GNAT Ada-Java Interfacing Suite User's Guide

Release 2014

AdaCore

May 20, 2014

CONTENTS

Category GNU Ada tools

Title GNAT Ada-Java Interfacing Suite User's Guide

Subtile A Toolkit for GNAT, the GNU Ada Compiler

Edition GNAT GPL Edition

Version 2014

Date 2014-05-20

Author AdaCore

CHAPTER

ONE

ABOUT THIS GUIDE

This guide describes the features and the use of GNAT-AJIS, the GNAT Ada-Java Interfacing Suite that can be used with the GNAT Ada development environment.

1.1 What This Guide Contains

This guide contains the following chapters:

- *Getting Started with GNAT-AJIS*, describes how to set up your environment and illustrates the use of the GNAT-AJIS toolset
- *Using ada2java to Generate Java Classes*, describes how to use `ada2java` to generate Java classes that can be used as a ''thin binding" to an Ada package specification.
- *Mapping Ada to Java*, describes how Ada features are mapped to Java by `ada2java`.
- *Advanced ada2java Topics*, describes some of the advanced aspects of the mapping of Ada to Java.
- *Using javastub to Generate Ada Package Specifications*, describes how to use the `javastub` utility.
- *Using JNI Directly*, describes how to write native methods in Ada with the same low-level style as in C.

1.2 What You Should Know before Reading This Guide

Before reading this manual you should be familiar with the following:

- The Ada programming language, and in particular the Ada 2005 Object-Oriented Programming and separate compilation enhancements (e.g., `limited with`, interfaces) that facilitate interfacing between Ada and Java.
- The *GNAT User Guide*
- The Java programming language

It will also be useful if you have a basic knowledge of the following:

- JNI (Java Native Interface), for example as described in *The Java Native Interface Programmer's Guide and Specification*, S. Liang, Addison-Wesley; 1999.

CHAPTER

TWO

GETTING STARTED WITH GNAT-AJIS

This chapter summarizes GNAT-AJIS's basic capabilities and illustrates how to use the GNAT-AJIS tools for some simple applications.

2.1 Introduction

GNAT-AJIS (GNAT Ada-Java Interfacing Suite) is a collection of GNAT add-on tools for developing mixed-language Ada / Java applications where the Java components run on a JVM and the Ada components are compiled natively. Through GNAT-AJIS you can realize the following scenarios:

1. In a Java application, invoke subprograms from natively-compiled Ada packages (i.e., either interface with an existing Ada API, or implement Java native methods in Ada);

1. In a natively compiled Ada program, access methods and fields from Java classes or objects.

GNAT-AJIS addresses these scenarios through an Ada binding to the JNI services and 'binding generator' tools that automate the generation of the necessary 'glue code':

`ada2java`

Takes an Ada package specification as input and produces one or more Java classes, with native methods corresponding to the Ada subprograms. This allows you to call Ada from Java.

`javastub`

Takes a Java classfile and produces an Ada package spec for the native methods found in these files. This allows you to implement Java native methods in Ada.

2.2 GNAT-AJIS Installation Structure

Installing the GNAT-AJIS tools results in the following directory structure. [1]

```
$GNATAJIS_INSTALL_DIR/
   bin/
      ada2java (Solaris, Linux) or ada2java.exe (Windows) -- executable
   include/
      ajis/
         Various `.ads` and `.adb` files
      gnatjni/
         Various `.ads` and `.adb` files
```

[1]

```
lib/
   libajis.so (Solaris, Linux) or ajis.dll (Windows)
   libgnatjni.so (Solaris, Linux) or gnatjni.dll (Windows)
   ajis.jar
   ajis/
      Various '.ali' files
   gnat/
      Various files
   gnatjni/
      Various '.ali' files
```

2.3 GNAT-AJIS / GNAT Compatibility

`ada2java` is based on ASIS and requires a compatible version of the GNAT compiler. To check the status of your installation, run `ada2java` with the `-v` switch; this will indicate the version of the GNAT compiler that was used to build the GNAT-AJIS suite. Your GNAT-AJIS installation is compatible with that GNAT version:

- Running `ada2java` requires using that specific GNAT version;
- On the other hand, the generated Ada files may be compiled with that version or any later one.

The `gnatjni` and `ajis` libraries have been prebuilt for a specific version of GNAT. If you need to compile them for some other version of GNAT, you can rebuild the libraries manually:

```
gprbuild -P ajis.gpr -XExternal_Build=false -XObject_Dir=<some-dir>
```

where <*some-dir*> is a local directory where the temporary objects will be placed.

2.4 A Simple Example: Calling Ada from Java

This section illustrates how to invoke an Ada subprogram (compiled natively) from Java running on a JVM. In summary, the steps are as follows:

- Make sure that the relevant environment variables are properly defined.
- Write a package specification for the subprogram(s) to be called from Java, and a corresponding package body.
- Invoke the GNAT-AJIS tool `ada2java` on the Ada package spec, to produce the corresponding Java classes (source files) and the necessary JNI 'glue' code (additional Ada source files). Providing the `-L libname` switch will cause a project file to be generated, which will help to automate some of the processing.
- Invoke the Java compiler `javac` on the Java source files;
- Invoke `gprbuild` on the project file generated by `ada2java`; this will compile the Ada files into a shared library (Solaris, Linux) or dll (Windows);
- Invoke the Java interpreter to run a Java main class that invokes methods from the Java classes generated by `ada2java`.

These steps will now be described in detail.

2.4.1 Environment Setup

Since you will be using both the Ada and Java toolsets, you need to ensure that several environment variables are set. You can automate this step by defining these variables in a shell script / batch file. For convenience you will also find

it useful to define an environment variable that 'points to' the root directory for the GNAT-AJIS tool installation. The description below assumes that `GNATAJIS_INSTALL_DIR` has this role.

PATH

Must contain the directories for the GNAT tools and for the GNAT-AJIS tools. The latter will be in the `$GNATAJIS_INSTALL_DIR/bin` directory. On Windows, it needs to contain the directory where the shared libraries are generated, typically `./lib` although you can override this.

LD_LIBRARY_PATH

On Solaris and Linux, must contain the directories where your native libraries will reside (generally the `./lib` subdirectory). This variable is not needed on Windows.

CLASSPATH

Must contain `$GNATAJIS_INSTALL_DIR/lib/ajis.jar`, which is the parent directory of the `com.adacore.ajis` Java package.

ADA_PROJECT_PATH

Must contain `$GNATAJIS_INSTALL_DIR/lib/gnat`, the directory that holds the GNAT project files needed for building applications with GNAT-AJIS.

2.4.2 An Ada Package

Assume that you would like to invoke an Ada procedure that displays the text `Hello from Ada`, followed by an integer value passed to Ada from Java. Declare a procedure `Hello` in a package spec `Hello_Pkg` (file `hello_pkg.ads`) and implement the body (file `hello_pkg.adb`):

```
package Hello_Pkg is
   procedure Hello (Item : in Integer);
end Hello_Pkg;

with Ada.Text_IO; use Ada.Text_IO;
package body Hello_Pkg is
   procedure Hello (Item : in Integer) is
   begin
      Put_Line("Hello from Ada: " & Integer'Image(Item));
   end Hello;
end Hello_Pkg;
```

2.4.3 Invoking ada2java

Change to the directory containing the Ada source files, and invoke the command

```
ada2java hello_pkg.ads -L hello_proj
```

This will generate a number of files and directories, including:

```
Hello_Pkg/
   Hello_Pkg_Package.java

Ada2Java/
   Library.java

hello_proj.gpr

Specs and bodies for the JNI_Binding package hierarchy
```

These have the following significance:

Directory `Hello_Pkg` In the absence of an option that specifies the output directory for the generated Java file, `ada2java` creates a new directory with the same name as the Ada input unit and places the Java file in this directory.

File `Hello_Pkg_Package.java` `ada2java` generates a Java source file with native method(s) corresponding to the visible subprogram(s) in the Ada package. (In general `ada2java` may generate several Java source files, based on the contents of the Ada package spec. In this example only one Java file is produced.) The name of this file is the same as the Ada unit, with `_Package` appended (since the input file is a package, rather than a procedure or function). The casing of the file name is the same as that specified on the Ada unit declaration.

Ada parameters are mapped to Java types; here Ada's `Integer` corresponds to the Java type `int`.

In skeletal form, here is the Java class that is generated:

```
package Hello_Pkg;

public final class Hello_Pkg_Package {

   static public void Hello (int Item){...}
   ...
}
```

Directory `Ada2Java` and file `Library.java`

`ada2java` generates the boilerplate file `Library.java` to automate the library load step.

File `hello_proj.gpr` This is a GNAT project file that automates building the application and loading the dynamic library.

Specs and bodies for the `JNI_Binding` package hierarchy These files provide various 'boilerplate' packages as well as the package containing the 'glue code' procedure whose signature complies with the required JNI protocol and which invokes the `Hello` procedure supplied in the original `Hello_Pkg` package.

2.4.4 Compiling the Java class

Invoke the Java compiler on the generated Java class:

```
$ javac Hello_Pkg/Hello_Pkg_Package.java
```

This will generate the classfile `Hello_Pkg_Package.class` in the `Hello_Pkg` directory.

2.4.5 Building the Application

Run `gprbuild`, using the project file generated by *ada2java* at an earlier step:

```
$ gprbuild -p -P hello_proj.gpr
```

This will generate a dynamic library – `libhello_proj.so` (Solaris, Linux) or `hello_proj.dll` – in the subdirectory `./lib` of the current directory, and will produce the necessary object files in the `./obj` subdirectory. The two subdirectories will be created if they do not already exist.

The dynamic library will be loaded automatically at run-time, by one of the generated Java classes.

2.4.6 Running the Program

Write a main Java class, for example a file `Hello.java`:

```
import Hello_Pkg.Hello_Pkg_Package;

public class Hello{
    public static void main(String[] args){
       Hello_Pkg_Package.Hello(100);
    }
}
```

Compile this class:

```
$ javac Hello.java
```

Run the Java program:

```
$ java Hello
```

This will produce the following output:

```
Hello from Ada: 100
```

Note that the library produced earlier must be locatable when the program is executed. On Solaris and Linux the directory containing the library would be specified by LD_LIBRARY_PATH. On Windows, that directory would be specified by the PATH environment variable. However, for the purpose of this introduction, you could simply copy or move the library to the same location as the "Hello.class" file.

For simplicity, Unix-style notation is used throughout this manual in depicting directories and other host system conventions. For Windows, please make the relevant transformations (e.g. \ for / in path names.

CHAPTER

THREE

USING ADA2JAVA TO GENERATE JAVA CLASSES

The `ada2java` tool takes one or more Ada package specs and produces as output a Java 'binding' to these packages, implemented through JNI. The binding consists of a set of Java classes, with methods that access the Ada package's visible entities.

More specifically, `ada2java` generates two sets of source files as output:

- The Java classes that make up the binding, and
- The necessary Ada 'glue code' that hides the details of how JNI is used for interfacing between Ada and Java.

You will need to compile the Java files to bytecodes for execution on a JVM, and you will need to compile the Ada files to native code in a dynamic library.

This chapter explains how to use the `ada2java` tool and describes the mapping from package spec contents to Java classes.

3.1 Using the Tool

The `ada2java` tool is invoked with at least one input file, and any number of switches, in any order:

```
$ ada2java {<switch> | <input-file>} <input-file> {<switch> | <input-file>}
```

Each `<input-file>` must be the name of a source file for an Ada package spec (including the extension).

The following `<switch>` values are allowed:

`-h`

Display help

`-c <JavaClassOutputDirectory>`

The root directory used as the destination for the output classes. The directory will be created if it does not already exist. In the absence of this switch, the current directory is used. See below for the relationship with the `-b` switch.

