

Programming the Internet in Ada 95 (submitted to Ada Europe '96)

S. Tucker Taft

March 15, 1996

Abstract

A new paradigm for computing is emerging based on the Internet and the World Wide Web, accompanied by a new standard programming *platform* based on the Java(tm) technology recently announced by Sun Microsystems [4]. The Java technology includes the definition for a platform-independent *byte code* representation for small applications called *applets*, which allows Java-enabled Web browsers to download and execute these Java applets using a byte code interpreter.

Although the Java byte-code representation was designed specifically for the new Java language, it turns out that the underlying semantic models of Ada 95 and Java are sufficiently similar that a very direct mapping from Ada 95 to Java byte codes is possible. We at Intermetrics are in the process of adapting our Ada 95 compiler front end to generate Java byte codes directly, thereby allowing the development of Java-compatible applets in Ada 95. This paper explains our approach, and provides details on the mapping from Ada 95 features to Java byte codes, as well as the mapping from certain Java features to Ada 95. We have found the combination of the Ada 95 and Java technologies to be very natural and powerful, providing the best characteristics of both technologies, with essentially no loss in functionality.

1 Background on Java(tm)

Periodically in the computer industry, a major shift occurs and the kind of computing being done changes in character. Accompanying such a shift, there is generally the emergence of a new *platform* on which this new computing takes place. In the 70's, the move was to minicomputers and workstations, and the platforms for this were generally Unix and VMS. In the 80's, the move was to PC's, and the predominant platform for this was the IBM PC running

*Intermetrics, Inc., 733 Concord Avenue, Cambridge MA 02138, USA

some variant of DOS or Windows. In the 90's, it looks like the move is to the Internet and the World Wide Web. The platform for this new kind of computing could very well be the Java(tm) technology [4] recently announced by Sun Microsystems.

Sun's Java(tm) technology has three fundamental elements:

- a new object-oriented language called *Java* based loosely on C++, heavily modified to provide a simpler, safer language [6];
- a specification for a Java *virtual machine* (Java VM) with an accompanying byte-code *class file* representation (herein called *J-code*) which is platform independent and is designed for efficient and secure transmission over the Internet for execution on a client machine by an interpreter for the Java VM [3, 7];
- and a set of standard Java classes that support building platform-independent graphical, multitasking, Internet-oriented applications [8].

The Java technology has been demonstrated through the creation of a platform-independent World Wide Web browser called *HotJava* [5], which can download small J-code applications called *applets* automatically when referenced from an HTML Web page, and execute them on the Java VM interpreter running on the client.

The ability to download applets easily and efficiently as part of browsing the World Wide Web has created tremendous excitement in the programming community. Without this capability, use of the Web is limited largely to viewing relatively static pages of information, and making relatively simple queries of a central server. With the applet capability, the Web suddenly emerges as a potentially global client/server system, with functionality, highly interactive graphic user interfaces, animations, etc., moved from a central server to a desktop client, where the excess computing capacity and fast response time reside.

1.1 Java and Ada 95

As it turns out, the particular surface syntax chosen by Sun for Java is not a critical element in the Java phenomenon. Any language which can map to the Java VM can be used as a language for developing Java-compatible applets. In particular, Ada 95 is a remarkably good fit to the Java VM – almost every Ada 95 feature has a very direct mapping using J-code, and almost every capability of J-code is readily represented in some Ada 95 construct. Furthermore, Ada 95 has a number of software-engineering-related advantages relative to Java, such as separation of logical interface from implementation, stronger compile-time type checking, true generic templates, enumeration types, higher-level data-oriented synchronization building blocks (protected types), etc.

At Intermetrics we are in the process of adapting our validated Ada 95 front end, called *AdaMagic(tm)*, to directly generate J-code class files. This paper details the mapping we have chosen between Ada 95 features and the capabilities of the Java virtual machine at the byte-code level, as well as the mechanisms and conventions we use for achieving full interoperability between Ada 95 packages/types and Java packages/classes at the source code level. Examples of translations from Ada 95 to Java byte codes are included. Longer discussions are included on some of the more difficult Ada to Java transformations, such as those for access-to-subprogram types, task entries, and nested subprograms, features which are not directly supported by the Java VM.

