Intro

Course based on the e-book

Object-oriented software in Ada 95. Michael A Smith in 2000

Published by: McGraw-Hill.
ISBN X-XXXXXX-XXX-X


```
e-mail         mas@brighton.ac.uk
WWW address    http://www.it.brighton.ac.uk/~mas
```


WWW pages for book are at:

```
http://www.brighton.ac.uk/ada95
```

On the World Wide Web page there are links to exercises from the book at:

```
http://www.brighton.ac.uk/ada95/exercise/home.html
```

On the World Wide Web page there are links to programs shown in the lectures:

```
http://www.brighton.ac.uk/ada95/programs/home.html
```

FTP link for programs:

```
ftp://ftp.brighton.ac.uk/pub/mas/ada95
```

File server link for programs:

```
p:\coursewk\mas\cs122
```

Mike Smith / University of Brighton / England

Compiling an Ada 95 program using GNAT

By hand

```
gnatchop file.ada

gnatmake main
```

Remember

- File name must equal unit name
- Each unit must be in a separate file

What's new in Ada 95

- Relaxation of some Ada 83 rules
  (Order of declarations, limited records, overloading "=",
  -1 is now of type Integer, 'Image attribute on Float)
- Inheritance (Tagged types)
- Child packages & hierarchical libraries
- Large standard library
  (strings, command line, numerics)
- Extended I/O library functions
- New Types Decimal, Modular
  (and new generic formal parameters)
- More attributes (e.g. 'class)
- Improved exception handling
- Access type extensions
- Interfacing to other languages
- Protected Types (Tasking)

Problems: Ada 83 programs

- Character
  Now has 256 positions

- New reserved words:
  `aliased protected requeue`
  `tagged until`

- Unconstrained Generic types:

  In Ada 83:
  It is legal to instantiate package with type `String` this is of course wrong.

```
with IO_Exceptions
generic
   type E_Type is Private
package Sequential_IO is ...
```

  However Ada 95 prevents this so spec is now:

```
with IO_Exceptions
generic
   type E_Type (<>) is Private
package Sequential_IO is ...
```

Programming languages

In assember

```
LDA  AMOUNT_OF_OF_APPLES  ; Load into the accumulator # pounds
MLT  PRICE_PER_POUND      ; Multiply by cost per pound of apples
STA  COST_OF_APPLES       ; Save result
```

In Fortran

```
      COST = PRICE * AMOUNT
```

Ada95

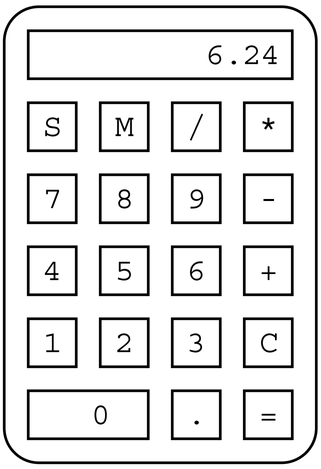
For example, in the programming language Ada95, to print the result of multiplying 10 by 5 the following programming language statement is written:

```
put( 10 * 5 );
```

A small problem

A local orchard sells some of its rare variety apples in its local farm shop. However, the farm shop has no electric power and hence uses a set of scales which just give the weight of the purchased product. A customer buying apples, fills a bag full of apples and takes the apples to the shop assistant who weighs the apples to determine their weight in kilograms and then multiples the weight by the price per kilogram.

If the shop assistant is good at mental arithmetic they can perform the calculation in their head, or if mental arithmetic is not their strong point they can use an alternative means of determining the cost of the apples.

Pocket calculator	**Step**	**Steps performed**
	1	Enter the cost of a kilo of apples: `C 1 . 2 0`
	2	Enter the operation to be performed: `*`
	3	Enter the number of kilos to be bought: `5 . 2`
	4	Enter calculate `=`

Solution in Ada95

Step	Line	Ada95 statements
	1	`Price_per_kilo   : Float;`
	2	`Kilos_of_apples : Float;`
	3	`Cost            : Float;`
1	4	`Price_per_kilo  := 1.20;`
2	5	`Kilos_of_apples := 5.2;`
3	6	`Cost:= Price_per_kilo*Kilos_of_apples;`
4	7	`Put( Cost );`
	8	`New_Line;`

Line	Description
1	Allocates a memory location called `Price_per_kilo` that is used to store the price per kilogram of apples. This memory location is of type `Float` and can hold any number that has decimal places.
2—3	Allocates memory locations: `Kilos_of_apples` and `Cost`.
4	Sets the contents of the memory location `Price_per_kilo` to 1.20. The = can be read as 'is assigned the value'.
5	Assign 5.2 to memory location `Kilos_of_apples`.
6	Sets the contents of the memory location `Cost` to the contents of the memory location `Price_per_kilo` multiplied by the contents of the memory location `Kilos_of_apples`.
7	Writes the contents of the memory location `Cost` onto the computer screen.
8	Starts a new line on the computer screen.

Step	Line	Ada95 statements
	1	`Price_per_kilo   : Float;`
	2	`Kilos_of_apples : Float;`
	3	`Cost            : Float;`
1	4	`Price_per_kilo  := 1.20;`
2	5	`Kilos_of_apples := 5.2;`
3	6	`Cost:= Price_per_kilo*Kilos_of_apples;`
4	7	`Put( Cost );`
	8	`New_Line;`

Ada95 statements	price_ per kilo	kilos_ of apples	cost
`price_per_kilo  : Float;` `kilos_of_apples : Float;` `cost            : Float;`	U	U	U
`price_per_kilo := 1.20;`	1.20	U	U
`kilos_of_apples := 5.2;`	1.20	5.2	U
`cost := price_per_kilo * kilos of apples;`	1.20	5.2	6.24
`put( cost );`	1.20	5.2	6.24

Line — **Ada95 statements**

```
1    declare
2      Price_Per_Kilo : Float;   --Price of apples
3      Kilos_Of_Apples: Float;   --Apples required
4      Cost           : Float;   --Cost of apples
5    begin
6      Price_Per_Kilo := 1.20;
7      Kilos_of_apples := 5.2;
8      Cost := Price_per_kilo * Kilos_of_apples;

9      Put( "Cost of apples per kilo  : " );
10     Put( Price_per_kilo );
11     New_Line;

12     Put( "Kilos of apples required K " );
13     Put( Kilos_of_apples );
14     New_Line;

15     Put( "Cost of apples           £ " );
16     Put( Cost );
17     New_Line;
18   end
```

```
declare
  Price_Per_Kilo : Float;          --Price of apples
  Kilos_Of_Apples: Float;          --Apples required
  Cost           : Float;          --Cost of apples
begin
  Price_Per_Kilo := 1.20;

  Put( "Cost of apples per kilo  : " );
  Put( Price_Per_Kilo );
  New_Line;

  Put( "Kilo's  Cost" );
  New_Line;
```

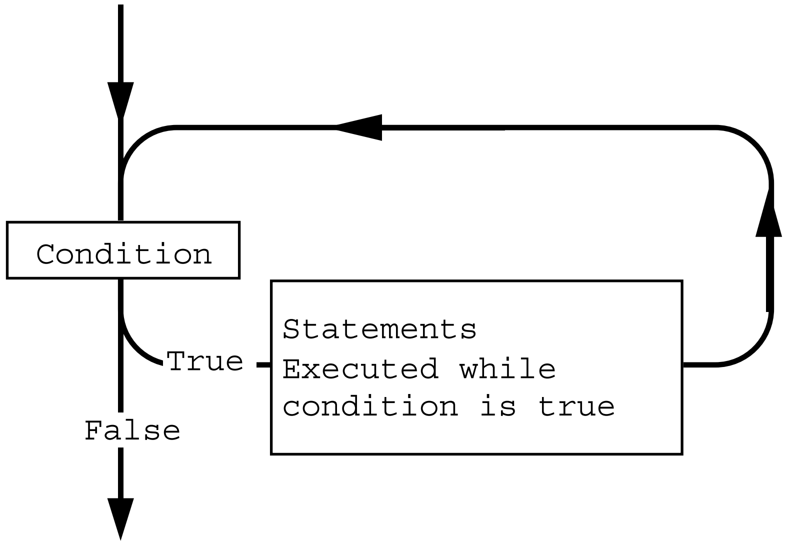
```
    kilos_of_apples           := 0.1;
```

```
    cost := price_per_kilo * kilos_of_apples;
    put( kilos_of_apples );
    put( "       " );
    put( cost );
    new_line;
```

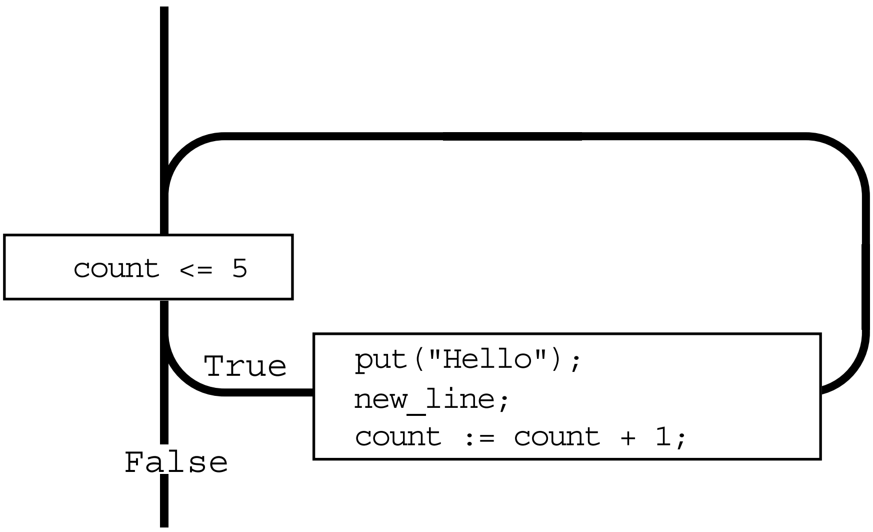
```
    kilos_of_apples           := 0.2;
```

```
    cost := price_per_kilo * kilos_of_apples;
    put( kilos_of_apples );
    put( "       " );
    put( cost );
    new_line;
  end;
```

while



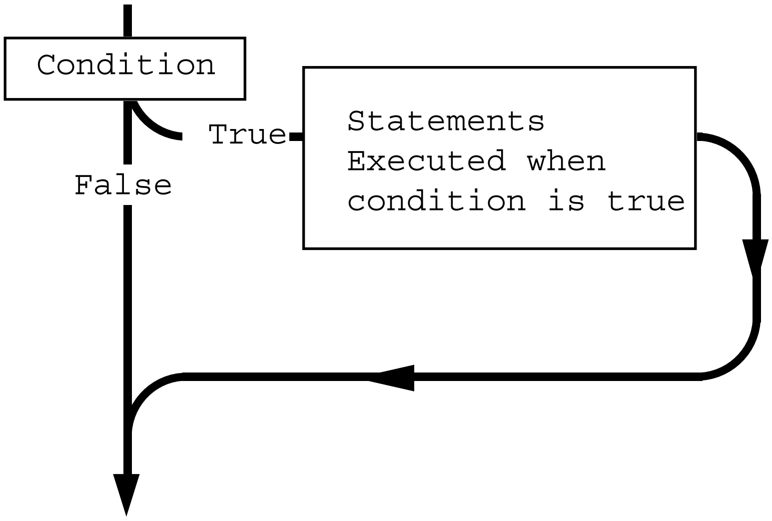

```
declare
  count : Integer;
begin
  count := 1;                  --Set count to 1

  while count <= 5 loop        --While count less than or equal 5
    put( "Hello" );            -- Print Hello
    new_line;
    count := count + 1;        -- Add 1 to count
  end loop;
end;
```



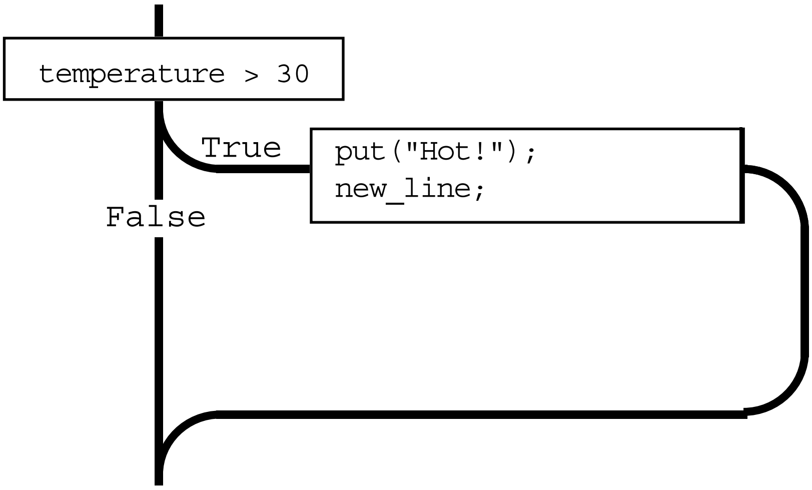
```
declare
  Price_Per_Kilo : Float;         --Price of apples
  Kilos_Of_Apples: Float;         --Apples required
  Cost           : Float;         --Cost of apples
begin
  Price_Per_Kilo := 1.20;

  Put( "Cost of apples per kilo  : " );
  Put( Price_Per_Kilo ); New_Line;

  Put( "Kilo's  Cost" ); New_Line;

  Kilos_Of_Apples := 0.1;

  while Kilos_Of_Apples <= 10.0 loop      --While lines to print
    Cost := Price_Per_Kilo * Kilos_Of_Apples; --Calculate cost
    Put( Kilos_Of_Apples );                   --Print results
    Put( "      " );
    Put( Cost );
    New_Line;
    Kilos_Of_Apples := Kilos_Of_Apples + 0.1; --Next value
  end loop;
end;
```

```
Cost of apples per kilo  : 1.20
Kilo's  Cost
0.1      0.12
0.2      0.24
0.3      0.36
0.4      0.48
0.5      0.60
0.6      0.72
0.7      0.84
0.8      0.96
0.9      1.08
1.0      1.12
1.1      1.32
1.2      1.44
1.3      1.56
...
9.9      11.88
10.0     12.00
```