`-b <BaseJavaBindingPackage>`

The base package for the generated Java classes; this will be relative to the directory specified in the `-c` switch, or relative to the current directory if no `-c` switch was supplied.

`-o <AdaGlueOutputDirectory>`

The destination directory for the 'glue' packages (ads and adb files) generated by ada2java. The current directory will be used if this switch is not supplied. The generated packages will need to be compiled into a dynamic library.

-P <ProjectFile>

The project file that applies to the processing of the <input-file> s submitted to ada2java. This can specify compiler switches, source directories, etc. <ProjectFile> must be a 'flat' project (sources from 'with'ed projects are not yet supported).

-L <LibraryName>

A mechanism for automating the loading of the native Ada dynamic library in Java. This switch causes the generation of a project file <LibraryName>.gpr in the directory specified by the -o switch (or in the current directory if the -o switch was not supplied). The resulting project file can be submitted to gprbuild to build the dynamic library:

```
$ gprbuild -p -P LibraryName.gpr
```

which will generate a lib/ subdirectory that contains the file lib<LibraryName>.so (Solaris, Linux) or <LibraryName>.dll (Windows). This library will be loaded automatically whenever one of the Java classes produced by ada2java is loaded; there is no need for the user to explicitly include an invocation of System.loadLibrary.

-M <MainName>

A mechanism for automating the creation of an Ada main subprogram, embedding both the native code and a JVM. See *Compiling as an Ada Main Subprogram* for more details. Implies -link-method=register_natives.

--main-class=<java main class>

Changes the name of the java main class to use, in case the -M switch is used. See *Compiling as an Ada Main Subprogram* for more details.

--link-mode=(export|register_natives)

The Java virtual machine has two ways of discovering the functions declared in the native environment. Either it checks the correspondence between the exported symbol and the Java native declaration name (export mode), or the JNI code registers manually the symbols using the Register_Native JNI function (register_natives mode). Note that if the code is not in a shared library but compiled with a main native subprogram, then only register_natives mode will work.

--library-kind=(dynamic|encapsulated)

Set the library generation method: normal or standalone (used by -L). Ada2Java can create two different kinds of library: dynamic or encapsulated. Dynamic is the default mode and generate a dynamic library which depends on GNATJNI and AJIS libraries on one hand, and the GNAT run-time on the other hand whereas an encapsulated library is autonomous: it contains all the necessary symbols from the three dependencies to be standalone.

--bound-package-root=<root package name> Set the name of the root glue Ada packages (default is JNI_Binding).

--bound-package-suffix=<package suffix> Set the suffix of the glue Ada packages (default is _JNI).

--no-monitor[-finalize]

--monitor[-finalize]-(check|protect)

Sets the default monitor for subprograms. See *Thread Safety*.

--[no-]attach-(parameter|access|controlling|ada2005)

Sets the default attachment policy. See *Managing Attachment to Java Proxies*

`--[no-]assume-escaped`

Controls whether checks for object ownership are enabled. See *Restrictions on Proxy-Owned Objects Passed to Subprograms*

`--[no-]java-enum`

Controls whether Java enumerations should be used to bind Ada enumerations, or if static integers should be used instead (Java enumerations are the default).

`--[no]unaliased-access`

Controls whether ada2java is allowed to create proxies on unaliased data. `--no-unaliased-access` is default. See *Aliasing*

Example:

```
$ ada2java -c mydir pack1.ads -b foo.bar
```

This results in the placement of the Java binding classes in the relative directory `mydir/foo/bar/`.

Note that the actual directory containing the generated Java classes will need to be on the `CLASSPATH` environment variable in order to successfully run a Java application that uses the binding.

3.2 Compiling and Running the Generated Code

3.2.1 Issues with the Ada Generated Code

Two sets of Ada units need to be compiled – the original packages and the generated ''glue" code. The Ada glue depends on the `ajis` project installed in the `lib/gnat` directory of the GNAT-AJIS installation.

It is highly recommended that you use the project generation switches `-L` (for a shared library) or `-M` (for an Ada main subprogram). However, even if these switches handle most cases, you may need to write your own build procedures to address more advanced usage. In such a situation please note that some compiler options may have an impact on the ajis library and thus need to be taken into consideration:

`-O2 -O3`

If you compile with a high optimization level, you should deactivate strict aliasing using the compiler switch `-fno-strict-aliasing`.

`-fstack-check`

The stack checking mechanism is based on signals that are deactivated by the GNAT AJIS library, so this switch will have no effect and should not be used.

`-fPIC`

On Linux / Solaris, all the code has to be relocatable, which is specified through the `-fPIC` switch. If you are creating a shared library that integrates components compiled externally, you have to ensure that they have been compiled using the `-fPIC` switch.

3.2.2 Compiling as an Ada Shared Library

The most common architecture of an Ada / Java program, and a Java / Native program in general, is to compile the native code into a shared library, and then load that shared library at run time. In this case, the main entry point is a Java main method, written by the developer.

In order to implement this scheme, you will need to create a SAL (Stand-Alone Library) project containing the sources of the input packages plus the 'glue', and use it to compile the library.

A simple standalone library project is generated if you use the `-L` switch. The generated project can then be compiled with `gprbuild`, for example:

```
$ ada2java my_package.ads -o ada -c java -P my_project.gpr -b base -L my_lib
$ gprbuild -p -P ada/my_lib.gpr
```

Note that the native library will then be loaded automatically by the generated Java glue code.

3.2.3 Compiling as an Ada Main Subprogram

If compiling the native code into a shared library is not practical, an alternative is to create an Ada main subprogram embedding a Java Virtual Machine.

`ada2java` provides an easy way to generate a project and an Ada main subprogram, through the `-M` switch. This switch takes the name of the main as parameter and will generate an Ada main that will automatically create a Java virtual machine, and then call a Java method defined as follows:

```
package <base_package>;

public class <main_name> {

   public static void main (String [] args) {
   }

}
```

This class (and thus the method implementation) has to be provided by the developer. If it is not present, the main subprogram will fail with an error at run time.

The generated main will look into the CLASSPATH environment variable to find the Java classes when initializing the Java virtual machine. So for example, if that you provide the following class:

```
package java_code;

import java_code.Test.Test_Package;

public class Main {

   public static void main () {
      Test_Package.Call_Something ();
   }

}
```

using the following Ada API:

```
package Test is
   procedure Call_Something;
end Test;
```

with the appropriate `test.gpr` project referencing the `Test` code, you will be able to compile and run the code as follows:

```
$ ada2java test.ads -P test -b java_code -o ada -c java -M Main
$ gnatmake -P ada/main.gpr
$ CLASSPATH=`pwd`:`pwd`/java:$CLASSPATH
```

```
$ export CLASSPATH
$ javac java_code/Main.java
$ ada/obj/main
```

You can explicitly specify the name of the Java main class to use, through the `--main-class` switch, e.g.:

```
$ ada2java test.ads -P test -b java_code -o ada -c java \
> -M Main --main-class=some.main.My_Main
```

In this case, the Ada main will look for a main subprogram in `some.main.My_Main`, instead of `java_code.Main`.

Note that you may need to define the `LPATH`, `LD_LIBRARY_PATH` or `PATH` environment variables so that the code can be compiled against `jvm.lib` or `libjvm.a`, and then run with `jvm.dll` or `libjvm.so`.

3.2.4 Compiling the Java Generated Classes

The Java application needs to load the library before any of the Ada subprograms are invoked. If you did not supply the `-L` switch to `ada2java`, then you will need to do this explicitly; conventional style is to invoke `System.loadLibrary ("<library-name>")` in a static initializer in the main Java class. This step is automated if you use the `-L` switch, as described above.

Before running the Java code, you need to ensure that the `CLASSPATH` environment variable contains both the directory of the generated Java code, and the JAR for the GNAT-AJIS-related predefined classes. The latter archive exists as `$GNATAJIS_INSTALL_DIR/lib/ajis.jar` where `GNATAJIS_INSTALL_DIR` is the root directory for the GNAT-AJIS installation.

3.3 Debugging an Ada / Java Application

The Ada code embedded in a Java application can be debugged through the standard GDB debugger. In order to do so, the following steps needs to be followed:

- Start the Java Virtual Machine on the Java application containing the Ada code to debug.
- Get the PID of the Java Virtual Machine. On Linux system, this is given by the ps command, on Windows, through the Task Manager.
- Start GDB giving the Java Virtual Machine as argument.
- Attach to the Java Virtual Machine, e.g.:

  ```
  $> attach <pid>
  ```

- You can then run a regular GDB session. For example, the following will set the GDB environment in Ada mode, break on all Ada exception, and then continue the application:

  ```
  $> set lang ada
  $> break exceptions
  $> continue
  ```

3.4 Pragma Annotate and ada2java

Pragma `Annotate` (see *GNAT Reference Manual*) has several uses in conjunction with the GNAT-AJIS tools, each with the form:

```
pragma Annotate (AJIS, *AJIS_annotation_identifier* {, *argument*});
```

GNAT-AJIS annotation names are defined in the package `AJIS.Annotations`, which is a part of the `ajis.gpr` project installed with GNAT-AJIS. You need to have visibility on this package using a `with` and possibly a `use` clause before being able to use these pragmas.

The following GNAT-AJIS annotation pragmas are supported:

- `Annotation_Renaming` – *Dealing with Name Clashes*
- `Assume_Escaped` – *Restrictions on Proxy-Owned Objects Passed to Subprograms*
- `Attached` – *Managing Attachment to Java Proxies*
- `Bind` *Removing function/procedure from binding layer*
- `Monitor` – *Thread Safety*
- `Rename` – *Dealing with Name Clashes*
- `Resolve_Ambiguous_Expression` – *Dealing with ambiguous operand in conversion errors*

CHAPTER

FOUR

MAPPING ADA TO JAVA

To allow an Ada package to be used from Java, `ada2java` generates one or more Java classes (source files that will need to be compiled to bytecodes by a Java compiler) based on the content of the visible part of the Ada package spec. This section explains and illustrates the mapping for each of the various kinds of entities declared in a package that can be used from Java.

In brief:

- Although there are some exceptions to this rule, in general a type and certain of its associated subprograms declared in an Ada package are mapped to a Java class with methods corresponding to the Ada subprograms. Such entities are said to be *attached* to the resulting class.
- Other entities declared in the Ada package map to static members defined in a 'default class' generated by `ada2java`. In particular, variables and constants in the Ada package map to private static fields in the default class and are accessed through ''getter" (and 'setter' for variables) methods. Such entities are said to be *unattached*.

If the default class is generated, its name is that of the original Ada package (with the same casing as the identifier in the package declaration) suffixed with `_Package`.

In the examples, only the portions of the Java classes needed by users of the classes are shown.

4.1 Types

Types used in the Ada package map to Java types in the generated class(es). This section explains the correspondence. As a general rule, note that while most forms of type declarations have a correspondence in Java, subtype declarations are ignored, as there is no equivalent to subtypes in Java. However, subtype constraints imposed on Ada entities, such as variables or formal parameters, must be respected when referenced from Java, and can result in exceptions when constraints are violated.

4.1.1 Scalar Types

The following table shows how Ada scalar types are mapped to Java primitive types:

Ada type	Java type
Integer type <= 32 bits	`int`
Integer type > 32 bits	`long`
`Boolean`	`boolean`
`Character`	`char`
Other enumeration type	`int`
Fixed-point type	`double`
Floating-point type	`double`

Constraint checks generated in the Ada glue code detect errors that may result from the range mismatches between Ada and Java. For example, since a 16-bit Ada integer will be mapped to 32-bit `int` in Java, the Java code might attempt to pass an out-of-range value to Ada. This will raise a `Constraint_Error` exception in Ada, which will be propagated back to Java as an `AdaException` exception.