In addition to defining the mapping from Ada 95 features to Java, we have also defined a mapping from Java features to Ada 95, so that full interoperability between the languages can be provided. The goal is that any capability of Java is accessible through Ada 95, including Java classes, Java *interface* types, the extension of Java classes with an Ada 95 type, etc., and any Java class implemented in Ada 95 is usable as a regular Java class by Java code. The mapping from Ada 95 to Java is embodied in the translations performed by the Ada to Java compiler. The mapping from Java to Ada is embodied in a tool that reads the output of the Java compiler (a class file), and generates an Ada package spec which may be used to call the operations and reference the data defined in the Java class.

As of this writing, the mapping between Ada 95 and Java is not yet fully implemented; we will identify those places where the mapping is still tentative.

2 Mapping Ada 95 Semantics to the Java Virtual Machine

The Java(tm) Virtual Machine (JVM) is specified in [7]. The instruction set of the JVM is similar to interpretive stack-based byte-code instruction sets used for languages such as Smalltalk [2], and UCSD Pascal [1]. However, the JVM instruction set has additional features to support the special security and efficiency requirements appropriate to its use as a platform-independent representation for automatically downloaded Web applets.

For example, rather than having a type-neutral *load word* or *store word* instruction, the byte-code includes separate operations for loading addresses and loading integers. Similarly, the class file used to hold the byte-code for a single Java class (roughly equivalent to one Ada compilation unit) includes a symbol table that identifies the type of every statically allocated variable, and every *instance variable* of a class. When the byte-code for an applet is down-loaded by a Web browser, this type information is used to drive a data-flow algorithm that verifies the type safety of the applet, ensuring that no disallowed conversions are performed, and that no uninitialized local variables are referenced.

This *strong typing* aspect of Java byte-code makes the job of translating Ada 95 to the byte-code somewhat more difficult. In particular, we must retain high-level type information throughout the translation process, rather than discarding it after performing static semantic checks in the compiler. On the other hand, the careful verification of the byte-code by the Web browser provides an excellent verification tool for the correctness of the translation, and thereby aids the debugging of the Ada to Java compiler.

2.1 Mapping Scalar Types

Java has 8 types that correspond to Ada's numeric and enumeration types: bool, byte, char, short, int, long, float, and double. Bool has a one-byte (8-bit) representation, with values of either false or true. Byte is a one-byte, signed integer type. Char has a two-byte, unsigned representation, and is used to represent Unicode characters (which is the same as ISO 10646 BMP, the code used for *Standard.Wide_Character* in Ada 95). Short is a two-byte, signed integer type. Int is 4 bytes, signed, and Long is 8 bytes, signed. Float and double are 4 and 8 byte IEEE floating point types, respectively.

For 7 of these 8 scalar types there are distinct array-component load and store operations (bool and byte use the same operation). For other operations, there are only 4 sets of op-codes defined, those for 4- and 8-byte integers, and those for 4- and 8-byte floats. In addition, there is a full, separate set of op-codes for loading, storing, and comparing addresses.

Most Java byte-code operations take one or two operands from the run-time stack, and generally produce a value on the top of the stack. The stack is 4-byte-aligned, and each stack object occupies either 4 or 8 bytes. In the Java VM specification, stack objects are said to occupy either one or two *slots*. In addition to a relatively conventional run-time stack, the Java VM includes a set of local variables, each again occupying one or two slots. These can be thought of as stack-resident variables, or alternatively as members of a large register set. There are no operations for taking the address of a local variable, so it is up to the interpreter how to implement local variables.

Besides referencing the stack or a local variable, an instruction may reference a component of an array or a *field* of a class. A field can either be a *static* field, in which case it corresponds to a variable of a library-level Ada package, or it can be a *nonstatic* field, in which case it corresponds to a component of an Ada record type. The symbol table included in each class file identifies the type and kind of each field. The load and store instructions for fields distinguish only between static and non-static; at run-time the interpreter (and byte-code verifier) uses the symbol table to determine the size and type of the field.

The mapping between Ada scalar (sub)types and Java's 8 basic types is relatively straightforward. For enumeration subtypes, we choose the smallest integer type that can represent all values. For integer subtypes, we choose an integer type that is big enough, and if a two-byte type is chosen, char or short is

chosen depending on whether the Ada subtype includes any non-negative values.

