

Re: Question about OO programming in Ada

Source: <http://coding.derkeiler.com/Archive/Ada/comp.lang.ada/2003-11/1100.html>

From: Ludovic Brenta (ludovic.brenta_at_insalien.org)

Date: 11/27/03

Date: 27 Nov 2003 11:15:43 +0100

"Ekkehard Morgenstern" writes:

[...]

> *"Robert I. Eachus" writes:*

[...]

>> *abstract function Register(Owner: in out Person'Class;*

>> *The_Vehicle: in out Vehicle)*

>> *return Registration'Class;*

>>

>> *Perhaps when you finish, you will end up returning a Registration_Ref,*

>> *but so far it isn't necessary to make that type public. (Notice that*

>> *you can always write Register(Someone, Car)'Access, if you really need*

>> *to get an access value to assign to a Registration_Ref, or in some cases*

>> *Register(Someone, Car)'Unchecked_Access.)*

>

> *Thanks. That's good to know. What does "Unchecked_Access" do?*

Assuming

```
The_Registration : Registration'Class := Register (Someone, Car);
```

The_Registration'Access returns an access to The_Registration, but the language rules make sure that there cannot be any "dangling references" to The_Registration. In other words, the compiler checks that the lifetime of The_Registration exceeds the lifetime of the The_Registration'Access. In a few cases, the compiler cannot ensure this statically at compile time, so it inserts run-time checks instead. These checks raise Program_Error at run time if they fail.

With Unchecked_Access, these rules are relaxed. Essentially, if you use Unchecked_Access, you are telling the compiler that you understand the issue of dangling references, and that you are dealing with this issue yourself.

For more insight, see the reference manual: 3.10.2 and 13.10. The RM is available on the web:

<http://www.adaic.org/standards/95lrm/html/RM-TTL.html>
<http://www.adaic.org/standards/95lrm/html/RM-3-10-2.html>
<http://www.adaic.org/standards/95lrm/html/RM-13-10.html>

[...]

> > *Later you will write (child) packages to deal with cars, trucks, buses,*
>
> *Why child packages? Does the package hierarchy matter for derived classes?*

Child packages can see the private part of their parent's specification. You use them when you require access to the internal definition of a data structure while still enjoying the benefits of separate compilation and of specifications. For example, if you have a type `Registration_Ref` which is in the private part of the parent package, then you can use that type in the child packages.

In OO programming, this is how you'd implement the equivalent of C++ "protected" methods, e.g.

```
package Parent is
  type Root_Class is tagged private;
  procedure Public_Method (Object : in out Root_Class);
private
  procedure Protected_Method (Object : in out Root_Class);
  type Root_Class is tagged record
    -- ...
  end record;
end Parent;

package Parent.Child is
  type Derived_Class is new Root_Class with private;
  procedure Public_Method (Object : in out Derived_Class);
  -- override the parent's method.
private
  type Derived_Class is new Root_Class with record
    -- ...
  end Derived_Class;
end Parent.Child;

package body Parent.Child is
  procedure Public_Method (Object : in out Derived_Class) is
  begin
    Protected_Method (Object); -- visible from child package
  end Public_Method;
end Parent.Child;
```

> > *ride motorcycles. This can be determined by nested dispatching, or more*
> > *usually by "if Owner in Motorcycle_Driver'Class then..."*
>
> *This is really a great feature! So I can practically test if any object is a*

> *member of a particular class, or set of classes?*

Yes. But note that, in Ada, a class is a set of types, whereas in C++ a class is a set of objects. So a more appropriate wording for the feature would be that you can test that an object belongs to a type or a set of types (with 'Class). Here, "if Owner in Motorcycle_Driver'Class then..." really means "if Owner is a Motorcycle_Driver or any type derived from Motorcycle_Driver".

[...]

