Ada programming language

An introduction for TTK4145

Kristoffer Nyborg Gregertsen

SINTEF ICT
Department of Applied Cybernetics

2014-02-25

Contents

Contents

What is Ada?

- Ada is a general purpose programming language designed for safety and maintainability
- High-integrity real-time and embedded systems
- Ordered by the US DoD in the late 70s to replace the hundreds of different languages used in military
- A french team won with the language Ada
- Named after Ada Lovelace
  - The worlds first programmer
  - Therefore **not** acronym ADA
- The language became an ISO standard in 1983

Ada 2005 and 2012

- Started with Ada 83 and major revision Ada 95
- Ada 2005 improved Ada 95 with features such as:
  - More Java-like object-orientation with interfaces
  - Extensions to the standard library
  - Tasking, real-time improvements
  - The Ravenscar profile for high-integrity systems
- Ada 2012 is the latest revision with features as:
  - Contract-based programming
  - Task affinities and dispatching domains
  - The Ravenscar profile for multiprocessor systems
- Difference between Ada 2005 and 2012 is not further discussed here...

Why use Ada?

- Ada was designed for use in high-integrity systems and has many safeguards against common programming faults
- Ada has excellent support for development and maintenance of large applications by its notation of packages
- Ada has built-in language support for tasking and a rich set of synchronization primitives
- Ada is used for safety critical projects such as:
  - International Space Station (ISS)
  - Airbus 320, 330, 340, 380
  - Boeing 737, 747-400, 757, 767, 777, 787
  - TGV and European Train Control System (ETCS)

Hello world!

```
with Ada.Text_IO;

procedure Hello_World is
begin
   Ada.Text_IO.Put_Line ("Hello␣World!");
end Hello_World;
```

- Notice that there are no curly brackets { }
- Ada.Text_IO is a package in the standard library
- Main procedure may have any name
- File called “hello_world.adb”
- Compile with: `gnatmake hello_world`

Variables

- Naming convention for all identifiers is Like_This
- Ada code is **not** case-sensitive
- Variables are declared with name first, then the type
- Variables may be initialized with a value when declared
- Compiler will warn about use of uninitialized variables

```
procedure My_Procedure is
   I : Integer := 10;
begin
   I := I + 1;
end My_Procedure;
```

Constants

- Constants may be of a type or just a named number
- Named numbers are of universal type:
  - No limits in size or precision
  - Somewhat like `#define PI 3.14` in C
- Constants of a type are like variables only ... *constant*

```
Pi      : constant := 3.14159265358979323846264 3;
Million : constant := 1_000_000;
Hex     : constant := 16#A5A5_BEEF#;
Binary  : constant := 2#1010010110100101_1011111011101111#;

CI : constant Integer := −1:
```

Scope

- Variables, types, routines and more all have a scope
- Defined in statement section by **declare – begin – end**
- Also defined by language constructs such as routines

```
procedure My_Procedure is
   X : Float := -1.0;          -- Declarations
begin

   X := X ** 2;                -- Statements, no declarations

   declare
      Y : constant := 0.1;     -- More declarations
   begin
      X := X + Y;              -- Statements
   end;

end My_Procedure;
```

Data-types

- Ada is a strongly typed language
- No implicit type-casting as in C
- Two primary classes of types:
  - Primitives
  - Composite
- The primitive types are sub-divided in:
  - Scalars
  - References

Subtypes

- A type may be defined as a subtype of another
- All values of a subtype are also values of parent
- All values of parent *need* not be values of subtype
- No typecasting is needed from subtype to type

```
declare
   subtype Decimal is Integer range 0 .. 9;
   D : Decimal;
   I : Integer := 1;
begin
   D := Decimal (I);   -- May cause constraint error
   D := D + 1;         -- May cause constraint error
   I := D;             -- Always safe
end;
```