Java's numeric semantics are defined to be non-overflowing, with wraparound when overflow would have occurred for integer types, and with IEEE infinity produced when overflow would have occurred for floating point types. This does not present a problem for Ada 95 floating point, because the *Machine_Overflows* attribute allows us to reflect this non-overflowing semantics to the Ada program. However, for Ada 95 signed integer arithmetic, overflow must be signaled rather than wrapping around. Our planned approach for dealing with this is to use 64-bit integer arithmetic to detect overflow on 32-bit operations, and to treat 64-bit integer arithmetic as *non-standard* integer types (as permitted in RM95 3.5.4(26)) with wrap-around semantics.

The Java VM also lacks unsigned comparison operations. For a 32-bit Ada 95 modular type, we plan to use 64-bit comparisons for implementing the Ada 95 ordering operators (the equality operators can use the normal 32-bit operations).

2.2 Mapping for Ada Access Types

The notion of a garbage-collected *heap* is built into the Java VM. There are separate byte-codes for allocating from the heap, one for arrays, and one for instances of a (non-array) class (essentially equivalent to an Ada record). All heap objects are manipulated using references in the Java byte-code. A reference takes up a single 32-bit slot on the stack, though the interpreter could support 64-bit references by a simple on-the-fly transformation of the byte-code as part of byte-code verification.

2.2.1 Access-to-Object Types.

A Java object reference corresponds closely to an Ada access-to-object type. However, there is no operation to create a reference to an object not allocated on the heap. Furthermore, because arrays and class instances are always manipulated via references, all Ada 95 arrays and records must be allocated on the heap. To support Ada 95's notion of an *aliased* object, in addition to allocating arrays and records on the heap, aliased elementary objects are also heap-allocated in our mapping.

Because Java references are only for arrays and class instances, we define *wrapper* classes for holding heap objects of an elementary Ada type. Each wrapper class contains a single field called *Value* of the appropriate Java numeric type, or of the Java type *Object* which is the root of the entire Java class hierarchy. When an access-to-elementary value is dereferenced, we perform a Java *field* selection on its Java reference to retrieve the *wrapped* value. If the wrapped value is an access value, an explicit conversion is performed on the value of class *Object* back down to the type designated by the access value.

2.2.2 Access-to-Subprogram Types.

Java has no direct equivalent to Ada 95's access-to-suprogram values. Our planned mapping is to define a new Java class whenever an Ada access-to-subprogram type is declared, with one method, *Indirect_Call*, which when invoked will call the subprogram designated by a value of the access type. Each place a *'Access* appears in the Ada source program for this access type, we will create an extension of its Java class, with an *Indirect_Call* method that calls the subprogram identified by the name preceding the *'Access*. A conversion between access-to subprogram types will also result in the creation of another extension of the Java class associated with the target access type, which will have a field being of the class associated with the source access type, and an *Indirect_Call* method which will simply invoke the *Indirect_Call* method of this field.

2.3 Mapping for Ada Array Types

The Java VM has direct support for one-dimensional arrays, indexed by integers going from zero to one less than the (run-time) length of the array. Ada has multi-dimensional arrays, and allows the low bound of any dimension to be non-zero. In the mapping, we convert the index or indices into a single zero-based value, after performing appropriate *Index_Checks*. We then use the appropriate Java byte-code to index into the array. The Java interpreter performs a run-time check to make sure the zero-based index is non-negative, and less than the length of the array. Because of this implicit check, we plan to omit explicit *Index_Checks* for one-dimensional Ada 95 arrays.

2.4 Mapping for Ada Record Types

Each distinct Ada 95 record type maps directly to a Java class. If the record type is untagged, then the primitive operations of the type are not handled specially, and just end up as *static* methods of a Java class. A static method is one that does not involve dynamic binding when called, and can be called without having to pass in an instance of the class.

If the record type is tagged, then the primitive operations become non-static *virtual* methods of the Java class associated with the type. The byte-code has two distinct operations for invoking virtual methods, *invoke_virtual* and *invoke_nonvirtual*. *Invoke_virtual* is used for a dynamically bound call, where the run-time class of an object (essentially the Ada 95 *tag*) determines which method table is consulted to find the code body to be executed. *Invoke_nonvirtual* is used for a statically bound call to a non-static method, where the method table is determined by the compile-time class, rather than the run-time class, of the object whose method is being invoked. These two operations correspond respectively to a dispatching call, and a non-dispatching call, on a primitive of an Ada 95 tagged type.

