FREE eBook

LEARNING ada

Free unaffiliated eBook created from **Stack Overflow contributors.**

#ada

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: ada

It is an unofficial and free ada ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official ada.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with ada

Remarks

Ada is an internationally standardized, high-level, object-oriented computer programming language that supports strong typing and structured programming. More information may be found here.

Versions

Version	Release Date
Ada 2012(TC-1)	2016-04-01
Ada 2012	2012-12-10
Ada 2005	2007-01-01
Ada 95	1995-12-10
Ada 83	1983-01-01

Examples

Installation or Setup

Ada is a programming language for which there exists multiple compilers.

- One of these compilers, and perhaps the most used, is GNAT. It is part of the GCC toolchain. It can be installed from several sources:
    - The yearly GPL release done by AdaCore, available for free on libre site. This version has undergone all internal testing that AdaCore does for its pro releases, is available on a large number of platforms. The compiler and its runtime are released under the GPL license, and, unless you are using no runtime, any executables you distribute will also be covered by this license. For academics and projects in their initial stages, this is not a problem.
    - The FSF gcc receives the same patches regularly. The version of GNAT might not be always up-to-date, but catches up regularly.
    - A number of contributors are packaging that FSF version for various Linux distributions (Debian-based systems, among others) and binaries for Mac OS X. Using the package manager from your distribution might be the simplest way to install GNAT. Such versions come with the standard GCC license, and allow you to write closed source

code.

- AdaCore also provides GNAT Pro, which comes with the standard GCC license which allows you to write closed source code. More importantly perhaps, it comes with support, should you have questions on the use of the language, tools, how to best implement something, and of course bug reports and enhancement requests.

Another number of compilers are listed in the Ada WikiBook, together with installation instructions. Getadanow.com features editions of FSF GNAT, ready-made for various operating systems on several types of hardware, or virtual machines. The site also collects resources for learning and sharing Ada.

Hello World

```
with Ada.Text_IO;

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

Alternatively, after importing the package Ada.Text_IO, you can say `use Ada.Text_IO;` in order to be able to use Put_Line without explicitly declaring what package it should come from, as such:

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

procedure Hello_World is
begin
    Put_Line ("Hello World");
end Hello_World;
```

If you are using the `gnat` compiler, this simple program can be compiled with

```
gnatmake hello_world
```

This will generate a number of files, including a `hello_world` (or `hello_world.exe` on Windows) that you can execute to see the famous message. The name of the executable is computed automatically from the name of the main Ada subprogram. In Ada a main subprogram can have any name. It only has to be a parameter-less procedure, that you give as an argument to `gnatmake`.

Other compilers have similar requirements, although of course the build command is different.

Version

The standard Ada programming language is defined in the *Ada Reference Manual*. Interim version changes and release notes are discussed in the corresponding rationale documents. Implementations typically document their compliance with the standard in the form of a user guide and/or reference manual, for example.

- Ada 2012

  - *Ada 2012 Language Reference Manual*
  - *Rationale for Ada 2012*

- Ada 2005

  - *Ada 2005 Language Reference Manual*
  - *Rationale for Ada 2005*

- Ada 95

  - *Ada 95 Language Reference Manua*l
  - *Rationale for Ada 95*

- Ada 83

  - *Ada 83 Language Reference Manua*l
  - *Ada 83 Rationale for the Design of the Ada® Programming Language*

Libraries

As for any programming language, Ada comes with extensive libraries to accomplish various tasks. Here are some pointers to some of them, although searching on github will lead some more.

- The Ada runtime itself, distributed will all compilers, includes an extensive set of packages and annexes, ranging from data structures and containers, to input/output, string manipulation, time manipulation, files, numeric computations, multi-tasking, command line switches, random numbers,...

- The GNAT compiler comes with its own extended runtime, with new packages in the `GNAT` hierarchy, that provide support for regular expressions, sorting, searching, unicode, CRC, time input/output, ...

- gnatcoll is a library that is available from AdaCore's libre site, and includes an extensive logging framework, extending applications with python, mmap, an extensive framework to interface with file systems, parsing email messages and mailboxes, an extensive framework to interact with databases in a type-safe manner, interface to various libraries like icon, readline, terminal colors, support for reference counted types for automatic memory management, JSON files,...

- XML/Ada is a library to parse and validate XML documents

- GtkAda is a full binding to the gtk+ library, that let's you write portable user interfaces on Unix, Windows and OSX.

- AWS is a framework to create web servers in Ada, with full support for various protocols like HTTP, Websockets,... and its own template system.

Read Getting started with ada online: https://riptutorial.com/ada/topic/3900/getting-started-with-ada

Chapter 2: Attribute Image

Introduction

Subtype attributes `'Image` and `'Value` will take, respectively, a scalar value and a string and they return, respectively, a string and a scalar value. The result of `'Image` can be input to `'Value` to get the original value. The converse is also true.

The `__Scalar_Object__'Image` attribute can be used directly on objects (since Ada 2012-TC-1).

