



IBM Software Group

Object-Oriented Programming and Structure Control in Ada 2005

Rational software



@business on demand software

Agenda

- Nested type extensions
 - Object factory
 - Explicit control of overriding
 - Multi-package type structures
 - Access to private units in private parts
 - Making limited types useful



Nested Type Extensions

- Ada 95 requires type extension to be at same “accessibility level” as its parent type
 - ▶ Cannot extend a type in a nested scope
- Restriction removed in Ada 2005
 - ▶ Can extend inside a subprogram, task, protected, or generic body
 - ▶ Still may not extend formal type inside generic body because of possible contract violations
 - Actual type might have additional operations requiring overriding
 - ▶ Checking performed on function return and allocators
 - May not create an object that might outlive its type
- Enables instantiation of generics in nested scopes, even if they use finalization, streams, or storage pools



Agenda

- Nested type extensions
- Object factory
- Explicit control of overriding
- Multi-package type structures
- Access to private units in private parts
- Making limited types useful



Object Factory

- Makes it possible to create an object given its tag
 - ▶ Useful for input operations

- Example would be the T'Class'Input stream attribute
 - ▶ Default implementation reads "tag" name from stream, and then dispatches to the corresponding <type>'Input routine
 - ▶ Currently no obvious way for user to implement such a dispatch

- Supported by a new kind of generic formal subprogram
 - ▶ is abstract



Object Factory (cont'd)

- New built-in generic function

```
generic
  type T (<>) is abstract tagged limited private;
  type Parameters (<>) is limited private;
  with function Constructor (Params : not null access Parameters)
    return T is abstract;
function Ada.Tags.Generic_Dispatching_Constructor
  (The_Tag : Tag;
   Params  : not null access Parameters) return T'Class;
```

- Each descendant of T must implement a Constructor operation which takes a single (access) parameter of type Parameters
- At execution, the tag is used to select the particular Constructor operation to call, passing Params
 - ▶ Can be used to provide a user-implemented T'Class'Input



Reading a Circle with the Object Factory

- Parameters is the file from which the Constructor function read the object

```
function Constructor (File : not null access Text_IO.File_Type)
    return Object is abstract;
function Constructor (File : not null access Text_IO.File_Type)
    return Circle is ...
```

- Instantiate factory

```
function Object_Factory is new Generic_Dispatching_Constructor
    (Object, Text_IO.File_Type, Constructor);
```

- Set appropriate tags

```
for Circle'External_Tag use "CIRCLE";
```

- Do it

```
An_Object : Object'Class :=
    Object_Factory (Tags.Internal_Tag (Text_IO.Get_Line (F)), F'Access);
```



Agenda

- Nested type extensions
- Object factory
- ❖ ■ Explicit control of overriding
- Multi-package type structures
- Access to private units in private parts
- Making limited types useful



Overriding Indicators

- Two dangers when overriding
- Spell operation name incorrectly or get profile wrong
 - ▶ Results in a new operation and not an overriding

```
type Thing is new Controlled with ...  
procedure Finalise (T : in out Thing);
```

- ▶ Finalise should be Finalize — when object of type Thing is finalized, nothing happens since the inherited null procedure is called
- Inadvertently use the name and profile of an existing primitive operation
 - ▶ Results in an overriding and not a new operation
 - ▶ Unwittingly redefined the behaviour of some operation



Use an Indicator

- Should write:

❖ `overriding`
`procedure Finalize (Object: in out Thing);`

- Or:

❖ `not overriding`
`procedure Operation (...);`

- Compiler will tell us if we get it wrong
 - ▶ Overriding indicators are optional, but recommended
 - ▶ Safety improvement



Agenda

- Nested type extensions
- Object factory
- Explicit control of overriding
- ❖ Multi-package type structures
- Access to private units in private parts
- Making limited types useful



Multi-Package Cyclic Type Structures

- Impossible to declare cyclic type structures across library package boundaries
 - ▶ Each type must be compiled before the types that depend upon it!
- Problem existed in Ada 83, but more prominent in Ada 95
- Hierarchical library units and tagged types favor a model where each library unit represents one abstraction of the problem domain
- Workarounds are awkward
 - ▶ Mutually-dependent types have to be lumped in a single library unit...
 - ▶ ... or unchecked programming has to be used



The Cyclic Type Conundrum

```
with Department;  
package Employee is  
  type Object is tagged private;  
  procedure Assign_Employee (Who           : in out Employee.Object;  
                             To_Department : in out Department.Object);  
private  
  type Object is tagged  
    record  
      Assigned_To : access Department.Object;  
    end record;  
end Employee;
```

Illegal circularity!

```
with Employee;  
package Department is  
  type Object is tagged private;  
  procedure Choose_Manager (For_Department : in out Department.Object;  
                           Who           : in out Employee.Object);  
private  
  type Object is tagged  
    record  
      Manager : access Employee.Object;  
    end record;  
end Department;
```



Solution: Limited With Clauses

- Gives visibility to a *limited view* of a package
 - ▶ Contains only types and nested packages
 - ▶ Types behave as if they were incomplete
 - ▶ Cycles are permitted among limited with clauses
 - ▶ Imply some kind of “peeking” before compiling a package

