A Comparison of the Object-Oriented Features of Ada 95 and Java™

Benjamin M. Brosgol
Ada Core Technologies
79 Tobey Road
Belmont, MA 02478
+1.617.489.4027 (Phone)
+1.617.489.4009 (FAX)
brosgol@gnat.com (Internet)

Abstract

Ada and Java offer comparable Object-Oriented Programming ("OOP") support, but through quite different approaches in both their general philosophies and their specific features. Each language allows the programmer to define class inheritance hierarchies and to exploit encapsulation, polymorphism, and dynamic binding. Whereas OOP forms the foundation of Java's semantic model, OOP in Ada is largely orthogonal to the rest of the language. In Java it is difficult to avoid using OOP; in Ada OOP is brought in only when explicitly indicated in the program. Java is a "pure" OO language in the style of Smalltalk, with implicit pointers and implementation-supplied garbage collection. Ada is a methodology-neutral OO language in the manner of C++ , with explicit pointers and, in general, programmer-controlled versus implementation-supplied storage reclamation. Java uses OOP to capture the functionality of generics ("templates"), exception handling, multi-threading and other facilities that are not necessarily related to object orientation. Ada supplies specific features for generics, exceptions, and tasking, independent of its OO model. Java tends to provide greater flexibility with dynamic data structures and a more traditional notation for OOP. Ada tends to offer more opportunities for optimization and run-time efficiency, and greater flexibility in the choice of programming styles.

1 Introduction

Ada [In95] and Java [GJS96] both offer comprehensive support for Object-Oriented software development, but through rather different approaches and, perhaps confusingly, at times using the same terms or syntactic forms with different meanings. Since both languages promise to see significantly expanded usage over the coming years, software developers should know how the languages compare, both in general and with respect to their OO features. This paper focuses on the latter point, contrasting the two languages' OO facilities from the perspectives of semantics, expressiveness/style, and efficiency. It roughly follows the approach of an earlier paper by J. Jørgensen [Jø93] that compares Ada and C++, and it supplements the results presented by S.T. Taft [Ta96]. The reader is assumed to be familiar with Ada but not necessarily with Java.

Section 2 summarizes that main features of Java. Section 3 describes how Java and Ada capture the essential concepts of "class" and "object" and how they treat encapsulation and modularization. Section 4 compares the languages' approaches to inheritance, and Section 5 summarizes the differences in their treatment of overloading, polymorphism and dynamic binding. Section 6 describes how Java and Ada provide control over a class's fundamental operations including initialization and finalization. Sections 7 and 8 respectively compare the languages' facilities for exception handling and multi-threading, both of which are defined in Java through OO features. Section 9 looks more generally at how OOP fits into the two languages, and Section 10 offers some conclusions and provides a summary comparison table.

As a comparison of the two languages, this paper frequently will present a feature of one language and show how it can be modeled in the other. A benefit of this approach is that it eases the task of describing the relative

This paper was presented at the TRI-Ada '97 Conference in St.Louis, MO (Nov. 1997) while the author was with Aonix.

semantics. However, an acknowledged drawback is that the reader may infer that we recommend the "modeling" approach as the way to actually develop software. This in not our intent; each language has its own natural style or styles, and trying to imitate Java by translating its features into equivalent Ada, or vice versa, is not encouraged. Moreover, whether a particular feature of one language has a direct analog in the other is generally not the issue that is relevant in practice. The question is how the totality of a language's features fit together to enable reliable and robust software development.

2 Java Summary

Although it is beyond the scope of this paper to describe Java in detail, this section provides an overview of the Java technology and the basic language features. Among the references for further information on Java are [Fl97], [AG98], and [GJS96].

2.1 Java Technology Elements

Sun [Su96] has described Java 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 – Flanagan [Fl97, pp. 3ff] summarizes how Java complies with these goals – but it is useful to distinguish among three elements:

- The Java language
- The predefined Java class library
- 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 and Eiffel. Java directly supports single inheritance and also offers a partial form of multiple inheritance through a feature known as an "interface".

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 is currently attracting, 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, deal with Graphical User Interfaces. Still others, introduced in Java 1.1, 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 at present for a Java compiler to generate a so-called *class file* instead: a sequence of instructions, known as "bytecodes", that 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 manipulation

- 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 access the client machine's file system

2.2 Language Overview

2.2.1 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 heavily resembling those found in C and C++
- a set of primitive data types for manipulating numeric, logical, and character data
- a facility for constructing dynamically allocated linear indexable data structures (arrays)
- 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`, `char`, `byte`, `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 32-bit quantity, with "wrap around" when an operation overflows. The floating-point types are defined with IEEE-754 semantics [AI85], 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.

Java lacks a number of data structuring facilities found in traditional languages:

- enumeration types
- heterogeneous data structures (records / structs)
- conditional data structures (variant records / unions)

An enumeration type can be simulated by constants (static "final" variables), a record/struct can be modeled by a class, and a variant record/union can be modeled by an inheritance hierarchy.

Significantly, Java also lacks an explicit pointer facility, a restriction motivated by security concerns. As a consequence, it is not possible to obtain a pointer to a method, functionality that languages like Ada and C/C++ provide for "callbacks" in GUI programming. We show below (§4.3) how to work around this omission.

Interfacing with non-Java code is provided through methods that are specified as *native*. The implementation of a native method is supplied by platform-dependent foreign code.

2.2.2 Execution model and multi-threading

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.

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. As will be discussed below (§8), the thread model is based on Java's OOP features.

2.2.3 "Programming in the large"

Java offers a variety of support for developing large systems. It provides a full complement of Object-Oriented features, with precise control over the accessibility of member names (that is, control over encapsulation) as will be detailed below.

The unit of program composition is the class, but this raises the problem of "namespace pollution" as large numbers of classes are developed. Java solves this in two ways. First, classes may be nested, a facility introduced in the Java 1.1 release. 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 compilable unit. In contrast, a Java package is an open-ended host-environment facility, generally a directory, which contains compiled class files. Java supplies an explicit construct, the `package` statement, which allows the programmer to identify the target package for the classes being compiled. 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.

Java's predefined environment is structured as a collection of packages, including `java.lang`, `java.util`, and many others. If a Java program explicitly imports a class or package, then it can access that class, or any class in the given package, by the class's "simple name" without the package name as qualifier. The general-purpose package `java.lang` is implicitly imported by every Java program, and its classes (such as `String`) are therefore automatically accessible without the package name as 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.[†]

Java does not include a facility for generic units ("templates" as they would be known in C++). Some of this functionality can be approximated in Java through use of the "root" class `Object` defined in package `java.lang`, but with much less compile-time protection than with Ada generics.

2.3 Java Application Example

To introduce the basic language constructs, here is a simple Java application that displays its command-line arguments:

```
class DisplayArgs{
  static int numArgs;
  public static void main( String[] args ){
    numArgs = args.length;
    for (int j=0; j < numArgs; j++) {
      System.out.println( args[j] );
```

[†] K. Arnold has noted [Ar96] that in C++ a programmer wanting to reference private members of a class can subvert the language rules by simply including the preprocessor directive

```
#define private public
```

```
    } // end for
  } // end main
} //end DisplayArgs
```

The *class* is the unit of compilation in Java, and is also a data type; here it is only the compilation unit property that is being exploited. A class declares a set of *members*: in this example the class `DisplayArgs` has two members, a *field* named `numArgs` of type `int`, and a *method* named `main`.

The field `numArgs` is declared `static`, implying that there is a single copy of the data item regardless of the number of instances of its enclosing class.

The method `main` takes one parameter, `args`, an array of `String` objects. It does not return a value (thus the keyword `void`) and is invokable independent of the number of instances of its enclosing class (thus the keyword `static`). The `public` modifier implies that the method name may be referenced anywhere that its enclosing class is accessible.

There is special significance to a method named `main` that is declared `public` and `static`, and that takes a `String` array as parameter and returns `void` as its result. Such a method is known as a class's "main method." When a class is specified on the command line to the Java interpreter, its main method is invoked automatically when the class is loaded, and any arguments furnished on the command line are passed in the `args` array. For example if the user invokes the application as follows:

```
java DisplayArgs Brave new world
```

then `args[0]` is `"Brave"`, `args[1]` is `"new"`, and `args[2]` is `"world"`. Similar to C and C++, the initial element in an array is at position 0. Since the number of elements in an array is given by the array's `length` field, the range of valid indexes for an array `x` goes from `0` to `x.length-1`. Unlike C and C++, an array is not synonymous with a pointer to its initial element, and in fact there is a run-time check to ensure that the value of an index expression is within range. If not, an exception is thrown.

The class `String` is defined in the `java.lang` package. A variable of type `String` is a reference to a `String` object, and a string literal such as `"Brave"` is actually a reference to a sequence of `char` values; recall that `char` is a 16-bit type reflecting the Unicode character set. There is no notion of a "nul" character terminating a `String` value; instead, there is a method `length()` that can be applied to a `String` variable to determine the number of `char` values that it contains. `String` variable assignment results in sharing, but there are no methods that can modify the contents of a `String`. To deal with "mutable" strings, the class `StringBuffer` may be used.

The Ada and Java `String` types are thus quite different. In Ada, a `String` is a sequence of 8-bit `Character` values, and declaring a `String` requires specifying the bounds either explicitly through an index constraint or implicitly through an initialization expression; further, in an array of `String` values each element must have the same bounds. In Java a `String` is a reference to an allocated sequence of 16-bit `char` values, and a length needs to be specified when the object is allocated, not when the reference is declared. Further, as with the `args` parameter to `main`, in an array of `String` values different elements may reference strings of different lengths.