> *I read in the OO part of the Rationale that a classwide type is actually represented internally with a set of key/value pairs, right? What kinds of other applications does that have?*

>

>> *But the second instance, the use of a classwide result type is more interesting. This is not to cause dispatching, but to actually return an object whose class is not known at compile time. A typical usage is:*

>>

>> *procedure Something (...) is*

>> *The_Registration: Registration'Class*

>> *:= Register(The_Owner, The_Vehicle);*

>> *begin*

>> *-- further operations on The_Registration, which may involve*

>> *-- dispatching calls.*

>> *end Something;*

>

> *I still wonder if The_Registration is actually a reference, or is it the*

> *object returned by Register()?*

It is the object returned by Register. But the actual type of the object is not known at compile time. All we know is that The_Registration is either a Registration, or any type derived from Registration.

> *I.e. if I wanted only a reference to the object, would I have to use an access to Registration'Class?*

Yes.

> *I wonder about this stuff because I'd like implement a linked list package, which provides one set of subprograms for all types of derived linked lists and list nodes.*

Here is a quick program that I wrote for the same purposes a while ago. It is a "polymorphic list", i.e. a linked list containing nodes of different types derived from List_Item. There are primitive operations on List that do not care what the actual type of each node is; others (the Put procedures) are primitive operations on the specific node types. These are dynamically dispatching.

with Ada.Text_IO;

procedure Polymorphic_List is

package List is

```
type List_Item;
type List_Item_Access is access List_Item'Class;
type List_Item is abstract tagged record
  Next_Item : List_Item_Access;
end record;

function Find_Last_Item(In_List : in List_Item_Access)
  return List_Item_Access;

procedure Add_Item(To : in out List_Item_Access;
  What : in List_Item_Access);

procedure Put(What : in List_Item) is abstract;
end List;
```

package body List is

```
function Find_Last_Item(In_List : in List_Item_Access)
  return List_Item_Access
is
  Previous : List_Item_Access := null;
  Current : List_Item_Access := In_List;
begin
  while Current /= null loop
    Previous := Current;
    Current := Current.Next_Item;
  end loop;
  return Previous;
end Find_Last_Item;

procedure Add_Item(To : in out List_Item_Access;
  What : in List_Item_Access)
is
  Last_Item : List_Item_Access := Find_Last_Item(To);
begin
  if What /= null then
    if Last_Item = null then
      To := What;
    else
      Last_Item.Next_Item := What;
    end if;
  end if;
end Add_Item;

end List;
```

```
package Parts is
  type Part_Identifier is (Part_A, Part_B);

  type Part(Text_Size : Positive) is new List.List_Item with record
    Part_Id : Part_Identifier;
    Text : String(1 .. Text_Size);
  end record;

  procedure Put(What : in Part);

  type Component is new List.List_Item with null record;

  procedure Put(What : in Component);

  type Component_B is new List.List_Item with null record;

  procedure Put(What : in Component_B);
end Parts;
```

```
package body Parts is
  procedure Put(What : in Part) is
  begin
    Ada.Text_Io.Put_Line("Part found of type " &
      Part_Identifier'Image(What.Part_Id) &
      " with description """" &
      What.Text & """"");
  end Put;

  procedure Put(What : in Component) is
  begin
    Ada.Text_Io.Put_Line("Component found.");
  end Put;

  procedure Put(What : in Component_B) is
  begin
    Ada.Text_Io.Put_Line("Component_B found.");
  end Put;
end Parts;

The_Linked_List : List.List_Item_Access := null;

Part_Description : constant String := "Some text for part A";
begin -- processing for Polymorphic_List

  -- TODO : initialise all the fields of Part.
  List.Add_Item(To => The_Linked_List,
    What => new Parts.Part(Text_Size => Part_Description'Length));
```

```
List.Add_Item(To => The_Linked_List,
             What => new Parts.Component_B);

List.Add_Item(To => The_Linked_List,
             What => new Parts.Component);

declare
  Iterator : List.List_Item_Access := The_Linked_List;
  use List; -- bring in "/"= and Put
begin
  while Iterator /= null loop
    Ada.Text_Io.Put_Line("Found an item:");
    Put(Iterator.all);
    Iterator := Iterator.Next_Item;
  end loop;
end;
end Polymorphic_List;

[...]
```