Each Java class, other than the special root class `Object`, has exactly one parent class (Java supports only single inheritance of *implementation*). All fields and methods of the parent class are inherited; the methods may be overridden, and must be if the parent's method is abstract. This corresponds very directly with Ada 95.

In our mapping, if the Ada 95 type is a record extension, then the Java class associated with its parent type becomes the parent class of its own Java class. For Ada 95 record types that are not extensions, we use `Object` as the parent when limited, and the special class `interfaces.java.Nonlimited` as the parent when nonlimited. This special class has a method `Directed_Copy` which is used to implement the Ada record assignment operation at run-time. Java does not support record assignment, only *reference* assignment, so we create a `Directed_Copy` operation for each nonlimited type, and invoke it to perform the component-by-component assignments. The method is called `Directed_Copy` because it takes a parameter, which when non-null and of the appropriate Java class, will be used as the target of the copy. If null or of the wrong class (which can happen with variant records – see below), a new heap object is allocated to hold the copy.

Variant records require special handling, as Java has no direct equivalent. The planned mapping is to define a small class hierarchy corresponding to the variant part of the record type, with each distinct variant being a separate Java class in the hierarchy. The variant part as a whole is represented in the byte-code as a reference to an instance of the root of this class hierarchy. At run-time, this reference would designate an instance of some leaf of the class hierarchy, according to which variant is determined by the discriminant(s) of the record object. When referencing a component of some variant, an explicit conversion to the Java class that includes that component would be performed in the byte-code. The normal Java run-time check associated with such a conversion (*checked cast*) would provide the `Discriminant_Check` required by Ada.

2.5 Mapping for Ada Protected Types

The Java VM has direct support for the notion of a *synchronized* method. Such a method acquires a lock on the associated object automatically when called, and releases the lock automatically upon return. This mechanism can be used quite directly to implement Ada 95's notion of protected subprograms. Each Ada 95 protected type becomes a Java class, an extension of a special class `interfaces.java.Root_Protected`, and each protected subprogram of the type becomes a (nonstatic) synchronized method of the class. `Invoke_nonvirtual` is used to call these methods, and the locking required by Ada 95 happens automatically. At the end of a method for a protected procedure, a call will be inserted on a `Scan_Queue` method of `Root_Protected`, if the protected type has any entries (see below).

Protected entries require a more complicated mapping. Our planned ap-

proach is to define a special class *Entry_Parameters*, and define an extension of it for each Ada entry defined, to hold the parameters passed to the entry, and with methods *Entry_Barrier* and *Entry_Body* to represent the entry barrier expression, and the code of the entry body, respectively. When an entry is called, an instance of this extension of *Entry_Parameters* will be created, and passed along with an identification of the target protected object to a (synchronized) *Entry_Call* method of the class *Root_Protected*.

The *Entry_Call* method would *invoke_virtual* the *Entry_Barrier* method to determine whether the entry was open, and if not, would place the *Entry_Parameters* instance on an appropriate entry queue. If the *Entry_Barrier* method returns True, then *Entry_Call* would immediately *invoke_virtual* the *Entry_Body* method to perform the entry body, and would then call *Scan_Queues* to check to see if other entry barriers have become true.

This approach is essentially the same as that used by our more *conventional* Ada run-time system, but we are using the Java class and virtual method concepts rather than explicit type descriptors and pointers to procedures.

2.6 Mapping for Ada Task Types

In our normal AdaMagic run-time model, we represent a task object as a reference to a *Task Control Block* record containing a protected object, and invoking a task entry becomes an invocation of an entry of this embedded protected object. The Task Control Block is itself an extension of an underlying Thread type.

Our mapping in Java will be essentially the same. A Task Control Block (TCB) will be an extension of the Java class `java.lang.Thread`, and will contain an object that is an instance of an extension of the *Root_Protected* class, *Protected_TCB*. The *Entry_Barrier* and *Entry_Body* methods for *Protected_TCB* will implement the appropriate semantics for task entries, checking on the state of the task to determine whether a given entry is open, and releasing the task to execute the appropriate accept statement when *Entry_Body* is invoked.