Syntax

- function Scalar'Image (Argument : Scalar'Base) return String;
- function Discrete'Image (Argument : Discrete'Base) return String;
- function Integer'Image (Argument : Integer'Base) return String;
- function Enumeration'Image (Argument : Enumeration'Base) return String;
- function Real'Image (Argument : Real'Base) return String;
- function Numeric'Image (Argument : Numeric'Base) return String;
- function Scalar'Value (Argument : String) return Scalar'Base;
- function Discrete'Value (Argument : String) return Discrete'Base;
- function Integer'Value (Argument : String) return Integer'Base;
- function Enumeration'Value (Argument : String) return Enumeration'Base;
- function Real'Value (Argument : String) return Real'Base;
- function **Scalar_Object**'Image return String;

Remarks

Note that `'Image` can incur implementation defined results (RM 3.5), namely when some graphic characters needed for the `String` result are not defined in `Character`. Consider the larger repertoires of `'Wide_Image` and `'Wide_Wide_Image`.

Ada 2012(TC-1)

The permission to use the attribute `__Scalar_Object__'Image` directly on an object was added in Ada 2012-TC-1 (April 2016).

Examples

Print out float using the Image attribute

Ada 2012(TC-1)

```
with Ada.Text_IO;

procedure Main is
```

```
   type Some_Float digits 8 range 0.0 .. 10.0;
   X : Some_Float := 2.71;
begin
   Ada.Text_IO.Put_Line (X'Image);
end Main;
```

Result

```
2.71000E+00
```

Print out integer using the Image attribute

Ada 2012(TC-1)

```
with Ada.Text_IO;

procedure Main is
   type Some_Integer is range -42 .. 42;
   X : Some_Integer := 17;
begin
   Ada.Text_IO.Put_Line (X'Image);
end Main;
```

Result

```
17
```

Print out enumeration using the Image attribute

Ada 2012(TC-1)

```
with Ada.Text_IO;

procedure Main is
   type Fruit is (Banana, Orange, Pear);
   X : Fruit := Orange;
begin
   Ada.Text_IO.Put_Line (X'Image);
   Ada.Text_IO.Put_Line (Pear'Image);
end Main;
```

Result

```
ORANGE
PEAR
```

Print out Enumeration using attribute Image

```
with Ada.Text_IO;

procedure Main is
   type Fruit is (Banana, Orange, Pear);
   X : Fruit := Orange;
begin
   Ada.Text_IO.Put_Line (Fruit'Image (X));
end Main;
```

Result

```
ORANGE
```

Print out Integer using attribute Image

```
with Ada.Text_IO;

procedure Main is
   X : Integer := 17;
begin
   Ada.Text_IO.Put_Line (Integer'Image (X));
end Main;
```

Result

```
17
```

Print out Float using attribute Image

```
with Ada.Text_IO;

procedure Main is
   X : Float := 2.71;
begin
   Ada.Text_IO.Put_Line (Float'Image (X));
end Main;
```

Result

```
2.71000E+00
```

As Inverses

```
with Ada.Text_IO;

procedure Image_And_Value is
   type Fruit is (Banana, Orange, Pear);
   X  : Fruit := Orange;
begin
   Ada.Text_IO.Put_Line (Boolean'Image
      (Fruit'Value (Fruit'Image (X)) = X
          and
       Fruit'Image (Fruit'Value ("ORANGE")) = "ORANGE"));
end Image_And_Value;
```

Result

```
TRUE
```

Read Attribute Image online: https://riptutorial.com/ada/topic/4290/attribute-image

Chapter 3: Enumeration

Syntax

- function Enumeration'Image (Argument : Enumeration'Base) return String;
- function Enumeration'Img return String; -- GNAT
- function Enumeration'Val (Argument : Universal_Integer) return Enumeration'Base;
- function Enumeration'Pos (Argument : Enumeration'Base) return Universal_Integer;
- function Enumeration'Enum_Rep (Argument : Enumeration'Base) return Universal_Integer;
- function **Literal**'Enum_Rep return Universal_Integer; -- GNAT
- function **Literal**'Address return System.Address;
- for Enumeration use (Literal_1 => **Universal_Integer**, Literal_n => **Universal_Integer**);
- (**Literal** in Enumeration) return Boolean;

Examples

Iterating literals

A literal inside a enumeration is a discrete type so we can use attribute `Image` to find out which literal it is as text form. Notice that this prints out the same word as in the code (but in upper case).

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

procedure Main is
   type Fruit is (Banana, Pear, Orange, Melon);
begin
   for I in Fruit loop
      Put (Fruit'Image (I));
      New_Line;
   end loop;
end;
```

Result

```
BANANA
PEAR
ORANGE
MELON
```

Using package Enumeration_IO

Instead of attribute `Image` and `Ada.Text_IO.Put` on enumeration literals we can only use the generic package `Ada.Text_IO.Enumeration_IO` to print out the literals.

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

```
procedure Main is
   type Fruit is (Banana, Pear, Orange, Melon);
   package Fruit_IO is new Enumeration_IO (Fruit); use Fruit_IO;
begin
   for I in Fruit loop
      Put (I);
      New_Line;
   end loop;
end;
```

Result

```
BANANA
PEAR
ORANGE
MELON
```

First character upper case rest lower case literals

Attribute `Image` capitalizes all characters of enumeration literals. The function `Case_Rule_For_Names` applies upper case for the first character and makes the rest lower case.

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Strings.Maps.Constants; use Ada.Strings.Maps.Constants;
with Ada.Strings.Fixed; use Ada.Strings.Fixed;

procedure Main is
   type Fruit is (Banana, Pear, Orange, Melon);
   function Case_Rule_For_Names (Item : String) return String is
   begin
      return Translate (Item (Item'First .. Item'First), Upper_Case_Map) & Translate (Item
(Item'First + 1 .. Item'Last), Lower_Case_Map);
   end;
begin
   for I in Fruit loop
      Put (Case_Rule_For_Names (Fruit'Image (I)));
      New_Line;
   end loop;
end;
```

Result

```
Banana
Pear
Orange
Melon
```

Title Case, Using Enumeration_IO, For a Subrange

Combining change of character case with Enumeration_IO and using a text buffer for the image. The first character is manipulated in place.

