Multilanguage Programming on the JVM: The Ada 95 Benefits

Franco Gasperoni
gasperon@act-europe.fr
ACT Europe
www.act-europe.com

Gary Dismukes
dismukes@gnat.com
Ada Core Technologies
www.gnat.com

Abstract

The latest trend in our industry, "pervasive computing", predicts the proliferation of numerous, often invisible, computing devices embedded in consumer appliances connected to the ubiquitous Internet. Secure, reliable applications combined with simplicity of use will make or break a company's reputation in this market.

The Java "write once, run anywhere" paradigm, introduced by Sun in the mid-90s, is embodied in a widely available computing platform targeting pervasive devices. Although the Java Virtual Machine was designed to support the semantics of the Java programming language, it can also be used as a target for other languages.

The Ada 95 programming language is a good match for the Java platform from the standpoint of its portability, security, reliability, and rich feature set. In this article we explain the features that have made Ada the language of choice for software-critical applications and how these features complement the Java programming language while increasing the overall reliability and flexibility of the Java platform.

1 Introduction

1.1 Reliability Matters

The latest trend in our industry, "pervasive computing", promises the proliferation of numerous, often invisible, computing devices embedded in consumer appliances connected to the ubiquitous Internet. These devices include smart TVs, set-top boxes, satellite broadcast receivers, game consoles, home networking systems, web-phones, web terminals, personal digital assistants, and automobile entertainment systems (see for instance [1],[2]).

According to a recent International Data study, information appliances will outsell consumer PCs by 2002 in the United States. By 2004, the global market for information appliances will surpass USD 17.8 billion, or 80 million units. This is just the tip of the iceberg. The business opportunity that pervasive computing represents dwarfs previous opportunities by an order of magnitude.

For users, pervasive computing promises simplicity of use, ubiquitous access, and reliability. While improvements in hardware technologies and the serendipitous appearance of the Internet have made pervasive computing possible, it is software that will control whether or not this potential will be realized. Why? Because today software is the reliability bottleneck.

The wider the customer base, the bigger the demands on device reliability, as users will be less technology savvy and hence less tolerant of bugs, viruses, and reboots. The surprising habit of using one's customer base as a giant beta-testing site after shipping the first release of a product is not acceptable in the realm of pervasive computing devices. The ability to build secure, reliable software will make or break a company's reputation in this market.

1.2 Java: A Platform for Pervasive Computing

The Java "write once, run anywhere" paradigm, introduced by Sun in the mid-90s, is a widely available computing platform targeting pervasive devices [3].

Java is really three things: a relatively simple object-oriented programming language, a virtual machine (JVM), and an extensive, ever-growing set of APIs with services ranging from mathematical functions to telephony, 2D graphics, and speech recognition.

Concerns about application security have found their way into the design of both the Java programming language and the JVM. Java's security features ensure that code running on the JVM cannot harm, crash, or compromise your system's integrity.

The JVM is a stack-based virtual machine which includes late binding, garbage collection, object-oriented features, exception handling, and synchronization capabilities. While the JVM does not directly provide threads, the Java API does. It is therefore possible to create multi-threaded applications for the Java platform.

The excitement about Java came from the ability to run Java applications anywhere that the JVM and the native code in the Java API have been ported. Up to now, portability has not been a strong point of the embedded systems industry. With a large array of evolving microprocessor architectures, Java is the first glimpse at addressing the portability problem, which is an important issue for the emerging pervasive computing industry.

1.3 Ada on the Java Platform: Reliability Benefits

Although the JVM was designed to support Java semantics, it can also be used as a target for other languages. This can provide substantial benefits, allowing Java and non-Java components to cohabit and communicate without incurring the complexity and overhead of the JNI (Java Native Interface) or the nonportability of native methods. Indeed, several compilers for C and C++ target the JVM; however, these languages' intrinsic insecurities, and their semantic mismatch with Java, require the programmer to adhere to restrictive feature subsets (see for instance http://grunge.cs.tu-berlin.de/vmlanguages.html).

The Ada 95 programming language is a good match for the Java platform, from the standpoint of portability, security, reliability, and features. As a matter of fact, several years ago a prototype Ada 95 compiler ("AppletMagic") was released which generates JVM bytecodes [4]. Recently, JGNAT, the GNU Ada 95 development environment targeting the Java Platform, was announced by ACT Europe and Ada Core Technologies [10].

JGNAT comprises a compiler that generates Java bytecodes for JVMs conforming to Sun's JDK 1.1 and above, and an interfacing tool that gives an Ada program transparent access to the complete Java API (and in fact to any set of class files). With JGNAT there is no need for the programmer to write platform-specific "glue" code; the class files generated by JGNAT are fully interoperable with the Java world and Java-enabled browsers.

The Ada 95 programming language is a standardized (ISO/ANSI) object-oriented programming language that is upwardly compatible with its predecessor Ada 83. Ada 83 was originally designed for large-scale and safety-critical applications (especially embedded applications). Over the years it has built up an impressive track record in this field. As an example, Ada was used in the GPS system of the Hertz Rent-A-Car navigation system.

Ada 95 extends its Ada 83 foundations and provides a unique combination of object-oriented, real-time, concurrency, and distributed programming features, leveraging on the fundamental distinction between interface and implementation introduced by Ada 83.

1.4 Security $\neq$ Reliability

Although related, security and reliability are two distinct concepts. While security ensures that erroneous or malicious code cannot compromise a system's integrity, reliability deals with the design and evolution of applications that produce correct results. Thus a programming language can be a good language for implementing secure systems, while not being suitable for designing reliable software.