2.7 Mapping for Exceptions

Java supports the notion of exceptions. However in Java, any object of a class descended from the special class *Throwable* may be thrown. For Ada, each user-defined exception becomes a separate Java class, descended from the special class *interfaces.java.Ada_Exception*. Java in general requires that a method declare what exceptions it can *throw* in a *throws clause*, and the compiler checks statically that the promise given in the throws-clause is not violated (note that this is different from C++; in C++, throws clauses are checked at run-time, which makes them generally less useful). However, exceptions that are descended from *java.lang.Error* or *java.lang.Runtime_Exception* need not be included in

a throws clause. We have chosen to make *interfaces.java.Ada_Exception* a descendant of *java.lang.Runtime_Exception* so that we won't violate Java's rules for throws clauses, given that Ada subprograms do not include anything analagous to throws clauses.

Ada has four distinct predefined exceptions, *Constraint_Error*, *Storage_Error*, *Program_Error*, and *Tasking_Error*. We handle *Tasking_Error* in the same way as a normal user-defined exception. The other three predefined exceptions are handled specially, since we rely on the Java interpreter to implement some of the checks that Ada associates with these exceptions. In particular, we associate *Constraint_Error* with the entire *java.lang.Runtime_Exception* class hierarchy, *Storage_Error* with the hierarchy rooted at *java.lang.VirtualMachine_Error*, and *Program_Error* with *java.lang.Linkage_Error*. When one of these predefined exceptions is listed in an exception handler, we generate a Java *catch* table entry for the entire associated hierarchy. Because *interfaces.java.Ada_Exception* is itself inside the *java.lang.Runtime_Exception* hierarchy, we make a special check in a handler for *Constraint_Error* to see if the exception is in *interfaces.java.Ada_Exception*, and if so, re-throw it.

A handler for *others* becomes a Java *catch* for *java.lang.Throwable*. However, Java uses the exception *java.lang.ThreadDeath* to signal task abort, so the generated *catch* handler starts with a check for *ThreadDeath*, which is re-thrown if detected.

2.8 Mapping for Ada packages

Each Ada package becomes two Java classes, one containing the entities declared in the package spec, and the other containing the entities declared in the package body. This separation is necessary because the elaboration of a package spec and a package body need not happen in immediate sequence. The elaboration of a package spec or body is generated as the *class init* method ("clinit") for the associated Java class, and so we need two classes to have two independently executable class init operations.

Note that in the presence of the pragma *Elaborate_Body* it would be possible to use a single Java class, because this pragma ensures that the package body is elaborated immediately after the spec.

The objects declared in a package become static fields of the Java class associated with the package. The subprograms declared in the package become static methods of the class, unless they are primitive operations of a tagged type. In the latter case, they become virtual, nonstatic methods of the class associated with the tagged type.

When a record type (tagged or untagged) is declared immediately inside a package spec, the normal mapping will produce separate classes for the package spec and the record type. However, to better match the normal Java combination of type and module into a single class, the compiler recognizes a special naming convention for record types declared inside a package – if the record

type name is the same as that of the package, or is the same with the suffix *_Obj* or *_Rec*, then the record type and the package spec are mapped to a single Java class (based on the package name). The primitive operations of the record type, if tagged, become the nonstatic methods of this class. The other subprograms become the static methods. The package-level variables become the static fields, while the components of the record type become the nonstatic fields.

This naming convention is used automatically by the tool that generates an Ada package given a Java class file.

2.9 Mapping for Nested Subprograms

As indicated above, a subprogram declared in a library-level package maps directly to a Java method, nonstatic or static according to whether or not the subprogram is a dispatching operation. Non-library-level subprograms (subprograms declared inside other subprograms or tasks) pose an additional challenge, because Java has no equivalent to nesting of subprograms inside other subprograms. The solution we have adopted is to create a Java *frame* class upon encountering a nested subprogram. The frame class is used to hold all of the variables of a subprogram that are referenced from any subprogram nested inside of it. The nested subprograms themselves become nonstatic methods of this frame class, and as such, receive as their implicit *this* parameter a reference to an instance of this class. This mechanism is essentially an implementation of the concept of *static links* or a *static chain*, a common method for implementing up-level references.

3 Mapping Java Features to Ada 95

In general, the mapping of features between Ada 95 and Java is one-to-one, and so the mapping of Java features to Ada 95 is implied by the mapping of Ada 95 features to Java. However, there are certain features of Java that have no direct analog in Ada 95, and for these we have endeavored to define mappings that allow Ada programmers to use the full power of the Java virtual machine.