Selection



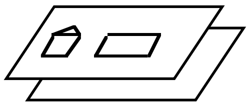
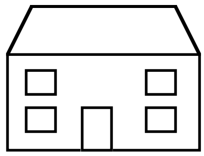

```
declare
  Temperature : Integer;
begin
  Temperature := 30;
  if  Temperature > 30 then      --If temperature greater than 30
    Put( "Hot!" );               --Say its hot
    New_Line;
  end if;
end;
```



```
declare
  Price_Per_Kilo   : Float := 1.20;
  Kilos_Of_Apples : Float := 0.0;
  Cost             : Float;
  Lines_Output     : Integer := 0;
begin
  Put( "Cost of apples per kilo  : " );
  Put( Price_Per_Kilo ); New_Line;
  Put( "Kilo's  Cost" ); New_Line;

  while Kilos_Of_Apples <= 10.0 loop       --While lines to print
    Cost := Price_Per_Kilo * Kilos_Of_Apples;  --Calculate cost
    Put( Kilos_Of_Apples );                    --Print results
    Put( "       " );
    Put( Cost );
    New_Line;

    Kilos_Of_Apples := Kilos_Of_Apples + 0.1; --Next value
    Lines_Output    := Lines_Output + 1;      --Add 1

    if Lines_Output >= 5 then                 --If printed group
      New_Line;                               -- Print line
      Lines_Output := 0;                      -- Reset count
    end if;
  end loop;
end;
```

```
Cost of apples per kilo  : 1.20
Kilo's  Cost
0.1      0.12
0.2      0.24
0.3      0.36
0.4      0.48
0.5      0.60

0.6      0.72
0.7      0.84
0.8      0.96
0.9      1.08
1.0      1.12

1.1      1.32
1.2      1.44
1.3      1.56
...

9.9      11.88
10.0     12.00
```

Problem Solving

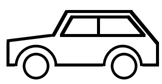
Architect's plan (model)	Finished house
	

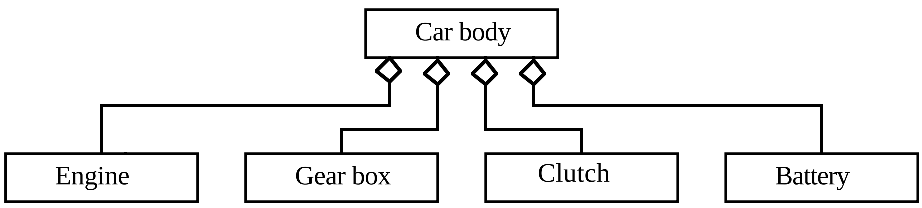
Objects

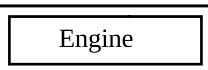
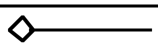
Object	Responsabilities
Telephone	● Establish contact with another phone point. ● Convert sound to / from electrical signals.
Computer	● Execute programs. ● Provide a tcp/ip connection to the internet.
Car	● Move ● Go faster / Slower. ● Turn left / right ● Stop

Container

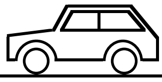
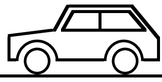
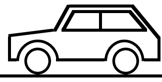
- The shell or body of the car.
- The engine.
- The gearbox.
- The clutch.
- The battery that provides electric power.



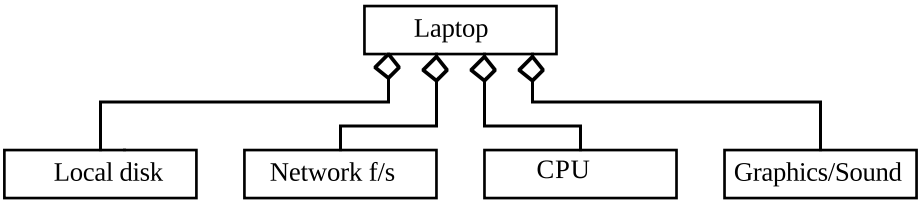


	Denotes an object. In this specific case the car engine.
	Denotes a relationship. For example, the engine is *part of* [contained in] the car shell.

Class

Carol's red car	Mike's silver car	Paul's blue car
		

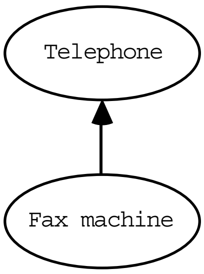
Class Object



Inheritance

Object	Responsabilities
Telephone	• Establish contact with another phone point. • Convert sound to / from electrical signals.
Fax machine	• Establish contact with another phone point. • Convert sound to / from electrical signals. • Convert images to / from electrical signals.
Computer	• Execute programs. • Provide a tcp/ip connection to the internet.

Inheritance diagram



Responsabilities:

Establish contact with another phone point.
Convert sound to / from electrical signals.

All the responsabilities of a telephone plus
Convert images to / from electrical signals.

Object, Method, Message

Class

Objects that share the same responsabilities and state information belong to the same class

Message

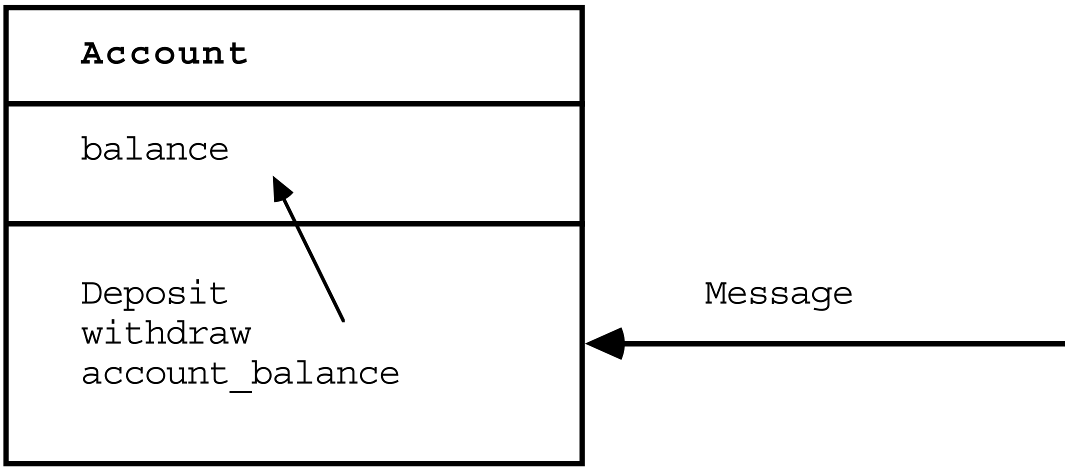
Invokes a method in an object a result may be returned.

Method

Inspects or mutates the object

Object

An instance of a class

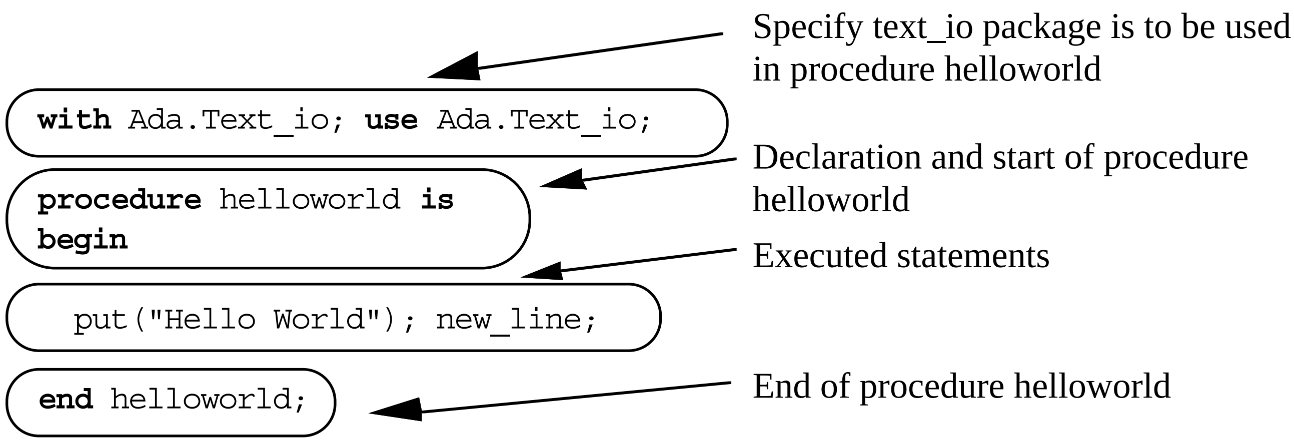


The Ada language

```
with Ada.Text_io; use Ada.Text_io;

procedure helloworld is
begin
  put("Hello World"); new_line;
end helloworld;
```

```
Hello World
```

Components of an Ada program



A countdown

```
with Ada.Text_io; use Ada.Text_io; -- Use Ada.Text_io
procedure main is
  count : Integer;                 -- Declaration of count
begin
  count := 10;                     -- Set to 10
  while count > 0 loop             -- loop while greater than 0
    if count = 3 then              -- If 3 print Ignition
      put("Ignition"); new_line;
    end if;
    put( Integer'Image( count ) ); -- Print current count
    new_line;
    count := count - 1;            -- Decrement by 1 count
    delay 1.0;                     -- Wait 1 second
  end loop;
  put("Blast off"); new_line;      -- Print Blast off

end main;
```

```
10
 9
 8
 7
 6
 5
 4
Ignition
 3
 2
 1
Blast off
```

Looping construct: while

```
while Count > 0 loop                    --loop while greater than 0

    -- Repeated statements

end loop;
```

Selection construct: if

```
    if Count = 3 then                   --If 3 print Ignition
      Put("Ignition"); New_Line;
    end if;
```

```
    if Count = 3 then
      Put("count is 3"); New_Line;
    else
      Put("count is not 3"); New_Line;
    end if;
```

```
    if Count = 3 then
      Put("Count is 3"); New_Line;
    else
      if Count = 4 then
        Put("Count is 4"); New_Line;
      else
        Put("Count is not 3 or 4"); New_Line;
      end if;
    end if;
```

```
  if Count = 3 then
    Put("Count is 3"); New_Line;
  elsif Count = 4 then
    Put("Count is 4"); New_Line;
  else
    Put("Count is not 3 or 4"); New_Line;
  end if;
```

Looping construct: `for`

```
for Count in 1 .. 10  loop           --count declared here
  Put( Integer'Image( Count ) );
end loop;
New_Line;
```

```
1 2 3 4 5 6 7 8 9 10
```

```
for Count in reverse 1 .. 10  loop
  Put( Integer'Image( Count ) );
end loop;
New_Line;
```

```
10 9 8 7 6 5 4 3 2 1
```

Looping constructs: `for` and `while`

```
with Ada.Text_Io;
use  Ada.Text_Io;
procedure Main is
  Count    : Integer;                  --count as Integer object
  Count_To : constant Integer := 10; --integer constant
begin
  Count := 1;
  while Count <= Count_To loop        --While loop
    Put( Integer'Image( Count ) );
    Count := Count + 1;
  end loop;
  New_Line;

  for Count in 1 .. Count_To loop     --count declared here
    Put( Integer'Image( Count ) );
  end loop;
  New_Line;

end Main;
```

```
 1 2 3 4 5 6 7 8 9 10
 1 2 3 4 5 6 7 8 9 10
```

Looping construct: Loop and exit

```
with Ada.Text_Io;
use  Ada.Text_Io;
procedure Main is
  Count    : Integer;                --count as Integer object
  Count_To : constant Integer := 10; --integer constant
begin
  Count := 1;
  loop
    Put( Integer'Image( Count ) );
    exit when Count = Count_To;      --Exit loop when ...
    Count := Count + 1;
  end loop;
  New_Line;

end Main;
```

Selection construct: case

```
    case Count is
      when 3      => Put("Count is 3"); New_Line;
      when 4      => Put("Count is 4"); New_Line;
      when others => Put("Count is not 3 or 4"); New_Line;
    end case;
```

```
  Ch := 'a';
  case Ch is
    when '0' | '1' | '2' | '3' | '4' |
         '5' | '6' | '7' | '8' | '9'   =>
      Put("Character is a digit");
    when 'A' .. 'Z' =>
      Put("Character is upper case English letter");
    when 'a' .. 'z' =>
      Put("Character is lower case English letter");
    when others     =>
      Put("Not an English letter or digit");
  end case;
  New_Line;
```

```
Character is lower case English letter
```

Input and output

```
put("hello");
```

```
put('h'); put('e'); put('l'); put('l'); put('o');
```

Program: cat

```
with Ada.Text_Io;
use  Ada.Text_Io;
procedure Simple_Cat is
Ch   : Character;                --Current character
begin
  while not End_Of_File loop     --For each Line
    while not End_Of_Line loop   --For each character
      Get(Ch); Put(Ch);          --Read / Write character
    end loop;
    Skip_Line; New_Line;         --Next line / new line
  end loop;
end Simple_Cat;
```

Function/Procedure	**Effect**
`end_of_file`	Delivers true when the end of the file is reached, otherwise it delivers false.
`end_of_line`	Delivers true when all the characters have been read from the current input line, otherwise it delivers false. NB. This does not include the new line character
`skip_line`	Positions the input pointer at the start of the next line. Any information on the current line is skipped.
`new_line`	Write the new line character to the output stream NB. On some systems new line is represented by two characters when output.

