How Ada 2005 Impacts CS1/2

Martin C. Carlisle
Department of Computer Science
2354 Fairchild Dr., Suite 6G101
U.S. Air Force Academy, CO 80840-6234
carlislem@acm.org

ABSTRACT
Ada 2005 is the second revision of the Ada standard. It makes changes to the language definition in a number of areas. We summarize which changes have the greatest impact on introductory computer science classes that use Ada.

1. INTRODUCTION

Ada 2005 has arrived. A number of resources are available for becoming familiar with the new language features [Ada05,Bar05,MS05,Wik05]; however, these seem to be addressed to the Ada practitioner rather than the Ada educator. We cover only those portions of the new language standard that will impact introductory programmers, such as those taking an introduction to computer science or data structures course.

2. RESERVED WORDS

Ada 2005 only adds three new reserved words to the language: `interface`, `overriding` and `synchronized`. These three reserved words may not be chosen as variable names (bringing the total number of reserved words to 72). The complete list is in Appendix A.

3. MORE ACCESS TYPES

3.1 Anonymous Access

Unlike the C family of programming languages, Ada 95 tended to require the use of named access or pointer types. We see this in the definition of a linked list, which requires type foreshadowing:

```
type List_Node;
type List is access all List_Node;
type List_Node is record
   …
   Next : List;
end record;
```

Using anonymous access types, we can define this more simply as:

```
type List_Node is record
   …
   Next : access List_Node;
end record;
```

Unfortunately, anonymous access types aren't allowable everywhere. In particular, you can not use an anonymous access type to instantiate `Ada.Unchecked_Deallocation`. Since conversions from anonymous access types to named access types are not automatic, you'll probably end up with the foreshadowing, unless you are willing to live with memory leaks.

Named access types also led to seemingly unnecessary type conversions when doing object-oriented programming. For example, with the following declarations:

```
   type Parent is tagged null record;
   type Parent_Access is access all Parent'Class;
   type Child is new Parent with null record;
   type Child_Access is access all Child'Class
   P : Parent_Access;
   C : Child_Access;
```

in the following code, we have:

```
   P := C;                -- illegal!!
   P := Parent_Access(C); -- legal
```

It seemed strange to force an explicit type conversion when the assignment of P to C can never fail, and simply is changing the view. We can avoid this problem by using anonymous access types instead of named ones.

```
   P : access Parent'Class;
   C : access Child'Class;
begin
   P := C;  -- legal!
```

Be careful, though. As previously mentioned, named access types convert automatically to anonymous ones, but the opposite isn't true:

```
   P1 : access Parent'Class;
   P2 : Parent_Access;
begin
   P2 := P1;  -- illegal!!
   P1 := P2;  -- legal
```

Anonymous access types are also now allowed as parameters, e.g.

```
procedure Catch(Ball : access Throwable);
```

3.2 Constant and Not Null Access

It allows for better software engineering to be able to specify that a particular pointer will never be null, or that you aren't allowed to change the contents of the item being referenced. The new constant and not null access types allow this to happen. Put simply, anywhere the word `access` appears, `not null` and `constant` can also show up. The `not null` qualifier can even be used to modify named access types. Some examples:

```
type Int_Access is access Integer;
type Book_Access is not null access Book;
type Book_Const_Access is not null access constant Book;
procedure Catch(Ball : not null access Throwable);
function Color(Car : not null access constant Automobile) return Color_Type;
-- Color may not change the value of Car.all
Y : not null Int_Access := new Integer'(5);
Z : not null Int_Access; -- illegal!
B : Book_Const_Access := new Book'(…);
…
B.Price := 9.99;  -- illegal!
```

The declaration of the variable `Z` is illegal, because Ada initializes access variables to null, and `Z` may not be null. Note that the declaration of `Y` solves this problem. The assignment to `B.Price` is illegal as the item pointed to must remain constant.