> *Now, if I had a function like this:*

```
>
> function Next( N : in out Node'Class ) return Node'Class;
>
> (btw, is it ok to use "in out" in this case with a function?)
>
> Can I return an object of type Node'Class, or do I have to return an access
> to it?
```

You must return an object of type Node, or of any type derived from Node. Not an access. You cannot use "in out" in functions.

```
> And the Node implementation, can it look like this:
>
> type Node is tagged limited
> record
> Succ : Node'Class;
> Pred : Node'Class;
> end record;
```

I don't think so. For one thing, the size of a Node would vary depending on the actual type of Succ and Pred. For another, not all types in Node'Class are known, as more types may be added in the future. This is where you need an "access Node'Class".

```
[...]
```

> *I gather from previous discussions that Node'Class represents an actual*

> *object, not a reference to it, so I'd have to use access types in this case,*

> *which would give*

```
>
> type Node_Ref is access Node'Class;
>
```

> *type Node is tagged limited*
> *record*
> *Succ : Node_Ref;*
> *Pred : Node_Ref;*
> *end record;*
>
> *Or do I err?*

This is correct, but to be more precise, Node'Class represents any specific type (not object) in the hierarchy of types rooted at Node.

> *If I use the latter version, and keep the function Next() as defined above,*
> *would the object be copied when it is returned? Or is that precluded by the*
> *fact that it's a limited type?*

All tagged and limited types are passed by reference no matter what, so no copying would take place.

> *But if I modify Next() to be something like*
>
> *function Next(N : Node_Ref) return Node_Ref;*
>
> *how can I avoid casting N and the return value from a derived class?*
>
> *What's the optimal way in Ada to do this?*

Here, you wouldn't have to cast, because all objects of type Node or derived from Node will have a Next member. Alternatively, Next could call a dispatching operation on Node_Ref.

[...]

> > *First of all, how do I accomplish returning a reference in Ada?*
> >
> > *Whether an object is passed by value, value–return, or reference depends*
> > *on the object's class, not on whether the parameter is in, in out, or*
> > *out. In parameters can be referenced but not changed, in out parameters*
> > *can be referenced and changed, and for out parameters the attributes are*
> > *defined at the point of the call, but there may not be an initial value.*
>
> *But what about return values from functions? Or does the return value in a*
> *function correspond to an "out" parameter in a procedure?*

Pretty much, yes. At least in my understanding.

[...]

> > *There are cases where the access parameter won't cause any*
> > *additional difficulties, and other cases where it will mean that*
> > *you need to explicitly free the storage to avoid memory leaks.*
>
> *Yeah, I know that problem can be addressed by creating an instance*
> *of the Unchecked_Deallocation procedure.*

comp.lang.ada: Re: Question about OO programming in Ada

There is an alternative unique to the Ada language. As per 13.11(18), you can declare an access type in a scope that is enclosed inside the scope of the object type. The compiler inserts an automatic deallocation at the point where the access type goes out of scope. This is a little bit tricky to understand, so here is an example.

```
with Ada.Finalization;
package Controlled is
  type Object is tagged private;
private
  type Object is new Ada.Finalization.Controlled with null record;
  procedure Initialize (O : in out Object);
  procedure Adjust (O : in out Object);
  procedure Finalize (O : in out Object);
end Controlled;
```

```
with Ada.Text_IO;
package body Controlled is
  procedure Initialize (O : in out Object) is
  begin
    Ada.Text_IO.Put_Line ("O is being initialized.");
  end Initialize;
```

```
  procedure Adjust (O : in out Object) is
  begin
    Ada.Text_IO.Put_Line ("O is being adjusted.");
  end Adjust;
```

```
  procedure Finalize (O : in out Object) is
  begin
    Ada.Text_IO.Put_Line ("O is being finalized.");
  end Finalize;
end Controlled;
```

```
with Ada.Text_IO;
with Controlled;
procedure Storage is
  procedure Proc is
    type Object_Access is access Controlled.Object;
    O : Object_Access := new Controlled.Object; -- O allocated.
  begin
    Ada.Text_IO.Put_Line ("Within Proc.");
  end Proc; -- type Object_Access goes out of scope here: O deallocated.
```

```
begin
  Ada.Text_IO.Put_Line ("Beginning of Storage.");
  Proc;
  Ada.Text_IO.Put_Line ("End of Storage.");
end Storage;
```

