FOSDEM09 7+8 February
www.fosdem.org Brussels, Belgium

The Object-Oriented Programming Model in Ada 2005

J-P. Rosen
Adalog
www.adalog.fr

Foreword

- Ada's model is quite different from other languages.

  (that's why you are here :-)

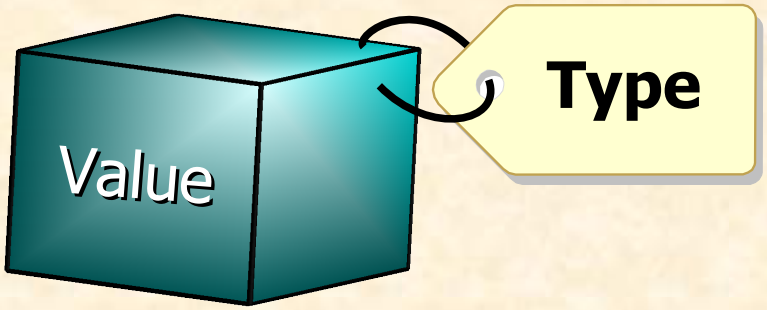
- Surprise n°1:

  Ada has no syntactic construct called "class"

Tagged types

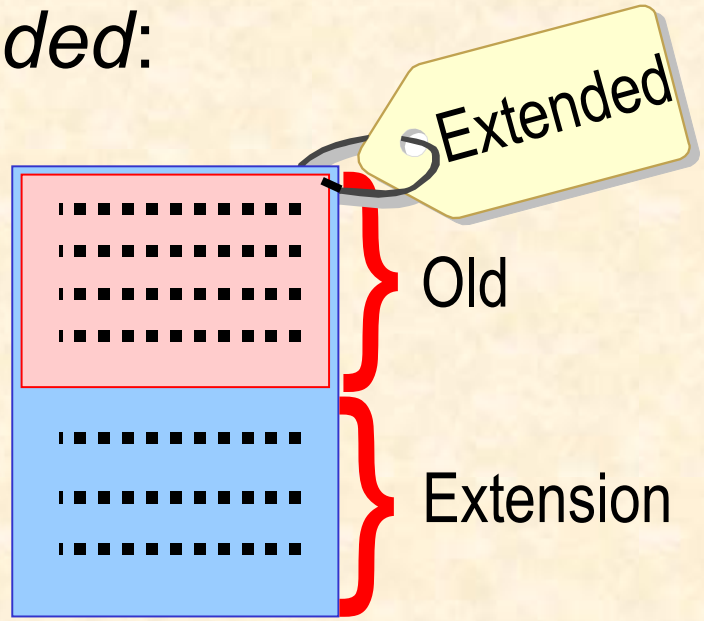
- Like a record:

```
type T is tagged
   record
      ...
   end record;
```



- A tagged type can be *extended*:

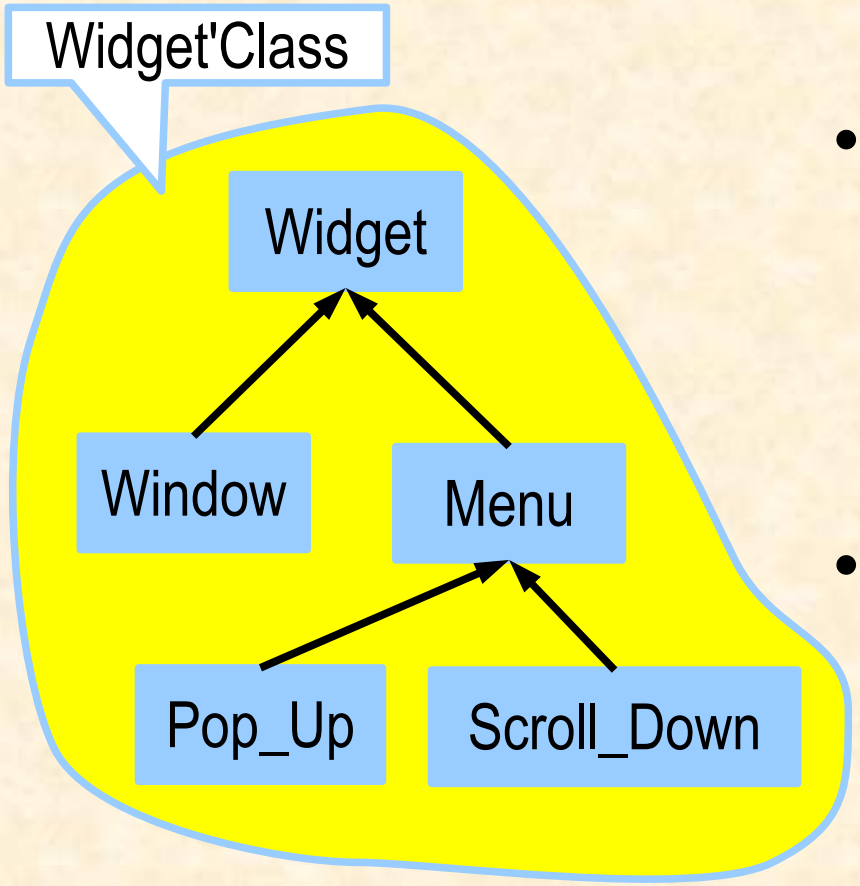
```
type Extended is new T with
   record
      ...
   end record;
```



Tagged types and privacy

```
package Demo is
   type Hidden is tagged private;

   type Public is tagged
      record
          Root_Component : Root_Component_Type;
      end record;

   type Public_With_Public_Components is new Public with
      record
         Some_Component : Some_Type;
      end record;

   type Public_With_Hidden_Components is new Public with private;

   type Hidden_With_Public_Components is new Hidden with
      record
         Some_Component : Some_Type;
      end record;

   type Hidden_With_Hidden_Components is new Hidden with private;
private
  ...
end Demo;
```

Class-wide types



- A type defines a set of values
- A class-wide type exists for every tagged type
  - Its values are the union of the values of the type and all of its derived types
- Clearly separate:
  - Specific type: a node
  - Class-wide type: the subtree defined by this node

Dynamic typing

- Strong typing: an object of a specific type contains only values from that type (and only that type)
- An object of a class-wide type holds any value from the class

```
function Object_Factory (...) return Widget.Instance'Class;

My_Window  : Widget.Window.Instance;  -- Only a window

Any_Widget : Widget.Instance'Class := Object_Factory (...);
```

Dynamic typing

- Strong typing: an object of a specific type contains only values from that type (and only that type)
- An object of a class-wide type holds any value from the class

```
function Object_Factory (...) return Widget.Instance'Class;

My_Window  : Widget.Window.Instance;   -- Only ...

Any_Widget : Widget.Instance'Class := Object_Factory (...);
```

Pointers not needed

Primitive vs. class-wide operation

- A package binds a type to its "primitive" operations (aka *methods*)

```
package Widget is
   type Instance is abstract tagged
      record
         X,Y : Coordinates;
      end record;
   procedure Paint (This : Widget.Instance);
   procedure Erase (This : Widget.Instance);
   ...
end Figure;
```

- A *subprogram* may operate on a class-wide type

```
procedure Move (Item : in out Widget.Instance'Class;
                X, Y : in     Coordinates) is
begin
   Erase (Item);
   Item.X := X;
   Item.Y := Y;
   Paint (Item);
end Move;
```