The principal processing of the main method above occurs in the `for` loop. The initialization part declares and initializes a loop-local `int` variable `j`. The test expression `j < numArgs` is evaluated before each iteration; if the value is `true` then the statement comprising the loop body is executed, otherwise the loop terminates. The loop epilog `j++`, which increments the value of `j`, is executed after each iteration.

The statement that is executed at each iteration is the method invocation

```
System.out.println( args[j] );
```

`System` is a class defined in the package `java.lang`, and `out` is a static field in this class. This field is of class `PrintStream`, and `println` is an instance method for class `PrintStream`. Unlike a static method, an instance method applies to a particular instance of a class, a reference to which is passed as an implicit parameter. The

`println` method takes a `String` parameter and returns `void`. Its effect is to send its parameter and a trailing end-of-line to the standard output stream, which is typically associated with the user's display.

The example also illustrates one of Java's comment conventions, viz. "`//`" through the next end-of-line.

3 Object and Class

Ada and Java differ in three fundamental ways with respect to their support for Object-Oriented Programming (OOP):

- The role of the class construct – combining a module with a data type, or providing separate features
- Whether pointers are implicit or explicit
- Whether garbage collection is automatic or not

Java's *class* construct serves as both a module (with control over visibility) and a data type. Moreover, Java lacks an explicit pointer facility but instead is "reference-based": as seen earlier, declaring a variable of class α reserves space only for a reference to an object. The reference, `null` by default, can designate an allocated instance from class α or from any of its subclasses. Garbage collection is automatic.

Ada supplies separate features, the *package* and the *tagged type*, to satisfy the two purposes of a class. A Java class is generally modeled by an Ada package whose specification immediately declares a tagged type. Since Ada is a traditional stack-based language where references (access values) need to be explicit, an Ada package declaring a tagged type `T` will generally declare an access-to-`T'Class` type. An implementation is permitted but not required to supply automatic garbage collection, and thus the application programmer generally needs to attend to storage reclamation through unchecked deallocation, storage pools, or controlled types.

The two languages have different vocabularies for their OO features. A Java *class* has *members*; a member is either a *field*, a *method*, or another class[†], and is either *per-class* or *per-instance*. Defining a member with an explicit modifier `static` makes it per-class, otherwise the member is per-instance.

The two languages use the term "object" in different senses. A Java object is the allocated instance that is referenced by a declared variable, whereas an Ada object is the declared variable itself. Unless otherwise indicated, we will hereafter use the term "object" in the Java sense.

As implied by the terminology, if a member field is per-class then there is exactly one copy regardless of the number of instances of the class. In contrast, there is a different copy of a per-instance field in each instance of the class.

If a member method is per-class, then it is invoked with the syntax

```
classname.methodname( parameters )
```

and its only parameters are the ones it explicitly declares. If a method is per-instance, then it is invoked with the syntax

```
ref.methodname( parameters )
```

and it takes *`ref`* as an implicit parameter named `this` which refers to the object for which the method is a member.

Like C and C++, and unlike Ada, Java requires empty parentheses for an invocation of a parameterless method. Java requires positional notation for parameters at method invocations; it does not support named associations.

[†] More strictly, a member may be a class or an interface. See §4.2 for a discussion of interfaces.

The ability to declare a class within another class (and in fact also locally within a method) was introduced in Java 1.1; in Java 1.0 a class could only be declared at the outermost level of a program.

An Ada *tagged type* has data components only, and these components are always per-instance. Java allows a per-instance field to be declared `final` (meaning that it is a constant), whereas in Ada a record component other than a discriminant is always a variable versus a constant.

A Java instance method takes an implicit parameter, `this`, which is an object reference. The corresponding Ada construct is a primitive subprogram taking an explicit parameter of the tagged type; a parameter of a tagged type is passed by reference. A Java `static` data member ("class variable") or `static` method ("class method") is modeled in Ada by a variable or subprogram declared in the same package as the tagged type.

The languages' different philosophies towards pointers lead to different tradeoffs. In Java, the implementation of objects (including arrays) through implicit references, and the existence of automatic storage reclamation, offer dynamic flexibility and notational succinctness, and free the programmer from the error-prone tasks of preventing storage leaks and avoiding dangling references. However, these same properties prevent Java from being used by itself as a systems programming language. There is no way in Java to define a type that directly models, say, a data structure comprising an integer, a sequence of characters, and some bit flags, with each field at a particular offset. Instead, one must define the data structure in another language and then resort to native methods to access the components.

In Ada the choice of indirection is always explicit in the program. If the programmer declares a `String`, then the semantic model is that storage for the array is reserved directly as part of the declaration; Ada does not allow a variable declaration such as

```
Alpha : String;
```

since the compiler would not know how much space to reserve. The benefits are flexibility of style (since the programmer, not the language, decides when indirection will be used), and run-time efficiency both in data space and time. Ada's discriminant facility allows the programmer to define a parameterized type with declaration-time control over size (number of elements in a component that is an array) and shape (which of several variants is present). Java has no such mechanism. The drawbacks to Ada's approach are some notational overhead to introduce the necessary declarations for the access types, run-time implementation complexity to deal with issues such as functions returning values from "unconstrained" types, and the need for the programmer to take appropriate measures to avoid storage leaks.

Java guarantees that each field in a class receives a default initialization based on its type: `null` for references, `false` for `boolean`, and the type's zero value otherwise. Default initial values are not assigned to methods' local variables, and the compiler will reject a program if it cannot deduce that all local variables are set before their values are referenced. Ada guarantees default initialization only for a variable from an access type (`null`), for a record component with a default initialization expression, and for a variable from a controlled type (through the `Initialize` procedure).

Both Java and Ada support *abstract classes* and *abstract methods* for such classes. The Ada terminology for these concepts is *abstract type* and *abstract operation*. In both languages an abstract class with abstract methods is to be completed when extended by a non-abstract class. As will be seen below (§4.3), an abstract class in Java can be used to work around Java's omission of a facility for passing methods as parameters.

3.2 Encapsulation and Visibility

3.2.1 Access Control

Both languages have mechanisms that enforce *encapsulation*; i.e., that control the access to declarations so that only those parts of the program with a "need to know" may reference the declared entities. Java accomplishes this with an *access control modifier* that accompanies a class or any of its members.

- A class may be declared `public`, in which case its name is accessible anywhere that its containing package is accessible. If a class is not specified as `public`, then it is accessible only from within the same package.
- A member is only accessible where its class is accessible, and an access control modifier may impose further restrictions.
  - A `public` member has the same accessibility as its containing class.
  - A `protected` member is accessible to code in the same package, and also to subclasses.
  - A `private` member is accessible only to code in the same class.
  - If no access control modifier is supplied, then the effect is known as "package" accessibility: the member is accessible only to code in the same package.

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 class that contains a 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.

3.2.2 "Final" entities

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. Ada's discriminant mechanism allows the value of a constant field to be set when an object is created; Java's "blank finals" (another feature introduced in Java 1.1) offer similar functionality.

3.2.3 Separation of interface and implementation

Perhaps surprisingly, given the otherwise careful attention paid to encapsulation, Java does not separate a class into a specification and a body. Rather, the method bodies occur physically within the class declaration, thus revealing to the user of the class more information than is needed. If one wants to make available the source code for a set of classes that reveals only the methods' signatures (and not their implementation) than a tool is needed to extract this from the compilable units.

Ada enforces a strict separation between a unit's specification and its body; the "package specs" that comprise a system's interface are legitimate compilation units and do not contain algorithmic code. The latter forms the implementation and is found in the package bodies. The price for the specification/body separation is some additional complexity in language semantics and hence also for the compiler implementation. For example, Ada is susceptible to "access before elaboration" problems, a sometimes subtle run-time error in which a subprogram is invoked before its body has been elaborated. Java allows forward references to classes and methods and thus the concept of "access before elaboration" does not arise. Sometimes the Java version of an Ada program suffering from access before elaboration will work properly, but in other cases the Java program will throw a run-time exception (for example, from a stack overflow caused by an infinite recursion, or from an attempt to dereference a null reference variable during its initialization).

3.2.4 Parameter protection

Ada provides parameter modes (`in`, `out`, `in out`) that control whether the actual parameter may be updated by the called subprogram. Thus the programmer knows by reading a subprogram's specification whether the parameters that it takes may be updated.

Java has no such mechanism: a method's implementation has read/write access to the objects denoted by its reference-type parameters. Thus in Java there is no way to tell from a method's signature whether it updates either the object denoted by its `this` parameter or the objects denoted by any other reference-type formal parameters.

Java's "call by value" semantics implies that a modification to a formal parameter of a primitive type has no effect on the actual parameter. In order to update, say, an `int`, the programmer must either declare a class with an `int` member or use an `int` array with one element. Both styles are clumsier and less efficient than Ada's approach with an `out`, `in out`, or `access` parameter.

3.2.5 Data abstraction and type differentiation

Ada's private type facility supports the software engineering principle of data abstraction: the ability to define a data type while exposing only the interface and hiding the representation. A variable of a private type is represented directly, not through a reference, and its operations are bound statically. Data abstraction in Java is part of the OO model, resulting in run-time costs for indirection, heap management, and dynamic binding.

A somewhat related facility is the ability to partition data into different types based on their operations, so that mismatches are caught at compile time. Ada's derived type and numeric type features satisfy these goals. Java does not have an analogous mechanism for its primitive types.