As an example, consider Java's choice of wrap-around semantics for its integer types. This choice is secure because an overflow will not cause the JVM to crash, but it is unfortunate from the reliability standpoint as will be shown in the examples below.

In this article we explain the features that have made Ada the language of choice for software-critical applications and show how these features complement the Java programming language while increasing the overall reliability and flexibility of the Java platform. Rather than provide an exhaustive treatment of the subject, we support this thesis using a number of short Java and Ada examples.

We hope these examples will inspire the reader to learn more about Ada 95 (see for instance [5], [6] or [12]) and identify the areas where the synergy between Ada and the Java platform will result in the level of reliability required by forthcoming intelligent appliances and pervasive computing systems.

2 The Devil is in the Details

A program is written once, but it will be read many times by many different developers: for reviews, corrections, and improvements. Today more than ever, companies must be in a position to easily extend and enhance their software. Better readability also helps catch errors earlier in the development cycle and hence improves code reliability.

Ada was designed to emphasize readability and addresses this issue both at the lexical and syntactic levels. Java, unfortunately builds on the the C lexical and syntactical foundations and inherits a number of C's readability problems. The following sections present examples illustrating some readability problems in Java and show the solutions provided by Ada.

2.1 Forgot Something?

Java, like C++ allows both the `/* ... */` and the `//` style comments. While the second form is safe, the first one can lead to unpleasant surprises as shown in the following code.

```
/* Some interesting comment.
   More text detailing the
   interesting comment.

   x = 0;  /* another comment */
```

Did the original programmer intend to comment out `x = 0;`? To avoid the above problem, Ada only provides the single-line form (comments in Ada start with `--`). The potential for cut-and-paste errors as well as general problems with readability when a comment block spans portions of text that might include code fragments argues strongly against the bracket form of comment.

2.2 Integral Surprise

Consider the following segment of code

```
int var = 8262;
int far = 0252;
int bar = 9292;
```

What's wrong? Maybe nothing. But how can you be sure that the original developer meant `far` to be `170` rather than `252`?

Java inherits from C integer literals and as such, in addition to decimal constants it offers hexadecimal and octal constants (octal constants start with a zero). Thus `0252` and `252` mean very different things.

In Ada `0252` and `252` mean the same thing, leading zeros are ignored. If the developer meant `far` to be initialized to the octal number `252` he would write:

```
far : integer := 8#252#;
```

Note incidentally that in Ada you can write any integer (and floating point) constant in any base from `2` to `16` and can use the underscore to separate digits to improve readability. For instance:

```
far : integer := 2#1010_1010#;
rel : float   := 2#1.1111_1111_1110#E11  --  floating constant 4095.0
flt : float   := 16#F.FF#E+2             --  floating constant 4095.0
```

These are small things, but they underscore Ada's philosophy of favoring readability.

2.3 Trick or Treat?

While C's side effects of `=` in boolean expressions have been limited by Java semantic rules, some remnant side effects are still lurking. Consider for instance the following chunk of code:

```
boolean condition_1 = ...;
boolean condition_2 = ...;
...
if (condition_1 = condition_2) {
   ...
}
```

Is there a problem? Was the originator of this code a Pascal, Delphi, or Basic aficionado who confused `=` with `==`, or did he actually mean to assign `condition_2` to `condition_1` and take the `if` branch if `condition_2` is true? You cannot know without studying the logic of the code in detail.

In Ada, like Pascal, from which Ada borrowed much of its syntactic flavor, `=` means equality testing and `:=` means assignment and assignments do not yield a value as they do in C or Java, which avoids all potential confusions between equality and assignment.

2.4 The Lure of Cut and Paste

As in C, Java allows programmers to ignore the value returned by functions. This can give rise to some interesting surprises, as the following code excerpt illustrates.

```
static int function_returning_an_int (int x) {...}

static void f () {
    int some_variable;
    int another_variable;
    ...
    another_variable = some_variable--;function_returning_an_int (1);
```

Did the author mean the above or did he intend to write something else, like

```
some_variable--;
another_variable = function_returning_an_int (1);
```

or

```
another_variable = some_variable - function_returning_an_int (1);
```

In Ada, the value returned by a functions cannot be ignored, if it is the compiler will issue a compile-time error. If the developer really intends to discard the value it can introduce a local temporary as shown below:

```
ignore : integer := function_returning_an_int (1);
```

2.5 Dangle Here, Dangle There, Dangle Everywhere

Possibilities for typos and cut-and-paste errors abound with C's penchant for trusting the programmer when it comes to lexical and syntactic issues. A now-traditional example of a possible typo is given below:

```
if (...); {
   ...
}
```

Is the `;` after the `if` intended ? Another problem that can occur with `if` (as well as `while` or `for` statements) is

```
if (...)
   x = 1;
   y = 2;
```

Here, because of the indentation of the `y = 2;` statement it is likely that the programmer really meant

```
if (...) {
   x = 1;
   y = 2;
}
```

In Ada if you write

```
if (...); then
   ...
end if;
```

or

```
if (...) then
   x := 1;
   y := 2;
-- no end if;
```

this will result in a compile-time error. The developer has to write what she means unambiguosly. Another reliability pitfall is the dangling-else problem shown below:

```
public static float sum_positive (float [] values, float default) {
   float sum = 0.0;
   if (values.length > 0)
       for (int i = 0; i < values.length; i++)
           if (values [i] > 0.0)
              sum += values [i];
   else
       sum = default;
   return sum;
}
```

The `else` is actually bound to the second `if` not the first one.