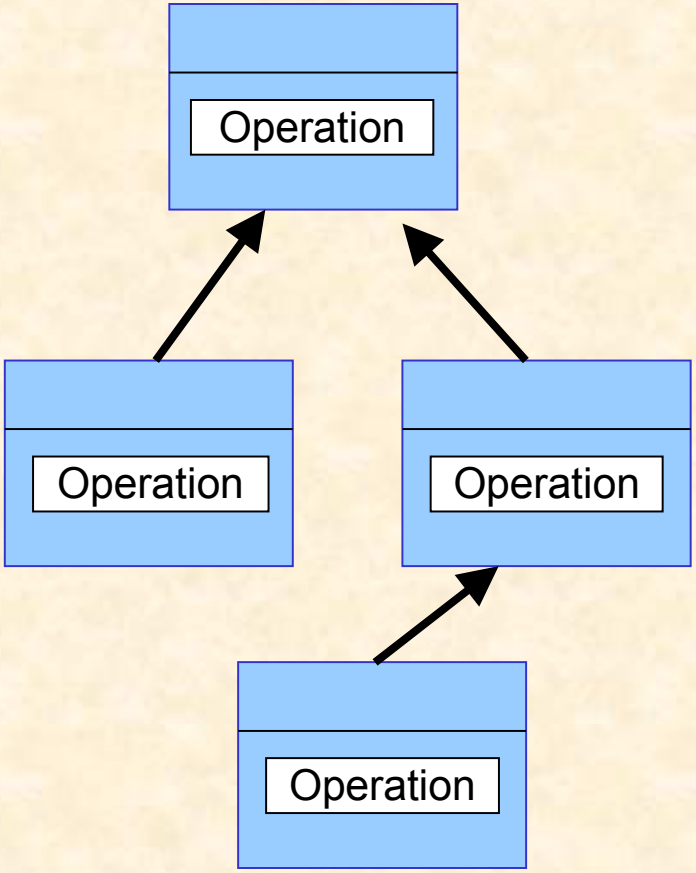
Static and dynamic binding

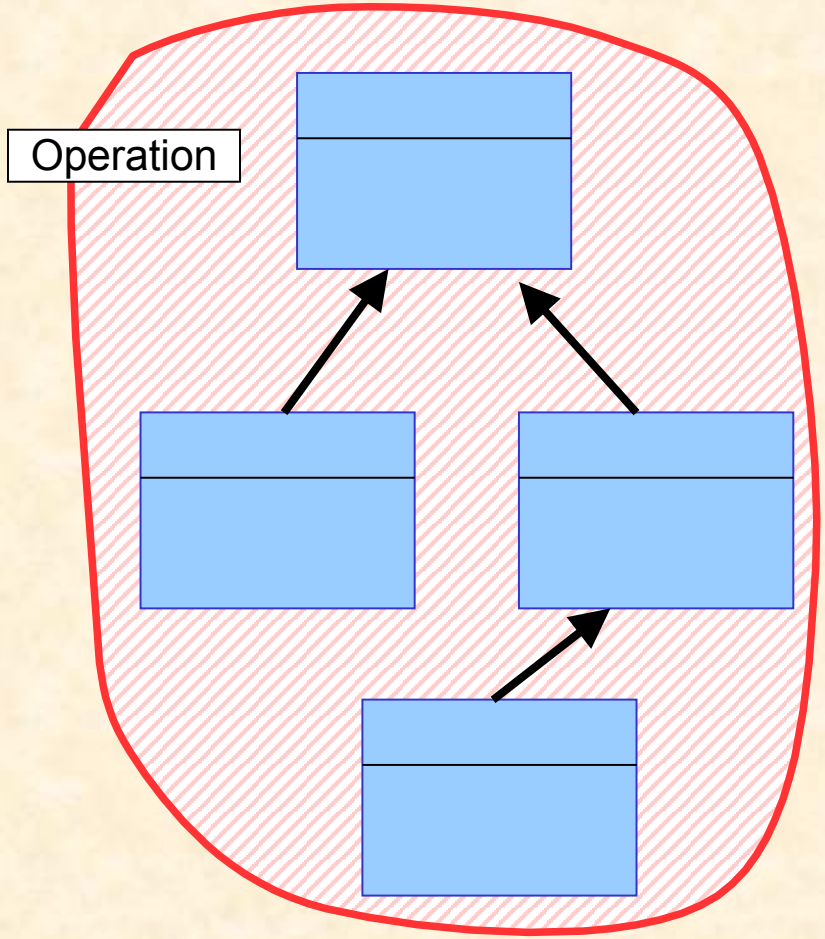
```
procedure Move (This : in out Widget.Instance'Class;
                X, Y : in     Coordinates) is
begin
   This.Erase;
   This.X := X;
   This.Y := Y;
   This.Paint;
end Move;
```

```
  P : Widget.Menu.Pop_Up.Instance;
  W : Widget.Window.Instance;
begin
   P.Paint;  -- or: Paint (P);
   W.Paint;  -- or: Paint (W);

   Move (P, X => 23, Y => 45);
   Move (W, Y => 19, X => 23);
  ...
```