Scalars

- Discrete scalars:
  - Enumeration types such as: `Boolean`
  - Integer types such as: `Integer`, `Natural`, `Positive`
- Real scalars:
  - Float types
  - Fixed types
- Range and storage size of types found by:
  - `Integer'First`
  - `Integer'Last`
  - `Integer'Size`

Enumeration

```
declare

   type Day is (Monday, Tuesday, Wednesday,
                Thursday, Friday, Saturday, Sunday);
   subtype Workday is Day range Monday .. Friday;

   D : Day;
   W : Workday := Monday;

begin
   W := W'Next;                 -- After this W is Tuesday
   D := W;                      -- Always safe
   W := Workday (D'Prev);       -- After this W is Monday
end;
```

Integers

- Ordinary integers and modular numbers
- Integers get constraint error when out of range
- Modular numbers wrap around
- Modular numbers also have binary-operators: **and**, **or**, **xor**

```
declare
   type Word is mod 2 ** 32;
   type Count is range 0 .. 2 ** 32 - 1;

   W : Word  := Word'Last;  -- 2 ** 32 - 1
   C : Count := Count'Last; -- 2 ** 32 - 1
begin
   W := W + 1; -- Will wrap around to 0
   C := C + 1; -- Will cause Constraint_Error!
end;
```

Real numbers

- Precision for pre-defined float types is machine dependent
- Possible to define float types with a minimal precision
- Fixed types have a fixed precision and are represented as integers on the hardware, they require no FPU

```
type Real is digits 7;
-- At least 7 digits precision, compiler will chose size

type USD is delta 0.01 range -10.0 ** 15 .. 10.0 ** 15;
-- Fixed precision of 0.01 from -1000 trillion to 1000 trillion
```

References

- May define access types for any type and routine
- Three classes of access types:
  - Those that may point only to objects on the heap
  - Those that may point to any object declared as **aliased**
  - Anonym access types that may point to any object
- Dereferenced using the **all** operator
- Need no explicit dereference when unambiguous
- Heap memory allocated with the **new** operator
- No garbage collector, need explicit deallocation!
- References are less used in Ada than in C

Example

```
declare

   type Heap_Access is access Integer;
   type All_Access is access all Integer;

   I : aliased Integer := 1;
   A : Heap_Access;                    -- null by default
   B : All_Access      := I'Access;   -- Points to I
   C : access Integer  := B;          -- Points to I

begin
  A     := new Integer'(2); -- Create integer with value 2
  B.all := 3;               -- After this I = 3
  B     := A;               -- After this B points to heap
  B.all := A.all + C.all;   -- After this A.all = B.all = 5
end;
```

Composite

- Composite types may contain:
  - Primitives
  - Other composite types
- Five classes of composite types:
  - **array**
  - **record**
  - **interface**
  - **protected**
  - **task**
- The three latter are discussed later

Array

- Consists of elements of *one* type
- May have several dimensions
- Any discrete type may be index
- May have anonymous array types
- An array type may have fixed or varying length by use of <>

```
type Vector is array (Integer range <>) of Real;

type Matrix is array (Integer range <>, Integer range <>)
  of Real;
```

Example

```
declare
   V : Vector (-10 .. 10) := (0 => 1.0, others => 0.0);

   M : Matrix := ((1.0, 2.0, 3.0),
                  (4.0, 5.0, 6.0),
                  (7.0, 8.0, 9.0));

   A : array (Weekday) of Natural := (Friday => 1,
                                      others => 0);

begin
   V (-10)     := V (0) + 2.0;                  -- Assignment
   V (2 .. 3)  := (5.0, 6.0);                   -- Slice assignment
   V           := V (-9 .. 10) & V (-10);       -- Rotate vector
   M (1, 1)    := M (2, 2);                     -- Two dimensional
   A (Monday)  := 2;                            -- Enumeration index
end;
```

Strings

- Three string types are defined in the standard library:
  - String which is an array of Character
  - Bounded_String with varying bounded length
  - Unbounded_String with varying unbounded length
- Length of String is fixed after declaration!
- Use unbounded string to get C++/Java-like strings

```ada
declare
   A : String := "Hello";
   B : String (1 .. 8);
   C : Unbounded_String := To_Unbounded_String (A);
begin
   B := A & "...";       -- Need same length for assignment
   C := C & " World!";   -- Append string to C
end;
```

