







The Ada Programming Language

Jérôme Hugues

ISAE/DMIA – jerome.hugues@isae.fr

These notes are released under the Creative Commons v2.0,  license, which means that:

-  : You are free to share this work,
-  : You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work),
-  : You may not alter, transform, or build upon this work.

Objective

The objective of this course is to provide an overview of the Ada programming language.

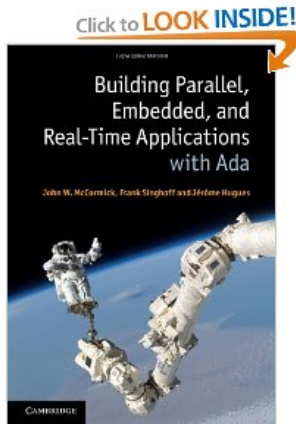
It covers the following topics:

- sequential programming;
- idioms for embedded systems programming.
- concurrency and the Ada Ravenscar profile;

Prerequisites

This lecture notes assume some basic knowledge on sequential programming (C, Java), concurrency and embedded systems.

For an in-depth introduction to Ada and its concurrency constructs, see McCornick, Singhoff, and Hugues 2010.



- 1 Introduction**
 - A (short) history of Ada
 - The Ada programming language
- 2 Ada imperative constructs
- 3 Ada object oriented constructs
- 4 Ada for embedded systems
- 5 Concurrency in Ada
- 6 Conclusion

The roots of Ada

A typical software has several *millions* SLOCs

- SLOC: Significant Lines Of Code
- Hundreds of concepts: data types, functions, bindings between entities
- usually several programming languages interconnected

Several issues: initial creation of the software, then evolution, maintenance and portability.

The language has a key role to address these issues.

Rationale for a (good) language

A programming language should be unambiguous: typing, semantics, human errors, ...

The power of expression should not impede simplicity: integrated support for concurrency, distribution, formal proof, ...

As part of the engineering cycle, it should also supports modularity, abstraction, portability, ease code review and be standardized.

The Ada language is one solution!

The history of Ada

Ada has been built to solve a software engineering issue: build “big” software, with a long life cycle for the military domain.

It has been started as a DoD effort in 1978. At that time, the US Army was using more than 400 languages.

Ada has been built from a set of strong requirements, refined as several proposals. The winner being a team at CII-Honeywell Bull, lead by Jean Ichbiah.

Ada became an ISO standard in 1983, then updated in 1995, 2005 and 2012.

Ada Lovelace

Ada was named after Ada Lovelace , who is sometimes credited as being the first computer programmer.

She worked on Charles Babbage's early mechanical general-purpose computer, the analytical engine.

Her notes on the engine include what is recognised as the first algorithm intended to be processed by a machine.



Ada vs. other languages

The number of programming languages is still quite high.
Python, Ruby, Haskell, Erlang, PHP, ...

- Not standardized, highly changing,
- More a set of APIs than a language

C++, ISO standard (JTC1/SC22/WG21)

- Not as rich as Ada, concurrency added in 2011!
- Weakly typed, pointers, ...

Java, proprietary “standard”

- Rich API for information system
- Trying to fill the gap for real-time systems,
- lost the embedded systems areas to C#

Programming in the large

Software engineering is not just about coding:

- test, validation, review, verification, compilation, debugging, configuration, ...
- Writing code is less than 20% of the budget

Compilation model is one weakness (as in C/C++): OS-specific headers, no standardized way to represent source code, use of Makefiles, ...

Deployment model (as in Java) is also an issue: configuring the JVM, managing versions, classpath, use of ant, maven, ...

The Ada answer

Ada propose a unified model for compilation and deployment, with a standardized library. It supports in its standard:

- Concurrency and realtime;
- Embedded systems;
- Fixed-point computations;
- Interfacing to other languages: C, COBOL, Fortan;
compiler-specific extensions for C++ and Java

Ada use in the industry

Ada is vastly used in critical systems:

- Ariane launchers (GNC);
- fly-by-wire computers for Airbus or Boeing planes;
- TGV trains and railway signalling systems;
- Air Traffic control (iFACTS and Eurocontrol)

The requirements put by certification authorities is usually a key driver for the adoption of Ada.

Ada supports

- imperative model of computation;
- object-oriented constructs (1995) and interfaces a-la Java (2005)
- contract-based programming (2012)
- concurrency (1983)

- 1 Introduction
- 2 Ada imperative constructs**
 - Hello World (revisited)
 - Packages
 - Functions
 - Types
 - Flow control
- 3 Ada object oriented constructs
- 4 Ada for embedded systems
- 5 Concurrency in Ada

Hello World! (Ada version)

Hello World is the canonical C example,

```
with Ada.Text_IO;  
  
procedure Hello is  
begin  
    Ada.Text_IO.Put_Line("Hello ,_world!");  
end Hello;
```


Hello World! (Ada version)

Hello World is the canonical C example,

- *with* (packages), main entrypoint

```
with Ada.Text_IO;
```

```
procedure Hello is
```

```
begin
```

```
    Ada.Text_IO.Put_Line("Hello ,_world!");
```

```
end Hello;
```

Hello World! (Ada version)

Hello World is the canonical C example,

- *with* (packages), main entrypoint
- calling `Put_Line` from `Ada.Text_IO`

```
with Ada.Text_IO;
```

```
procedure Hello is  
begin
```

```
    Ada.Text_IO.Put_Line("Hello ,_world!");  
end Hello;
```

Compiling Ada programs

An Ada program is made of *packages* and one particular main entry point. A construct can be used iff it is visible, either in the local scope, or with'ed.