Command line arguments

The unix program `echo`

```
with Ada.Text_Io, Ada.Command_Line;
use  Ada.Text_Io, Ada.Command_Line;
procedure Echo is
begin
  for I in 1 .. Argument_Count loop  --For each argument
    Put( Argument(I) );              --  Print it
    if I /= Argument_Count then      --  If not last
      Put(" ");                      --    Print separator
    end if;
  end loop;
  New_Line;
end Echo;
```

The unix program `cat`

```
with Ada.Text_Io, Ada.Command_Line;
use  Ada.Text_Io, Ada.Command_Line;
procedure Cat is
  Fd   : Ada.Text_Io.File_Type;          --File descriptor
  Ch   : Character;                      --Current character
begin
  if Argument_Count >= 1 then
    for I in 1 .. Argument_Count loop    --Repeat for each file
      Open( File=>Fd, Mode=>In_File,     --Open file
            Name=>Argument(I) );
      while not End_Of_File(Fd) loop     --For each Line
        while not End_Of_Line(Fd) loop   --For each character
          Get(Fd,Ch); Put(Ch);           --Read / Write character
        end loop;
        Skip_Line(Fd); New_Line;         --Next line / new line
      end loop;
      Close(Fd);                         --Close file
    end loop;
  else
    Put("Usage: cat file1 ... "); New_Line;
  end if;
end Cat;
```

Attributes: 'First and 'Last

```
with Ada.Text_Io;
use  Ada.Text_Io;
procedure Main is
begin
  Put("Smallest Integer ");
  Put( Integer'Image( Integer'First ) ); New_Line;
  Put("Largest  Integer ");
  Put( Integer'Image( Integer'Last ) ); New_Line;
  Put("Integer (bits)   ");
  Put( Integer'Image( Integer'Size ) ); New_Line;
end Main;
```

Machine using a 32 bit word size	Machine using a 64 bit word size
Smallest integer -32768 Largest integer 32767 Integer (bits) 16	Smallest integer -2147483648 Largest integer 2147483647 Integer (bits) 32

```
with Ada.Text_Io;
use  Ada.Text_Io;
procedure Main is
begin
  Put("Smallest Float ");
  Put( Float'Image( Float'First ) ); New_Line;
  Put("Largest  Float ");
  Put( Float'Image( Float'Last ) ); New_Line;
  Put("Float (bits)   ");
  Put( Integer'Image( Float'Size ) ); New_Line;
  Put("Float (digits) ");
  Put( Integer'Image( Float'digits ) ); New_Line;
end Main;
```

Machine using a 32 bit word size	Machine using a 64 bit word size
Smallest Float -3.40282E+038 Largest Float 3.40282E+038 Float (bits) 32 Float (digits) 6	Smallest Float -1.79769313486232E+308 Largest Float 1.79769313486232E+308 Float (bits) 64 Float (digits) 15

Type declarations

Type declaration	An instance of T will Declare
type T **is range** 0 .. 250_000;	An object which can hold whole numbers in the range 0 .. 250_000.
type T **is digits** 8;	An object which can hold a floating point number which has a precision of 8 digits.
type T **is digits** 8 **range** 0.0 .. 10.0;	An object which can hold a floating point number which has a precision of 8 digits and can store numbers in the range 0.0 .. 10.0.

Using types: Conversion between types

```
procedure Main is
  type Apples        is range 0 .. 100;
  type French_Apples is range 0 .. 100;
  Number             : Apples;
  Number_From_France : French_Apples;
begin
  Number := 10;
  Number_From_France := French_Apples( Number );
end Main;
```

Using types: Countdown program revisited

```
with  Ada.Text_Io;
use   Ada.Text_Io;
procedure Main is
  type Count_Range is range 0 .. 10;
  Count : Count_Range := 10;            --Declaration of count
begin
  for Count in reverse Count_Range loop
    if Count = 3 then                   --If 3 print Ignition
      Put("Ignition"); New_Line;
    end if;
    Put( Count_Range'Image( Count ) ); --Print current count
    New_Line;
    Delay 1.0;                          -- Wait 1 second
  end loop;
  Put("Blast off"); New_Line;          --Print Blast off
end Main;
```

Input and Output: Packages required

```
with Ada.Text_io;
package Integer_io  is new Ada.Text_io.Integer_io( Integer );

with Ada.Text_io;
package Float_io    is new Ada.Text_io.Float_io( Float );
```

```
with Ada.Text_io, Integer_io, Float_io;
use  Ada.Text_io, Integer_io, Float_io;
procedure main is
```

```
with Ada.Text_io, Ada.Integer_Text_io, Ada.Float_Text_io;
use  Ada.Text_io, Ada.Integer_Text_io, Ada.Float_Text_io;
procedure main is
```

Using types: Type safety

```
  type Miles      is digits 8 range 0.0 .. 25_000.0;
  type Kilometres is digits 8 range 0.0 .. 50_000.0;
```

```
with Ada.Text_Io, Ada.Float_Text_Io;
use  Ada.Text_Io, Ada.Float_Text_Io;
procedure Main is
  type Miles      is digits 8 range 0.0 .. 25_000.0;
  type Kilometres is digits 8 range 0.0 .. 50_000.0;
  London_Paris       : Miles;
  Paris_Geneva       : Kilometres;
  London_Paris_Geneva: Kilometres;
begin
  London_Paris := 210.0;  --Miles
  Paris_Geneva := 420.0;  --Kilometres
  London_Paris_Geneva :=
    Kilometres( London_Paris * 1.609_344 ) + Paris_Geneva;
  Put("Distance london - paris - geneva (Kms) is " );
  Put( Float( London_Paris_Geneva ), Aft=>2, Exp=>0 );
  New_Line;
end Main;
```

Using types: Type safety

```
procedure Dec is
  type     Power_Points  is              range 0 .. 6;
  type     Room_Size     is              range 0 .. 120;
  subtype Lecture_Room  is Room_Size range 0 .. 80;
  subtype Tutorial_Room is Room_Size range 0 .. 20;

  Points_In_504   : Power_Points;          --Power outlets
  People_In_504   : Lecture_Room;          --Size lecture room
  People_In_616   : Tutorial_Room;         --Size tutorial room
begin
  Points_In_504 := 3;                      --OK
  Points_In_504 := 80;                     --Error / Warning

  People_In_504  := 15;                    --OK
  People_In_616  := People_In_504;         --OK

  People_In_504  := Points_In_504;         -- Type Mismatch

  People_In_504  := Lecture_Room( Points_In_504 );  --Force

  People_In_504  := 50;                    --OK
  People_In_616  := People_In_504;         --Constraint error
end Dec;
```

Errors in above code

Line	Problem
`Points_In_504:= 80;`	The range of values allowed for the object `Points_In_504` does not include 80. This error will usually be detected at compile-time.
`People_In_504  := Points_In_504;`	The objects on the LHS and RHS of the assignment statement are of different types and will thus produce a compile-time error.
`People_In_616  := People_In_504;`	Will cause a constraint error when executed, as the obejct People_In_504 contains 50. In this example, the error could in theory be detected at compile-time.

Subtypes Natural and Positive

```
subtype Natural  is Integer range 0 .. Integer'last;
subtype Positive is Integer range 1 .. Integer'last;
```

Subtypes

```
type    Speed_mph     is range 0 .. 25_000;
subtype Train_speed   is Speed_mph range 0 .. 130;
subtype Bus_speed     is Speed_mph range 0 .. 75;
subtype Cycling_speed is Speed_mph range 0 .. 30;
subtype Person_speed  is Speed_mph range 0 .. 15;
```

Using subtypes

```
with Ada.Text_Io;
use  Ada.Text_Io;
procedure main is
  --        Type and subtype declarations for speeds
  T0715   : Train_Speed;  --07:15 Brighton - London
  B0720   : Bus_Speed;    --07:20 Brighton - London
begin
  T0715 := 55;  --Average speed Brighton - London (Train)
  B0720 := 35;  --Average speed Brighton - London (Bus)
  if T0715 > B0720 then
    Put("The train is faster then the bus");
  else
    Put("The bus is faster then the train");
  end if;
  New_Line;
end Main;
```

Types vs. subtypes

Criteria	Types	Subtype
Instances may be mixed with	Only with instances of the same type.	Only with instances of subtypes derived from the same type.
May have a constraint	Yes	Yes

Type implementation: Base type

```
type    Speed_Mph     is range 0 .. 25_000;
```

```
type    Anonymous     is -- implementation defined
subtype Speed_Mph     is Anonymous range 0 .. 25_000;
```

```
Put("The base range of the type T2 is " );
Put( Integer(T2'Base'First) ); Put( " .." );
Put( Integer(T2'Base'Last) ); New_Line;
```

`root_integer` and `root_real`

Root type	Range / precision
`Root_Integer`	`System.Min_Int .. System.Max_Int`
`Root_Real`	`System.Max_Base_Digits`

```
type Exam_mark is new Integer range 0 .. 100;
type Exam_mark is range 0 .. 100;
```

type **Exam_Mark** is	Base type	Minimum range of root type
new Integer **range** 0 .. 100;	Root_Integer	`System.Min_Int .. System.Max_Int`
range 0 .. 100;	Implementation defined	Implementation defined but must hold 0 .. 100

When performing arithmetic with an instance of a type's base type, no range checks take place. This allows an implementor to implement the base type in the most efficient or effective way for a specific machine. However, the exception `Constraint_Error` will be generated if the resultant arithmetic evaluation leads to a wrong result. For example, the exception `Constraint_Error` is generated if an overflow is detected when performing calculations with the base type.

Using types: consequence of base type

```
with Ada.Text_Io;
use  Ada.Text_Io;
procedure Main is
  type Exam_Mark is new Integer range 0 .. 100;
  English   : Exam_Mark;          --English exam mark
  Maths     : Exam_Mark;          --Maths     "    "
  Computing : Exam_Mark;          --Computing "    "
  Average   : Exam_Mark;          --
begin
  English   := 72;
  Maths     := 68;
  Computing := 76;
  Put("Average exam mark is ");
  Average   := (English + Maths + Computing) / 3;
  Put( Exam_Mark'Image(Average) ); New_Line;
end Main;
```

Of course, for this to work the `Root_Integer` type must be sufficiently large to hold the sum of: `english+maths+computing`.

Warning

```
  type Exam_mark is range 0 .. 100;
```

May cause problems

Constrained & Unconstrained types

```
type Exam_mark is new Integer range 0 .. 100;
```

Declaration	Instance is	Commentary
`Exam_mark`	Constrained	Constrained to the range 0 .. 100.
`Exam_mark'Base`	Unconstrained	No range checks applied to assignment of this variable. An implementor may allow this to have a range greater than the base range of the root type
`Integer`	Constrained	Constrained to the base range of `Integer`. Which is implementation dependant.
`Integer'Base`	Unconstrained	No range checks apply, may have a range greater than `Integer`.

Enumerations

Without: Enumerations

```
with Ada.Text_Io;
use  Ada.Text_Io;
procedure Main is
  Car_Colour : Integer;
begin
  Car_Colour := 1;
  case Car_Colour is
    when 1      => Put("A red car"); New_Line;
    when 2      => Put("A blue car"); New_Line;
    when 3      => Put("A green car"); New_Line;
    when others => Put("Should not occur"); New_Line;
  end case;
end Main;
```

```
  type Colour is (Red,Blue,Green);
```

With: Enumerations

```
with Ada.Text_Io;
use  Ada.Text_Io;
procedure Main is
  type Colour is (Red,Blue,Green);
  Car_Colour : Colour;
begin
  Car_Colour := Blue;

  case Car_Colour is
    when Red    => Put("A red car"); New_Line;
    when Blue   => Put("A blue car"); New_Line;
    when Green  => Put("A green car"); New_Line;
  end case;
end Main;
```

Type Character

```
type Character is ( nul, soh,             -- etc
                    ' ', '!', '"',        -- etc
                    '@', 'A', 'B', 'C',   -- etc
                   );
```

```
 type Binary_Digit is ( '0','1' );
  B_Digit : Binary_Digit := '0';
```

The attributes 'Val and 'Pos

```
with Ada.Text_Io, Ada.Integer_Text_Io;
use  Ada.Text_Io, Ada.Integer_Text_Io;
procedure Main is
begin
  Put("Character 'A' has internal code ");
  Put( Character'Pos('A') ); New_Line;
  Put("Code 99 represents character    ");
  Put( Character'Val(99) ); New_Line;
end Main;
```

```
Character 'A' has internal code     65
Code 99 represents character    c
```

Operators: + - * / mod rem

+	Addition
-	Subtraction
*	Multiplication
/	Division

mod	Modulus
rem	Remainder

```
mod  | -5  -3   0   3   5
-------------------------
 -5  |  0  -2  Err  1   0
 -3  | -3   0  Err  0   2
  0  |  0   0  Err  0   0
  3  | -2   0  Err  0   3
  5  |  0  -1  Err  2   0
```

```
rem  | -5  -3   0   3   5
-------------------------
 -5  |  0  -2  Err -2   0
 -3  | -3   0  Err  0  -3
  0  |  0   0  Err  0   0
  3  |  3   0  Err  0   3
  5  |  0   2  Err  2   0
```

```
 **  | -3  -1   0   1   3
-------------------------
 -3  | Err Err  1  -3 -27
 -1  | Err Err  1  -1  -1
  0  | Err Err  1   0   0
  1  | Err Err  1   1   1
  3  | Err Err  1   3  27
```

The implementation of a ** b can be performed by multiplication in any order.