These C pitfalls are addressed in Ada by systematically requiring proper bracketing for `if`, and `loop` statements, by using syntactic closers for all statement constructs that allow nesting. For instance, the above procedure would be written in Ada as

```
type Vector is array (integer range <>) of float;
--  Defines an array type whose component type is float, whose
--  index type is integer, and whose bounds are not fixed and
--  can vary between different instances of type Vector.
...
function sum_positive (values : Vector; default : float) return float is
   sum : float := 0.0;
begin
   if (values'length > 0) then
       for i in values'range loop
           if (values (i) > 0.0) then
              sum := sum + values (i);
           end if;
       end loop;
   else
       sum := default
   end if;
   return sum;
end sum_positive;
```

As a side note, because arrays can have arbitrary bounds, Ada complements the `length` array attribute provided in Java with several other array attributes (`first` yields the index of the first array entry, while `range` yields the range between the first and last index values in the array).

3 Blade Runner

This section takes a look at some issues that arise at run time when executing a Java application.

3.1 Wraparound Semantics

Java has introduced a number of very beneficial safety checks that are carried out at run time to help ensure software quality. These checks are equivalent to (and have been inspired by) the ones offered by Ada and include: array bounds checking, integer division by zero, null-pointer dereferencing, etc.

When it comes to overflow semantics for integers, Java has chosen wrap-around semantics. This means that in the following code

```
byte b = 127;
b++;
```

`b++` will yield a value of `-128` for `b`. An identical problem will occur for other Java 16-, 32-, and 64-bit integer types. This means that `x+1` is not necessarily greater than `x`.

To avoid undetected overflow situations Ada provides overflow checks. Thus the following code

```
x : integer := integer'last;
--  Initialize x to the biggest integer
y : integer := x + 1;
```

will raise `Constraint_Error`. The issue that was once raised against overflow checking was a concern for efficiency. This criticism is now obsolete. For one thing it is always possible to disable overflow checks in specific regions of code or the overall program. However, a better answer is that this is not necessary since modern computer hardware makes these checks very inexpensive (because the likelihood of an overflow occurring is low) and hence speculative execution techniques actually hide the cost of the untaken branch. In addition, modern compiler optimization technology is capable of eliminating a large number of these checks.

3.2 Ad Eternum

Wrap-around semantics for integers has an embarrassing side effect. Consider the following code:

```
public static void send_bytes_to_port (byte first, byte last) {
   for (byte b=first; b <= last; b++) {
       ...  // send b to port
   }
```

What can go wrong? The above `for` loop can loop forever. (Hint: what happens if `last==127` ?). The same problem can occur with all integer types.

Ada addresses this problem by having a `for` loop which is a first-class citizen (not just a shorthand for a `while` loop). The above code would be written in Ada as:

```
type byte is range -128 .. 127;
--  In Ada you can declare scalar types. More on this later.
--  The above should probably be declared as an unsigned integer.
--  More on this later.

procedure send_bytes_to_port (first : byte; last : byte) is
begin
   for b in first .. last loop
       ...  --  will never loop for ever
   end loop;
end send_bytes_to_port;
```

What should be noted here is that the loop variable `b` is implicitly declared by the `for` loop construct. It takes its type from the bounds and the loop is executed from `first` to `last` without any infinite loop problems.

3.3 Please Elaborate

The process of executing initialization code before the `main` program starts is known as elaboration in Ada. In C, the initialization of data prior to program execution is restricted to initializing global variables with static values. C++ allows general expressions to be used to initialize global variables but does not address the issue of the order in which such initialization have to be carried out (which gives rise to some interesting surprises).

Java allows `static` fields to be initialized with arbitrary expressions and allows the embedding of arbitrary elaboration code in the form of `static { ... }` statements between the methods of a class. The rules regarding the order in which intra-class initialization have to occur are well defined. However, because of the dynamic nature and data-driven nature of class loading, the ordering of initialization among classes depends on the order in which these classes are dynamically loaded, which in turn depends on the underlying input data and hence is not necessarily deterministic. There are two problems with this approach:

1. If you're not careful you may pick up a default initialization versus an explicit initialization on referencing a variable.

2. The rules only really make sense in an interpreted environment. If you are compiling Java, there is no way to know what the correct order should be without generating interpretive code.

This means that the programmer needs to worry about these ordering problems himself, as the following examples shows.

```
public class A {
    static { System.out.println ("A begin"); }
    public static int x = B.g();
    public static int f () {
       System.out.println ("A.f()");
       return A.x + 1;
    }
    static { System.out.println ("A end"); }
}
public class B {
    static { System.out.println ("B begin"); }
    public static int z = 99;
    public static int y = A.f();
    public static int g () {
       System.out.println ("B.g()");
       return B.y + 1;
    }
    static { System.out.println ("B end"); }
}
public class C {
    public static void main (String [] args) {
       int i;
```

```
            if (args.length > 0)
                i = B.z;
            System.out.println ("A.x = " + A.x);
            System.out.println ("B.y = " + B.y);
        }
    }
```

If the program is invoked without parameters, the printed output is:

```
A begin
B begin
A.f()
B end
B.g()
A end
A.x = 2
B.y = 1
```

otherwise, if the above program is invoked with parameters, the output is:

```
B begin
A begin
B.g()
A end
A.f()
B end
A.x = 1
B.y = 2
```

Needless to say, reliability is not improved by Java's semantic choices in this domain.

Ada 95 is designed to be a safe language, and a programmer-beware approach is clearly not sufficient. Consequently, the language provides three lines of defense:

1. Standard rules (that we omit for brevity's sake).
2. Dynamic elaboration checks. Dynamic checks are made at run time, so that if some entity is accessed before it is elaborated (typically by means of a subprogram call) then the exception `Program_Error` is raised.
3. Elaboration control. Facilities are provided for the programmer to specify the desired order of elaboration.