This eases compilation: simply walk through the transitive closure of dependencies expressed by “withes”.

Using the gnat compiler, this means calling:

```
$ gnatmake hello.adb
```

gnatmake is a wrapper tool that handles all actions required to build the “hello” binary.

Ada packages

An Ada package defines a namespace, use to group a set of consistent concerns.

All Ada packages used must be with'ed. The keyword `use` can be used to avoid full qualification of entities.

```
with Ada.Text_IO; use Ada.Text_IO;
```

```
procedure Hello is  
begin  
    Put_Line("Hello ,_world!");  
end Hello;
```

Ada packages – specification

A package defines the *specification* of types and functions in a public part, and a private one. The public part is the only one that can be manipulated by the user.

It provides the relevant interfaces to the user, but also the compiler (size of types, ...).

```
package Ada.Text_IO is
    type File_Type is limited private;
    — more stuff

    procedure Put_Line (Item : String);
    — more stuff
private
    — Full declaration of File_Type ...
end Ada.Text_IO;
```

Ada packages – body

The body of a package completes the specification: additional private types, functions, etc.

```
package body Ada.Text_IO is  
    type Hidden_Type is record ..  
    end record;  
  
    procedure Put_Line (Item : String) is  
    begin  
        — more stuff  
    end Put_Line;  
end Ada.Text_IO;
```

About functions

Ada makes a distinction between functions (that returns something) and procedure (no return value).

```
function My_Function (arg1:Type1 ; Arg2 : Type2 ...)
    return Type_Result is
    — declarations
begin
    — instructions
    return Something;
end Nom_fonction;
```

```
procedure My_Procedure (arg1: model type1; arg2:type2...);
```

It is a good practice to define functions before usage, although not required.

Parameters passing modes

Formal parameters of a procedure can be either

- in: set by the caller;
- out: set by the callee;
- in out: set by the caller and the callee;

This avoids error-prone manipulation of pointers like in C.

Note: Ada 2012 also extends this to functions.

Typing system

Ada is *strongly* typed: one can combine only types that are equal; while in C, one can combine homogeneous types.

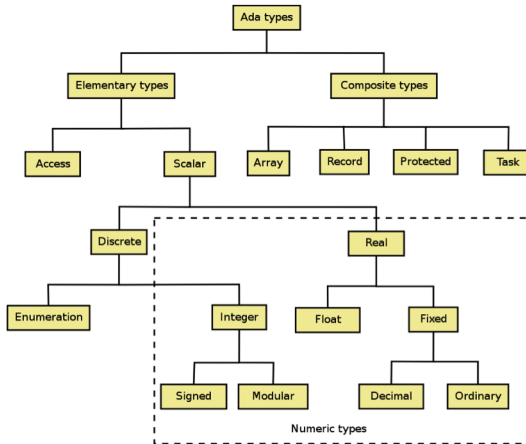
Ada has predefined types: boolean, integer, float, ...

User can define enumerations, records, discriminated types (unions), access types (pointers)

Typical operators are =, <, >, +, - ..

“:=” is the affectation operator, “/=” for inequality.

Type hierarchy



Type vs subtype

Ada distinguish new type from subtype.

A new type has each own set of primitives (e.g. operators).

A subtype refines an existing type, e.g. Positive and Natural are subtypes of Integer.

```
package Types is
  type Integer_1 is new Integer;
  type Integer_2 is range 1 .. 10;
  A : Integer_1 := 8;
  B : Integer_2 := A; — illegal!
  type Integer_A is range 1 .. 10;
  subtype Integer_B is Integer_1 range 7 .. 10;
  A : Integer_1 := 8;
  B : Integer_2 := A; — OK
end Types;
```

Subtype and inheritance

Subtypes inherit primitives from their parents.

```
procedure Derived_Types is
  package Pak is
    type Integer_1 is range 1 .. 10;
    procedure P (I : in Integer_1);
    — primitive operation, assumes 1 .. 10
    type Integer_2 is new Integer_1 range 8 .. 10;
    — must not break P's assumption
  end Pak;

  use Pak;
  A : Integer_1 := 4;
  B : Integer_2 := 9;
begin
  P (B); — OK, call the inherited operation
end Derived_Types;
```