Hence `a**4` could be implemented as `a*a*a*a` or `(a*a)**2`.

Operators: Membership operators

in	is a member of
not in	is not a member of

```
if Ch in 'A' .. 'Z' then
  Put("Character is Upper Alphabetic"); New_Line;
end if;
```

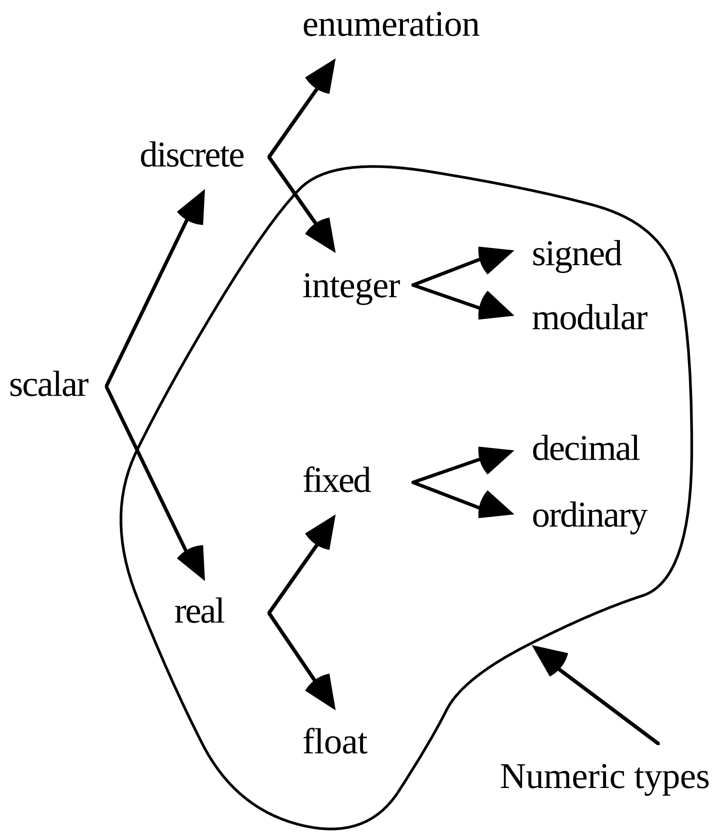
```
if Ch not in 'A' .. 'Z' then
  Put("Character is not Upper Alphabetic"); New_Line;
end if;
```

```
with Ada.Text_Io, Ada.Integer_Text_Io;
use  Ada.Text_Io, Ada.Integer_Text_Io;
procedure Main is
  subtype Exam_Mark is Integer range 0 .. 100;
  Mark : Integer;
begin
  Get( Mark );
  if Mark in Exam_Mark then
    Put("Valid mark for exam"); New_Line;
  end if;
end Main;
```

Standard types

Type	Classification	An instance of the type
`Boolean`	Enumeration	Holds either True or False.
`Character`	Enumeration	Holds a character based on the ISO 8859-1 character set. In which there are 256 distinct characters.
`Float`	Float	Holds numbers which contain a decimal place.
`Integer`	Integer	Holds whole numbers.
`Wide_character`	Enumeration	Holds a character based on the ISO 10646 BMP character set. In which there are 65536 distinct characters.

Type hierarchy



Component	Example declaration	Note
Scalar		
discrete		
Enumeration	**type** colour **is** (Red, Green, Blue);	1
Integer	**type** Miles **is range** 0 .. 10_000;	
Signed		
Modular	**type** Byte **is mod** 256;	2
Real		
Fixed		
Ordinary	**type** Miles **is delta** 0.1 **range** 0.0 .. 10.0;	3
Decimal	**type** Miles **is delta** 0.1 **digits** 8;	
Float	**type** Miles **is digits** 8 **range** 0.0 .. 10.0;	4

Note 1 *The enumeration types include the inbuilt types* `Character`, `Wide_character` *and* `Boolean`.

Note 1 *The enumeration types include the inbuilt types* `Character`, `Wide_character` *and* `Boolean`.

Note 2 *A modular type implements modular arithmetic. Thus, the following fragment of code:*

```
type Byte is mod 256;
count : Byte := 255;
begin
  count := count + 1;
```

would result in `count` *containing 0.*

Note 3 *A fixed point number is effectively composed of two components: the whole part and the fractional part stored in an integer value. This can lead to more efficient arithmetic on a machine which does not have floating point hardware or where the implementation of floating point arithmetic is slow. It also provides a precise way of dealing with numbers that have a decimal point.*
An alternative notation for a decimal fixed point type is:
type Miles **is delta** 0.1 **digits** 8 **range** 0.0 .. 10.0;
However, even though all compilers must parse this type declaration they only need to support it if the compiler implements the Information systems Annex.

Note 4 *A floating point number.*
An alternative type declaration is: **type** Miles **is digits** 8; *which defines the precision 8 digits but not the range of values that may be stored.*

Relational operators

=	equal
/=	not equal
<	less than
>	greater than
<=	less than or equal
>=	greater than or equal

```
with Ada.Text_Io, Ada.Integer_Text_Io;
use  Ada.Text_Io, Ada.Integer_Text_Io;
procedure Main is
  Temperature : Integer;  --Temperature in Centigrade
  Hot         : Boolean;  --Is it hot
begin
  Get( Temperature );
  if Temperature > 24 then
    Put("It's warm"); New_Line;
  end if;
  Hot := Temperature > 30;
  if Hot then
    Put("It's hot"); New_Line;
  end if;
end Main;
```

Boolean operators

`and`	logical and Note: Both LHS and RHS evaluated
`or`	logical or Note: Both LHS and RHS evaluated
`and then`	logical and Note: RHS only evaluated if LHS TRUE
`or else`	logical or Note: RHS only evaluated if LHS FALSE

Operators: `Boolean` use of

```
procedure main is
  Day,Month : Natural;
  Christmas : Boolean;
begin
  Get( Day ); Get( Month );
  if Day = 25 and Month = 12 then
    put("Happy Christmas"); new_line;
  end if;
  Christmas := Day = 25 and Month = 12;
end main;
```

`if Month = 2 and then Day = 29 then`
`  -- The 29th of February`
`end if;`
	`if Month = 2 then`
	`    if Day = 29 then`
	`      -- The 29th of February`
	`    end if;`
	`  end if;`

Operators: Monadic `Boolean` operators

`not`	not

```
if not (Month = 2) then
  Put("Month is not February"); New_Line;
end if;
```

Operators: Bitwise operators

and	bitwise and
or	bitwise or

```
K : constant := 1024;
type Word16 is mod 64 * K;
Pattern : Word16;
```

the following code sets the top nibble of the two byte word `pattern` to zero.

```
Pattern := Pattern and 16#FFF#;
```

sets bit 9 in the two byte word `pattern` to 1.

```
Pattern := Pattern or 2#0000001000000000#;
```

Procedures and functions

```
  type Miles      is digits 8 range 0.0 .. 25_000.0;
  type Kilometres is digits 8 range 0.0 .. 50_000.0;

  function M_To_K_Fun(M:in Miles) return Kilometres is
    Kilometers_Per_Mile : constant := 1.609_344;
  begin
    return  Kilometres( M * Kilometers_Per_Mile );
  end M_To_K_Fun;
```

```
with Ada.Text_Io, Ada.Float_Text_Io;
use  Ada.Text_Io, Ada.Float_Text_Io;
procedure Main1 is
  type Miles      is digits 8 range 0.0 .. 25_000.0;
  type Kilometres is digits 8 range 0.0 .. 50_000.0;

  function M_To_K_Fun(M:in Miles) return Kilometres is
    Kilometers_Per_Mile : constant := 1.609_344;
  begin
    return  Kilometres( M * Kilometers_Per_Mile );
  end M_To_K_Fun;

  No_Miles : Miles;

begin
  Put("Miles  Kilometres"); New_Line;
  No_Miles := 0.0;
  while No_Miles <= 10.0 loop
    Put( Float(No_Miles), Aft=>2, Exp=>0 ); Put("    ");
    Put( Float( M_To_K_Fun( No_Miles ) ), Aft=>2, Exp=>0 );
    New_Line;
    No_Miles := No_Miles + 1.0;
  end loop;
end Main1;
```

Procedures

```
   type Miles      is digits 8 range 0.0 .. 25_000.0;
   type Kilometers is digits 8 range 0.0 .. 50_000.0;

   procedure M_To_K_Proc(M:in Miles; Res:out Kilometers) is
     Kilometers_Per_Mile : constant := 1.609_344;
   begin
     Res :=  Kilometers( M * Kilometers_Per_Mile );
   end M_To_K_Proc;
```

```
with Ada.Text_Io, Ada.Float_Text_Io;
use  Ada.Text_Io, Ada.Float_Text_Io;
procedure Main is
   type Miles      is digits 8 range 0.0 .. 25_000.0;
   type Kilometers is digits 8 range 0.0 .. 50_000.0;

   procedure M_To_K_Proc(M:in Miles; Res:out Kilometers) is
     Kilometers_Per_Mile : constant := 1.609_344;
   begin
     Res :=  Kilometers( M * Kilometers_Per_Mile );
   end M_To_K_Proc;

   No_Miles : Miles;
   No_Km    : Kilometers;

begin
   Put("Miles  Kilometers"); New_Line;
   No_Miles := 0.0;
   while No_Miles <= 10.0 loop
     Put( Float(No_Miles), Aft=>2, Exp=>0 ); Put("    ");
     M_To_K_Proc( No_Miles, No_Km );
     Put( Float( No_Km ), Aft=>2, Exp=>0 );
     New_Line;
     No_Miles := No_Miles + 1.0;
   end loop;
end Main;
```

Formal and actual parameters

Terminology	Commentary
Formal parameter	The parameter used in the declaration of a function or procedure. For example, in the function `M_To_K_Fun` the formal parameter is `M`.
Actual parameter	The object passed to the function or procedure when the function or procedure is called. For example, in the procedure `M_To_K_Proc` the actual parameters are No_Miles and No_Km. An expression may also be passed as an actual parameter to a function or procedure, provided the mode of the formal parameter is not **out**;

Mode of formal parameters

Mode	Used in	Effect
`in`	function or procedure	The formal parameter is initialised to the contents of the actual parameter and may be read from only.
`in out`	procedure only	The formal parameter is initialised to the contents of the actual parameter and may be read from or written to. When the procedure is exited the new value of the formal parameter replaces the old contents of the actual parameter.
`out`	procedure only	The formal parameter is **not** initialised to the contents of the actual parameter and may be read from or written to. When the procedure is exited the new value of the formal parameter replaces the old contents of the actual parameter. Ada83: An out formal parameter may not be read from

Using mode in out

```
procedure Swap(First:in out Integer; Second:in out Integer) is
  Temp : Integer;
begin
  Temp := First;
  First := Second; Second := Temp;
end Swap;
```

Putting it all together

```
with Ada.Text_Io, Ada.Integer_Text_Io, Swap;
use  Ada.Text_Io, Ada.Integer_Text_Io;
procedure Main is
  Books_Room_1 : Integer;
  Books_Room_2 : Integer;
begin
  Books_Room_1 := 10; Books_Room_2 := 20;
  Put("Books in room 1 ="); Put(Books_Room_1); New_Line;
  Put("Books in room 2 ="); Put(Books_Room_2); New_Line;
  Put("Swap around"); New_Line;
  Swap(Books_Room_1, Books_Room_2);
  Put("Books in room 1 ="); Put(Books_Room_1); New_Line;
  Put("Books in room 2 ="); Put(Books_Room_2); New_Line;
end Main;
```

```
Books in room 1 =          10
Books in room 2 =          20
Swap around
Books in room 1 =          20
Books in room 2 =          10
```

Formal parameter specified by: (Using as an example an `Integer` formal parameter)	Write to formal parameter allowed.	Read from formal parameter	Can be used as a parameter to
`item: Integer`	✗	✔	procedure or function
`item: in Integer`	✗	✔	procedure or function
`item: in out Integer`	✔	✔	procedure only

`item: out Integer`	✔	✔	procedure only

Recursion

- Split the natural number into two components
  - The first digit (remainder when number divided by 10)
  - The other digits (number divided by 10).