3.3 Modularization

Java has two main mechanisms for modularization and namespace control: the package and the class. Ada does not have a direct analog to the Java package; the way in which compilation unit names are made available to the Ada compiler is implementation dependent. On the other hand, Java does not have a feature with the functionality of Ada's *child units*, a facility that allows a hierarchical namespace for compilation units. Inner classes in Java need to be physically nested within a "top level" class and are analogous to nested packages, not child packages, in Ada.

Inner classes, although a welcome addition in Java 1.1, do introduce some complexity that can lead to programming errors. A typical mistake for a Java programmer is to reference a per-instance entity from the implementation of a static method; inner classes provide further opportunities to make this kind of error. This error is less likely in Ada, since the analog to a per-instance field is a component of a record type, which is syntactically distinct from a variable in a package.

Java allows at most one public class per source file; if present, the public class must have the same name as the file. Ada does not have analogous restrictions; a source file is not an Ada semantic construct.

3.4 Run-Time Type Interrogation

Java has comprehensive support for run-time processing of data types. The class `java.lang.Class` is the class for classes; if `K` is a class, then `K.class` is the `Class` object that represents `K`. Java 1.1 has also introduced a package `java.lang.reflect` which can be used to obtain information about a class and its members. This is the basis for the Java Beans introspection mechanism.

Ada provides the `'Tag` and `'External_Tag` attributes to represent tagged types as run-time values, but for more extensive support it is necessary to rely on auxiliary bindings and packages.

3.5 Example

The following is a simple Java class and its Ada analog. Here as well as throughout this paper, we employ Java's lexical conventions for the Java examples, and Ada lexical conventions for the Ada examples.

```
public class Point {
   public int x, y;

   public static int numPts = 0;

   public boolean onDiagonal(){
   // Returns true if this is on 45° line
      return (x == y);
   }

   public static boolean onVertical( Point p,
                                     Point q ){
   // Returns true if p and q are on a line
   //   parallel to the y-axis
      return (p.x == q.x);
   }

   public void put(){
   // Displays x and y
       System.out.println( "x = " + x );
       System.out.println( "y = " + y );
   }
}
```

```
package Point_Pckg is
  type Point is tagged
    record
      X, Y : Integer;
    end record;
  Num_Pts : Integer := 0;

  type Point_Class_Ref is access all Point'Class;

  function On_Diagonal( This : Point )
    return Boolean;
  -- Returns true if This is on 45° line

  function On_Vertical( P, Q : Point'Class )
    return Boolean;
  -- Returns true if P and Q are on a line
  --   parallel to the y-axis

  procedure Put( This : in Point );
  -- Displays X and Y components
end Point_Pckg;
```

```
with Ada.Text_IO; use Ada.Text_IO;
package body Point_Pckg is
  function On_Diagonal( This : Point )
    return Boolean is
  begin
    return This.X = This.Y;
  end On_Diagonal;

  function On_Vertical( P, Q : Point'Class )
```

```
    return Boolean is
  begin
    return P.X = Q.X;
  end On_Vertical;

  procedure Put( This : in Point ) is
  begin
    Put_Line( "X = " & Integer'Image( This.X ) );
    Put_Line( "Y = " & Integer'Image( This.Y ) );
  end Put;
end Point_Pckg;
```

Alternatively, in the Ada version we could declare `On_Diagonal` and/or `Put` with an **`access`** `Point` parameter; this would allow dynamic binding based on passing an access value instead of a designated object.

The style of declaring all the fields `public` is not recommended; it is employed here to avoid introducing too much complexity into an initial example. Below (§6.1) we will revise both the Java and Ada versions to take better advantage of encapsulation.

The Ada version is longer than Java's for several reasons. First, modeling one language's constructs in another almost always requires more code in the target language. If we were to compose an arbitrary Ada package, the Java version would require additional verbiage for features that it does not directly support. Second, Ada's separation of a package into a specification and a body entails some (helpful) redundant syntax. Third, the use of access values is explicit in Ada, requiring a type declaration. Fourth, Java's C-based syntax is fairly terse (e.g., with "{" and "}" delimiters) whereas Ada is more natural-language oriented, using "begin" and "end".

A Java declaration

```
Point p = new Point();
```

corresponds to the Ada declaration

```
P : Point_Class_Ref := new Point;
```

Java thus uses the same name, `Point`, for both the reference (to an instance of either `Point` or any of its direct or indirect subclasses) and an instance that is specifically a `Point`. Ada requires different types for the two uses: `Point_Class_Ref` for the reference, and `Point` for the actual object.

More generally, Ada also allows declaring a `Point` data object directly, without going through a reference:

```
P_Obj : Point;
```

As observed earlier, Java has no analog to such a declaration: all objects in Java live on the heap and are referenced indirectly.

Invoking a method in Java uses "dot" notation:

```
p.onDiagonal()
```

whereas the corresponding Ada construct is a subprogram call

```
On_Diagonal( P.all )
```

or, if we had declared `On_Diagonal` to take an access parameter, then

```
On_Diagonal( P )
```

In each of these cases the binding is dynamic, based on the type of the designated object. For the Java class shown above there is no statically bound call equivalent to the Ada function invocation `On_Diagonal( P_Obj )`.

Field selection uses the same syntax in both languages – `p.x` – where the selection's legality is based on the declared type of the reference object `p`, regardless of the type of the designated object. In Ada, we can optionally include an explicit ".all" – `p.all.x` – to indicate the dereference.

A Java static method such as `onVertical`, which takes a parameter of a class type (versus a primitive or array type) corresponds to an Ada subprogram taking a parameter of a class-wide type. This issue will be addressed further (§4.1) when we discuss inheritance in the two languages.

Separating the class construct into two features allows Ada to be somewhat more general than Java, in several ways. Ada allows dynamic binding to be based on other than the first parameter to a primitive subprogram for a tagged type, and in fact allows dynamic binding based on several parameters (provided that each is from the same type). Moreover, Ada's package mechanism is a full-fledged modularization facility. Using a Java class simply as a module versus a data template is going somewhat against the grain of the language; for example, preventing a class from being instantiated requires declaring a `private` constructor, a style that is somewhat clumsy.

4 Inheritance

Java supports class hierarchies based on single inheritance, and also simulates multiple inheritance through a class-like mechanism known as an *interface*.

4.1 Simple Inheritance

In Java, inheritance is obtained by extending a class; for example:

```
public class ColoredPoint extends Point{
  protected int color;
  public void reverseColor(){
    color = -color;
  }
  public void put(){
    super.put();
    System.out.println( "color = " + color );
  }
}
```

The extending class is said to be the *subclass*, and the class being extended is the *superclass*. Each non-private instance method defined in the superclass is implicitly inherited by the subclass, with the superclass's implementation. The subclass may override with its own implementation any of these that the superclass did not specify as `final`. Static methods are not inherited. `ColoredPoint` inherits `onDiagonal()` as is, overrides the implementation of `put()`, and introduces a new method `reverseColor()`.

The "`super.`*member*" notation is used within an instance method to reference fields or methods defined for the immediate superclass. Thus the invocation `super.put()` calls the `put()` method for `Point`, which is `ColoredPoint`'s superclass.

In Ada, inheritance is realized through type derivation.

```
with Point_Pkg; use Point_Pkg;
package Colored_Point_Pkg is
  type Colored_Point is new Point with private;

  type Colored_Point_Class_Ref is
    access all Colored_Point'Class;

  procedure Reverse_Color( This :
                           in out Colored_Point );
  procedure Put( This : in Colored_Point );

private
```

```
  type Colored_Point is new Point with
    record
      Color : Integer;
    end record;
end Colored_Point_Pkg;
```

```
with Ada.Text_IO; use Ada.Text_IO;
package body Colored_Point_Pkg is
  procedure Reverse_Color( This :
                           in out Colored_Point )is
  begin
    This.Color := -This.Color;
  end Reverse_Color;

  procedure Put( This : in Colored_Point ) is
  begin
   Put( Point(This) ); -- Invoke parent operation
   Put_Line ( "Color = " &
              Integer'Image( This.Color ) ) ;
  end Put;
end Colored_Point_Pkg;
```

A protected per-instance member in Java (`color` in the above example) corresponds to a component of a private tagged type in Ada.

In Java, code for a subclass of `ColoredPoint` automatically has access to the protected member `color`. In Ada, the implementation of a descendant of `Colored_Point` has visibility to the `Color` component if the new type is declared in a child package, but not if it occurs in a unit that simply `with`'s `Colored_Point_Pkg`.

In Java, `onVertical` is a static method for `Point` and thus is not inherited. The corresponding Ada procedure `on_Vertical` takes a class-wide parameter and thus likewise is not inherited.

If a Java class does not explicitly extend another class, then it implicitly extends the ultimate ancestral class `Object`. The `Object` class allows the user to define "generic" container classes, heterogeneous arrays, and similar constructs. For example:

```
Object[] arr;
arr    = new Object[2];
arr[0] = new String("hello");
arr[1] = new Point();
String s = (String)(arr[0]);
Point p  = (Point)(arr[1]);
```

The assignments to `s` and `p` require the casts on the right side; a run-time check guarantees that the types of the array elements are correct.

Ada has no immediate analog to Java's `Object` class, although the types `Controlled` and `Limited_Controlled` in `Ada.Finalization` serve this role to some extent. This absence is not critical, since Ada's generic facility allows defining container data structures, and an access value designating a class-wide type offers heterogeneity. On the other hand, the provision of a "root" class is convenient since a number of useful methods are defined there, such as `toString()`. If a class overrides `toString()` then the resulting method is invoked implicitly in certain contexts such as an operand to concatenation.