3.3 Anonymous Subprogram Access

Ada 95 had named access types for subprograms. This allowed a programmer to create a callback, or an iterator. Sometimes, it is desirable to use a nested subprogram as an iterator, as:

```
procedure Iterate (L : access List;
   Iterator : access procedure(Item : access List_Item));
…
procedure Add_To_Each(L : access List; Amount : in Natural) is
   procedure Add_One(Item : access List_Item) is
   begin
      Item.Value := Item.Value + Amount;
   end Add_One;
begin
   L.Iterate(Add_One'access);
end Add_To_Each;
```

The nesting here is important, as `Add_One` needs to access the value of `Amount`. This kind of iterator would be impossible in Ada 95 (except using the GNAT-specific `'Unrestricted_Access` attribute).

4. CONTEXT CLAUSES

Context clauses specify the dependency of a package on another. There are two new kinds of context clauses that can be used in a package specification, `private with` and `limited with`.

4.1 Improving Abstraction (private with)

Suppose you are implementing an abstract data type for a queue. You might implement this using a linked list as the underlying data structure. The package specification would then look like the following:

```
with Lists;
generic
   type Queue_Item is private;
package Queues is
   type Queue is tagged private;
   procedure Enqueue(This : in out Queue; Item : in Queue_Item);
   procedure Dequeue(This : in out Queue; Item : out Queue_Item);
   …
private
…
end Queues;
```

The client of the package should not need to know anything about the package `Lists`. In fact, you'd like to be able to have the compiler enforce that the package `Lists` is not used anywhere in the public portion of the specification of `Queues`. The `private with` clause solves this problem. By changing `with Lists` to `private with Lists`, the client of the package knows that they don't have to be concerned with lists, and the developer will have the compiler's help to ensure that the implementation of a queue doesn't leak into its public specification.

4.2 Mutually-Recursive Types (limited with)

Sometimes, you want to have two different mutually recursive types. This is particularly useful when mapping types from other languages, such as Java or C# [Car06,CSH02]. In Ada 95, these two types would have to be declared in the same package, as:

```
package Married is
   type Wife_Type;
   type Wife_Access is access all Wife_Type'Class;
   type Husband_Type;
   type Husband_Access is access all Husband_Type'Class;
   type Husband_Type is tagged record
      Wife : Wife_Access;
   end record;
   type Wife_Type is tagged record
      Husband : Husband_Access;
   end record;
end Married;
```

In Ada 2005, we can use a "`limited with`" clause to establish the mutual recursion and place the two types in separate packages. The "`limited with`" allows a package to declare access types for types from the other package. (This limitation means this package specification really needs to know nothing from the other package other than type names. It can only be used in a package specification). The example in Ada 2005 follows:

```
limited with Wife_Pkg;
package Husband_Pkg is
   type Husband_Type is tagged record
      Wife : access Wife_Pkg.Wife_Type;
   end record;
end Husband_Pkg;

limited with Husband_Pkg;
package Wife_Pkg is
   type Wife_Type is tagged record
      Husband : access Husband_Pkg.Husband_Type;
   end record;
end Wife_Pkg;
```

5. OBJECT-ORIENTED ENHANCEMENTS

5.1 Object.Method syntax

Most object-oriented programming languages use an object.method syntax for method calls. The designers of Ada 95, when adding dispatching to the language, chose to instead use the procedural syntax from Ada 83, "`method(object);`". Although this may be considered simply "syntactic sugar," when an OO hierarchy is nested through several packages, this can become awkward:

```
Grandparent_Pkg.Parent_Pkg.Child_Pkg.Method(This => object, Parm2 =>…);
```

If a new parent class is added, then all of the method calls need to be modified (unless of course "`use`" clauses are present, though these are usually frowned upon). In Ada 2005, however, this is much more simply written as:

```
object.Method(Parm2 =>…);
```

There is no lack of clarity about where Method will be found, and the code is less fragile with respect to changes in the class hierarchy.