For example:

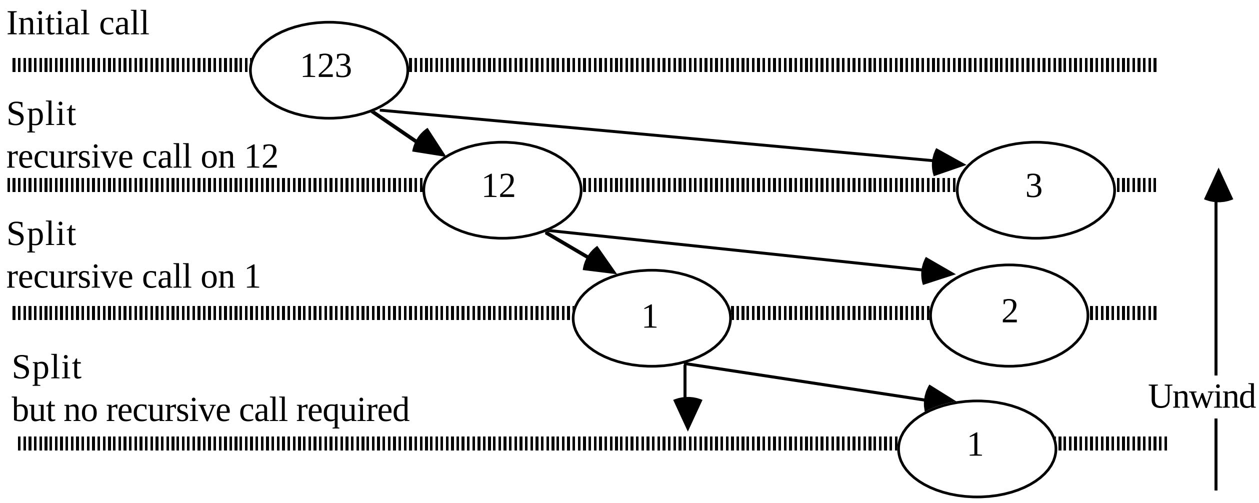
123 would be split into:

3 (first digit)

12 (other digits).

- If the other digits is greater than or equal to 10 then write the other digits by recursively calling the code to write a decimal number.

- Output the first digit as a character.



Recursion: Ada code

```
with Ada.Text_Io; use Ada.Text_Io;
procedure Write_Natural( Num : Natural) is
  First_Digit  : Natural;           --Unit digit
  Other_Digits : Natural;           --All except first digit
begin
  First_Digit  := Num rem 10;       --Split 1234  =>    4
  Other_Digits := Num / 10;         --            => 123
  if Num >= 10 then                 --Print other digits
    Write_Natural( Other_Digits );  --Recursive call
  end if;
  Put( Character'Val( First_Digit + Character'Pos('0') ) );
end Write_Natural;
```

Recursion example

```
with Ada.Text_Io, Write_Natural;
use  Ada.Text_Io;
procedure Main is
begin
  Write_Natural( 123 );                  New_Line;
  Write_Natural( 12345 );                New_Line;
end Main;
```

```
123
12345
```

Overloading of functions

```
with Ada.Integer_Text_Io;
use  Ada.Integer_Text_Io;
procedure Answer_Is( N:in Integer;
                     Message:in Boolean := True) is
begin
  if Message then Put("The answer = "); end if;
  Put( N, Width=>1 );
  if Message then New_Line; end if;
end Answer_Is;

with Ada.Text_Io, Ada.Integer_Text_Io;
use  Ada.Text_Io, Ada.Integer_Text_Io;
procedure Is_A_Int( An_Int:in Integer ) is
begin
  Put("The parameter is an Integer:  value = ");
  Put( An_Int, Width=>1 ); New_Line;
end Is_A_Int;

with Ada.Text_Io, Ada.Float_Text_Io;
use  Ada.Text_Io, Ada.Float_Text_Io;
procedure Is_A_Float( A_Float:in Float ) is
begin
  Put("The parameter is a Float:     value = ");
  Put( A_Float, Aft=>2, Exp=>0 ); New_Line;
end Is_A_Float;
```

```
with Is_A_Int, Is_A_Float, Is_A_Char;
procedure Main is
  procedure Is_A( The:in Integer )   renames Is_A_Int;
  procedure Is_A( The:in Float )     renames Is_A_Float;
  procedure Is_A( The:in Character ) renames Is_A_Char;
begin
  Is_A( 'A' );
  Is_A( 123 );
  Is_A( 123.45 );
end Main;
```

```
The parameter is a Character: value = A
The parameter is an Integer:  value = 123
The parameter is a Float:     value = 123.450
```

Different number of parameters to a function

```
function Max2( A,B:in Integer ) return Integer is
begin
  if A > B then
    return A;         --a is larger
  else
    return B;         --b is larger
  end if;
end Max2;
```

```
with Max2;
function Max3( A,B,C:in Integer ) return Integer is
begin
  return Max2( Max2( A,B ), C );
end Max3;
```

```
with Ada.Text_io, Ada.Integer_Text_Io, Max2, Max3;
use  Ada.Text_io, Ada.Integer_Text_Io;
procedure Main is
  function Max(A,B:in Integer)   return Integer renames Max2;
  function Max(A,B,C:in Integer) return Integer renames Max3;
begin
  Put("Larger of 2 and 3 is "); Put( Max(2,3) );    New_Line;
  Put("Larger of 2 3 4   is "); Put( Max(2,3,4) );  New_Line;
end Main;
```

```
Larger of 2 and 3 is 3
Larger of 2 3 4   is 4
```

Default parameters

```
function  Sum( P1:in Integer := 0;
               P2:in Integer := 0;
               P3:in Integer := 0;
               P4:in Integer := 0  ) return Integer is
begin
  return P1 + P2 + P3 + P4;
end Sum;

with Ada.Text_Io, Ada.Integer_Text_io;
use  Ada.Text_Io, Ada.Integer_Text_io;
procedure Answer_Is( N:in Integer;
                     Message:in Boolean := True ) is
begin
  if Message then Put("The answer = "); end if;
  Put( N, Width=>1 );
  if Message then New_Line; end if;
end Answer_Is;
```

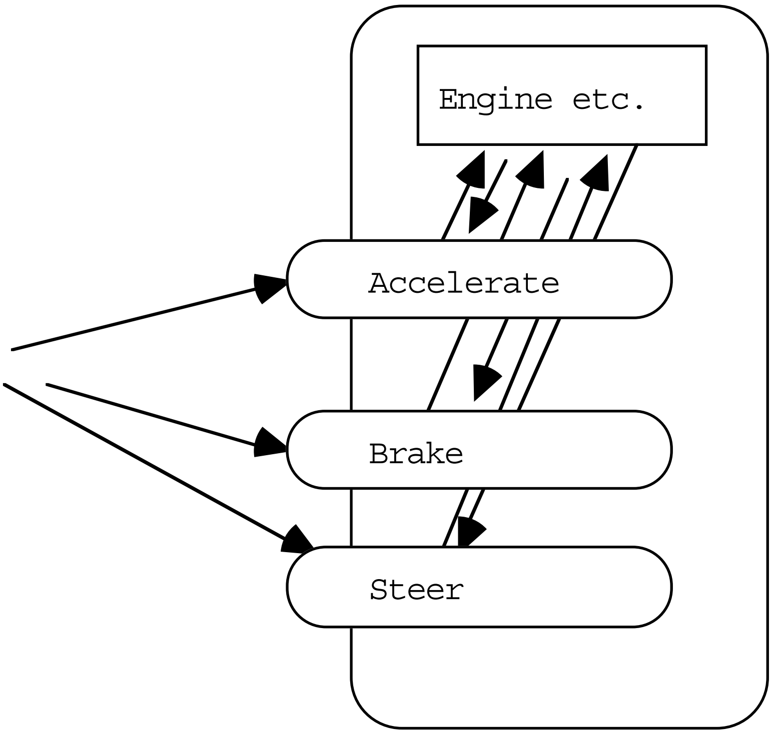
```
  Answer_Is( 27, True );                    -- By position
  Answer_Is( 27, Message => False );        -- By name
```

```
with Sum, Answer_Is;
procedure Main is
begin
  Answer_Is( Sum );
  Answer_Is( Sum( 1, 2 ) );
  Answer_Is( Sum( 1, 2, 3 ) );
  Answer_Is( Sum( 1, 2, 3, 4 ), Message => False );
  New_Line;
end Main;
```

```
The answer = 0
The answer = 3
The answer = 6
10
```


The class

Actions required to drive an automatic car.



object	An item that has a hidden internal structure. The hidden structure is manipulated or accessed by messages sent by a user.
message	A request sent to the object to obey one of its methods.
method	A set of actions that manipulates or accesses the internal state of the object. The detail of these actions is hidden from a user of the object.

A package to represent a bank account

- Deposit money into the account.
- Withdraw money from the account.
- Deliver the account balance.
- Print a mini statement of the amount in the account.

```
with Ada.Text_Io, Class_Account, Statement;
use  Ada.Text_Io, Class_Account;
procedure Main1 is
  My_Account: Account;
  Obtain    : Money;
begin
  Statement( My_Account );

  Put("Deposit £100.00 into account"); New_Line;  --Deposit
  Deposit( My_Account, 100.00 );
  Statement( My_Account );

  Put("Withdraw £80.00 from account"); New_Line;  --Withdraw
  Withdraw( My_Account, 80.00, Obtain );
  Statement( My_Account );

  Put("Deposit £200.00 into account"); New_Line;  --Deposit
  Deposit( My_Account, 200.00 );
  Statement( My_Account );

end Main1;
```

```
Deposit( My_Account, 100.00 );
```

```
Withdraw( My_Account, 80.00, obtain );
```

Results

```
Mini statement: The amount on deposit is £ 0.00

Deposit £100.00 into account
Mini statement: The amount on deposit is £100.00

Withdraw £80.00 from account
Mini statement: The amount on deposit is £20.00

Deposit £200.00 into account
Mini statement: The amount on deposit is £220.00
```

Components of a package

- The specification
- The implementation

Ada package component	Object-oriented component
Specification	The type used to elaborate the object, plus the specification of the messages that can be sent to an instance of the type.
Implementation	Implementation of the methods that are evoked when a message is sent to the object.

```
package Class_Account is

  type Account is private;
  subtype Money  is Float;
  subtype Pmoney is Float range 0.0 .. Float'Last;

  procedure Deposit ( The:in out Account; Amount:in Pmoney );
  procedure Withdraw( The:in out Account; Amount:in Pmoney;
                      Get:out Pmoney );
  function  Balance ( The:in Account ) return Money;

private
  type Account is record
    Balance_Of : Money := 0.00;      --Amount in account
  end record;
end Class_Account;
```

Specification of a package

```
package Class_account is
  type    Account is private;

  subtype Money   is Float;
  subtype PMoney  is Float range 0.0 .. Float'Last;

  procedure statement( the:in Account );
  procedure deposit(the:in out Account; amount:in PMoney);
  procedure withdraw(the:in out Account; amount:in PMoney;
                     get:out PMoney);
  function  balance( the:in Account ) return Money;
private
  type Account is record
     balance_of : Money := 0.00;
  end record;

end Class_account;
```

Public Interface

Private Interface

Used to declare instance of the class

Types and subtypes used in messages

Messages that can be sent to an `Account` object

Hidden representation of class `Account`

Implementation of a package

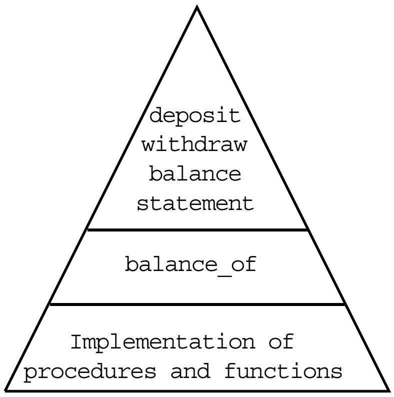
```
package body Class_Account is

  procedure Deposit ( The:in out Account; Amount:in Pmoney ) is
  begin
    The.Balance_Of := The.Balance_Of + Amount;
  end Deposit;

  procedure Withdraw( The:in out Account; Amount:in Pmoney;
                      Get:out Pmoney ) is
  begin
    if The.Balance_Of >= Amount then
      The.Balance_Of := The.Balance_Of - Amount;
      Get := Amount;
    else
      Get := 0.00;
    end if;
  end Withdraw;

  function  Balance( The:in Account ) return Money is
  begin
    return The.Balance_Of;
  end Balance;

end Class_Account;
```

Visibility

Visible to a client of the package.

-

Invisible to a client of the package



Component is:

In the public part of the package specification.

-

In the private part of the package specification.

-

In the body of the package.

with and **use**

```
with Ada.Text_Io, Class_Account, Statement;
procedure Main is
  My_Account: Class_Account.Account;
  Obtain     : Class_Account.Money;
begin
  Statement( My_Account );

  Ada.Text_Io.Put("Deposit £100.00 into account");
  Ada.Text_Io.New_Line;
  Class_Account.Deposit( My_Account, 100.00 );
  Statement( My_Account );

  Ada.Text_Io.Put("Withdraw £80.00 from account");
  Ada.Text_Io.New_Line;
  Class_Account.Withdraw( My_Account, 80.00, Obtain );
  Statement( My_Account );

  Ada.Text_Io.Put("Deposit £200.00 into account");
  Ada.Text_Io.New_Line;
  Class_Account.Deposit( My_Account, 200.00 );
  Statement( My_Account );

end Main;
```

Use a **use** clause	Do not use a **use** clause
Program writing is simplified. Confusion may arise at to which package the item used is a member of.	A program must explicitly state which package the component is taken from. This can reduce the possibility of program error due to accidental misuse.

