A Comparison of Ada and Java™ as a Foundation Teaching Language*

Benjamin M. Brosgol

Ada Core Technologies
79 Tobey Road
Belmont, MA 02478
(617) 489-4027 (phone)
(617) 489-4009 (FAX)

brosgol@gnat.com
http://www.gnat.com

Introduction

Java has entered the software arena in unprecedented fashion, upstaging languages and technologies that are longstanding players in the industry. Almost unheard of before 1995, the language and its surrounding technology are attracting increasing attention not merely in the hardware and software communities but also among lay users and in the popular press. This phenomenon has not escaped the attention of academia, and a growing number of colleges and universities are looking at Java as a candidate "foundation" language on which to base computer science curricula.

This paper looks at the role of a programming language for teaching computer science courses and compares Ada and Java against the identified criteria. It concludes that Ada is the superior choice, based on technical factors such as its more comprehensive feature set and its methodology-neutral design, and also taking into account external factors including the availability of good but inexpensive compilation environments.

Section 1 provides a brief overview of the Java technology. Section 2 identifies the criteria relevant to choosing a programming language for a foundation-level Computer Science course. Section 3 compares Ada and Java with respect to the criteria related to technical aspects of the language, and Section 4 compares the languages with respect to external factors. Section 5 summarizes the conclusions of this analysis.

---

* This is an updated version (March 2000) of a paper originally presented by the author, while an employee of Aonix, at the ASEET '98 symposium.

Appendix A furnishes a summary of the Java language. It is assumed that the reader is familiar with Ada, and thus an Ada language summary is not included. Web sites with links to Ada information include [SIGAda] and [AdaResourceAssoc].

1 The Java Technology*

Java is portrayed on one of Sun's Web pages [Sun] as a "simple, object-oriented, network-savvy, interpreted, robust, secure, architecture neutral, portable, high-performance, multithreaded, and dynamic language". This is an impressive string of buzzwords but it is useful to distinguish among three elements:

- The Java language
- The classes comprising the Java API (Application Programmer Interface)
- The Java execution platform, also known as the *Java Virtual Machine* or simply *JVM*.

In brief, Java is an Object-Oriented language with features for objects/classes, encapsulation, inheritance, polymorphism, and dynamic binding. Its surface syntax has a strong C and C++ accent: for example, the names of the primitive types and the forms for the control structures are based heavily on these languages. However, the OO model is more closely related to so-called "pure" OO languages such as Smalltalk. Java directly supports single inheritance and also offers a partial form of multiple inheritance through a feature known as an "interface". Related classes and interfaces may be grouped in a *package*, which is a host system facility (typically a directory) and not a syntactic element of the language.

Java supplies exception handling and multi-threading, defined through OOP.

A key property of Java is that objects are manipulated indirectly, through implicit references to explicitly allocated storage. The JVM implementation performs automatic garbage collection, as a background thread.

One can use Java to write stand-alone programs, known as *applications*, in much the same way that one would use Ada, C++, or other languages. Additionally, and a major reason for the attention that Java originally attracted, one can use Java to write *applets* – components that are referenced from HTML pages, possibly downloaded over the Internet, and executed by a browser or applet viewer on a client machine.

Supplementing the language features is a comprehensive set of predefined classes. Some support general-purpose programming: for example, there are classes that deal with string handling, I/O, and numerics. Others, such as the Abstract Windowing Toolkit and Swing, deal with Graphical User Interfaces. Others

---

* The material in this section and in Annex A is based on [Bro97]. Further information on Java may be found in [Fl97], [AG96], and [GJS96].

support specialized areas including distributed component development, security, and database connectivity.

There is nothing intrinsic about the Java language that prevents a compiler from translating a source program into a native object module for the target environment, just like a compiler for a traditional language. However, it is more typical for a Java compiler to generate a so-called *class file* instead: a sequence of instructions, known as "bytecodes", which are executed by a Java Virtual Machine. This facility is especially important for downloading and running applets over the Internet, since the client machine running a Web browser might be different from the server machine from which the applet is retrieved. Obviously security is an issue, which is addressed through several mechanisms, including:

- Restrictions in the Java language that prevent potentially insecure operations such as pointer arithmetic and unchecked deallocation
- The implementation of the JVM, which performs a load-time analysis of the class file to ensure that it has not been compromised
- The implementation of the browser or applet viewer, which checks that a downloaded applet does not invoke methods that directly access the client machine
- Security-related classes in the API

2 Criteria for a Foundation Language

Very few programming languages have been designed specifically for teaching; fewer have achieved widespread acceptance. Pascal is perhaps an exception, but several omissions have led to its decreasing significance over the years. It lacks, among other things, a modularization construct more comprehensive than a procedure or function, and facilities for exception handling. Ironically, the very goal that Pascal set out to achieve – language simplicity – proved to be its downfall. As technology evolved and the demand for larger applications increased, the limitations of Pascal became more apparent. Some vendors responded by providing added functionality, for example Object-Oriented Programming in TurboPascal, but these extensions were non-standard. Programs that exploited such facilities proved non-portable to other environments.

Although the choice of a foundation language for computer science is based on many criteria, we can divide these into two main categories: intrinsic elements related to specific language features, and extrinsic factors such as tool support.

2.1 Intrinsic criteria

- Learnability

  One aspect of this criterion is language simplicity; in general, the smaller the number of features, the easier it is to learn the semantics. However, this goal must be placed in context and balanced against other requirements. As evidenced by the Pascal experience, a simple language can lead to complexity

in program development if the language lacks the necessary functionality. Moreover, if a user or student needs to master a large and complex API in order to program effectively, then language simplicity is not of much benefit.

Learnability implies that the language should not contain semantic "traps and pitfalls" that programmers need to be alert to, that it should offer simple solutions to simple problems, and that its design should be reasonably orthogonal so that students do not need to cope with numerous special case rules. Moreover, a foundation language needs to strike a balance between being sufficiently high level to provide a natural notation for modeling a range of software problems, while having an understandable mapping to an underlying target machine and operating system environment. If the language is too low level then it will be tedious and error-prone to use; if it is too high level then it may either fail to reflect important efficiency concerns or else present a confusing picture of what is actually going on at run time.

- Expressiveness / generality

  The language should be powerful enough to express the variety of data structures and algorithmic elements covered in a foundation course, and should also go beyond Pascal in a number of areas such as module definition and exception handling. The language should support a variety of development methods, since no one approach is appropriate for all applications.

  A foundation course typically serves as the basis for upper-level specialized courses that survey the programming language landscape. It is thus useful if the language used in a foundation course exposes the issues that will be addressed in such later courses.

- Encouragement of sound software engineering, including Object-Oriented Programming

  Strong typing, separate compilation, and encapsulation should be provided. The language should also have a robust approach to reusability; a good test case is how one can define a reusable module for container data structures such as stacks and queues.

  With all the attention that object orientation is receiving in the professional software community, universities are increasingly motivated to introduce OOP early into their computer science curricula. Thus it is useful if a foundation language is object oriented, with support for classes, inheritance, polymorphism, dynamic binding, and related topics.

- Support for concurrent programming

  Given the importance of multithreading in modern environments, the language should support the definition of concurrent threads of control and a mechanism for threads to synchronize and communicate. This support may be either through the API or specific language syntax.