For an enumeration type, a Java final class is created, with the same name as the enumeration type. This class defines the possible values for the enumeration.

Example:

```
package Pckg is
   type Enum is (A, B, C);
end Pckg;
```

will give:

```
package Pckg;

public final class Enum {
   public static final int A = 0;
   public static final int B = 1;
   public static final int C = 2;
}
```

Representation clauses for enumeration types are not currently supported.

A discussion of subprogram formal parameters of scalar types may be found in *Subprogram parameters*.

4.1.2 Arrays

Mapping Ada arrays to Java arrays would be very expensive, since it would imply a copy of the whole array each time a parameter has to be passed. Thus for efficiency an Ada array type is mapped to a dedicated 'proxy' class with methods that serve as accessors to attributes and components. For example:

```
package Ex1 is
   type T1 is array(Integer range <>) of Float;
end Ex1;
```

will yield the following class:

```
public final class T1 extends com.adacore.ajis.internal.ada.AdaProxy {
   ...
   public T1 (int First_1, int Last_1){...}

   final public double Get_Element_At (int Index_1){...}

   final public void Set_Element_At (int Index_1, double Value){...}

   final public int First (){...}
```

```
   final public int Last (){...}

   final public int Length (){...}
}
```

A subprogram that takes a parameter of the Ada array type is mapped to a method taking a parameter of the corresponding Java 'proxy' class; note that this method is located in the default class, and not in the proxy class.

4.1.3 Strings

Directly passing `String` data between Ada and Java would require expensive copying, and thus an alternative approach is used. The Ada type `String` is mapped to the Java class `AdaString`, which encapsulates the accesses.

More specifically, an Ada parameter of type `String` of any mode, and an Ada `access String` parameter, are both mapped to a Java parameter of type `AdaString`.

For efficiency, an `AdaString` object caches both its Ada and Java string values after they have been computed. As an example, if the Ada spec is:

```
package Pckg is
   procedure P (V : String);
end Pckg;
```

then the generated Java will be:

```
public final class Pckg_Package {
   public static void P (V : AdaString) {...}
}
```

If we now write:

```
AdaString str = new AdaString ("A string from Java");
Pckg_Package.P (str);
Pckg_Package.P (str);
```

Only the first call will require the expensive string translation from Java to Ada. The second invocation will directly use the cached value.

Please note that Java strings are UTF16-encoded, whereas the corresponding Ada strings will be UTF8-encoded. This may have significant impact when computing character offset on Java strings.

4.1.4 Simple Record Types

Simple (that is, not tagged) record types are mapped to Java final classes. Components are accessed through a set of generated accessors ('getter' / 'setter' methods). As a current limitation, `ada2java` does not yet support accessing discriminant components.

Example:

```
package Pckg is
   type R is
      record
         F1 : Integer;
         F2 : Float;
      end record;
end Pckg;
```

will give:

```
package Pckg;
public final class R {
   public R () {...}

   public final int F1 () {...}
   public final void F1 (int Value) {...}

   public final double F2 () {...}
   public final void F2 (double Value) {.../}
}
```

A component that has an access-to-record type is treated as though it were of the record type itself. For example:

```
package Pckg is
   type R is
      record
         F1 : Integer;
         F2 : Float;
      end record;
   type S is
      record
         G1 : R;
         G2 : access R;
      end record;
end Pckg;
```

will result in both a class `R` as above, and the following class `S`:

```
package Pckg;
public final class S {
   public S () {...}

   public R G1 () {...}
   public void G1 (R Value) {...}

   public R G2 () {...}
   public void G2 (R Value) {...}
}
```

Only one level of indirection is implemented; `ada2java` does not support access to access-to-record.

A private (untagged) type is treated like a record type, except that it does not have any component-accessing methods. (A later release of `ada2java` will generate methods for accessing discriminants if the type has any.)

4.1.5 Tagged Types

A tagged type is mapped to a Java class of the same name. If the Ada type is abstract, then the Java type will be abstract as well.

General principles

A primitive (i.e., dispatching) subprogram of a tagged type is mapped to a corresponding Java instance method. A current restriction is that the first parameter of the Ada subprogram must be a controlling parameter; otherwise the subprogram is mapped to a method in the default class. (Thus a function that delivers a value of the tagged type, but

has no controlling parameter, is mapped to a method in the default class, and not to a method in the class corresponding to the Ada type.) The first Ada parameter is mapped to the Java method's implicit `this` parameter.

A subprogram with a class-wide parameter is mapped to a method of the tagged type's Java class whose corresponding parameter has the Java class type. However, as this is not properly a dispatching primitive of the Ada type, it is declared as a final method.

As an example:

```
package Ex1 is
   type T is tagged null record;
   procedure P1 (X : in out T; F : Float);
   procedure P2 (X : T'Class);
   procedure Q1 (I : Integer; X : T);
   procedure Q2 (I : Integer; X : T'Class);
   function F return T;
end Ex1;
```

is mapped to:

```
public final class Ex1_Package {
   ...
   static public void Q1 (int I, Ex1.T X){...}

   static public void Q2 (int I, Ex1.T X){...}

   static public T F (){...}
} // Ex1_Package

public class T extends com.adacore.ajis.internal.ada.AdaProxy {
   ...
   public void P1 (double F){...}

   final public void P2 (){...}
} // T
```

Ada type hierarchies

Hierarchies of Ada types are preserved in the generated Java classes. Therefore, the following structure:

```
type R is tagged record;

type R_Child is new R with null record;
```

will result in:

```
public class R {...}

public class R_Child extends R {...}
```

Consistency of Java types is guaranteed at run time. For example, the following function:

```
package Pckg is
   function F return R'Class;
end Pckg;
```

will result in:

```
public final class Pckg_Package {
   public R F () {...}
}
```

However, if the actual type of the returned object is `R_Child`, then the value returned by the Java function will be of the Java type corresponding to `R_Child`.

Java class hierarchies

It is possible to extend a Java class that was generated by `ada2java` from an Ada tagged type.

For example:

```
package Rec_Pckg
   type Rec is tagged null record;
   procedure P (R : Rec);
end Rec_Pckg;
```

results in a Java class `Rec` with an instance method `P`:

```
class Rec extends com.adacore.ajis.internal.ada.AdaProxy {
   ...
   public void P(){...}
}
```

You can then write:

```
class Rec_Child extends Rec {
     public void P () {
        System.out.println ("Hello from Java");
     }
}
...
Rec ref = new Rec_Child();
ref.P();  // Displays "Hello from Java"
```

4.2 Global Variables and Constants

A package containing global variables (that is, variables declared in the package spec's visible part) is mapped to a default class containing 'getter' and 'setter' methods for accessing and updating the variables. Global constants are treated analogously, but they have only a 'getter' method. Variables of limited types also only have a 'getter' method. Note that Ada named numbers, which are really just values intended for use in static compile-time computations, are not mapped to Java.

For example:

```
package Globals is
   V : Integer;
   C : constant Integer := 100;
   N : constant := 3.14159;
end Globals;
```

will result in the default class:

```
public class Globals_Package {
```

```
   static public int V (){...}

   static public void V (int Value){...}

   static public int C (){...}
}
```

4.3 Subprograms

Ada procedures and functions are mapped to Java methods. Nondispatching subprograms are marked as `final`. Dispatching subprograms are discussed in *Tagged Types*.

4.3.1 Method placement

In general, nondispatching subprograms are mapped to methods defined in the default class.

For example, if the input package spec is:

```
package Pkg is
  function F return Integer;
end Pkg;
```

then `ada2java` will generate the following default class:

```
public class Pkg_Package{
   ...
   public static int F(){...}
}
```

However, there are cases where the subprogram can be attached to the class of its first parameter. Attachment can be enabled / disabled depending on user requirement. In this case, the explicit initial Ada parameter is mapped to the implicit `this` parameter in Java. See *Managing Attachment to Java Proxies* for further details.

4.3.2 Subprogram parameters

The following rules and restrictions apply to the types of subprogram formal parameters:

- Scalar types
  - Access-to-scalar types are not supported.
  - A scalar type with mode `in` is mapped to the corresponding Java type. For example:

    ```
    procedure P (V : Integer);
    ```

    will result in:

    ```
    public void P (int V) {...}
    ```

  - A formal scalar with mode `out` or `in out` will be mapped to a corresponding 'wrapper' class: `BooleanRef`, `CharacterRef`, `DoubleRef`, `IntegerRef`, and `LongRef` respectively encapsulating the primitive type `boolean`, `character`, `double`, `int` and `long`).

    Each of these classes defines `setValue` and `getValue` methods for accessing the encapsulated value.

The Java application needs to construct an object of the relevant wrapper class and pass it to the method that corresponds to the Ada subprogram. After the return from the method, the Java application can invoke the `getValue` method to retrieve the new value of the actual parameter.

- Record and private types
  - An Ada `in`, `in out`, or `out` formal parameter of a record or private type (either tagged or untagged), is mapped to a Java formal parameter of the class corresponding to the Ada type. Similarly, an Ada formal parameter of an access-to-record-type or access-to-private-type (either anonymous or named) is mapped to a Java formal parameter of the class corresponding to the Ada type.

    Example:

    ```
    package Example is
       type R is null record;
       type Access_R is access all R;
       procedure P(V1 : R;
                   V2 : out R;
                   V3 : in out R;
                   V4 : access R;
                   V5 : Access_R);
    end Example;
    ```

    The resulting Java class is:

    ```
    public final class Example_Pckg {
       ...
       public void P (R V1, R V2, R V3, R V4, R V5){...}
    }
    ```

  - An Ada `out` or `in out` parameter of an access-to-record (or access-to-private) type is mapped to a nested class. For example:

    ```
    package Example is
       type R is null record;
       type Access_R is access all R;
       procedure P(V : out Access_R);
    end Example;
    ```

    will generate the default class and a class for `R`

    ```
    public class Example_Package {
       ...
       static public void P (R.Ref V){...}
    }

    public class R extends com.adacore.ajis.internal.ada.AdaProxy {
       ...
       public static class Ref implements com.adacore.ajis.IProxyRef {
          public void setValue (Object r) {...}
          public Object getValue () {...}
       }
    }
    ```

    The Java application needs to construct an object of the class `R.Ref` and pass it to `P`. On return, the `getValue` method may be called to retrieve the value in the `out` parameter returned by the Ada procedure.

  - Further indirection, such as an access-to-access type for a formal parameter, is not supported.