A personnel account manager

```
[a]  Deposit

[b]  Withdraw

[c]  Balance

Input selection: a

Amount to deposit : 10.00
```

```
[a]  Deposit

[b]  Withdraw

[c]  Balance

Input selection: b
Amount to withdraw : 4.60
```

```
[a]  Deposit

[b]  Withdraw

[c]  Balance

Input selection: c

Balance is  5. 40
```

Method	Responsibility
menu	Set up the menu that will be displayed to a user. Each menu item wil be deined as a String.
event	Return the menu item selected by a user of the TUI.
message	Display a message from the user.
dialogue	Get a response from the user.

```
package Class_TUI is

  type Menu_Item is ( M_1, M_2, M_3, M_4, M_Quit );
  type TUI is private;

  procedure Menu( The:in out TUI; M1,M2,M3,M4:in String );
  function  Event( The:in TUI ) return Menu_Item;
  procedure Message( The:in TUI; Mes:in String );
  procedure Dialog(The:in TUI; Mes:in String; Res:out Float);
  procedure Dialog(The:in TUI; Mes:in String; Res:out Integer);
private
  -- Not a concern of the client of the class
end Class_TUI;
```

Use

```
screen : TUI;
```

```
[a]  Print
[b]  Calculate

Input selection:
```

```
Menu( Screen, "Print", "Calculate", "", "" );
```

Example

```
    Message( Screen, "Distance converter" );
    Dialog ( Screen, "Enter distance in miles", Miles );
    Message( Screen, "Distance in kilometers is " &
                     Float'Image( Miles * 1.6093 )  );
```

```
with Class_Account, Class_TUI;
use  Class_Account, Class_TUI;
procedure Main is
  User     : Account;              --The users account
  Screen   : TUI;                  --The display screen
  Cash     : Money;                --
  Received : Money;                --
```

```
  function Float_Image( F:in Float ) return String is
    Res : String( 1 .. 10 );       --String of 10 characters
  begin
    Put( Res, F, 2, 0 );           --2 digits - NO exp
    return Res;
  end Float_Image;
```

```
begin
  loop
    Menu( Screen, "Deposit", "Withdraw", "Balance", "" );
    case Event( Screen ) is
      when M_1 =>                                     --Deposit
        Dialog( Screen, "Amount to deposit", Cash );
        if Cash <= 0.0 then
          Message( Screen, "Must be >= 0.00" );
        else
          Deposit( User, Cash );
        end if;
      when M_2 =>                                      --Withdraw
        Dialog( Screen, "Amount to withdraw", Cash );
        if Cash <= 0.0 then
          Message( Screen, "Must be >= 0.00" );
        else
          Withdraw( User, Cash, Received );
          if Received <= 0.0 then
            Message( Screen, "Not enough money" );
          end if;
        end if;
      when M_3 =>                                     --Balance
        Message( Screen, "Balance is " &
                         Float_Image( Balance(User)) );
      when M_Quit =>                                  --Exit
        return;
      when others =>                                  --Not used
        Message( Screen, "Program error");            --oops
    end case;
  end loop;
end Main;
```

```
package Class_TUI is

  type Menu_Item is ( M_1, M_2, M_3, M_4, M_Quit );
  type TUI is private;

  procedure Menu( The:in out TUI; M1,M2,M3,M4:in String );
  function  Event( The:in TUI ) return Menu_Item;
  procedure Message( The:in TUI; Mes:in String );
  procedure Dialog(The:in TUI; Mes:in String; Res:out Float);
  procedure Dialog(The:in TUI; Mes:in String; Res:out Integer);
private
  type TUI is record
    Selection : Menu_Item := M_Quit;
  end record;
end Class_TUI;
```

```
with Ada.Text_Io, Ada.Float_Text_Io, Ada.Integer_Text_Io;
use  Ada.Text_Io, Ada.Float_Text_Io, Ada.Integer_Text_Io;
package body Class_TUI is
  procedure Menu( The:in out TUI; M1,M2,M3,M4:in String ) is

    Selection      : Character;
    Valid_Response : Boolean := False;
```

```
    procedure Set_Response(Choice:in Menu_Item; Mes:in String) is
    begin
      if Mes /= "" then                  --Allowable choice
        The.Selection := Choice; Valid_Response := True;
      end if;
    end Set_Response;
```

```
    procedure Display_Menu_Item(Prompt, Name:in String) is
    begin
      if Name/="" then
        Put(Prompt & Name); New_Line(2);
      end if;
    end Display_Menu_Item;
```

```
begin   -- Menu
  while not Valid_Response loop
    Display_Menu_Item( "[a]   ", M1 );
    Display_Menu_Item( "[b]   ", M2 );
    Display_Menu_Item( "[c]   ", M3 );
    Display_Menu_Item( "[d]   ", M4 );
    Put( "Input selection: "); Get( Selection ); Skip_Line;
    case Selection is
      when 'a' | 'A' => Set_Response( M_1, M1 );
      when 'b' | 'B' => Set_Response( M_2, M2 );
      when 'c' | 'C' => Set_Response( M_3, M3 );
      when 'd' | 'D' => Set_Response( M_4, M4 );
      when 'e' | 'E' => Set_Response( M_Quit, "Quit" );
      when others    => Valid_Response := False;
    end case;
    if not Valid_Response then
      Message( The, "Invalid response" );
    end if;
  end loop;
end Menu;
```

```
function  Event( The:in TUI ) return Menu_Item is
begin
  return The.Selection;
end;
```

```
procedure Message( The:in TUI; Mes:in String ) is
begin
  New_Line; Put( Mes ); New_Line;
end Message;
```

```
procedure Dialog(The:in TUI; Mes:in String; Res:out Float) is
  begin
    New_Line(1); Put( Mes & " : " );
    Get( Res ); Skip_Line;
  end Dialog;

  procedure Dialog(The:in TUI; Mes:in String; Res:out Integer) is
  begin
    New_Line(1); Put( Mes & " : " );
    Get( Res ); Skip_Line;
  end Dialog;

end Class_TUI;
```

Data Structures

```
Max_Chs : constant := 10;
type Gender    is ( Female, Male );
type Height_Cm is range 0 .. 300;
type Person is record
  Name   : String( 1 .. Max_Chs );  --Name as a String
  Height : Height_Cm := 0;          --Height in cm.
  Sex    : Gender;                  --Gender of person
end record;
```

```
Mike    : Person;
```

```
Mike.Name   := "Mike      ";
Mike.Height := 183;
Mike.Sex    := Male;
```

```
Mike    := (Name=> "Mike      ", Height=> 183, Sex=> Male);
```

```
Corinna, Mike, Miranda : Person;
Taller                 : Person;
```

```
Mike    := (Name=>"Mike      ", Height=>183, Sex=>Male);
Corinna:= (Name=>"Corinna   ", Height=>171, Sex=>Female);
Miranda:= (Name=>"Miranda   ", Height=>74,  Sex=>Female);

Taller  := Mike;

if mike = Taller then
  Put("Mike taller"); New_Line;
end if;
if Mike /= Taller and Corinna /= Taller then
  Put("Miranda taller"); New_Line;
end if;
```

Limited

```
type Person is limited record
  name   : String( 1 .. MAX_CHS );  -- Name as a String
  height : Height_cm := 0;          -- Height in cm.
  sex    : Gender;                  -- Gender of person
end record;
```

```
mike    : Person;
corinna : Person;
```

```
mike := corinna;  -- Fails to compile as record can not be copied
```

Nested records

```
type Bus is record
  Driver : Person;         --Bus driver
  Seats  : Positive;       --Number of seats on bus
end record;

London : Bus;
```

```
London.Driver.Name   := "Jane      ";
London.Driver.Sex    := Female;
London.Driver.Height := 168;
London.Seats         := 46;
```

```
type    Gender    is ( Female, Male );
type    Height_Cm is range 0 .. 300;
subtype Str_Range is Natural range 0 .. 20;
type    Person( Chs: Str_Range ) is record     --Name length
  Name   : String( 1 .. Chs );                 --As String
  Height : Height_Cm   := 0;                   --Height in cm.
  Sex    : Gender;                             --Gender
end record;
```

```
  Mike   : Person(4);     --Constrained
  Corinna: Person(7);     --Constrained
  Younger: Person(10);    --Constrained
```

```
 Mike   := (4, Name=>"Mike"   , Height=>183, Sex=>Male);
 Corinna:= (7, Name=>"Corinna", Height=>171, Sex=>Female);
```

```
Younger := Corinna;   -- Fail at run-time
```

Unconstrained record

```
type     Gender    is ( Female, Male );
type     Height_Cm is range 0 .. 300;
subtype Str_Range is Natural range 0 .. 20;
type     Person( Chs:Str_Range := 0 ) is record --Length of name
   Name   : String( 1 .. Chs );                --Name as String
   Height : Height_Cm   := 0;                  --Height in cm.
   Sex    : Gender;                            --Gender
end record;


declare
   Mike    : Person;      --Unconstrained
   Corinna: Person;       --Unconstrained
   Younger: Person;       --Unconstrained
begin
   Mike    := (4, Name=>"Mike"    , Height=>183, Sex=>Male);
   Corinna:= (7, Name=>"Corinna", Height=>171, Sex=>Female);
   Younger := Corinna;

   if Corinna = Younger then
     Put("Corinna is younger"); New_Line;
   end if;
end;
```

Declaration	**The object `Mike` is**	**Comment**
`Mike: Person;`	Unconstrained	The variable `Mike` may be compared with or assigned any other instance of `Person`.
`Mike: Person(4);`	Constrained	May only be assigned or compared with another `Person(4)`.

```
type     Occupation is (Lecturer, Student);
type     Mark       is range 0 .. 100;
subtype Str_Range  is Natural range 0 .. 20;
type     Person( Chs : Str_Range :=0;
   Role: Occupation:=Student ) is record
   Name   : String( 1 .. Chs );      --Name as string
   case Role is                      --Variant record
     when Lecturer =>                -- Remember storage overlaid
       Class_Size: Positive;         --Size of taught class
     when Student  =>
       Grade      : Mark;            --Mark for course
       Full_Time : Boolean := True; --Attendance mode
   end case;
end record;
```

```
declare
  Mike : Person;                --Unconstrained
  Clive: Person;                --Unconstrained
  Brian: Person(5,Student);     --Constrained
begin
  Mike :=(4, Lecturer, Name=>"Mike", Class_Size=>36);
  Clive:=(5, Student,  Name=>"Clive", Grade=>70,Full_Time=>True);
  -- insert --
end;
```

Invalid statement	**Reason**
`Clive.Role:= STUDENT`	Not allowed to change just a discriminant as this would allow data to be modified/extracted as the wrong type. Detectable at compile-time.
`Mike.Grade:= 0`	Access to a component of the data structure which is not active. Mike is a lecturer and hence has no grade score. Detected at run-time.
`Brian := (5, Lecturer, Name=>"Brian", Class_Size=>36);`	The object `Brian` is constrained to be a student. Detectable at compile-time.

Arrays

```
Computers_In_Room : array ( 1 .. 5 ) of Natural;
```

```
  Computers_In_Room(1) := 20;
  Computers_In_Room(2) := 30;
  Computers_In_Room(3) := 25;
  Computers_In_Room(4) := 10;
  Computers_In_Room(5) := 15;
```

1	2	3	4	5
20	30	25	10	15

Index used to access contents of

array collection `computers_room`

```
  for I in 1 .. 5 loop
    Put("Computers in room "); Put( I, Width=>1 ); Put(" is ");
    Put( Computers_In_Room(I), Width=>2 ); New_Line;
  end loop;
```

```
Computers in room 1 is 20
Computers in room 2 is 30
Computers in room 3 is 25
Computers in room 4 is 10
Computers in room 5 is 15
```

Using types with array declarations

```
type    Rooms_Index is range 1 .. 5;
type    Rooms_Array is array ( Rooms_Index ) of Natural;

Computers_In_Room : Rooms_Array;
```

Type / Subtype	Description
Rooms_index	A type used to define an object that is used to index the array.
Rooms_array	A type representing the array.

```
  Computers_In_Room : Rooms_array;

  -- Set up contents of Computers_In_Room

  for I in Computers_In_Room'Range loop
    Put("Computers in room "); Put( Integer(I), Width=>1 );
    Put(" is "); Put( Computers_In_Room(I), Width=>2 ); New_Line;
  end loop;
```

Attributes: array

```
type    Marks_Index is new Character range 'a' .. 'f';
type    Marks_Array is array (Marks_Index ) of Natural;
Marks : Marks_Array;
```

Attribute	Description	Value
Marks'Length	A Universal integer representing the number of elements in the one dimensional array.	6
Marks'First	The first subscript of the array which is of type Marks_Range	'a'
Marks'Last	The last subscript of the array which is of type Marks_Range	'f'
Marks'Range	Equivalent to Marks'First .. Marks'Last	'a'..'f'

The game tick-tack-toe

<table>
<tr><td>X’s first move</td><td>O’s first move</td><td>X’s second move</td><td>O’s second move</td></tr>
<tr>
<td><table><tr><td>X</td><td></td><td></td></tr><tr><td></td><td></td><td></td></tr><tr><td></td><td></td><td></td></tr></table></td>
<td><table><tr><td>X</td><td></td><td></td></tr><tr><td></td><td></td><td></td></tr><tr><td></td><td>O</td><td></td></tr></table></td>
<td><table><tr><td>X</td><td></td><td>X</td></tr><tr><td></td><td></td><td></td></tr><tr><td></td><td>O</td><td></td></tr></table></td>
<td><table><tr><td>X</td><td>O</td><td>X</td></tr><tr><td></td><td></td><td></td></tr><tr><td></td><td>O</td><td></td></tr></table></td>
</tr>
<tr><td>X’s third move</td><td>O’s third move</td><td>X’s fourth move</td><td></td></tr>
<tr>
<td><table><tr><td>X</td><td>O</td><td>X</td></tr><tr><td></td><td>X</td><td></td></tr><tr><td></td><td>O</td><td></td></tr></table></td>
<td><table><tr><td>X</td><td>O</td><td>X</td></tr><tr><td></td><td>X</td><td></td></tr><tr><td>O</td><td>O</td><td></td></tr></table></td>
<td><table><tr><td>X</td><td>O</td><td>X</td></tr><tr><td></td><td>X</td><td></td></tr><tr><td>O</td><td>O</td><td>X</td></tr></table></td>
<td>As can be seen to go first is a clear advantage</td>
</tr>
</table>

Method	Responsibility
`Add`	Add the player’s mark to the board. The player’s move is specified by a number in the range 1 to 9 and their mark by a character.
`Valid`	Return true if the presented move is valid. The method checks that the move is in the range 1 to 9 and that the specified cell is not occupied.
`State`	Returns the state of the current game. Possible states are: Win, Playable, and Draw.
`Cell`	Returns the contents of a cell on the board. This method is included so that the code that displays the board can be separated from the code that manipulates the board.
`Reset`	Reset the board back to an initial state.

