

GNAT User's Guide

Supplement for the JVM Platform

The GNU Ada Environment
GNAT Version 2011

© Copyright 1998-2008, AdaCore

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Java is a trademark of Sun Microsystems, Inc.

About This Guide

This guide describes the features and the use of GNAT for the JVM, the Ada development environment for the Java platform. This guide also explains how to use the Java API from Ada and how to interface Ada and the Java programming language.

Before reading this manual you should be familiar with the *GNAT User Guide* as a thorough understanding of the concepts and notions explained there is needed to use GNAT effectively.

What This Guide Contains

This guide contains the following chapters:

- [Chapter 1 \[Getting Started with GNAT for the JVM\], page 3](#), gives an overview of GNAT and its tools and explains how to compile and run your first Ada program for the Java platform.
- [Chapter 2 \[Ada & Java Interoperability\], page 7](#) explains how the Java API and the services of any JVM class can be used from Ada. This section also explains how Ada services can be exported to Java programmers.
- [Chapter 3 \[Viewing Class Files with jvmlist\], page 9](#), describes `jvmlist`, a utility to disassemble a JVM `.class` file to view its contents: bytecode, constant pool (i.e. symbol table), debugging info, etc. This utility can also embed the original source code into the assembly listing.
- [Chapter 4 \[Stripping Debug Info with jvmstrip\], page 11](#), describes `jvmstrip` a utility that strips a `.class` file, removing all of its debugging info to reduce the file size.
- [Chapter 5 \[Building Archives with jarmake\], page 13](#), describes the `jarmake` tool to make a single `.jar` file for an application built with GNAT. This is useful when you want to ship a self-contained application built with GNAT to someone who does not have GNAT installed. This tool is very useful when creating "gnapplets" (GNAT applets, see [Chapter 9 \[Creating Gnapplets with GNAT\], page 49](#)).
- [Chapter 6 \[Using the Java API with jvm2ada\], page 15](#), describes the `jvm2ada` interfacing tool that takes any JVM `.class`, `.zip` or `.jar` files as input and generates Ada package specs as output. The resulting Ada specs can be used by Ada programs to interface to Java.
- [Chapter 7 \[Java-Specific Pragmas\], page 19](#) explains some special pragmas that have been introduced to support certain aspects of interfacing between Ada and Java.
- [Chapter 8 \[Mapping Java into Ada\], page 33](#) gives details on how the Java API and, in general, any Java class spec is mapped into an Ada package specification by the `jvm2ada` tool.
- [Chapter 9 \[Creating Gnapplets with GNAT\], page 49](#), explains how you can create "gnapplets" (GNAT applets).
- [Chapter 10 \[Debugging Ada Programs\], page 53](#), describes how to run and debug Ada programs.
- [Chapter 11 \[Limitations\], page 57](#), describes the language constructs, libraries and switches that are not supported by GNAT for the JVM.

What You Should Know Before Reading This Guide

Before reading this document readers should be familiar with the *GNAT User Guide* and have a conceptual understanding of the Java technology.

Related Information

For further information about GNAT, Ada, and the Java technology, we recommend consulting the following documents:

- *GNAT User Guide*, contains introductory and reference material for the GNAT development environment.
- *Ada 2005 Language Reference Manual*, contains all reference material for the Ada programming language.
- *The Java Tutorial: Object-Oriented Programming for the Internet*, 2nd edition, by Mary Campione and Kathy Walrath, published by Addison Wesley.
- *The Java Virtual Machine Specification*, by Tim Lindholm and Frank Yellin, published by Addison Wesley.

Conventions

Following are examples of the typographical and graphic conventions used in this guide:

- `source code`, and utility program names.
- 'Option flags'.
- 'File Names'.
- *Variables*.
- *Emphasis*.
- [optional information or parameters]
- Examples are described by text
and then shown this way.

Commands that are entered by the user are preceded in this manual by the "\$ " characters (dollar sign followed by space). If your system uses this sequence as a prompt, then the commands will appear exactly as you see them in the manual. If your system uses some other prompt, then the command will appear with the \$ replaced by whatever prompt character you are using.

1 Getting Started with GNAT for the JVM

1.1 Overview

The Java(TM) technology, introduced by Sun Microsystems, is a paradigm whose goal is to add platform-independent programming flexibility to Internet, Intranet and Extranet applications, embedded devices such as Internet appliances, consumer electronics, smart cards, etc.

The Java technology comprises, a simple object-oriented programming language (Java), a comprehensive set of libraries (Java API), and a virtual machine (JVM) offering the same object code interface on all platforms (bytecode).

Although the Java environment comes with a default programming language, this language is not a fundamental component of the technology. Any programming language that can be mapped onto the JVM can be used to develop Java applications.

The GNAT system offers an Ada 2005 programming environment for the Java platform. In addition to a bytecode compiler, binder and linker, GNAT comprises a Java-to-Ada binding generator which produces the Ada 2005 specs of the services contained in any Java `.class` file or API. In addition to all of the conventional GNAT tools, a bytecode disassembler and a `.class` file stripper are also provided with GNAT for the JVM.

Furthermore, because the `.class` files generated by the GNAT compiler are fully compliant with Sun's standard, the user can employ any JVM to run Ada applications, any JVM debugger to debug Ada code, and can use any of the Java tools that operate on `.class` files (e.g. `jar`, `javap`, etc.).

As a side note, the GNAT system is implemented in Ada and its sources are available under the GPL.

1.2 GNAT Tools

Most tools are regular GNAT tools that have been slightly adapted for use with GNAT for the JVM. They are used in the same fashion as their corresponding GNAT equivalent. These tools are:

- `jvm-gnatcompile`: the compiler, compiles an Ada unit into one or more JVM `.class` files.
- `jvm-gnatbind`: the binder, generates an Ada source file containing the elaboration code for the Ada application to run.
- `jvm-gnatlink`: the linker, compiles the source file generated by `jvm-gnatbind`. `jvm-gnatlink` provides no linking capabilities since the linker is directly embedded into the JVM. To gather the `.class` files of an application into a single file, one can use the `zip` or `jar` commands provided with your Java Development Kit.
- `jvm-gnatmake`: the automatic make program, automatically determines the set of sources needed by an Ada compilation unit, and executes the necessary compilations, binding, and link.
- `jvm-gnatls`: the library browser, displays information about compiled units, including dependences on the corresponding sources files, and consistency of compilations.

- `jvm-gnatfind`: the find utility, provides an easy way to locate the declaration and references for an Ada entity.
- `jvm-gnatxref`: the cross-referencer, allows you to generate a full report of all cross-references in a given set of Ada units.

The GNAT tools which have been specifically developed for the JVM are:

- `jvmlist`: The GNAT disassembler, (see [Chapter 3 \[Viewing Class Files with jvmlist\]](#), [page 9](#)) disassembles a JVM `.class` file to view its contents: bytecode, constant pool (i.e., symbol table), debugging info, etc. This utility will also embed the original source code into the assembly listing. This utility is independent of the original programming language and works equally well on programs containing a mixture of Ada and Java code.
- `jvmstrip`: The GNAT strip utility, (see [Chapter 4 \[Stripping Debug Info with jvm-strip\]](#), [page 11](#)) is a utility that strips a `.class` file, removing all of its debugging info to reduce file size. This tool is also programming-language independent.
- `jarmake`: The GNAT archiver tool, (see [Chapter 5 \[Building Archives with jarmake\]](#), [page 13](#)) takes `.class` files as input and recursively collects into an uncompressed zip archive all the `.class` files needed by the `.class` files specified on the command line. This tool can be used to prepare self-standing applications or applets that you can ship. This tool is programming-language independent.
- `jvm2ada`: The GNAT interfacing tool, (see [Chapter 6 \[Using the Java API with jvm2ada\]](#), [page 15](#)) takes `.class` files, or zip archives as input and generates Ada package specifications as output. The resulting Ada package specs can be `with`-ed by Ada programs to interface to Java services.

1.3 Java Development Kits that you can use with GNAT

Because GNAT generates class files that are fully compliant with Sun's JVM standard, you can use any Java Virtual Machine and bytecode tools that meet the Sun Java platform standard.

1.4 Compiling Your First Application with GNAT

To compile the following "Hello GNAT for the JVM" program put the following in file `'hello.adb'`:

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Hello is
begin
  Put_Line ("Hello GNAT for the JVM.");
end Hello;
```

then type:

```
$ jvm-gnatmake hello
```

This command will generate file `'hello.class'`. To run it, assuming you are using Sun's JDK (Java Development Kit), you can just type

```
$ java hello
```

It's as simple as that. To compile more complex Ada applications use `jvm-gnatmake` as usual. If you want to use the GNAT compiler, binder, and linker separately, you will need to

individually invoke the appropriate `jvm-gnatcompile`, `jvm-gnatbind` and `jvm-gnatlink` commands.

2 Ada & Java Interoperability

One aspect of Java that makes it an interesting platform is its growing set of API classes. It is therefore fundamental that the API be made available to the Ada programmer transparently. It is also important that the Ada programmer be able to write libraries or APIs for the Java platform in Ada, and that these libraries be easily usable in any Java application. GNAT guarantees full interoperability between Ada and Java.

To achieve this goal, constructs that can appear in a Java class at the specification level are mapped to Ada either by means of a corresponding Ada feature or by using an implementation-defined Ada pragma.

In addition we have taken great care that the mapping from Java to Ada is completely automatic. This means that GNAT comes with no Ada bindings for the Java API, but instead provides a tool (see [Chapter 6 \[Using the Java API with jvm2ada\]](#), page 15) which is able to produce Ada specifications from any set of JVM `.class` files.

2.1 Importing Java Services to Ada

To access the services provided by the Java API or by any set of JVM `.class` files, you should proceed as follows:

1. If your Java code is in source form, compile it using any Java compiler.
2. If you just want to import a variable or a subprogram from a Java class use `pragma Import` (see [Section 7.4 \[Pragma Import Java\]](#), page 28) in the Ada code where you want to import the Java service.
3. More generally, you can use the `jvm2ada` utility to produce the Ada specs (containing the appropriate Java-specific pragmas) for the `.class`, `.zip`, or `.jar` files containing the Java services you would like to use from Ada (see [Chapter 6 \[Using the Java API with jvm2ada\]](#), page 15). Note that `jvm2ada` preserves, in the generated Ada specs, the names of the original Java services (for a detailed explanation of the Java to Ada mapping see [Chapter 8 \[Mapping Java into Ada\]](#), page 33).
4. `with` the needed Ada specs and use their services as usual.

It's as simple as that.

2.2 Exporting Ada Services to Java

To export a set of Ada services to Java you should:

1. Use `pragma Export` and other Java interfacing pragmas inside the Ada code (see [Chapter 7 \[Java-Specific Pragmas\]](#), page 19). This gives you complete control of what is being generated and allows you to decide very precisely what the exported services look like on the Java side.
2. Compile your Ada code with the GNAT compiler.
3. Use `javap` to display the spec of the `.class` files generated by GNAT whose services you would like to use in your Java code.

3 Viewing Class Files with `jvmlist`

The `jvmlist` tool takes JVM `.class` files as input (directly or regrouped in an uncompressed `.zip` or `.jar` file) and disassembles it to view its contents: bytecode, constant pool (i.e., symbol table), debugging info, etc. This utility can also embed the original source code in the assembly listing. `jvmlist` is independent of the original programming language and works equally well on programs containing a mixture of Ada and Java code.

3.1 Running `jvmlist`

The form of the `jvmlist` command is