4.3.3 Overloaded operators

Java doesn't allow operators overloading. When operators are overloaded in Ada, the corresponding Java name is set by the binding generator to `OP_<operator_name>`. For example:

```
type Complex is record ...

function "+" (Left, Right : T) return T;
```

will generate on the Java side:

```
public class Complex {

   public Complex OP_PLUS (Complex Right) {
   ...
   }

}
```

Here's a list of the equivalence between Ada operators and Java names:

Ada operator	Java name
=	OP_EQUAL
>	OP_GT
<	OP_LT
>=	OP_GE
<=	OP_LE
or	OP_OR
and	OP_AND
xor	OP_XOR
+	OP_PLUS
-	OP_MINUS
/	OP_DIV
*	OP_MUL
**	OP_EXP

4.4 Subprogram Access Types

Accesses to subprograms - sometimes referred to as callbacks - can't be directly bound to Java. It is not possible to give a reference to a Java function in a type-safe fashion. ada2java generates an abstract class with an abstract member of the correct profile for each access type to be bound, the implementation of its abstract primitive being the implementation of the subprogram access. For example:

```
type P_Acc is access all procedure (V : Integer);

procedure Call_P_Acc (Proc : P_Acc);
pragma Annotate (AJIS, Assume_Escaped, False, Call_P_Acc, "Proc");
```

will be bound into:

```
abstract public class P_Acc {
   abstract public P_Acc_Proc (int V);
}

void Call_P_Acc (P_Acc Proc);
```

and can be used in, for example, the following scenario:

```
Proc p = new Proc () {

   public P_Acc_Proc (int V) {
      System.out.println ("CALLED WITH " + V);
   }

};

Pckg_Package.Call_P_Acc (p);
```

Note the use of the pragma `Annotate` on the `Call_P_Acc` method. The Java implementations of bound subprogram access types are not actually accesses to subprograms, but instances of Java objects. It's not possible to store such an object on the Ada side afterwards, since the complete information can't be kept in the access type. Therefore, the programmer must ensure that no escape of the value is done, and take responsiblity for that by declaring the parameter as being not escaped. Further details on escapement can be found in *Restrictions on Proxy-Owned Objects Passed to Subprograms*.

4.5 Exceptions

Exceptions are bound into classes derived from `com.adacore.ajis.NativeException` It is then possible to throw or handle them directly in Java code.

Example:

```
package Example is
   An_Exception : exception;
   procedure Raise_An_Exception;
end Example;

package body Example is
   procedure Raise_An_Exception is
   begin
      raise An_Exception;
   end Raise_An_Exception;
end Example;
```

The resulting Java class is:

```
public final class An_Exception extends com.adacore.ajis.NativeException {
   ...
}
```

And can be used in e.g.:

```
try {
   Example_Package.Raise_An_Exception ();
} catch (An_Exception e) {
  // process the exception
}
```

4.6 Renamings

Renamings of objects and subprograms are supported by `ada2java`. Object renamings are mapped in the same way as global objects, by means of 'setter' and 'getter' methods in the default class for the containing package. A subprogram renaming is represented by a method with the name of the renaming that invokes the renamed subprogram, declared in the appropriate class. In other words, the same rules that apply to other subprograms apply to subprogram renamings.

4.7 Generics

Generic packages and subprogram can't be directly bound to Java. However, packages and subprograms instances and bound like regular packages and subprograms.

4.8 Predefined Environment

In order to access descendants of `Ada` or `GNAT` from Java, you need to manually invoke `ada2java` on the Ada source files from the GNAT installation directories, to generate the corresponding Java binding classes. This step will be automated in a future release of GNAT-AJIS.

4.9 Current Limitations

The following features are not supported:

- *Discriminants.* Discriminants are not accessible from the Java class generated for a discriminated type.
- *Anonymous arrays.* Objects with an anonymous array type are not supported, but array type declarations which declare a constrained first subtype are supported.
- *Interfaces.* No mapping is currently provided from Ada interface types to Java interfaces.
- *Tasking features.* Tasks and protected objects/types are ignored.

CHAPTER

FIVE

ADVANCED ADA2JAVA TOPICS

This chapter discusses a number of issues that `ada2java` users should be aware of.

5.1 Dealing with Name Clashes

If Ada subprograms from the same package spec produce the same Java profile, the binding generator will detect the problem and generate only the first entity. Other entities of similar name will be ignored with a warning. To prevent this, you can use the `Rename` pragma to define the Java name corresponding to an Ada entity:

The binding generator may generate code containing *ambiguous operand in conversion* errors. See *Dealing with ambiguous operand in conversion errors* to solve these errors.

```
pragma Annotate (AJIS, Rename, <identifier>, <static_string_expression>);
```

The `<identifier>` argument denotes the Ada entity. The `<static_string_expression>` is the name that will be used for the corresponding Java entity generated by `ada2java`.

Example:

```
package Example is
  type I1 is new Integer;
  type I2 is new Integer;

  function F return I1;
  pragma Annotate (AJIS, Rename, F, "F_I1");

  function F return I2;
  pragma Annotate (AJIS, Rename, F, "F_I2");
end Example;
```

This will result in two Java functions:

```
int F_I1 ();
int F_I2 ();
```

Adding a pragma to an Ada package specification is not always practical, and indeed may be impossible if the specification is from an external library. With GNAT-AJIS, you can provide the pragma in a separate Ada file, applying it to an Ada entity that is itself a renaming declaration.

This entity must be marked by another `Annotate` pragma:

```
pragma Annotate (AJIS, Annotation_Renaming, <identifier>).
```

where <identifier> denotes an entity defined by an Ada renaming declaration. The Annotation_Renaming pragma applies to all AJIS pragmas that are specified for <identifier>.

```
package Example is
  type I1 is new Integer;
  type I2 is new Integer;

  function F return I1;
  function F return I2;
end Example;

with Example;
package Renamings is
  function F return Example.I1 renames Example.F;
  pragma Annotate (AJIS, Annotation_Renaming, F);
  pragma Annotate (AJIS, Rename, F, "F_I1");

  function F return Example.I2 renames Example.F;
  pragma Annotate (AJIS, Annotation_Renaming, F);
  pragma Annotate (AJIS, Rename, F, "F_I2");
end Renamings;
```

Entities annotated with pragma Annotate(AJIS, Annotation_Renaming) will not be mapped to Java entities; they are assumed to be used only to define annotations.

5.2 Dealing with ambiguous operand in conversion errors

The Ada code created by the binding generator may contain "ambiguous operand in conversion" errors.

To solve them:

- ask the binding generator to generate Ada code resolving the ambiguous expression (using Resolve_Ambiguous_Expression AJIS Annotate pragma)
- remove the unused possible interpretation(s) from the generated code. (see *Removing function/procedure from binding layer*)

```
pragma Annotate (AJIS, Resolve_Ambiguous_Expression, <identifier>, True);
```

The <identifier> argument denotes the Ada entity to bind using extra code to resolve the "ambiguous operand in conversion" error.

Example:

```
with AJIS.Annotations; use AJIS.Annotations;

package Example is

  type T is tagged null record;

  function F return T;
  pragma Annotate (AJIS, Resolve_Ambiguous_Expression, F, True);

  function F return Integer;

end Example;
```

Adding a pragma to an Ada package specification is not always practical, and indeed may be impossible if the specification is from an external library. With GNAT-AJIS, you can provide the pragma in a separate Ada file.

```
with AJIS.Annotations; use AJIS.Annotations;

with Example;

package Config is

  function F return Hello.T renames Hello.F;
  pragma Annotate (AJIS, Resolve_Ambiguous_Expression, F, True);

end Config;
```

5.3 Removing function/procedure from binding layer

Any unused function/procedure can be removed from binding using the Bind AJIS Annotate pragma.

```
pragma Annotate (AJIS, Bind, <identifier>, False);
```

The `<identifier>` argument denotes the Ada entity to remove from the binding layer.

Example:

```
with AJIS.Annotations; use AJIS.Annotations;

package Example is

  type T is tagged null record;

  function F return T;
  pragma Annotate (AJIS, Bind, F, False);

  function F return Integer;

end Example;
```

Adding a pragma to an Ada package specification is not always practical, and indeed may be impossible if the specification is from an external library. With GNAT-AJIS, you can provide the pragma in a separate Ada file.

```
with AJIS.Annotations; use AJIS.Annotations;

with Example;

package Config is

  function F return Hello.T renames Hello.F;
  pragma Annotate (AJIS, Bind, F, False);

end Config;
```

5.4 Memory Model

An object on the Ada side – a so-called *native object* – is accessed in Java through a *proxy object*: an instance of a class (the *proxy* class) generated from the Ada object's type.

The proxy object contains a reference to the native object. Invoking a member function on the proxy object results in a call of a native subprogram on the native object. The native object may be either allocated or declared.

This section explains the implications of this model on the usage of the generated binding.

5.4.1 Requirements for Non-null Parameter Values

If an Ada subprogram's formal parameter is not an access parameter (i.e., it has `in`, `in out` or `out` mode), then invoking the corresponding Java method requires a non-null reference to a proxy object. For example:

```
package P is
   type T is null record;

   procedure Proc (V : out T);
end P;
```

results in the following Java classes:

```
public class T {
   ...
}

public final class P_Package {
   public static Proc (T V) {
      ...
   }
   ...
}
```

You need to ensure that the `V` parameter passed to `Proc` is not `null`. Hence, the following will throw an exception, since `v` is implicitly initialized to `null`:

```
T v;
P_Package.Proc (v);
```

A simple way to provide a non-null reference is to initialize it to an allocated object:

```
T v = new T ();
P_Package.Proc (v);
```

5.4.2 Allocating Ada Objects from Java

If `A` is an Ada type that is mapped by `ada2java` to a Java class `J`, then the execution of a Java constructor `J()` will create two objects:

- A Java object (the 'proxy' object) of class `J`, allocated on the Java heap and subject to Garbage Collection by the JVM, and
- An Ada object of type `A`, allocated on the Ada heap, referenced from the proxy object.

5.4.3 Automatic Creation of Native Objects

Under certain circumstances, the generated Java code may construct extra native objects on the Ada heap. To help explain this, here is an example where such allocation is not needed, namely, a function returning an access value:

```
package P is
   type T is record ... end record;

   type T_Acc is access all T;
```

```
   function Create_T_Acc return T_Acc;
end P;
```

The generated code will look like:

```
public class T { ...}

public final class P_Package {
   final public T Create_T_Acc () { ...}
   ...
}
```

On the Java side, the method `Create_T_Acc` returns a proxy object that contains the value of the pointer returned by the call of the Ada function. So the user can write:

```
T v = P_Package.Create_T_Acc ();
```

and then access the data of `v`. Note that all the standard precautions that apply in using an Ada pointer have to be taken in this case as well. In particular, after the object has been freed on the Ada side, there should be no further references to it from Java. The Ada programmer needs to document how the returned value should or should not be used, and the Java programmer needs to adhere to these guidelines.

However, a function returning a record rather than an access value raises additional issues:

```
package P is
   type T is record ... end record;

   function Create_T return T;
end P;
```

results in a default class similar to the one in the previous example:

```
public final class P_Package {
   final public T Create_T () { ... }
   ...
}
```

The Java programmer can write:

```
T v = P_Package.Create_T ();
```

But in this case, the value returned by the function is not a pointer. A new Ada native object is automatically allocated on the heap, initialized to a copy of the value returned by the Ada function, and is referenced through the proxy object constructed by the Java method `Create_T`. This is similar to the Ada situation where calling `Create_T_Acc` would not involve a copy of a `T` object, whereas `Create_T` would.

Because of implementation constraints there may be more than one copy involved in the call of `Create_T` (but only one native object will be created on the heap). This will be discussed further in *Clone and Copy Semantics*.

5.4.4 Native Ownership

As illustrated above, native objects may be created automatically by Java methods corresponding to Ada subprograms. This raises the issue of when/how such objects are to be deallocated.

In general, the `ada2java` approach is based on the following principles:

- The environment (Java or Ada) that allocates an object is responsible for its deallocation;
- In Java, all deallocation is performed implicitly, by the Garbage Collector;

- In Ada, the programmer is responsible for manually deallocating the objects;
- Dangling references should be prevented; i.e., the Ada object should not be deallocated as long as there are still live references to the object.

A native object created from Java is said to be *owned* by its proxy object. It has been created by the Java program, and it must be freed by the Java environment.

Such an object is tightly linked to its proxy – when the proxy object doesn't exist (i.e. is garbage collected), the native object becomes inaccessible. [1]

The native object deallocation will occur automatically, when the Java proxy is garbage collected (more specifically through the implementation of the `finalize` method).

