Unit 5: Introduction to ADA

"*Ada remains the Rodney Dangerfield of computer programming languages, getting little respect despite a solid technical rationale for its existence.*"
Alexander Wolf

What is it?
Ada is a programming language designed for real-time and embedded systems named in honour of Augusta Ada Byron, Countess of Lovelace (1816-1851), the assistant and patron of Charles Babbage.

Where did it originate?
Released in 1980, Ada was originally designed by Jean Ichbiah of CII Honeywell Bull, under contract to the United States Department of Defense (DoD). It was designed to replace the hundreds of programming languages used by the DoD at the time. From 1983 to 1996, the number of programming languages fell from 450 to 37.

How widespread is it?
Kind of unknown.

What makes it special?
Ada has features relevant to programming practical real-world systems:

- readability
- strong typing
- programming in the large
- exception handling
- data abstraction
- tasking
- generic units

Ada compilers also offer range and overflow checks, helping to detect defects earlier in the software development lifecycle.

Why is it used?
Ada is often used in mission critical and real-time applications. Even in todays environment Ada still incorporates strong typing, parallel processing, exception handling and generics.

Who is using it?

Ricky Sward of MITRE Corporation recently said of Ada: "*The original goals of Ada still apply to systems where a loss of life may occur, i.e. safety critical, high integrity systems.*"[1] So far from being a language whose time is over, there seems to be a renaissance of Ada. Examples include:

- *Paris Metro, Line 14*: Uses the Matra Transport automatic piloting system which controls the line's train traffic, regulates the train speed, manages several alarm devices and allows for traffic of both automatic and non-automatic trains on the same line.
  - 87,000 LOC
- *French national railroad, SNCF*: Equip trains with position and transmission systems that automatically provide location, speed, distance, switch, operations, and safety information about each train. Initial prototype was 300,000 LOC of C, but was difficult to maintain and suffered reliability problems.
  - 22,000 LOC
- *Paris -Roissy airport shuttle*: Automatic pilot for light driverless shuttle for Paris-Poissy airport.
  - alarm control unit: 50,085 LOC safety critical, 11,662 LOC non-safety
  - automatic section drivers: 186,440 LOC safety critical, 30,632 LOC non-safety
- *Swiss Postbank*: electronic funds transfer.
- *Banque Nationale de Paris*:
- *Boeing 777*: AIMS (Airplane Information Management System), 99% of software written in Ada. 613,00 LOC, 2 AIMS handle the six primary flight and navigation displays, central maintenance functions.
- *Boeing 7E7 Dreamliner*: Common Core System, 80-100 applications running simultaneously controlling avionics and utility functions.
- En Route Automation Modernization (ERAM): Lockheed Martin

Legacy issues

Ada is *only* 30 years old, and is an extremely well designed programming language. There is no need to abandon it just because it isn't "modern". Alexander Wolfe[2] states, "*... these days, Ada is just considered a remnant of bloated military engineering practices*".

---

[1] Sward, R.E., "The rise, fall and persistence of Ada", SIGAda 2010, pp.71-74 (2010)

[2] Wolfe, A., "There's still some life left in Ada", QUEUE, pp.28-31, 2004.

Basic Ada

This unit will cover the following topics:

1. Program structure
2. Compiling
3. Types
4. Variables
5. Operators and Expressions
6. Example: metres to feet-inches
7. Making decisions
8. Loops
9. Example: counting characters
10. I/O
11. User Defined types
12. Errors
13. Math packages

1. Program structure

A simple Ada program, illustrating the program structure in the classical "Hello World!" context, is shown below.

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Hello is
begin
    --program to print a greeting message
    put_line("Hello world!");
end Hello;
```

The first line allows the program to access I/O routines for text from the library **Ada.Text_IO**. The reserved words are shown in bold type. The second line introduces the procedure and gives it the name **Hello**. The reserved words **begin** and **end** delimit the program block. The first line in the block is a comment, the second outputs the text string "**Hello world!**".

 Note the filename should mirror the main procedure. For above it should be **hello.adb**.

 Semicolons terminate statements.

 Comments begin with **--**.

2. Compiling

Ada programs can be compiled using the GNAT compiler for gcc. So consider the following program:

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

To compile this, create a file called **hello.adb**, and use the following sequence to compile it:

```
> gcc -c hello.adb
```

The **-c** switch tells gcc only to do a compilation. This creates an object file "**hello.o**". It also creates an 'Ada Library Information' file, "**hello.ali**". To build an executable file, use **gnatbind** to bind the program and **gnatlink** to link it.

