

Welcome to Ada 2005



John Barnes

Manchester, 2006

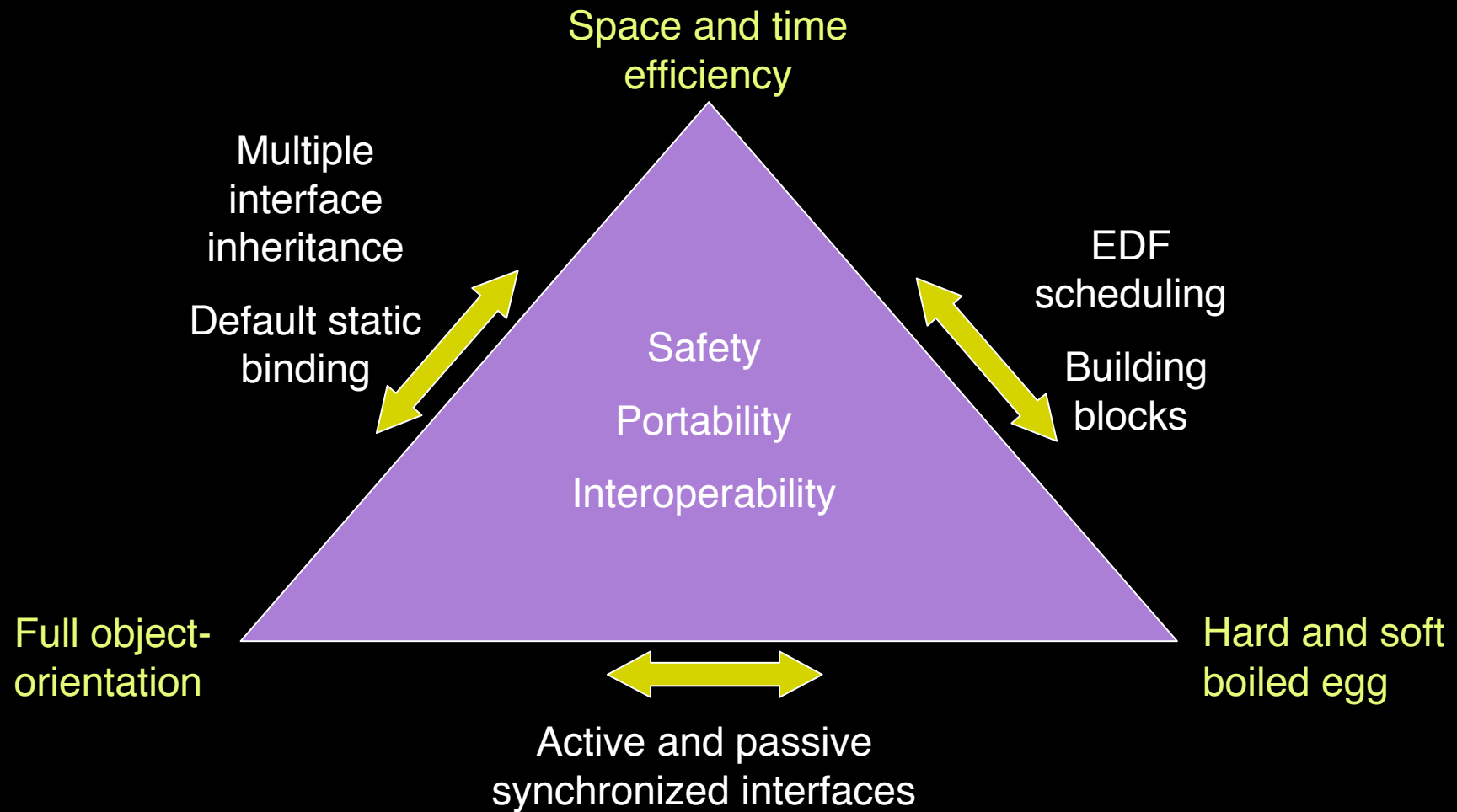
Ada is Alive and Evolving

- Ada 83: “no subsets, no supersets”
 - ▶ Somewhat uptight
- Ada 95: “portable power to the programmer”
 - ▶ Added flexibility, kept reliability
- Ada 2005: “putting it all together” ...
 - ▶ Even more flexible, more reliable too
 - Safety and portability of Java
 - Efficiency and flexibility of C/C++
 - Unrivalled standardized support for real-time / high-integrity systems