A native object not owned by a proxy – for example, one that has been obtained from the Ada API through an Ada pointer – will not be deleted automatically.

The ownership state of a native object may be queried through the `getOwner` method.

This function returns a value from the enumeration `com.adacore.ajis.IProxy.Owner`, either `NATIVE` or `PROXY`, specifying who is responsible for managing the memory. For example after these assignments:

```
T v1 = new T ();
T v2 = P_Package.Create_T_Acc ();
T v3 = P_Package.Create_T ();
```

the following relationships hold:

`v1.getOwner ()`	`PROXY`
`v2.getOwner ()`	`NATIVE`
`v3.getOwner ()`	`PROXY`

Native objects referenced by `v1` and `v2` will be deallocated when the corresponding Java proxy is garbage-collected.

You may change the owning attribute of a referenced native object, through the `setOwner` method of proxy classes.

This should be used very carefully, as it may generate memory leaks or corruption. Doing the following:

```
T v1 = new T ();
v1.setOwner (Owner.NATIVE);
```

will deactivate the object deallocation on finalization. The Java programmer becomes responsible for explicitly deallocating the native object.

Note that, while moving the owning from PROXY to NATIVE is a relatively safe operation, raising the possibility of memory leaks but not object corruption, moving in the opposite direction should be done with great care. A object managed by the NATIVE side may be deallocated by the Ada application at any moment, or be declared as the field of another object or even on the native stack in situations such as callbacks. If it's not clear where the object is coming from, cloning it (resulting automatically into an object managed by the proxy) is often the best solution.

As will be described below, only objects that are known to be managed though pointers can have their ownership changed. An object obtained, for example, from a global variable or field will need to retain its known ownership.

5.4.5 Object Allocators

The generated code keeps track of how native objects have been obtained, and restricts their operations accordingly. Three possible allocators are recognized:

DYNAMIC Such objects are created either through bound constructors, accessed from native access types, or through an automatic copy from the bound code. The ownership of dynamic objects can be changed.

[1]

STATIC Such objects are accessed from global variables, fields, or callback parameters that are not access types. The ownership of these objects is always `NATIVE` and cannot be changed.

UNKNOWN In some cases, for example on copying values of tagged types, it is not possible to determine whether the object has been allocated statically or dynamically. In such cases you can access the allocator through which an object is referenced using the `IProxy.getAllocator ()` function.

5.4.6 Restrictions on Proxy-Owned Objects Passed to Subprograms

Passing a (reference to a) native object as an actual parameter to an Ada subprogram whose corresponding formal parameter is either an access parameter or of an access type could in some cases result in a dangling reference to the native object, whereas in other situations it might be harmless. In order to provide the desired safety while allowing the needed generality, `ada2java`'s approach is similar to the way that the Ada standard supplies both the `'Access` and `'Unchecked_Access` attributes for composing pointers to data objects.

In summary, when the Java side is responsible for pointer memory management (i.e. the object is owned or it is accessed through `'Access` or `'Unrestricted_Access`), then an exception will be thrown in Java on an attempt to pass a reference to a native object as an actual parameter when the formal is of a named or anonymous access type. However, this check can be suppressed either locally (for a given formal parameter) or globally (for all calls).

Here is a summary of the troublesome scenario:

- A reference to a proxy object is passed as a parameter to a Java method, and the reference to the corresponding native object is then passed to a native Ada subprogram whose formal is an access parameter or of an access type;
- The Ada subprogram copies the parameter to a global variable and ultimately returns;
- The proxy object ultimately becomes inaccessible and is garbage collected by the JVM;
- The native object is deallocated as an effect of the proxy object's finalization;
- The deallocated object is still designated by the global variable.

To prevent this, an exception is thrown at step 1 above.

Here is an example:

```
-- Original Ada package
package P is
   type T is
      record
         A, B : Integer;
      end record;

   type A_T is access all T;

   G : A_T;

   procedure Set_G (V : A_T);
end P;

package body P is
   procedure Set_G (V : A_T) is
   begin
      G := V;
   end Set_G;
end P;
```

```
// Java statements
T v = new T ();
P_Package.Set_G (v); // Throws an exception
```

If the invocation of `Set_G` did not throw an exception, `G` would become a dangling pointer when the proxy object referenced by v is garbage collected.

However, if the invocation of `Set_G` did not cause an assignment of its formal parameter `V` to a global variable, then passing `v` as an actual parameter would be harmless. `ada2java` allows the Ada API to enable such uses, and thus to deactivate the check:

```
pragma Annotate (AJIS, Assume_Escaped, <Condition>, <Subprog>, <Formal_Param>)
```

where:

- `<Condition>` is either `True` or `False`
- `<Subprog>` is the name of the Ada subprogram
- `<Formal_Param>` is the affected formal parameter, expressed as a `String` literal

When `<Condition>` is `False`, the Ada subprogram is responsible for ensuring that the formal parameter is not copied to a global variable, or used to set a field of a structure that will be used outside of the scope of the subprogram. When `<Condition>` is `True`, an exception is thrown when a proxy object is passed as an actual parameter to the `<Formal_Param>` parameter of `<Subprog>`.

Here is an example:

```
package P is
   type T is
     record
        A, B : Integer;
     end record;

   type A_T is access all T;

   G : A_T;

   procedure Safe_Set_G (V : A_T);
   pragma Annotate (AJIS, Assume_Escaped, False, Safe_Set_G, "V");
end P;

package body P is
   procedure Safe_Set_G (V : A_T) is
   begin
      G := new T'(V.all);
   end Safe_Set_G;
end P;

*// Java statements*
T v = new T ();
P_Package.Safe_Set_G (v); *// OK*
```

The invocation `Safe_Set_G (v)` is safe since the Ada subprogram does not let the formal parameter 'escape'.

The 'escaped' checks can be globally activated through the `ada2java` switch `--assume-escaped`, and globally deactivated through the switch `--no-assume-escaped`.

The default is `--assume-escaped`. An explicit pragma overrides the global configuration switch.

Note that canceling escape checks should be done with great care, as there is no way to ensure that no escaping has occurred. In many situations, the programmer's intent is to create an object on the Java side and then store the object

on the native side. In such a case, canceling proxy ownership through the `IProxy.setOwner` method will have the desired effect, for example:

```
// Java statements
T v = new T ();
v.setOwner (Owner.NATIVE);
P_Package.Set_G (v); // OK
```

5.4.7 Parameter Mode Documentation

ada2java documents the mode of the parameters generated by the binding. This mode is only documented for proxies, not scalar values. The following modes are possible:

passed by value The object is passed by value. In calls from Java to Ada, it means that the proxy cannot be null. Proxy or native ownership doesn't matter. In callbacks implemented in Java, proxies for these parameters will be natively-owned and statically allocated. For returned value, the returned Ada object is copied to the returned proxy.

passed by reference (escapable) The object is passed by reference, and potentially escaped by the Ada code. The object has to be natively owned and dynamically allocated.

passed by reference (non escapable) The object is passed by reference, but is known not to be escaped by the native code. Any proxy can by passed to parameters of that type. When implementing a callback with a parameter of this type, or for returned values, the proxy will be natively owned and dynamically allocated.

passed by reference (static) The object is passed by reference, but its reference is coming from a location place, typically a global variable or a field. This only applies to returned values. The returned proxy will be natively owned and statically allocated.

5.5 Aliasing

As explained in the previous chapter, Ada object are managed through Java proxies. The creation of such proxies is obtained through an access type. For example, in the following code:

```
package P is
   type T is
     record
        A, B : Integer;
     end record;

   V1 : aliased T;

   type T2 is record
      V2 : aliased T;
   end record;

   type A is array (Integer range <>) of aliased T;

   V3 : aliased A (1 .. 10);
end P;
```

the binding generator will generate accessors to V1, V2 and V3 so that fields of T can be modified directly. It's possible to write:

```
V1 ().A (1);

T2 v = new T2 ();
v.A (2);

V3 ().Get_Element_At (10).A (3);
```

Accessing variables generate a proxy pointing directly to the address in memory. These proxies can be manipulated on their own, as any other, for example:

```
T ptr = V1 ();
ptr.A(1);
```

In the above example, ptr is a Java proxy containing a pointer to the global variable V. Such a pointer would typically be obtained through a `'Access` applied on the Ada variable V1.

It's sometimes unconvenient to declare aliased data just for the purpose of binding generation. GNAT offers a mechanism to retreive an access to any piece of data, `'Unrestricted_Access`. The use of such method is however dangerous. On certain architectures, e.g. *sparc-solaris*, forcing to retreive an access on a missaligned data will issue a program crash. In order to avoid that, by default, ada2java does not generate such accesses and create proxies only on aliased data.

This behavior can be deactivated with the ada2java argument `--unaliased-access`, which will allow access to unaliased data. Making data aliased should always be prefered to the use of that option.

5.6 Thread Safety

By default, the generated Java code is thread-safe, with locking logic that prevents multiple Java threads from accessing native methods at the same time. In summary, a Java method in a proxy object acquires a lock (a binary semaphore) before invoking the corresponding native method, and releases the lock when the native method returns (either normally or abnormally). The semaphore is global versus per Ada package; e.g. different Java threads are not allowed to invoke native methods simultaneously even if the native methods correspond to subprograms from different packages.

Even if the Java application does not explicitly create any threads, there are still two threads – the main (user) thread and the garbage collector; thus locking is needed in this case also.

The default locking behavior is not always appropriate, however. In particular, if the native code is using tasking, with mutual exclusion enforced on the Ada side, then there is no need for locking in the Java code (and indeed such locking could have undesirable consequences including deadlock).

User control over the generation of locking code is obtained through the following pragma:

```
pragma Annotate (AJIS, Locking, <Subprogram>, <LockControl>);
```

where `<Subprogram>` is the subprogram name, and `<LockControl>` is one of `Disable`, `Check` and `Protect`.

`Protect` Default setting. The generated code automatically brackets each invocation of the named native `<Subprogram>` within a `lock ... unlock` region. If a thread `t1` invokes `<Subprogram>` while some other thread `t2` holds the lock, then `t1` will be queued until the lock is released. Thus simultaneous calls of native methods are permitted but will entail queuing.

`Disable` The generated code contains no locking around invocations of `<Subprogram>`, and one thread may invoke `<Subprogram>` while some other thread is executing a native method (even one whose `LockControl` is `Protect`.

Check The generated code contains no locking around invocations of <Subprogram>, and the application must ensure that any such invocation is within a lock ... unlock region. An exception is thrown if a thread invokes <Subprogram> without holding the lock.

The following examples illustrate the multithreading behavior and the effects of the <LockControl> argument to the pragma.

```
package P is
   procedure P1 (I : Integer);
   pragma Annotate (AJIS, Locking, P, Disable);

   procedure P2 (I : Integer);
   pragma Annotate (AJIS, Locking, P, Disable);
end P;
```

Concurrent invocations of P1 and P2 are allowed (i.e., locking code is not generated automatically).

Code protected by the lock can be provided by hand by the Java developer. The lock is created in the library class generated for the binding, <base_package>.Ada2Java.Library, under the identifier lock. Thus:

```
import base.P_Package;
import base.Ada2Java.Library;

public class Main {
   public static void main (String [] args) {
      Library.lock.lock ();

      try {
         P_Package.P1 (0);
         P_Package.P2 (0);
      } finally {
         Library.lock.unlock ();
      }
   }
}
```

Following standard Java coding style, the lock / unlock logic should always appear in a try ... finally block, in order to release the lock even if an unexpected exception is propagated.