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Characters.Handling; use Ada.Characters.Handling;

procedure Main is
    type Fruit is (Banana, Pear, Orange, Melon);
    package Fruit_IO is new Enumeration_IO (Fruit);
    Buffer : String (1 .. Fruit'Width);
begin
   for I in Fruit range Pear .. Fruit'Last loop
       Fruit_IO.Put (To => Buffer,
                     Item => I,
                     Set => Lower_Case);
       Buffer (Buffer'First) := To_Upper (Buffer (Buffer'First));
       Put_Line (Buffer);
   end loop;
end;
```

Result

```
Pear
Orange
Melon
```

Read Enumeration online: https://riptutorial.com/ada/topic/5930/enumeration

Chapter 4: Files and I/O streams

Remarks

The Ada standard library provides for I/O of traditional files of text or binary data, as well as I/O of streamed files. Files of binary data will be sequences of values of a type, while stream files can be sequences of values of possibly different types.

To read and write elements of different types from/to stream files, Ada uses subprograms denoted by types' attributes, namely `'Read`, `'Write`, `'Input`, and `'Output`. The latter two will read and write array bounds, record discriminants, and type tags, in addition to the bare input and output that `Read` and `'Write` will perform.

Examples

Create and write to file

The procedures `Create`, `Put_Line`, `Close` from the package `Ada.Text_IO` is used to create and write to the file `file.txt`.

```
with Ada.Text_IO;

procedure Main is
   use Ada.Text_IO;
   F : File_Type;
begin
   Create (F, Out_File, "file.txt");
   Put_Line (F, "This string will be written to the file file.txt");
   Close (F);
end;
```

Resulting file `file.txt`

```
This string will be written to the file.txt
```

Create And Write To A Stream

The subtypes' stream-oriented attributes are called to write objects to a file, bare and using binary default representations.

```
with Ada.Streams.Stream_IO;

procedure Main is
   type Fruit is (Banana, Orange, Pear);
   type Color_Value is range 0 .. 255;
   type Color is record
```

```
      R, G, B : Color_Value;
   end record;

   Fruit_Colors : constant array (Fruit) of Color :=
     (Banana => Color'(R => 243, G => 227, B => 18),
      Orange => Color'(R => 251, G => 130, B => 51),
      Pear   => Color'(R => 158, G => 181, B => 94));

   use Ada.Streams.Stream_IO;

   F : File_Type;

begin
   Create (F, Name => "file.bin");
   for C in Fruit_Colors'Range loop
      Fruit'Write (Stream (F), C);
      Color'Write (Stream (F), Fruit_Colors (C));
   end loop;
   Close (F);
end Main;
```

Resulting File

```
00000000  00 2e f3 00 e3 00 12 00  01 2e fb 00 82 00 33 00
00000010  02 2e 9e 00 b5 00 5e 00
```

Open And Read From Stream File

Read the data of Create And Write To A Stream back into a program.

```
with Ada.Streams.Stream_IO;

procedure Main is
   --
   --  ... same type definitions as in referenced example
   --
   Fruit_Colors : array (Fruit) of Color;

   use Ada.Streams.Stream_IO;

   F : File_Type;
   X : Fruit;
begin
   Open (F, Mode => In_File, Name => "file.bin");
   loop
      Fruit'Read (Stream (F), X);
      Color'Read (Stream (F), Fruit_Colors (X));
   end loop;
exception
   when End_Error =>
      Close (F);
   pragma Assert  -- check data are the same
     (Fruit_Colors (Banana) = Color'(R => 243, G => 227, B => 18) and
      Fruit_Colors (Orange) = Color'(R => 251, G => 130, B => 51) and
      Fruit_Colors (Pear)   = Color'(R => 158, G => 181, B => 94));
end Main;
```

Read Files and I/O streams online: https://riptutorial.com/ada/topic/8865/files-and-i-o-streams

Chapter 5: Genericity in Ada

Examples

Generic Subprograms

Generic subprograms are usefull to create a subprograms that have the same structure for several types. For example, to swap two objects:

```
generic
    type A_Type is private;
procedure Swap (Left, Right : in out A_Type) is
    Temp : A_Type := Left;
begin
    Left := Right;
    Right := Temp;
end Swap;
```

Generic Packages

In Ada generic package, upon instantiation, data are duplicated; that is, if they contain global variables, each instance will have its own copy of the variable, properly typed and independent from the others.