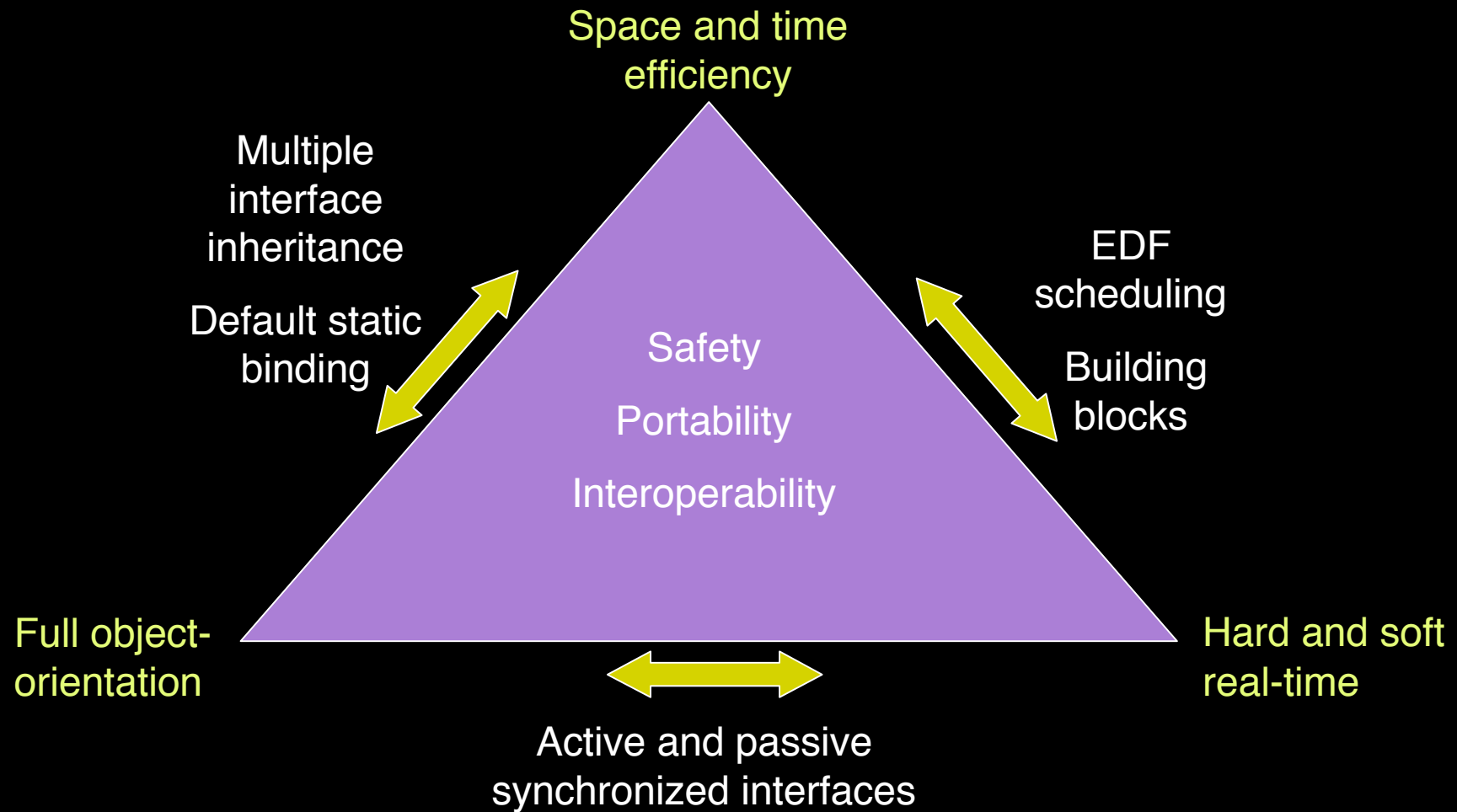
WG9 Guidelines

- Address identified problems that are interfering with Ada's adoption in its major application areas (high reliability, real-time, embedded, large and complex)
- Two specific issues
 - ▶ Ravenscar for real-time
 - ▶ Problem of mutually dependent types across two packages
- Improve Real-Time, High-Integrity
- Improve OO – add Java-like interfaces
- Avoid secondary standards
 - ▶ Incorporate vectors and matrices from 13813