```
> gnatbind hello
> gnatlink hello
```

This creates an executable program called '**hello**'. The same could be achieved is one step:

```
> gnatmake hello.adb
```

3. Types

Ada has five built-in types: **integer**, **float**, **character**, **Boolean**, **string**

3.1 Integers

An **integer** is a whole number of the form:

```
37      -- the number 37
```

The clauses **'first** and **'last** can be used to enquire to the smallest and largest integers on a system, for example: **integer'first**.

3.2 Floats

Real numbers differ from integers in that they contain a decimal point. For example:

```
327.39    291.0     1.3e-7
```

These values are of type **float**. Examples of real variables and constants include:

```
radius, circumference : float;
percentage : float := 0.0;
pi : constant float := 3.1415927;
```

 Integers and reals can also contain underscore characters between adjacent digits, which make long sequences of digits more readable, but do not affect the value. For example:

```
2_000_000       3.141_592_653
```

The first example makes the value two million easier to read, as it would be expressed as 2,000,000.

 Integers are *assumed* to be decimal, but Ada also provides a facility to deal with bases other than 10 - these are known as *based* numbers. For example:

```
8#377#          - 377 octal
2#1111_1111#    - 11111111 binary
```

Ada provides the following enquiry functions:

```
float'digits  - the number of decimal digits accuracy
```

```
float'small   - the smallest positive value
float'large   - the largest positive value
float'epsilon - the 'granularity' of the representation
```

3.3 Characters and strings

Characters consist of 128 characters of the ASCII character set. Examples of character declarations include:

```
a, b, c : character;
period : constant character := '.';
```

 I/O is overloaded, so unlike C there is no need to specify formatting when using input or output functions.

A string is a composite type which is *built-in*. For example:

```
h : constant string := "jedi";
```

3.4 Boolean

There are two Boolean values and they are represented by true and false.

```
flag : Boolean := false;
success, failure : Boolean;
```

3.5 Constraints

A type declaration may be followed by a constraint. Consider the following example:

```
x : integer range 1..10;
```

Constraints do not define a new type, rather they define a *subtype* by placing a restriction on the base type.

4. Variables

4.1 Identifiers

As with other languages, an identifier must start with a letter, and may include letters, digits and the underscore character. For example, the following are valid identifiers:

```
alpha     beta      area51    area_51
```

There is no case sensitivity, so PI, Pi, pI, and pi are all the same.

4.2 Declarations

A simple declaration in Ada is of the form:

```
c : integer;
```

This introduces the variable c as an integer. An optional *initial value* can also be applied:

```
c : integer := 12;
```

A *constant* can be declared in the following manner:

```
pi : constant real := 3.14159;
```

Constants *must* be given an initial value, which may not be changed later in the program.

5. Operators and Expressions

5.1 Assignment

Assignment is performed using the **:=** operator. For example:

```
pay := basic_pay + overtime_pay;
```

5.2 Mathematical Operators

Arithmetic operators are similar to other programming languages, with the exception of the exponentiation (raising to a power) operator, where ** is used.

```
+, -, *, /, **
```

For example:

```
19 / 5 = 3
19.0 / 5.0 = 3.8
5**3 = 125.0
5**(-3) = 1.0/125.0
```

 Ada does not permit non-integer exponents for **.

 The expression **x * -4** is not allowed in Ada, probably because a similar **x--4** would imply a comment delimiter --. It will result in a compile error of the form "*missing operand*". This should be modified to **x * (-4)** and **x-(-4)**.

Ada provides two separate operators for the remainder of an integer division operation:

```
a rem b          Remainder operator rem
a mod b          Remainder operator rem
```

The best way to explain their difference is via example:

```
12 rem 5         2
12 mod 5         2
12 rem -5        2
12 mod -5        -3
```

Ok, so you're looking at the last answer and asking why?? aren't you? Well, the **mod** function is defined as the amount by which a number exceeds the largest integer multiple of the divisor that is not greater than that number. In the case of 12 mod -5, imagine that 12 sits between 10 and 15, and 12-15 is -3.

5.3 Relational and logical operators

There are six *relational* operators:

```
=                -- equals
/=               -- not equals
<, <=, >, >=
```

Strings can be compared using these operators. For example "Lara" < "Mara" would evaluate to true.

There are four *logical* operators:

```
and or xor not
```

For example:

```
number rem 2 = 0 or number > 0
```

This is true if either of the two relational expressions is true, and false only if they are both false. The operators **and**, **or** and **xor** all have the same precedence, which is lower than the relational operators, hence parentheses are not required. Parentheses can be included:

```
(element rem 2) = 0 and element > 0
```

5.4 Numeric Type Conversion

Numeric conversion is achieved using two functions:

`real(3)`	converts 3 to 3.0
`integer(7.4)`	converts to integer rounded to the nearest integer

5.5 Membership Tests

To determine whether or not a value is within a certain range the membership tests **in** and **not in** can be used. For example

```
month in 1 .. 12
```

This is true if the value of pay is within the range 1 to 12 inclusive, and is false otherwise. It is equivalent to the more complicated expression:

```
month >= 1 and month <= 12
```

5.6 Absolute Value

The absolute value can be calculated using the **abs** function:

```
abs(-2.7) = 2.7
```

5.7 String Concatenation

The operator **&** is used to construct a single string from two smaller strings.

```
"darth" & "vader"   -- result is "darthvader"
```

6. Example: metres to feet-inches

The following example converts an input in metres (metric) to corresponding feet and inches (imperial).

```
with ada.Text_IO; use Ada.Text_IO;
with ada.Integer_Text_IO; use Ada.Integer_Text_IO;
with ada.Float_Text_IO; use ada.Float_Text_IO;
procedure convert is
    yards, feet, inches : integer;
    metres : float;
    conversion_factor : constant float := 39.37;
begin
    put("Number of metres: ");
    get(metres);

    inches := integer(metres * conversion_factor);
    feet := inches / 12;
    inches := inches rem 12;
    yards := feet / 3;
    feet := feet rem 3;

    put(metres, aft=>2, exp=>0); put(" metres = ");
    put(yards); put(" yards,");
    put(feet, width => 2); put(" feet,");
    put(inches, width => 3); put(" inches");

    new_line;
end convert;
```

When compiled and executed gives the following output:

```
Number of metres: 10
10.00 metres =          10 yards, 2 feet, 10 inches
```

The interesting part is in the output using the **put** functions. When outputting a float, the **aft** clause specifies the number of decimal places, and **exp** the exponent. When outputting an integer, the clause **width** denotes the width of the output field.

7. Making decisions

7.1 Basic if

The simplest form of the **if** statement is:

```
if expr then
    ....
end if;
```

For example:

```
if num rem 2 = 0 then
    put(" even ");
end if;
```

7.2 if-else

An **else** statement can also be added:

```
if num rem 2 = 0 then
    put(" even ");
else
    put(" odd ");
end if;
```

7.3 if-elsif-else

Or a series of choices:

```
if num = 0 then
    put(" zero ");
elsif num rem 2 = 0 then
    put(" even ");
else
    put(" odd ");
end if;
```

Another example involves characters:

```
if ch in 'a' .. 'z' then
    put("lower case");
elsif ch in 'A' .. 'Z' then
    put("upper case");
```

```
elsif ch in '0' .. '9' then
    put("digit");
else
    put("other character");
end if;
```

 The syntax of the **if** statement in Ada solves the "dangling-else" problem inherent to languages such as C.

7.4 case

Ada also provides a **case** statement. For example:

```
case month is
    when 4 | 6 | 9 | 11 =>
        = 30;
    when 2 =>
        if leap_year then
            ndays = 29;
        else
            ndays = 28;
        end if;
    when 1 | 3 | 5 | 7 | 8 | 10 | 12 =>
        ndays = 31;
end case;
```

Choices must be constant expressions or constant discrete ranges, and are separated by the symbol "|", which should be read as "or".

The last when statement could have been re-written as

```
when others =>
    ndays = 31;
```

 Ada provides a null statement which specifies an empty action or no-operation.

```
null;
```

8. Loops

8.1 basic loop

Execute the same sequence of statements repeatedly. The most basic loop is shown in the example below.

```
number: integer;
loop
    get(number);
    put(number);
end loop;
```

This essentially loops forever. To exit, we add an **exit when** statement, of the form:

```
sum := 0.0;
loop
    get(number);
    exit when number < 0.0;
    sum := sum + number;
end loop;
```

This loop repeats, reading in a **number** and adding it to **sum** until a **number** less than zero is entered. This allows the loop to exit in the middle. To allow it to exit at the end, the **exit** statement should be the last before the **end loop**. The **exit** statement is equivalent to

```
if number < 0.0 then
    exit;
end if;
```

8.2 while loop

There is also a **while** loop of the form:

```
while expr loop
    ...
end loop;
```

If the value of *expr* is true, then the body of the loop is executed. For example:

```
n : integer := 12;
i : integer := 1;
fact : integer := 1;
```

```
while i <= n loop
    fact := fact * i;
    i := i + 1;
end loop;
```

8.3 for loop

For a known number of iterations, there is the **for** loop of the form:

```
for index in x..y loop
    ...
end loop;
```

The *index* has values from *x* to *y*. The factorial in the previous example, is shown below using a for loop.