Java allows fields to have the same name in a class and in a subclass, a feature known as "shadowing". Within the subclass the forms `super.fieldname` and `this.fieldname` (or simply `fieldname`) resolve any potential ambiguity. In contrast, Ada regards a data object of a tagged type as a single record, and a name in the

extension part is not allowed to duplicate a name of a field in the parent part. In this regard Java can claim to be less sensitive than Ada to changes in a superclass: if a new field is added to a class, then the subclasses do not need to be recompiled (see [GJS, §13.4.5]). A disadvantage to Java is that since the "`super`." notation applies at one level only, there is no direct way to access the fields in an ancestor of the superclass.

In Ada, modifying a tagged type by adding a new field `F` will cause a descendant type whose extension part has a field named `F` to become illegal. It is possible to avoid this problem by introducing a "wrapper" record type that is the sole component of the tagged type; the new field would be added to the wrapper type and would not clash with a name in the extension part. The same principle would need to be applied in the declaration of the extension part, to allow new fields to be added there without clashing with field names in further descendants.

Both Java and Ada allow a reference to be viewed as though it designates an object of a different type in the same class hierarchy. In Java this is known as a *cast*, in Ada it is a *view conversion* to a class-wide type. The semantics is roughly the same in both languages, with cast/conversion always allowed "towards the root", and also permitted "away from the root" but with a run-time check that the designated object is in the target class or one of its subclasses. Unless the source type is either a (direct or indirect) ancestor or descendant of the target type, the cast/conversion is illegal in both languages.

Testing if an object is in an inheritance hierarchy rooted at a particular type is realized in Ada through the `Boolean` expression

```
X in T'Class
```

where `X` is an object of a tagged type, and `T` is a tagged type. The equivalent Java form, which delivers a `boolean` value, is

```
x instanceof T
```

where `x` is a reference and `T` is a reference type. The expression returns `true` if the cast `(T)x` would succeed at run time (that is, if `x` references an object that is either of type `T` or of one of its descendant types).

Ada allows querying whether an object of a class-wide type is of a specific tagged type, through the `'Tag` attribute.

```
X : Point'Class := ... ;
...
if X'Tag = Point'Tag then
   ...
elsif X'Tag = Colored_Point'Tag then
   ...
end if;
```

Java obtains this effect through the `class` field of a class object (introduced in Java 1.1) and the `getClass()` method of the class `Object`, each of which delivers a value of class `Class`.

```
Point p = ...;
if (p.getClass() == Point.class)
   {...}
else if (p.getClass() == ColoredPoint.class)
   {...}
```

The following examples illustrate these semantic points.

```
class MyClass{
  void myFunc(){System.out.println("myFunc 1");}
  int myInt;
}
class MyOtherClass extends MyClass{
  void myFunc(){System.out.println("MyFunc 2");}
```

```
  void myOtherFunc(){
    Sytem.out.println{"myOtherFunc");
  }
  int myOtherInt;
}
class YourClass{
  void yourFunc(){ return; }
  int yourInt ;
}

class Test {
  public static void main(String [ ] args){
    MyClass      ref1 = new MyOtherClass();
    MyOtherClass ref2 = new MyOtherClass();
    YourClass    ref3 = new YourClass();

    ref1.myFunc();        //  (1) myFunc 2
    ref1.myOtherFunc() ;  //  (2) Illegal

    ((MyOtherClass)ref1).myOtherFunc(); // (3) OK
    ((MyClass)ref2).myFunc(); //(4) OK,myFunc 2
    ((MyClass ref3).myFunc(); // (5) Illegal cast

    if (ref1 instanceof MyOtherClass){  //(6)
      (MyOtherClass)ref1.myOtherFunc(); // OK
    }
  }
}
```

The statement at line (1) in `main` dynamically binds to the version of `myFunc` declared for `MyOtherClass`. Line (2) is illegal since the method `myOtherFunc` is not defined for `ref1`'s class type. Line (3) is legal and results in a run-time test that `ref1` references an object from `MyOtherClass` or any of its subclasses. The cast at line (4) is not needed, but in any event the call is bound dynamically to the version of `myFunc` found in the class of the referenced object, versus the class indicated by the cast. Line (5) is illegal since `YourClass` and `MyClass` are not related through an inheritance chain. The `if` statement at line (6) "downcasts" a reference in order to invoke a method defined for a subclass but not for the superclass.

Here is a corresponding set of Ada package specifications and a main procedure:

```
package MyClass_Pkg is
  type MyClass is tagged
    record
      MyInt : Integer;
    end record:
  type MyClass_Ref is access all MyClass'Class;
  procedure MyFunc( This : in MyClass );
end MyClass_Pkg;

with MyClass_Pkg; use MyClass_Pkg;
package MyOtherClass_Pkg is
  type MyOtherClass is new MyClass with
    record
      MyOtherInt : Integer;
    end record;

  type MyOtherClass_Ref is
    access all MyOtherClass'Class;

  procedure MyFunc( This : in MyOtherClass );
```

```
  procedure MyOtherFunc( This : in MyOtherClass);
end MyClass_Pkg;

package YourClass_Pkg is
  type YourClass is tagged
    record
      YourInt : Ingeger;
    end record;

  type YourClass_Ref is
    access all YourClass'Class;

  procedure YourFunc( This : in YourClas );
end YourClass_Pkg;

with MyClass_Pkg, MyOtherClass_Pkg, YourClass_Pkg;
use MyClass_Pkg, MyOtherClass_Pkg, YourClass_Pkg;
procedure Main is
  Ref1 : MyClass_Ref       := new MyOtherClass;
  Ref2 : MyOtherClass_Ref  := new MyOtherClass;
  Ref3 : YourClass_Ref     := new YourClass;
begin
  MyFunc( Ref1.all );      --(1)
  MyOtherFunc( Ref1.all ); --(2)
  MyOtherFunc(
    MyOtherClass'Class( Ref1.all ) );  --(3)
  MyFunc( MyClass'Class( Ref2.all ) ); --(4)
  MyFunc( MyClass'Class( Ref3.all ) ); --(5)

  if Ref1.all in MyOtherClass'Class then  --(6)
    MyOtherFunc( MyOtherClass'Class(Ref1.all) );
  end if;
end Main;
```

Line (1) dynamically binds to the version of `MyFunc` for `MyOtherClass`, since `Ref1` designates an object of this type. Line (2) is illegal since `MyOtherFunc` is not a primitive operation for `MyClass`. Line (3) is legal with a run-time check to ensure that `Ref1` designates an object of a type rooted at `MyOtherClass`. Line (4) is legal, but the view conversion to `MyClass'Class` is not needed. Line (5) is illegal since there is no view conversion defined between "sibling" types, The `if` statement at line (6) does an explicit test before attempting the conversion "away from the root".

In Java, selecting a member from a cast expression provides static binding using the reference type of the cast when the member is a field, but dynamic binding using the type of the designated instance when the member is a method. Strictly speaking, Ada has the same semantics (for a view conversion to a class-wide type), but the syntactic difference between selecting a field (with "dot" selection) and invoking a subprogram on a parameter of a class-wide type will likely prevent any confusion between static and dynamic binding. And in any event Ada does not allow the same component name to be used in both a parent record and an extension part, so the issue of static versus dynamic interpretation of field names does not arise.

A common OOP style is "passing the buck": the implementation of a method for a subclass invokes the overridden method from the superclass. In the `ColoredPoint` example at the beginning of this section, that class's implementation of `put` needs to invoke the `put` method for `Point`. The Ada style is to invoke the `Put` procedure for `Point` on a view conversion of the `ColoredPoint` parameter to the specific parent type `Point`. Java uses a special syntax, `super.put()`, for this effect, but it applies only to the immediate superclass (that is, a reference `super.super.put()` form the implementation of `put` in a descendant of `ColoredPoint` would be illegal).

4.2 Multiple Inheritance and Interfaces

Multiple inheritance – the ability to define a class that inherits from more than one ancestor – is a controversial topic in OO language design. Although providing expressive power, it also complicates the language semantics and the compiler implementation. C++ provides direct linguistic support (see [Wa93] and [Ca93] for arguments pro and con), as does Eiffel; on the other hand, Smalltalk and Simula provide only single inheritance.

Java takes an intermediate position. Recognizing the problems associated with implementation inheritance, Java allows a class to extend only one superclass. However, Java introduces a class-like construct known as an *interface* and allows a class to inherit from – or, in Java parlance, *implement* – one or more interfaces. Thus a user-defined class always extends exactly one superclass, either `Object` by default or else a class identified in an `extends` clause, but it may implement an arbitrary number of interfaces.

Like a class, an interface is a reference type, and it is legal to declare a variable of an interface type. Like an abstract class, an interface does not allow creation of instances. Interface inheritance hierarchies are permitted; an interface may extend a parent interface. However, there are significant restrictions that distinguish an interface from a class:

- Each method defined by an interface is implicitly `abstract` (that is, it lacks an implementation) and `public`
- An interface is not allowed to have any `static` methods
- Each variable in an interface is implicitly `static` and `final` (constant)

An interface thus has no implementation and no "state". When a class implements an interface, it must provide a "body" for each method declared in the interface. Some interfaces are general purpose, providing functionality that can be added to any class. Other interfaces are more specialized, and are intended only for particular class hierarchies.

Perhaps somewhat oddly at first glance, it is often useful to define empty interfaces (i.e. interfaces with no members), known as "marker" interfaces. Examples of predefined marker interfaces are `java.lang.Cloneable` (implemented by classes that support cloning) and `java.io.Serializable` (implemented by classes that support "flattening" to an external representation). Whether a class implements a given marker interface may be exploited by users of the class, assuming that implementing the interface affects how the class implements some of its other methods.