Record

- Similar to struct in C (but more powerful of course)
- May be defined with default values as shown below

```
declare

   type Complex is
      record
         Re : Float := 0.0;
         Im : Float := 0.0;
      end record;

   A : Complex := (Re => 1.0, Im => 1.0);
   B : Complex := (A.Im, A.Re);

begin
   B.Re := B.Re ** 2;
end;
```

Program flow

- Ada supports standard program flow constructs:
  - **if** ... **then** ... **elsif** ... **else**
  - **case**
  - **loop**
  - **for**
  - **while**
  - **goto** (!!!)
- We'll discuss all constructs but **goto**
- The **goto** statement is considered harmful but is needed for automated code generators and in some special cases

if ... then ... elsif ... else

- Uses Boolean expressions (only)
- Boolean expressions need to be grouped with ()
- Notice = for equality and /= for inequality
- The **elsif** keyword removes ambiguity
- The **elsif** and **else** parts are optional

```
if (A and B) or C then
   ...
elsif (X = Y) xor (X /= Z) then
   ...
else
   ...
end if;
```

Case

- Like switch in C, but more powerful and no fall-through
- Works for all discrete types (integer and enumeration types)
- Character used in example below

```
case Input is
   when 'u' =>
      ...
   when 'x' | 'X' | 'q' | 'Q' =>
      ...
   when 'a' .. 'e' =>
      ...
   when others =>
      ...
end case;
```

Loop

- Ada has a construct for an eternal loop
- Broken by keyword **exit**
- Loops may be named (removes need for evil **goto**)

```
loop
   ...
   exit when Answer = 43;
   ...
end loop;

Outer:
loop
   ...
   loop
      ...
      exit Outer when Answer = 43;
   end loop;
   ...
end loop Outer;
```

For

- Iterates over a given discrete range
- The iterator is a constant within loop
- Use keyword **reverse** to iterate in reverse order
- May be broken using **exit**

```
for I in 1 .. 10 loop
   for J in reverse 1 .. 10 loop
      ...
   end loop;
end loop;

for D in Day loop
   ...
end loop;
```

While

- Like while in C
- Iterates as long as Boolean expression is true
- May be broken using **exit**

```
declare
   X : Float := 1.00000000001;
begin
   while X < 1000.0 loop
      X := X ** 2;
   end loop;
end;
```

Exceptions

- Exceptions are used for error handling
- There are some predefined exceptions:
  - Constraint_Error when value out of range
  - Program_Error for broken program control structure
  - Storage_Error when memory allocation fails
- User defined exceptions are allowed
- Exceptions are handled before **end** in a block
- After an exception is handled the block is left
- Unhandled exceptions propagate downward on call stack, program halts with error message when bottom is reached

Example

```
declare
   Wrong_Answer : exception;
begin
   ...
   if Answer /= 43 then
      raise Wrong_Answer with "Answer␣not␣43";
   end if;
   ...
exception
   when Wrong_Answer =>
      Answer := 43;
   when E : others =>
      Put_Line (Exception_Message (E));
end;
```

Routines

- There are two types of routines:
  - Procedures without return value
  - Functions with return value
- Functions *should* not have side effects
- No empty () for routines without arguments
- Routines may have default values for arguments

Procedures

- Arguments of procedures may by marked as:
  - **in**, read only (default)
  - **out**, write only
  - **in out**, read and write
- Arguments marked as **in** may be passed by copy
- Arguments marked as **out** and **in out** passed by reference
- Two procedure specifications (prototypes) are shown below

```
procedure Swap (A, B : in out Integer);
procedure Print (S : String; N : Natural := 1);
```