- API functionality

  The language should provide access to a comprehensive Application Program Interface for purposes such as Input-Output, numeric processing, character and string handling, network communications, and GUIs. Of course, a large API somewhat conflicts with the earlier-stated goal of learnability; the API should be structured in a consistent way and make effective use of the language's definitional facilities.

2.2 Extrinsic criteria

- Availability of tools

  For university environments, compilation systems need to be of high quality but low cost. Relevant factors are ease of use, accuracy of diagnostic messages, and compile-time performance; run-time efficiency is less critical.

- Availability of good textbooks

  Texts need to be geared to students versus professional programmers, with detailed coverage of basic data structures and algorithms.

- Portability / stability / standardization

  Source code portability across different environments is important, since many universities have a mix of PCs, UNIX workstations, and possibly others.

- Status as "hot" technology (i.e., where the jobs are)

  Given the competition for students and the pressure to teach marketable skills, there are some benefits to teaching a language that is popular in the marketplace. This tends to conflict with the previous goal, however, since the "hottest" technology tends to be the least stable.

In the remainder of this paper we will compare Java and Ada with respect to these factors

3 Intrinsic Factors

3.1 Learnability

3.1.1 Java Learnability

To some extent the design of Java was a reaction to the size and complexity of C++, and Java certainly has fewer features than C++ or Ada. However, a potential adopter of Java for teaching a foundation CS course should recognize that Java is not immune to semantic subtleties, irregularities, and "gotchas". Here are a few examples.

A heritage of the C syntactic underpinning is that Java suffers from the classical "dangling else" problem: the code fragment

```
if ( redLight )
   if ( noTraffic )
      turnRight();
else
   goFaster();
```

does not do what the indentation implies; the method `goFaster()` is invoked when `redLight` is `true` and `noTraffic` is `false`, not when `redLight` is `false`.

Another error-prone construct, also courtesy of Java's C syntax ancestry, is the use of "=" as an operator symbol:

```
void foo(){
   boolean b1, b2;
   ...
   if (b1=b2){ ... }
}
```

The above code is legal, but anyone reading it will wonder if the programmer had intended the condition as the equality test "`b1==b2`" rather than the assignment "`b1=b2`".

Java uses the same error-prone rules for numeric literals as C and C++. Two main problems stand out: the mathematically confusing role of a leading "0" to indicate an octal base for an integer literal, and the absence of a separator character such as an underscore. Since Java has 64-bit integer and floating-point types, literals may have up to 18 digits, and the absence of a separator character makes it tedious and frustrating for a human reader to check that literals have been correctly specified. For example, here is the Java literal for the `long` value 2^{63}-1:

```
0X7FFFFFFFFFFFFFFFL
```

Here is a corresponding Ada literal:

```
16#7FFF_FFFF_FFFF_FFFF#
```

There should not be any doubt that the Ada version is much easier to read and much less susceptible to error.

Java's integer arithmetic is a curious hybrid of signed and unsigned. In unsigned fashion, arithmetic operations exhibit "wrap-around" behavior rather than overflowing, and narrowing casts truncate high-order bits. But comparisons, and also conversions to String, treat integer values as signed. Another oddity is that a hexadecimal-base integer literal is treated as a signed 32- or 64-bit value, while a decimal integer literal is treated as unsigned. Thus the following initialization is legal and is equivalent to assigning -1 to i1:

```
int i1 = 0XFFFFFFFF;  // -1
```

However, the following is illegal since the decimal literal value exceeds 2^{31}-1:

```
int i2 = 4294967295;  // 2**32 - 1
```

Simple interactive IO (especially numeric input) is surprisingly complicated, requiring intermediate constructors to perform the appropriate filtering. It probably makes most sense for an instructor to introduce a SimpleIO class that hides the ugliness, but this is not part of the standard.

Ideally, a language used in teaching a foundation course, especially where some of the students might lack previous programming experience, should not contain "forward references" to concepts that are needed before they have been covered. In Java, however, the requirement for a method to include a "`throws`" clause for classes of propagated checked exceptions will almost invariably result in students meeting examples of exceptions before the topic is explained in the lectures.

A fundamental impediment to learning Java is that understanding the OO paradigm is a prerequisite for effectively using the language. For example, exceptions are based on OOP, so students will encounter (or trip over) inheritance before they understand how it works, another "forward reference". Java's OO-centricity will also be discussed below in connection with other factors, but it definitely affects the way the language is taught and learned. OOP pays the highest dividends in the construction of large, complicated systems, and thus students will not necessarily appreciate its usage in small examples. Moreover, due to its OO core, Java can make some simple things seem difficult. Since a class is both a module and a data template it is not easy to define a class that is a "pure module" (i.e., which cannot be instantiated). One needs to employ a rather high-powered feature to get this effect, namely a private constructor.

These problems are not just theoretical. In summarizing the results of using Java in an undergraduate computer science curriculum, Martin [Mar98, p. 38] observed:

> *Experience gained so far would suggest that ... OO is far from being a natural way of viewing the world. The learning curve, especially for ab-initio students who are faced not only with grasping concepts about stored programs, control structures, variables and references but also with OO concepts of class, instance and inheritance in a language which has a number of unhelpful, 'ugly' features, is extremely high.*

Gibbons [Gib98, p.40] argues similarly:

> *There is no doubt that the principal features of OOP (namely encapsulation of related data and code into objects, and inheritance into a hierarchy of object classes) are of great importance when it comes to constructing large software systems. However it is a long journey from the first programming course to the construction of large software systems.*

One of the reasons that OO concepts may be difficult in a first-level course is that there is a large conceptual distance between a record that contains data fields and a class that contains both data and methods. A record that contains data fields is easy to visualize as a run-time structure (i.e., it has a natural

implementation model). The same cannot be said for an instance of a class when the class has methods as members. Indeed, when attempting to provide an implementation model, an instructor will typically need to first provide an oversimplified explanation; e.g., where each class instance contains pointers to the code for the various instance methods. Eventually the truth will probably emerge, with diagrams of method dispatch tables (one per class).

Other aspects of Java's OO style are potential sources of confusion. The implicit "`this`" parameter is subtle, and it might not be immediately clear to students why there are different rules for invoking a method on `this` versus on `super` (the former is dynamically bound, the latter is bound statically). Indeed, a major underpinning of the "class as template and as module" design, the difference between static and per-instance members, leads to a very common error: a static method attempting to reference a per-instance member. Moreover, despite the similarity in syntax between selecting a field and invoking a method, the semantics is quite different when the target is a reference that is cast to a superclass: the field selection retrieves the given field as defined by the superclass (even if shadowed by a field with the same name in the sublcass) whereas the method invocation is dynamically bound.

3.1.2 Ada Learnability

Ada is a large language, and although size by itself does not imply that a language is difficult to learn, in fact the interrelationship of Ada's facilities does present a challenge to a teacher. Fortunately, some of the most troublesome problems have been solved in Ada 95. One example is the need in Ada 83 to either instantiate a generic unit or to depend on a non-standard package for simple interactive numeric I/O. The Ada 95 predefined environment includes `Ada.Integer_Text_IO` and other packages for this purpose.