The `instanceof` operator may be used to test whether an object is of a class that implements a particular interface.

Here is an example of a simple interface:

```
interface Checkpointable{
  void save();
    // Construct and store copy of the object
  void restore();
    // Restore the value of the object from the
    // saved copy
}
```

A class can implement this interface in order to allow "checkpoint"ing the value for an object via a field that can later be retrieved. This might be useful if the programmer wants to cancel some updates to the object and restore the checkpointed value.

```
class CheckpointablePoint
  extends Point implements Checkpointable{

  protected Point p;  // saved value
```

```
  public void save(){
    p = new Point();
    p.x = x;
    p.y = y;
  }

  public void restore(){
    x = p.x;
    y = p.y;
  }

  public static void main ( String[ ] args ) {
    Point q = new CheckpointablePoint();
    q.x = 10;
    q.y = 20;
    q.put();  // Displays 10 and 20
    ((Checkpointable)q).save();
    q.x = 100;
    q.y = 200;
    ((Checkpointable)q).restore();
    q.put();  // Displays 10 and 20
  }
}
```

This example illustrates the "mixin" style of multiple inheritance; the `Checkpointable` interface defines the properties that need to be added to an arbitrary class.

Ada provides "mixin" multiple inheritance through a generic unit parameterized by a formal tagged type; the "mixin" properties are added by deriving from the formal tagged type. Any operations needed to implement these properties are supplied as additional generic formal parameters.

```
generic
  type T is tagged private;
  type T_Class_Ref is access all T'Class;
  with procedure Formal_Save ( From : in T;
                               To   : out T);
  with procedure Formal_Restore ( From : in T;
                                  To   : out T);
package Generic_Checkpointable is
   type Checkpointable_T is new T with private;
   procedure Save(This : in out Checkpoitable_T);
   procedure Restore( This :
                        in out Checkpointable_T);
private
  type Checkpointable_T is new T with
    record
      Ref : T_Class_Ref;
    end record;
end Generic_Checkpointable;
```

We could have expressed the generic to take a formal tagged type derived from `Point`, but this would be unnecessarily restrictive. Since the `Point` operations do not need to be overridden to work for checkpointable points, the formal type parameter can be expressed more generally as a formal tagged private type. Thus any tagged type can be extended with checkpointable operations.

For simplicity we are ignoring the potential storage leakage caused by the implementation of `Save`, which overwrites the `Ref` field. More realistically, we could either have an additional generic formal parameter for the reclamation procedure, which would then be called by `Save` before overwriting `This.Ref`, or else declare the formal type `T` as controlled.

```
   type T is
     new Ada.Finalization.Controlled with private;
```

To add the checkpointable operations to the `Point` tagged type from §3.4, we first need to implement the save and restore operations for this type.

```
package Point_Pkg.Checkpointable is
   procedure Save( From : in Point;
                   To :   out Point );
   procedure Restore( From : in Point;
                      To   : out Point );
end Point_Pkg.Checkpointable;
```

The implementation of `Save` and `Restore` simply copy the fields from the source object to the target object. We can instantiate the generic package to obtain a type that is in the inheritance hierarchy for `Point` and which adds the required `Save` and `Restore` operations.

```
with Point_Pkg.Checkpointable,
     Generic_Checkpointable;
use Point_Pkg.Checkpointable;
package Checkpointable_Point_Pkg is
   new Generic_Checkpointable(
         Point_Pkg.Point,
         Point_Pkg.Point_Class_Ref
         Save, Restore );
```

Java's interface mechanism for multiple inheritance has a number of advantages. Interface types are full-fledged reference types and participate in dynamic binding. The notation is succinct but expressive, simpler than the Ada version with generics. The Java predefined class library makes heavy use of interfaces, including the "marker" interfaces mentioned above, which reflect clonability and serializability.

However, there are also some drawbacks. Sometimes an interface supplies methods that should only be called from the body of the class that implements the interface, but since all methods in an interface are public there is no way to enforce such encapsulation. In the Ada version these operations could be declared in the private part of the generic package specification. Further, the restrictions on the contents of an interface are not necessarily obvious (fields are always static, whereas methods are always per-instance). Moreover, a name clash anomaly can arise with interfaces; as the next example shows, it is possible to construct two interfaces that cannot be jointly implemented by the same class.

```
interface Interfacel{
  int func();
}
interface Interface2 {
  boolean func();
}
```

An attempt to define a class that implements both `Interface1` and `Interface2` will fail, since Java's overloading rules prohibit defining multiple methods with the same signature that are differentiated only by the result type. Since interfaces will often be developed independently, this may turn out to be a significant problem in practice. This is not an issue if the interfaces define methods with identical signatures including the result type, since a class that implements both interfaces only supplies one method with the same name as in the interfaces.

Interestingly, Java uses inheritance (from `Object`) to simulate generics, whereas Ada uses generics to simulate multiple inheritance. The languages were opting to avoid semantic and implementation complexity, but at the cost of some complexity for the user.

4.3 Methods as Parameters

In Java, an abstract method may be used to model what in Ada would be realized through passing a subprogram as a generic parameter or using an access-to-subprogram value as a run-time value. For example, here is a typical usage of an access-to-subprogram type in Ada; given a function `F(J)` that takes an integer parameter and returns an integer result, and a positive integer `N`, compute the sum `F(1) + F(2) + ... F(N)`.

```
package Utilities is

  type Func_Ref is
    access function(J : Integer) return Integer;


  function Sum( F: Func_Ref; N : Positive )
    return Integer;
  -- Returns the sum of values F(J) for J in 1..N
end Utilities;

package body Utilities is
  function Sum( F: Func_Ref; N : Positive )
    return Integer is
    Subtotal : Integer := 0;
  begin
    for J in 1..N loop
      Subtotal := Subtotal + F.all(J);
    end loop;
    return Subtotal;
  end Sum;
end Utilitiies;

function Square(N : Integer) return Integer is
begin
  return N**2;
end Square;

with Square, Utilities, Ada.Text_IO;
use Utilities, Ada.Text_IO;
procedure Test is
   Result : Integer;
begin
  Result := Sum( Square'Access, 4 ) ;
  Put_Line( Integer'Image( Result ));  -- 30
end Test;
```

One way to express this in Java is through inheritance from an abstract class:

```
abstract class Utilities{
  abstract int f( int j ) ;

  public int sum( int n ){
    int subtotal = 0;
    for (int j = 1; j <= n; j++){
      subtotal += f(j);
    }
    return subtotal;
  }
}
```

```
class SquareUtilities extends Utilities{
  static int square( int j ){
    return j*j;
  }
  int f( int j ){
    return square(j);
  }
  // sum is inherited automatically
  public static void main ( String[] args ){
    int result;
    result = new SquareUtilities().sum( 4 );
    System.out.println( result );  // 30
  }
}
```

Although inheritance from an abstract class does simulate the effect of subprograms as parameters, the style has more the flavor of a workaround than a solution. Ada's approach is more direct, with the call of `Sum` identifying the function that is to be used. The need to override an abstract method and use dynamic binding does not make the intent especially clear.

Another approach is to model a method pointer by an object whose class has a method member with the required signature.

```
interface FuncRef{
   int f( int j );
}
```

```
class Utilities{
  public int sum( FuncRef func, int n ){
    int subtotal = 0;
    for (int j = 1; j <= n; j++){
      subtotal += func.f(j);
    }
    return subtotal;
  }
}
```

```
class Square implements FuncRef{
  public int f(int j){
    return j*j;
  }
}
```

```
class Test{
  public static void main ( String[] args ){
    int result;
    Square s = new Square();
    result   = new Utilities().sum(s, 4);
    System.out.println( result );  // 30
  }
}
```

This is perhaps clearer than the version with the abstract class, but still not as direct as the Ada approach with access values designating subprograms.

5 Summary of Overloading, Polymorphism and Dynamic Binding

Java and Ada both allow overloading, but Ada is more general in allowing overloading for operator symbols and also overloading of functions based on the result type. As noted earlier, the absence of overloading based on

function result type makes it possible to define Java interfaces that cannot be jointly implemented by the same class.

Polymorphism, the ability of a variable to be in different classes of an inheritance hierarchy at different times, is implicit in Java. If `p` is declared as a reference to class `C`, then `p` can designate an instance of either `C` or any of its direct or indirect subclasses. In contrast, polymorphism is explicit in Ada, through a variable of an access-to-class-wide type. In Java a variable of type `Object` is completely polymorphic, capable of designating an object from any class. The closest Ada analog is a variable of an access type whose designated type is `Ada.Finalization.Controlled'Class`, which can designated an object of any type derived from `Controlled`.

Instance method invocation in Java is in general bound dynamically: in a call `p.method(...)` the version of `method` that is invoked is the one defined by the class of the object that `p` designates. Static binding applies, however, in several situations: an invocation `super.method(...)`; an invocation of a `final` method; or an invocation of a `private` method (from within the class defining the method). A `static` method is also bound statically. Ada subprograms calls are in general bound statically: dynamically binding only occurs when the actual parameter is of a(n) (access-to) class-wide type `T'Class` and the called subprogram is a primitive operation for the tagged type `T`.