Ada 2005: Putting It All Together



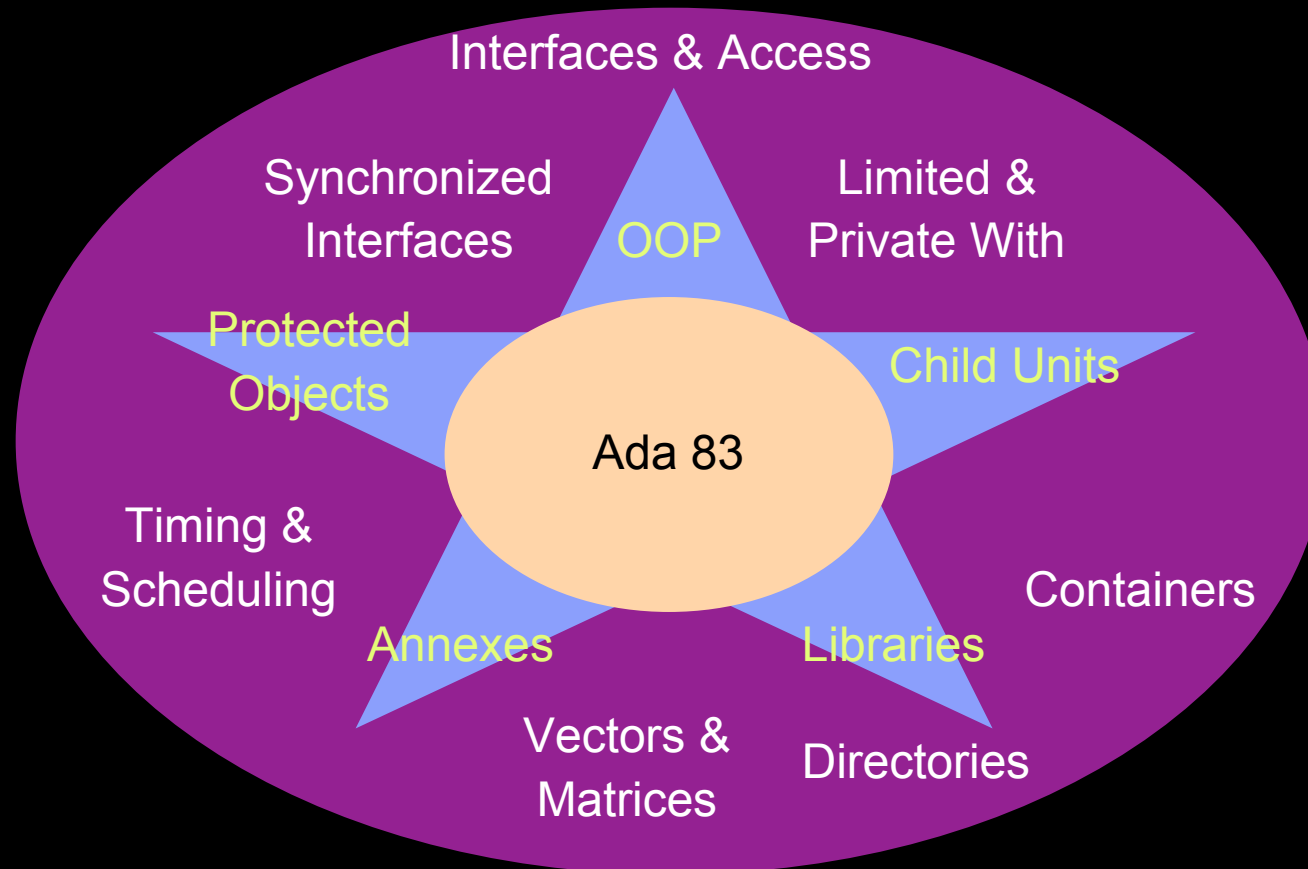
Ada 2005: Putting It All Together



Key Issues

- Safety – Ada is the premier language for safety critical software
 - ▶ Amendments carefully designed so as to not open any safety holes
 - ▶ Several amendments provide even more safety
- Portability
 - ▶ Additions to predefined library
 - ▶ More real-time - EDF, Ravenscar
- Interoperability
 - ▶ Interfaces - both active and passive couple OO with real-time
 - ▶ Object.Operation notation – both synchronized and unsynchronized
 - ▶ Pragma Unchecked_Union for interoperating with C/C++ libraries