Ada does have a few constructs whose run-time effects may surprise the unwary. One is a loop parameter that hides an outer variable:

```
declare
   Alpha : array( 1..l0) of Integer;
   J     : Integer;
begin
   ...
   for J in Alpha'Range loop  -- Hides outer J
      ...
      exit when Alpha(J) < 0;
      ...
   end loop;
   Put( Integer'Image( J ) );  -- Displays uninitialized outer J
end;
```

The compiler may or may not give a warning message for this error. The analogous construct in Java would be illegal, since a declaration in an inner loop is not allowed to hide an outer variable.

Another area in Ada that is potentially confusing is elaboration order. Even relatively simple programs may require elaboration control pragmas in order to avoid references to non-initialized data or "access

before elaboration" errors. Ada 95 has introduced some pragmas that are easier to use than Ada 83's `pragma Elaborate`, but users still need to be alert to the potential problem. Java, however, has related issues with respect to the execution effects of class loading, and in any event the elaboration order issues arise mainly because Ada separates unit specifications from unit bodies, a design decision that supports encapsulation and reduces compilation costs.

Numeric literals present semantic complications in any strongly-typed language that is to be implemented on hardware with multiple sizes for primitive numeric data. Ada's approach, based on the conceptual types *universal_integer* and *universal_real*, is not especially obvious; types in general are a difficult notion for many students to master, and conceptual types introduce an additional level of abstraction. Java's approach, in which literals are typed but where rules specify the contexts in which implicit conversions (casts) are applied, is arguably simpler than Ada's semantics.

The attribute notation in Ada is fairly complex, and it generally takes students some time before they have an intuitive grasp of what is used where. Some attributes apply to (sub)types, some apply to objects, and some apply to both. The syntax can appear unnecessarily heavy; students wonder why they need to utter `Enum_Type'Pos(My_Enum_Variable)` versus `My_Enum_Variable'Pos`, where `Enum_Type` is an enumeration type and `My_Enum_Variable` is a variable of that type.

Two of the most basic concepts in Ada are type and subtype, yet the distinction between the two is slippery. Although the syntax of a type declaration suggests that a type is being created, the semantics implies that both a type and a subtype are the result. In fact, a user can effectively program in Ada while ignoring such subtleties, but it is still a potential source of confusion.

Ada is not an especially orthogonal language; restrictions or special case rules are sometimes needed in order to avoid implementation problems or run-time inefficiencies. One example, which users of the Ada 95 access type facilities have encountered, is the static subtype conformance check that is performed for the `'Access` attribute. This attribute, when applied to an aliased object of a constrained subtype, is illegal if the designated subtype in the access type declaration is unconstrained. This in itself is not a major problem; the main issue is that some Ada instructors are not familiar with the rule, which can introduce frustrations for students. Another restriction motivated more by implementation concerns than by semantic problems is the inability to pass as a run-time parameter an access-to-subprogram value designating a nested subprogram, when the access type is declared at an outer scope. There are also some gaps in functionality that may seem puzzling: why does Ada provide access-to-constant types, but not access-to-constant parameters? Some of these issues may be addressed in the next revision of the Ada language standard.

On the other hand, unlike Java, Ada does not require familiarity with OOP for effective use of the language, and hence it allows the instructor to defer covering OO concepts until more basic constructs have been introduced. Its approach to OOP provides different constructs for "class as module" (a package),

"class as object template" (a tagged type), and "class as the type of a polymorphic variable" (an access value with a class-wide designated type). This does introduce some semantic subtlety, especially the concept of a class-wide type, but it also avoids Java's complications over "`this`" versus "`super`" and static versus per-instance members.

3.1.3 Conclusions on Learnability

Although Java is a smaller and simpler language than Ada, it has a number of properties that make it more difficult to learn. Probably the most significant is the need for an instructor to introduce OO concepts quite early, before students have understood the reasons why OOP is of benefit. Ada is easier to partition into different levels of language features. The first tier comprises the "Pascal subset" (data types, statements, expressions) plus simple packages (without private parts) and overloading. Just these features are sufficient for conveying a large number of concepts and will serve to give students a reasonably gentle introduction. The second tier consists of encapsulation (private types), child units, exception handling, and generics. The third tier comprises OOP and tasking. A foundation course can introduce these tiers sequentially.

Ada strikes a reasonable balance between its high-level features and its goal for an efficient stack-based run-time model. Java's heap-based semantics requires specialized run-time support (most obviously a garbage collector) and disguises issues that computer science students need to be familiar with.

3.2 Expressiveness / generality

Ada offers a large number of features that Java lacks, and conversely (but less significantly) Java supplies some functionality that is missing or restricted in Ada.

3.2.1 Ada features that Java lacks

Many of Java's omissions are related to data types. Here is a sampling:

- Enumeration types
- Record and variant record types
- "Pointers" (access types)
- Fixed-point types
- Subtypes, range constraints
- Strongly-typed numerics

Most of these are basic data structuring facilities that all programmers should be familiar with. Especially in a foundation course, students should be exposed to record types and array types and understand the issues raised by explicit pointers. Simulating an enumeration type by a set of static constants is clumsy and sacrifices the ability to iterate over the set of values. Using a class to capture the effect of a fixed-point type implies run-time overhead and notational inconvenience (for example: no literals, no operator symbols, and no cast operation to/from integer or floating-point types). Modeling a variant record by a

class hierarchy may be cumbersome and introduce unwanted generality. Using classes instead of records induces a level of indirection and the need for garbage collection. The absence of strong typing for numeric data means, for example, that all 32-bit integers are of type `int`.

For a foundation course, Java therefore provides too narrowly focused a view of data structuring. Moreover, the presence of garbage collection and the absence of explicit pointers mean that students do not need to learn about fundamental issues such as storage leakage and dangling references. From a pedagogical point of view, it is generally best to introduce a solution after the students have appreciated the problem. Gibbons [Gib98] calls this the "why" approach to teaching and argues that for computer science majors it is the appropriate style. The situation is somewhat analogous to deciding if a calculator should be allowed in a primary school in teaching arithmetic. After the pupils understand the concepts behind long division then they can use a calculator to ease their work, but giving this to them before means that they might fail to learn the basic concepts/issues.

Other Java's omissions are related to methods / subprograms:

- Ability of a method to update a parameter from a primitive type
- Ability to constrain as read-only a method's access to the fields in an object passed as a parameter
- Methods themselves as parameters or data objects
- Nested methods
- "Named parameter associations", default values for parameters
- Overloading based on function result type
- Overloading of operator symbols

The simplifications introduced by these omissions come at a price. If a method is to update, say, an `int` (which can easily be done in Ada procedure through an `out` or `in out` parameter) then two approaches are feasible: either define the method to return the `int` as a result, or else declare a class with an `int` field and have the method take a parameter of this class. The first approach is not applicable if the method already returns a non-void value, and the second approach is somewhat clumsy and requires heap usage.

In Java there is no way to enforce read-only access to a parameter that is of a class type; through the formal parameter the method can update the value of any (non-final) field. In Ada a parameter can be specified as "`in`", or alternatively a named access-to-constant type may be used.

Simulating a method as a parameter or a data object entails the following steps:

- Declare an interface with one member, an abstract method with the relevant signature
- Declare a class that implements the interface through the method in question
- Construct an object of this class

The effect is a reference to an object that contains the desired method as an instance member. However, it is both conceptually and implementationally more complex than the equivalent technique in Ada (an access-to-subprogram type, and the `'Access` attribute).