If the Java example given above is translated into Ada, the exception `Program_Error` will be raised at execution time signaling a circularity. In addition, the JGNAT compilation system will emit a warning at bind time indicating the circularity.

4 Gone with the Wind

Is an application whose initial release works without faults reliable? Only if its quality does not deteriorate whith subsequent changes, upgrades, and modifications. Given the importance of software reuse it is important that a programming language be engineered to reduce programmer mistakes during both initial development and incremental enhancement.

The following section illustrates possible oversights that could occur when maintaining software written in Java.

4.1 Overloading Confusions

Although it restricts the set of implicit conversions offered by C and C++, Java does allow some cases of implicit conversion. Unfortunately the combination of implicit conversions and overloading can result in nasty errors as illustrated in the following example.

```
public class Try {
    static void p (float f) { System.out.println ("p (float)"); }
}
public class Client {
    public static void main (String args[]) { Try.p (1); }
}
```

The above innocuous code will print `p (float)` when run. Assume that some time after class `Try` has been written another method `p` needs to be added to class `Try`. as shown below.

```
public class Try {
    static void p (float f) { System.out.println ("p (float)"); }
    static void p (int i)   { System.out.println ("p (int)"); }
}
```

Guess what will be printed when `Client` is recompiled and the application is run: `p (int)`. This is a serious issue because the developer modifying `Try` might not even be aware of the existence of class `Client`. This problem does not arise in Ada, because in Ada conversions have to be explicit.

The Ada equivalent of the first example would not compile since `1` is an integer literal and `p` only takes `float` parameters. If the programmer wanted to call `p (float)` he would have had to insert an explicit conversion: `float (1)` to insist that that is what he wanted or simply write 1.0 (which is probably what the original developer meant in the first place but Java did not catch this oversight).

Other variants of the above problem can arise in Java. Here is another example:

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

```
        { System.out.println ("p (Base)"); }
}
public class Client {
    public static void main (String args[])
        { Try.p (new Deriv ()); }
}
```

When the above application is run it will print `p (Base)`. If at some later time `Try` is modified as follows

```
public class Try {
    public static void p (Base obj)
        { System.out.println ("p (Base)"); }
    public static void p (Deriv obj)
        { System.out.println ("p (Deriv)");}
}
```

the application will instead print `p (Deriv)` if `Client` is recompiled. In Ada, depending how you write the above code, the compiler will either flag the call in the first example as not matching any procedure or it will flag the call in the second example as ambiguous. In either cases the programmer will have to clearly state his intentions.

Ada was explicitly designed to avoid the possibility of such behavior changes due to interface changes, and the programmer is notified by the compiler rather than silently altering the program's behavior.

While on the topic on overloading, it's worth mentioning some other shortcomings of Java's overloading model. Unlike Ada, Java only supports overloading based on parameter profiles and doesn't permit overloaded functions to be distinguished based on result type. This is an unfortunate restriction because, while this simplifies the language to some degree, it means that a class cannot declare multiple methods with the same parameter profile but different result types. It also means that if there are several Java interfaces that declare methods with the same names and parameter profiles but different result types, then it is not possible to define a class that simultaneously implements those interfaces. Another limitation is that Java does not support operator overloading, which is a very useful feature in Ada for the implementation of mathematical data types (e.g., rational numbers, vectors, matrices) and is a feature of Ada that can significantly enhance the readability of programs.

4.2 Program Consistency

The previous example gave us an opportunity to introduce the notion of program consistency. C, C++, Java, and Ada programs are built from several components compiled separately and then linked (statically or dynamically) into a final executable. In C and C++ it's possible to link inconsistent objects, that is objects that have been compiled making inconsistent assumptions about the interface to imported variables and routines.

In Java, where linking happens dynamically as the program runs, an elaborate set of rules has been defined to ensure that changes are compatible so as to avoid inconsistencies that could compromise the security of the JVM. This, however, does not guarantee that

the final application will run reliably (again illustrating that security and reliability are not the same thing). For instance, consider the following example. Initially the code looks like:

```
public class Base { }
public class Deriv extends Base { }
public class Try {
    static void p (Base obj) { System.out.println ("p (Base)"); }
}
public class Client {
    public static void main (String args[]) { Try.p (new Deriv ()); }
}
```

After compiling and running the above, the output will be `p (Base)`. If class `Try` is modified as follows

```
public class Try {
    public static void p (Base obj)
       { System.out.println ("p (Base)"); }
    public static void p (Deriv obj)
       { System.out.println ("p (Deriv)");}
}
```

and file `Try.java`. recompiled, program execution will still print `p (Base)`. If on the other hand class `Client` is recompiled after changing `Try`, the program outcome will be `p (Deriv)`. This is bothersome, but since there is no strict consistency requirement there is not much one can do in Java.

Ada, unlike C, C++, or Java, has the fundamental rule that all the objects that are linked in the final executable must be produced from a consistent set of sources. This means that the specifications used to compile all the units that compose the applications must match. To enforce this, Ada compilation systems have an additional step (and tool) that precedes linking called the binder. It is the binder's responsibility to check that the objects that are being linked are built with consistent specifications. It is impossible to skip or forget the binding step since this produces the necessary startup and elaboration code to launch the Ada application. Thus, in the previous case, if the equivalent of class `Try` were to be changed without recompiling also class `Client`, an inconsistency error would be produced by the binder or, alternatively, the binder can automatically recompile the obsolescent units.