Primitive vs.class-wide operation

- Primitive:
  - ☞inherited, redefinable
- Class-wide:
  - ☞Same operation for everybody



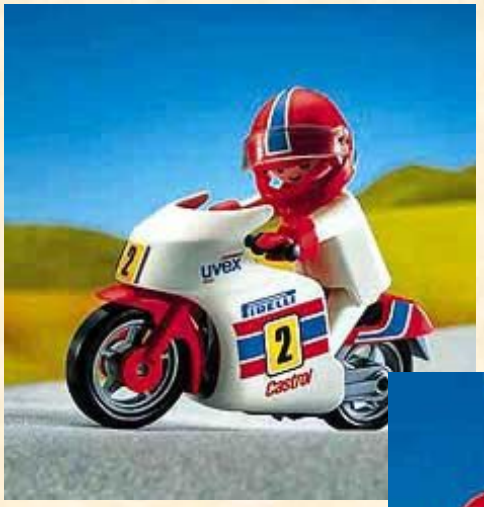


OOP and pointers

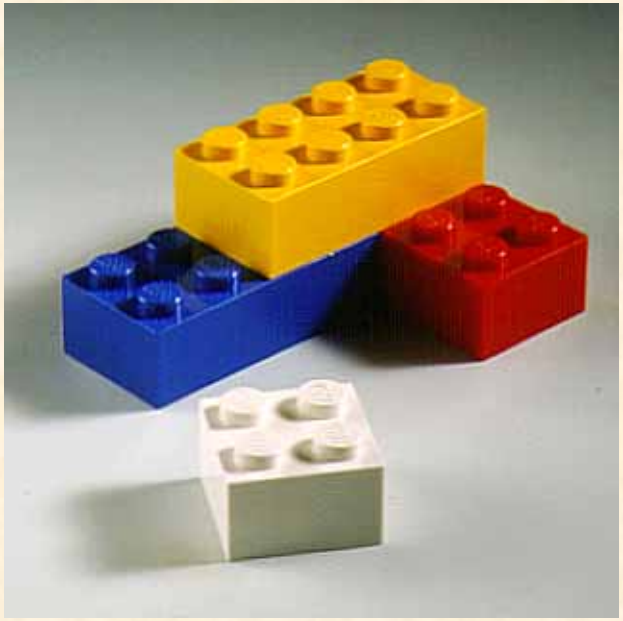
- All combinations are available

```
type     Root     is tagged ...;
type     Acc_Root is access Root;
subtype  Tree     is Root'Class;
type     Acc_Tree is access Tree;


A : Root;                    -- Exactly this (specific) type;
                             -- On the stack

B : Acc_Root := new Root;    -- Exactly this (specific) type;
                             -- On the heap

C : Tree := ...;             -- Any type derived from Root;
                             -- On the stack

D : Acc_Tree := new ...;     -- Any type derived from Root;
                             -- On the heap
```