3.1 Mapping Java Constructors to Ada 95

As with certain other object-oriented programming languages, Java has a special kind of operation called a *constructor* which is used to create objects, optionally with some number of parameters to control the construction. In Ada 95, there is no special notion of a constructor, other than aggregates which are only of useful for visible types or visible extensions, and the underlying Ada support for user-defined default initialization. For parameterized construction of an object of a private type or private extension in Ada 95, a normal function or procedure is used.

A Java constructor is invoked by means of the *new* operation, and is defined by defining an operation whose name is the same as that of the enclosing class. Within the definition of a Java constructor, the object being constructed is referenced using the normal implicit *this* parameter. A Java constructor for a given class (other than the root class `java.lang.Object`) always starts, either explicitly or implicitly, with a call on some other constructor, either for the parent (*super*) class, or for the same class.

To define a Java constructor in Ada 95, a function or procedure is defined with its *convention* specified as *Java_Constructor*. In Ada 95, every subprogram has a *convention*, specifiable with a pragma `Import`, `Export`, or `Convention`, which the programmer can use to specify how the subprogram is to be called or implemented by the compiler.

For our Ada to J-code compiler, we have defined a number of conventions to control the mapping from Ada 95 to Java VM features. The convention *Java_Constructor* specifies that the subprogram should become a constructor in the byte code. For Ada functions with the convention *Java_Constructor*, a call on the Ada function implicitly invokes the J-code *new* operation, which actually performs heap allocation, and the result of the new operation is passed in as an additional parameter to the function. Ada procedures with a convention *Java_Constructor* are for use when defining other constructors; as mentioned above, a constructor must start with a call on another constructor, passing it a reference to the object being constructed.

This mapping for Java constructors is best illustrated by example. To define a Java constructor in Ada, the following Ada code would be used:

```
type Widget_Obj is new Frame_Obj with record
    size : Integer;
end record;
type Widget_Ptr is access all Widget_Obj'Class;

function new_Widget(
    size : Integer;
    this : Widget_Ptr := null)
    return Widget_Ptr;
pragma Convention(Java_Constructor, new_Widget);

...

function new_Widget(
    size : Integer;
    this : Widget_Ptr := null)
return Widget_Ptr is
    Result : Widget_Ptr :=
        Widget_Ptr(new_Frame(Frame_Ptr(this)));
```

```

begin
    Result.size := size;
    return Result;
end new_Widget;

```

The *new_Widget* constructor starts with a call on the *new_Frame* constructor, passing it only the *this* parameter. This is the idiom used to conform to the Java requirement that every constructor starts with a call on some other constructor. The *new_Frame* constructor is presumed to return its final parameter as its result.

For normal usage, an Ada constructor function is called without providing the final, defaulted parameter. For example, this constructor would be used with a call such as:

```

X : Widget_Ptr := new_Widget(42);

```

In this case, the last parameter is allowed to default, which is the idiom to have the compiler implicitly invoke the *new* byte-code for the *Widget* class and pass the result as the final *this* parameter. This constructor could also be called from another constructor, with the *this* parameter specified explicitly, analogous to the call on *new_Frame* inside *new_Widget*, above.

3.2 Mapping Java Interface Types to Ada 95

Java supports a limited form of multiple inheritance based on *interface* types. A Java class must specify only one immediate parent/super class, but it may declare that it *implements* one or more interface types. No method or field definitions are inherited from an interface type. Instead, when a class claims to implement an interface type, it is required to implement all of the methods declared in the definition of the interface type. A reference to a class that implements an interface may be converted to a reference to that interface. Given a reference to some interface, a call on any of the methods specified in the interface type's definition may be performed. A special byte-code, *invoke_interface*, is used for such a call. It results in a *very* dynamic binding to the implementation of the method, using the run-time class of the object designated by the reference to determine which particular implementation of the method to execute.