```
for i in 1..n loop
    fact := fact * i;
end loop;
```

You do not have to declare the loop *index*, because it is implicitly declared when it appears in the **for** loop. There is also a form of the **for** loop which takes the values of the index range in reverse order:

```
for index in reverse x..y loop
    ...
end loop;
```

8.4 exit and exit when

The exit statement is used to terminate an enclosing loop, causing execution to continue after the loop. The exit statement also allows for a optional condition expression (when), to determine whether or not a loop should be terminated. For example:

```
if m < 0 then
    exit;                OR        exit when m < 0;
end if;
```

9. Example: Counting characters

```
with ada.Text_IO; use Ada.Text_IO;
with ada.Integer_Text_IO; use Ada.Integer_Text_IO;
procedure sentence is
    upperC, lowerC, blankC, punctC : integer := 0;
    ch : character;
begin
    put("Enter a sentence: ");
    loop
        get(ch);
        if ch in 'a' .. 'z' then
            lowerC := lowerC + 1;
        elsif ch in 'A' .. 'Z' then
            upperC := upperC + 1;
        elsif ch = ' ' then
            blankC := blankC + 1;
        else
            punctC := punctC + 1;
        end if;

        exit when ch = '.';
    end loop;

    put("Uppercase characters = ");
    put(upperC, width => 1); new_line;
    put("Lowercase characters = ");
    put(lowerC, width => 1); new_line;
    put("Blank characters = ");
    put(blankC, width => 1); new_line;
    put("Punctuation characters = ");
    put(punctC, width => 1); new_line;
end
```

An example of this program running is:

```
Enter a sentence: The quick brown fox jumped over the lazy dog.
Uppercase characters =  1
Lowercase characters = 35
Blank characters =  8
Punctuation characters =  1
```

10. I/O

10.1 I/O Packages

Ada provides several packages which deal with I/O: Text_IO, Sequential_IO, Direct_IO, Stream_IO. The latter three deal with binary data, Text_IO deals with plain text. A summary of the packages can be found here:

```
http://www.infeig.unige.ch/support/ada/gnatlb/index.html
```

Three core packages used are:

```
ada.text_io              - character I/O
ada.integer_text_io      - integer I/O
ada.float_text_io        - floating-point I/O
```

10.2 get and put

The two core I/O functions are **get** and **put**. Ada uses overloading to use the same function to input various types of data. For example for integers:

```
get(num);
get(num, n);
```

where **num** is the variable to be input/output, and **n** is field size for input/output. For floats:

```
get(num);
```

where **num** is the variable to be input/output.

A *width* parameter allows for formatting. When a larger value than is absolutely necessary is specified for the width, an appropriate number of spaces is output in front of the number. The following statements are equivalent:

```
put(number, 2);
put(number, width => 2);
```

The number of print positions before the decimal point is controlled by the **fore** parameter, the number of digits printed after the decimal point by the **aft** parameter, and the number of print positions used in the exponent by the **exp** parameter.

```
amount : float := 12345.678;

put(amount, fore => 3);
```

```
put(amount, aft => 5);
put(amount, fore => 1, aft => 3, exp => 2);
put(amount, 1, 3, 2);

number =   1.2345678E+04
number =  1.23457E+04
number = 1.235E+4
number = 1.235E+4
```

10.3 new_line, get_line and put_line

The function **new_line** adds a carriage return to the output. To add *n* number of carriage returns, use:

```
new_line(n);
```

Similar output can be achieved using the **put_line** function. The following are equivalent:

```
put_line("hello");
put("hello"); new_line;
```

Strings can be concatenated in **put_line**:

```
put_line("darth" & "vader");
```

To print text and an integer in the same **put_line** using the function **integer'image**:

```
number : integer := 42;
put_line("darth" & integer'image(number));
```

And floats can be similarly output using the **float'image** function. The function **image** returns the string representation of its argument.

The **get_line** function inputs a line of text until the receiving string fills up. For example:

```
str : string(1..100);
i : integer;
get_line(str,i);
```

The variable **i** holds the length of the string.

10.4 Miscellaneous I/O control functions