Types and qualified expressions

The compiler may infer types statically. Otherwise, the expression must be qualified:

```
type Enum is (A, B, C);  
E : Enum := A; — OK
```

```
procedure Bad is  
  type Enum_1 is (A, B, C);  
  procedure P (E : in Enum_1) is ... — omitted  
  type Enum_2 is (A, X, Y, Z);  
  procedure P (E : in Enum_2) is ... — omitted
```

```
begin  
  P (A); — illegal: ambiguous  
  P (Enum_1'(A)); — OK  
end Bad;
```

Types manipulation

```
I: Integer := Integer (10);  
   — Unnecessary explicit type conversion  
J: Integer := 10;  
   — Implicit conversion from universal integer  
K: Integer := Integer '(10);  
   — Use the value 10 of type Integer: qualified expression  
   — (qualification not necessary here).
```

Attributes of discrete types

Discrete types are integer, enumerations, ...

— *General form:* `<type>'Attribute`

`Integer 'First;` — *Lower bound*

`Integer 'Last;` — *Upper bound*

`T'Pos(X);` — *X position in enumeration*

`T'Val("X");` — *Return the value from string "X"*

`T'Image (X);` — *Image of X*

Array type

Ada is a compiled language, meant for critical systems. Arrays are first class citizen, can be fully bounded at compile time.

— *Bounded array*

```
subtype Index_Sub_Type is Index_Type range First .. Last;  
type A is array (Index_Sub_Type) of Element_Type;
```

— *Unbounded type*

```
type String is array (Positive range <>) of Character;  
Input : String := Read_From_File; —  
Length computed at run-time
```

— *Multi-dimension array*

```
type Character_Display is  
    array (Positive range <>, Positive range <>) of Character;
```


Manipulating arrays

Ada has run-time protection mechanisms to defend against array overrun. In case of error, an exception is raised at run-time.

— Array operators

```
Vector_A (1 .. 3) := Vector_B (3 .. 5);
```

— Concatenation

```
Name := First_Name & '_' & Last_Name;
```

— Attributes

```
Hello_World : constant String := "Hello_World!";
```

— Hello_World'Range -> 1 .. 12

```
Length : Integer := Hello_World'Length; — length
```

```
First : Integer := Hello_World'First; — index of first element
```

```
Last : Integer := Hello_World'Last; — index of last element
```

Records

```
package Records is
```

```
  type Basic_Record is record
```

```
    A : Integer;
```

```
    B : Boolean;
```

```
  end record;
```

```
  A_Record : Basic_Record := Basic_Record'(A => 42, B => False);
```

```
  Another_Record : Basic_Record := (B => False, A => 42);
```

```
  Yet_Another : Basic_Record := (42, False);
```

```
end Records;
```

Discriminants and unions

A type can have one element as a parameter:

```
package Discriminants is
  type Discriminated_Record (Size : Natural) is record
    A : String (1 .. Size);
  end record;
  type Traffic_Light is (Red, Yellow, Green);
  type Variant_Record (Option : Traffic_Light) is record
    -- common components
    case Option is
      when Red =>
        -- components for red
      when Yellow =>
        -- components for yellow
      when Green =>
        -- components for green
    end case;
  end record;
end Discriminants;
```

Fixed-point types

Fixed-point are part of the language, allowing control over truncation in computations, e.g. controller.

```
package Fixed_Point is  
  
  — decimal fixed point  
  type Duration is delta 10.0**(-9) digits 9;  
  
  — ordinary fixed point  
  type Duration is delta 10.0**(-9) range -1.0 .. 1.0;  
  for Duration'Small use 10.0**(-9);  
  
end Fixed_Point;
```

Access types

Access types (pointers) are true types in Ada.

```
package Access_Types is
  type Day_Of_Month is range 1 .. 31;

  type Day_Of_Month_Access is access Day_Of_Month;
  — To refer to dynamically allocated variables

  type Day_Of_Month_Access_2 is access all Day_Of_Month;
  — To refer also to stack-allocated variable

  type Callback_Procedure is access procedure (Id
: Integer; Text: String);
  procedure Process_Event (Id : Integer; Text: String);
  My_Callback : Callback_Procedure := Process_Event 'Access;
end Access_Types;
```

Access and aliased

Contrary to C, Ada lets you control types that can be turned into pointer (manipulating their address)

```
package Alias is
  type General_Pointer is access all      Integer;
  type Constant_Pointer is access constant Integer;
  I1: aliased constant Integer := 10;
  I2: aliased Integer;
  P1: General_Pointer := I1'Access;  — illegal
  P2: Constant_Pointer := I1'Access; — OK, read only
  P3: General_Pointer := I2'Access;  — OK, read and write
  P4: Constant_Pointer := I2'Access; — OK, read only
  P5: constant General_Pointer := I2'Access;
— read and write only to I2
end Alias;
```