Ada 95 has no direct equivalent to Java's interface type. However, much the same capability could be built by adding a component of the interface type to a type that claims to implement the type, with the primitive operations of the interface type simply forwarding the call to the corresponding operation of the enclosing type, using an access discriminant. To convert from a pointer to an object of the enclosing type to a pointer to an object of the interface type, one would simply use *'Access* on the nested component of the interface type. For example, here is a way one might create an extension of an *Applet_Obj* type that

provides the *run* operation required of all types that implement the *Runnable* interface type:

```
type Active_Obj;

type Runnable_Obj(encloser : access Active_Obj'Class) is
  new java.lang.Runnable.Runnable_Obj with null record;

type Active_Obj is new Applet_Obj
  with record
    Runnable : aliased Runnable_Obj(Active_Obj'Access);
    -- Active_Obj implements Runnable
    ...
  end record;
procedure run(this : access Active_Obj);
procedure run(this : access Runnable_Obj);
...
procedure run(this : access Runnable_Obj) is
begin
  -- Just forward the call to the
  -- "run" operation of Active_Obj
  run(this.encloser)
end run;
```

To simplify this idiom, we have defined the convention *Java_Interface* which when specified on a type, indicates that all primitive operations of that type are assumed to forward any calls through the corresponding operations of whatever type they are enclosed in. Hence, the above example can be simplified to the following:

```
-- First, we add the special convention to the Ada definition
-- for java.lang.Runnable.
package java.lang.Runnable is
  type Runnable_Obj is tagged limited null record;
  pragma Convention(Java_Interface, Runnable_Obj);
  type Runnable_Ptr is access all Runnable_Obj'Class;

  procedure run(this : access Runnable_Obj);
  pragma Import(Java, run);
end java.lang.Runnable;

-- Now we can indicate that a type implements java.lang.Runnable
-- by the following simpler idiom:
with java.lang.Runnable; use java.lang.Runnable;
package ... is
```

```

type Active_Obj is new Applet_Obj
  with record
    Runnable : aliased Runnable_Obj;
    -- Active_Obj implements Runnable
    ...
  end record;
type Active_Ptr is access all Active_Obj'Class;

procedure run(this : access Active_Obj);
pragma Convention(Java, run);
...
end ...;

```

When the Ada/Java compiler encounters such a use of a component having a type with convention *Java_Interface*, it automatically adds to the generated class file for the enclosing type an entry indicating that it *implements* the class associated with the enclosed *Java_Interface* type.

When the Ada programmer wants to pass a pointer *X* designating a type like *Active_Obj* to an operation that requires a *Runnable* pointer, the programmer writes *X.Runnable'Access* instead of simply *X*. For example, presuming there is a *new_Thread* constructor that takes a parameter of type *Runnable_Ptr*, the following would work given *X* of type *Active_Ptr*:

```
My_Thread : Thread_Ptr := new_Thread(X.Runnable'Access);
```

4 An Example of the Mapping from Ada 95 to J-code

We have implemented a number of applets in Ada 95. Here is an example of the Ada 95 code and the generated J-code for a relatively small function that handles the *MouseDown* event. First, here is the Ada 95 for the *MouseDown* function, with a bit of the enclosing package:

```

with java.applet.Applet; use java.applet.Applet;
with java.awt.Event; use java.awt.Event;
...
package LifeRect is
  type LifeRect_Obj is new Applet_Obj with private;

  function mouseDown(L : access LifeRect_Obj;
    evt : Event_Ptr; X, Y : Integer)
    return Boolean;
...

```

```

private

    type LifeRect_Obj is new Applet_Obj with record

        Runnable : aliased Runnable_Obj;
        -- means LifeRect "implements" Runnable

        ThreadDead : boolean := false;
        -- Whether the thread has been paused by the user.
        ...
    end record;

end LifeRect;

with java.io.PrintStream; use java.io.PrintStream;
with java.lang.System; use java.lang.System;
...
package body LifeRect is

    -- Pause the thread when the user clicks the mouse.
    function mouseDown(L : access LifeRect_Obj;
        evt : Event_Ptr; X, Y : Integer)
    return Boolean is
    begin
        println(stdout, "mouse down");
        if L.ThreadDead then
            start(L);
        else
            stop(L);
        end if;
        L.ThreadDead := not L.ThreadDead;
        return True;
    end mouseDown;
    ...

end LifeRect;

```

The above subprogram becomes a virtual method of the java class LifeRect. The following byte codes for this method were generated, as part of the LifeRect.class file produced by the compiler:

```

Method 5: Name: mouseDown    Signature: (Ljava/awt/Event;II)Z
Flags: ACC_PUBLIC
Attributes (1):
Code: MaxStack:3, MaxLocals:5, Code Length:78