Java does not have general block structure: a method may not be declared local to another method. It may be declared in an inner class that is local to a method, but that introduces some conceptual complexity. From an instructional viewpoint, the traditional Algol 60-style block structure would be the preferred mechanism to present.

The absence of named parameter associations is another language simplification that results in development complication. Although having a relatively small syntactic and semantic cost, this feature would have a large payoff in increasing a program's understandability.

The inability to overload a method based on the return value implies that there may be independently-developed interfaces that cannot be simultaneously implemented by the same class, since they may declare methods with the same signature that differ only in their result type.

The inability to overload operator symbols implies a sacrifice in readability for classes that model mathematical constructs.

Some further omissions are somewhat less critical in a university course:

- Goto statement
- Exponentiation operator
- Array functionality (slicing; allowing a lower bound other than 0)
- Aggregates
- Low-level facilities

However, other omissions are more significant

- Generic units
- Interfacing to code or data in other languages

The effect of Java's lack of generic units (templates) will be described below.

Java has nothing close to Ada's support for interfacing to foreign code. Java "native methods" are far weaker and less portable than Ada's interfacing facilities.

In a presentation at OOPSLA '98 [Steele98], Guy Steele Jr. (one of the designers of Java) cited three major omissions from Java as decisions he found hindering the language's usability: "light-weight" (strongly-typed) user-defined numeric types, operator symbol overloading, and generic templates. These definitional facilities would allow the language to "grow" without needing an amalgam of *ad hoc* features. As Steele puts it, "We need to put tools for language growth in the hands of the users" [Steele98, p. 14]. Ada has

been abiding by this principle since its inception, and it supplies each of the three facilities that Steele has listed.

3.2.2 Java features that Ada lacks

Java has a somewhat more flexible OO model than Ada, as will be described below.

Exceptions in Java are more general than in Ada. In Java, a constructor for an exception class can be parameterized; when a program throws an exception, it can construct a new exception instance that contains a field reflecting relevant information from the point where the exception was thrown. A handler can subsequently retrieve the contents of this field. Ada provides more restricted functionality, through the `Raise_Exception` procedure that can take a `String` parameter, and the `Exception_Message` function that can be called from a handler to retrieve the `String` that was passed. Java also allows the programmer to supply a `finally` clause for unconditional cleanup at the end of a block, regardless of whether an exception was raised.

The Java primitive types include a 64-bit integer type (`long`) and a 64-bit floating-point type (`double`). Whether these are provided in Ada is implementation dependent.

Java provides a `continue` statement as part of its loop control; this would need to be modeled in Ada through other constructs.

3.2.3 Conclusions on language expressiveness

Although there are a few areas where Java offers more flexibility, Ada clearly provides more comprehensive features, especially in the realm of data structuring that is so critical in a foundations course. Several of Java's omissions would ordinarily classify as fatal flaws, and were it not for the publicity and excitement surrounding Java it is unlikely that the language would be drawing such attention at present from academia.

3.3 Encouragement of sound software engineering

Software engineering comprises a collection of principles and techniques that support the predictable development of correct, adaptable, and efficient systems. Although the relative role of a programming language versus other factors (such as the quality of the compiler and supporting tools, the development methodology, the personnel's talent, the project management approach) has long been the subject of debate, it is clear that a language's features can either inhibit or promote good software engineering.

3.3.1 Language safety

Java and Ada both place a high priority on this goal but have addressed it through difference approaches. Java omits features that could lead to run-time insecurities; thus no explicit pointers, no "unchecked conversions", etc. Ada includes such features but makes it explicit in the source program when they are being used. For example, to defeat the language's type checking one must explicitly "with" the predefined

`Ada.Unchecked_Conversion` generic function. Ada also supplies a directive, `pragma Restrictions`, that identifies features that are not used; a safety-critical application can employ this pragma to effectively restrict the language features to a safe subset.

Java prevents reading non-initialized data. In the case of a local variable in a block, the compiler will reject the program if it cannot prove that a variable is assigned before it is referenced. (This is of course in general an unsolvable problem; the language rules are overly conservative in that some programs will be rejected even though at run-time the variable is in fact set before being used.) For all other variables – static or instance fields, or array components – a default initialization is used: `false` for `boolean`, zero for numeric types, and null for references. Ada prevents reading non-initialized access values (since **`null`** is the default), but for other types an attempt to read the value of a non-initialized variable is a bounded error.

Some aspects of Java safety stem from the JVM execution environment versus the language itself; for example, the invocation of the class verifier on "untrusted" class files. This tool applies, however, whether the source program for the class file is written in Java or in another language such as Ada.

3.3.2 Compile-time and run-time checking

Although both languages can claim strong typing, Ada allows finer type differentiation and thus stronger typing than Java. For example, an Ada program can define distinct types, all with the same underlying representation such as 32-bit signed integers. In Java, as noted earlier, these would all be of the type `int`; defining a new type requires a class whose member is of that type, but this imposes the overhead of heap usage and garbage collection.

Both languages protect against run-time errors through checking that can raise/throw an exception. However, and somewhat curiously, Java's rules for integer arithmetic specify "wrap around" behavior versus throwing an exception when an overflow occurs. It is surprising and error prone that, for a signed integral type, adding two positive integers can yield a negative result.

Ada allows the programmer to include range constraints on object declarations; Java does not. This omission simplifies the language semantics but complicates the programmer's job and will likely entail run-time costs (for example, invoking a programmer-supplied function that checks whether a constraint is satisfied). In Ada such checks are implicitly generated by the compiler when needed and can often be eliminated by an optimizer. The explicit function call in Java will be much more difficult for an optimizer to remove.

3.3.3 Readability/maintainability

A number of readability-related topics were described earlier, in connection with learnability. In general, Java's C syntactic foundation has well-recognized drawbacks that can cause problems (but, lacking pointers, at least Java does not inherit the C confusion of pointers and arrays).

In Ada the dependence of one compilation unit on others is always explicit and immediately clear, either in "with" clauses clause at the beginning of the "client" unit or through the syntax of a child unit's name. In Java, an externally defined class can simply be referenced (by its "fully qualified" name, including its package) by a client; there is no need for an import statement. Thus it is not as easy in Java for a human reader to know which external classes are required; one must peruse the entire source file.

A Java method or constructor must include a "`throws`" clause identifying the classes of any "checked" exceptions that it might propagate, regardless of whether the propagation is direct via a `throw` statement or indirect via a method that it calls. Ada does not have an analogous feature; a subprogram specification reflects how the subprogram returns normally (`out`, `in out`, and `access` parameters, and function result type) but not how it returns abnormally. In principle Java's additional detail is helpful to the program reader, but whether this holds in practice is subject to debate. The information is not necessarily complete; for example, `RuntimeException` (which covers index violations, etc.) is an "unchecked" exception class and thus need not be specified. Moreover, since an overriding method in a subclass is not allowed to throw more exceptions than are permitted by the overridden method's `throws` clause, some notational awkwardness may result. The subclass's implementation of the method must handle any (checked) exceptions that are specified in `throws` clauses of methods that it calls but that are not included in its `throws` clause of the overridden version in the superclass. Further, there are some loopholes in Java's rules. The `newInstance` method from class `Class` invokes a constructor; if that constructor throws an exception that is not caught by `newInstance`, then the same exception is propagated even though its class is not specified on `newInstance`'s throws clause.