4.3 Too Much Dispatching

C++, Java, and Ada have three different philosophical views when it comes to dispatching. In C++, only methods that are marked as `virtual` can dispatch, and other (nonstatic) methods never dispatch. Calls to `virtual` methods are dispatching by default (but there is a way to make these calls nondispatching). In Java, all nonstatic methods are dispatching and all calls to such methods will dispatch. The only exception to this rule is described in the following example.

```
public class Base {
    public void p () { System.out.println ("Base.p"); }
}
public class Deriv extends Base {}
public class Client extends Deriv {
    public void p () { System.out.println ("Client.p"); }
    public static void main (String args[]) { new Client().foo (); }
    void foo () { super.p (); }
}
```

When run, the above program prints `Base.p`. However, if `Deriv` is later modified to override `Base.p` as shown below

```
public class Deriv extends Base {
    public void p () { System.out.println ("Deriv.p"); }
}
```

then the program's will output `Deriv.p`. Is this intended? With Java's way of doing things it's hard to know. If the original programmer really meant to invoke `Base.p`. systematically, then the later addition of `Deriv.p` introduces an error and furthermore there is no simple way to fix this.

In Ada this problem does not arise. All nonstatic methods in Ada are potentially dispatching. However, it is at the point of the call that you decide whether to have a dispatching call. In addition, Ada offers the ability to statically select the precise method to call, thus making the programmer's intent unambiguous.

5 Abstract Away

The purpose of abstraction is to provide a concise view of something while hiding the irrelevant details. Java provides good abstraction mechanisms with classes. Unfortunately it does not have a complete abstraction model for all of its data types. In addition, the lack of separation between the specification and the implementation of a class can make code hard to read because the specification and implementation are textually intermixed, inhibiting the programmer from getting a clear view of the salient interface of the abstraction. The following sections will illustrate some abstraction loopholes in Java.

5.1 Scalar Abstraction

In Java you cannot declare new scalar or array types. This means that you can mistakenly mix values that represent conceptually very different objects. Consider for instance the following code excerpt:

```
for (int w = start_weight; w <= end_weight; w++)
    for (int l = start_length; l <= end_length; l++)
        do_something (w, l);
```

When reading this code how can you be sure `do_something (w, l)` is the intent of the original developer and not `do_something (l, w)`? Instead you have to look at the definition of `do_something` and hope the parameter names are clear. If they aren't then you have to dig into `do_something()` to figure that out. Furthermore, how can you be sure during the course of processing a weight or a length that one of these entities does not get assigned a negative value, thereby propagating a value which has no meaning throughout the execution. The answer is simple. You can't. You just hope your testing was thorough.

Ada provides a double defense against this problem. For one thing you can create bona fide data types. In addition, you can constrain the bounds of these types to be what makes sense for your application. For instance you can write:

```
type weight is range 0 .. 1_000;
type length is range 0 .. 3_000;
```

When you subsequently see

```
for w in start_weight .. end_weight loop
    for l in start_length .. end_length loop
        do_something (w, l);
```

and the program successfully compiles you can rest assured that the parameters have been passed in the correct order. To clarify things even further, assuming the names of the two parameters in `do_something` are `the_weight` and `the_length` you can even write

```
        do_something (the_weight => w, the_height => l);
```

The above type definitions also solve the range-checking problem. Every time an object of type `weight` is assigned a value which is not in the range 0 to 1000, an exception is raised. Moreover, Ada allows subtypes to be defined from the original types. Subtypes can further constrain the range of a type as shown in the following example.

```
type weight is range 0 .. 1_000;
type length is range 0 .. 3_000;

subtype human_weight is weight range 0 .. 400;
--  Instead of 400 we could have used any expression

w : weight := ...;
h : human_weight := ...;
l : length := ...;

w := h;          -- OK
h := 2 * w;      -- OK, but a check is made to ensure h in 0 .. 400
l := w;          -- Compile-time error. Ada is strongly typed.
l := length (w); -- OK, explicit conversion
```

5.2 Do You Want to Switch?

Java has no proper enumeration types, except for its type `boolean`. Given the semantics of `switch` statements and the interaction with the `break` instruction, this can yield unpleasant surprises. Here is an example:

```
public static final int LOW    = 0;
public static final int MEDIUM = 1;
public static final int HIGH   = 2;
...
switch (alert) {
   case LOW :
     ...
     break;
   case MEDIUM :
     ...
     break;
   case HIGH :
     ...
}
```

First of all, if one of the `break` statements is forgotten, then disaster strikes because of the semantic rules of `switch` statements in Java, which are inherited from C. Assume that the developer was careful and added the appropriate `break` statements. What can go wrong? Several things.

Suppose at a later stage it is decided to add processing for `VERY_HIGH`. You will notice that there is no `break` statement at the end of the last `case`. This is perfectly valid since it is the last entry in the `switch` statement. However, when the code is later transformed into

```
switch (alert) {
   case LOW :
     ...
     break;
   case MEDIUM :
     ...
     break;
   case HIGH :
     ...
   case VERY_HIGH :
     ...
}
```

by a different developer, a bug is most likely introduced. What is worse is that there is no way to ensure that all `switch` statements dealing with alerts have been updated to incorporate the `VERY_HIGH` case.