Ada 2005: Putting It All Together



Topics as discussed in Rationale

- Object oriented programming
- Access types
- Structure, visibility and limited types
- Tasking and real-time facilities
- Exceptions, numerics, generics etc
- Standard library – containers
- Epilogue – compatibility

- PS There are three new reserved words
 - ▶ **interface** – Used to declare interfaces in OO
 - ▶ **synchronized** – Used for tasks and protected objects in OO
 - ▶ **overriding** – Helps prevent errors in OO

Overview - Object Oriented Programming

- Prefixed notation
 - ▶ Familiar from other languages
- Interfaces
 - ▶ Permit sensible multiple inheritance and synchronized too
- Pascal will tell you about
 - ▶ Nested extensions
 - ▶ Object factory functions
 - ▶ Overriding indicators

Prefixed Notation

- Instead of

```
P.Op (X, ... );           -- operation dominates
```

- We can write

```
X.Op ( ... );           -- object dominates
```

- Provided

- ▶ X is of a tagged type T (declared in package P)
- ▶ Op is a primitive or class-wide operation of T
- ▶ X is the first parameter of Op

- Key advantage, package P not mentioned

Example 1 - Geometry

- **Root type Geometry.Object**

```
package Geometry is
  __type Object is abstract tagged
  __record
    ___X_Coord, Y_Coord: Float;
  ___end record;

  function Area(Obj: Object) return Float is abstract;
  function Moment(Obj: Object) return Float is abstract;
end Geometry;
```

- ▶ Object is abstract, it has components X_Coord, Y_Coord
- ▶ And operations Area, Moment

Add Type Circle

```
package Geometry.Circles is
  __type Circle is new Object with
    __record
      __Radius: Float;
    __end record;

  __function Area(C: Circle) return Float;
  __function Moment(C: Circle) return Float;
end;

package body Geometry.Circles is
  __function Area(C: Circle) return Float is
  __begin
    __return  $\pi$  * C.Radius**2;          -- note  $\pi$ 
  __end Area;

  __function Moment(C: Circle) return Float is
  __begin
    __return 0.5 * C.Area * C.Radius**2;  -- note call of Area
  __end Moment;
end Geometry.Circles;
```

Prefixed Notation

- We can write

```
C.Area
```

- Or

```
Area (C) ;
```

- Prefixed looks same whether a function call or a component

```
C.Area    C.Radius    C.X_Coord
```

Example 2 - People

```
package People is
  _type Person is abstract tagged
  ___record
    ___Birthday: Date;
    ___Height: Cms;
    ___Weight: Kgs;
  ___end record;

  type Man is new Person with
  ___record
    ___Bearded: Boolean;      -- whether he has a beard
  ___end record;

  type Woman is new Person with
  ___record
    ___Births: Integer;      -- how many children she has borne
  ___end record;

  ... -- various operations
end People;
```

Multiple Inheritance

- Remember Flatland? (book by E A Abbott in 1884)
 - ▶ People are two-dimensional geometrical objects
 - Ladies are lean and linear
- We would like to say

```
type Flatlander is new People.Person and Geometry.Object;
```

- Problems
 - ▶ What components and operations are inherited?
 - ▶ Suppose components clash or are duplicated?
 - ▶ Suppose same operation has different implementations?

Ada 2005 Solution

- Only one parent can have components
- Only one parent can have concrete operations (with bodies)
 - ▶ No problem with specifications
- Solution
 - ▶ Multiple ancestors allowed

```
type T is new A and B and C with  
  record ... end record;
```

- ▶ Only the first in list (A) can have components and concrete operations
- ▶ Others (B and C) must be *interfaces* (A could be an interface also)
- ▶ A is the *parent*, B and C are *progenitors*

Interface

- An Interface is

An abstract tagged type

- ▶ It cannot have components
- ▶ It can have abstract operations and null procedures

- Null procedures are written

```
procedure Nothing( ... ) is null;
```

- ▶ They have no body but behave as if the body is null.