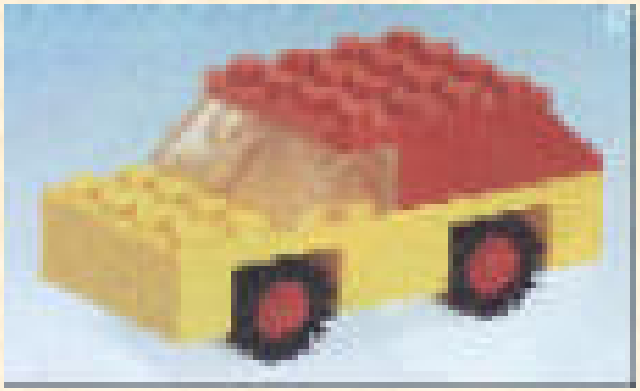
The building-block approach



The building-block approach



The building-block approach



The building-block approach

The building-block approach

The "class" design pattern

- Packages support encapsulation
- Tagged types support dynamic binding
- A class = Encapsulation + dynamic binding
  - ☞ Design pattern: a tagged type in a package

```
package Widget is
   type Instance is tagged private;
   procedure Paint (This : Instance);
   ...
private
   ...
end Widget;
```

```
package Widget.Menu is
   type Instance is new Widget.Instance with private;
   procedure Paint (This : Instance);
   ...
private
   ...
end Widget.Menu;
```

Facets

```
package Item is
  type Instance is tagged ...
  -- Operations on Item.Instance
end Item;

with Item;
generic
  type An_Item is new Item.Instance with private;
package DeLuxe_Item is
  type Instance is new An_Item with ...
  -- Operations on DeLuxe_Item.Instance
end DeLuxe_Item;

with Item;
package Television is
  type Instance is new Item.Instance with ...
  -- Operations on Television.Instance
end Television;

with DeLuxe_Item;
package Television.DeLuxe is
   new DeLuxe_Item (Television.Instance);
```

Class data and methods

```
package Counting_Class is
   type Instance is tagged ....
  -- Methods...

  type Calls_Range is range 0..100_000;
  function Counter return Calls_Range;

end Counting_Class;

package body Counting_Class is
  The_Counter : Calls_Range := 0;

  function Counter return Calls_Range is
  begin
    return The_Counter;
  end Counter;

  -- Body of methods...

end Counting_Class;
```

Private inheritance

```
package Graphical_Object is
   type Instance is abstract tagged private;
   procedure Draw (Object : Instance) is abstract;
private
   ...
end Graphical_Object;

with Graphical_Object;
package Numbered_Object is
   type Instance is new Graphical_Object.Instance with
      record
         Number : Natural := 0;
      end record;
  ...
end Numbered_Object;

with Graphical_Object, Numbered_Object;
package Square is
   type Instance is new Graphical_Object.Instance with private;
   procedure Draw (Object : Instance);
private
  type Instance is new Numbered_Object.Instance with ...;
end Square;
```

The user sees only the properties of a Graphical_Object.
The body of Square can use properties inherited from Numbered_Object.

Interfaces

- A special case of *abstract* tagged type
  - An interface has no components
  - Methods of an interface must be abstract or null

```
with Ada.Text_IO; use Ada.Text_IO;
with Calendar;
package Persistance is
  type Services is interface;

  procedure Read  (F    : in  File_Type;
                   Item : out Services) is abstract;
  procedure Write (F    : in  File_Type;
                   Item : in  Services) is abstract;
  procedure Set_Expiration (To_Date : in Calendar.Time) is null;

end Persistance;
```

Using interfaces

- A tagged type can be derived from one tagged type and several interfaces

```
type Persistant_Window is
  new Widget.Window.Instance and Persistance.Services;
```

- An interface can be derived from several interfaces

```
type Printable_Persistant is interface
   and Persistance.Services
   and Printable.Services;
```

- An interface has an associated class-wide type as well

```
procedure Save (This : Persistance.Services'Class);
```

Special interfaces

- Protected interfaces

```
type Semaphore is protected interface;
procedure P (This : in out Semaphore) is abstract;
procedure V (This : in out Semaphore) is abstract;
```

  - ☞ Must be implemented by a protected type

- Task interfaces

```
type Buffer is task interface;
procedure Get (This : Buffer; Item : out Data) is abstract;
procedure Put (This : Buffer; Item : in  Data) is abstract;
```