Ada addresses these potential problems by allowing full-fledged enumeration types and forcing case statements to cover all possible values that they might need to handle. Thus, in Ada one could write:

```
type alert_kind (LOW, MEDIUM, HIGH);
...
case (alert) is
   when LOW =>
     ...
   when MEDIUM =>
     ...
   when HIGH =>
     ...
end case;
```

The first thing to notice is that no `break` statement is necessary because Ada's `when` clauses have an automatic `break` at their end. Furthermore when the type `alert_kind` is modified into

```
type alert_kind (LOW, MEDIUM, HIGH, VERY_HIGH);
```

without modifying the case statement, an error is issued by the compiler. Incidentally, note that like Java and C, Ada has a default clause (spelled "`when others`" in Ada) that subsumes all missing values. Incidentally, note that the use of `others` in a case statement for an enumeration value is problematic for the same reasons explained above for Java, that is adding a new enumeration element will lead to existing code having an effect that might not have been intended. The difference between Ada and Java here is that Java `switch` statements have a default "`when others`", whereas in Ada the "`when others`" has to be written explicitely.

5.3 Limit Your Liabilities

Java has copy semantics for all scalar types, but pointer semantics for arrays and class instances. This can be quite confusing, especially if your team has to switch between Java and other programming languages.

One of the benefits of Ada on the JVM is to provide both value and pointer semantics. It is up to the developer to make a choice that is appropriate at the implementation level. What's more, Ada can restrict the use of assignment and comparison by using the keyword `limited`. Here is a code excerpt showing the problem and its fix in Ada:

```
public class Queue {
   ...
   public void insert (int elmt) {...}
   public int  get () {...}
   //  extracts the first element in the Queue
}
```

```
...
static public Queue global_q = ...;

Queue q = global_q;
q.get ();
```

Here the call to `q.get()` will remove the first element of `global_q` as well. The problem is that there is no way in Java for the implementor of class `Queue` to restrict the ability of its clients to make this kind of mistake. Ada addresses this problem by providing `limited` types. A `limited` type is a type whose objects cannot be compared (unless the programmer explicitly defines an "=" operation) or assigned. For instance, in the above example one could write:

```
package Queues is
   type Queue is limited private;
   procedure insert (q : Queue; elmt : integer);
   function  get (q : Queue) return integer;
private
   type Queue is .. -- details of a Queue
end Queues.
```

Given the presence of the limited keyword in the type declaration, the Ada compiler will flag the following assignment as a compilation error:

```
global_q : Queue;
...
q : Queue := global_q;  --  Compilation error
```

5.4 Separate Specification and Implementation

Java does not separate the specification of a class (i.e., its list of public methods and fields) from the implementation details of the class. Everything is grouped in the class and a single source file. This is not a readability issue for small classes, but it becomes one for larger classes, especially in the presence of nested classes.

The Ada building block for modularity is the package. (The Ada package should not be confused with the Java package, which is a way of providing a namespace for a related set of classes along with some additional inter-class visibility.)

Packages in Ada are used to define groups of logically related items, ranging from simple collections of common constants, variables, and subprograms to full-fledged encapsulated data types.

An Ada package enforces clear separation between the specification of the package, that is, information that is usable by the clients of the package, and its implementation. The internal information is hidden, and thereby protected from deliberate or inadvertent use by other programmers.

As an example, the following example shows the specification and implementation of a package called `Queues`.

```
--  Specification of package Queues
package Queues is
   type Queue is limited private;
   procedure insert (q : Queue; elmt : integer);
   function  get (q : Queue) return integer;
private
   type Queue is .. -- details of a Queue type
end Queues.

--  Implementation details of package Queues
package body Queues is
   ------------
   -- insert --
   ------------
   procedure insert (q : Queue; elmt : integer) is
     ...  --  local data, types, subprograms
   begin
     ...  --  implementation of insert goes here
   end insert;

   ---------
   -- get --
   ---------
   function get (q : Queue) return integer is
     ...  --  local data, types, subprograms
   begin
     ...  -- implementation of get goes here
   end get;
end Queues.
```

Readability is an important benefit obtained by separating the specification from the implementation. Moreover, this also makes it easier to replace one implementation of the services offered by a package by another.

An interesting side effect obtained by separating specification and implementation is diminishing the amount of recompilations needed after changing the implementation of a package body. In Java, when the source of a class file is changed, either you have to recompile all classes using that class or you have to track each change to see if it is binary compatible with the rest of the system (not an easy task).

In Ada this is not the case. Typically the specification of an Ada package is put in a different source file than the body of the package. For instance, in JGNAT, the specification of package `Queues` would be put in file `queues.ads` while its body would be placed in file `queues.adb`. If changes are made to `queues.adb`., then only that file needs to be recompiled. The Ada package spec is a type-safe version of the C or C++ header files (`.h` files).

In Java, tools like `javadoc` can extract a class specification and present it in HTML format, as long as the source code does not contain certain types of syntax errors. Having a

tool that extracts spec information rather than requiring the user to separate specification from implementation, as in Ada, makes the compiler writer's life simpler at the expense of more user work. The user must rerun his specification-extraction tools every time the source file of a Java class changes.

6 Missing Features

This section lists some of the features missing from the Java programming language whose absence forces developers to use convoluted programming idioms or necessitates the outright duplication of code. This increases the chance of a mistake and hides the original intent of the programmer, resulting in code that is harder to read. The final result is decreased reliability.

6.1 Unsigned Integers

Even though Java provides wrap-around semantics for integers, it provides no unsigned types (except for `char` which is not really an integer type). because of the lack of unsigned comparison. For instance, for the type `byte`, the number `0xff` is always less than `0xfe`. Thus, performing unsigned comparison requires a series of signed comparisons. Try to write down the unsigned comparison `x<y` using the available Java operators. It is not trivial and the intent of the resulting code is likely to be unclear to the reader.