```

```

0:  getstatic #200
    <Field LifeRect_.mouseDown__2__elab_bool Z>
3:  ifne 14 (offset 11)
6:  new #70 <java/lang/ClassCircularityError>
9:  dup
10: invokevirtual #73
    <Method java/lang/ClassCircularityError.<init>()V>
13: athrow
14: getstatic #91
    <Field java/lang/System.out Ljava/io/PrintStream;>
17: dup
18: ifnonnull 29 (offset 11)
21: new #93 <java/lang/NullPointerException>
24: dup
25: invokevirtual #95
    <Method java/lang/NullPointerException.<init>()V>
28: athrow
29: astore 4
31: aload 4
33: checkcast #97 <java/io/PrintStream>
36: getstatic #56 <Field LifeRect.temp_obj__8 [C>
39: invokevirtual #101
    <Method java/io/PrintStream.println([C)V>
42: aload_0
43: getfield #81 <Field LifeRect.ThreadDead Z>
46: ifeq 56 (offset 10)
49: aload_0
50: invokevirtual #107 <Method LifeRect.start()V>
53: goto 60 (offset 7)
56: aload_0
57: invokevirtual #203 <Method LifeRect.stop()V>
60: aload_0
61: aload_0
62: getfield #81 <Field LifeRect.ThreadDead Z>
65: ifne 72 (offset 7)
68: iconst_1
69: goto 73 (offset 4)
72: iconst_0
73: putfield #81 <Field LifeRect.ThreadDead Z>
76: iconst_1
77: ireturn

```

Code attributes (1):

Line number table (7): (Start,Line)

(14,527) (42,529) (49,530) (56,532) (60,534)
(76,535) (78,536)
End of code attributes.

5 Conclusion

We have designed a full mapping between Ada 95 and Java, and as of this writing, have implemented all but those aspects requiring significant out-of-line code (such as record assignment and protected types). The process of defining this mapping has reinforced our early impression that Java and Ada 95 are semantically very compatible. The combination of the two is remarkably seamless, and loses essentially nothing while gaining the best aspects of both languages. Ada 95 provides significantly more compile-time checking, as well as a number of nice compile-time features such as enumeration types, generic templates, and user-defined operators, while Java provides automatic garbage-collected storage-management, platform independence of both byte-code and its rich class library, and integration with the World Wide Web.

As a final anecdote, we have already found the combination of the two technologies to be uniquely productive. We have been able to develop applets in Ada 95 that are just as functional as corresponding Java applets, and that take full advantage of the standard Java library. Furthermore, with one applet that we translated, after implementing full constraint checking in the Ada to Java compiler, we immediately began to reap the benefits of Ada's notion of range checks. The first *Constraint_Error* that was raised identified a small error in the hand translation from the original Java applet into Ada. After fixing that error, the next *Constraint_Error* raised identified a deep logic error in the original Java applet, which ultimately turned out to be due to the randomization mechanism used to initialize the game.

Our early experience has confirmed our belief that the combination of Ada 95 and Java is an excellent technology for programming the Internet, benefiting from the rapidly growing commercial support for Java, and the excellent software engineering advantages of Ada.

References

- [1] Bowles, Kenneth L.: UCSD Pascal. *Byte Magazine* (May 1978)
- [2] Goldberg, A., Robson, D.: *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, Reading, MA (1983)
- [3] Gosling, James: Java Intermediate Bytecodes. *ACM SIGPLAN Workshop on Intermediate Representations (IR '95)*, *ACM SIGPLAN Notices* (March 1995)

- [4] Sun Microsystems: *The Java(tm) Language Environment - A White Paper*. <http://www.javasoft.com/whitePaper/java-whitepaper-1.html> (1996)
- [5] Sun Microsystems: *The HotJava(tm) Browser - A White Paper*. <http://www.javasoft.com/HotJava/overview/> (1995)
- [6] Sun Microsystems: *The Java Language Specification*. <ftp://ftp.javasoft.com/docs/javaspec.ps> (1996)
- [7] Sun Microsystems: *The Java Virtual Machine Specification*. <http://www.javasoft.com/doc/vmspec/html/vmspec-1.html> (1996)
- [8] Sun Microsystems: *Java API Documentation*. <http://www.javasoft.com/JDK-1.0/api/packages.html> (1996)