  - ☞ Must be implemented by a task

- Synchronized interfaces

```
type Barrier is synchronize interface;
procedure Wait (This : Barrier) is abstract;
```

  - ☞ Can be implemented by either a protected or task type

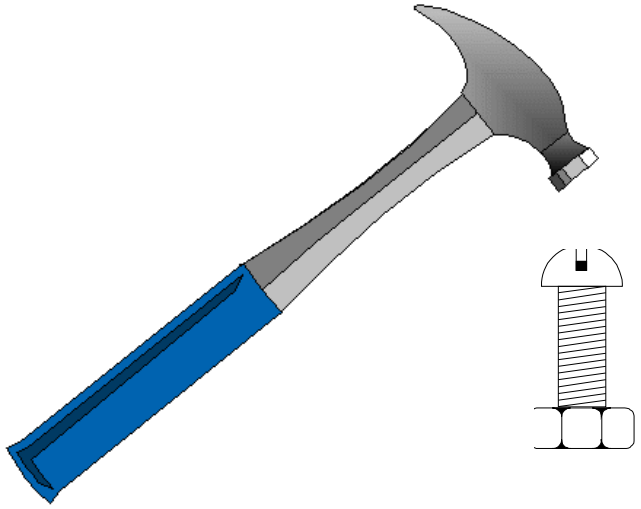
OOP in Ada vs. other languages

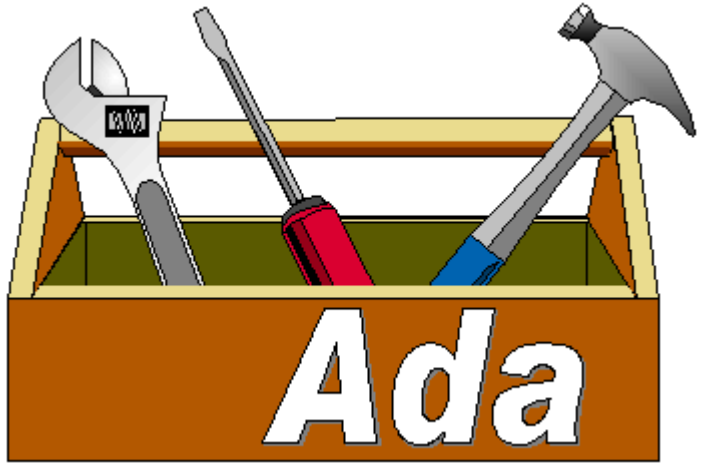
- In other languages, classes are used to:
  - ☞Define modules
    - Ada has packages
  - ☞Reuse algorithms
    - Ada has generics
  - ☞Simulate concurrency
    - Ada has tasking
  - ☞Control visibility
    - Ada has private parts
  - ☞ ...

Problems that are solved by using inheritance in other languages may have a better solution in Ada

When your only tool is a hammer...

...every problem will look like a nail!





OOP in Ada vs. other languages

- Modules and classes are othogonal
  - Classes are the only structuring unit in Eiffel, Java, C#. Files in C++.
  - Some visibility control with packages (Java) and namespaces (Eiffel, C++)
- Difference between specific and class-wide types
  - No equivalent in other languages.
- OOP is not related to pointers
  - All objects are references in Eiffel (create), Java, C# (new).
  - Dispatching on addresses only in C++.
- Several controlling operands, controlling result
  - Not available in other languages

Conclusion

- "Building blocks" approach

  OOP paradigms are built from basic blocks that serve also other purposes

  - ☞ Less specialized than pure OO languages
  - ☞ More flexible
  - ☞ Makes it easier to write "mixed" applications

- An original model

  Not a simple copy of other languages mechanisms

  - ☞ Keeps as much strong typing as possible
  - ☞ Allows subtle relations between objects
  - ☞ Requires more care in design and writing

Ada is pushing the state-of-the-art in OOP