- Tagged incomplete type
 - ▶ Incomplete type whose completion must be tagged
 - ▶ May be used as parameter and as prefix of ' Class

- No syntax for declaring a limited view: implicitly created by the compiler



Example of a Limited View



```
package Department is
  type Object is tagged;
end Department;
```

Implicit!

```
with Employee;
package Department is
  type Object is tagged private;
  procedure Choose_Manager (For_Department : in out Department.Object;
                           Who              : in out Employee.Object);

private
  type Object is tagged
  record
    Manager : access Employee.Object;
  end record;
end Department;
```



Solving the Cyclic Type Conundrum

```
package Department is
  type Object is tagged;
end Department;
```

Implicit!



```
limited with Department;
package Employee is
  type Object is tagged private;
  procedure Assign_Employee (Who           : in out Employee.Object;
                             To_Department : in out Department.Object);

private
  type Object is tagged
  record
    Assigned_To : access Department.Object;
  end record;
end Employee;
```

```
with Employee;
package Department is
  type Object is tagged private;
  procedure Choose_Manager (For_Department : in out Department.Object;
                           Who            : in out Employee.Object);

private
  type Object is tagged
  record
    Manager : access Employee.Object;
  end record;
end Department;
```



Incomplete Types and Dereferencing

- Access types declared using the limited view are access-to-incomplete
 - ▶ Would not be very useful: numerous restrictions on incomplete types
- Become access-to-complete in the presence of a nonlimited with clause

```
limited with Department;  
package Employee is  
  ..  
private  
  type Object is tagged  
  record  
    Assigned_To : access Department.Object;  
  end record;  
end Employee;
```

```
with Department;  
package body Employee is  
  An_Employee : Employee.Object := ...;  
  Her_Department : Department.Object := An_Employee.Assigned_To.all;  
  ..  
end Employees;
```

This with clause ... makes this dereference legal



Language Design Principles and Restrictions

- Detect errors early
 - ▶ References to types declared in limited views checked at compile time
- Limited view must be constructible from purely syntactic information
 - ▶ Constructs that require name resolution are not part of the limited view
 - ▶ Package renamings and instantiations
 - ▶ Tagged-ness may be determined syntactically
- Limited with clauses used to resolve circularities, not to restrict visibility
 - ▶ Limited with clause not allowed if there is already a normal with clause
 - ▶ Limited with clause not allowed on a body
 - ▶ Limited with clause not allowed with use clauses



Agenda

- Nested type extensions
- Object factory
- Explicit control of overriding
- Multi-package type structures
- ❖ Access to private units in private parts
- Making limited types useful



Access to Private Units in Private Parts

- Private child packages allow decomposition and hiding of the implementation details
 - ▶ Not visible to the outside world
- Only private packages and bodies can reference a private child
- Often convenient for public packages to use implementation details without making them visible
- Impossible to use a private unit in declarations appearing in the private part of a public package



Solution: Private With Clause

- Private with clause gives visibility to a unit, but only at the beginning of the private part

```
private package App.Secret_Details is
  type Inner is ...;

  ... -- Various operations on Inner, etc.
end App.Secret_Details;
```



```
private with App.Secret_Details;
package App.User_View is
  type Outer is private;

  ... -- Various operations on Outer visible to the user
  -- Type Inner may not be used here.
private
  type Outer is
    record
      Secret : Secret_Details.Inner; -- OK to use Inner here.
      ...
    end record;
  ...
end App.User_View;
```



Agenda

- Nested type extensions
- Object factory
- Explicit control of overriding
- Multi-package type structures
- Access to private units in private parts
- ❖ ■ Making limited types useful



Making Limited Types Useful

- Limited types prevent copying of values
 - ▶ Have limitations unrelated to copying
- Aggregates not available: no full coverage checking
- Functions cannot be used to construct values of limited types
 - ▶ Can only return existing global objects: not too useful
 - ▶ Mysterious “return by reference” mechanism
- Limited types are unnecessarily hard-to-use
 - ▶ Restrictions do not improve safety
 - ▶ Types often made nonlimited to avoid running into difficulties
- Lift unnecessary restrictions while preserving safety
 - ▶ In particular, prevent copying of values



Solution: Aggregates for Limited Types

- Aggregates only allowed for initialization, not general assignment
 - ▶ Must be built in place
- New syntax for components for which no value can be written
 - ▶ Tasks, protected objects
 - ▶ Also causes default initialization

```
protected type Semaphore is ...;
```

```
type Guarded_Object is limited
```

```
  record
```

```
    Guard      : Semaphore;
```

```
    Count      : Integer;
```

```
    Finished   : Boolean := False;
```

```
  end record;
```



```
X : Guarded_Object := (Guard      => <>, -- Create a new Semaphore.  
                      Value      => 0,  
                      Finished => <>); -- Gets False.
```



Solution: Functions Returning Limited Types

- Again, only allowed for initialization
- New form of return statement to build an object directly in its final resting place
 - ▶ No copying of the result of the function



```
function Read_Guarded_Object return Object is  
begin  
    return New_Object : Object do  
        Ada.Integer_Text_IO.Get (New_Object.Value);  
        New_Object.Finished := New_Object.Value < 0;  
    end return;  
end Random_Object;
```