```
set_col(p)    -- set the current column to p in output
set_line(p)   -- set the current line to p in output
```

```
new_page      -- go to the next page of output
new_line      -- go to the next line of output
skip_line     -- go to the start of next line of input
skip_page     -- go to the start of next page of input
```

11. User Defined types

11.1 Enumeration types

Enumeration types define a new type and a set of values for that type. For example:

```
type colour is (red, yellow, blue, green);
type status is (on,off);
type lens is (fisheye, wideangle, standard, telephoto);
type month is
     (jan,feb,mar,apr,may,jun,jul,aug,sep,oct,nov,dec);
```

Enumeration types define an ordering, so they can be compared using relational operators. For example **on** < **off**. They can also be used in loops:

```
for m in lens loop
for this_month in jan .. dec loop
for this_month in month'first .. month'last loop
for this_month in month loop
```

11.2 Subtypes

A subtype allows a constrained version of a type to be *named*.

```
subtype gregorian is integer range 1582..2100;
```

11.3 Attributes

Ada defines several attributes of types (shown in **bold** below):

```
integer'first       -smallest possible integer
integer'last        -largest possible integer
month'first         -value is jan
month'pos(jan)      -value is 0
month'val(11)       -value is dec
month'succ(mar)     -value is apr
month'pred(mar)     -value is feb
```

The function **image** returns the string representation of its argument. For example:

```
integer'image(7+9)  -the string " 16"
```

11.4 Types and Integers

Consider the following variables :

```
number_of_rice_grains : integer;
page_number : integer;
day_of_year : integer;
```

The problem with these variables, as in most languages is that they are all of the same type, even though they all have differing constraints. All are non-negative, but **number_of_rice_grains** is likely to be a large number, whereas, **page_number** may be limited to 5000, and **day_of_year** to 365 or 366. Most languages also don't provide a means of checking for errors. Ada does.

```
type days is range 1..366;
type pages is range 1..5000;

page_number : days;
day_of_year : pages;
number_of_rice_grains : integer;
```

Now a statement of the form:

```
day_of_year = 367;
```

would cause an error, because **day_of_year** cannot have a value greater than 366.

 Newly defined types *inherit* the operators and functions defined for objects of type integer.

11.5 Types and Floats

The same can be achieved for floats:

```
type smll_flt is digits 5;
type mdm_flt is digits 7 range 1.0..1.0E6;
type lrg_flt is digits 10 range 0.0..1.0E3
```

For example variables of type **smll_flt** can represent numbers with an accuracy of 5 digits, such as 2.7159, 2.7159E7; while variables of type **mdm_flt**, can represent 3.141593, and 31415.93, but NOT -3.141593, because of the range constraint included.

11.6 I/O for User Types

In order to input and output user types, an instantiation of the package **enumeration_io** must be made. For example:

```
type month is
     (jan,feb,mar,apr,may,jun,jul,aug,sep,oct,nov,dec);

package month_io is new enumeration_io(month);
use month_io;
```

This allows I/O for the enumerated values in **month**, using the usual I/O functions such as get and put. Failure to input one of the specified enumerated values will raise a **data_error** exception.

12. Errors

Ada is EXTREMELY good at checking for errors at compile time and run time.

12.1 Constraint Error

Consider the following code:

```
with ada.text_io; use ada.text_io;
with ada.integer_text_io; use ada.integer_text_io;
procedure errors is
    small_num : integer range 1..10;
    large_num : integer range 100..1000;
begin
    get(small_num);
    get(large_num);
    small_num := large_num;
    put(small_num); new_line;
end errors;
```

At compile time, the following message appears:

```
errors.adb:9:18: warning: value not in range of subtype of
"Standard.Integer" defined at line 4
errors.adb:9:18: warning: "Constraint_Error" will be raised
at run time
```

At run time, when an attempt is made to enter the value 50, the following exception is raised:

```
raised CONSTRAINT_ERROR : errors.adb:7 range check failed
```

13. Math Packages

Numerical functions can be found in the package **Ada.Numerics.Elementary_Functions** . Here are some of the core functions:

`arccos(x)`	arc cosine (inverse cosine)
`arcsin(x)`	arcsine (inverse sine)
`arctan(x)`	arctangent (inverse tangent)
`cos(x)`	cosine
`exp(x)`	*e* raised to the *x*th power
`log(x)`	natural (base *e*) logarithm
`log(x,b)`	logarithm to base **b**.
`sin(x)`	sine
`sqrt(x)`	square root of x, for x≥0
`tan(x)`	tangent