```
generic
    type T is private;
package Gen is
    type C is tagged record
        V : T;
    end record;
    G : Integer;
end Gen;
```

Generic Parameters

Ada offers a wide variety of generic parameters which is difficult to translate into other languages. The parameters used during instantiation and as a consequence those on which the generic unit may rely on may be variables, types, subprograms, or package instances, with certain properties. For example, the following provides a sort algorithm for any kind of array:

```
generic
    type Component is private;
    type Index is (<>);
    with function "<" (Left, Right : Component) return Boolean;
    type Array_Type is array (Index range <>) of Component;
procedure Sort (A : in out Array_Type);
```

Read Genericity in Ada online: https://riptutorial.com/ada/topic/9322/genericity-in-ada

Chapter 6: Implementing the producer-consumer pattern

Introduction

A demonstration of how the producer-consumer pattern is implemented in Ada.

Syntax

- function Scalar'Image (Argument : Scalar'Base) return String;
- task Task_Name;
- task Task_Name is Entries end;
- task body Task_Name is Declarations begin Code end;
- entry Entry_Name;
- accept Entry_Name;
- exit;

Remarks

The examples *should* all ensure proper task termination.

Examples

Using a synchronized buffer

```
with Ada.Containers.Synchronized_Queue_Interfaces;
with Ada.Containers.Unbounded_Synchronized_Queues;
with Ada.Text_IO;

procedure Producer_Consumer_V1 is
   type Work_Item is range 1 .. 100;

   package Work_Item_Queue_Interfaces is
     new Ada.Containers.Synchronized_Queue_Interfaces
          (Element_Type => Work_Item);

   package Work_Item_Queues is
     new Ada.Containers.Unbounded_Synchronized_Queues
          (Queue_Interfaces => Work_Item_Queue_Interfaces);

   Queue : Work_Item_Queues.Queue;

   task type Producer;
   task type Consumer;

   Producers : array (1 .. 1)  of Producer;
   Consumers : array (1 .. 10) of Consumer;
```

```
   task body Producer is
   begin
      for Item in Work_Item loop
         Queue.Enqueue (New_Item => Item);
      end loop;
   end Producer;

   task body Consumer is
      Item : Work_Item;
   begin
      loop
         Queue.Dequeue (Element => Item);
         Ada.Text_IO.Put_Line (Work_Item'Image (Item));
      end loop;
   end Consumer;

begin
   null;
end Producer_Consumer_V1;
```

Notice that I've been lazy here: There is no proper termination of the consumer tasks, once all work items are consumed.

Producer-Consumer pattern using the Ada Rendezvous mechanism

A synchronous producer-consumer solution ensures that the consumer reads every data item written by the producer exactly one time. Asynchronous solutions allow the consumer to sample the output of the producer. Either the consumer consumes the data faster than it is produced, or the consumer consumes the data slower than it is produced. Sampling allows the consumer to handle the currently available data. That data may be only a sampling of the data produced, or it may be already consumed data.

```
------------------------------------------------------------------
-- synchronous PC using Rendezvous --
------------------------------------------------------------------
with Ada.Text_IO; use Ada.Text_IO;

procedure PC_Rendezvous is
   task Producer;
   task Consumer is
      entry Buf(Item : in Integer);
   end Consumer;
   task body Producer is
   begin
      for I in 1..10 loop
         Put_Line("Producer writing" & Integer'Image(I));
         Consumer.Buf(I);
      end loop;
   end Producer;
   task body Consumer is
      Temp : Integer;
   begin
      loop
         select
            accept Buf(Item : in Integer) do
               temp := Item;
            end;
```

```
            Put_Line("Consumer read" & Integer'Image(Temp));
         or
            terminate;
         end select;
      end loop;
   end Consumer;

begin
   null;
end PC_Rendezvous;
```

Producer-Consumer with a sampling consumer

This example uses the main procedure as the producer task. In Ada the main procedure always runs in a task separate from all other tasks in the program, see minimal example.

```
------------------------------------------------------------------
-- Sampling Consumer --
------------------------------------------------------------------
with Ada.Text_IO; use Ada.Text_IO;

procedure Sampling_PC is
   protected Buf is
      procedure Write(Item : in Integer);
      function Read return Integer;
      procedure Set_Done;
      function Get_Done return Boolean;
   private
      Value : Integer := Integer'First;
      Is_Done : Boolean := False;
   end Buf;
   protected body Buf is
      procedure Write(Item : in Integer) is
      begin
         Value := Item;
      end Write;
      function Read return Integer is
      begin
         return Value;
      end Read;
      procedure Set_Done is
      begin
         Is_Done := True;
      end Set_Done;
      function Get_Done return Boolean is
      begin
         return Is_Done;
      end Get_Done;
   end Buf;

   task Consumer;
   task body Consumer is
   begin
      while not Buf.Get_Done loop
          Put_Line("Consumer read" & Integer'Image(Buf.Read));
      end loop;
   end Consumer;

begin
```

```
   for I in 1..10 loop
     Put_Line("Producer writing" & Integer'Image(I));
     Buf.Write(I);
   end loop;
   Buf.Set_Done;
end Sampling_PC;
```

Multiple Producers and Consumers Sharing the same buffer

This example shows multiple producers and consumers sharing the same buffer. Protected entries in Ada implement a queue to handle waiting tasks. The default queuing policy is First In First Out.