In certain cases, locking is required but the logic is more complex than simply protecting each native method invocation with a semaphore. For example, it may be necessary to invoke a sequence of native methods as an atomic action. This effect can be achieved through the Check setting for *LockControl*:

```
package P is
   procedure P1 (I : Integer);
   pragma Annotate (AJIS, Locking, P1, Check);

   procedure P2 (I : Integer);
   pragma Annotate (AJIS, Locking, P2, Check);
end P;
```

The Java program has to acquire the library lock before attempting any native call. Invoking P1 or P2 outside of a section protected by the lock will throw a Java exception.

The locking behavior can be changed globally through the ada2java --[no-]locking switch. More specifically, here are the permitted values for this switch:

--locking-protect Default setting; globally sets *LockControl* as Protect for all subprograms.

--locking-check Globally sets *LockControl* as Check for all subprograms.

--no-locking Globally disables locking (i.e., sets *LockControl* as Disable for all subprograms).

Note that an AJIS *Locking* pragma takes precedence over the global switch.

The `finalize` method invoked during garbage collection does not correspond to a subprogram from the Ada package that is input to `ada2java`, and it is not affected by the `--[no-]locking` switch. Instead, the locking logic used for the `finalize` call of Java proxies during garbage collection is determined by the `--[no-]locking-finalize` switch:

`--locking-finalize-protect` Default setting; sets *LockControl* as `Protect` for `finalize`.

`--locking-finalize-check` Sets *LockControl* as `Check` for `finalize`.

`--no-locking-finalize` Disables locking (i.e., sets *LockControl* as `Disable` for `finalize`)

A typical usage of these two switches would be to set `Check` as the *LockControl* for all subprograms, and `Protect` as the *LockControl* for `finalize` methods:

```
ada2java --locking-check --locking-finalize-protect p.ads
```

5.7 Proxies and Native Object Equality

Proxy classes are generated with an `equals` implementation that calls the corresponding Ada = operation. For example:

```
package P is
   type T is
      record
         F : Integer;
      end record;
end P;
```

The proxy class can be used as follows:

```
T v1 = new T ();
T v2 = new T ();
v1.F (0); // "setter" method for field F
v2.F (0); // "setter" method for field F
```

and now the result of `v1.equals (v2)` is `true`. This corresponds to 'shallow' equality, in contrast with == which tests pointer identity.

A new proxy is created each time a native function returns a pointer. For example:

```
package P is
   type T is null record;
   type A_T is access all T;
   G : A_T := new T;
   function Return_G return A_T;
end P;

package body P is
   function Return_G return A_T is
   begin
      return G;
   end Return_G;
end P;

AT p1 = P_Package.Return_G ();
AT p2 = P_Package.Return_G ();
```

Now `p1 == p2` is `false`, since a new proxy is created by each function return, but `p1.equals (p2)` is `true`,

The association between a proxy and its native object is lost when the proxy is passed to a native method. For example:

```
package P is
   type T is null record;
   type A_T is access all T;
   function Return_This (This : A_T) return A_T;
end P;

package body P is
   function Return_This (This : A_T) return A_T is
   begin
      return This;
   end Return_G;
end P;

T v = new T ();
```

Now the result of `(v == P_Package.Return_This (v))` is `false`

Java reference equality has special semantics in the case of cross-language inheritance, due to the use of a shadow native object – see *Shadow Object Equality* for more details.

5.8 Clone and Copy Semantics

A proxy class generated from a non-limited Ada type includes a `clone` method. The base class for all proxies, `AdaProxy`, implements the `Cloneable` interface, and defines a public method `clone`.

The `clone` method performs a 'shallow copy' of all the fields, except for the native object reference. The native object reference in the cloned proxy is a pointer to a newly allocated Ada object. This new native object is itself a shallow copy of the original native object.

Thus cloning a proxy does not result in the sharing of the native object by the original proxy and the cloned proxy. The latter points to a new native object. This behavior is needed to avoid a dangling reference (to the native object) when the original proxy is garbage collected.

If the proxy class corresponds to a limited type, then the generated clone method will throw an exception.

Additional semantics for clone and copy, in connection with cross-language inheritance, are covered below (see *Shadow Object Cloning*).

5.9 Cross-Language Inheritance

This section discusses a number of issues related to cross-language inheritance; i.e., defining a Java class as an extension of a proxy class for an Ada tagged type.

5.9.1 Inheriting from a Java Proxy

As explained in *Tagged Types*, `ada2java` maps an Ada tagged type to a non-final Java proxy class. You can extend this class in Java. For example:

```
package P is
   type T is tagged null record;
```

```
   procedure Prim (V : T);
end P;
```

results in the following Java class:

```
public class T {
   public void Prim () {
      ...
   }
   ...
}
```

which may be extended, with the instance method overridden:

```
public class T_Child extends T {
   public void Prim () {
      ...
   }
}
```

An object of the subclass `T_Child` is also a proxy; constructing such an object allocates a native object, referred to as a 'shadow native object'. Its properties will be described below (*The Shadow Native Object*).

The Java program can then invoke the `Prim` method, with standard Java dispatching behavior. For example:

```
T v1 = new T ();
T v2 = new T_Child ();

v1.Prim (); // Will call the native Prim
v2.Prim (); // Will call the overridden Java Prim
```

5.9.2 Cross Language Dispatching from Ada

A method overridden in Java can be called in Ada using the usual Ada dispatching mechanism. For example:

```
package P is
   type T is tagged null record;

   procedure Prim (V : T);

   procedure Call_Prim (V : T'Class);
end P;

package body P is

   procedure Prim (V : T) is
   begin
      --  Native Prim implementation;
   end Prim;

   procedure Call_Prim (V : T'Class) is
   begin
      Prim (V); -- Dispatching call.
   end Call_Prim;

end P;
```

The invocation of `Prim` in `Call_Prim` is dispatching, and the `Prim` for the type of the actual parameter will be called. In Java, this procedure can be used in conjunction with a cross-language extension of T, e.g.:

```
class T_Child extends T {
   public Prim () {
      ...
   }
}

T v = new T_Child ();
P_Package.Call_Prim (v);
```

The Java program is invoking the native Ada `Call_Prim` procedure, which in turn dispatches to the Java method `Prim` in `T_Child`.

5.9.3 The Shadow Native Object

This section describes in more detail the semantics of the shadow native object.

Basic Properties

As seen above, cross-language dispatching is supported; an Ada dispatching call may result in the invocation of a Java method on a proxy object. This is possible because of the *shadow native object* concept.

For any tagged type `T` declared in a package spec that is input to `ada2java`, a new type is automatically generated that extends `T` with a component that references a proxy object. The Ada declaration is:

```
type Shadow_T is new T with
   record
      Link_To_Proxy : Java_Object;
   end record;
```

If `T_Child` is a Java class that extends `T`, then constructing an instance of `T_Child` will create a native `Shadow_T` object instead of a regular `T` object. This type overrides every controlling primitive of T, and delegates the dispatching to the Java side. If a method corresponding to an Ada primitive operation is not overridden in Java, then the subprogram from the parent Ada type will be automatically called.

The use of a shadow object introduces a tight relationship between a Java proxy and its Ada native object. Basically, there is a roundtrip dependency between the two, so that a Java proxy object is associated with a unique Ada shadow native object and vice versa.

This has several non-trivial implications that are described in the sections below.

Memory Management

The reference from the shadow native object to the Java object is called a *global reference*. [2]

There are two kinds of global references: regular (usually simply referred to as global references) and weak. A regular global reference prevents the object from being garbage collected, whereas a weak global reference does not.

When a native object is owned by its proxy, the proxy is responsible for releasing the native memory. In this case, the reference from the native object to the proxy is a weak global reference: garbage collection will not be prevented, and both the proxy and then the native object will be released upon collection.

However, when the native object is owned by the native side, then the native object may continue to exist even if the proxy has become inaccessible from Java. In such a case, a global reference is used, so that the garbage collector is prevented from collecting the Java object. Such a global reference will be automatically released when the Ada object is actually deallocated.

[2]

Forcing the deallocation of the native object through `IProxy.deallocateNativeObject` will release the global reference as well.

Switching the owner of the native object between NATIVE and PROXY will switch the reference from a regular global reference to a weak one.

In order to avoid a potential memory leak, the link between the shadow native object and the Java proxy has to be broken manually when the shadow native object is no longer needed on the Ada side. This may be done in two ways:

- Deallocating the native shadow object, from Ada.
- Invoking the `unlink` method on the proxy, from Java

Shadow Object Equality

Java reference equality (==) is not consistent with native pointer equality, but Java object equality (`equals`) is. That is, if `A` and `B` are two native pointers where `A = B`, then on the Java side `A.equals(B)` is `true` but `a==b` is `false`.

However, since the association between shadow object and proxy is one-to-one, proxy equality is meaningful. For example:

```
package P is
   type T is tagged null record;
   type T_Access is access all T'Class

   procedure Identity (Item : T_Access) return T_Access;
   -- Returns Item as its result
end P;
```

On return from `Identity`, the 'glue code' between Java and Ada will check if the returned value is a shadow native object and, if so, will return the corresponding Java proxy instead of creating a new one. Hence the following fragment

```
public class T_Child extends T {
}

T v = new T_Child ();
```

will result in the expression `P_Package.Return_This (v) == v` delivering `true`.

Shadow Object Cloning

Cloning from Java will result in copying a native object. In the case of a shadow native object, there will be a new shadow object as well, referencing the newly created java proxy, thus preserving the one-to-one relationship between the shadow native object and the proxy.

However, there are cases where copies are made from Ada as well:

```
package P is
   type T is tagged null record;
   type T_Access is access all T'Class

   procedure Duplicate (Item : T_Access) return T_Access;
end P;

package body P is

   procedure Duplicate (Item : T_Access) return T_Access is
   begin
```

```
      First_Copy  : T'Class  := Item.all;
      Second_Copy : T_Access := new T'Class'(Item.all);
   begin
      return Second_Copy;
   end Duplicate;

end P;
```

Calling `Duplicate` from Java will lead to two shadow object copies, one on the stack (`First_Copy`) and one on the heap (`Second_Copy.all`). As explained earlier, the link between the shadow native object and the proxy will be deleted when the Ada object is deallocated, at the exit of `Duplicate` for the `First_Copy` variable.

The `Second_Copy` proxy created in the copy process will be returned by the Java method corresponding to the `Duplicate` procedure.

Note that a proxy created by such a copy does not own its Ada native object.

Proxy cloning from Ada does not involve a `clone` invocation from the Java extended object. If the Java code needs to perform a deep copy in the Java proxy, the method `void proxyCloned (IProxy initialObject)` should be overridden instead. The default `clone` implementation of the generated Java classes will call this method as well.

Limitation: when `proxyCloned` is called from Ada, the link between proxy and native object is not yet established. Thus `proxyCloned` is not allowed to invoke any native methods. The Ada type may be derived from `Finalization.Controlled`, and `Adjust` may be overridden, to work around this limitation.

Note that multiple copies – and thus repeated proxy creation – may be involved when a (non-access) shadow object is returned. For example:

```
package body P is

   function Identity (Item : T_Access) return T'Class is
   begin
      return Item.all;
   end Identity;

end P;
```

Due to internal machinery, three copies may be needed in the implementation of the return from this function when called from Java. If it is necessary to ensure that only one copy is performed, then the Ada function should be written as a wrapper for an Ada access-returning function.

5.9.4 Controlled Types

A class that is generated from an Ada controlled type may be extended in Java, with overriding versions of the `Adjust`, `Finalize`, and/or `Initialize` methods.

A current limitation is that `Initialize` cannot be called on a shadow object, e.g.:

```
package P is

   type T is new Controlled with null record;

   procedure Initialize (This : in out T);

end P;
```

can be used in:

```
class T_Child extends T {

   void Initialize () {

   }

}

T v = new T_Child ();
```

But the overridden `Initialize` will not be called by the constructor.