In Java there is an important (but perhaps subtle) difference in the semantics between an invocation of a method on `super` versus on any other reference. If `p` is a reference to an object of class `C`, then the invocation `p.method(...)` is dynamically bound based on the type of the designated object, but `super.method(...)` is statically resolved to `method(...)` defined for `C`'s superclass. Perhaps confusingly, `this.method()` is bound dynamically, in contrast to `super.method()`. The Ada view conversion approach has more consistent semantics.

In Ada, an object `X` of a class-wide type `T1'Class` can be "view converted" to a specific tagged type `T2` that is either a descendant or ancestor of `T1`, with a run-time check that `X` is in `T2'Class` if `T2` is a descendant of `T1`. If the view conversion `T2(X)` is passed as an actual parameter to a primitive operation of `T2`, the binding is static, not dynamic. Java lacks an equivalent facility for forcing static binding. Even if a reference `x` to a `t1` object is cast to type `t2`, a method invocation `((t2)x).f()` is dynamically bound to the method `f()` in `t1`, not the version in `t2`. The Ada analog to a Java cast is thus a view conversion to a class-wide type, not to a specific type.

In Ada it is possible to obtain dynamic binding through access values designating aliased class-wide variables rather than allocated objects; this avoids the overhead of heap management. Java has no such mechanism: all objects go on the heap.

6 User-Controlled Basic Behavior

A number of operations dictate the fundamental behavior for instances of a data type: construction/initialization, cloning/assignment, finalization, and the test for equality. Both Java and Ada allow the author of the type to specify these operations' availability and implementation, though with some stylistic and semantic differences.

6.1 Construction/Initialization

A Java class may include one or more constructors; a constructor is a method with special syntax and semantics that is called during object allocation. Here is a revised version of the `Point` class shown earlier, with two constructors to arrange initialization, and with `protected` versus `public` fields for encapsulation:

```
public class Point{
  protected int x, y;
  protected static int numPts = 0;

  public Point(int x, int y) {
    this.x = x;
```

```
    this.y = y;
    numPts++;
  }
  public Point(){ this(0, 0); }

  public boolean onDiagonal (){
    return (x == y);
  }
  public void put(){
     System.out.println ( "x = " + x ) ;
     System.out.println ( "y = " + y ) ;
  }
}
```

A constructor has the same name as the class and lacks a return type (and hence does not need a `return` statement). Java's overloading rules allow the declaration of multiple constructors; this is a common style. A constructor is invoked as an effect of an object allocation, for example

```
Point p1 = new Point(10, 20);
  // Now p1.x is 10, p1.y is 20, and
  // numPts is 1
Point p2 = new Point();
  // Now p2.x and p2.y are both 0, and
  // numPts is 2
```

A constructor is called after the object's instance variables have been set to their default values and after any explicit initializers have been executed. In the above example, the assignment to `p1` allocates a `Point` object and invokes the two-parameter constructor that sets `p1`'s `x` and `y` fields and increments `numPts`.

The assignment to `p2` invokes a parameterless constructor (colloquially known in Java as a "no-arg" constructor), which calls the other constructor to initialize the object's fields to zero and to increment `numPts`.

An explicit constructor invocation is only permitted as the first statement of another constructor; the invocation will either specify `this` (for a constructor in the same class) or `super` (for a constructor in the superclass). If the first statement of a constructor is not an explicit invocation of another constructor, then an implicit call of `super()` is inserted.

Since Java's accessibility control is orthogonal to its constructor facility, the ability to allocate objects can be encapsulated. At the extreme, if a constructor is declared `private` then explicit object allocation is allowed only within the class's implementation.

Ada has no immediate analog to Java constructors, but simulates their effect through controlled types. An explicit `Initialize` procedure coupled with providing default expressions to initialize components is the Ada analog to a Java no-arg constructor, and discriminants can sometimes be used to model Java constructors taking arguments. In some situations a Java constructor may be modeled simply with an Ada function that delivers a value of the type in question.

```
with Ada.Finalization; use Ada.Finalization;
package Point_Pkg is
  type Point(Init_X, Init_Y : Integer := 0)
    is new Controlled with private;

  type Point_Class_Ref is access all Point'Class;

  procedure Initialize(Obj : in out Point) ;
  function On_Diagonal(This : Point)
    return Boolean;
  procedure Put( This : in Point );
```

```
private
  Num_Pts : Integer := 0;
  type Point( Init_X, Init_Y : Integer := 0 )
    is new Controlled with
     record
       X : Integer := Init_X,;
       Y : Integer := Init_Y;
     end record;
end Point_Pkg;

with Ada.Text_IO; use Ada.Text_IO;
package body Point_Pkg is
  procedure Initialize( Obj : in out Point ) is
  begin
    Num_Pts := Num_Pts + 1;
  end Initialize

  function On_Diagonal( This : Point )
    return Boolean is
  begin
    return This.X = This.Y;
  end On_Diagonal;

  procedure Put(This : in Point ) is
  begin
    Put_Line( "X = " & Integer'Image( This.X )) ;
    Put_Line( "Y = " & Integer'Image( This.Y )) ;
  end Put;
end Point_Pkg;
```

Here is an example of its usage:

```
P1 : Point_Class_Ref := new Point(10,20);
-- Now P1.X is 10, P1.Y is 20,
--  and Num_Pts is 1
P2 : Point_Class_Ref := new Point;
-- Now P2.X and P2.Y are both 0, and
-- Num_Pts is 2
```

6.2 Finalization

The root class `java.lang.Object` supplies a protected method `finalize()` that can be overridden to do any resource cleanup required before the object is garbage collected. For example, if an object contains fields that reference open files, then it is generally desirable to close the files when the object is reclaimed, versus waiting until program termination.

As a matter of style, the first statement in a `finalize` method should be the invocation

```
super.finalize();
```

to ensure performing any finalization required for the superclass.

Finalization in Ada differs somewhat from Java both stylistically and semantically. Since Ada does not guarantee garbage collection, finalization is generally used in Ada to obtain type-specific storage management such as reference counts. In Java there is no need for `finalize` to do storage reclamation. Semantically, `Finalize` is called in Ada not only when a controlled object is deallocated but also when it goes out of scope or is the target of an assignment. Moreover, since the Java language does not define when garbage collection occurs, a programmer cannot predict exactly when `finalize` will be called, or the order in which finalizers are executed. In fact, the JVM may exit without garbage collecting all outstanding objects, implying that some finalizers might never be invoked.

A Java object is not necessarily freed immediately after being finalized, since the `finalize` method may have stored its `this` parameter in an outer variable, thus "resurrecting" it [F197, p. 62]. Nonetheless `finalize()` is invoked only once. In Ada it is possible under some circumstances for the `Finalize` procedure to be invoked more than once on the same object; the programmer can attend to such a possibility by storing a `Boolean` component in the controlled object that keeps track of whether finalization has already occurred.

In Java, an exception thrown by `finalize()` is lost (i.e., not propagated), despite the fact that the signature for `finalize` in class `Object` has a "`throws`" clause. In Ada it is a bounded error for a `Finalize` procedure to propagate an exception.

6.3 Control over cloning/assignment and equality

Ada has the concept of a *limited* type; viz., one for which assignment is unavailable and the "=" and "/=" operations are not provided by default. In Java the assignment and equality operations copy and compare references, not objects, and are available for all classes. However, Java recognized that the concepts of copying (or "cloning") an object, or comparing two object for logical equivalence are class specific and thus need to be definable by a class author.

Java's object cloning mechanism was designed to satisfy two goals:

- Allow the class author to decide whether any given class should support cloning
- Provide a simple default ("shallow copying") for those classes that support cloning, overridable where more sophisticated processing is required.

Java provides two features to satisfy these goals. The "marker" interface `Cloneable` needs to be implemented by any class for which cloning is to be supported. The protected method `clone` in class `Object` checks that the `Cloneable` interface is implemented (otherwise throwing a `CloneNotSupported-Exception`), allocates a new object that is a byte-by-byte copy of the this object, and returns a reference to the new object. The `clone` method is implicitly inherited, though as a protected method is it not directly invokable by clients of the subclass.

The object equivalence method, in class `Object`, has the following signature:

```
public boolean equals( Object obj );
```

Its default implentation returns `true` if the object denoted by `this` and by `obj` are equal based on a "shallow" byte-by-byte comparison, and returns `false` otherwise.

A limited type is the Ada analog to a Java class that does not implement `Cloneable`. Attempting assignment for objects of a limited type is detected at compile time in Ada; invoking the `clone` method for a class that does not implement the `Cloneable` interface throws a run-time exception in Java.

A controlled type is the Ada analog to a Java class that implements `Cloneable` and overrides `clone`. In Ada the programmer would need to define type-specific `Finalize` and `Adjust` procedures. The fact that assignment in Ada for a controlled type invokes both `Finalize` and `Adjust` makes Ada somewhat more flexible than Java. In Java a statement

```
p1 = p2.clone();
```

does not invoke `p1.finalize()`. If `p1` designated an object with resources that needed cleaning up with a finalizer, then this cleanup would be deferred until `p1` was garbage collected. The analogous Ada construct is an assignment statement, and `Finalize` is guaranteed to be invoked on the target as part of the effect of the assignment.

Corresponding to the Java equals method is the Ada "=" operation. In both languages the programmer can override the default implementation to reflect logical equivalence.

7 Exception Handling

Java and Ada share a similar attitude about exception handling but realize their support through somewhat different mechanisms. In Java, exceptions are defined through the language's Object Oriented features. An occurrence of an exception is an allocated object of some class that inherits from the class `java.lang.Throwable`, or, more typically, from its subclass `Exception`. An exception object is thrown either implicitly from the compiled code (for example, an attempt to dereference `null`) or explicitly by a `throw` statement. It is then either caught by a local handler (in a `catch` clause following a `try` block) or else propagated up the dynamic call chain.