In Java, a class includes not only the elements that are needed by clients or subclasses, but also private members and the code comprising the bodies for all of the methods. Ada provides a syntactic separation between a unit's specification and its body, enforced with rules that prohibit unit bodies from appearing in a package specification. The benefits of this separation are clarity to the reader/maintainer, who can immediately identify a module's interface without needing a tool to strip out irrelevant detail, and reduced compilation costs. In general, compiling a unit does not require any information from the bodies of "with"ed units, and thus a change to the body of a unit does not make a client unit obsolete. In contrast, when the JDK's `javac` tool compiles a source file, if this file references a class from another file that has been modified but not recompiled, then this second file will be automatically recompiled. (The Java language specification simply states [GJS96, p. 237]: "Java development tools should support automatic recompilation as necessary whenever source code is available.") The price that Ada pays for the specification / body separation is some semantic complexity, including the introduction of elaboration order issues.

3.3.4 Encapsulation

Encapsulation involves localizing the placement of program entities so that the effect of changes can be predicted and reduced. One element of encapsulation, the physical separation of a module's specification from its body, was just discussed. Another aspect is the control over the visibility of entities defined in a module. Ada provides three principle perspectives, based on the placement of a declaration:

- An entity declared in the visible part of a package specification is visible to "client" units (i.e., units that "with" the package), to child units, and to the package's implementation (its private part and its body)
- An entity declared in the private part of a package specification is visible only within its own implementation and within the implementation of child units
- An entity declared in a package body is visible only within that package body (and thus also to subunits)

Java provides roughly the same functionality, but through access modifiers on the declarations of members and constructors. The location of a declaration in an Ada package (visible part, private part, body) models the accessibility of the corresponding method or static field in Java (`public`, `protected`, and `private`, respectively). There is no direct Ada analog to Java's "package" accessibility. Moreover, modeling a Java private per-instance data member in Ada requires some circumlocution: a tagged private type with a component that is an access value to an incomplete type whose full declaration is in the package body.

Another aspect of Java's encapsulation model is the ability to specify an entity as `final`, implying that its properties are frozen at its declaration. If a per-instance method in a class is declared final, then each subclass inherits the method's implementation and is not allowed to override it. (The application of `final` to a static method makes no sense semantically, since static methods are not inherited, but is permitted.) If a class itself is declared `final`, then no subclasses are allowed. If a variable is declared `final`, then it is a constant after its initialization.

The application of `final` to a method or class enables certain optimizations; for example, the invocation of a `final` method can be compiled with static versus dynamic binding, since the called method is the same for each class in the hierarchy.

Java's notion of "final" is not really applicable in Ada's semantic model, except for the concept of a final variable, which directly corresponds to an Ada constant (or, in the case of a "blank final", to an Ada discriminant).

3.3.5 Object-Oriented Programming

A detailed comparison of the OOP support in Ada and Java may be found in [Bro97]. Here are some of the major points:

- In Java a class is both a data template and a compilable module; Ada provides distinct features for these roles, the tagged type and the package. One effect of this difference is that Java has a considerably more concise and "lightweight" notation than Ada for expressing classes and inheritance. Another effect is on the syntax for method invocation. In Java, invoking a method `foo()` on an object reference `ref` in Java uses "dot" notation `ref.foo()` whereas in Ada the analogous form `foo(ref.all)` passes the object as an explicit parameter.
- In Java single inheritance is obtained by declaring a subclass that extends at most one superclass; if none is specified then the "root" class `Object` is extended by default. In Ada single inheritance is obtained by declaring a type that derives from a tagged type, and there is no "ultimate ancestral type" analogous to Java's class `Object`.
- In Java multiple inheritance is modeled by declaring a class that implements the relevant interfaces. In Ada multiple inheritance is modeled by several features: for example a "mixin" corresponds to a generic package that takes a formal tagged type parameter and extends it with the necessary components and operations.
- Polymorphism in Java is obtained by declaring a variable of a class type; it may then reference an object from that class or any of its subclasses. The equivalent effect in Ada is achieved by declaring a variable of an access-to-class-wide type.
- Dynamic binding in Java occurs when a method is invoked on a reference to an object, except when the method is `private` or `final`, or when it is invoked on the special reference variable `super`. Dynamic binding in Ada occurs when an actual parameter is of a class-wide type and the subprogram is a primitive subprogram for the root type of the actual parameter.
- The use of pointers is implicit in Java and explicit in Ada. To avoid storage leakage in Ada either the implementation needs to provide garbage collection, or the class author needs to declare a controlled type with appropriate operations, or the class user needs to employ unchecked deallocation.
- Since assignment in Java has reference semantics, the class author needs to decide whether and how to provide cloning. The analogous Ada functionality is a controlled type with overridden versions of `Finalize` and `Adjust`.
- Both Java and Ada support abstract classes / tagged types and abstract methods / subprograms.
- Classes in Java are bona fide objects (of the class `Class`), and Java has a rich set of facilities for run-time type interrogation. Ada was designed with the view that types are compile-known entities, and thus its support for additional generality is fairly modest (for example, the package `Ada.Tags`).

In summary, both languages provide Object-Oriented Programming features that are roughly equivalent in functionality and in any event more than adequate for purposes of a foundations course. Java has a more succinct and traditional OO notation, with a straightforward approach to simulating multiple inheritance. Ada is more general than Java in some areas (for example by allowing dynamic binding based on function result, and by allowing the same subprogram to be invoked either statically or dynamically based on how the call is written). However its approach to multiple inheritance is somewhat more complex than Java's, and OO programs in Ada are more verbose than in Java.

3.3.6 Reuse

Generic modules

One of the key notions of reuse is the ability to use the same code for multiple purposes, varying only some property that is specified at the point of reuse. A typical example is a "container" data type such as a stack

or a queue. The operations on an object of such a type (for example, inserting an element or removing an element) are independent of the type of the element. It would be tedious and error prone if the programmer needed to copy source files and modify them in order to reuse the code.

Ada was the first widely-used language to address this issue; that is one of the main purposes of the generic facility. A stack or queue type can be defined by a generic package parameterized by the element type; the package will declare the necessary operations and provide an implementation that is independent of the specific properties of the element type. Reusing the generic package entails instantiating it with the desired element type. A beneficial side effect of this mechanism is that any given data object of the resulting container type is known to comprise only elements that are all of the element type supplied at the instantiation. (If heterogeneity is desired, the generic package can be instantiated with an access-to-class-wide type).

Java does not include a facility for generic units (or "templates" as they would be known in C++). Some of this functionality can be approximated in Java through use of the "root" class `Object` as element type, but with much less compile-time protection than with Ada generics. There is no way to ensure at compile time that a container of `Object` values comprises components having one specific type; if the programmer wishes to enforce such a restriction this must be done with run-time checks. The result is code that is less clear and less efficient than the Ada equivalent.

Binary compatibility

Another aspect of reuse is the "revision problem". Software components do not stay unchanged; their authors make revisions over time to remove bugs, improve performance, and so on. In a typical software shop, this is a known issue and is handled with appropriate configuration management tools. Java, however, takes a unique view on this issue, because of the distributed environment (the Internet) for which it has been designed and promoted. The author of a given class does not know who will be using the component and has no access to client or subclass source files, and in the other direction the user of a class generally has only the binary class file and not the source version. If Java's goal of the Internet as a development platform is to be fulfilled, then the language must promote *binary compatibility*. As far as is possible, client code that links with one version of a class file should link with a revised version, without error and without needing to be recompiled. The Java language specification identifies minimal requirements for binary compatibility; for example, adding a new member to a class. Interestingly, binary compatibility is not the same as source compatibility. Class files may link correctly even though their corresponding source files do not compile without change [GJS96, p. 241].