Memory management

```
with Ada.Unchecked_Deallocation;  
  
procedure Deallocation_Sample is  
  type Vector is array (Integer range <>) of Float;  
  type Vector_Ref is access Vector;  
  procedure Free_Vector is new Ada.Unchecked_Deallocation  
    (Object => Vector, Name => Vector_Ref);  
  VA, VB: Vector_Ref;    V    : Vector;  
begin  
  VA := new Vector (1 .. 10);  
  VB := VA; — points to the same location as VA  
  VA.all := (others => 0.0);  
  — ... Do whatever you need to do with the vector  
  Free_Vector (VA); — VA is now null  
  V := VB.all; — VB is not null!  
               — access to a dangling pointer is erroneous  
end Deallocation_Sample;
```

Limited types

A limited type cannot be copied. Useful to ensure there is no duplicate reference of the same object, in inconsistent states. E.g. a memory pool, a device descriptor, etc.

```
package Limited_Types is

  type Entity is limited record
    Counter : Integer := 0;
    — Reference counter.

    Data : Pool_Access := null;
  end record;

end Limited_Types;
```


Exception

Ada relies on exception to handle errors. It defines several exceptions:

- `Constraint_Error` in case of invalid access to data;
- `Storage_Error` for invalid memory access;
- `Program_Error`, `Tasking_Error`;

```
function Sqrt (X : Float) return Float is  
begin  
  if X < 0.0 then  
    raise Constraint_Error;  
  end if;  
  — compute square root of X
```

User-defined exceptions

```
with Ada.Exceptions; use Ada.Exceptions;
procedure Insert
  (New_Name    : Name; New_Number : Number)
is
  Name_Duplicated : exception;
  Directory_Full  : exception;
begin
  if New_Name = Old_Entry.A_Name then
    raise Name_Duplicated;
  end if;
  New_Entry := new Dir_Node'(New_Name, New_Number);
exception
  when Storage_Error => raise Directory_Full;
  when E : others =>
    Put_Line (Exception_Information(E));
end Insert;
```

if-then-else

```
if Temperature >= 40.0 then
    Put_Line ("It 's_extremely_hot");
elsif Temperature >= 30.0 then
    Put_Line ("It 's_hot");
else
    Put_Line ("It 's_freezing");
end if;
```

Alternatives

Alternatives relies on discrete types, all values must be covered.

```
case X is
  when 1 =>
    Walk_The_Dog;
  when 5 =>
    Launch_Nuke;
  when 8 | 10 =>    — 8 or 10
    Sell_All_Stock;
  when others =>    — default value
    Self_Destruct;
end case;
```

Loops

```
Endless_Loop :    loop           — naming the loop is optionnal  
    Do_Something;  
end loop Endless_Loop;
```

```
While_Loop :  
    while X <= 5 loop  
        X := Compute_Something;  
    end loop While_Loop;
```

```
Until_Loop :  
    loop  
        X := Compute_Something;  
        exit Until_Loop when X > 5;  
    end loop Until_Loop;
```

for Loops

The loop variable is implicitly declared, and cannot be modified

```
For_Loop :  
  for J in Integer range 1 .. 10 loop  
    Do_Something (J)  
  end loop For_Loop;
```

```
Array_Loop :  
  for J in X'Range loop  
    X (J) := Get_Next_Element;  
  end loop Array_Loop;
```

Ada generic

Ada generic allows for limited template programming:

```
generic
  type Element_T is private; — Generic formal type parameter
procedure Swap (X, Y : in out Element_T);

procedure Swap (X, Y : in out Element_T) is
  Temporary : constant Element_T := X;
begin
  X := Y;
  Y := Temporary;
end Swap;

procedure Swap_Integers is new Swap (Integer);
— Instanciation
```

Ada generic package

Generic can also be at the level of a package, e.g.

```
generic
  Max : Positive;
  type Element_T is private;
package Generic_Stack is
  procedure Push (E: Element_T);
  function Pop return Element_T;
end Generic_Stack;

package body Generic_Stack is
  Stack: array (1 .. Max) of Element_T;
  Top  : Integer range 0 .. Max := 0;
  — ...
end Generic_Stack;

package Float_100_Stack is new Generic_Stack (100, Float);
```


Formal parameters of a generic

formal parameters define the “contract” associated to a generic

- type T (<>) is [abstract] [tagged] [limited] private
- type T (<>) is [abstract] new Parent [with private]
- type T is (<>);
- type T is range <>;

- 1 Introduction
- 2 Ada imperative constructs
- 3 Ada object oriented constructs**
 - Motivations
 - Ada object model
 - Dynamic dispatching
- 4 Ada for embedded systems
- 5 Concurrency in Ada
- 6 Conclusion

From imperative to object oriented programming

Several problems to address when implementing large software:

- What are the relevant data types? (problem space)
- What are the required fonctions? (solution space)

Imperative programming is limited to statements, the data model is weak, one needs:

- to extend types
- add new functions that apply to multiple types

Let us consider an alarm system¹

¹Example borrowed from https://libre.adacore.com/Software_Matters/

An alarm system in Ada

Two functions:

- Log alarms
- Handle alarms

```
with Ada.Calendar;  
package Alerts is  
    type Alert is private;  
  
    procedure Handle (A : in out Alert);  
    procedure Log      (A : Alert);  
  
private  
    type Alert is record  
        Time_Of_Arrival : Ada.Calendar.Time;  
        Cause           : String (1 .. 200);  
    end record;  
end Alerts;
```

Different levels of alarm

Let us use a discriminated type for alerts

```
with Calendar; use Calendar;
with Persons; use Persons;
package Alerts is
  type Priority is (Low, Medium, High);
  type Alert (P : Priority) is private;
  procedure Handle (A : in out Alert);
  procedure Log (A : Alert);
  procedure Set_Alarm (A : in out Alert; Wait : Duration);
private
  type Alert (P : Priority) is record
    Time_Of_Arrival : Time;
    Cause : String (1 .. 100);
    case P is
      when Low => null;
      when Medium => Technician : Person;
      when High =>
        Engineer : Person;
        Ring_Alarm_At : Time;
    end case;
  end record;
end Alerts;
```

Different levels of alarm (cont'd)

```
procedure Handle (A : in out Alert) is
begin
  A.Time_Of_Arrival := Calendar.Clock;
  A.Cause            := Get_Cause (A);
  Log (A);
  case A.P is
    when Low      => null;
    when Medium => A.Technician := Assign_Technician;
    when High    =>
      A.Engineer := Assign_Engineer;
      Set_Alarm (A, Wait => 1800);
  end case;
end Handle;

procedure Set_Alarm
  (A : in out Alert; Wait : Duration) is
begin
  A.Ring_Alarm_At := A.Time_Of_Arrival + Wait;
  — raise Constraint_Error if A.Priority /= High
end Set_Alarm;
```

Extensions

Modifying a type forces the modification of many code snippets, see:

```
type Alert (P : Priority) is record
  case P is
    — ...
    when Emergency => — new field
  end case;
end record;

procedure Some_Routine (A : in out Alert) is
begin
  case A.P is
    when Emergency => — new case
  end case;
end Some_Routine;
```

Towards Object-Oriented paradigm

Object-orientation has been introduced to ease software evolution:
consider a software as set of interacting objects ...
but also to factor out code for similar types

Ada'95 is the first normalized object oriented language

- Compared to C++ or Java, it makes the link to object explicit through particular types, not new keyword
- This reduces moving from imperative to OO
- Allow one to mix imperative and OO styles in one program

Tagged types

Introducing “tagged” types

```
package Alerts is
```

```
  type Alert is tagged record — tagged type are classes  
    Time_Of_Arrival : Calendar.Time;  
    Cause           : String (1 .. 200);  
end record;
```

```
  procedure Handle (A : in out Alert);  
— Primitive operations (methods)  
  procedure Log      (A : Alert);  
end Alerts;
```

Type derivation

A derived type inherits primitives and components of its father

```
package Alerts is
```

```
  type Alert is tagged record
```

```
    Time_Of_Arrival : Calendar.Time;
```

```
    Cause           : String (1 .. 200);
```

```
  end record;
```

```
  procedure Handle (A : in out Alert);
```

```
  procedure Log      (A : Alert);
```

```
  type Low_Alert is new Alert with null record;
```

```
  — inherits fields Time_OF_Arrival, Cause
```

```
  — inherits Handle, Log primitives
```

```
  — equivalent to Handle (A : in out Low_Alert); ..
```

```
end Alerts;
```

Type derivation (cont'd)

A derived type can add new components, new primitives

```
package Alerts is
  type Alert is tagged record
    — ...
  end record;
  procedure Handle (A : in out Alert);

  type Medium_Alert is new Alert with record
    Technician : Person; — added
  end record;
  procedure Handle (A : in out Medium_Alert);
  — redefined

  type High_Alert is new Alert with record
    Engineer : Person;
    Ring_Alarm_At : Calendar.Time;
  end record;
  procedure Set_Alarm — specific
    (A : in out High_Alert; Wait : Duration);
  procedure Handle (A : in out High_Alert);
end Alerts;
```

Using the alarm system

The compiler may infer the correct usage at compile time

```
with Alerts; use Alerts;
procedure Client is
  A   : Alert;
  A_L : Low_Alert;
  A_M : Medium_Alert;
  A_H : High_Alert;
begin
  A.Time_Of_Arrival :=      — OK
  A_L.Time_Of_Arrival :=
  A_M.Time_Of_Arrival :=
  A_H.Time_Of_Arrival :=
  A.Engineer :=              — undefined
  A_L.Engineer :=
  A_M.Engineer :=
  A_H.Engineer := ;         — OK
end Client;
```