Since the user may introduce new fields when declaring an exception class, a constructor may be defined to initialize these fields appropriately when an object of this class is allocated. These fields may be retrieved by the code that catches the exception.

Java distinguishes between *unchecked* and *checked* exceptions[†]. The unchecked exceptions comprise the classes `Error` and `RuntimeException` and their subclasses. They reflect the exceptions that result from internal failures (for example, a JVM bug) and those that correspond to some of the violations of dynamic semantics (for example an array index out of bounds). All other exceptions are checked.

A Java method must explicitly specify, in a `throws` clause that is part of its signature, any checked exceptions that it throws but does not catch locally. If it lacks a `throws` clause, then it must catch all checked exceptions thrown by itself or propagated by the methods that it invokes. When a method is overridden by a subclass, the new version's `throws` clause is not allowed to specify any exceptions outside the classes of exceptions given by the overridden method's `throws` clause.

Java's model is more general than Ada's, but Ada is more efficient for several reasons:

- There is no need for dynamic allocation in Ada in connection with raising exceptions.
- The number of kinds of exceptions in Ada is statically known. Since Java allows per-instance classes, and classes local to a recursive method, the number of exception classes is dynamic in Java.

During the original Ada development, the design team looked at the possibility of requiring subprograms to specify the exceptions that they propagate. They eventually rejected this approach for pragmatic reasons, concluding that the programmer would too often just specify that a subprogram could raise any exception, information that just adds clutter to the code without improving readability. Java has taken a compromise view, requiring that uncaught checked exceptions be specified in a `throws` clause but not requiring this for unchecked exceptions.

Here is an example of exception handling in Ada:

```
package My_Pack is

  procedure My_Proc( N : in Integer );
  -- May raise My_Error

  exception My_Error;

end My_Pack;
```

[†] The terminology is possibly misleading. "Checked" and "unchecked" do not refer to compiler-generated checks for exception conditions at run time, but rather to the compiler's checking whether a method specifies in a `throws` clause the classes of exceptions that it can propagate.

```
with Ada.Exceptions; use Ada.Exceptions;
package body My_Pack is
  procedure My_Proc( N : in Integer ) is
  begin
    if N<0 then
       Raise_Exception(
         My_Error'Identity,
         Integer'Image(N) );
    end if;
  end My_Proc;
end My_Pack;

with My_Pack, Ada.Exceptions, Ada.Text_IO;
use My_Pack, Ada.Exceptions, Ada.Text_IO;
procedure My_Main is
begin
  My_Proc( -1 );
exception
  when E : My_Error =>
    Put_Line( Exception_Message( E ) );
end My_Main;
```

A corresponding Java version is as follows:

```
class MyClass(
  static class MyError extends Exception(
    MyError(String s ) {
      super(s);
   }
  }
  static void myProc( int n ) throws MyError{
    if (n < 0 ) {
      throw new MyEror( String.valueOf( n ) );
    }
  }
  public static void main (String[] args ) {
    try{
      myProc( -1 );
    }
    catch (MyError e ){
      System.out.println( e.getMessage() );
      // displays "-1"
    }
  }
}
```

The predefined class `java.lang.Exception` provides a constructor that takes a `String` parameter, and a method `getMessage` that retrieves the `String` that was passed to this constructor. Since constructors are not inherited, the constructor for `MyError` needs to be explicitly supplied.

8 Thread Support

Ada and Java are unusual in providing direct linguistic support for concurrent programming. In Ada the *task* is the unit of potential concurrency, and tasks may communicate and synchronize directly (through *rendezvous*), through operations on *protected objects*, or through accesses to shared data objects specified as *atomic*. The tasking model is orthogonal to the language's OOP support.

In Java, the support for concurrency is defined in terms of the OO features. The package `java.lang` supplies a class, `Thread`, whose objects embody potentially concurrent executing entities, and also an interface,

`Runnable`. A principal method in the `Thread` class is `run`, which is the code actually executed after a `Thread` object's `start` method is called. The `Thread` class also contains a number of methods relevant to thread synchronization.

One way to obtain multi-threading is to extend the `Thread` class and override the `run` method to carry out the necessary processing. The program needs first to invoke a constructor for the given `Thread` subclass (which allocates the resources, such as a run-time stack, necessary to support the thread's execution), and then to call the object's `start` method. The `start` method calls `run`, which may be either an infinite loop or a bounded sequence of statements.

However, Java's single inheritance model precludes subclassing `Thread` if the new class also needs to extend some other class. The solution is to implement the `Runnable` interface. This interface supplies one method, `run`, which needs to be overridden. The program can call a `Thread` constructor passing as a parameter the object of the class that has implemented `Runnable`. Invoking this thread's `start` method will cause its `run` method to be called.

Mutual exclusion may be achieved in several ways. A low-level mechanism is to specify the modifier `volatile` on a field that is being accessed by different threads. The effect is similar to Ada's `pragma Atomic`, inhibiting optimizations and thus ensuring that each access to the field is to its current value, and also requiring indivisibility with respect to thread scheduling.

A second mechanism is the `synchronized` statement (effectively a conditional critical region) which "locks" an object that the executing thread may then access. This statement is not generally recommended, since it distributes the thread-sensitive code across the program versus centralizing it in the implementation of the affected classes.

The preferred mechanism is to encapsulate shared data in an object of a class whose methods are specified as `synchronized`. Associated with each object is a conceptual lock that is held by the thread currently executing one of the object's synchronized methods. Analogously there is a lock for the class itself, for any `static` fields. If a thread θ attempts to call a synchronized method for an object (or class) that is locked by another thread, θ is suspended. When a lock is released at the conclusion of a synchronized method, some thread suspended on the lock is awakened and given the lock (as an atomic action). Whether priorities affect which thread is awakened depends on the underlying operating system.

If a synchronized method needs to wait for a `boolean` condition, then the Java idiom is

```
while ( !condition ) wait();
```

and, in the other direction, if a synchronized method takes an action that may set one of the conditions to `true`, then it should invoke the `notify` method to awaken one thread waiting for the condition, or `notifyAll` to awaken all threads waiting for the condition.

Java's exception semantics partially avoids an anomaly that arises in the Ada tasking model. In Ada an exception raised in a task body's statements but not handled is not propagated; rather, the task simply becomes completed, with no notification to the rest of the program. This situation does not arise in Java for checked exceptions, since the `run` method (both in the `Thread` class and in the `Runnable` interface) lacks a `throws` clause. Thus any overriding version of `run` is required to catch all checked exceptions; it is a compile-time error if an implementation of `run` does not catch all checked exceptions thrown by itself or by the methods that it invokes. However, unchecked exceptions such as one resulting from a `null` dereference need not be caught, and a Java thread whose `run` method throws such an exception will terminate silently.

Exceptions are always synchronous in Ada. In Java an asynchronous exception is thrown when the `stop` method of a thread is invoked; the affected thread then completes abnormally. The closest analog in Ada is an asynchronous `select` statement.

Here is an Ada example for a bounded buffer, containing Integer elements, which is used by producers and consumers. A producer task is blocked when the buffer is full, and a consumer task if blocked when the buffer is empty.

```
package Buffer_Pkg is
  type Integer_Array is
    array (Positive range <> ) of Integer;
  protected type Buffer(Max : Natural) is
    entry Put( Item : in Integer ) ;
    entry Get( Item : out Integer );
  private
    Data : Element_Array(1..Max);
    Next_in, Next_Out : Integer := 1;
    Count             : Natural := 0;
  end Buffer;
end Buffer_Pkg;

package body Buffer_Pkg is
  protected body Buffer is
    entry Put( Item : in Integer )
      when Count < Max is
    begin
      Data(Next_In) := Item;
      Next_In       := (Next_In mod Max)+1;
      Count         := Count+1;
     end Put;

     entry Get( Item: out Integer )
       when Count > 0 is
     begin
       Item     := Data(Next_Out);
       Next_Out := (Next_Out mod Max)+1;
       Count    := Count-1;
     end Get;
  end Buffer;
end Buffer_Pkg;

with Buffer_Pkg; use Buffer_Pkg;
procedure Producer_Consumer is
  Buff : Buffer(20) ;

  task Producer;
  task body Producer is
  begin
    for J in 1..100 loop
      delay 0.5;
      Buff.Put( J );
    end loop;
  end Producer;

  task Consumer;
  task body Consumer is
    K : Integer;
  begin
    for J in 1..100 loop
        Buff.Get( K );
        delay 1.0;
    end loop;
  end Consumer;
```

```
begin
  null; -- wait for tasks to terminate
end Producer_Consumer;
```