Ada provides arbitrary unsigned types, which are called modular types because they generalize the notion of C unsigned type and allow for wrap-around semantics with any base, not just for powers of 2. For instance, you can write

```
type Unsigned_8 is mod 2**8;
--  An unsigned byte

type Hash_Index is mod 1021;
--  The values of an object of type Hash_Index go from 0 to 1020.
--  Objects of type Hash_Index have wrap-around semantics. This
--  type could be used as the index of a hash table.
```

6.2 Fixed-Point Types

The representation of integer types is exact (albeit bounded) in all programming languages. However, real types are approximate and introduce problems of accuracy which can have very subtle reliability effects. In Ada, real types are subdivided into floating point types, which have a relative error bound, and fixed-point types, which have an absolute error bound. For instance in Ada you can write

```
type Dollars is delta 0.01 digits 11;
--  A fixed-point type with 11 digits max and an
--  absolute error of 0.01.

Money : Dollars;
```

```
--  Money can range from -999_999_999.99 to +999_999_999.99
```

Fixed-point types such as the above can be used, for instance, in accounting applications, and the Ada standard provides an annex (the Information Systems annex) that specifies a number of libraries for interfacing Ada with COBOL, as well as providing some of the basic COBOL facilities such as "picture" editing familiar to COBOL programmers.

Fixed-point types are not only useful for accounting and financial computations. Because fixed-point types have absolute error bounds, programmers can define types that express units such as voltage, intensity, pressure, length, and weight. For instance:

```
type Volts is delta 0.001 digits 6 range 0.0 .. 240.0;
--  From 0.0 .. 240.00 with 6 digits of precision e.g. 135.459 volts
```

While it is possible to simulate the effect of fixed-point types using integers (indeed integers are generally the underlying representation of fixed point), this is clearly error prone and yields code that is hard to read. Note that fixed-point types are particularly relevant for embedded implementations of the JVM that do not provide support for floating point.

6.3 Limited Parameter Passing Mechanism

Like C, but unlike C++ and Ada, the only parameter passing mechanism in Java is copy-in. This makes it very convoluted to write something like a swap routine for scalars. Basically you have to wrap your scalar inside an object and create a spurious class or array to achieve this.

Ada has three parameter modes: `in`, `in out` and `out`. `in` parameters cannot be modified inside the routine, whereas `in out` parameters can be modified and the updated value is available to the caller. `out` parameters are those whose value is computed inside a routine and whose result value is sent back to the caller. The difference between `in out` and `out` is that the programmer needs to ensure that an `in out` parameter is initialized before the call. As an example, here is a swap routine written in Ada:

```
procedure Swap (X, Y : in out Item) is
   Tmp : constant Item := X;
begin
   X := Y;
   Y := Tmp;
end Swap;
```

6.4 Pointers to Functions

Consider the following problem. You have to implement an integration routine `integrate` which takes a function `f` as a parameter as well as two floating point bounds `a` and `b` over which to integrate `f`. `f` takes a `float` parameter and returns a `float`. How can you write `integrate` in Java ?

The initial impulse of rushing to use pointers to functions must be repressed since there is no such concept in Java. One possibility would be to use interfaces as shown in the following example:

```
public interface Function_Interface {
   public float f (float x);
}
public class Numeric_Analysis {
   public static float integrate
     (Function_Interface r, float a, float b){
      float x;
      ... // calls to r.f (x)
   }
}
public class Foo implements Function_Interface {
   public static float compute (float x) { ... }
   public float f (float x) { Foo.compute (x); }
}
public class Client {
   public static client () {
      Foo r = new Foo ();
      //  must create a Foo instance to pass it to integrate
      float val = Numeric_Analysis.integrate (r, 1.0 10.5);
      ...
   }
}
```

The problem with this approach is that every class that contains a function that you want to integrate must implement interface `Function_Interface`. If the class whose function you want to integrate is in a library, you have to create your own view of that class. If it is a class in your application you have to modify it as done in class `Foo` above. If you have a class containing several such functions then you have to create a sister class for each additional function that you want to integrate. All in all this is not terribly convenient nor is it very readable. So another approach is needed.

The alternative approach is to create an abstract class `function_reference` with a method `f`. Then for each function that we want to integrate we have to create a class derived from `function_reference` where `f` is overridden and calls the actual function to integrate. This is illustrated in the following example.

```
public abstract class Function_Reference {
   public abstract float f (float x);
}
public class Numeric_Analysis {
   public static float integrate
     (Function_Reference r, float a, float b)
   {
      float x;
      ... // calls to r.f (x)
   }
}
```

```
public class Foo {
   public static float fun (float x) {...}
   ...
}
public class fun_ptr extends Function_Refernce {
   public float f (float x) { return Foo.fun (x); }
}
public class Client
   public static client () {
      fun_ptr ref = new fun_ptr ();
      //  must create a fun_ptr instance to pass it to integrate
      float val = Numeric_Analysis.integrate (ref, 1.0 10.5);
      ...
   }
}
```

Needless to say this approach is laborious and not very readable either.

Like C and C++, Ada provides explicit pointers. However, unlike C and C++, the Ada pointer type model is based on strong typing and is safe and reliable. Ada's pointer model is similar to the Java reference model, except that it has been extended to program entities such as scalars and functions. When it comes to functions, one of the benefits of strong typing is to provide a reliable way for doing call-backs. As an example, here is how you would do the above in Ada:

```
type Integrand is access function (x : float) return float;
function integrate (f : Integrand; a, b : float);

function fun (x : float) return float;

procedure client is
   val : float := integrate (fun'access, 1.0, 10.5);
   ...
end client;
```

6.5 Generics

One feature of C++ that was not adopted by Java is static polymorphism. Both static and dynamic polymorphism provide the ability of dealing with different types within a unified framework. However, while dynamic polymorphism selects the needed construction form the framework at run time based on the underlying type, static polymorphism performs this selection statically, at compile time.