Using the alarm system (cont'd)

The compiler may infer automatically the correct function to call, no dispatching at run-time.

```
with Alerts; use Alerts;
procedure Client is
  A      : Alert;
  A_L    : Low_Alert;
  A_M    : Medium_Alert;
  A_H    : High_Alert;
begin
  Handle (A);
  Handle (A_L);
  Handle (A_M);
  A_H.Handle; — Ada 2005
  Set_Alarm (A_M, 1800); — undefined
  A_H.Set_Alarm (1800);
end Client;
```

Calls and inheritance

One can use delegation to first execute the parent's code, then child-specific code.

```
procedure Handle (A : in out Alert) is  
begin  
  A.Time_Of_Arrival := Calendar.Clock;  
  A.Cause           := Get_Cause (A);  
  Log (A);  
end Handle;
```

```
procedure Handle (A : in out Medium_Alert) is  
begin  
  Handle (Alert (A)); — First handle as plain Alert  
  A.Technician := Assign_Technician;  
end Handle;
```

Note: one can also preserve abstraction barriers, Medium_Alert may be defined in another package.

OO and unconstrained type

How to define a variable that stores any variant of `Alert`?

Answer: The “`Class`” attribute

- `T'Class` is any descendant of `T`;
- Allows dynamic dispatching calls

```
with Alerts; use Alerts;  
function Get_Alert return T'Class;  
  
procedure Process_Alerts is  
begin  
  loop — infinite loop  
    declare  
      A : Alert'Class := Get_Alert;  
    begin  
      Handle (A); — dispatching call  
    end;  
  end loop;  
end Process_Alerts;
```

Ada and dynamic dispatching

Either the subprogram call can be computed statically, or dynamic call

```
AL : Low_Alert;  
Handle (AL); — non dispatching call  
A   : Alert'Class := Get_Alert;  
Handle (A); — dispatching call  
A   : High_Alert'Class :=  
Handle (A); — dispatching call  
  
procedure Handle (A : in out Alert) is  
begin  
    A.Time_Of_Arrival := Calendar.Clock;  
    A.Cause           := Get_Cause (A);  
  
    Log (Alert'Class (A)); — Force redispaching  
end Handle;
```


overriding keyword

The overriding keyword highlights a method overriding one from his parent.

```
package Alerts is  
  type Alert is tagged record  
    — ...  
  end record;  
  procedure Handle (A : in out Alert);  
  
  type Medium_Alert is new Alert with record  
    Technician : Person; — added  
  end record;  
  overriding procedure Handle (A : in out Medium_Alert);  
  — redefined  
  
end Alerts;
```

This is defined as a safeguard mechanism against typo and unwanted effect.

Note: this is equivalent to @Override in Java

Dynamic dispatching: a comparison

- Ada
 - All methods may be dispatching
 - User decides for each call
- C++
 - Dispatching call only for virtual methods
 - Always dispatching
- Java
 - Always dispatching

Ada dispatching model allows for a finer control on dispatching, thus allow for safer programming even for safety critical systems.

Abstract types

Abstract types are a good candidate for the root of a type hierarchy, can have methods, but no variable of this type.

```
package Alerts is
  type Alert is abstract tagged private;

  procedure Handle (A : in out Alert);
  procedure Log      (A : Alert) is abstract;

  A : Alert; — wrong
private
  type Alert is tagged record
    Time_Of_Arrival : Calendar.Time;
    Cause            : String (1 .. 200);
  end record;
end Alerts;
```

How to do a constructor in Ada?

No default mechanism in the language, as opposed to Java or C++
Supported by an external type

```
package Ada.Finalization is  
  
  type Controlled is abstract tagged private;  
  procedure Initialize (Object : in out Controlled);  
  procedure Adjust     (Object : in out Controlled);  
  procedure Finalize   (Object : in out Controlled);  
  
  — ...
```

Calls to Controlled type methods are inserted by the compilation chain when creating, copying or finalizing an object.

Controlled types: an example

```
with Ada.Finalization; use Ada.Finalization;
generic
  type Data_Type is private;
package Linked_List_Package is
  type Linked_List is new Controlled with private;
  procedure Add (List      : in out Linked_List; New_Data :
Data_Type);
  procedure Finalize (List : in out Linked_List); — reset
— Initialize and Adjust are left to the default (no-op)
private
  type Node; — Incomplete declaration for self-reference
  type Node_Ptr is access Node;
  type Node is record — Completion of forward declaration
    Data : Data_Type;
    Next : Node_Ptr;
  end record;
  type Linked_List is new Controlled with record
    Count : Natural := 0;
    Head  : Node_Ptr;
  end record;
end Linked_List_Package;
```

- 1 Introduction
- 2 Ada imperative constructs
- 3 Ada object oriented constructs
- 4 Ada for embedded systems**
 - Representation clauses
 - Code efficiency
- 5 Concurrency in Ada
- 6 Conclusion

Representation clauses

One can attach a particular representation to enumerations, e.g. to map to low-level micro-controller registers:

```
type Status is  
    (Off, On, Busy);
```

```
for Status use  
    (Off => 0,  
     On  => 1,  
     Busy => 3);
```

```
for Status'Size use 2;
```

The compiler perform all required conversions to align data.