Example

```
procedure Swap (A, B : in out Integer) is
   T : Integer := A;
begin
   A := B;
   B := T;
end Swap;

procedure Print (S : String; N : Natural := 1) is
begin
   for I in 1 .. N loop
      Put_Line (S);
   end loop;
end Print;
...
Swap (This, That);             -- Ordinary call
Swap (B => That, A => This);   -- Named arguments
Print ("Hello", 10);
Print ("World!");
```

Functions

- Takes **in** arguments only in Ada 2005
- Ada 2012 also allows **in out** (use sparingly)
- Returns one value

```
function Sum (A, B : Integer) return Integer is
begin
   return A + B;
end Sum;

...

declare
   C : Integer;
begin
   C := Sum (1, 2);
   C := Sum (C, 3);
end;
```

Overloading

- Several routines may have the same name
- Which routine is called depends on:
  - Argument types for procedures and functions
  - Return type for functions
- Overloading decided at compile time
- Needs to be unambiguous, or compilation will fail

Packages

- The building blocks of Ada applications
- Two parts the specification (.ads) and body (.adb):
- Specification has a public and a private section:
  - Public section contain declarations visible to users
  - Private section allows to hide complexity – **abstraction**
  - Public section may define a limited view of types
  - Private section defines the full type – **Abstract Data Types**
- Body contains implementation of routines
- The body may also have internal declarations and routines

Example

```
-- File: simple_queue.ads

package Simple_Queue is

   type Queue is limited private;

   procedure Enqueue (Q : in out Queue;
                      E : in     Item);

   procedure Dequeue (Q : in out Queue;
                      E :    out Item);

   function Length (Q : Queue) return Natural;

private
   type Queue is
      record
         ...
      end record;
end Simple_Queue;
```

Example

```
-- File: simple_queue.adb

package body Simple_Queue is
   procedure Enqueue (Q : in out Queue;
                      E : in     Item) is
   begin
      ...
   end Enqueue;

   procedure Dequeue (Q : in out Queue;
                      E :    out Item) is
   begin
      ...
   end Dequeue;

   function Length (Q : Queue) return Natural is
   begin
      ...
   end Length;
end Simple_Queue;
```

Example

```
--  File: test.adb

with Simple_Queue;
use Simple_Queue;

procedure Test is
   A, B, I : Item;
   Q : Queue;
begin
   ...
   Enqueue (Q, A);
   Enqueue (Q, B);
   ...
   while Length (Q) > 0 loop
      Dequeue (Q, I);
      ...
   end if;
end Test;
```

Object-orientation

- Similar OO-model as Java:
  - Classes
  - Interfaces
- OO-model based on **tagged** records
- Interfaces were introduced with Ada 2005
- The definition of a class and its methods are usually gathered in a package, no link between class and file as in Java
- Abstract classes may have abstract and null methods
- For interfaces only abstract and null methods are allowed
- Dispatching calls only for class-wide types (Type'Class)

Example

```ada
-- File: animals.ads

package Animals is

   type Animal is abstract tagged limited private;

   procedure Make_Sound (This : Animal) is null;

   type Any_Animal is access all Animal'Class;

private

   type Animal is abstract tagged limited null record;

end Animals;
```

Example

```
-- File: animals-milk_producers.ads

package Animals.Milk_Producers is

   type Milk_Producer is limited interface;

   function Milk_Capacity (This : Milk_Producer) return Float
      is abstract;

   type Any_Milk_Producer is access all Milk_Producer'Class;

end Animals.Milk_Producers;
```

Example

```
-- File: animals-cows.ads

with Animals.Milk_Producers;
use Animals.Milk_Producers;

package Animals.Cows is

   type Cow is new Animal and Milk_Producer with private;

   procedure Make_Sound (This : Cow);
   function Milk_Capacity (This : Cow) return Float;

private

   type Cow is new Animal and Milk_Producer with
      record
         Milk : Float := 5.2;
      end record;

end Animals.Cows;
```

