by creating a file called

```
GNUmakefile.wrapper.objc.postamble
```

and adding to this file something like the following line

```
ADDITIONAL_LIB_DIRS = -L/opt/nicola/MyLibrary/
```

Please note that you will need to install your library anyway before you can use the java wrappers, because the library must be found at run-time as well!

A classical case in which you need this trick is if you are building the java wrappers with the library itself (that is, if you have a `GNUmakefile` which includes both `library.make` and `java-wrapper.make` and which you use to compile both. In that case, you need to add

```
ADDITIONAL_LIB_DIRS = -L../../$(GNUSTEP_OBJ_DIR)
```

to your `GNUmakefile.wrapper.objc.preamble` to make sure that, when the java wrappers are built, the compiler will find the original library (and you also need to make sure that `library.make` is included before `java-wrapper.make` so that the library is compiled before the wrappers.

4.10 Controlling the Quick Reference Documentation

This section documents special options which control the quick reference documentation which JIGS automatically generates with the wrappers. You may safely skip this section at a first reading. You need to read the section 4.9 before reading this one.

4.10.1 Generating Documentation for Packages

By default, JIGS will treat the Java classes as separate classes, and document them accordingly. If you are rather creating a complete package you may want javadoc documentation for the package. This is not very different, it just additionally includes a useful index with the list of all the classes in the package.

To get this behaviour, you need to write a

```
GNUmakefile.wrapper.java.preamble
```

and set `Reference_JAVADOC_FILES` in it. For example, when wrapping the GNUstep Base Library we do:

```
Reference_JAVADOC_FILES = gnu.gnustep.base
```

4.10.2 Installation Directory

By default, JIGS will install the quick reference documentation into

`$(GNUSTEP_INSTALLATION_DIR)/Developer/libMyLibrary/Java/Reference/`

where of course `libMyLibrary` is replaced with your library name. If you want to change it, you need to write a

`GNUmakefile.wrapper.java.preamble`

and set `Reference_DOC_INSTALL_DIR` in it. For example, when compiling the GNUstep Base Library wrappers we use

`Reference_DOC_INSTALL_DIR = Developer/Base/Java/`

which installs the documentation into

`$(GNUSTEP_INSTALLATION_DIR)/Developer/Base/Java/Reference/`

4.11 Generating RPMs for the wrapper library

Historically, generating RPMs for the wrappers was a complicated process, because generation of java wrappers was strongly coupled to building the library. Now it's no longer so, and the two processes are now totally independent, which means that you can build RPMs for java wrappers normally as for any other gnustep package – in the simple, standard way – just add `PACKAGE_NAME` and `VERSION` to your `GNUmakefile`, and provide corresponding `.spec.in` and `.debug.spec.in` files.