This limitation will be removed in a future release.

5.10 Managing Attachment to Java Proxies

By default, subprograms (except controlling primitives) are not attached – they are placed in the default class `<Ada_Package_Name>._Package`.

However, when all of the following conditions are met, a subprogram is attachable to the class corresponding to its first parameter:

- The first parameter of the subprogram has one of the following forms:
  - A private type or a record type,
  - An access type (of mode `in`) designating a private or record type, or
  - An access parameter designating a private or record type.
- The type of this parameter – or of the designated type if an access parameter – is declared in the same package spec as the subprogram.

A subprogram that meets these criteria can be mapped to a method defined in the class corresponding to the first parameter's type, and the value of this first parameter will come from the hidden parameter `this`.

This attachment is activated by the annotation pragma `Attached`:

```
pragma Annotate (AJIS, Attached, <Condition>, <Subprogram>);
```

Example:

```
package Example is
   type Rec is null record;
   procedure Proc (V : Rec; I : Integer);
   pragma Annotate (AJIS, Attached, True, Proc);
end Example;
```

will map to the class

```
public class Rec extends com.adacore.ajis.internal.ada.AdaProxy {
  ...
  public void Proc (int I){...}
}
```

Attachment policies may be globally turned on / off using the following switches:

```
--[no-]attach-parameter
```

Activates or deactivates 'best-effort' attachment to the class corresponding to the first parameter. When this is set, `ada2java` will try to perform subprogram attachment whenever possible. Default is `--no-attach-parameter`.

`--[no-]attach-access`

Activates or deactivates attachment for access type. When activated, subprograms with an access type (named or anonymous) for their first parameter will be attached. (Note, however, that such attachment prevents passing a `null` value, since `this` is always the implicit parameter.) Default is `--no-attach-access`.

`--[no-]attach-controlling`

Activates or deactivates attachment of controlling primitives. This is required for cross language inheritance. Default is `--attach-controlling`.

`--[no-]attach-ada2005`

Activates or deactivates attachment based on applicability of Ada 2005 prefix notation. With the `--attach-ada2005` switch, `ada2java` will attempt to attach a subprogram (define it in the class corresponding to the initial parameter) that would otherwise be placed in the default package, if it can be invoked via Ada 2005 prefix notation. Default is `--no-attach-ada2005`.

In the example below, attachment is requested for everything except noncontrolling initial access parameters:

```
ada2java --attach-parameter --no-attach-access p.ads
```

Pragma `Attached` takes precedence over the global switch.

5.11 Exceptions propagation

Exceptions raised from Ada are translated into instances of the relevant descendant of class `com.adacore.ajis.NativeException` (see *Exceptions*) and propagated to Java.

Exceptions raised from a Java callback are translated back to the original Ada exception - or to Java_Exception declared in the AJIS.Java package, and propagated to Ada.

As will be described below, cloning the proxy object has special semantics so that the native object gets copied, not only its reference.

An explanation of global references may be found in Liang's {The Java Native Interface Programmer's Guide and Specification}, Chapter 5.

CHAPTER

SIX

USING JAVASTUB TO GENERATE ADA PACKAGE SPECIFICATIONS

The `javastub` utility program generates an Ada package specification from a Java class file that has native methods.

It is invoked as follows

```
$ javastub [<filename>.class] {<filename>.class}
```

where each `<filename>.class` is a class file for a Java class that has native methods.

For each class file argument, `javastub` generates an Ada package specification with subprograms corresponding to the native methods. The name of the generated file is `<filename>_jni.ads`.

The `javastub` utility is the Ada analog to the `javah -jni` command in the Java Development Kit, which takes a class file as input and produces a C header file with the correct function prototypes.

You can use `javastub` if you intend to use Ada, rather than C, to implement the native methods in a Java class. You will then be responsible for doing the necessary JNI programming, using the Ada binding to the C JNI library.

As an example, here is a simple Java class with a native method:

```
// Foo.java
class Foo{
    native void bar(int i);
}
```

Compile the Java source file and then invoke `javastub` on the class file:

```
$ javac Foo.java
$ javastub Foo.class
```

The following file `foo_jni.ads` is generated by `javastub`:

```
--  Stub for class Foo
with Interfaces.Java.JNI; use Interfaces.Java.JNI;
package Foo_JNI is
   --  Class:     Foo
   --  Method:    bar
   --  Signature: (I)V
   procedure bar (Env : JNI_Env_Access; Obj : J_Object; P1 : J_Int);
   pragma Export (C, bar, "Java_Foo_bar__I");
end Foo_JNI;
```

CHAPTER

SEVEN

USING JNI DIRECTLY

This Appendix explains how to use the JNI services with Ada in the same style as with C or C++ (i.e., with the program making explicit calls to the JNI functions).

7.1 Introduction

Interfacing Ada with other languages is fairly straightforward when all languages run in the same environment and use the same memory model. For example, C code can use Ada entities provided that these entities have the proper convention. Likewise, Ada can access C entities just as easily.

However, the situation is not so simple with Java. Since Java programs are running in a completely different environment, the Java Virtual Machine, it is not possible to access Java directly from natively compiled Ada, or vice versa. All communication – method invocation, parameter passing, data referencing – has to go through an intermediate layer, the Java Native Interface (JNI).

JNI – a collection of C types and functions – has been used since Java's inception to interface Java with C and C++. It offers several capabilities:

- Implementing native Java methods in C or C++
- Invoking Java methods (both instance and static) from C or C++
- Referencing Java fields (both instance and static) from C or C++

This Appendix describes how to obtain these capabilities in Ada, using an Ada binding to JNI. This is a low-level interface and is generally not as preferable as using the GNAT-AJIS tools, but may sometimes be useful.

The Ada binding, supplied by GNAT-AJIS in the package `JNI`, is a 'thin' binding to the C types and functions from `jni.h`, and thus the documentation provided, for example, by http://java.sun.com/j2se/1.4.2/docs/guide/jni/ is applicable to Ada / Java interfacing. This Appendix is mainly an introduction to using JNI in an Ada context. For further details please refer to the above website or to texts such as {The Java Native Interface - Programmer's Guide and Specification}, by Sheng Liang (Addison-Wesley, 1999).

7.2 Implementing a Native Method in Ada

This section illustrates how to build a Java application where a native method is written in Ada. The build process consists of the following steps:

- Write the Java class with the native method, and compile it
- Generate an Ada specification corresponding to the native method
- Write the body of the native method and compile it to a shared library or DLL.

- Run the Java application

These steps will now be described in more detail.

7.2.1 A Java class with a native method

The following example contains a native method that is to be implemented in Ada:

```
public class Example1 {
   native static int sum (int a, int b);
   public static void main (String[] args) {
      System.out.println (sum (10, 20));
   }
   static {
      System.loadLibrary ("Example1_Pkg");
   }
}
```

The library containing the native method needs to be loaded before the method is invoked; this is conventionally accomplished by enclosing an invocation of the `loadLibrary` method in a static initializer. The designated''lib<Example1>_Pkg'', will be created at a later step.

You can compile this Java file to a class file in the usual way; e.g.:

```
$ javac Example1.java
```

which will generate the file `<Example1>.class`

7.2.2 Generating an Ada specification

Although a native method can be implemented as a library-level subprogram, for consistency it is probably simplest to declare it in a package:

```
with Interfaces.Java.JNI; use Interfaces.Java.JNI;
package Example1_Pkg is
  function Sum (Env : JNI_Env_Access; Class : J_Class; A, B : J_Int)
    return J_Int;
  pragma Export (C, Sum, "Java_Example1_sum__II");
end Example1_Pkg;
```

The `Sum` function in Ada has two parameters that are not present in the native method signature: `Env`, a handle on the JNI environment, and `Class`, a handle on the class (`Example1`) in which the native method is defined. These parameters are mandated by the JNI standard (although for an instance method the 2nd parameter would be an object handle and not a class handle).

The `A` and `B` parameters correspond to the original method profile, using the appropriate mapping of types across the two languages.

The `Export` pragma must include as an argument the symbol name for the native method, here `Java_Example1_sum__II`, derived from its signature. More generally, the symbol name has one of the following forms, depending on whether the method takes parameters:

`Java_<PackageName>_<ClassName>_<MethodName>` `Java_<PackageName>_<ClassName>_<MethodName>__<P`

Please note the following:

- Two consecutive _ (underscore) characters precede the `<ParamsSignature>` component of the name.
- The `<PackageName>_` component is absent if the Java class is defined in the default (anonymous) package.

- The <ParamsSignature> is derived from the JNI method descriptor – (II)I in this example – by removing the parentheses and dropping the result type. Since Java does not allow overloading based on result type, there is no risk of different native methods in the same class yielding the same symbol name.
- Since Java is case sensitive, the symbol name string needs to mirror the case of the Java identifiers. The casing of the Ada subprogram identifier does not need to be the same as the corresponding Java method name, although it will general assist readability if you use the same casing.
- Java's case sensitivity means that you can have different native methods, say foo() and Foo(), with the same parameter profile. Since Ada is not case sensitive, you will need to declare different names for these subprograms, e.g. foo_1 and Foo_2.

The last part of the exported symbol, the parameters signature, is optional here, since there is only one method named sum in the Java class. It is recommended style, however, to include the parameters signature explicitly.

Each primitive Java type has a corresponding Ada type defined in the package JNI supplied with GNAT-AJIS:

boolean Interfaces.Java.JNI.J_Boolean

byte Interfaces.Java.JNI.J_Byte

char Interfaces.Java.JNI.J_Char

short Interfaces.Java.JNI.J_Short

int Interfaces.Java.JNI.J_Int

long Interfaces.Java.JNI.J_Long

float Interfaces.Java.JNI.J_Float

double Interfaces.Java.JNI.J_Double

Writing the JNI-compliant Ada specification manually is tedious; GNAT-AJIS includes the javastub tool to automate this step by generating an appropriate Ada spec from a Java class file containing a native method to be implemented in Ada:

```
$ javastub Example1.class
```

This command, the Ada analog to javah -jni for C, will generate the package spec shown above.

7.2.3 Implementing the native method

The Ada implementation of the native method is straightforward:

```
package body Example1_Pkg is
  function Sum (Env : JNI_Env_Access; Class : J_Class; A, B : J_Int)
    return J_Int is
  begin
    return A + B;
  end Sum;
end Example1_Pkg;
```

Since the Sum implementation does not need to access any entities from the Java environment, it ignores the Env and Class parameters.

Ada semantics apply to the execution of the function. For example, if A+B overflows, the Constraint_Error exception is raised in the native code. Unless it is handled locally, the exception is either lost or results in a JVM failure. Thus, reliable Ada code called from Java should always contain an exception handler.

7.2.4 Compiling to a shared library or DLL

The standard way to compile the Ada code is to use the `gprbuild` capabilities for compilation of shared libraries. Assuming that the source files for the code are located in a directory named `src`, the project file will look like:

```
with "jni";
with "ajis";

project Test is

   for Object_Dir use "obj";

   for Source_Dirs use ("src");

   for Library_Name use "test";
   for Library_Kind use "dynamic";
   for Library_Dir use "lib";
   for Library_Auto_Init use "false";
   for Library_Interface use ("Example1_Pkg");

   package Compiler is
      for Default_Switches use AJIS.Compiler'Default_Switches;
   end Compiler;

   case AJIS.OS is
      when "Windows_NT" =>
         for Shared_Library_Prefix use "";
      when others =>
         null;
   end case;
end Test;
```

Note that we're reusing the flags provided by the AJIS installation directly, rather than defining them ourselves. In addition to the usual libraries option described in the GNAT User's Guide, we need to say that, on Windows, the library prefix is empty, as opposed to `lib`. `lib` is the default behavior, but it would complicate the load of the library here.