Specification

```
package Class_Board is

  type Board      is private;
  type Game_State is ( Win, Playable, Draw );

  procedure Add( The:in out Board; Pos:in Integer;
                 Piece:in Character );
  function  Valid( The:in Board; Pos:in Integer ) return Boolean;
  function  State( The:in Board ) return Game_State;
  function  Cell( The:in Board; Pos:in Integer )
                return Character;
  procedure Reset( The:in out Board );
  -- Not a concern of the client
end Class_Board;
```

A program to use `Class_Board`

```
with Class_Board, Ada.Text_Io, Ada.Integer_Text_Io, Display;
use  Class_Board, Ada.Text_Io, Ada.Integer_Text_Io;
procedure Main is
  Player : Character;            --Either 'X' or 'O'
  Game   : Board;                --An instance of Class Board
  Move   : Integer;              --Move from user
begin
  Player := 'X';                             --Set player

  while State( Game ) = Playable loop        --While playable
    Put( Player & " enter move (1-9) : "); --  move
    Get( Move ); Skip_Line;                  -- Get move
    if Valid( Game, Move ) then              --Valid
      Add( Game, Move, Player );             -- Add to board
      Display( Game );                       -- Display board
      case State( Game ) is                  --Game is
        when Win      =>
          Put( Player & " wins");
        when Playable  =>
          case Player is                     --Next player
            when 'X'    => Player := 'O';    -- 'X' => 'O'
            when 'O'    => Player := 'X';    -- 'O' => 'X'
            when others => null;             --
          end case;
        when Draw      =>
          Put( "It's a draw ");
      end case;
      New_Line;
    else
      Put("Move invalid"); New_Line;         --for board
    end if;
  end loop;
  New_Line(2);
end Main;
```

Displaying the Board

```
with Class_Board, Ada.Text_Io;
use  Class_Board, Ada.Text_Io;
procedure Display( The:in Board ) is
begin
  for I in 1 .. 9 loop
    Put( Cell( The, I ) );
    case I is                           --after printing counter
      when 3 | 6  =>                    -- print Row Separator
        New_Line; Put("---------");     --
        New_Line;
      when 9      =>                    -- print new line
        New_Line;
      when 1 | 2 | 4 | 5 | 7 | 8 =>    -- print Col separator
        Put(" | ");
    end case;
  end loop;
end Display;
```

Specification

```
package Class_Board is

  type Board      is private;
  type Game_State is ( Win, Playable, Draw );

  procedure Add( The:in out Board; Pos:in Integer;
                 Piece:in Character );
  function  Valid( The:in Board; Pos:in Integer ) return Boolean;
  function  State( The:in Board ) return Game_State;
  function  Cell( The:in Board; Pos:in Integer )
                 return Character;
  procedure Reset( The:in out Board );
private
  subtype Board_Index is Integer range 1 .. 9;
  type    Board_Array is array( Board_Index ) of Character;
  type Board is record
    Sqrs  : Board_Array := ( others => ' ' );   --Initialize
    Moves : Natural     := 0;
  end record;
end Class_Board;
```

Type / Subtype	Description
`Board_Index`	A subtype used to describe an index object used to access an element of the noughts and crosses board. By making `Board_Index` a subtype of `Integer`, `Integers` may be used as an index of the array.
`Board_Array`	A type used to describe a noughts and crosses board.

Implementation

```
package body Class_Board is

procedure Add( The:in out Board; Pos:in Integer;
                 Piece:in Character ) is
  begin
    The.Sqrs( Pos ) := Piece;
  end Add;


  function Valid(The:in Board; Pos:in Integer) return Boolean is
  begin
    return Pos in Board_Array'Range and then
           The.Sqrs( Pos ) = ' ';
  end Valid;


  function Cell(The:in Board; Pos:in Integer) return Character is
  begin
    return The.Sqrs( Pos );
  end Cell;

  procedure Reset( The:in out Board ) is
  begin
    The.sqrs  := ( others => ' ');    --All spaces
    The.moves := 0;                   --No of moves
  end reset;
```

```
  function  State( The:in Board ) return Game_State is
    subtype Position   is Integer range 1 .. 9;
    type Win_Line      is array( 1 .. 3 ) of Position;
    type All_Win_Lines is range 1 .. 8;
    Cells: constant array ( All_Win_Lines ) of Win_Line :=
      ( (1,2,3), (4,5,6), (7,8,9), (1,4,7),
        (2,5,8), (3,6,9), (1,5,9), (3,5,7) ); --All win lines
    First : Character;
  begin
    for Pwl in All_Win_Lines loop             --All Pos Win Lines
      First := The.Sqrs( Cells(Pwl)(1) ); --First cell in line
      if First /= ' ' then                    -- Looks promising
        if First = The.Sqrs(Cells(Pwl)(2)) and then
           First = The.Sqrs(Cells(Pwl)(3)) then return Win;
        end if;
      end if;
    end loop;
    if The.Moves >= 9 then                    --Check for draw
      return Draw;                            -- Board full
    else
      return Playable;                        -- Still playable
    end if;
  end State;

end Class_Board;
```

Results

<table>
<tr><td>X’s first move</td><td>O’s first move</td><td>X’s second move</td><td>O’s second move</td></tr>
<tr>
<td>X | |

  | |

  | |</td>
<td>X | |

  | |

  | O |</td>
<td>X | | X

  | |

  | O |</td>
<td>X | O | X

  | |

  | O |</td>
</tr>
<tr><td>X’s third move</td><td>O’s third move</td><td>X’s forth move</td><td></td></tr>
<tr>
<td>X | O | X

  | X |

  | O |</td>
<td>X | O | X

  | X |

O | O |</td>
<td>X | O | X

  | X |

O | O | X</td>
<td>As can be seen to go first is a clear advantage</td>
</tr>
</table>

Multidimensional arrays

```
  Size_Ttt : constant := 3;
  subtype Board_Index is Integer range 1 .. Size_Ttt;
  type    Board_Array is
          array( Board_Index, Board_Index ) of Character;
  type Board is record
    Sqrs : Board_Array := ( others => (others => ' ') );
  end record;
 The: Board;
```

```
  The.Sqrs(1,2) := 'X';
  The.Sqrs(2,3) := 'X';
  The.Sqrs(3,2) := 'X';
```

```
  procedure Display( The:in Board ) is
  begin
    for I in Board_Array'Range(1) loop    --For each Row
      for J in Board_Array'Range(2) loop  -- For each column
        Put( The.Sqrs( I,J ) );           --  display counter;
        case J is                         --  column postfix
          when 1 | 2 => Put(" | ");
          when 3     => null;
        end case;
      end loop;
      case I is                           -- row postfix
        when 1 | 2  => New_Line; Put("----------"); New_Line;
        when 3      => New_Line;
      end case;
    end loop;
  end Display;
```

Alternative ways of declaring multidimensional arrays

```
Size_Ttt : constant := 3;
subtype Board_Index is Integer range 1 .. Size_Ttt;
type    Board_Row   is array( Board_Index ) of Character;
type    Board_Array is array( Board_Index ) of Board_Row ;
type Board is record
  Sqrs : Board_Array := ( others => (others => ' ') );
end record;
The: Board;
```

```
The.Sqrs(1)(2)  := 'X';
The.Sqrs(2)(3)  := 'X';
The.Sqrs(3)(2)  := 'X';
```

Initialising an array

```
type Colour      is ( Red, Green, Blue );
type Intensity   is range 0 .. 255;
type Pixel_Array is array( Colour ) of Intensity;
```

```
Dot          : Pixel_Array;
```

```
Dot := ( 0,    0,    0 );                         --Black
Dot := ( 255, 255, 255 );                         --White
```

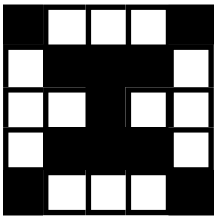
```
Dot := ( Red=> 255, Green=>255, Blue=>255);       --White
```

```
Dot := Pixel_Array'( Red=>255, others=>0 );       --Red
```

```
Dot := ( Red=>255, Green=>255, Blue=>0  );        --Yellow
Dot := ( Red | Blue => 255, Green=>0  );          --Purple
Dot := ( Red .. Blue => 127 );                    --Grey
```

Two dimensional initialisation

```
Bits_Cursor : constant Positive := 5;
type Bit    is new Integer range 0 .. 1;
type Cursor_Index is new Positive range 1 .. Bits_Cursor;
type Cursor       is array( Cursor_Index,
                            Cursor_Index ) of Bit;

Cursor_Style : Cursor;
```



Black is represented by 1

White is represented by 0

```
Cursor_Style := Cursor'( (1, 0, 0, 0, 1),
                         (0, 1, 1, 1, 0),
                         (0, 0, 1, 0, 0),
                         (0, 1, 1, 1, 0),
                         (1, 0, 0, 0, 1) );
```

```
Cursor_Style := Cursor'( 1=> ( 1=>1, 5=>1, others => 0 ),
                              2=> ( 2..4   =>1, others => 0 ),
                              3=> ( 3=>1,         others => 0 ),
                              4=> ( 2..4   =>1, others => 0 ),
                              5=> ( 1=>1, 5=>1, others => 0 ) );
```

```
Cursor_Style := Cursor'( 1|5=> ( 1|5 =>1, others => 0 ),
                         2|4=> ( 2..4=>1, others => 0 ),
                         3   => ( 3   =>1, others => 0 ) );
```

Array of Array

```
Bits_Cursor: constant Positive := 5;
type Colour       is ( Red, Green, Blue );
type Intensity    is range 0 .. 255;
type Pixel_Array is array( Colour ) of Intensity;

type Cursor_Index is new Positive range 1 .. Bits_Cursor;
type Cursor        is array( Cursor_Index,
                             Cursor_Index ) of Pixel_Array;

Cursor_Style : Cursor;
```

```
Cursor_Style :=
Cursor'( 1|5=> ( 1|5 => (others=>127), others => (others=>0) ),
         2|4=> ( 2..4=> (others=>127), others => (others=>0) ),
         3  => ( 3   => (others=>127), others => (others=>0) ) );
```

A histogram

Method	Responsibility
Add_To	Add a character to the histogram, recording the updated total number of characters.
Put	Write a histogram to the output source representing the currently gathered data.
Reset	Clear any previously gathered data, setting various internal objects to an initial state.

```
with Ada.Text_Io, Class_Histogram;
use  Ada.Text_Io, Class_Histogram;
procedure Main is
  Ch:Character;                       --Current character
  Text_Histogram: Histogram;          --Histogram object
begin
  Reset(Text_Histogram);              --Reset to empty

  while not End_Of_File loop          --For each line
    while not End_Of_Line loop        --For each character
      Get(Ch);                        --Get current character
      Add_To( Text_Histogram, Ch );   --Add to histogram
    end loop;
    Skip_Line;                        --Next line
  end loop;

  Put( Text_Histogram );              --Print histogram

end Main;
```

Specification

```
package Class_Histogram is
  type Histogram is private;
  Def_Height : constant Positive := 14;

  procedure Reset( The:in out Histogram );
  procedure Add_To( The:in out Histogram; A_Ch:in Character );
  procedure Put(The:in Histogram; Height:in Positive:=Def_Height);
private
  type    Alphabet_Index is new Character range 'A' .. 'Z';
  type    Alphabet_Array is array (Alphabet_Index) of Natural;

  type Histogram is record
    Number_Of    : Alphabet_Array := ( others => 0 );
  end record;
end Class_Histogram;
```