Geometry.Object as an Interface

```
package Geometry is
  __type Object is interface;

  _ procedure Move(Obj: in out Object;
                  New_X, New_Y: in Float) is abstract;
  __function X_Coord(Obj: Object) return Float is
  abstract;
  __function Y_Coord(Obj: Object) return Float is
  abstract;

  __function Area(Obj: Object) return Float is abstract;
  __function Moment(Obj: Object) return Float is abstract;
end Geometry;
```

Flatlander

- Can now declare a type Flatlander
 - ▶ Abstract type with parent Person and progenitor Object

```
package Flatland is
  __type Flatlander is abstract new Person and Object with
  __record
    _____X_Coord, Y_Coord : Float;
    _____-- any new components we wish
  __end record;

  __procedure Move(F: in out Flatlander; New_X, New_Y: in Float);
  __function X_Coord(F: Flatlander) return Float;
  __function Y_Coord(F: Flatlander) return Float;
end;
```

- ▶ Abstract since did not want to implement Area and Moment yet
- ▶ Should really make Flatlander private so components are hidden
- ▶ Body is straightforward

And now – A_Square

```
package Flatland.Squares is
  __type Square is new Flatlander with
    __record
      __Side: Float;
    __end record;

  __function Area(S: Square) return Float;
  __function Moment(S: Square) return Float;
end;

package body Flatland.Squares is
  __function Area(S: Square) is
    __begin
      __return S.Side**2;
    __end Area;

  __function Moment(S Square) is
    __begin
      __return S.Area * S.Side**2 / 6.0;
    __end Moment;
end Flatland.Squares.

A_Square: Square;      -- Dr Abbott's alias
```

Summary of Interfaces

- If we have

```
type T is new A and B and C with
    record ... end record;
```

- Then T must *implement* the interfaces B and C
- Although all specific operations of an interface must be abstract, a class wide operation can be concrete, for example

```
function Distance(Obj: Object'Class) return Float is
begin
    return Sqrt(Obj.X_Coord**2 + Obj.Y_Coord**2);
end Distance;
```

- It will dispatch to call functions X_Coord and Y_Coord

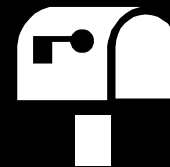
Synchronized Interfaces

- Interface concept generalized to apply to protected and task types
- “Limited” interface can be implemented by:
 - ▶ Limited or non-limited tagged type or interface
 - ▶ Synchronized interface
- “Synchronized” interface can be implemented by:
 - ▶ Task interfaces or types (“active”)
 - ▶ Protected interfaces or types (“passive”)

Example of Synchronized Interface

- Example of protected object interface implementing (extending) a synchronized interface

```
type Buffer is synchronized interface;  
procedure Put(Buf : in out Buffer;  
              Item : in Element) is abstract;  
procedure Get(Buf : in out Buffer;  
              Item : out Element) is abstract;  
  
protected type Mailbox(Capacity : Natural) is new Buffer with  
    entry Put(Item : in Element);  
    entry Get(Item : out Element);  
private  
    Box_State : ...  
end Mailbox;
```



Prefixed notation

- Note how

```
type Buffer is synchronized interface;  
procedure Put(Buf : in out Buffer;  
             Item : in Element) is abstract;
```

- Is matched by

```
protected type Mailbox(Capacity : Natural) is new Buffer with  
  entry Put(Item : in Element);
```

- The first parameter of Put becomes the protected object
 - ▶ Tasks and protected objects always use prefixed notation

Overview - Access Types

- Constant and null control
 - ▶ More uniform rules
- More anonymous access types
 - ▶ Not just as access parameters (and discriminants)
- Anonymous access to subprogram types
 - ▶ For downward closures etc
- Access types and discriminants
 - ▶ Bug fixes

Not Null Everywhere

```
type Acc_Animal is not null access all Animal'Class;
```

```
-- An Acc_Animal must not be null  
-- and so must be initialized  
-- (otherwise Constraint_Error).
```

```
type Pig is new Animal with ... ;  
Empress_of_Blandings : aliased Pig := ... ;
```

```
My_Animal : Acc_Animal := Empress_Of_Blandings'Access;
```