Mapping to hardware

```
type Data is range 0 .. 127;
for Data' Size use 7;
type Register is record
    Send : Data;
    Recv : Data;
    Stat : Status;
end record;

for Register use record
    Send at 0 range 0 .. 6;
    Recv at 0 range 7 .. 13;
    Stat at 0 range 14 .. 15;
end record;

Device_Register : Register;
for Device_Register use at 8#100#;
```


Accessing memory

Ada defines the following pragma:

- **Volatile**: no cache or register, direct access to memory;
- **Atomic**: atomic access to data.

```
package Shared_Buffer is  
  Buffer_Array : array (0 .. 1023) of Character;  
  pragma Volatile_Components (Buffer_Array);  
  Next_In , Next_Out : Integer := 0;  
  pragma Atomic (Next_In , Next_Out);  
end Shared_Buffer;
```

pragma Suppress

pragma Suppress disables some runtime checks, can be used iff the software has been completely tested, or formally proved using e.g. SPARK.

Suppressing these checks has a positive impact on memory and time resources, iff used carefully!

pragma Restrictions

pragma Restrictions forbids the use of some Ada constructs (e.g. tasking, object-oriented, ...).

This enables the implementation of highly efficient runtimes like the ORK+ kernel Vardanega, Zamorano, and Puente 2005.

```
pragma Restrictions (No_Task_Allocator);  
— No allocators for task types  
pragma Restrictions (No_Task_Hierarchy);  
— All tasks depend directly from env. task ...  
pragma Restrictions (No_Allocator);  
— There are no occurrences of an allocator  
pragma Restrictions (No_Recursion);  
— As part of the execution of a subprogram,  
— the same subprogram is not invoked  
pragma Restrictions (No_Dispatch);
```

Binding to other languages

The annex B of the Ada Reference Manual defines how to bind Ada to other languages through the use of the pragma Import and pragma Export mechanisms:

```
function Gethostbyname (Name : in C.Address)
    return Host_Entry_Address;
pragma Import (C, Gethosbyname , "gethostbyname");
```

Attaching interrupts

Ada uses a protected object to bind interrupt handler to an interrupt ID, defined in the `System` package.

```
protected Message_Driver is  
    entry Get (M : out Message);  
  
private  
    Current : Message := None;  
    procedure Handle;  
    pragma Attach_Handler (Handle, Device_IT_Id);  
    pragma Interrupt_Priority;  
end Message_Driver ;
```

- 1 Introduction
- 2 Ada imperative constructs
- 3 Ada object oriented constructs
- 4 Ada for embedded systems
- 5 Concurrency in Ada**
 - Support for time, concurrency
 - The Ravenscar Profile
- 6 Conclusion

Ada has a rich semantic model for addressing concurrency, in the following we focus on the Ravenscar profile Dobbing, Burns, and Vardanega 2003.

It is a subset of concurrency constructs that are statically analyzable using Response Time Analysis.

It relies on elements from the code Ada language (chapter 9) and Annex D on Real-Time Systems.

Ada idioms for concurrency

Ada defines two entities for concurrency:

- task: an independent unit of computation. Task can either be singleton, or form a type;
- protected object: a set of primitives associated to a lock, and eventually a barrier.

In some implementation, task and protected objects are mapped onto OS primitives, e.g. POSIX threads, mutexes and condition variables.

Per the standard, an Ada runtime must provide at least 30 levels of priority, one being reserved to interrupts. The actual range of priority usually depends on the host environment.

```
subtype Any_Priority is Integer
  range Implementation-Defined;
subtype Priority is Any_Priority
  range Any_Priority 'First .. Impl_Defined;
subtype Interrupt_Priority is Any_Priority
  range Priority 'Last + 1 .. Any_Priority 'Last;
Default_Priority : constant Priority :=
  (Priority 'First + Priority 'Last)/2;
```

Monotonic clock

Real-Time systems require a precise monotonic clock to monitor time.

Let us consider the following clock:

Resolution : 2^{-31} s

$$40 \text{ ms} = 0.04 + \frac{21}{25} \times 2^{-31}$$

$$10 \text{ ms} = 0.01 - \frac{3}{25} \times 2^{-31}$$

$$40 \text{ ms} - 4 * 10 \text{ ms} = 2^{-30} (\neq 0)$$

The accumulation of such errors may lead to serious bugs, like the Patriot missile.

Monotonic clock in Ada

Ada defines a clock primitive in two packages: `Ada.Calendar` (regular clock) and `Ada.Real_Time` (real-time clock).

Each clock has its own definition of time which may differ, as their origin.