```
jvmlist [switches] file [file ... file]
```

where *file* can be one of the following:

- the name of a `.class` file (possibly without the `.class` suffix);
- the name of an uncompressed zip archive (in this case the `jvmlist` command applies to all `.class` files within the archive);
- the name of a `.class` file within an uncompressed zip archive (e.g. `rt.jar/java/lang/Object.class`).

File names can be prefixed with directory information.

The output of `jvmlist` contains a listing of all the fields and methods declared inside the `.class` file, in addition to various other class information such as the class it extends, the interfaces it implements, etc.

If you select switch `-c`, then `jvmlist` will also print the bytecode for each method. The bytecode mnemonics used by `jvmlist` are the same as those documented in Sun's JVM book *The Java Virtual Machine Specification* by Lindholm and Yellin. If you select switch `-c`, `jvmlist` will also embed the original source code in the bytecode. For now, `jvmlist` only looks for source files in the current directory.

3.2 Switches for `jvmlist`

The following switches are available with the `jvmlist` utility:

- | | |
|-----------------|---|
| <code>-c</code> | Display bytecode. By default <code>jvmlist</code> does not display the bytecode of each method. This switch specifies that bytecode should be displayed. |
| <code>-g</code> | Embed source code. This switch implies <code>-c</code> and embeds the original source code within the disassembled bytecode. If the input <code>.class</code> file does not contain source file information, or if the source file cannot be located in the current directory (the one where <code>jvmlist</code> was invoked), then this switch is equivalent to <code>-c</code> . |
| <code>-l</code> | Display line number tables. |
| <code>-p</code> | Display the constant pool. |
| <code>-t</code> | Display local variable tables. |
| <code>-v</code> | Verbose. Outputs the name of each class file for which an Ada spec is generated. |

- V Very verbose. Implies -v. Each item that is encountered in a zip or jar file is listed on the screen. Class files are preceded with a ->, other items are listed with a message saying that the item is skipped. Only class files whose name is listed twice (one preceded with a -> and the following one without the ->) have a corresponding Ada spec generated for them, other class files are ignored (because for instance they are not public classes).

4 Stripping Debug Info with `jvmsstrip`

The `jvmsstrip` tool takes a `.class` files as input (directly or packaged in an uncompressed `.zip` or `.jar` file) and strips off all of its debugging info to reduce its size. This utility is independent of the original programming language and works equally well on programs containing a mixture of Ada and Java code.

4.1 Running `jvmsstrip`

The form of the `jvmsstrip` command is

```
jvmsstrip [switches] file [file ... file]
```

where *file* can be one of the following:

- the name of a `.class` file
- the name of an uncompressed zip archive (in this case the `jvmsstrip` command applies to all `.class` files within the archive);

File names can be prefixed with directory information.

The output of `jvmsstrip` is a stripped `.class` file which replaces the original input file. If the input is an archive, then all of the `.class` files within the archive are stripped and the updated archive replaces the input file.

4.2 Switches for `jvmsstrip`

The following switches are available with the `jvmsstrip` utility:

`-v` Verbose.

5 Building Archives with `jarmake`

When building an Ada application with GNAT, a number of JVM `.class` files are generated. In addition to potentially being numerous, the generated classes depend on the library `jgnat.jar` which is installed with GNAT.

If you need to ship your Ada application or gnapplet to people who do not have GNAT installed this can be cumbersome as you would need to ship the `.class` files of your application along with `jgnat.jar`.

To automate such process we have provided `jarmake`, the GNAT archiver tool. `jarmake` takes `.class` files as input and recursively collects into an uncompressed zip archive all the `.class` files needed by the `.class` files specified on the command line. This tool can be used to prepare self-standing applications or gnapplets.

This utility is independent of the original programming language and works equally well on programs containing a mixture of Ada and Java code.

5.1 Running `jarmake`

The form of the `jarmake` command is

```
jarmake [switches] file [file ... file]
```

where *file* can be one of the following:

- the name of a `.class` file
- the name of an image, sound or any other file that you want to bundle with your application.

File names can be prefixed with directory information.

The output of `jarmake` is an uncompressed zip archive containing the files specified on the command line along with the `.class` files they recursively reference.

5.2 Switches for `jarmake`

The following switches are available with the `jarmake` utility:

`-Lzip-archive`

When searching for `.class` files, look in the uncompressed zip archive *zip-archive*.

- `-j` Do not skip `.class` files in the Java API. By default `jarmake` skips all the `.class` files in the Java API. By using this switch you are asking `jarmake` to include Java API classes in the output zip archive. If you set this flag you should also provide a `-L` flag giving the location of the Java API zip archive.
- `-k` Keep going even if not all of the `.class` files are found. By default, `jarmake` will stop if it cannot find all the needed `.class` files. By setting this switch `jarmake` will emit a warning message when it cannot find a `.class` file it is looking for and will continue.
- `-m` Add a `Main-Class` attribute to the manifest for the first class encountered that has a main method.

- n** Do not include the `‘.class’` files of the library `‘jgnat.jar’` in the output archive. By default these files are included in the output archive so that the archive is autonomous.
- o *zip-archive*** Name of the output uncompressed zip archive. If this switch is not specified, then the default name is `‘gnapplet.jar’`.
- q** Quiet.
- v** Verbose.

6 Using the Java API with jvm2ada

The `jvm2ada` tool takes JVM `.class` files as input (directly or regrouped in an uncompressed `.zip` or `.jar` file) and generates Ada specs as output.

6.1 Running jvm2ada

The form of the `jvm2ada` command is

```
jvm2ada [switches] file [file ... file]
```

where *file* can be any of the following:

- the name of a `.class` file (possibly without the `.class` suffix);
- the name of an uncompressed zip archive (in this case the `jvm2ada` command applies to all `.class` files within the archive);
- the name of a `.class` file within an uncompressed zip or jar archive (e.g. `rt.jar/java/lang/Object.class`).

File names can be prefixed with directory information.

The output of `jvm2ada` is an Ada source file for each `.class` file processed. The Ada source file contains a package spec giving the Ada declaration for the services exported by the corresponding `.class` file. The name of the Ada package is obtained by concatenating the name of the Java class to the name of the Java package containing the class. As an example, a Java class `someName` occurring within Java package `some.pack` yields the Ada package `some.pack.someName` and is in a file named `'some-pack-somename.ads'`.

Unless switch `-o` is used (see [Section 6.2 \[Switches for jvm2ada\]](#), page 15), the Ada files generated are placed in the directory where the `jvm2ada` command is invoked.

6.2 Switches for jvm2ada

The following switches are available with the `jvm2ada` utility:

`-Izip-archive`

When looking for a source file (to find the parameter names of a Java method), search the uncompressed zip archive *zip-archive*. See [Section 6.4 \[Parameter Names and Source Search Paths\]](#), page 17, for details. (This switch is not yet supported.)

`-Lzip-archive`

When searching for `.class` files, look in the uncompressed zip archive *zip-archive*. See [Section 6.5 \[Class File Search Paths\]](#), page 17, for details.

`-k` Keep original JVM identifiers. By default, identifiers encountered in a JVM `.class` file are mangled whenever needed to turn them into proper Ada identifiers. When this switch is set, identifiers are left as is in the generated Ada package spec. See [Section 6.6 \[Identifier Mangling\]](#), page 18, for details.

`-o dir` Output to *dir*. Put all generated Ada source files into directory *dir* rather than the current directory.

`-q` Quiet.

- s** Map Sun-specific classes into Ada specs. By default, Sun's classes are not mapped into Ada even if they are public, because they are typically not part of the API at hand (certainly they are not part of the Java API, even though the corresponding jar file contains them). Sun's classes are the classes in packages `sun`, `sunw`, and `com.sun`.
- v** Verbose.
- w** Overwrite existing file names. Normally `jvm2ada` regards it as a fatal error if there is already a file with the same name as a file it would otherwise output. This switch bypasses this check, and any such existing files will be silently overwritten.

6.3 Running `jvm2ada` on the Java API

In this section, we'll assume that the environment variable `JAVA_SDK` points to the root installation of your java SDK (e.g. `/jdk-1.5.0/`).

To be able to access the Java API you need to process it to generate an Ada package spec for each public class in the API. In order to manually do that, you will need first to create a uncompressed version of the `jce.jar` file:

```
$ jar -xf "$JAVA_SDK/jre/lib/jce.jar"
$ jar -0cf uncompressed_jce.jar javax META-INF
```

then, run `jvm2ada` on the Java library archives (`rt.jar`, `uncompressed_jce.jar`, `charsets.jar` and `jsse.jar`).

```
$ jvm2ada -jgnat "$JAVA_SDK/jre/lib/rt.jar" \
  "-Luncompressed_jce.jar" \
  "-L$JAVA_SDK/jre/lib/charsets.jar" \
  "-L$JAVA_SDK/jre/lib/jsse.jar"
```

This will create, in the current directory, an Ada package spec for each public Java class. If you would like to output the Ada specs in some other directory use `jvm2ada` switch `-o`.

It's possible perform this operation automatically, in a single command, using the script `bind_jre.sh` (or `bind_jre.cmd` on windows):

```
$ bind_jre.sh $JAVA_SDK
```

Please note that because of some `jvm2ada` limitations, you may have compilation errors with some of the generated packages. The most common one is a name clash, for example:

```
java-util-concurrent-locks-reentrantreadwritelock.ads:54:13: "WriteLock"
conflicts with declaration at
java-util-concurrent-locks-reentrantreadwritelock-writelock.ads:9
```

In order to fix this kind of conflict, you will need to edit the Ada generated code and fix manually the name clash, for example by adding a suffix `"_C"` to the declaration of `WriteLock` at line 54.

In case of a more complex problem, the generic workaround is to manually wrap the class into a simpler interface, and then use the resulting binding of this interface. This occurs on very rare cases, the large majority of the generated binding from the API being usable without any change.

6.4 Parameter Names and Source Search Paths

Note: Only point 2. below is implemented.

When generating the Ada spec for a ‘.class’ file, jvm2ada tries to preserve the original names of method parameters. If the ‘.class’ file was compiled enabling the generation of debugging tables (switch ‘-g’ in Sun’s JDK javac compiler), parameter names are stored in the .class file. If not jvm2ada proceeds as follows:

1. If the name of the original source file is present in the ‘.class’ file, jvm2ada tries to locate this source by looking at the uncompressed zip archives specified by the **-Izip-archive** switches, in the order in which these switches occur. Once found, jvm2ada uses the source file to locate parameter names. If the original source is not around you can always communicate parameter names by creating a Java source file containing the appropriate method specs. For instance to give the names of the parameters of method `someMethod` in class `someClass` in package `some_package` you could create the following source file:

```
package some_package;
// You must provide the appropriate Java package for someClass

public class someClass {
    public int someMethod (int someName, float anotherName) {}
    // The methods for which you want to name the parameters must have the
    // same signature as the methods found in the .class file. The body
    // can be empty.
}
```

2. If the appropriate source file cannot be located jvm2ada assigns arbitrary parameter names of the form `P1_type`, `P2_type`, etc. where *type* denotes the flattened type name for the corresponding parameter. The reason for appending *type* to the parameter name is to allow the Ada programmer to resolve possible overloading resolution conflicts of the following kind

```
public class Base { }
public class Deriv extends Base {
    public static void p (Base obj) { ... }
    public static void p (Deriv obj) { ... }
}
```

The overloaded procedure `p` above are translated by jvm2ada to the following Ada specs:

```
procedure p (P1_Base : access Base.Typ'Class);
procedure p (P1_Deriv : access Deriv.Typ'Class);
```

The problem that arises is shown by the following example:

```
type Deriv_Ref is access all Deriv.Typ'Class;
Obj : Deriv_Ref := ...;
procedure p (Obj);                -- Ambiguous call
procedure p (P1_Base => Obj);      -- OK
procedure p (P1_Deriv => Obj);      -- OK
```

6.5 Class File Search Paths

When processing a ‘.class’ file, jvm2ada may need to locate other ‘.class’ files. For instance, to know whether the JVM class being processed is a Java exception, jvm2ada must traverse the inheritance tree and must therefore locate the ‘.class’ files of the ancestor classes.

If `jvm2ada` does not find the `‘.class’` file it is looking for, then a warning message is emitted. The order in which `jvm2ada` searches `‘.class’` files is given below.

1. If the `‘.class’` file being processed belongs to an uncompressed zip archive, `jvm2ada` will look there first.
2. The uncompressed zip archives specified by a `-Lzip-archive` switch are searched next, in the order in which the `‘-L’` switches occur.

6.6 Identifier Mangling

`jvm2ada` retains, whenever possible, the identifiers it finds in the `‘.class’` files it processes. This is not always possible, however, because Java's set of legal identifiers is bigger than Ada's. To address these issues `jvm2ada` proceeds as follows: If switch `-k` is set, the original identifiers found in the JVM `.class` are left unchanged. You will have to change these yourself in the generated packages if these are illegal Ada identifiers. If switch `-k` is not set then:

- Every identifier which is an Ada reserved word or any of the words `“Standard”`, `“Ref”`, `“Typ”`, `“Arr”`, `“Arr_2”`, `“Arr_3”`, is suffixed with `_K`. For instance, `Abort` is mapped to `Abort_K`.
- A single underscore is replaced by `“U”`.
- A leading underscore is replaced by `“U_”`.
- A trailing underscore is replaced by `“_U”`.
- A letter `“U”` is placed between every two consecutive underscores.
- If two or more identifiers generated in an Ada spec lead to an Ada name conflict, then `jvm2ada` will add a trailing `_K` at the end of the second occurrence, a trailing `_K2` at the end of the third occurrence, a trailing `_K3` at the end of the fourth occurrence, etc. The cases currently caught are: identical variable names, identical variable and subprogram names, identical variable and child package name. More complex cases are not yet handled. In particular, we do not yet detect the case where we have two identical field names in a record `B` and a record `D` derived from `B`. In these cases you will have to revise the generated Ada spec to allow it to compile.

7 Java-Specific Pragmas

The typical way to import services from Java classes is to use the `jvm2ada` tool to automatically generate the specification of the corresponding `.class` file. This specification contains the appropriate Java-specific pragmas.

In some cases you may wish to import just one routine to your Ada code or you may prefer to group certain services from multiple `.class` files into a single Ada spec (for instance if you are trying to provide a simplified view of the Java API).

In such cases it is useful to understand how the various Java-specific pragmas work. Another situation where you may have to use these pragmas explicitly is when exporting Ada services to Java.

This chapter introduces the features and pragmas that are needed for full support of interfacing between Java and Ada.

7.1 Creating Java Interfaces: Pragma `Java_Interface`

Java offers a special kind of class called an interface. Interfaces provide a limited but useful form of multiple inheritance. A Java interface is basically an abstract class with no fields and whose methods are all abstract. Instead of inheriting from an interface, a Java class `C` is said to implement the interface, which means that `C` must provide an implementation for all of the abstract methods declared in the interface.

The key point to note about interfaces is that a class `C` can implement several interfaces at the same time, and this mechanism is orthogonal to the fact that `C` may be extending some other class.

To make a Java interface available to an Ada program we have provided the pragma `Java_Interface`. Its syntax is:

```
pragma Java_Interface (type-name);
```

where *type-name* is the name of a type `T` declared earlier, immediately within the same declarative part where the pragma occurs, and where the type has the following characteristics that reflect the restrictions on Java interfaces:

1. `T` must be an abstract tagged type with a null record extension.
2. `T` must be derived from `java.lang.Object.Typ` (see [Section 8.3 \[Java References and java.lang.Object\]](#), page 33).
3. `T` must have an access discriminant named `Self` with `java.lang.Object.Typ'Class` as its designated type.
4. All of `T`'s primitive operations must be abstract.
5. `T` must have `Java Convention`.

Here is an example of using pragma `Java_Interface`:

```
with java.lang.Object;
package Foo is
  type Typ (Self : access java.lang.Object.Typ'Class)
    is abstract new java.lang.Object.Typ with null record;
  pragma Java_Interface (Typ);

  type Ref is access all Typ'Class;
```

```
    procedure Proc (This : access Typ; Val : Integer) is abstract;  
    function  Func (This : access Typ) return Integer is abstract;  
  
private  
    pragma Convention (Java, Typ);  
end Foo;
```

7.2 Using Java Interfaces

In order to declare an Ada type that implements one or more Java interfaces it is necessary to use a simple programming idiom that is specially recognized by the compiler. This mechanism is not restricted for use only with types imported from Java, but can be applied to any Ada tagged type *T*.

The idiom consists in specifying a discriminant *D* for each Java interface *Interf* that type *T* implements. The type of *D* must be some access type whose designated type is *Interf*. Discriminants such as *D* are not represented by an actual field in the object, but rather serve as a symbolic shorthand to indicate the special characteristics of type *T* to the compiler.

As an example, the following package spec declares a type `Bar.Type` which implements interface `Foo.Type`:

```
with Foo;
package Bar is
  type Typ (Foo_I : Foo.Ref) is tagged record
    Field : Integer;
  end record;
  -- Discriminant Foo_I above signals that Bar.Type implements the
  -- Foo.Type interface (Foo_I stands for Foo Interface). The
  -- compiler does not create a field for Foo_I but marks the
  -- generated .class file as implementing interface Foo.Type.

  procedure Proc (This : access Typ; Val : Integer);
  function Func (This : access Typ) return Integer;
  -- Unless Bar.Type is itself marked abstract, Bar.Type must
  -- provide an implementation for subprograms Proc and Func.
  -- Right now if you omit these subprograms the GNAT compiler will not
  -- complain, but when loading the .class file corresponding to Bar.Type
  -- the JVM will halt execution with a verifier error.
end Bar;
```

As mentioned in the example, unless *T* is abstract, *T* must provide an implementation for each of the abstract operations of the *Interf* interface (currently this check is not done by the GNAT compiler but is caught later on by the JVM).

A second use of these special interface discriminants is to enable conversions between pointers to type *T* and pointers of its implemented interface types as the example below demonstrates.

```
with Foo;
with Bar;
package Client is
  X : Bar.Ref := new Bar.Type (null);
  -- Create an object of type Bar.Type. To satisfy Ada's semantic rules
  -- we must provide a value for Foo_I, but this value is ignored.

  Y : Foo.Ref := X.Foo_I;
  -- OK, upward conversion allowed, no checks.
  -- Referencing discriminant Foo_I is a convenient way to convert X
  -- to a Foo.Ref. The compiler transforms all references to Foo_I
  -- into references to the selector itself, in this case X.

  Val : Integer := Foo.Func (Y);
  -- Dispatching call
end Client;
```

A conversion can also be made from an object of the class-wide interface reference type to an implementing reference type, by selecting the `Self` field (which is of type `java.lang.Object.Typ'Class`), and then applying a downward tagged type conversion (assuming that `T` derives directly or indirectly from `java.lang.Object`). Such a downward conversion will involve a run-time check, to ensure that the source object belongs to the target type's class. The package spec below illustrates one such downward conversion.

```
with Foo;
with java.lang.Object;
package Zar is
  type Typ (Foo_I : Foo.Ref) is new java.lang.Object.Typ with record
    Field : Integer;
  end record;

  type Ref is access all Typ'Class;

  procedure Proc (This : access Typ; Val : Integer);
  function Func (This : access Typ) return Integer;

  X : Zar.Ref := new Zar.Typ (null);
  Y : Foo.Ref := X.Foo_I;

  Z : Zar.Ref := Zar.Ref (Y.Self);
  -- OK, downward conversion, run-time check that Y designates an
  -- object in Zar.Obj'Class.
  -- Again the compiler ignores the special discriminant Self and
  -- returns the selector itself, in this case Y.
end Zar;
```

In both of the cases shown above, the compiler recognizes the special idiom of selecting the interface or `Self` discriminant as meaning a reference to the object itself, reinterpreting the type of the object appropriately.

The above mechanism can also be used within another `Java_Interface` type as illustrated by the following example:

```
with java.lang.Object;
package Zoo is
  type Typ (Self : access java.lang.Object.Typ'Class) is
    new abstract java.lang.Object.Typ with null record;
  pragma Java_Interface (Typ);

  type Ref is access all Typ'Class;

  procedure Interface_Op (This : access Typ) is abstract;
private
  pragma Convention (Java, Typ);
end Zoo;

with java.lang.Object; use java.lang.Object;
with Foo;
with Zoo;
package Woo is
  type Typ (Foo_I : Foo.Ref;
    Zoo_I : Zoo.Ref)
    is new abstract java.lang.Object.Typ with null record;
  pragma Java_Interface (Typ);

  type Ref is access all Typ'Class;
```



```

-- Woo must list all of the abstract operations of interfaces
-- Foo and Zoo.

procedure Proc (This : access Typ; Val : Integer) is abstract;
function Func (This : access Typ) return Integer is abstract;
procedure Interface_Op (This : access Typ) is abstract;

procedure New_Op (This : access Typ) is abstract;
-- A new operation of the interface
private
  pragma Convention (Java, Typ);
end Woo;

```

Another interesting example is the declaration of a type `Bar.Child.Typ` that derives from `Bar.Typ` and implements interface `Woo.Typ`, as shown below:

```

with Foo;
with Woo;
package Bar.Child is
  type Typ (Foo_I : Foo.Ref;
            Woo_I : Woo.ref)
    is new Bar.Typ (Foo_I) with null record;
  -- Note how Foo_I is used to constrain Bar.Typ. This is just to
  -- satisfy Ada semantics requirements and has no other implications.

  type Ref is access all Typ'Class;

  procedure Proc (This : access Typ; Val : Integer);
  function Func (This : access Typ) return Integer;
  procedure Interface_Op (This : access Typ);
  procedure New_Op (This : access Typ);
end Woo;

```

An interesting point to note is when an Ada tagged type `Deriv` derives from an Ada tagged type `Base` which implements a number of interfaces. If `Deriv` does not implement any additional interface there is no need to specify interface discriminants for `Deriv`, since it can simply inherit those of `Base`.

7.3 The Java_Constructor Pragma

7.3.1 Background on Java Constructors

A Java constructor is a special method that must be invoked immediately after allocating an object, in order to initialize the object. Given the following Java class:

```
public class C {
  public int field;
  public C ()      { field = 3; }
  public C (int i) { field = i; }
}
```

then the statement `C obj = new C (3)` accomplishes two things:

1. It allocates a new instance of class `C` in the Java heap and sets `obj` to point to this object;
2. It then calls the constructor that takes an `int` parameter, passing `obj` to it as a hidden parameter and the value 3 for its `int` parameter.

If no constructor is provided, as in the following class:

```
class D extends C {
  float f;
}
```

then a default constructor

```
public D () {
  super ();
}
```

is automatically generated for class `D`. The call of `super()` inside this default constructor (known as a *no-arg* constructor) invokes the no-arg constructor of the superclass of `D`, that is, the constructor of class `C`.

Generally speaking, the first statement of every constructor must either be a call to another constructor of the class, or a call to a constructor of the superclass. For instance, given a constructor