Ada is more concerned with preventing version skew and reducing (re)compilation costs in a traditional development environment than with binary compatibility across the Internet.

3.3.7 Efficiency

Run-time efficiency was a major objective of Ada and dictated a conservative (stack-based) run-time model. Ada was designed to support embedded, real-time applications, which require knowing the maximum time that any operation will require, and ensuring that space is not exhausted during program execution.

Although Java was originally developed to program hand-held devices with embedded software, and although there has been some recent work to apply Java to real-time systems, its reliance on heap management and automatic garbage collection introduce an intrinsic overhead compared with stack-based languages such as Ada. Run-time efficiency is admittedly not the most important quality for a language used in teaching, but on the other hand it would be a mistake for an instructor to ignore this issue. Designing efficient algorithms is difficult and is sensitive to data structure choice; a language such as Ada that offers both stack and heap-based data allows the opportunity to compare and contrast different approaches. In Java the choices are much more limited.

3.3.8 Conclusions on Software Engineering Support

As a modern, OO language Java addresses many software engineering concerns that computer science students need to be aware of. In some cases (such as binary compatibility and protection against reading non-initialized data) Java goes further than almost any other language. However, Ada too was designed with software engineering as a primary goal. It offers better support than Java in areas that are more germane to a foundation course, for example providing a standard generic mechanism, stronger type checking, and an efficient stack-based run-time model.

3.4 Support for concurrent programming

A more complete comparison of Java threads and Ada tasking may be found in [Bro98].

Ada and Java have taken quite different approaches to supporting concurrency. Ada provides a general, high-level model based on explicit communication (the rendezvous), and a structured approach to mutual exclusion (protected objects), while also supplying lower-level mechanisms and specific scheduling semantics and control that may be needed for real-time and other applications. In contrast, Java's approach relies on the classical monitor construct for mutual exclusion, and "pulsed" / "broadcast" signals for thread synchronization and communication. Java is thus susceptible to the well-known "nested monitor" deadlock, if synchronized methods from different objects invoke each other. This will not occur in Ada for implementations that support `Ceiling_Locking`: a task executing a protected operation will be at a priority higher than any other task that could call a protected operation on the same object, hence it will not be preempted by such a task. Java's signal-oriented approach to communication is error-prone and may entail context switches that are not needed in Ada. Moreover, the weak semantics for priorities in Java means that thread scheduling is implementation dependent.

Although certain elements of Java's thread model provide functionality not found in Ada (for example, thread groups), in general Ada's approach to concurrency is more reliable, more portable, and more efficient than Java's.

3.5 API functionality

Java has an impressively extensive class library, covering a number of areas that are not in the Ada language standard (or in any other language standard). Examples include the Abstract Windowing Toolkit ("AWT"), Swing, applets, networking, database connectivity, internationalization, and security. While only a fraction of these are relevant in a foundation course, their presence will be useful in upper-level courses that explore specialized topics.

In its specialized needs annexes, Ada does offer some standard packages with functionality not founded in the Java API. Moreover, the Java API is accessible to Ada programs through Ada's interfacing facilities, and several compiler vendors including Aonix and Intermetrics implement the necessary interfacing pragmas. However, and not surprisingly, someone familiar with the Java API will find it simpler to access from Java than from Ada.

4 External Factors

A perfect language with non-existent or buggy compilers would be only of theoretical interest. This section looks at selection factors beyond those related to language features.

4.1 Availability of tools that are high quality but inexpensive

A university environment has somewhat different criteria for a compilation environment than an organization that needs to develop production quality software. Budgetary constraints often narrow down the selections to those whose tools are affordable, and run-time efficiency is not as important as compile-time performance, good error diagnostics, and a familiar or "user friendly" environment.

Ada 83 failed to make major inroads into academia, in part because compiler developers tended to look at this sector as a less-than-profitable market versus as an opportunity for engendering Ada-friendly attitudes. Ada 95 did not repeat this mistake. A free Ada compiler based on the GNU gcc technology ("GNAT") was developed at NYU and released at the same time as the language standard. Ada Core Technologies, Inc., continues to upgrade this compiler and makes a public version openly available on the Web. Other vendors are also distributing free or low-cost versions of their Ada products, to encourage their adoption by universities.

The free Ada offerings are competitive with Sun's Java Development Kit, and thus compiler availability and affordability are not significantly different for the two languages. In fact, the intuitive user interface offered by several Ada vendors may be an attractive alternative to Java development environments. As expressed by Culwin [Cul98, p. 33]:

The absence of any commercial I[nteractive] D[evelopment] E[nvironment] that is suitable for novice developers is another expressed concern. Although all major tools manufacturers have produced commercial quality IDEs, and have made them freely available to educators for evaluation, none of these are seen as suitable for novice use due to their inherent complexity.

4.2 Availability of good textbooks

Although there is a large choice of textbooks for Java, most are aimed at either professional programmers or else students familiar with C or C++. Only recently have texts become available that are appropriate for introductory Computer Science courses. This delay is not completely surprising, since the language's OO-centricity implies that an author cannot cover topics in the same way as for C, Pascal, or C++. In the past, authors have produced textbooks for new languages by making adaptations to existing texts. This approach does not work with Java; because of the language's object orientation and other unique attributes, textbooks needed to be written "from scratch". Moreover, as noted by Culwin [Cul98], just as some texts are getting published, they are becoming obsolete because of revisions to the language or the API.

The overall number of Ada-related books is smaller than for Java but larger than one might anticipate – Feldman [Fel98, p. 573] cites 17 published books since 1995 with Ada 95 as their focus. Of these, five texts are oriented towards first-year students, for example [FK96] and [Ska97].

In summary, there is a more-than-adequate variety of high-quality Ada texts available for use at colleges and universities. Since Ada is a stable international standard, these books are less likely to become obsolete than texts on Java.

4.3 Portability / standardization

One of Java's main claims is code portability, sometimes advertised as "Write Once, Run Everywhere", but this applies at several levels. One is the class file representation, which is portable to (i.e., may be loaded and run on) any platform with an implementation of the JVM. However, this effect is independent of the source language; any language that can be compiled into "byte codes", including Ada, can reap the portability benefits of the JVM. An early project at Intermetrics named Applet Magic demonstrated that compiling Ada to Java bytecodes was feasible. More recently, Ada Core Technologies' JGNAT product has shown that a full, validatable Ada compiler can be targeted to the JVM, thus providing the same level of class file portability as the Java language.