5.1.1 Caveat Dispatcher

A word of caution is in order. By default, all method calls in Java and C# are dispatching; however, this is not so in Ada! Consider the following declarations:

```
type Grand_Parent is abstract tagged null record;
type Classwide_Ptr is access all Grand_Parent'class;
procedure Operation1(This : access Grand_Parent) is abstract;
procedure Operation2(This : access Grand_Parent);

type Parent is new Grand_Parent with null record;
overriding procedure Operation1(This : access Parent);
-- overriding is optional, indicates that this procedure must
-- override an inherited method
not overriding procedure Operation3(This : access Parent);
-- not overriding is optional, indicates that this procedure must NOT
-- override an inherited method

type Kid1 is new Parent with null record;
overriding procedure Operation1(This : access Kid1);

type Kid2 is new Parent with null record;
overriding procedure Operation2(This : access Kid2);
```

and the following method body:

```
procedure Operation1 (This : access Parent) is
begin
   Put_Line("P:Operation1");
   This.Operation2;
end Operation1;
```

What happens when the method `Operation1` is called with an object of type `Kid2`? Those accustomed to programming in Java or C# will expect the `Operation2` from `Kid2` to be called. But instead, the `Operation2` of `Grand_Parent` is called. This is because dispatching is not the default behavior in Ada. It occurs when a classwide declaration is present (`'Class`). Since this is of type `Parent` (not `Parent'Class`), `Operation2` for the class `Parent` is called. Since `Parent` inherits `Operation2` from `Grand_Parent`, the `Operation2` defined for `Grand_Parent` is called.

To get dispatching, you should ensure that `'Class` appears in the object's declaration, (possibly by creating a new variable or using a type conversion) as:

```
procedure Operation1 (This : access Parent) is
   Classwide_This : access Parent'Class := This;
begin
   Put_Line("P:Operation1");
   Classwide_This.Operation2;
end Operation1;
```

5.2 Interfaces

C++ allows programmers to have multiple inheritance. That is, an object class can have more than one parent class. The designers of Java decided this created too many difficulties (what if more than one parent has an attribute with the same name, or a method with the same signature?) They instead provided an interface mechanism. Interfaces don't provide any inheritance, but are more like joining clubs than being children. You get to say you're a member of the club if you follow the rules (implement the methods prescribed by the interface). C# and now Ada have joined this bandwagon.

As a simple example, we could say:

```
type Resizeable is interface;
procedure Resize(Object : access Resizeable) is abstract;

type Square is new Shape and Resizeable;
procedure Resize(Object : access Square); -- must be done!!
```

Here `Square` inherits from `Shape`, but also promises to implement all of the methods in the interface `Resizeable`. As such, it is allowed to claim membership in the class `Resizeable` and passes the following test:

```
declare
   S : Square;
begin
   if S in Resizeable'Class then
```

`S` could also be passed to any procedure that required a `Resizeable` as a parameter.

Interestingly, a root tagged type can not implement any interfaces. One key difference between the interface mechanism provided by Ada 2005, and those provided by Java and C#, is that Ada allows interface methods to be specified as **`null`**. If an interface has **`null`** methods, classes that implement the interface do not have to provide implementations for these methods. If the method is not implemented, a method with a **null** body will automatically be created by the compiler. For what are hopefully obvious reasons, only procedures may be declared **`null.`**

```
type Resizeable is interface;
procedure Resize(Object : access Resizeable) is null; -- legal
function Width(Object : access Resizeable)
   return Natural is null; -- illegal!!
```

A more complete description of interfaces is given in Section 2.4 of the Ada 2005 Rationale [Bar05].

6. ASSERTIONS/EXCEPTIONS

It is good software engineering practice to check that the preconditions of a subprogram are met. **`pragma`** `Assert` provides a very convenient mechanism for doing so. For example, if a particular procedure requires that a list contain at least two elements, it can begin with:

```
pragma Assert(L.Next /= null, "List doesn't contain two or more elements");
```

If the assertion fails, the exception `Assertion_Error` is raised. So, this is simply a shorthand for:

```
if not(L.Next /= null) then
   Ada.Exceptions.Raise_Exception(Assertion_Error'Identity,
      "List doesn't contain two or more elements");
end if;
```

Or, to introduce another nice Ada 2005 feature, this could also be written as:

```
if not(L.Next /= null) then
   raise Assertion_Error with "List doesn't contain two or more elements";
end if;
```

Even though it is just a bit of syntactic sugar, the **`pragma`** `Assert` is very convenient, which will hopefully make assertions more prevalent. And, for efficiency, assertion checks can be removed with **`pragma`** `Assertion_Policy(Ignore)`. Thus, we can create lots of assertions for debugging purposes, knowing that when we deliver the final product, we can quickly remove them.