```
public C (int i, int j) { this (i + j); }
```

The call `this (...)` invokes another constructor in the same class whose profile matches the parameters specified in `(...)`. As another example, consider:

```
public D (int k) { super (k); }
```

where again `super (...)` invokes a constructor in the superclass whose profile matches the parameters specified in `(...)`.

The observant reader will note that in both of the original constructors of class `C`, there are no calls to either `this (...)` or `super (...)`. When no such call is explicitly given, the Java compiler automatically inserts calls to the no-arg constructor in the superclass. If the superclass does not have a no-arg constructor (more on this below), then you must explicitly insert calls to `super (...)` or `this (...)`.

As noted above, a class might not have a no-arg constructor. This can occur only when explicit constructors are defined in the class. In this case, the no-arg constructor is not automatically generated for the class, and if a no-arg constructor is desired, you must add it yourself. For instance, in the following class:

```

public class A {
    int ival;
    public A (int i) { ival = i; }
}

public class B extends A {
    float fval;
    public B (float f) { fval = f; }
}

```

the Java compiler will issue a compile-time error reporting that no constructor matching `A ()` was found in class A, because the compiler tries to insert such a call at the beginning of B. To correct this problem the Java programmer must either add a no-arg constructor `A ()` in class A, or else change the definition of B's constructor to contain an explicit constructor, e.g., as follows:

```

public B (float f) {
    super ();
    fval = f;
}

```

7.3.2 Using Java Constructors in Ada

To assert that an Ada function *function-name* should be mapped to a Java constructor of some Ada *tagged-type*, we have introduced the `Java_Constructor` pragma. Its syntax is as follows:

```
pragma Java_Constructor (function-name);
```

where *function-name* is the name of a function declared immediately within the same declarative part where the pragma occurs, and the function must have the following characteristics:

1. The function's result type is an access type designating a class-wide type with convention Java declared at the same declarative level as the function (`access tagged-type'Class`);
2. The first function parameter is named `This`, and its type is a named access type designating *tagged-type'Class* which may have a `null` default value;
3. If the constructor invokes other constructor then the first declaration in the function body should contain an object declaration with a default initial expression of the form `constructor-func (... , This)`, where the *constructor-func* is a `Java_Constructor` function which belongs either to *tagged-type* or to the parent type of *tagged-type*;

The effect of a `Java_Constructor` pragma is to compile *function-name* into a constructor for the class corresponding to *tagged-type*. In addition, whenever *function-name* is invoked with a `null` value for parameter `This`, the compiler calls the *tagged-type* object allocator and passes in the pointer to the newly allocated object in lieu of the value `null`.

A `Java_Constructor` pragma is a program unit pragma. It can appear in the same places where an `Inline` pragma for *function-name* can appear. The `Java_Constructor` pragma applies to all the overloaded *function-name* subprograms declared immediately within the declarative region containing the pragma.

As an example, the following Java code:

```

public class C {
    public int field;
    public C ()          { field = 3; }
}

```

```
    public C (int i)          { field = i; }  
    public C (int i, int j) { this (i + j); }  
}
```

is equivalent to the following Ada:

```

with java.lang.Object; -- more on this package in the coming sections
use java.lang.Object;
package C is
  use java.lang;

  type Typ is new java.lang.Object.Typ with record
    Field : Integer;
  end record;

  type Ref is access all Typ'Class;

  function new_C (This : Ref := null) return Ref;
  function new_C (I : Integer; This : Ref := null) return Ref;
  function new_C (I, J : Integer; This : Ref := null) return Ref;

private
  pragma Java_Constructor (new_C);
end C;

package body C is
  function new_C (This : Ref := null) return Ref is
    Super : Object.Ref := Object.new_Object (Object.Ref (This));
  begin
    This.Field := 3;
    return This;
  end new_C;

  function new_C (I : Integer; This : Ref := null) return Ref is
    Super : Object.Ref := Object.new_Object (Object.Ref (This));
  begin
    This.Field := I;
    return This;
  end new_C;

  function new_C (I, J : Integer; This : Ref := null) return Ref is
    Ignore : Ref := new_C (I + J, This);
  begin
    return This;
  end new_C;
end C;

```

7.3.3 Java Constructors and Ada Allocators

An interesting question raised by the `Java_Constructor` pragma is the interaction between Ada allocators and constructors. For instance a client of package `C` given in the previous section could write:

```

with C;
procedure Client is
  Obj_1 : C.Ref := new_C;
  Obj_2 : C.Ref := new C.Typ; -- what happens here ???

```

What GNAT does in the allocator case is to call the no-arg constructor if present (in the example `new_C (This : Ref := null)`). If there is no no-arg constructor then an error is emitted by the GNAT compiler (this last check is currently not yet supported, and there will be an exception at run time).

7.4 Pragma Import Java

For convention Java, pragma **Import** has the following syntax:

```
pragma Import ([Convention    =>] Java,
               [Entity        =>] Local_Name
               [, [External_Name =>] String_Expression]);
```

where *Local_Name* is the name of an object, subprogram, record component, exception, or package, while *String_Expression* is a string giving the Java name of the imported entity. If *String_Expression* is missing it is taken to be the *Local_Name*, all in lower-case letters.

7.4.1 Importing Packages

If the *Local_Name* of an **Import** pragma is the name of a package spec *P*, then all the entities declared in *P* must be explicitly imported from Java. The *String_Expression* of such an **Import** pragma gives the name of the Java class corresponding to *P* and can be a simple class name or it can have the form *java_package_name.class_name*, which indicates that the class *class_name* corresponding to *P* belongs to Java package *java_package_name*. If *java_package_name* is missing, the class belongs to the anonymous Java package.

The precise rules when importing a package *P* are:

- All the entities declared inside *P* must be imported either by means of the **Import** pragma or by using other Java-specific pragmas.
- *P* should contain at most one tagged or untagged record type whose name must be *Typ*. *Typ* models the record part of the class corresponding to *P*.
- *P* can contain at most one exception, whose **Import** pragma must have exactly the same *String_Expression* as for *P*. (In *jvm2ada* such an exception is present only if the class corresponding to *P* derives, directly or indirectly, from class `java.lang.Throwable`. The name we have selected for such an exception is **Except**.)
- *P* should not contain task types or protected types.
- The *String_Expression* of the **Import** pragma for an object, subprogram, or record component declared in *P* must be a simple name (it cannot contain any “.” characters).
- If *P* contains nested packages, these must themselves contain an **Import** pragma (and the above rules apply recursively).

As a first example consider the following package:

```
with java.lang.Object; -- more on this package in the coming sections
package root.outer.Child is
  type Typ is new java.lang.Object.Typ with record
    x : Integer;
    pragma Import (Java, x, "x");

    Y : Integer;
    pragma Import (Java, Y, "Y");
  end record;

  type Ref is access all Typ'Class;

  procedure Dispatching_Op (This : access Typ; I : Integer);
  function Non_Dispatching_Op (F : Float) return Integer;

  function New_Child (This : Ref := null) return Ref;
  Global : Integer;
```

```

private
  pragma Import (Java, Dispatching_Op, "someProcedure");
  pragma Import (Java, Non_Dispatching_Op, "someFunction");
  pragma Java_Constructor (New_Child);
  pragma Import (Java, Global);
end root.outer.Child;
pragma Import (Java, Outer.Child, "root.outer.CHILD");

```

This package imports into Ada the services of a class whose spec in Java looks like:

```

package root.outer;
public class CHILD extends java.lang.Object {
    public int x;
    public int Y;

    public void someProcedure (int i);
    public static int someFunction (float f);

    public CHILD ();

    public static int global;
}

```

Note that in the Ada spec, the Java methods `someProcedure` and `someFunction` have been named `Dispatching_Op` and `Non_Dispatching_Op`.

7.4.2 Importing Exceptions

If the *Local_Name* of an `Import` pragma is the name of an exception *E*, the *String_Expression* of such an `Import` pragma gives the name of the JVM class corresponding to *E* and can be a simple class name or it can have the form *java_package_name.class_name* which says that the JVM class *class_name* corresponding to *E* belongs to Java package *java_package_name*. If *java_package_name* is missing, the JVM class belongs to the anonymous Java package.

When importing an exception you should make sure that the imported JVM class is indeed a Java exception, i.e. it derives from `java.lang.Throwable`.

As an example here is an excerpt of the spec of class `java.lang.Throwable` generated by `jvm2ada`:

```

package java.lang.Throwable is
  type Typ ...;
  type Ref is access all Typ'Class;

  Except : Exception;
  ...
private
  pragma Import (Java, Except, "java.lang.Throwable");
  ...
end java.lang.Throwable;
pragma Import (Java, java.lang.Throwable, "java.lang.Throwable");

```

7.4.3 Importing Record Components

If the *Local_Name* of an `Import` pragma is the name of a record field, then the record field must be declared in a record whose convention is Java and the record must be declared in

a package specification which is itself imported. In this case *String_Expression* must be a simple name (i.e. contains no dots) giving the name of the imported field.

7.4.4 Importing Dispatching Subprograms

If the *Local_Name* of an **Import** pragma is the name of a dispatching subprogram (i.e., a primitive operation of a tagged type), then the subprogram must be declared in a package specification which is itself imported. In this case *String_Expression* must be a simple name (i.e. contains no dots) giving the name of the imported subprogram.

7.4.5 Importing Objects

If the *Local_Name* of an **Import** pragma is the name of an object and the object is declared in a package specification which is itself imported the *String_Expression* must be a simple name (i.e. contains no dots) giving the name of the imported Java static field.

An **Import** pragma for an object can be given even though such an entity does not occur in a package spec with an **Import** pragma. In this case the *String_Expression* of the **Import** pragma must give the complete Java name of the imported as shown in the following example:

```
procedure Foo is
  Var : Integer;
  pragma Import (Java, Var, "pack.Foo.the_var");
begin
  Var := 3;
end Foo;
```

7.4.6 Importing Non-Dispatching Subprograms

If the *Local_Name* of an **Import** pragma is the name of a non-dispatching subprogram and the subprogram is declared in a package specification which is itself imported the *String_Expression* must be a simple name (i.e. contains no dots) giving the name of the imported Java static method.

An **Import** pragma for a non-dispatching subprogram can be given even though such an entity does not occur in a package spec with an **Import** pragma. In this case the *String_Expression* of the **Import** pragma must give the complete Java name of the imported as shown in the following example:

```
procedure Foo is
  X : Integer;
  function Compute (I : Integer) return Integer;
  pragma Import (Java, Compute, "pack.Bar.calc");
begin
  X := Compute (3);
end Foo;
```

7.5 Pragma Export Java

In the absence of pragma **Export**, the name of any Ada object, field, or subprogram compiled into a class file is the name of the corresponding Ada entity in lower-case letters.

For exceptions, record types and packages, the names of the generated class files are all in lower case.

By using pragma **Export** the user can change the default name that is generated by the GNAT compiler. In addition, for Ada packages it can also specify which Java package they belong to. For convention Java, the pragma **Export** has the following syntax:

```
pragma Export ([Convention    =>] Java,
              [Entity        =>] Local_Name
              [, [External_Name =>] String_Expression]);
```

where *Local_Name* is the name of an object, subprogram, record component, record type, exception, or package, and *String_Expression* is a string giving the Java name of the exported entity. If *String_Expression* is missing it is taken to be the *Local_Name*, all in lower-case letters.

7.5.1 Exporting Objects, Subprograms, and Record Components

NOTE: Exporting of record components is not yet supported.

If the *Local_Name* of an **Export** pragma is the name of an object, record component, or subprogram (but not a top-level subprogram), *String_Expression* must be a simple name (i.e., it contains no `.` characters), giving the name of the corresponding entity at the JVM level. As an example, when compiling the following package specification:

```
package C is
  type Typ is tagged record
    Field : Integer;
    pragma Export (Java, Field, "THE_FIELD");
  end record;

  function Instance_Op (This : access Typ; I : Integer) return Integer;

  Var : Integer;
  function Op (J : Integer) return Integer;

private
  pragma Export (Java, Instance_Op, "dispatch_op");
  pragma Export (Java, Var, "the_var");
end C;
```

this is interpreted as the following two class specification at the JVM level:

```
public class c {
  public static int the_var;
  public static int op (int j);
}
public class c$typ {
  public int THE_FIELD;
  public int dispatch_op (int i) {...}
}
```

Note that when exporting an object, subprogram, or record component you cannot specify its JVM class, as this is determined by the compiler.

7.5.2 Exporting Exceptions

If the *Local_Name* of an **Export** pragma is the name of an exception *E*, then the *String_Expression* of such an **Export** pragma gives the name of the generated JVM class for the Ada exception, overriding the name that would have been given by the compiler. *String_Expression* can be a simple class name, or it can have the form

java_package_name.class_name

indicating that the generated class belongs to Java package *java_package_name*. If the name *java_package_name* is missing, the class is defined to belong to the anonymous Java package.

Care must be taken not to use the same class name for two Ada exceptions, packages or record types when they belong to different source files located in the same directory, since one `.class` file would overwrite the other.

7.5.3 Exporting Packages or Record Types

NOTE: Exporting of packages is not yet supported.

If the *Local_Name* of an `Export` pragma is the name of a package spec or record type *P*, then the *String_Expression* of such an `Export` pragma gives the name of the generated JVM class, overriding the name that would have been given by the compiler. *String_Expression* can be a simple class name, or it can have the form *java_package_name.class_name* indicating that the generated JVM class belongs to Java package *java_package_name*. If *java_package_name* is missing, the JVM class belongs to the anonymous Java package.

Care must be taken not to use the same class name for two Ada exceptions, packages or record types when they belong to different source files located in the same directory, since one `.class` file would overwrite the other.

If the same `Export` pragma is specified for a package spec and a record type contained inside it, then the GNAT compiler will map both of these in the same JVM class. For instance without `Export` pragmas the following code generates 2 JVM `.class` files: `'outer$child.class'` and `'outer$child$rec.class'`.

```
package Outer.Child is
  type Rec is tagged record
    F : Float;
  end record;
  procedure Proc (This : Rec);
  -- This always goes in the same .class file as type Rec

  function Global (I : Integer) return Rec;
  -- This always goes in the same .class file as the package
end Outer.Child;
```

If the same `Export` pragma is used a single class file is generated (`'CHILD.class'` in JVM package `root.outer`).

```
package Outer.Child is
  type Rec is record
    X : Float;
  end record;
  pragma Export (Java, Rec, "root.outer.CHILD");

  procedure Proc (This : Rec);
  function Global (I : Integer) return Rec;
  -- Both subprograms are generated in the same .class file
end Outer.Child;
pragma Export (Java, Outer.Child, "root.outer.CHILD");
```

8 Mapping Java into Ada

This chapter details the mapping used by `jvm2ada` to map Java ‘.class’ files into Ada package specs. It is assumed that the reader is familiar with the Java language.

8.1 Identifiers

See [Section 6.6 \[Identifier Mangling\]](#), page 18.

8.2 Scalar Types

Java scalar types are mapped into Ada scalar types as follows:

<code>boolean</code>	<code>(1 byte)</code>	<code>maps into</code>	<code>Standard.Boolean</code>
<code>char</code>	<code>(2 bytes)</code>	<code>" "</code>	<code>Standard.Wide_Character</code>
<code>byte</code>	<code>(1 byte)</code>	<code>" "</code>	<code>Standard.Short_Short_Integer</code>
<code>short</code>	<code>(2 bytes)</code>	<code>" "</code>	<code>Standard.Short_Integer</code>
<code>int</code>	<code>(4 bytes)</code>	<code>" "</code>	<code>Standard.Integer</code>
<code>long</code>	<code>(8 bytes)</code>	<code>" "</code>	<code>Standard.Long_Integer</code>
<code>float</code>	<code>(4 bytes)</code>	<code>" "</code>	<code>Standard.Float</code>
<code>double</code>	<code>(8 bytes)</code>	<code>" "</code>	<code>Standard.Long_Float</code>

As a convenience for the Ada programmer subtypes are used to express the correspondence between primitive numeric Java types and the Ada scalar types defined in package `Standard`.

We have chosen to place these subtypes at the root of the Ada version of the Java API, i.e., in package `Java`. Thus these subtypes are directly available in the Ada version of the API and at hand for users of the API. The code excerpt below gives the beginning of package `java`:

```
package java is
  pragma Preelaborate;

  subtype boolean is Standard.Boolean;
  subtype char    is Standard.Wide_Character;
  subtype byte    is Standard.Short_Short_Integer;
  subtype short   is Standard.Short_Integer;
  subtype int     is Standard.Integer;
  subtype long    is Standard.Long_Integer;
  subtype float   is Standard.Float;
  subtype double  is Standard.Long_Float;
  ...
end java;
```

8.3 Java References and `java.lang.Object`

When it comes to composite objects such as arrays and records, Java differs from Ada in the fact that it only has reference semantics. More precisely, in Java you can only allocate an object in the garbage-collected heap and obtain a reference to such object. All object reads and writes are done via this reference. In addition, you cannot copy an object as a whole into another object: there is no default deep-copy operation in Java.

In mapping Java services to Ada, we have preserved its reference semantics, as shown in the code excerpt below which shows the salient part of how class `java.lang.Object` is mapped into Ada:

```

package java.lang.Object is
  pragma Preelaborate;

  type Typ is tagged limited private;
  type Ref is access all Typ'Class;

  function new_Object (This : Ref := null) return Ref;
  -- The constructor
  ...
private

  type Typ is tagged limited null record;
  ...
end java.lang.Object;

```

As a first remark, tagged types imported from Java should be limited since, as mentioned before, no object assignment operation exists on the JVM. Second, unlike Java, in Ada we need to define two types: one for the actual tagged type (type `Typ`) and one for the actual references (type `Ref`). This means that while in Java you can write something like:

```

import java.lang.Object;
class client {
  void foo () {
    Object obj = new Object ();
  }
}

```

in Ada you have to write:

```

with java.lang.Object; use java.lang.Object;
procedure Foo is
  obj : Object.Ref = new_Object;
begin
  null;
end Foo;

```

Furthermore, for now we impose some restrictions on types that extend types that are declared in packages imported from Java.

Since the parent type of such a type extension has convention `Java`, the extended type inherits convention `Java` (even though declared within a normal Ada package). This means that the extended type should not contain any component declarations that would not be appropriate in an equivalent Java class.

In particular, a type that extends from a Java-convention parent type should not have any components of the following kinds:

- components with default initialization (excepting access components that are initialized by null)
- components of composite types (arrays, records, tasks, protected types)
- components of private types whose full type is a composite type

The reason for these restrictions is that each of the above forms of components requires some kind of run-time initialization at the time an object of the containing type is created. Such initialization needs to happen before any user code can reference the components. However, in the presence of user-defined constructors, which are executed immediately after object creation (as required by the JVM), this is difficult for the compiler to support.

The workaround for cases where composite components are desired is instead to declare components of access types that designate the types you want to use. The allocation and

initialization of those access components can then be performed as part of the actions of your own user-defined constructor function (see [Section 7.3 \[The Java_Constructor Pragma\]](#), [page 24](#)).

If we were to allow such components, the consequences of failing to heed the above restrictions would include the creation of objects that are not fully allocated or initialized, with the potential for crashing the program.

The GNAT compiler will reject the attempt to declare a Java-convention record type with any of the restricted forms of components by flagging each offending component. We plan to try relaxing these restrictions in a future release.

8.4 Array Types

Let's illustrate the mapping with an example. Assume you would like to map a Java array of `int` into Ada. For instance, in Java you might write:

```
int[] obj;
// obj is a reference to an array of int

obj = new int [3];
// Allocate an array-of-int object with 3 elements.
// The index of the first element is zero.
```

The above is mapped into the following Ada declarations:

```
type int_Arr_Obj is array (Natural range <>) of int;
type int_Arr      is access all int_Arr_Obj;

obj : int_Arr;
-- int[] obj;

obj := new int_Array_Obj (0 .. 2);
-- obj = new int [3];
```

Java does not have multidimensional arrays, it only has arrays of arrays. As a result

```
int[][] obj2 = new int [3][2];
```

is mapped into

```
type int_Arr_2_Obj is array (Natural range <>) of int_Arr;
type int_Arr_2      is access all int_Arr_2_Obj;

Obj2 : int_Arr_2 := new int_Arr_2_Obj (0 .. 2, 0 .. 1);
-- obj2 = new int [3][2];
```

The final question that remains to answer is where are the various array type definitions located. Scalar array types have been placed in package `Java`, while array types associated with a given JVM class are placed in the Ada package spec for that class. Indeed when processing a JVM class `C`, `jvm2ada` generates the following at the beginning of the Ada package spec corresponding to `C`:

```
package C is
  pragma Preelaborate;

  type Typ is ...;
  type Ref is access all Typ'Class;

  type Arr_Obj is array (Natural range <>) of Ref;
  type Arr      is access all Arr_Obj;
```

```
type Arr_2_Obj is array (Natural range <>) of Arr;  
type Arr_2     is access all Arr_2_Obj;  
  
...  
end C;
```

8.5 The Ada Package Java

```

package java is
  pragma Preelaborate;

  subtype boolean is Standard.Boolean;
  subtype char    is Standard.Wide_Character;
  subtype byte    is Standard.Short_Short_Integer;
  subtype short   is Standard.Short_Integer;
  subtype int     is Standard.Integer;
  subtype long    is Standard.Long_Integer;
  subtype float   is Standard.Float;
  subtype double  is Standard.Long_Float;

  -- boolean array types: boolean [], boolean [][], boolean [][][]

  type boolean_Arr_Obj is array (Natural range <>) of boolean;
  type boolean_Arr     is access all boolean_Arr_Obj;

  type boolean_Arr_2_Obj is array (Natural range <>) of boolean_Arr;
  type boolean_Arr_2     is access all boolean_Arr_2_Obj;

  type boolean_Arr_3_Obj is array (Natural range <>) of boolean_Arr_2;
  type boolean_Arr_3     is access all boolean_Arr_3_Obj;

  -- char array types: char [], char [][], char [][][]

  type char_Arr_Obj is array (Natural range <>) of char;
  type char_Arr     is access all char_Arr_Obj;

  type char_Arr_2_Obj is array (Natural range <>) of char_Arr;
  type char_Arr_2     is access all char_Arr_2_Obj;

  type char_Arr_3_Obj is array (Natural range <>) of char_Arr_2;
  type char_Arr_3     is access all char_Arr_3_Obj;

  -- byte array types: byte [], byte [][], byte [][][]

  type byte_Arr_Obj is array (Natural range <>) of byte;
  type byte_Arr     is access all byte_Arr_Obj;

  type byte_Arr_2_Obj is array (Natural range <>) of byte_Arr;
  type byte_Arr_2     is access all byte_Arr_2_Obj;

  type byte_Arr_3_Obj is array (Natural range <>) of byte_Arr_2;
  type byte_Arr_3     is access all byte_Arr_3_Obj;