```
------------------------------------------------------------------
-- Multiple producers and consumers sharing the same buffer --
------------------------------------------------------------------
with Ada.Text_IO; use Ada.Text_Io;

procedure N_Prod_Con is
   protected Buffer is
      Entry Write(Item : in Integer);
      Entry Read(Item : Out Integer);
   private
      Value  : Integer := Integer'Last;
      Is_New : Boolean := False;
   end Buffer;
   protected body Buffer is
      Entry Write(Item : in Integer) when not Is_New is
      begin
         Value := Item;
         Is_New := True;
      end Write;
      Entry Read(Item : out Integer) when Is_New is
      begin
         Item := Value;
         Is_New := False;
      end Read;
   end Buffer;

   task type Producers(Id : Positive) is
      Entry Stop;
   end Producers;
   task body Producers is
      Num : Positive := 1;
   begin
      loop
         select
            accept Stop;
            exit;
         or
            delay 0.0001;
         end select;
         Put_Line("Producer" & Integer'Image(Id) & " writing" & Integer'Image(Num));
         Buffer.Write(Num);
         Num := Num + 1;
      end loop;
   end Producers;

   task type Consumers(Id : Positive) is
      Entry Stop;
```

```
   end Consumers;

   task body Consumers is
      Num : Integer;
   begin
      loop
         select
            accept stop;
            exit;
         or
            delay 0.0001;
         end select;
         Buffer.Read(Num);
         Put_Line("Consumer" & Integer'Image(ID) & " read" & Integer'Image(Num));
      end loop;
   end Consumers;
   P1 : Producers(1);
   P2 : Producers(2);
   P3 : Producers(3);
   C1 : Consumers(1);
   C2 : Consumers(2);
   C3 : Consumers(3);
begin
   delay 0.2;
   P1.Stop;
   P2.Stop;
   P3.Stop;
   C1.Stop;
   C2.Stop;
   C3.Stop;
end N_Prod_Con;
```

Read Implementing the producer-consumer pattern online:
https://riptutorial.com/ada/topic/8632/implementing-the-producer-consumer-pattern

Chapter 7: Outputting numbers

Introduction

Ada's standard packages provide for output of all numeric types. The format of output can be adjusted in many ways.

Remarks

Note how each time a generic package is instantiated with a numeric type. Also, there are both defaults to be set for the whole instance, and also ways to override `Width`, say, when calling `Put` with this parameter.

Examples

Print integers, generously using space

Instances of `Integer_IO` have a settings variable `Default_Width` which the number of characters that each output number will take.

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

procedure Print_Integer is
    subtype Count is Integer range -1_000_000 .. 1_000_000;

    package Count_IO is new Integer_IO (Count);
    X : Count;
begin
    Count_IO.Default_Width := 12;

    X := Count'First;
    while X < Count'Last loop
        Count_IO.Put (X);
        Count_IO.Put (X + 1);
        New_Line;

        X := X + 500_000;
    end loop;
end Print_Integer;
```

Result

```
    -1000000
     -500000
           0
      500000
```

Print Integers, Using Base 16 (Hexadecimal)

A settings variable `Default_Base` is set on the instance of `Ada.Text_IO.Integer_IO`; also, `Default_Width` is set so that output cannot have leading space.

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

procedure Print_Hex is
    subtype Count is Integer range -1_000_000 .. 1_000_000;

    package Count_IO is new Integer_IO (Count);
    X : Count;
begin
    Count_IO.Default_Width := 1;
    Count_IO.Default_Base := 16;

    X := Count'First;
    while X < Count'Last loop
        Count_IO.Put (X);
        New_Line;

        X := X + 500_000;
    end loop;
end Print_Hex;
```

Result

```
-16#F4240#
-16#7A120#
16#0#
16#7A120#
```

Print Decimal Fixed Point Numbers, aka Money

`Ada.Text_IO.Editing` offers formatting decimal fixed point values using “picture strings”. These describe output using “magical” characters for separators, currency signs, etc.

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

procedure Print_Value is

    Max_Count      : constant := 1_000_000;

    type Fruit is (Banana, Orange, Pear);
    subtype Count is Integer range -Max_Count .. +Max_Count;

    type Money is delta 0.001 digits 10;

    package Fruit_IO is new Enumeration_IO (Fruit);
    package Money_IO is new Editing.Decimal_Output
      (Money,
       Default_Currency => "CHF",
       Default_Separator => ''');
```

```
    Inventory : constant array (Fruit) of Count :=
      (Banana => +27_420,
       Orange => +140_600,
       Pear   => -10_000);

    Price_List : constant array (Fruit) of Money :=
      (Banana => 0.07,
       Orange => 0.085,
       Pear   => 0.21);

    Format : constant Editing.Picture :=
      Editing.To_Picture ("<###BZ_ZZZ_ZZ9.99>");
begin
    Fruit_IO.Default_Width := 12;

    for F in Inventory'Range loop
        Fruit_IO.Put (F);
        Put          (" | ");
        Money_IO.Put (Item => Inventory (F) * Price_List (F),
                      Pic => Format);
        New_Line;
    end loop;
end Print_Value;
```

Result

```
BANANA       |  CHF     1'919.40
ORANGE       |  CHF    11'951.00
PEAR         | (CHF     2'100.00)
```

Print Multiple Items On One Line

Combine the instances of the `_IO` packages, use the right one with its numeric type.

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

procedure Print_Inventory is
    type Fruit is (Banana, Orange, Pear);
    subtype Count is Integer range -1_000_000 .. 1_000_000;

    package Fruit_IO is new Enumeration_IO (Fruit);
    package Count_IO is new Integer_IO (Count);

    Inventory : constant array (Fruit) of Count :=
      (Banana => 27_420,
       Orange => 140_600,
       Pear   => -10_000);

begin
    Fruit_IO.Default_Width := 12;

    for F in Inventory'Range loop
        Fruit_IO.Put (F);
        Put          (" | ");
        Count_IO.Put (Inventory (F));
```

```
        New_Line;
    end loop;
end Print_Inventory;
```

Result

```
BANANA        |    27420
ORANGE        |   140600
PEAR          |   -10000
```

Read Outputting numbers online: https://riptutorial.com/ada/topic/8940/outputting-numbers

Chapter 8: package Ada.Text_IO

Introduction

Package `Ada.Text_IO` is used for putting text or getting text from files or console.

Examples

Put_Line

Prints out string with a newline.