Example

```
-- File: animals-cows.adb

with Ada.Text_IO;
use Ada.Text_IO;

package body Animals.Cows is

   procedure Make_Sound (This : Cow) is
   begin
      Put_Line ("Mooooooooo");
   end Make_Sound;

   function Milk_Capacity (This : Cow) return Float is
   begin
      return This.Milk;
   end Milk_Capacity;

end Animals.Cows;
```

Example

```
-- File: test.adb

with Animals.Cows, Animals.Milk_Producers;
with Ada.Text_IO, Ada.Float_Text_IO;
use Animals, Animals.Milk_Producers, Animals.Cows;
use Ada.Text_IO, Ada.Float_Text_IO;

procedure Test is
   A : Any_Animal          := new Cow;
   B : Any_Milk_Producer := Any_Milk_Producer (A);
begin
   A.Make_Sound;
   Put (B.Milk_Capacity);
   New_Line;
end Test;
```

Tasking

- Ada has rich built-in support for tasking and synchronization
- Programs with tasks are easy to write compared to C/POSIX
- Multitasking programs are portable between computer architectures and operating systems
- Several scheduling policies are supported:
  - FIFO within fixed priorities
  - Round-robin within fixed priorities
  - Earliest Deadline First (EDF)
- These policies are similar to those in POSIX

Single tasks

- A single task may be created using keyword **task**
- Need package Ada.Real_Time for Time, Clock and Milliseconds
- The task has default priority since none is given

```
task Periodic;

task body Periodic is
   Next : Time := Clock;
begin
   loop
      delay until Next;
      ...
      Next := Next + Milliseconds (100);
   end loop;
end Periodic;
```

Task type

- A task types allow several similar task instances to be created
- May give a primitive argument called a *discriminant* in Ada

```
task type Worker (P : Priority; N : Character) is
   pragma Priority (P);
end Worker;

task body Worker is
begin
    Put_Line ("My␣name␣is␣" & N);
    ...
end Worker;

A : Worker (100, 'A');
B : Worker (200, 'B');
```

Execution

- Tasks are *ready* for execution when they enter scope, which task starts executing depends on scheduling
- If the tasks are in local scope, the creating task cannot leave this scope before the tasks have terminated
- Tasks that are created on library level (within a package) live for the entire execution of the program
- Tasks may also be created on heap using the **new** command

Communication

- Tasks may communicate and synchronize:
  - Synchronously through task *rendezvous*
  - Asynchronously through protected objects
- For synchronous communication a task may:
  - Have several entries used for rendezvous
  - Block waiting for several entries using **select**
  - Have a timeout when waiting on a entry
  - Have an immediate alternative if no entry is ready
- Protected objects are discussed later

Example

```
task type Runner is
   entry Start;  --  One entry, no arguments
end Runner;

task body Runner is
begin
   accept Start; --  Block here
   ...           --  Do something
end Runner;

declare
   A, B : Runner;
begin
   A.Start;      --  Start A first
   delay 1.0;
   B.Start;      --  Start B one second later
end;             --  Block here until A and B are done
```

Example

```
task type Server (S : Integer) is
   entry Write (I : Integer);
   entry Read  (I : out Integer);
end Server;

task body Server is
   N : Integer := S;
begin
   loop
      select
         accept Write (I : Integer) do
            N := I;
         end;
      or
         accept Read (I : out Integer) do
            I := N;
         end;
      end select;
   end loop;
end Server;
```

Timeout and immediate alternative

```
select
   accept Signal;
   ...  -- Do this if a task calls Signal within one second
or
   delay 1.0;
   ..   -- Else do this
end select;

select
   accept Signal;
   ..   -- Do this if a task is already blocked on Signal
else
   ...  -- Else do this immediately (same as zero timeout)
end select;
```

Protected objects

- Special composite type used for synchronization
- May have a single protected object or class of objects:
  - **protected** Name
  - **protected type** Name
- Protected objects may have:
  - Entries with a guard – may block calling tasks
  - Procedures for exclusive access to internal data
  - Functions for reading internal data (read-only)
- Entries are open or locked depending on the Boolean guard
- Calling tasks are queued on an entry (usually FIFO)