A Java version is as follows, illustrating both a `Thread` subclass and a `Runnable` implementation. Recall that `protected` has quite different meanings in Ada and Java.

```
class Buffer{
  protected final int max;
  protected final Object[] data;
  protected int nextIn=0, nextOut=0, count=0;

  public Buffer( int max ){
    this.max  = max;
    this.data = new Object[max];
  }
  public synchronized void put(Object item)
  throws InterruptedException{
    while (count == max) { wait(); }
    data[nextIn] = item;
    nextIn       = (nextIn+1) % max;
    count++;
    notify(); //a waiting consumer, if any
  }

  public synchronized Object get()
  throws InterruptedException{
    while (count == 0) { wait(); }
    Object result = data[nextOut];
    nextOut       = (nextOut+1) % max;
    count--;
    notify(); // a waiting producer, if any
    return result;
  }
}
class Producer implements Runnable{
  protected final Buffer buffer;
  Producer( Buffer buffer ){
    this.buffer=buffer;
  }

  public void run() {
    try{
      for (int j=1; j<=100; j++){
        Thread.sleep( 500 );
        buffer.put( new Integer(j) );
      }
    }catch (InterruptedException e)
    {return;}
  }
}
class Consumer extends Thread{
  protected final Buffer buffer;
  public Consumer( Buffer buffer ){
    this.buffer = buffer;
  }
  public void run(){
    try {
      for (int j=1; j<=100; j++){
        Integer p = (Integer)(buffer.get());
        int k = p.intValue();
```

```
        Thread.sleep( 1000 );
      }
    } catch (InterruptedException e)
      {return;}
  }
}
public class ProducerConsumer{
  static Buffer buffer = new Buffer(20);
  public static void main(String[] args){
    Producer p  = new Producer(buffer);
    Thread   pt = new Thread(p);
    Consumer c  = new Consumer(buffer);
    pt.start();
    c.start();
  }
}
```

The `put` and `get` methods each include a `throws` clause specifying `InterruptedException`, since the `wait` method in class `Thread` includes such a clause and `put` and `get` do not catch exceptions from this class. However, the implementation of each `run` method needs to catch `InterruptedException`: `get` and `put` can propagate exceptions from this class, and the `run` method, lacking a `throws` clause, must catch any exceptions propagated from methods that it invokes. It would not have been possible to simply put a `throws InterruptedException` clause on the implementation of the `run` method, since the signature of this method in the superclass `Thread` and in the interface `Runnable` lacks a `throws` clause. As noted earlier, a method is not allowed to throw more kinds of exceptions than are specified in the `throws` clause of the version of the method that it is overriding.

Java's use of the OOP features for threads offers some advantages. For example, it is simple to define an array of `Thread` objects, thus allowing common operations to be performed on them independent of their specific behavior. In Ada the task type mechanism only partially satisfies this need; in general, one must declare an array of `Task_Id` values and associate such values with individual tasks. Moreover, it is convenient to have a class that both captures the requirements for concurrency and participates in an inheritance hierarchy. In Ada the concurrency (for objects of a tagged type) would need to be modeled by components that are tasks or protected objects. (However, there is an interaction between OOP and synchronization, known as the "Inheritance Anomaly", that interferes with combining the two in a seamless manner. A discussion of this issue and how it applies to Java and Ada, is found in [Br98].)

Ada's rendezvous model offers a higher-level approach to task communication than Java's wait/notify mechanism, and Ada's protected object feature is somewhat more expressive, and potentially more efficient, than the corresponding Java facility with synchronized methods. Ada distinguishes protected entries from protected subprograms, thus making it explicit when queue management (versus just object locking) is needed, and also allowing concurrent reads of protected functions. In Java there is nothing syntactically different about a synchronized method that might block versus one that doesn't, preventing some optimizations. In Ada, if the implementation provides different queues for different protected entries, then the evaluation of some of the barrier conditions might not be necessary. In Java the condition tests that are triggered by a notify or notifyAll invocation are in normal user code versus being syntactically distinguished as barriers, and are not optimizable. Moreover, the need for an explicit loop on the condition test, and a corresponding explicit invocation of `notify` or `notifyAll`, are potentially error prone. In Ada the entry barrier explicitly shows the condition being tested, and the notification of waiting tasks is automatic at the end of a protected procedure or entry.

9 Integration of OO Features into the Language

In Java the OO features are the language's essence. Arrays, strings, exceptions, threads, and the predefined class library heavily exploit the OO model; indeed, classes are themselves objects (of the class `Class`) and may be

manipulated as run-time values. The only major facility that does not depend on object orientation is the set of primitive types (`char, int`, and so on), but even here the "wrapper" classes `Character, Integer`, etc., allow an OO view. Java's designers omitted traditional data structuring facilities such as record/structure types and variant record/union types since these features could be simulated by classes and inheritance. An effect of the OO-centric design is that one needs to be explicit in order to *avoid* using Java's OO features: for example, by declaring a class or method `final` (thus preventing the class from being extended or the method from being overridden), or by declaring a class all of whose members are `static`.

Java lacks generic units (templates), but uses Object Orientation, in particular the root class `Object` from which all classes implicitly inherit, to simulate some of the applications of generics. For example, an Ada generic linked list package can be modeled by a Java linked list class whose element type is `object`. However, simulating generics through the class Object sacrfices compile-time checking, introduces run-time overhead (for checking casts), and prevents constraining the elements of a "container" data structure to be all of one type.

Java lacks an ability to use a method as a run-time value; again, this effect is simulated through OOP.

Ada, on the other hand, provides a clear separation between the OO features and the rest of the language, and one needs to be explicit in order to use OOP. In the predefined environment, OOP is exploited where relevant, for example in `Ada.Streams`, but most of the packages are defined more simply using private (non-tagged) types or other software engineering features. A software developer using Ada has an assortment of facilities to draw from (including private types, generic units, tasking, access-to-subprogram types, and OOP) and can choose those that are of relevance to the problem.

10 Conclusions

Both Ada and Java are *bona fide* Object-Oriented languages. Java treats object orientation as its central mission; this brings a number of advantages (consistent integration of features, safe automatic storage reclamation, dynamic flexibility in data structures) but also a few drawbacks (run-time overhead due to implicit heap usage and management, absence of several fundamental data structuring facilities, occasional awkwardness in applying an OO style to a problem whose essence is process/function-related). Ada treats object orientation as one methodology, but not necessarily the only one, that might be appropriate for a software developer, and hence its OO support complements and supplements a more traditional set of stack-based "third-generation language" features. For example, in many situations data encapsulation (Ada private types) will be sufficient; the full generality of OOP is not needed.

Ada's advantages include run-time efficiency - a program only incurs heap management costs if it explicitly uses language features involving dynamic allocation – design method neutrality, and standardization. On the other hand, the need for explicit access types in Ada introduces notational overhead compared with Java, and if the implementation does not provide garbage collection then an Ada developer will need to employ controlled types or an equivalent mechanism to avoid storage leaks.

In any event the choice of Ada or Java is not necessarily "either/or". Support for interfacing with Java and generating Java bytecodes from an Ada source program is presently provided by Aonix [Ao97] using Intermetrics' AdaMagic technology; mixed-language systems with Ada and Java allow the software developer to exploit the benefits of both languages.

The accompanying table summarizes the OO-related features of the two languages.

Feature	Java	Ada
Class	Combines module and data template	Separated into two features, package and tagged type
Pointers	Implicit	Explicit (access values)

Storage reclamation	Garbage collection provided by implementation	Implementation –provided garbage collection, controlled type or storage pool defined by class author, or unchecked deallocation by application programmer
Method invocation syntax	`obj.func()`	`func(obj)`
Inheritable operation	Non-private, non-final per-instance method in a class	Primitive subprogram for a tagged type
Single inheritance	A class that extends a superclass	A type that is derived from a tagged type
Multiple inheritance	A class that extends one superclass and implements one or more interfaces	A generic package with a formal tagged type, extended in the specification
Polymorphism	Implicit for any variable of a class or interface type	Explicit through an access type whose designated type is class-wide
Method binding	Dynamic except for methods that are static, final, private, or from super	Static except for parameter of class-wide type passed to primitive operation
Control over fundamental operations	Constructors, finalize, clone, equals	Controlled types-Initialize, Finalize, Adjust
Encapsulation	Access modifier for class member	Placement of entity declaration in a package (visible part, private part, body); child unit
Generic template	Simulated by a component with class `Object`	General-purpose mechanism for compile-time parameterization
Namespace control	Packages, inner classes	Packages, child units, nested packages

References

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

[AI85] ANSI/IEEE *IEEE Standard for Binary Floating-Point Arithmetic*; ANSI/IEE Std. 754-1985; 1985.

[Ao97] Aonix; "ObjectAda Integrated Development Environment", v 7.1; 1997.

[Ar96] K. Arnold; *The Java Programming Language*, Professional Development Seminar delivered to Boston ACM Chapter, October 1996

[Br98] B. Brosgol; A Comparison of the Concurrency Features of Ada 95 and Java", *Proc. SIGAda '98*; ACM SIGAda; 1998.

[Ca93] T.A. Cargill, "The Case Against Multiple Inheritance in C++", in *The Evolution of C++* (J. Waldo, ed.), pp. 101-110, The MIT Press, 1993.

[FI97] D. Flanagan; *Java in a Nutshell* (2nd edition), O'Reilly & Associates, 1997.

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

[In95] Intermetrics, Inc.; *Ada Reference Manual – Language and Standard Libraries*, ISO/IEC 8652;1995

[Jø93] J. Jørgensen; "A Comparison of the Object Oriented Features of Ada 9X and C++", in *Ada Europe '93 Conference Proceedings*; Paris, France; June 1993.

[Su96] Sun; "The Java ™ Language; and Overview", document available on-line at `http:\\java.sun.com/docs/overview/java/java-overview-1.html`

[Ta96] S.T. Taft; "Programming the Internet in Ada 95", in *Ada Europe '96 Conference Proceedings*; Montreux, Switzerland; June 1996.

[Un96] Unicode Consortium; *The Unicode Standard*: *Worlwide Character Encoding, Version 2.0*; Addison-Wesley, 1996, ISBN 0-201-48345-9

[Wa93] J. Waldo, "The Case For Multiple Inheritance in C++", in *The Evolution of C++* (J. Waldo, ed.), pp. 111-120, The MIT Press, 1993.