```
with Ada.Text_IO;

procedure Put_Text is
    use Ada.Text_IO;
    S : String := "Hello";
begin
    Put_Line ("Hello");
    Put_Line (Standard_Output, "Hello");
    Put_Line (Standard_Error, "Hello error");
    Put_Line (S & " World");
end;
```

Result

```
Hello
Hello
Hello error
Hello World
```

Read package Ada.Text_IO online: https://riptutorial.com/ada/topic/8839/package-ada-text-io

Chapter 9: Packages

Syntax

- with Package_Name_To_Include;
- package New_Package_Name renames Package_To_Rename;
- use Package_Name;
- package Parent_Name.Child_Name is

Remarks

Package provides:

- Code encapsulation
- Separate compilation
- Hide procedures, functions, operators on private types

Similarities or analogous in other languages:

- C++ namespace
- Java packages

Examples

More on Packages

In the Hello World, you were introduced to the package `Ada.Text_IO`, and how to use it in order to perform I/O operations within your program. Packages can be further manipulated to do many different things.

Renaming: To rename a package, you use the keyword `renames` in a package declaration, as such:

```
package IO renames Ada.Text_IO;
```

Now, with the new name, you can use the same dotted notation for functions like `Put_Line` (i.e. `IO.Put_Line`), or you can just `use` it with `use IO`. Of course, saying `use IO` or `IO.Put_Line` will use the functions from the package `Ada.Text_IO`.

Visibility & Isolation: In the *Hello World* example we included the Ada.Text_IO package using a `with` clause. But we also declared that we wanted to `use Ada.Text_IO` on the same line. The `use Ada.Text_IO` declaration could have been moved into the declarative part of the procedure:

```
with Ada.Text_IO;
```

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

In this version, the procedures, functions, and types of `Ada.Text_IO` are directly available inside the procedure. Outside the block in which use `Ada.Text_IO` is declared, we would have to use the dotted notation to invoke, for example:

```
with Ada.Text_IO;

procedure hello_world is
begin
   Ada.Text_IO.Put ("Hello, ");        --  The Put function is not directly visible here
   declare
      use Ada.Text_IO;
   begin
      Put_Line ("world!");             --  But here Put_Line is, so no Ada.Text_IO. is needed
   end;
end hello_world;
```

This enables us to isolate the use … declarations to where they are necessary.

Parent-Child Relationship

As a way of subdividing Ada programs, packages may have so-called children. These can be packages, too. A child package has a special privilege: it can see the declarations in the parent package's private part. One typical use of this special visibility is when forming a hierarchy of derived types in object oriented programming.

```
package Orders is
   type Fruit is (Banana, Orange, Pear);
   type Money is delta 0.01 digits 6;

   type Bill is tagged private;

   procedure Add
     (Slip   : in out Bill;
      Kind   : in     Fruit;
      Amount : in     Natural);

   function How_Much (Slip : Bill) return Money;

   procedure Pay
     (Ordered : in out Bill;
      Giving  : in     Money);

private
   type Bill is tagged record
      --  ...
      Sum : Money := 0.0;
   end record;
end Orders;
```

Any Ada unit that is headed by `with Orders;` can declare objects of type `Bill` and then call operations `Add`, `How_Much`, and `Pay`. It does not, however, see the components of `Bill`, nor even of `Orders.Bill`, since the full type definition is hidden in the **private** part of `Orders`. The full definition is not hidden form child packages, though. This visibility facilitates type extension if needed. If a type is declared in the child package as derived from `Bill`, then this inheriting type can manipulate `Bill` 's components directly.

```
package Orders.From_Home is
   type Address is new String (1 .. 120);

   type Ordered_By_Phone is new Bill with private;

   procedure Deliver
     (Ordered : in out Ordered_By_Phone;
      Place   : in     Address);

private
   type Ordered_By_Phone is new Bill with
      record
         Delivered : Boolean := False;
         To        : Address;
      end record;
end Orders.From_Home;
```

`Orders.From_Home` is a child package of `Orders`. Type `Ordered_By_Phone` is derived from `Bill` and includes its record component `Sum`.

Read Packages online: https://riptutorial.com/ada/topic/7322/packages

Chapter 10: Parameterized Types

Introduction

All composite types other than arrays can have discriminants, which are components with special properties. Discriminants can be of a discrete type or an access type. In the latter case the access type can be a named access type or it can be anonymous. A discriminant of an anonymous access type is called an access discriminant by analogy with an access parameter.

Examples

Discriminated record types

In the case of a discriminated record type, some of the components are known as discriminants and the remaining components can depend upon these. The discriminants can be thought of as parameterizing the type and the syntax reveals this analogy. In this example we create a type that provide a square matrix with a positive as parameter :

```
type Square(X: Positive) is
    record
        S: Matrix(1 .. X, 1 .. X);
    end record;
```

Then to create a square of 3 by 3, just call yout type Square like this :

```
Sq: Square(3);
```

Variant Record Structures

A discriminant of a record type may influence the structure of objects. A choice of components may exists in an object according as a discriminant had had a particular value when the object was created. To support this variation, a record type's definition includes a distinction by cases that depends on the discriminant:

```
type Fruit is (Banana, Orange, Pear);

type Basket (Kind : Fruit) is
   record
      case Kind is
         when Banana =>
            Bunch_Size      : Positive;
            Bunches_Per_Box : Natural;
         when Pear | Orange =>
            Fruits_Per_Box  : Natural;
      end case;
   end record;
```

Then to create a box for bananas,

```
Box : Basket (Banana);
```

The `Box` object now has two record components in addition to its discriminant, `Kind`, namely `Bunch_Size` and `Bunches_Per_Box`.

Read Parameterized Types online: https://riptutorial.com/ada/topic/9311/parameterized-types

Chapter 11: Scalar Types

Introduction

In Ada's hierarchy of types, elementary types have sets of logically indivisible values. Among these types are the access types (pointer types) and the scalar types. The scalar types can be categorised as *enumeration*, *character*, and *numeric*. These types form the subject of this topic. In addition to the sets of values, types have set of operations applicable to the respective scalars, such as *successor*, or `"+"`.

Syntax

1. **type** … **is** …

Parameters

Ellipsis	What
… (1)	to receive the type's name
… (2)	to receive the type's characteristics using keywords: **delta**, **digits**, **range**

Remarks

All scalar type definitions except enumeration and modular integers may include a **range** constraint.

A range constraint specifies a lower bound and an upper bound of the set of values to include in the type. For fixed point types, specifying a range is mandatory: values of these types will be understood to be multiples of a small fraction of two, for example, of $1/2^5$. The smaller these fractions become, the more precise the representation, at the cost of range that can be represented using the bits available.

Further aspects of type definitions may be given, such as a desired `Size` in bits and other representational items. Ada 2012 adds aspects of contract based programming like `Static_Predicate`.

Examples

Enumeration

```
type Fruit is (Banana, Orange, Pear);
```

```
Choice : Fruit := Banana;
```

A character type is an enumeration that includes a character literal:

```
type Roman_Numeral is
    ('I', 'V', 'X', 'L', 'C', 'D', 'M', Unknown);`
```

Singed Integer

```
type Grade is range 0 .. 15;

B   : Grade := 11;
C   : Grade := 8;
Avg : Grade := (B + C) / 2;  -- Avg = 9
```

Modular Integer

These are the “bit fiddling” types. They have logical operators, too, such as **xor**, and they “wrap around” at the upper bound, to 0 again.

```
type Bits is mod 2**24;

L : Bits := 2#00001000_01010000_11001100# or 7;
```

Floating Point

A floating point type is characterised by its (decimal) digits which state the minimal precision requested.

```
type Distance is digits 8;

Earth : Distance := 40_075.017;
```

Fixed Point (Ordinary)

A fixed point type definition specifies a *delta*, and a range. Together, they describe how precisely real values should be approximated as they are represented by powers of two, not using floating point hardware.

```
Shoe_Ounce : constant := 2.54 / 64.0;
type Thickness is delta Shoe_Ounce range 0.00 .. 1.00;

Strop : Thickness := 0.1;  -- could actually be 0.09375
```

Fixed Point (Decimal)

Decimal fixed point types are typically used in accounting. They are characterised by both a *delta* and a number of decimal digits. Their arithmetical operations reflect the rules of accounting.

```
type Money is delta 0.001 digits 10;

Oil_Price : Money := 56.402;
Loss      : Money := 0.002 / 3; -- is 0.000
```

Read Scalar Types online: https://riptutorial.com/ada/topic/9297/scalar-types

Chapter 12: Task

Syntax

- task Task_Name;
- task Task_Name is Entries end;
- task body Task_Name is Declarations begin Code end;

Examples

One simple task

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

procedure Main is
   task My_Task;
   task body My_Task is
   begin
      Put_Line ("Hello from My_Task");
   end;
begin
   Put_Line ("Hello from Main");
end;
```

Result

The order of `Put_Line` can vary.

```
Hello from My_Task
Hello from Main
```

One simple task and one loop

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

procedure Main is
   task My_Task;
   task body My_Task is
   begin
      for I in 1 .. 4 loop
         Put_Line ("Hello from My_Task");
      end loop;
   end;
begin
   Put_Line ("Hello from Main");
end;
```

Result

The order of `Put_Line` can vary.

```
Hello from My_Task
Hello from Main
Hello from My_Task
Hello from My_Task
Hello from My_Task
```

One simple task and two loops

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

procedure Main is
   task My_Task;
   task body My_Task is
   begin
      for I in 1 .. 4 loop
         Put_Line ("Hello from My_Task");
      end loop;
   end;
begin
   for I in 1 .. 4 loop
      Put_Line ("Hello from Main");
   end loop;
end;
```

Result

The order of `Put_Line` can vary.

```
Hello from My_Task
Hello from My_Task
Hello from Main
Hello from My_Task
Hello from Main
Hello from My_Task
Hello from Main
Hello from Main
```

Two simple task and two loops

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

procedure Main is
   task My_Task_1;
   task My_Task_2;

   task body My_Task_1 is
   begin
```

```
      for I in 1 .. 4 loop
         Put_Line ("Hello from My_Task_1");
      end loop;
   end;

   task body My_Task_2 is
   begin
      for I in 1 .. 4 loop
         Put_Line ("Hello from My_Task_2");
      end loop;
   end;
begin
   null;
end;
```

Result

The order of `Put_Line` can vary.