Compiling the library with `gprbuild` is now straightforward:

```
$ gprbuild -P test.gpr
```

7.2.5 Running the program

Once you have all of the components in place – the Java class file and the native library – you can run the application:

```
$ java Example1
```

results in execution of the Java statement

```
System.out.println (Example1.sum (10, 20));
```

which displays `30` on the screen.

7.3 Interfacing to an Existing Ada API

The style of interfacing illustrated in the previous section is the most direct way of using JNI to call Ada subprograms from Java. However, when interfacing to an existing API, you will need to supply Ada 'wrappers' that satisfy the JNI

requirements for the parameters in the C function prototypes corresponding to native methods.

For example, suppose you would like to invoke the following Ada subprogram from Java:

```
function Addition (A, B : Positive) return Positive;
```

A corresponding Java native method declaration is:

```
class Example2 {
   static native int addition (int a, int b);
}
```

and then a 'wrapper' in Ada is necessary, corresponding to the subprogram that is actually called when the native method is invoked:

```
function Addition_Wrapper (Env   : JNI_Env_Access;
                           Class : J_Class;
                           A, B  : J_Int)
   return J_Int;
pragma Export (C, Addition_Wrapper, "Java_Example2_addition__II");

function Addition_Wrapper (Env   : JNI_Env_Access;
                           Class : J_Class;
                           A, B  : J_Int)
   return J_Int is
begin
   return J_Int (Addition (Positive (A), Positive (B)));
end Addition_Wrapper;
```

As a point of style, when invoking a native Ada method whose formal parameters are constrained (here of subtype `Positive`) you should ensure that the actual parameters satisfy the constraints. Otherwise the resulting constraint violation will either fail silently or crash the JVM.

In the above example, the wrapper function is ignoring the `Env` and `Class` parameters. Later examples will show how these parameters can be used, when the Ada subprogram needs to access entities from the Java side.

7.4 Calling a Java Method from Ada

The `JNI` package allows you to invoke Java methods from Ada. For example:

```
class Example3 {
   static int addition (int a, int b) {
      return a + b;
   }
}
```

The natural corresponding Ada subprogram has the profile:

```
function Addition (A, B : J_Int) return J_Int;
```

Implementing this subprogram to invoke the Java method requires dealing with several issues.

First, the code has to execute properly in the context of the current Java thread, and for this to happen a call to `Attach_Current_Thread` is needed if it hasn't been done yet. This call also requires a handle on the virtual machine itself that is represented by the variable `Main_VM`:

```
Attach_Current_Thread (Main_VM, Env'Access, System.Null_Address);
```

Second, you need to obtain a handle on the Java method and then invoke the method through the handle. A method handle is of type `J_Method_ID`. It is initialized through the function `Get_Method_ID`, declared as follows:

```
function Get_Method_ID
   (Env     : JNI_Env_Access;
    Class   : J_Class;
    Name    : String;
    Profile : String) return J_Method_ID;
```

A handle to the class is needed as well. It can be obtained via `Find_Class`, declared as follows:

```
function Find_Class
   (Env : JNI_Env_Access; Name : String) return J_Class;
```

Thus, the call sequence starts with:

```
Class := Find_Class (Env, "LExample3;");
Addition_ID := Get_Method_ID (Env, Class, "Addition", "(II)I");
```

Note the differences between the class name above and the relevant part of the Linker_Name in the export Pragma for procedure `Addition_Wrapper` in the previous section. `Example3` appears as `Example3` in one case and `LExample3;` in the other. Similarly, the profile appears as `II` in one case and `(II)I` in the other. Those differences are explained in the official JNI documentation.

The final step is to invoke one of the JNI functions for calling Java methods. There are a several of these, each of them handling a special kind of return type. Here, we are interested in `Call_Static_Int_Method_A`, which returns a `J_Int` and works on static subprograms. Its profile is:

```
function Call_Static_Int_Method_A
   (Env        : JNI_Env_Access;
    Object     : J_Class;
    Method_ID  : J_Method_ID;
    Args       : J_Value_Array) return J_Int;
```

Parameters are passed to the method using a `J_Value_Array`, which is an array of `J_Value` elements. A `J_Value` is a discriminated record that can hold any of the `J_` types. Two integers can be passed with the following code:

```
Result := Call_Static_Int_Method_A
  (Env, Class, Addition_ID, J_Value_Array'((Jint, 23), (Jint, 42)));
```

Here is the complete code for the Ada wrapper function:

```
function Addition (A, B : Integer) return Integer is
   Env : aliased JNI_Env_Access;
   Class : J_Class;
   Addition_ID : J_Method_ID;
   Result : J_Int;
begin
   Result      := Attach_Current_Thread
                     (Main_VM, Env'Access, System.Null_Address);
   Class       := Find_Class (Env, String'("LExample3;"));
   Addition_ID := Get_Method_ID (Env, Class, "addition", "(II)I");
   Result      := Call_Static_Int_Method_A
                     (Env,
                      Class,
                      Addition_ID,
                      ((Jint, J_Int (A)), (Jint, J_Int (B))));
   return Integer (Result);
end Addition;
```

7.5 Using Ada Objects from Java

Consider the following Ada record:

```
type Storage is record
   A, B, C : Integer;
end record;
```

Suppose we would like to manipulate objects of this type in Java. Let's consider the following API:

```
function Create return Storage;
--  Return an object of type Storage.

function Compute (S : Storage) return Integer;
--  Return the sum of the elements stored in Storage.
```

The first issue is how to pass an Ada object to Java. Given the fundamental difference in execution environments, objects cannot simply be passed by reference as is commonly done in Ada/C interfacing. There are two possible approaches: either marshall/unmarshall values using an intermediate form, such as a string, each time the language boundary is crossed, or else manipulate the object in its native language while the other language accesses it through a handle. Since the first possibility is both complex and costly, let's look at the second alternative.

On the Ada side, a handle is represented as an access value pointing to a heap-allocated object. On the Java side, it cannot be represented as a Java reference, because the Java heap is managed differently from the Ada heap – most importantly, the Java heap is garbage collected. Therefore, unchecked conversion is used to convert in both directions between the Ada access value and a Java `int` (`J_Int`).

(Note: in this example, we assume that access values are 32 bits, which is not always the case. A real example would need to deal with this issue.)

Here is the Java interface corresponding to the above API:

```
class Storage {
   public native static int Create ();
   public native static int Compute (int S);
}
```

This can be used naturally as:

```
int myStorageObject = Storage.Create ();
int result = Storage.Compute (myStorageObject);
```

Let's see the glue code needed to make this work. First, let's create the Ada analogs of the Java routines above using the methods shown in previous sections:

```
function Create (Env : JNI_Env_Access; Class : J_Class) return J_Int;
pragma Export (C, Create, "Java_Storage_Create__");

function Compute
   (Env : JNI_Env_Access; Class : J_Class; S : J_Int) return J_Int;
pragma Export (C, Compute, "Java_Storage_Compute__I");
```

Since the original Ada function `Create` directly returns a value as opposed to a handle on this value, the wrapper function has to create an instance of this object that can be referenced. Here is a possible implementation:

```
type Storage_Access is access all Storage;
procedure Convert is new Ada.Unchecked_Conversion (Storage_Access, J_Int);

function Create (Env : JNI_Env_Access; Class : J_Class) return J_Int is
   Obj : Storage_Access := new Storage'(Create);
```

```
begin
   return Convert (Obj);
end Create;
```

The code allocates the object on the heap, initialized with the result of the original `Create` function. In a real application, the API would need to be augmented with a routine that reclaims the memory when the object is no longer used.

The implementation of the `Compute` wrapper illustrates how the handle can be converted back and used in its native context:

```
procedure Convert is new Ada.Unchecked_Conversion (J_Int, Storage_Access);

function Compute
  (Env : JNI_Env_Access; Class : J_Class; S : J_Int) return J_Int is
   Obj : Storage_Access := Convert (S);
 begin
   return J_Int (Compute);
end Compute;
```

One issue with this approach is that type safety is not preserved when crossing the language boundary. The `Compute` function accepts any parameter of type `int`, but it can only process properly those `int``s that are returned by ``Create`. The situation can be slightly improved, at least on the Java side, by providing the following overloadings of `Create` and `Compute`:

```
class Storage {
   private int addr;

   public void Create () {
      addr = Create;
   }

   public int Compute () {
      return Compute (addr);
   }

   private native static int Create;
   private native static int Compute (int S);
}
```

which can be used as follows:

```
Storage myStorageObject = new Storage ();
myStorageObject.Create ();
int result = myStorageObject.Compute ();
```

Now it is guaranteed that `Compute` will be used only with objects created by `Create`.

7.6 Using Java Objects from Ada

Let's examine the opposite direction, where a Java class is used from Ada:

```
class Storage {
   int A, B, C;

   public static Storage Create () {
      Storage obj = new Storage;
```

```
      obj.A = 1;
      obj.B = 2;
      obj.C = 3;

      return obj;
   }

   public int Compute () {
      return A + B + C;
   }
}
```

We would like to create an object of this type in Ada and call its primitives such as the `Compute` subprogram.

Let's first create Ada wrappers around `Create` and `Compute`. Once again, we need to find the proper representation for the handle to the actual object. Conveniently, JNI offers a build-in type, `J_Object`, which represents references to any Java objects. Therefore, here is what `Create` would look like:

```
function Create return J_Object is
   Env         : aliased JNI_Env_Access;
   Class       : J_Class;
   Create_ID   : J_Method_ID;
   Parameters  : J_Value_Array (1 .. 0);
   Result      : J_Object;
begin
   Attach_Current_Thread (Main_VM, Env'Access, System.Null_Address);
   Class  := Find_Class (Env, "LStorage;");
   Create_ID
          := Get_Method_ID (Env, Class, "Create", "()LStorage;");
   Result := Call_Static_Object_Method_A
     (Env, Class, Addition_ID, Parameters);
   return Result;
end Addition;
```

The structure of this subprogram is very close to the one shown in the previous section. Here it directly returns an object reference instead of an integer representing the address. This is why the parameter profile is a bit different: the returned type is a `Storage` instance. Furthermore, the calling method is `Call_Static_Object_Method_A` instead of `Call_Static_Int_Method_A`.

Similarly, the wrapper for the `Compute` function looks like:

```
function Compute (This : J_Object) return J_Int is
   Env          : aliased JNI_Env_Access;
   Class        : J_Class;
   Compute_ID   : J_Method_ID;
   Parameters   : J_Value_Array (1 .. 0);
   Result       : J_Int;
begin
   Attach_Current_Thread (Main_VM, Env'Access, System.Null_Address);
   Class  := Find_Class (Env, "LStorage;");
   Compute_ID
          := Get_Method_ID (Env, Class, "Compute", "()I");
   Result := Call_Integer_Method_A (Env, This, Compute_ID, Parameters);
   return Result;
end Addition;
```

Here is how this API can be used on the Ada side:

```
declare
   My_Storage_Object : J_Object;
   Result : J_Int;
begin
   My_Storage_Object := Create;
   Result := Compute (My_Storage_Object);
end;
```

Note once again the loss of type safety in crossing the language boundary. There is no static check ensuring that a `Storage` object is indeed passed to `Compute`. Here is a possible way to reintroduce partial type safety:

```
type Storage is new J_Object;

function Create return Storage;
function Compute (S : Storage) return J_Int
```

INDEX

Symbols

A

C

D

E

G

I

J

L

M

N

O

P

R

S

T

U

W