7. NEW LIBRARIES

Ada 2005 comes with a large collection of new packages in the standard library. The most interesting for introductory programmers are likely to be `Ada.Directories`, the `Ada.Containers` hierarchy and the related functions `Ada.Strings.Hash` and `Ada.Strings.Unbounded.Hash`.

`Ada.Directories` contains all of the needed functions for interacting with the OS's hierarchical file system. It contains routines to create and remove directories, and set and change the current directory. Additionally, you can search directories, and find the size and modification time of files. More complete detail is given in Section 7.4 of the Ada 2005 Rationale [Bar05].

The `Ada.Containers` hierarchy contains vectors, doubly-linked lists, hashed maps (a simple hash table), ordered maps (similar to a simple hash table, but can be traversed in order), hashed sets (a collection of unordered values, where a value is either present in the set or not) and ordered sets. For the hashed containers, you can either use the built-in hash functions for strings and unbounded strings (`Ada.Strings.Hash` and `Ada.Strings.Unbounded.Hash`, respectively), or define your own. Full details on these packages can be found in Section 8 of the Ada 2005 Rationale [Bar05].

In addition to the new packages, some small changes have been made to familiar packages. For assignments that involve doing timing tests, the new `Seconds` and `Minutes` in `Ada.Real_Time` may come in handy. There is also a new function `Get_Line` in `Ada.Text_IO` that returns a string. This makes it very easy to get a line from the keyboard without having to worry about a maximum line length, as:

```
declare
   Line : String := Ada.Text_IO.Get_Line;
begin
```

For those who would normally accomplish this kind of thing with unbounded strings, the new `Ada.Strings.Unbounded.Text_IO` package may be a welcome addition.

8. CONCLUSIONS

Ada 2005 provides a large number of new features that will impact CS1 and CS2 courses using Ada. We have provided a quick overview of the changes we expect to have the biggest impact on introductory computer science courses that are taught with Ada. This is more meant to be a quick taste than a full explanation. The interested reader is referred to the Rationale [Bar05] for more detailed information on these features as well as the changes in places such as internationalization, real time and scheduling, complex numbers, and vector and matrix operations.

REFERENCES

[Ada05] *Ada 2005 Language Reference Manual*. http://www.adaic.org/standards/ada05.html.
[Bar05] Barnes, John. 2005. *Rationale for Ada 2005.* http://www.adaic.org/standards/rationale05.html.
[Car06] Carlisle, Martin. *A# Home Page*. http://asharp.martincarlisle.com.
[CSH02] Carlisle, Martin; Sward Ricky and Jeffrey Humphries. "Weaving Ada 95 into the .NET Environment." *Proceedings of SIGAda '02*, Houston TX, December 2002.
[MS05] Miranda, Javier and Edmond Schonberg. 2005. GNAT and Ada 2005. http://www.adacore.com/wp-content/files/attachments/Ada_2005_and_GNAT.pdf.
[Wik05] *Ada 2005 Wiki.* http://en.wikibooks.org/wiki/Ada_Programming/Ada_2005.

A Ada 2005 RESERVED WORDS

abort, abs, abstract, accept, access, aliased, all, and, array, at, begin, body, case, constant, declare, delay, delta, digits, do, else, elsif, end, entry, exception, exit, for, function, generic, goto, if, in, interface, is, limited, loop, mod, new, not, null, of, or, others, out, overriding, package, pragma, private, procedure, protected, raise, range, record, rem, renames, requeue, return, reverse, select, separate, subtype, synchronized, tagged, task, terminate, then, type, until, use, when, while, with, xor