```
Hello from My_Task_1
Hello from My_Task_1
Hello from My_Task_2
Hello from My_Task_1
Hello from My_Task_2
Hello from My_Task_1
Hello from My_Task_2
Hello from My_Task_2
```

A task that increment a number after entry

The user can call `Incrementor.Increment` `K` number of times by pressing a key within `'0' .. '9'` and it's possible to call `Incrementor.Increment` faster than the `task Incrementor` can increment `I`.

```
with Ada.Text_IO;
with Ada.Integer_Text_IO;

procedure Main is
   use Ada.Text_IO;
   task Incrementor is
      entry Increment;
   end;
   task body Incrementor is
      use Ada.Integer_Text_IO;
      I : Integer := 0;
   begin
      loop
         accept Increment;
         I := I + 1;
         Put (I, 0);
         delay 0.1;
      end loop;
   end;
   K : Character;
begin
   loop
```

```
      Get_Immediate (K);
      if K in '0' .. '9' then
         for I in 1 .. Natural'Value (K & "") loop
            Incrementor.Increment;
         end loop;
      end if;
   end loop;
end;
```

Interrupt Handling

Interrupts are handled by a protected procedure with no parameters.

```
------------------------------------------------------------------
-- Interrupt Counting Package --
------------------------------------------------------------------
with Ada.Interrupts.Names; use Ada.Interrupts.Names;

package Ctl_C_Handling is

   protected CTL_C_Handler is
      procedure Handle_Int with
        Interrupt_Handler,
        Attach_Handler => SIGINT;
      entry Wait_For_Int;
   private
      Pending_Int_Count : Natural := 0;
   end Ctl_C_Handler;

   task CTL_Reporter is
      entry Stop;
   end CTL_Reporter;

end Ctl_C_Handling;
```

The package body shows how the protected procedure works. In this case a boolean is not used in the protected object because interrupts arrive faster than they are handled. The task CTL_Reporter handles the received interrupts.

```
with Ada.Text_IO; use Ada.Text_IO;
with Ctl_C_Handling; use CTL_C_Handling;
with Ada.Calendar; use Ada.Calendar;

package body Ctl_C_Handling is

   -------------------
   -- CTL_C_Handler --
   -------------------

   protected body CTL_C_Handler is

      ----------------
      -- Handle_Int --
      ----------------

      procedure Handle_Int is
      begin
```

```
      Pending_Int_Count := Pending_Int_Count + 1;
   end Handle_Int;

   ------------------
   -- Wait_For_Int --
   ------------------

   entry Wait_For_Int when Pending_Int_Count > 0 is
   begin
      Pending_Int_Count := Pending_Int_Count - 1;
   end Wait_For_Int;

end CTL_C_Handler;

------------------
-- CTL_Reporter --
------------------

task body CTL_Reporter is
   type Second_Bin is mod 10;
   type History is array(Second_Bin) of Natural;

   ---------------------
   -- Display_History --
   ---------------------

   procedure Display_History(Item : History) is
      Sum : Natural := 0;
   begin
      for I in Item'Range loop
         Put_Line("Second: " & Second_Bin'Image(I) & " :" & Natural'Image(Item(I)));
         Sum := Sum + Item(I);
      end loop;
      Put_Line("Total count: " & Natural'Image(Sum));
      New_Line(2);
   end Display_History;

   One_Second_Count : Natural := 0;
   Next_Slot : Second_Bin := 0;
   Next_Second : Time := Clock + 1.0;
   Ten_Second_History : History := (Others => 0);

begin
   loop
      Select
         Accept Stop;
         exit;
      else
         select
            CTL_C_Handler.Wait_For_Int;
            One_Second_Count := One_Second_Count + 1;
         or
            delay until Next_Second;
            Next_Second := Next_Second + 1.0;
            Ten_Second_History(Next_Slot) := One_Second_Count;
            Display_History(Ten_Second_History);
            Next_Slot := Next_Slot + 1;
            One_Second_Count := 0;
         end Select;
      end Select;
   end loop;
```

```
   end CTL_Reporter;
end Ctl_C_Handling;
```

An example main program to exercise this package is:

```
-----------------------------------------------------------------
-- Ada2012 Interrupt Handler Example --
-----------------------------------------------------------------
with Ada.Text_IO; use Ada.Text_IO;
with Ctl_C_Handling; use CTL_C_Handling;

procedure Interrupt01 is
begin
   Delay 40.0;
   CTL_Reporter.Stop;
   Put_Line("Program ended.");
end Interrupt01;
```

Read Task online: https://riptutorial.com/ada/topic/7345/task

Credits

S. No	Chapters	Contributors
1	Getting started with ada	B98, Community, Jacob Sparre Andersen, Jaken Herman, Jossi , manuBriot, trashgod
2	Attribute Image	B98, Jacob Sparre Andersen, Jossi
3	Enumeration	B98, Jossi, Simon Wright
4	Files and I/O streams	B98, Jossi
5	Genericity in Ada	Aznhar, B98
6	Implementing the producer-consumer pattern	Jacob Sparre Andersen, Jim Rogers, Jossi
7	Outputting numbers	B98
8	package Ada.Text_IO	Jossi
9	Packages	B98, Jaken Herman, Jossi
10	Parameterized Types	Aznhar, B98
11	Scalar Types	B98
12	Task	Jim Rogers, Jossi