They also define set of operators for basic manipulation.

Ada defines two constructs to suspend a task:

- `delay Relative_Time`: the task is awoken after `Relative_Time` elapses. It is a minimum waiting time;
- `delay until Absolute_Time`: the task is awoken at a predetermined instant in the future.

Note: usually delay unit $D \neq \text{delay } D - \text{Clock}$ because of preemption.

About the Ravenscar profile

The Ravenscar profile has been defined to ease transition from cyclic executives to preemptive ones. It is a set of restrictions defining the following model:

- FIFO per priority level and PCP locking policy;
- inter-task communication using PO;
- restrictions to forbid dynamic modification of the task set (number of tasks, priority);
- restrictions to avoid using some constructs like delay, Calendar clock, ...

Ravenscar profile restrictions

```
pragma Task_Dispatching_Policy ( FIFO_Within_Priorities );  
pragma Locking_Policy ( Ceiling_Locking ); pragma Detect_Blocking;  
pragma Restrictions (  
    No_Abort_Statements , No_Dynamic_Attachment ,  
    No_Dynamic_Priorities , No_Implicit_Heap_Allocations ,  
    No_Local_Protected_Objects , No_Local_Timing_Events ,  
    No_Protected_Type_Allocators , No_Relative_Delay ,  
    No_Requeue_Statements , No_Select_Statements ,  
    No_Specific_Termination_Handlers , No_Task_Allocators ,  
    No_Task_Hierarchy , No_Task_Termination ,  
    Simple_Barriers , Max_Entry_Queue_Length => 1 ,  
    Max_Protected_Entries => 1 , Max_Task_Entries => 0 ,  
    No_Dependence => Ada.Asynchronous_Task_Control ,  
    No_Dependence => Ada.Calendar , No_Dependence =>  
    Ada.Execution_Time.Group_Budget , No_Dependence =>  
    Ada.Execution_Time.Timers , No_Dependence =>  
    Ada.Task_Attributes );
```

Defining a task

Per Ravenscar, the specification of a task is limited to its name and two optional pragmas:

- `pragma Priority` defines a task priority, or `System.Default_Priority` if not set;
- `pragma Storage_Size` defines the stack size.

All tasks must be defined in package specifications

```
package Task_Example is  
  task Real_Time_Sensor is  
    pragma Priority (10);  
    pragma Storage_Size (128 * 1_024);  
  end Real_Time_Sensor;  
end Task_Example;
```

Defining a task body

```
with Ada.Real_Time;      use Ada.Real_Time;

package body Task_Example is
  task body Real_Time_Sensor is
    P : constant Time_Span := Milliseconds (40);
    D : Time := Clock + P;
  begin
    loop
      -- Read sensor
      delay until D;
      D := D + P;
    end loop;
  end Real_Time_Sensor;
end Task_Example;
```


Defining a protected object

- `pragma Priority` defines the PCP priority

```
package Protected_Object is

  protected Sampling_Port is
    pragma Priority (System.Priority 'Last);

    procedure Send_Data (Value : Integer);
    entry Wait_Data (Value : out Integer);
  private
    Barrier : Boolean := False;
    Data : Integer;
  end Sampling_Port;

end Protected_Object;
```

Defining a protected object

```
package body Protected_Object is
  protected body Sampling_Port is
    procedure Send_Data (Value : Integer) is
    begin
      Data := Value;
      Barrier := True;
    end Send_Data;

    entry Wait_Data (Value : out Integer) when Barrier is
    begin
      Barrier := False;
      Value := Data;
    end Wait_Data;
  end Sampling_Port;
end Protected_Object;
```

Some notes on protected objects

The Ravenscar sets some limits on PO:

- at most one entry, to ease analysis;
- if PCP is misconfigured, an exception is raised;
- the barrier is evaluated when a task manipulates the PO, it cannot depend on parameters of entry or functions.
- blocking operations are forbidden in a PO;
- at most one task may be blocked on the barrier (exercise!);
- if an exception is raised, the PO state is restored: locks are freed.

- 1 Introduction
- 2 Ada imperative constructs
- 3 Ada object oriented constructs
- 4 Ada for embedded systems
- 5 Concurrency in Ada
- 6 Conclusion**

Some final words

This is just a quick overview of Ada, other topics may include:

- Advanced scheduling disciplines: priority-bands, EDF, Round-Robin;
- Execution time timers; Task identification; asynchronous control;
- pre/post conditions; object-oriented; interfaces; ...
- formal proof on Ada code; coding patterns for High-Integrity systems

Ada is a rich language compared to C or Java, with lots of built-in elements for high-integrity systems.

Some web resources:

- Ada Wikibook : http://upload.wikimedia.org/wikibooks/en/8/8d/Ada_Programming.pdf
- Tutorial Lovelace :
<http://www.adahome.com/Tutorials/Lovelace/lovelace.htm>
- Toolchain : <http://libre.adacore.com>