Example

- Protected object implementing a counting semaphore
- Uncommon to implement low-level semaphore using high-level protected object, normally other way around
- Done here since semaphore has well known behavior
- Notice the private part of the protected object, this part may also contain entries, procedures and functions for internal use

```
protected type Semaphore (N : Positive) is
   entry Lock;
   procedure Unlock;
   function Value return Natural;
private
   V : Natural := N;
end Semaphore;
```

Example

```
protected body Semaphore is

   entry Lock when V > 0 is
   begin
      V := V - 1;
   end Lock;

   procedure Unlock is
   begin
      V := V + 1;
   end Unlock;

   function Value return Natural is
   begin
      return V;
   end Value;

end Semaphore;
```

Example

```
task type Worker (Mutex : not null access Semaphore);

task body Worker is
begin
   Mutex.Lock;
   Put ("Starting... ");
   delay 1.0;
   Put_Line ("Done!");
   Mutex.Unlock;
end Worker;

declare
   Mutex : aliased Semaphore (1);
   A, B, C : Worker (Mutex'Access);
begin
   null;
end;
```

Advanced features

- Possible to get the number of tasks blocked on an entry using Entry_Name'Count
- Possible to move a task to the queue of another entry using **requeue** Entry_Name
- To requeue the other entry must have the same arguments or none
- A protected procedure may be used as interrupt handler
- A protected object with an interrupt handler must be at library level, that is, in a package

Example

```
pragma Unreserve_All_Interrupts;

protected Controller is

   entry Wait_Termination;

private

   entry Wait_Final;
   procedure Ctrl_C;
   pragma Attach_Handler (Ctrl_C, SIGINT);

   Count : Natural := 0;
   Final : Boolean := False;

end Controller;
```

Example

```
protected body Controller is

   entry Wait_Termination when Count > 0 is
   begin
      Count := Count - 1;
      requeue Wait_Final;
   end Wait_Termination;

   entry Wait_Final when Final is
   begin
      null;
   end Wait_Final;

   procedure Ctrl_C is
   begin
      Count := Wait_Termination'Count;
      Final := Wait_Final'Count > 0;
   end Ctrl_C;

end Controller;
```

Asynchronous abort

- Abort code asynchronously after a timeout or on a signal
- Use **delay** or **delay until** for timeout
- Use entry of protected object for signal

```
select
   delay 5.0;
   ... -- Do this when aborted
then abort
   ... -- Abort this code after 5 seconds
end select;

select
   Controller.Wait_Termination;
   ... -- Do this when aborted
then abort
   ... -- Abort when entry above is open
end select;
```

The Ravenscar profile

- The full Ada concurrent constructs have been considered non-deterministic and unsuited for high-integrity applications
- Historically the cyclic-executive has been preferred
- The Ravenscar profile defines a restricted sub-set of the concurrent constructs that are:
  - Deterministic and analyzable
  - Bounded in memory requirements
  - Sufficient for most real-time applications
- The profile also allows for efficient run-time environments by removing features requiring extensive run-time support

Some Ravenscar restrictions

- Tasks and protected objects are only allowed declared statically on library level and tasks may not terminate
- No task entries, tasks communicate only through protected objects and suspension objects
- Protected objects may have at most one entry with a simple Boolean guard and a queue length of one, no requeue
- No dynamic change of task priority with the exception of changes caused by ceiling locking
- No select and asynchronous control

Standard library

- Input / Output
- Containers (like C++/STL and Java)
- Real-time features
  - Real-time clock and task delay (high precision)
  - Scheduling such as EDF, Round Robin and dynamic priorities
  - Event handlers for task termination and timing events
  - Execution-time, overrun detection and group budgets
  - Synchronous and asynchronous task control
- Distributed programming
- Linear algebra (built upon LAPACK and BLAS)
- Networking sockets (TCP and UDP)

Thank you!