The other level at which portability applies is the source code. This is addressed by Java language semantics, which attempts to eliminate implementation-dependent behavior. Here are several examples:

- Evaluation order in expressions is left-to-right
- The size and behavior for each primitive type is fully specified, including the requirement for `float` and `double` to supply IEEE-754 arithmetic
- All variables must be initialized before they are used

However, the goal of complete implementation independence is compromised in several places:

- The use of priorities in thread scheduling is implementation dependent
- Current implementations of the AWT show platform-specific behavior (not just a different "look and feel")
- Efficiency problems with IEEE arithmetic on certain hardware have led some to request relaxing the requirements on floating point arithmetic
- The time at which finalizers run depends on the Garbage Collection algorithm employed by the implementation

Moreover, Sun's approach to Java standardization has raised the concern that there will not be an opportunity for the necessary technical review. This reaction is reasonable, since releases of both the language and the API have shown some volatility. Several features in Java V1.0 were removed in V1.1 (for example, one of the forms of the import statement, and one of the access modifiers), and the AWT event model was completely and incompatibly revised. There are indications that things are now settling down, but even so the Sun/Microsoft feud has users wondering what the Java language actually will be, independent of what happens with ISO. The risk with most standardization efforts is that it will come either too early, freezing decisions before they have been fully analyzed, or too late, after incompatible implementations have emerged and after mischievous vendors have introduced proprietary enhancements. Java may encounter both of these drawbacks.

In contrast to the situation with Java, both Ada 83's and Ada 95's standardization efforts were contracted and managed projects, with the involvement and review of the full international community. Although Ada permits implementation-dependent decisions in a number of areas (for example the choice of "by copy" or "by reference" for a formal parameter of an aggregate type), experience with portability of Ada source code has been extremely favorable. Vendors did not rush out with premature implementations of the language, and at least with Ada 95, good quality compilers were available not long after the publication of the standard.

Portability might not seem like a high priority for a language to be used at a university, but given the range of equipment and operating systems that are typically present, this goal is important. Ada has a known "track record", with vendors who are in consensus on the need for and role of a language standard. Java's future is less certain, and the politically-charged bickering among some of the giants of the computer industry is not reassuring.

4.4 Status as "hot" technology

Java's emergence and on-going maturation form one of the most impressive developments in the history of software technology. Although it is too early to predict the scope of its influence, and although there have been some well-publicized commercial failures of Java projects, Java offers an inspired view of distributed computing, an elegant OO model, and a way to use the Internet as an execution platform. Universities

rightly recognize the demand for Java and see the need to effectively incorporate the language and its surrounding technological elements into their computer science curricula.

Though Ada 95 became a standard at roughly the same time that Java first emerged, Ada has received a somewhat quieter reception in the software community. Therein lies some irony. Java is sometimes given credit as a revolutionary development, but some of the major components of the Java technology are not at all new. The language borrowed heavily from Smalltalk, Modula-3, and C++, and the JVM idea is similar to the UCSD Pascal "P-Code" approach from the early 1980's. In contrast, people unfamiliar with Ada 95 probably imagine a minor upgrade of the original Ada 83 language. Ada 95 is in fact a nearly 100% upward-compatible extension, but its enhancements are significant and in several areas (such as its support for interfacing) the language revision broke new ground. In some important ways Ada 95 is a more revolutionary departure from its predecessor, than Java is from its antecedents.

It is outside the scope of this paper to address the reasons why Ada has not caught the software public eye in the same way as Java or other technologies. Nonetheless, it is worth noting that Ada is increasingly becoming the language of choice in a number of areas, including high-integrity and safety-critical systems. And although its development was sponsored by the U.S. Department of Defense, Ada continues to attract users from the commercial sector. (A snapshot of some of these uses is provided by Feldman [Fel98, pp. 571ff].)

A problem with settling on a current "hot" language as the basis of a computer science curriculum is that in time the enthusiasm will dampen. If the language does not meet the essential criteria for teaching a foundations course then at some point in the future the instructor or the department will need to decide whether to continue with it or go to the trouble and expense of making a change. It is better to make the more appropriate decision at the outset.

5 Conclusions

When analyzed against the criteria for a foundation language, in almost all areas Ada is stronger than Java. Ada has considerable more functionality, including basic data structuring features that are essential for students to understand. Ada is methodology neutral and has a traditional stack-based run-time model; Java is built around OOP, makes extensive use of the heap, and includes automatic garbage collection. Teaching Java effectively is quite difficult, since OOP topics must be introduced before students understand their purpose. Java's heap requirements rule out some areas (such as low-level programming) that a foundation language should support. Ada supplies strongly-typed generics; Java can simulate generics to some extent but requires programmer-supplied run-time checking to preserve type safety. Ada's concurrency model is better defined, more reliable, and more efficient than Java's threads. Ada is currently a stable international standard; Java's status is less certain.

Java's main strengths are its well-designed OOP approach, its safety, its comprehensive API, and the interest that it is attracting in the software community.

In summary, some of the very language design decisions intended to attract software developers to Java (relative simplicity, the absence of pointers) become disadvantages for a foundation Computer Science course. There is a place for Java in the curriculum, *after* students have learned Ada. As has been discovered by a large and growing number of colleges and universities [Feldman], using Ada as the foundations language offers students appropriate exposure to a range of language features and concepts, in a methodology-neutral fashion, with excellent support from available compilation environments and textbooks.

Appendix A: Java Language Summary

A.1 General-purpose features

At one level Java can be regarded as a general-purpose programming language with traditional support for software development. Its features in this area include the following:

- simple control structures and expression syntax based on C and C++
- a set of primitive data types for manipulating numeric, logical, and character data
- a facility for constructing dynamically allocated arrays, with bounds checking on indexing
- a facility for code modularization ("methods")
- limited block structure, allowing the declaration of variables, but not methods, local to a method
- exception handling

The primitive data types in Java are `boolean`, `byte`, `char`, `short`, `int`, `long`, `float`, and `double`. Java is distinctive in specifying the exact sizes and properties of these primitive types. For example, an `int` is a 2's complement signed 32-bit quantity, with "wrap around" when an operation overflows. The floating-point types are defined with IEEE-754 semantics, implying that results and operands may be signed zeroes, signed infinities, and NaN ("Not a Number"). Unlike Ada, Java does not allow range constraints to be specified for variables from numeric types.

Java's `char` type is 16-bit Unicode. Its source representation is likewise based on Unicode, although external files may be stored in 8-bit format with conversion on the fly during file loading. Like C and C++ but unlike Ada, Java's treatment of identifiers is case sensitive.

As has been discussed above (Section 3.2.1), Java lacks a number of data structuring facilities found in traditional languages.

Java's execution model is based on a run-time stack (more generally, one stack per created thread). When a method is called, its parameters and local variables are stored on the (calling thread's) stack. For a parameter or variable of a primitive type, the stack contains the variable's value directly. In all other cases

the stack contains a *reference* that can designate an allocated object of the item's type. Parameter passing is thus always "call by value", with the value of the actual parameter copied to the formal parameter at the point of call. However, since objects are represented indirectly, the effect is to copy a reference and thus the formal and actual parameters refer to the same object.

A.2 Object Orientation

Although Java's syntax is based on C and C++, Java's OO semantics is closer to Smalltalk. A Java *class* serves three purposes: a compilable module, a type for polymorphic variables, and a template for dynamically allocated objects. A class has *members*: methods, fields (variables), and nested classes. A member may be *static* (shared by each instance) or per-instance; the latter is the default. A class typically declares one or more *constructors*, which provide parameterized initialization.

A class is a single syntactic unit; it is not separated into a "specification" and a "body".

An *interface*, based on a feature in Objective C, is a restricted form of class with no variable state and no method implementations. *Inheritance* involves extending a class and/or implementing an interface. A class is allowed to extend exactly one superclass (the class `Object` by default) but may implement an arbitrary number of interfaces. A variable declared of a class `K` or interface `I` is *polymorphic*; it may reference a constructed (dynamically allocated) object from `K`, or from any class that directly or indirectly either extends `K` or implements `I`.