- No check is needed on a dereference to ensure that the value is not null
- So

```
Number_Of_Legs : Integer := My_Animal.Legs;
```

runs faster

Anonymous Access Types

- As well as in
 - ▶ access parameters
 - ▶ access discriminants
- In 2005 we can also use anonymous access types for
 - ▶ stand-alone objects
 - ▶ components of arrays and records
 - ▶ renaming
 - ▶ function return types

As Array and Record Components

```
type Horse is new Animal with ... ;
```

```
Napoleon, Snowball : access Horse := ... ;  
Boxer, Clover : access Pig := ... ;
```

```
Animal_Farm: constant array (Positive range <>) of  
    access Animal'Class :=  
    (Napoleon, Snowball, Boxer, Clover);
```

```
type Noahs_Ark is  
    record
```

```
        Stallion, Mare : access Horse;  
        Boar, Sow : access Pig;  
        Cockerel, Hen : access Chicken;  
        Ram, Ewe : access Sheep;
```

```
        . . .  
    end record;
```

For Function Result

- Can also declare

```
function Mate_Of(A : access Animal'Class)  
    return access Animal'Class;
```

- We can then have

```
if Mate_Of(Noahs_Ark.Ewe) /= Noahs_Ark.Ram then  
    -- Better get Noah to sort things out!  
end if;
```

Access to Subprogram

- Remember Tinman?
- Ada 83 had no requirement for subprograms as parameters of subprograms
- Considered unpredictable since subprogram not known statically
- We were told to use generics
 - ▶ It will be good for you
 - ▶ And implementers enjoy generic sharing

Ada 95 Introduced...

- Simple access to subprogram types

```
type Integrand is access function(X : Float) return Float;
```

```
function Integrate(Fn : Integrand; Lo, Hi : Float) return Float;
```

- To integrate \sqrt{x} between 0 and 1 we have

```
Result := Integrate(Sqrt'Access, 0.0, 1.0);
```

- But suppose we want to do

$$\int_0^1 \int_0^1 xy \, dx \, dy$$

- ♣ A double integral where the thing to be integrated is itself an integral

We can try

```
with Integrate;
procedure Main is

    function G(X : Float) return Float is
        function F(Y : Float) return Float is -- F is nested in G.
            begin
                return X*Y; -- Uses parameter X of G.
            end F;
        begin
            return Integrate(F'Access, 0.0, 1.0); -- Illegal in 95.
        end G;

    Result: Float;
begin
    Result := Integrate(G'Access, 0.0, 1.0); -- Illegal in 95.
    ...
end Main;
```

- Accessibility problem - cannot take 'Access of a subprogram at an inner level to the access type
- We could move G to library level
- But F has to be internal to G because F uses the parameter X of G

Anonymous Access to Subprogram

- Ada 2005 has anonymous access to subprogram types similar to anonymous access to object types
- The function Integrate becomes

```
function Integrate  
  (Fn : access function (X : Float) return Float;  
   Lo, Hi : Float) return Float;
```

- The parameter Fn is of anonymous type
- It now all works
- Note how the profile for the anonymous type is given within the profile for Integrate

Prolific Profiles

- Since profiles can be nested the collector of Ada curiosities will love

```
type A is access function (X: access function (Y: access function
  (Z: access function return Float) return Float) return Float)
  return Float;
```

- Or even, since profiles can also occur in the result type

```
type A is access function return
  access function return
  access function return ...;
```

- No limit to the number of consecutive reserved words in Ada 2005!

Overview - Structure, Visibility and Limited Types

- Pascal will deal with these in a moment
 - ▶ Multi-package type structures
 - ▶ Access to private units in private parts
 - ▶ Partial parameter lists for formal instantiations
 - ▶ Making limited types useful

Overview – Tasking and Real-Time

- Alan will deal with these after coffee
 - ▶ Ravenscar
 - ▶ Task termination
 - ▶ Execution time clocks and timers
 - ▶ Timing events
 - ▶ Dynamic ceiling priorities for protected objects
 - ▶ Additional scheduling/dispatching

Timing Events

- A means of defining code that is executed at a future point in time
- A handler is used

```
package Ada.Real_Time.Timing_Events is  
  type Timing_Event is tagged limited private;  
  type Timing_Event_Handler  
    is access protected procedure(Event : in out Timing_Event);  
  ...  
  procedure Set_Handler(Event : in out Timing_Event;  
                        In_Time: Time_Span;  
                        Handler: Timing_Event_Handler);  
  ...  
end Ada.Real_Time.Timing_Events;
```