One might wonder whether static polymorphism is needed at all in the presence of dynamic polymorphism. Indeed Java takes the stance that static polymorphism is unnecessary, since a variable of type Object can reference instances from any class. However, consider the following situation. Say you want to sort an array of items (numbers or objects) from the `m`-th to the `n`-th elements inclusive. The Java API provides over fifteen `sort`

routines in `java.lang.util.Arrays`, duplicating what is basically the same algorithm. In the case of `sort`, the java API does this duplication for you (which results in having all of the `sort` routines being loaded with your application whether you are using two, three or all of them). However there are other cases where the routine you need is not available in the Java API. In this case you have to perform this duplication yourself.

Ada has a complete framework for static polymorphism called generics. As an example, the following code gives the generic procedure to sort an array of `Items`.

```
generic
   type Item is private;
   --  The array component type

   type Array_Type is array (integer range <>) of Item;
   --  The type of arrays to be sorted

   with function ‘‘<’’ (x, y : Item) return boolean is <>;
   --  The comparison function. The ‘‘is <>’’ tells the compiler to
   --  use a function with the same signature and name as this one
   --  if such a function exists at the point of instantiation.
procedure sort (C : in out Array_Type);
--  The specification of the sort routine

--  The implementation of the sort routine
procedure sort (c : in out Array_Type) is
   min : integer;
   tmp : Item;
begin
   for k in c'first .. c'last - 1 loop
      min := k;
      for j in k + 1 .. c'last loop
         if c (j) < c (min) then
            min := j;
         end if;
      end loop;
      tmp     := c (k);
      c (k)   := c (min);
      c (min) := tmp;
   end loop;
end sort;
```

Here is an example of using this generic to sort an array of integers:

```
type Int_Array is array (integer range <>) of integer;
a : Int_Array (1 .. 100) := ...;
--  The bounds could be anything and do not have to be static
```

```
procedure int_sort is new sort (Item       => integer,
                                Array_Type => Int_Array);
--  Generic instantiation.
--  A copy of sort is created with the right types.
--  By default it will use integer ‘‘<’’ as the comparison function

int_sort (a);
int_sort (a (m .. n));
--  The array slice a (m .. n) denotes the array object from the
--  m-th element to the n-th element. So this call will sort a from
--  the m-th element to the n-th element.
```

If you have an array of some arbitrary type, you just need to define a comparison function, and make it available to the instantiation as shown in the following example:

```
type Stuff is ...;
type Stuff_Array is array (integer range <>) of Stuff;
b : Stuff_Array (x .. y) := ...;

function ‘‘<’’ (p, q : Stuff) return boolean;

procedure stuff_sort is new sort (Item       => Stuff,
                                  Array_Type => Stuff_Array);

stuff_sort (B);
```

7 Conclusion

There are a number of other features that enhance Ada's reliability relative to Java that have not been addressed in this article. Foremost is the concurrency and real-time programming model offered by Ada. This model, along with Ada's comprehensive set of facilities to lay out objects in memory precisely and portably, are very relevant to the field of pervasive computing and real-time JVMs [8]. Unfortunately we had to limit their coverage for brevity's sake (see [9] and [11] for detailed coverage of this subject).

Humans make mistakes. Programmers are no exception. While the Java programming language fixes the security problems of C and C++, it only partially addresses reliability. Ada was designed with program reliability as one of its principal goals, and as such it offers an interesting programming tool to complement Java when building reliable software systems running on the Java platform.

8 Thanks

The authors would like to thank Ben Brosgol of Ada Core Technologies for helpful discussions, comments and suggestions regarding this paper.

References

[1] "Pervasive Computing 2000", IT Conference, National Institute of Standards and Technology, Gaithersburg, Maryland, January 25-26, 2000. http://www.nist.gov/pc2000/.

[2] "Embedded Opportunities", by Franco Gasperoni, in Reliable Software Technologies - Ada-Europe 1998, Lecture Notes in computer Science 1411, pp. 1-13, 1998.

[3] "The Java Tutorial Second Edition", by Mary Campione and Kathy Walrath, Addison Wesley 1998.

[4] "Programming the Internet in Ada 95", by Tucker Taft, Reliable Software Technologies - Ada-Europe 1996, Lecture Notes in computer Science 1088, pp. 1-16, 1996.

[5] "Ada 95 - 2nd edition", by John Barnes, Addison Wesley, 1998.

[6] "Ada 95 Problem Solving and Program Design", 3rd ed. by Feldman, M.B. and Elliot B. Koffman, Addison-Wesley, 1999.

[7] "The Java Programming Language", by Ken Arnold and James Gosling, Addison Wesley, 1996.

[8] "Real-Time Java API", Real-Time for Java Experts Group, Sun Microsystems' JSR-000001, http://www.rtj.org.

[9] "Concurrency in Ada", by Alan Burns and Andy Wellings, 1998, Cambridge University Press.

[10] See `http://www.gnat.com/texts/products/pjava_set.htm`.

[11] "A Comparison of the Concurrency Features of Ada 95 and Java", by Ben Brosgol, SIGAda '98 Conference Proceedings; Washington, DC, November 1998.

[12] "A Comparison of Ada And Java as a Foundation Teaching Language", by Ben Brosgol, Ada Yearbook 2000.