A class member may be specified as *final*. A final variable is a constant; a final method cannot be overridden; a final class cannot be extended.

A.3 "Programming in the large"

In addition to its OOP features, Java supports large system construction in several ways: method overloading, nested classes, class repositories known as *packages*, and encapsulation (name accessibility control).

Methods may be overloaded based on formal parameter types; however, Java does not permit method overloading based on result type, nor does it allow operator symbols to be overloaded.

The unit of program composition is the class, but this raises the possibility of "namespace pollution" when large numbers of classes are developed. Java solves this in two ways. First, classes may be nested, a facility introduced in Java 1.1. Thus a class may declare other classes as members; since the inner classes are referenced using "dot" notation, their names do not conflict with any other classes. Second, Java provides a namespace management feature, the *package*, as a repository for classes. Despite the similarity of names, the term "package" has different meanings in Ada and Java. An Ada package is a syntactic and semantic construct, a compatible unit. In contrast, a Java package is an open-ended host-environment facility, generally a directory, which contains compiled class files. Java's `package` statement identifies

the package that will serve as the destination for the unit being compiled and the source for classes referenced by this unit. If a source file does not supply an explicit `package` statement, then its classes go into an "unnamed" package, by default the same directory as the site of the `.java` file.

Class members and constructors may be specified with access modifiers that affect their visibility and thus their degree of encapsulation. A *public* entity is visible wherever the class is accessible. A *protected* entity is visible to all subclasses. A *private* entity is visible only within the class itself. In the absence of a specific access modifier, an entity is visible only within the package that contains the enclosing class.

Java's predefined environment is structured as a collection of packages, including `java.lang`, `java.util`, and many others. Package names may be nested arbitrarily; generally a name such as `alpha.beta.gamma` corresponds to a subdirectory `alpha/beta/gamma` on a host machine.

If a Java program needs to access a class `K` defined in package `P`, then in general it must do one of three things:

- include the statement `import P.*;` which allows each class in `P` to be referenced by its "simple name" (i.e, not including the package name as prefix)
- include the import statement `import P.K;` which allows this class to be referenced simply as `K`
- specify the "fully qualified name" `P.K` at each reference to the class

The exception to this rule is the general purpose package `java.lang`, which every Java program implicitly imports and whose classes are therefore automatically accessible without the package name prefix.

Like Ada, but unlike C and C++, Java does not supply a preprocessor. Higher level and better structured mechanisms provide the needed functionality more reliably than preprocessor directives.

A.4 Concurrency support

Java offers built-in support for multi-threaded programs, with user-definable threads that can communicate through objects whose methods are explicitly marked as `synchronized`. An object that contains `synchronized` methods has a "lock" that enforces mutually exclusive access, with calling threads suspended waiting for the lock as long as any `synchronized` method is being executed by some other thread. Java's thread model is based on its OOP features; the programmer needs to extend the `Thread` class or implement the `Runnable` interface, and override the `run()` method to specify the desired behavior.

A.5 Java Example

The following example illustrates a class hierarchy comprising the classes `Point` and `ColoredPoint`:

```
class Point {
   int x, y;
   private static int numPts = 0;  // accessible only within Point
```

```
   Point(int x, int y){ // constructor
      numPts++;
      this.x = x;
      this.y = y;
   }
   boolean onDiagonal(){
      return (x == y);
      // returns true if this is on the 45° line
   }

   public void put(){
       System.out.print(" numPts=" + numPts);
       System.out.print( " x=" + x);
       System.out.print( " y=" + y );
   }
}

class ColoredPoint extends Point{
   String color = "transparent";
   ColoredPoint(int x, int y, String color){ // constructor
      super(x, y);
      this.color = color;
   }

   public void resetColor(){
     color = "transparent";
   }
   public void put(){
     super.put();
     System.out.print( " color=" + color );
   }
}
```

`ColoredPoint` inherits `onDiagonal()`, overrides `put()`, and introduces a new method `resetColor()` and a new constructor. Here is a code fragment showing how these might be used in practice:

```
{ Point p1, p2;
  ColoredPoint p3;
  p1 = new Point( 10, 10 );
  p1.put();  // Displays: numPts=1 x=10 y=10
  p2 = new ColoredPoint( 20, 30, "red" );
  p3 = new ColoredPoint( 40, 30, "blue" );
  p2.put();  // Displays: numPts=3 x=20 y=30 color=red
  p1=p3;            // OK
  p1.resetColor();  // Illegal
  ((ColoredPoint)p1).resetColor(); // OK
  p3.resetColor();  // OK
}
```

In the above fragment, `p1` is polymorphic, and the method invocations are bound dynamically. The cast of `p1` to `ColoredPoint` is needed in order to invoke the `resetColor` method; a run-time check will test that `p1` references an object of a class in the inheritance hierarchy rooted at `ColoredPoint`.

References

[AG98] K. Arnold and J. Gosling, *The Java™ Programming Language (2nd edition)*; Addison-Wesley, 1998.

[Bro97] B. Brosgol; "A Comparison of the Object-Oriented Features of Ada 95 and Java™", *Proc. Tri-Ada '97*, ACM SIGAda, 1997.

[Bro98] B. Brosgol; "A Comparison of the Concurrency Features of Ada and Java", *Proc. SIGAda '98*, ACM SIGAda, 1998.

[Cul98] F. Culwin, "Editorial – Justifying Java?", in *ACM SIGPLAN Notices*, Vol. 33, no. 4, April 1998.

[Fel98] M. Feldman, "Ada 95 in Context", in *Handbook of Programming Languages, Vol. I: Object-Oriented Programming Languages* (P. H. Salus, editor); Macmillan Technical Publishing; 1998.

[FK96] M. Feldman and E. Koffman, *Ada 95 Problem Solving and Program Design*; Addison-Wesley, 1996.

[Gib98] J. Gibbons, "Structured programming in Java", in *ACM SIGPLAN Notices*, Vol. 33, no. 4, April 1998.

[GJS96] J. Gosling, B. Joy, and G. Steele; *The Java™ Language Specification*, Addison-Wesley, 1996.

[Int95] Intermetrics, Inc.; Ada Reference Manual - Language and Standard Libraries, ISO/IEC 8652:1995.

[Mar98] P. Martin, "Java, the good, the bad and the ugly", in *ACM SIGPLAN Notices*, Vol. 33, no. 4, April 1998.

[Ska97] J. Skansholm, *Ada from the Beginning* (3rd edition); Addison-Wesley, 1997.

[Steele98] G. L. Steele, Jr.; "Growing a Language", OOPSLA '98

Web sites

[AdaIC] Ada Information Clearinghouse (this web site is now administered by the Ada Resource Association); `http://www.adaic.org`

[AdaResourceAssoc] Ada Resource Association; `http://www.adaresource.org`

[Feldman] M. Feldman, *Ada as a Foundation Programming Language*, February 1997; `http://www.seas.gwu.edu/faculty/mfeldman/CS1-2.html`

[SIGAda] ACM Special Interest Group on Ada; `http://www.acm.org/sigada/`

[Sun] Sun, *The Java™ Language: an Overview*, `http://java.sun.com/docs/overviews/java/java-overview-1.html`