Example: Boiling an Egg

```
protected Egg is
  procedure Boil(For_Time: in Time_Span);
private
  procedure Is_Done(Event: in out Timing_Event);
  Egg_Done: Timing_Event;
end Egg;

protected body Egg is

  procedure Boil(For_Time: in Time_Span) is
  begin
    Put_Egg_In_Water;
    Set_Handler (Egg_Done, For_Time, Is_Done'Access);
  end Boil;

  procedure Is_Done ... ;
end Egg;

Egg.Boil(Minutes(4));
-- Now read newspaper whilst waiting for egg.
```

Boiling Several Eggs – crude way

- Aim to share one handler, but need to distinguish eggs
- Need several events, could try

```
type Colour is (Black, Blue, Red, Green, Yellow, Purple);  
Eggs_Done: array (Colour) of aliased Timing_Event;
```

```
Set_Handler(Eggs_Done(Red), For_Time, Is_Done'Access);
```

```
procedure Is_Done(E: in out Timing_Event) is  
begin  
  for C in Colour loop  
    if E'Access = Eggs_Done(C)'Access then  
      -- egg in holder colour C is ready  
      ...  
      return;  
    end if;  
  end loop;  
  -- falls out of loop - unknown event!  
  raise Not_An_Egg ;  
end Is_Done;
```

Boiling Several Eggs – fancy way

- Use type extension and view conversions

```
type Egg_Event is new Timing_Event with  
  record  
    Event_Colour: Colour;  
  end record;
```

```
Eggs_Done: array (Colour) of Egg_Event;  
... Eggs_Done(Red).Event_Colour := Red; ...  
Set_Handler(Eggs_Done(Red), For_Time, Is_Done'Access);
```

```
procedure Is_Done(E: in out Timing_Event) is  
  C: Colour;  
begin  
  if Timing_Event'Class(E) in Egg_Event then  
    C := Egg_Event(Timing_Event'Class(E)).Event_Colour;  
    -- egg in holder colour C is ready  
    ...  
  else  
    -- unknown event - not an egg event!  
    raise Not_An_Egg;  
  end if;  
end Is_Done;
```

Overview - Exceptions, Numerics, Generics, ...

- Exceptions
 - ▶ Minor only – raise with message and null occurrence
- Numerics
 - ▶ Problem fixes for modular types and fixed point
- Pragmas & restrictions
 - ▶ Several more
- Generics
 - ▶ Package parameters, object constructor etc

Pragma Unsuppress

- Overrides Suppress

```
function "*" (Left, Right) return Frac is
  pragma Unsupprsss(Overflow_Check);      -- NOTE
begin
  return Standard."*" (Left, Right);
exception
  when Constraint_Error =>
    if (Left>0.0 and Right>0.0) or (Left<0.0 and Right<0.0) then
      return Frac'Last;
    else
      return Frac'First;
    end if;
end "*";
```

Generic Package Parameters

- Can use `<>` for partial matching

```
generic
  with package DLL is new Doubly_Linked_Lists(<>);
  with package V is new Vectors(Index_Type => <>,
                                Element_Type => DLL.Element_Type,
                                "=" => DLL."=");
function Equals(The_Vector: V.Vector,
                The_List: DLL.List) return Boolean;
```

- if we only want element types to match and not equality then

```
generic
  with package DLL is new Doubly_Linked_Lists(<>);
  with package V is new Vectors(Element_Type => DLL.Element_Type,
                                others => <>);
function Convert(The_Vector: V.Vector) return DLL.List;
```

- ▶ Usual rules for **others**

Overview - Library Stuff

- Time zones and leap seconds
- Operational environment – Directories and Environment variables
- More string subprograms
- Wider and wider characters
- Vectors and matrices (13813++)
- Containers

More Calendar

- Three children of calendar -Time_Zones, Arithmetic, Formatting
 - ▶ mostly about time zones and leap seconds

- Also

```
type Day_Name is (Monday, Tuesday, Wednesday,  
                  Thursday, Friday, Saturday, Sunday);  
function Day_Of_Week(Date: Time) return Day_Name;
```

- And, Year_Number is extended

```
subtype Year_Number is Integer range 1901 .. 2399;
```

- Another 300 years. Long live Ada!!