At execution this gives:

```
$. /storage
Beginning of Storage.
O is being initialized.
Within Proc.
O is being finalized.
End of Storage.
```

(this with GNAT 3.15p on GNU/Linux).

Notice how O is being finalized? Because the type `Object_Access` goes out of scope, the compiler causes O to be deallocated for you. No need for `Unchecked_Deallocation` here, and no memory leak.

- > *BTW, how do I use the Finalization package? I'd like to use things like*
- > *constructors and destructors sometimes to be able to initialize / cleanup*
- > *objects when they're created or destroyed.*

See above.

- > *BTW, can I instantiate a generic package in a procedure body (initialized*
- > *with a value passed as a parameter)? (I didn't try yet)*

Yes, but only in a declare block. e.g.

```
procedure Proc is
  -- instantiate here...
begin
  -- statements
  declare
    -- or here.
  begin
    -- statements
  end;
  -- statements
end Proc;
```

- > *The problem with arrays is that they need to be preallocated to a*
- > *particular size, which can consume a lot of memory. Lists and Nodes*
- > *occupy only the memory that they actually use.*
- >
- > *And which method is faster?*

Arrays are usually faster, because memory allocation is a rather expensive operation. The run-time system does quite a lot of bookkeeping for each allocation and deallocation, so if you allocate N objects in one time it is better.

Note that arrays can be sized dynamically, e.g.

```
type Array_Of_Objects is array (Natural range <>) of Object;
```

```
procedure Proc (Number_Of_Objects_To_Allocate : in Natural) is
  A : Array_Of_Objects (1 .. Number_Of_Objects_To_Allocate);
begin
  -- process the objects
end Proc;
```

If you need a structure that can grow and shrink, or if your arrays can grow very large, one strategy could be a linked list of arrays.

> > *(And you probably chose those packages from a library instead of*
> > *"rolling your own.")*
>
> *What kind of library would you recommend to me? I'd need some linked lists,*
> *queues, containers and so on, but I'd like to have versions that are*
> *completely generic.*

Personally I would recommend Charles
(<http://home.earthlink.net/~matthewjheaney/charles>) but there are others.

[...]
> *I plan to use Ada for a major software project in which I might even have to*
> *implement an Ada compiler for a virtual machine that I've designed myself,*
> *and which I might also implement in Ada.*

This is indeed a major undertaking. Perhaps re-targetting GNAT would be a cheaper alternative which you would consider. At least you'd start with the parser and most of the run-time library already written. You'd "only" need a new code generator and a port of the RTL for the VM.

[...]
> *How do I avoid casting access types between derived classes?*

By using dynamically dispatching subprograms?

[...]
> *All of Ada's features are very promising for the development of*
> *well-designed bug-free software. :-)*
> *(because, you know, I'm not worried about myself there, because I can write*
> *bug-free software in any programming language, but if I write a development*
> *environment with its own language, I want to choose a language that enforces*
> *or at least encourages bug-free programming for those less experienced than*
> *people like me, also, and Ada seems to be the ideal choice! Even if I*
> *support other languages -- if my VM is designed for Ada, and Ada is the*
> *prime language for it, then all other languages can benefit from it as*
> *well.)*

Even for very experienced people, I think Ada still helps write bug-free software. Of course, you can write bug-free software in any language. But Ada reduces the cost of doing that.

comp.lang.ada: Re: Question about OO programming in Ada

--

Ludovic Brenta.