  -- short array types: short [], short [][], short [][][]

  type short_Arr_Obj is array (Natural range <>) of short;
  type short_Arr     is access all short_Arr_Obj;

  type short_Arr_2_Obj is array (Natural range <>) of short_Arr;
  type short_Arr_2     is access all short_Arr_2_Obj;

  type short_Arr_3_Obj is array (Natural range <>) of short_Arr_2;
  type short_Arr_3     is access all short_Arr_3_Obj;

  -- int array types: int [], int [][], int [][][]

```

```

type int_Arr_Obj   is array (Natural range <>) of int;
type int_Arr       is access all int_Arr_Obj;

type int_Arr_2_Obj is array (Natural range <>) of int_Arr;
type int_Arr_2     is access all int_Arr_2_Obj;

type int_Arr_3_Obj is array (Natural range <>) of int_Arr_2;
type int_Arr_3     is access all int_Arr_3_Obj;

-- long array types: long [], long [][], long [][][]

type long_Arr_Obj   is array (Natural range <>) of long;
type long_Arr       is access all long_Arr_Obj;

type long_Arr_2_Obj is array (Natural range <>) of long_Arr;
type long_Arr_2     is access all long_Arr_2_Obj;

type long_Arr_3_Obj is array (Natural range <>) of long_Arr_2;
type long_Arr_3     is access all long_Arr_3_Obj;

-- float array types: float [], float [][], float [][][]

type float_Arr_Obj   is array (Natural range <>) of float;
type float_Arr       is access all float_Arr_Obj;

type float_Arr_2_Obj is array (Natural range <>) of float_Arr;
type float_Arr_2     is access all float_Arr_2_Obj;

type float_Arr_3_Obj is array (Natural range <>) of float_Arr_2;
type float_Arr_3     is access all float_Arr_3_Obj;

-- double array types: double [], double [][], double [][][]

type double_Arr_Obj   is array (Natural range <>) of double;
type double_Arr       is access all double_Arr_Obj;

type double_Arr_2_Obj is array (Natural range <>) of double_Arr;
type double_Arr_2     is access all double_Arr_2_Obj;

type double_Arr_3_Obj is array (Natural range <>) of double_Arr_2;
type double_Arr_3     is access all double_Arr_3_Obj;

end java;
```


8.6 Use of Limited-With Clauses by jvm2ada

Consider the following

```
public class C extends B implements I {
    public D link;
    ...
}
```

When `jvm2ada` is invoked to process ‘`C.class`’ to create an Ada spec for `C`, it is difficult to know whether reference type `D` is involved in a circular dependency with class `C`. First of all, `D.class` may not even be available yet. Even if it were, classes `C` and `D` could be involved in a complex circular relationship with other classes, some of which may not be available. In addition, in the presence of circular dependencies there is no good reason for using a `with type` clause in one package spec and a regular `with` clause in another.

There are three cases in which a class `C` refers to another class `X`:

1. `C` extends `X`.
2. `C` implements `X`.
3. `X` is used as a reference (e.g., a reference to an object or array of objects of type `X`).

In the first case, the Ada package spec corresponding to `C` needs to have a regular `with` clause for the package spec corresponding to `X`. In all other cases a `with type` clause suffices.

As a result, `jvm2ada` will generate the following `with` and `limited with` clauses for the class `C` above:

```
with B;
limited with I;
limited with D;
```

8.7 Java Packages

A Java package *pack* is mapped into an empty Ada package spec *pack*. For instance, Java package `java.lang` is mapped into the following Ada spec:

```
package java.lang is
    pragma Preelaborate;
end java.lang;
```

8.8 Java Classes

A Java class `X.Y.D` extending a class `W.Z.B` is mapped into an Ada package `X.Y.D` containing a tagged type definition `Typ` that extends `W.Z.B Typ` and a reference type definition `Ref` as shown below:

```
with Java; use Java;

package X.Y.D is
    pragma Preelaborate;

    -----
    -- Type Declarations --
    -----

    type Typ;
    type Ref is access all Typ'Class;
```

```

-- Array type declarations for X.Y.D

type Arr_Obj is array (Natural range <>) of Ref;
type Arr      is access all Arr_Obj;

type Arr_2_Obj is array (Natural range <>) of Arr;
type Arr_2     is access all Arr_2_Obj;

type Arr_3_Obj is array (Natural range <>) of Arr_2;
type Arr_3     is access all Arr_3_Obj;

-- The actual type declaration for X.Y.D

type Typ is new W.Z.B.Typ
  with record
    -----
    -- Field Declarations --
    -----
    ...
  end record;

-----
-- Constructor Declarations --
-----
...

-----
-- Method Declarations --
-----
...

-----
-- Variable Declarations --
-----
...

private
  pragma Convention (Java, Typ);
  ... -- other pragmas are generated here
end X.Y.D;
pragma Import (Java, X.Y.D, "W.Z.B");

```

In addition, all of the static variables of *X.Y.D* are mapped into variables of the Ada package and all static methods of *X.Y.D* are mapped into nondispatching subprograms in the Ada package.

Each instance method of class *X.Y.D* is converted into a primitive operation of type *X.Y.D.Typ* whose first parameter is of type “*access Typ*” and whose remaining parameters are as in the Java class.

Constructors in class *X.Y.D* are mapped into subprograms in package *X.Y.D* as described in a previous section (see [Section 7.3 \[The Java_Constructor Pragma\]](#), page 24).

8.9 Abstract Classes

Java’s abstract classes are exactly equivalent to Ada’s abstract tagged types and are mapped into such types by *jvm2ada*.

8.10 Nested Classes

A class `Inner` nested inside a class `Outer` is mapped into a child package named `Outer.Inner`.

8.11 Java Interface

See [Section 7.1 \[Creating Java Interfaces with Pragma `Java_Interface`\]](#), page 19.

8.12 Java Class Implementing Interfaces

See [Section 7.2 \[Using Java Interfaces\]](#), page 21.

8.13 Java Exceptions

When processing a JVM class, `jvm2ada` must figure out whether this class is a Java exception, i.e. whether it derives, directly or indirectly from class `java.lang.Throwable`.

To determine this, `jvm2ada` traverses the inheritance tree and must locate the ‘`.class`’ files of the ancestor classes (see [Section 6.5 \[Class File Search Paths\]](#), page 17). If the class is indeed a Java exception, an Ada exception is added to the generated Ada spec as shown in the following example:

```
package pack1.pack2;
public class C extends java.lang.Throwable { ... }
```

is mapped into

```
package pack1.pack2.C is
...
  Except : Exception;
...
end pack1.pack2.C;
```

8.14 Static Fields

A static field in Java is equivalent to a regular variable in Ada and is mapped accordingly by `jvm2ada`.

8.15 Final Static Fields

A Java `final` static field is equivalent to an Ada constant. When importing a final static field from Java, `jvm2ada` maps each such field to an Ada deferred constant with an associated pragma `Import Java`.

8.16 Instance Fields

An instance field is mapped by `jvm2ada` into a field of its corresponding tagged type.

8.17 Volatile and Transient Fields

The JVM `volatile` and `transient` attributes are currently ignored by `jvm2ada`.

8.18 Static Methods

A Java static method is equivalent to a regular nondispatching subprogram in Ada and is mapped that way by `jvm2ada`.

8.19 Instance Methods

Each instance method is converted into a primitive operation whose first parameter is of type “`access Typ`” and whose remaining parameters are as given for the Java method.

8.20 Abstract Methods

Java's abstract methods are exactly equivalent to Ada's abstract primitive operations and are mapped accordingly by `jvm2ada`.

8.21 Native Methods

In Java one can assert that a certain method is **native**, i.e., that its implementation is provided in some native language such as C or Ada, external to the JVM. The `jvm2ada` tool ignores the **native** attribute when mapping JVM methods into Ada, since JVM methods are invoked in exactly the same way regardless of whether they have a **native** attribute.

8.22 Final Classes and Final Methods

Java has a way to restrict further derivation from a class type or further overriding of a primitive operation. For instance, given

```
public class Base { ... }
public final class Deriv extends Base { ... }
```

it is possible to create subclasses of `Base` but not of `Deriv`, since `Deriv` is marked **final**. Likewise, given

```
public class Base {
  public      int service_1 () { ... }
  public final int service_2 () { ... }
}
```

it is possible to override method `service_1` in every subclass of `Base`, whereas `service_2` cannot be overridden.

Limiting type derivation and primitive operation overriding is not directly possible in Ada. We have therefore chosen for the time being to ignore **final** classes and methods (a comment is emitted before them but nothing more). If a Java final class is extended or a final method overridden in Ada, an exception (or verifier error) will be emitted at execution time.

8.23 Visibility Issues

- A public Java class is mapped into a public Ada package spec.
- A non-public Java class containing **public** nested classes is mapped into an empty Ada package (much in the same way Java packages are mapped into Ada packages). This ensures that the parent Ada package is defined for the child classes.

- A **public** Java member is mapped into an entity declared in the public part of the corresponding Ada package if all of its parameter and return types refer to a public Java class.
- A **protected** Java member is mapped into an entity declared in the public part of the corresponding Ada package and treated like a **public** Java member as described in the above bullet point. Because in Ada there is no concept similar to the **protected** qualifier in Java, if you use a protected entity from Ada in a way that is forbidden by the JVM a run-time exception will occur.
- A **private** Java member corresponds to an Ada entity declared in a package body. `jvm2ada` ignores all such entities.
- Java entities declared as non-public (i.e. with no Java access qualifiers) are visible only to Java classes belonging to the same Java package and are intended to be accessed only from classes in the same API as C. `jvm2ada` ignores all such entities.

8.24 Java Implicit Upcasting in Ada

Given the following two class definitions:

```
public class Base {
    public static void proc (Base p, Base q) {...}
}
public class Derived extends Base {...}
```

Java allows the following code to be written (and Java programmers take advantage of the following):

```
Base    obj1 = new Base ();
Derived obj2 = new Derived ();
proc (obj1, obj2);
```

The implicit conversion from the pointer to the `Derived` type to the `Base` type is completely safe and is allowed in Java much as Ada allows implicit conversion between similar anonymous access types (in Java when you write "`Derived obj2`" you are really saying that `obj2` is a pointer to an object of type `Derived` whose pointer type is anonymous). If `jvm2ada` mapped class `Base` into:

```
package Base is
    type Typ;
    type Ref is access all Typ'class;
    ...
    procedure proc (P1 : Base.Ref; P2 : Base.Ref);
    ...
end Base;
```

The call to `proc` in Ada would have to look like

```
obj1 : Base.Ref    := new_Base;
obj2 : Derived.Ref := new_Derived;
proc (obj1, Base.Ref (obj2));
```

which is verbose (especially when using real class names) without any fundamental reason (writing "`proc (obj1, obj2.all'access)`" would hardly be any terser). To address this inconvenience we have used access parameters when generating the Ada equivalent to `proc`:

```
procedure proc (P1 : access Base.Typ'Class;
               P2 : access Base.Typ'Class);
```

allowing us to write:

```

obj1 : Base.Ref    := new_Base;
obj2 : Derived.Ref := new_Derived;
proc (obj1, obj2);

```

The observant reader will notice that this new mapping into Ada of procedure `proc` is not equivalent to the first one since in the case of access parameters Ada checks that the parameters are not `null` (and an exception will be raised if they are).

This problem is worked around by disabling the `null` access check in package specs imported from Java. Note that this is a temporary expedient until a solution to this problem is agreed upon by the ISO WG9-sponsored Ada Rapporteur Group (ARG).

8.25 Mixing Ada Strings and Java Strings

To facilitate the use of regular Ada strings in Java routines we have added the following type definition and subprograms to the body of package `java.lang.String` generated by `jvm2ada`:

```

-----
-- Java String to Ada String Conversion Routines --
-----

type String_Access is access all Standard.String;
function "+" (S : java.lang.String.Ref) return String_Access;
function "+" (S : Standard.String) return java.lang.String.Ref;

```

With the above you can write:

```

procedure P (JS : java.lang.String.Ref);
function F return java.lang.String.Ref;

S_Ptr String_Access := +F;
P ("hello GNAT for the JVM");

```

8.26 An Example

As an example consider the following Java class

```
package A.B;

public class Foo
    extends java.awt.event.ComponentAdapter
    implements java.io.Serializable
{
    // Instance Variables

    public int i_Field;
    public float f_Field;

    // Constructors

    public Foo () {}
    public Foo (java.lang.String s) {}

    // Instance Methods

    public void first_op (java.lang.Thread t) {}
    public int second_op () {return 1;}

    // Static Variables

    public static int i_Var;
    public static float f_Var;

    // Static Methods

    public static void proc (Foo obj, int [] b) {}
    public static int funct (Foo [] [] a, int l, int h) {return 1;}
}
```

After processing this class using the following `jvm2ada` command (*java-dir* is the location of Sun's JDK 1.2 installation):

```
jvm2ada -Ljava-dir/jdk1.2.2/jre/lib/rt.jar Foo.class
```

we obtain the following Ada spec (comments in italics have been added for explanatory purposes):

```
with Java; use Java;
with java.awt.event.ComponentAdapter;
with type java.awt.event.ComponentListener.Ref is access;
with type java.io.Serializable.Ref is access;
with type java.lang.String.Typ is tagged;
with type java.lang.Thread.Typ is tagged;

package A.B.Foo is
    pragma Preelaborate;

    -----
    -- Type Declarations --
    -----

    type Typ;
    type Ref is access all Typ'Class;
```

```

type Arr_Obj is array (Natural range <>) of Ref;
type Arr      is access all Arr_Obj;

type Arr_2_Obj is array (Natural range <>) of Arr;
type Arr_2     is access all Arr_2_Obj;

type Arr_3_Obj is array (Natural range <>) of Arr_2;
type Arr_3     is access all Arr_3_Obj;

type Typ
  (ComponentListener_I : java.awt.event.ComponentListener.Ref;
   -- In the Ada mapping all implemented interfaces must appear
   -- in the list of discriminants. The discriminants corresponding
   -- to those inherited from the parent type (in this case
   -- Component_Listener) must be used to constrain the parent
   -- type in the Ada tagged type definition.
   Serializable_I : java.io.Serializable.Ref)
is new
  java.awt.event.ComponentAdapter.Typ (ComponentListener_I)
with record

  -----
  -- Field Declarations --
  -----

  i_Field : java.Int;
  pragma Import (Java, i_Field, "i_Field");

  f_Field : java.Float;
  pragma Import (Java, f_Field, "f_Field");

end record;

-----
-- Constructor Declarations --
-----

function new_Foo (This : Ref := null)
  return Ref;

function new_Foo (P1 : access java.lang.String.Typ'Class;
  This : Ref := null)
  return Ref;

-----
-- Method Declarations --
-----

procedure first_op (This : access Typ;
  P1 : access java.lang.Thread.Typ'Class);

function funct (P1 : A.B.Foo.Arr_2;
  P2 : java.Int;
  P3 : java.Int)
  return java.Int;

procedure proc (P1 : access A.B.Foo.Typ'Class;
  P2 : java.Int_Arr);

```



```

function second_op (This : access Typ)
    return java.Int;

-----
-- Variable Declarations --
-----

i_Var : Java.Int;
f_Var : Java.Float;

private

pragma Convention (Java, Typ);
pragma Java_Constructor (new_Foo);
pragma Import (Java, first_op, "first_op");
pragma Import (Java, funct, "funct");
pragma Import (Java, proc, "proc");
pragma Import (Java, second_op, "second_op");
pragma Import (Java, i_Var, "i_Var");
pragma Import (Java, f_Var, "f_Var");

end A.B.Foo;
pragma Import (Java, A.B.Foo, "A.B.Foo");

```

The following code shows you how to use Foo's services in your code.

```

with java.lang.String; use java.lang.String;
with A.B.Foo; use A.B.Foo;
use A.B;
use Java;

procedure Client is
    O : Foo.Ref      := new_Foo ("hello there");
    AO : Foo.Arr_2    :=
        new Foo.Arr_2_Obj'
            (0..9 =>
                new Foo.Arr_Obj' (0..3 => new_Foo));

    I : int          := funct (AO, AO'Last, 7);
    AI : Java.Int_Arr := new Java.Int_Arr_Obj (0..5);

begin
    proc (O, AI);
end Client;

```


9 Creating Gnapplets with GNAT

This chapter explains how you can use GNAT to create a “gnapplet” (GNAT applet).

The examples provided with your GNAT installation contain a couple of gnapplet examples. This chapter explains the steps that you need to take to create your own gnapplets.

9.1 Extending `java.applet.Applet.Typ`

The first thing you need to do to create an applet is to extend `java.Applet.Applet.Typ` as shown in the following example:

```
with java; use java;
with java.applet.Applet;
with java.awt.Graphics;

package Animate is
  -- Typ implements the same interfaces as Applet.Typ and no
  -- new interfaces, so we do not need to add discriminants to the
  -- type definition below.

  type Typ is new java.applet.Applet.Typ with record
    Count : Integer;
  end record;
  type Ref is access all Typ'Class;

  procedure Init      (This : access Typ);
  procedure Start     (This : access Typ);
  procedure Stop      (This : access Typ);
  procedure Destroy   (This : access Typ);

private
  pragma Convention (Java, Typ);
end Animate;
```

In addition to extending `java.applet.Applet.Typ`, some functions from `java.applet.Applet` need to be overridden, namely:

```
procedure Init (This : access Typ);
-- This function is called the first time the JVM initializes the
-- the applet. This is where you should put your own initializations
-- as well as the call to the elaboration code for the Ada runtime
-- library, as shown in the next section.

procedure Start (This : access Typ);
procedure Stop  (This : access Typ);
-- These routines are called when your applet starts or stops
-- running (e.g., when it becomes visible to the user,
-- or when the user moves to another page)

procedure Destroy (This : access Typ);
-- Called by the browser or applet viewer to inform this applet
-- that it is being reclaimed and that it should destroy any
-- resources that it has allocated. The stop method will always
-- be called before destroy.
-- A subclass of Applet should override this method if it has
-- any operation that it wants to perform before it is destroyed.
```

In addition you may want to override the following two methods:

```

procedure Paint (This      : access Typ;
                 Graphics : access Java.Awt.Graphics.Typ'Class);
-- Called every time the applet needs to be repainted.
-- Every drawing should be done on Graphics. See the Java API
-- documentation for more info.

procedure Update (This : access Typ;
                 G      : access java.awt.Graphics.Typ'Class);
-- The AWT calls this method in response to a call to
-- repaintupdate or paint. See the Java API
-- documentation for more info.

```

9.2 Initializing and Finalizing the GNAT Runtime

When writing the code for your gnapplet you must remember to initialize the GNAT runtime upon startup of your applet. You must also remember to finalize the GNAT runtime upon destruction of your applet. This is easy to do: you just need to call the routine `Adainit` in method `Init` and the routine `Adafinal` in method `Destroy`. As usual `Adainit` and `Adafinal` have been generated by `jvm-gnatbind` if switch `-n` is selected.

The only additional thing you need to know is the name of the class file where `jvm-gnatbind` generates these routines. The name of this class file is `ada_gnapplet-name`, where `gnapplet-name` is the name of the gnapplet package where `java.applet.Applet.Typ` was derived.

As an example here is the body of methods `Init` and `Destroy` of the `Animate` gnapplet example given in the previous section:

```

package body Animate is
...
  procedure Init (This : access Typ) is
    procedure Adainit;
    pragma Import (Ada, Adainit, "ada_animate.adainit");
  begin
    Adainit;
    -- other initializations go here, after the call to Adainit
  end Init;

  procedure Destroy (This : access Typ) is
    procedure Adafinal;
    pragma Import (Ada, Adafinal, "ada_animate.adafinal");
  begin
    -- other finalizations go here, before the call to Adafinal
    Adafinal;
  end Destroy;
...
end Animate

```

9.3 Compiling the Gnapplet

Once you have written your gnapplet, you need to compile it. This is done in the usual fashion, except for the fact that because there is no main program you need to call the binder and the linker by hand. As an example, to compile the gnapplet given in package `Animate` given in the previous example you can type:

```

$ jvm-gnatmake animate
$ jvm-gnatbind -n animate.ali

```

```
$ jvm-gnatlink animate.ali  
$ jarmake -o animate.jar animate$typ.class
```

The `jarmake` command is particularly important for packaging all the `.class` files needed by your gnapplet in a single zip archive.

9.4 Creating the HTML file

The last step before running your gnapplet is to create an HTML file that will be loaded into your viewer or browser. This HTML file should include a special tag that indicates where you want to run your applet, and what size its allotted window should be. The `WIDTH` and `HEIGHT` parameters below are mandatory. The `ARCHIVE` parameter is only required when you created a zip archive (as you did above with `jarmake`). As an example, a minimal html file for the previous example could be:

```
<html><head></head></head>  
<body>  
  <APPLET CODE="animate$typ.class"  
    ARCHIVE="animate.jar"  
    WIDTH=200 HEIGHT=200> </APPLET>  
</body> </html>
```


10 Debugging Ada Programs

Because GNAT generates class files that are fully compliant with Sun's JVM standard, you can use any JVM debugger, such as Sun's `jdb`, with GNAT.

The minor drawback of using a JVM debugger directly is that for those Ada constructs that are not directly available in the Java programming language (e.g. attributes), you need to know how GNAT compiles these into bytecode in order to retrieve their value.

The purpose of this chapter is precisely to explain how the GNAT compiler compiles Ada constructs into bytecode so that you can use any Java debugger on your Ada application.

Note that this chapter is not yet complete. If you need to understand what the GNAT compiler generates for a particular Ada construct which is not documented below, we suggest running the command `jvmlist -g` on the generated JVM `.class` files.

10.1 Ada Compilation Units and JVM Class Files

Unless pragma `Export` is used (see [Section 7.5 \[Pragma Export Java\]](#), page 30), lower-case letters are used for the names of all JVM class files generated from the compilation of an Ada unit. Likewise, the names of all of the entities generated inside a class file are lower-case.

The compilation of a nongeneric Ada library unit *P* always generates a JVM class file (`'p.class'`). In addition to the class file `'p.class'`, separate class files are emitted for each nested exception, record type, and tagged type declared in unit *P*. More specifically:

- A subprogram library unit *P* is compiled into a JVM class file `'p.class'` containing all the objects and nondispatching operations defined inside *P*.
- The specification and body of a library package *P* are compiled into a single JVM class file `'p.class'` containing all of the objects and nondispatching operations defined in the spec or body of *P*.
- Any tagged or untagged record type *R* declared inside a package or subprogram *P* is treated like a static inner class in Java, that is, a new JVM class `'p$r.class'` is generated containing *R*'s fields as instance variables, and, if *R* is a tagged type, its associated dispatching operations.
- A package *Q* nested inside an Ada unit *P* does not result in a separate class. Entities declared within the nested package will generally be associated as members of the containing library package's class (except in the case of exceptions or type declarations that, as usual, result in their own class). However, the names of the corresponding fields and methods resulting from the nested package will be given expanded names that include the name of the outermost library package followed by the names of any enclosing nested packages, and where adjacent pairs of simple names in the expanded are separated by dollar sign characters (e.g., `'pqproc'`).
- A child unit *P.Q* is compiled into the JVM class file `'p$q.class'`. All the rules described here are applied recursively with respect to *Q*'s contents.
- A generic package instantiation *R* nested inside an Ada unit *P* is treated exactly like a nested package.
- A generic subprogram instantiation *S* nested inside an Ada unit *P* treated exactly like a nested subprogram.

- An Ada exception *E* declared inside an Ada unit *P* is treated like a member class in Java and is compiled into JVM class ‘*p\$e.class*’.
- A task type or protected type *T* nested inside an Ada unit *P* is compiled into JVM class file ‘*p\$t.class*’.
- A subprogram *N* nested inside another subprogram *P* will be treated as a static method of the enclosing library unit's class and will be given an expanded name that includes the names of any enclosing subprograms (e.g., ‘*pkg\$p\$n*’). In addition, a special class will be generated for the nested subprogram's enclosing subprogram to contain fields for any objects of the enclosing subprogram that are referenced by the nested subprogram. The name of this special Activation Record class is constructed by appending the prefix ‘*__AR_*’ to the name of the enclosing subprogram (e.g., ‘*__AR_pkg\$p*’).
- Any other Ada type or construct *X* occurring inside an Ada unit *P* that needs to generate a standalone JVM class will be compiled into ‘*p\$x.class*’.

As an example, the compilation of the following package:

```
package Outer is
  package Nested is
    type Typ is tagged record
      Field : Integer;
    end record;

    E : exception
  end Nested;

  type Rec is record
    X : Float;
  end record;
end Outer;
```

yields the following JVM class files: ‘*outer.class*’, ‘*outer\$nested\$typ.class*’, ‘*outer\$nested\$e.class*’, ‘*outer\$rec.class*’.

10.2 Lexical Elements

All Ada identifiers are mapped into the corresponding lower case identifiers when generating symbolic references for the JVM, unless `pragma Export` is used (but note that certain names corresponding to internal entities generated by the GNAT front end may include upper-case letters).

10.3 Enumeration Types

An Ada enumeration type is converted into a Java 1-byte, 2-byte, 4-byte or 8-byte integer whose size best matches the value of the largest enumeration literal.

Character types are treated like regular Ada enumeration types. More specifically, an Ada `Character` type is mapped into a Java `byte`, whereas an Ada `Wide_Character` type is mapped onto the equivalent 2-byte Java `char` type.

An Ada boolean type is treated like a standard Ada enumeration type with 2 values and is consequently mapped into a Java `byte`.

10.4 Integer Types

Each signed integer type is mapped into the smallest corresponding JVM integer type whose size is able to represent all required integer values:

Short_Short_Integer	<i>is mapped into</i>	byte	(1 byte)
Short_Integer	" "	short	(2 bytes)
Integer	" "	int	(4 bytes)
Long_Integer	" "	long	(8 bytes)
Long_Long_Integer	" "	long	(8 bytes)

Each modular type is also mapped into the smallest corresponding Java integer type whose size is able to represent all required modular values.

10.5 Floating Point Types

The Ada predefined floating point types map very naturally onto Java's IEEE 32-bit float and IEEE 64-bit double:

Short_Float	<i>is mapped into</i>	float	(4 bytes)
Float	" "	float	(4 bytes)
Long_Float	" "	double	(8 bytes)
Long_Long_Float	" "	double	(8 bytes)

User-defined floating point types are mapped into `Float` where possible, and `Long_Float` otherwise.

11 Limitations

Due to constraints of the JVM environment, or to implementation limitations, GNAT only supports a subset of the Ada language and GNAT run-time.

Language constructs not supported:

- Types imported from Java do not support enumeration attributes (e.g. `'Image`)
- Exception streams and attributes
- Representation items (13.1)
- `pragma Pack` (ignored) (13.2)
- Representation attributes (13.3)
- Record layout (13.5)
- Machine Code Insertions (13.8)
- `Unchecked_Conversion` between different non scalar types (13.9)
- Limited support on `Ada.Streams` package (13.13)
- `'Size` attribute on non scalar objects
- `'Storage_Size` attribute on non-task objects
- `'First_Bit`, `'Last_Bit`, `'Position` attributes
- `'External_Tag` attribute
- `'Pred`, `'Succ` attribute for modular types
- `'Version` and `'Body_Version` attributes
- Limited support of the `'Val` attribute
- Function returning unconstrained array will have wrong `'First` and `'Last`
- User-defined `Storage_Pools`
- Limited support for controlled types
- `'Address` on non-aliased, non-local objects
- `System.Address` comparisons, other than `"=`" and `"/=`"
- Some forms of scalar object renaming (e.g. renaming of dereferenced access value)
- Pragma `Import`, `Export`, and `Convention` other than `Ada` and `Java`
- Pragma `Interrupt_Handler`, `Attach_Handler`
- Asynchronous abort of tasking constructs and tasks
- Access-to-protected-subprogram types
- Incomplete types completed in package bodies
- `Wide_String` and `Wide_Wide_String` (e.g. ACATS test c250001)
- Null arrays with multiple dimensions

Switches not supported:

- `-gnatE` (dynamic elaboration)

Run-time units not supported:

- Ada.Sequential_IO.C_Streams, Ada.Storage_IO, Ada.Text_IO.C_Streams,
Ada.Wide_Text_IO.C_Streams, Ada.Direct_IO.C_Streams
- Ada.Real_Time.Timing_Events
- Ada.Asynchronous_Task_Control
- Ada.Command_Line.Environment
- Ada.Directories
- Ada.Exceptions.Traceback
- Ada.Interrupts,
- Ada.Task_Attributes,
- Ada.Task_Termination,
- Interfaces.C.Extensions, Interfaces.Cobol, Interfaces.C.Pointers, Interfaces.CPP, Inter-
faces.C.Strings, Interfaces.Fortran, Interfaces.Packed_Decimal
- Machine_Code, System.Machine_Code
- GNAT.Alivec
- GNAT.Array_Split
- GNAT.Lock_Files, GNAT.Socket
- GNAT.Exceptions, GNAT.Expect, GNAT.AWK, GNAT.CGI, GNAT.CRC32,
GNAT.MD5, GNAT.SHA1, GNAT.Spibol
- GNAT.Byte_Swapping
- GNAT.Calendar
- GNAT.Command_Line
- GNAT.Compiler_Version
- GNAT.Current_Exception, GNAT.Debug_Pools, GNAT.Debug_Uilities,
GNAT.Exception_Actions, GNAT.Exception_Traces, GNAT.Memory_Dump
- GNAT.Dynamic_Tables
- GNAT.Float_Control
- GNAT.IO
- GNAT.OS_Lib
- GNAT.Perfect_Hash_Generators
- GNAT.Secondary_Stack_Info
- GNAT.Table
- GNAT.Task_Stack_Usage
- GNAT.Time_Stamp
- GNAT.Thread, GNAT.Signal
- GNAT.String_Split, GNAT.Wide_String_Split, GNAT.Wide_Wide_String_Split
GNAT.Traceback

Index

-		-w (jvm2ada)	16
-c (jvm2ada)	9		
-I (jvm2ada)	15	C	
-j (jarmake)	13	Conventions	2
-k (jarmake)	13		
-k (jvm2ada)	15	J	
-L (jarmake)	13	jarmake	13
-L (jvm2ada)	15	jvm2ada	15
-m (jarmake)	13	jvmlist	9
-n (jarmake)	14	jvmstrip	11
-o (jarmake)	14		
-o (jvm2ada)	15	T	
-q (jarmake)	14	Typographical conventions	2
-q (jvm2ada)	9, 10, 11, 15		
-s (jvm2ada)	16		
-v (jarmake)	14		
-v (jvm2ada)	16		

Table of Contents

About This Guide	1
What This Guide Contains	1
What You Should Know Before Reading This Guide	2
Related Information	2
Conventions	2
1 Getting Started with GNAT for the JVM....	3
1.1 Overview	3
1.2 GNAT Tools	3
1.3 Java Development Kits that you can use with GNAT	4
1.4 Compiling Your First Application with GNAT	4
2 Ada & Java Interoperability	7
2.1 Importing Java Services to Ada	7
2.2 Exporting Ada Services to Java	7
3 Viewing Class Files with jvmlist	9
3.1 Running jvmlist	9
3.2 Switches for jvmlist	9
4 Stripping Debug Info with jvmstrip	11
4.1 Running jvmstrip	11
4.2 Switches for jvmstrip	11
5 Building Archives with jarmake	13
5.1 Running jarmake	13
5.2 Switches for jarmake	13
6 Using the Java API with jvm2ada	15
6.1 Running jvm2ada	15
6.2 Switches for jvm2ada	15
6.3 Running jvm2ada on the Java API	16
6.4 Parameter Names and Source Search Paths	17
6.5 Class File Search Paths	17
6.6 Identifier Mangling	18

7	Java-Specific Pragmas	19
7.1	Creating Java Interfaces: Pragma <code>Java_Interface</code>	19
7.2	Using Java Interfaces	21
7.3	The <code>Java_Constructor</code> Pragma	24
7.3.1	Background on Java Constructors	24
7.3.2	Using Java Constructors in Ada	25
7.3.3	Java Constructors and Ada Allocators	27
7.4	Pragma Import Java	28
7.4.1	Importing Packages	28
7.4.2	Importing Exceptions	29
7.4.3	Importing Record Components	29
7.4.4	Importing Dispatching Subprograms	30
7.4.5	Importing Objects	30
7.4.6	Importing Non-Dispatching Subprograms	30
7.5	Pragma Export Java	30
7.5.1	Exporting Objects, Subprograms, and Record Components	31
7.5.2	Exporting Exceptions	31
7.5.3	Exporting Packages or Record Types	32
8	Mapping Java into Ada	33
8.1	Identifiers	33
8.2	Scalar Types	33
8.3	Java References and <code>java.lang.Object</code>	33
8.4	Array Types	35
8.5	The Ada Package <code>Java</code>	37
8.6	Use of Limited-With Clauses by <code>jvm2ada</code>	39
8.7	Java Packages	39
8.8	Java Classes	39
8.9	Abstract Classes	40
8.10	Nested Classes	41
8.11	Java Interface	41
8.12	Java Class Implementing Interfaces	41
8.13	Java Exceptions	41
8.14	Static Fields	41
8.15	Final Static Fields	41
8.16	Instance Fields	41
8.17	Volatile and Transient Fields	41
8.18	Static Methods	42
8.19	Instance Methods	42
8.20	Abstract Methods	42
8.21	Native Methods	42
8.22	Final Classes and Final Methods	42
8.23	Visibility Issues	42
8.24	Java Implicit Upcasting in Ada	43
8.25	Mixing Ada Strings and Java Strings	44
8.26	An Example	45

9	Creating Gnapplets with GNAT	49
9.1	Extending <code>java.applet.Applet.Type</code>	49
9.2	Initializing and Finalizing the GNAT Runtime	50
9.3	Compiling the Gnapplet	50
9.4	Creating the HTML file	51
10	Debugging Ada Programs	53
10.1	Ada Compilation Units and JVM Class Files	53
10.2	Lexical Elements	54
10.3	Enumeration Types	54
10.4	Integer Types	55
10.5	Floating Point Types	55
11	Limitations	57
	Index	59