Directories

- package `Ada.Directories` provides
 - ▶ Directory and file operations
 - ▶ File and directory name operations
 - ▶ File and directory queries
 - ▶ Directory searching
 - ▶ Operations on directory entries
- Enables one to mess about with file names, extensions and so on
- They tell me it is jolly good for Unix and Windows

Environment Variables

- package Ada.Environment_Variables
- Enables one to peek and poke at OS variables
 - ▶ Suppose there is a variable called Ada whose value gives language version

```
if not Exists("Ada") then
    raise Horror;           -- a tragedy!
end if;

Put("Current value of Ada is "); Put_Line(Value("Ada"));

if Value("Ada") /= "2005" then
    Put_Line("Revitalizing Ada now");
    Set("Ada", "2005");
end if;
```

More String Subprograms

- Problems with 95
 - ▶ Conversions between Bounded_String and String and between Unbounded_String and String are required rather a lot
 - ▶ Searching part of a bounded or unbounded string is a pain
 - Find the instances of “bar” in “barbara barnes”
 - ▶ Get_Line is a pain
- To copy a file and add “--” to each line we can now write

```
while not End_Of_File loop
  Put_Line("--" & Get_Line);      -- new function Get_Line
end loop;
```

More Identifier Freedom

- Ada 83 – identifiers in 7-bit ASCII boy, devil, goat
- Ada 95 – identifiers in 8-bit Latin-1 garçon, dæmon, chèvre
- Ada 2005 – identifiers in 16-bit BMP++ _____, _____, _____

```
_____ : access Pig renames Napoleon;  
_____ : Horse;
```

- And in Ada.Numerics

```
 $\pi$  : constant := Pi;
```

- Also Wide_Wide_Character and Wide_Wide_String

Vectors and Matrices

- Incorporates missing stuff from ISO/IEC 13813
- Generic packages
 - ▶ `Ada.Numerics.Generic_Real_Arrays`
 - ▶ `Ada.Numerics.Generic_Complex_Arrays`
- Various arithmetic operations acting on vectors and matrices
 - ▶ Plus: linear equations, inverse, determinant, eigenvalues and vectors
- Goals
 - ▶ To provide a foundation for bindings to libraries such as the BLAS
 - ▶ To make simple, frequently used, linear algebra operations immediately available without fuss

Containers

A package `Ada.Containers` plus lots of children

- ▶ `Vectors`, `Doubly_Linked_Lists`
 - ▶ `Hashed_Maps`, `Ordered_Maps`
 - ▶ `Hashed_Sets`, `Ordered_Sets`
 - also indefinite versions of the above
 - ▶ `Ada.Containers.Generic_Array_Sort`
 - and constrained version
-
- Provide the most commonly required data structure routines
 - Use uniform approach where possible so that conversion is feasible
 - Make them reliable
 - ▶ thou shalt not corrupt thy container

Vectors & Lists

- Uniform approach, many routines common, thus
- Elements can be referred to
 - ▶ By cursor
- Insert, Append, Prepend, Delete, etc.
- Various searching, sorting and iterating procedures, e.g.,

```
procedure Iterate  
  (Container : in Vector/List;  
   Process : not null  
    access procedure (Position : in Cursor));
```

- ▶ Note anonymous access to subprogram parameter

Maps & Sets

- Uniform approach, (hashed or ordered), many routines common
- Elements can be referred to
 - ▶ By cursor
- Ordered map is typically stored as a balanced tree
 - ▶ Reliable $O(\log N)$ performance likely
 - ▶ Slow but sure (not really slow)
- Hashed map typically uses chains of buckets
 - ▶ Unreliable but typically $O(1)$ performance if not too loaded
 - ▶ Fast and fragile (not really fragile)

Conclusions - Ada 2005

- Meets goals set by WG9
 - ▶ More flexible
 - ▶ More reliable
- Is highly compatible with Ada 95
- Continues to meet the needs of demanding users



For more information

Read the Rationale

Listen to Pascal and Alan

Read Programming in Ada 2005

The End



Ada goes galactic

programming in

Ada 2005

JOHN BARNES