Implementation

```
with Ada.Text_Io, Ada.Float_Text_Io, Ada.Characters.Handling;
use  Ada.Text_Io, Ada.Float_Text_Io, Ada.Characters.Handling;
package body Class_Histogram is

  procedure Reset(The:in out Histogram) is
  begin
    The.Number_Of := ( others => 0 );   --Reset counts to 0
  end Reset;
```

```
  procedure Add_To(The:in out Histogram; A_Ch:in Character) is
    Ch : Character;
  begin
    Ch := A_Ch;                         --As write to ch
    if Is_Lower(Ch) then                --Convert to upper case
      Ch := To_Upper( Ch );
    end if;
    if Is_Upper( Ch ) then              --so record
      declare
        C : Alphabet_Index := Alphabet_Index(Ch);
      begin
        The.Number_Of(C) := The.Number_Of(C) + 1;
      end;
    end if;
  end Add_To;
```

```
  procedure Put(The:in Histogram;
                Height:in Positive:=Def_Height) is
    Frequency    : Alphabet_Array;          --Copy to process
    Max_Height   : Natural := 0;            --Observed max
  begin
    Frequency := The.Number_Of;             --Copy data (Array)
    for Ch in Alphabet_Array'Range loop     --Find max frequency
      if Frequency(Ch) > Max_Height then
        Max_Height:= Frequency(Ch);
      end if;
    end loop;

    if Max_Height > 0 then
      for Ch in Alphabet_Array'Range loop --Scale to max height
        Frequency(Ch):=(Frequency(Ch)*Height)/(Max_Height);
      end loop;
    end if;

    for Row in reverse 1 .. Height loop    --Each line
      Put( "   | " );                      --start of line
      for Ch in Alphabet_Array'Range loop
        if Frequency(Ch) >= Row then
          Put('*');                        --bar of hist >= col
        else
          Put(' ');                        --bar of hist <  col
        end if;
      end loop;
      Put(" | "); New_Line;                --end of line
    end loop;
    Put("   +----------------------------+"); New_Line;
    Put("     ABCDEFGHIJKLMNOPQRSTUVWXYZ " ); New_Line;
    Put("   *  = (approx) ");
    Put( Float(Max_Height) / Float(Height), Aft=>2, Exp=>0 );
    Put(" characters "); New_Line;
  end Put;
end Class_Histogram;
```

```
Ada is a language developed for the American department
of defense.
Ada is named after the first programmer Ada (Byron) Lovelace
who helped Charles Babbage with his work on the analytical engine.
She was the daughter of the poet Lord Byron.
```

```
|     *                      |
|     *                      |
|     *                      |
| *   *                      |
| *   *                      |
| *   *                      |
| *   *                      |
| *   *                      |
| *   *         *  * *       |
| *  **  *     **  * *       |
| *  **  *   * **  * *       |
| *  **  **  * **  ***       |
| ** ******  ***** ***       |
| *********  ***** ***  * *  |
+----------------------------+
  ABCDEFGHIJKLMNOPQRSTUVWXYZ     *  = 2.071
```

Unconstrained arrays

```
function Sum( List:in Numbers_Array ) return Integer is
  Total : Integer := 0;
begin
  for I in List'range loop          --Depends on # of elements
    Total := Total + List( I );
  end loop;
  return Total;
end Sum;
```

```
type Numbers_Array is array ( Positive range <> ) of Integer;
```

```
Computers_In_In_Room :Numbers_Array(513..519) := (2,2,2,3,2,1,3);
```

Unconstrained arrays: Example

```
package Pack_Types is
  type Numbers_Array is array ( Positive range <> ) of Integer;
end Pack_Types;
```

```
with Ada.Text_Io, Ada.Integer_Text_Io, Pack_Types;
use  Ada.Text_Io, Ada.Integer_Text_Io, Pack_Types;
procedure Main is
  Computers_In_Room :Numbers_Array(513..519) := (2,2,2,3,2,1,3);

  function Sum( List:in Numbers_Array ) return Integer is
    Total : Integer := 0;
  begin
    for I in List'range loop         --Depends on # of elements
      Total := Total + List( I );
    end loop;
    return Total;
  end Sum;

begin
  Put("The total number of computers is: " );
  Put( Sum( Computers_In_In_Room ) ); New_Line;

  Put("Computers in rooms 517, 518 and 519 is: " );
  Put( Sum( Computers_In_In_Room( 517 .. 519 ) ) ); New_Line;
end Main;
```

```
The total number of computers is:          15
Computers in rooms 517, 518 and 519 is:     6
```

Strings

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

```
procedure Main is
  type String is array ( Positive range <> ) of Character;
  Institution : String(1 .. 22);
  Address     : String(1 .. 20);
  Full_Address: String(1 .. 44);
begin
  Institution := "University of Brighton";
  Address     := "Brighton East Sussex";
  Full_Address:= Institution & ", " & Address;
  Put( Full_Address ); New_Line;
end Main;
```

```
University of Brighton, Brighton East Sussex
```

Dynamic arrays

```
function Reverse_String( Str:in String ) return String is
  Res : String( Str'Range );                  --Dynamic bounds
begin
   for I in Str'Range loop
     Res( Str'First+Str'Last-I ) := Str( I );
   end loop;
   return Res;
end Reverse_String;
```

```
procedure Main is
begin
  Put( Reverse_String( "madam i'm adam" ) ); New_Line;
end Main;
```

```
mada m'i madam
```

Name and address class

Method	Responsibility
Set	Set the name and address of a person. The name and address is specified with a / character separating each line.
Deliver_Line	Deliver the n'th line of the address as a string.
Lines	Deliver the number of lines in the address.

```
package Class_Name_Address is
  type Name_Address is private;

  procedure Set( The:out Name_Address; Str:in String );
  function Deliver_Line( The:in Name_Address;
    Line:in Positive ) return String;
  function Lines( The:in Name_Address ) return Positive;
private
  Max_Chs : constant := 200;
  subtype Line_Index   is Natural    range 0 .. Max_Chs;
  subtype Line_Range   is Line_Index range 1 .. Max_Chs;

  type Name_Address is record
    Text   : String( Line_Range );    --Details
    Length : Line_Index := 0;         --Length of address
  end record;
end Class_Name_Address;
```

```
package body Class_Name_Address is

  function Spaces( Line:in Positive ) return String;

  procedure Set( The:out Name_Address; Str:in String ) is
  begin
    if Str'Length > Max_Chs then
      Set( The, Str( Str'First .. Str'First+Max_Chs-1 ) );
    else
      The.Text( 1 .. Str'Length ) := Str;
      The.Length := Str'Length;
    end if;
  end Set;
```

```
function Deliver_Line( The:in Name_Address;
                       Line:in Positive ) return String is
                       Line_On : Positive := 1;
begin
  for I in 1 .. The.Length loop
    if Line_On = Line then
      for J in I .. The.Length loop
        if The.Text(J) = '/' then
          return Spaces(Line_On) & The.Text(I .. J-1);
        end if;
      end loop;
      return Spaces(Line_On) & The.Text(I..The.Length);
    end if;
    if The.Text(I) = '/' then Line_On := Line_On+1; end if;
  end loop;
  return "";
end Deliver_Line;
```

```
function Lines( The:in Name_Address ) return Positive is
  No_Lines : Positive := 1;
begin
  for I in  1 .. The.Length loop
    if The.Text(I) = '/' then No_Lines := No_Lines + 1; end if;
  end loop;
  return No_Lines;
end Lines;
```

```
  function Spaces( Line:in Positive ) return String is
    Spaces_Are : String( 1 .. Line ) := (others=>' ');
  begin
    return Spaces_Are;
  end Spaces;

end Class_Name_Address;
```

```
with Ada.Text_Io, Ada.Integer_Text_Io, Class_Name_Address;
use  Ada.Text_Io, Ada.Integer_Text_Io;
procedure main is
  Name    : Name_Address;
  Address : String := "A.N.Other/Brighton/East Sussex/UK";
begin
  Set( Name, Address );
  Put( Address ); New_Line; Put("There are ");
  Put( Lines( Name ) ); Put(" lines"); New_Line;
  for I in 1 .. Lines(Name)+1 loop
    Put("Line #"); Put(I); Put("  ");
    Put( Deliver_Line(Name, I) ); New_Line;
  end loop;
end Main;
```

```
A.N.Other/Brighton/East Sussex/UK
There are      4 lines
Line #     1  A.N.Other
Line #     2    Brighton
Line #     3      East Sussex
Line #     4        UK
Line #     5
```

An electronic piggy bank

Method	Responsibility
`deposit`	Deposit money into a named person's account.
`withdraw`	Withdraw money from a named person's account.
`balance`	Obtain the balance in a named person's account.
`statement`	Print a statement for a named account.

```
with Class_Account;
use  Class_Account;
package Class_Piggy_Bank is
  type Piggy_Bank is private;                   --Class
  subtype Money  is Class_Account.Money;        --Make visible
  subtype Pmoney is Class_Account.Pmoney;       --Make visible

  procedure New_Account( The:in out Piggy_Bank; No:out Positive );
  procedure Deposit   ( The:in out Piggy_Bank; No:in Positive;
                        Amount:in Pmoney );
  procedure Withdraw  ( The:in out Piggy_Bank; No:in Positive;
                        Amount:in Pmoney; Get:out Pmoney );
  function Balance( The:in Piggy_Bank;
                    No:in Positive) return Money;
  function Valid( The:in Piggy_Bank;
                  No:in Positive) return Boolean;

private
  No_Accounts : constant := 10;
  subtype Accounts_Index is Integer range 0 .. No_Accounts;
  subtype Accounts_Range is Accounts_Index range 1 .. No_Accounts;
  type    Accounts_Array is array ( Accounts_Range ) of Account;
  type Piggy_Bank is record
    Accounts: Accounts_Array;          --Accounts in the bank
    Last    : Accounts_Index := 0;     --Last account
  end record;
end Class_Piggy_Bank;
```

```
with Ada.Text_Io, Ada.Integer_Text_Io, Ada.Float_Text_Io,
     Class_Piggy_Bank;
use  Ada.Text_Io, Ada.Integer_Text_Io, Ada.Float_Text_Io,
     Class_Piggy_Bank;
procedure Statement( Bank:in Piggy_Bank; No:in Positive ) is
  In_Account : Money;
begin
  Put("Mini statement for account #");
  Put( No, Width=>3 ); New_Line;
  Put( "The amount on deposit is £" );
  In_Account := Balance( Bank, No );
  Put( In_Account, Aft=>2, Exp=>0 );
  New_Line(2);
end Statement;
```

```
with Ada.Text_io, Class_Piggy_Bank, Statement;
use  Ada.Text_io, Class_Piggy_Bank;
procedure Main is
  Bank_Accounts: Piggy_Bank;          --A little bank
  Customer     : Positive;            --Customer
  Obtain       : Money;               --Money processed
begin
  New_Account( Bank_Accounts, Customer );
  if Valid( Bank_Accounts, Customer ) then
    Statement( Bank_Accounts, Customer );

    Put("Deposit £100.00 into account"); New_Line;
    Deposit( Bank_Accounts, Customer, 100.00 );
    Statement( Bank_Accounts, Customer );

    Put("Withdraw £60.00 from account"); New_Line;
    Withdraw( Bank_Accounts, Customer, 60.00, Obtain );
    Statement( Bank_Accounts, Customer );

    Put("Deposit £150.00 into account"); New_Line;
    Deposit( Bank_Accounts, Customer, 150.00 );
    Statement( Bank_Accounts, Customer );
  else
    Put("Customer number not valid"); New_Line;
  end if;
end Main;
```

```
Mini statement for account #  1
The amount on deposit is £ 0.00

Deposit £100.00 into account
Mini statement for account #  1
The amount on deposit is £100.00

Withdraw £80.00 from account
Mini statement for account #  1
The amount on deposit is £20.00

Deposit £200.00 into account
Mini statement for account #  1
The amount on deposit is £220.00
```

```
package body Class_Piggy_Bank is

  procedure New_Account(The:in out Piggy_Bank; No:out Positive) is
  begin
    if The.Last = No_Accounts then
      raise Constraint_Error;
    else
      The.Last := The.Last + 1;
    end if;
    No := The.Last;
  end New_Account;

 procedure Deposit ( The:in out Piggy_Bank; No:in Positive;
                     Amount:in Pmoney ) is
  begin
    Deposit( The.Accounts(No), Amount );
  end Deposit;

  procedure Withdraw( The:in out Piggy_Bank; No:in Positive;
                     Amount:in Pmoney; Get:out Pmoney ) is
  begin
    Withdraw( The.Accounts(No), Amount, Get);
  end Withdraw;

  function Balance( The:in Piggy_Bank;
                   No:in Positive) return Money is
  begin
    return Balance( The.Accounts(No) );
  end Balance;

  function Valid( The:in Piggy_Bank;
                 No:in Positive) return Boolean is
  begin
    return No in 1 .. The.Last;
  end Valid;
end Class_Piggy_Bank;
```

Dynamic arrays

```
function Reverse_String( Str:in String ) return String is
  Res : String( Str'Range );                  --Dynamic bounds
begin
    for I in Str'Range loop
      Res( Str'First+Str'Last-I ) := Str( I );
    end loop;
    return Res;
end Reverse_String;
```

```
with Ada.Text_Io, reverse_string; use Ada.Text_Io;
procedure Main is
begin
  Put( Reverse_String( "madam i'm adam" ) ); New_Line;
end Main;
```

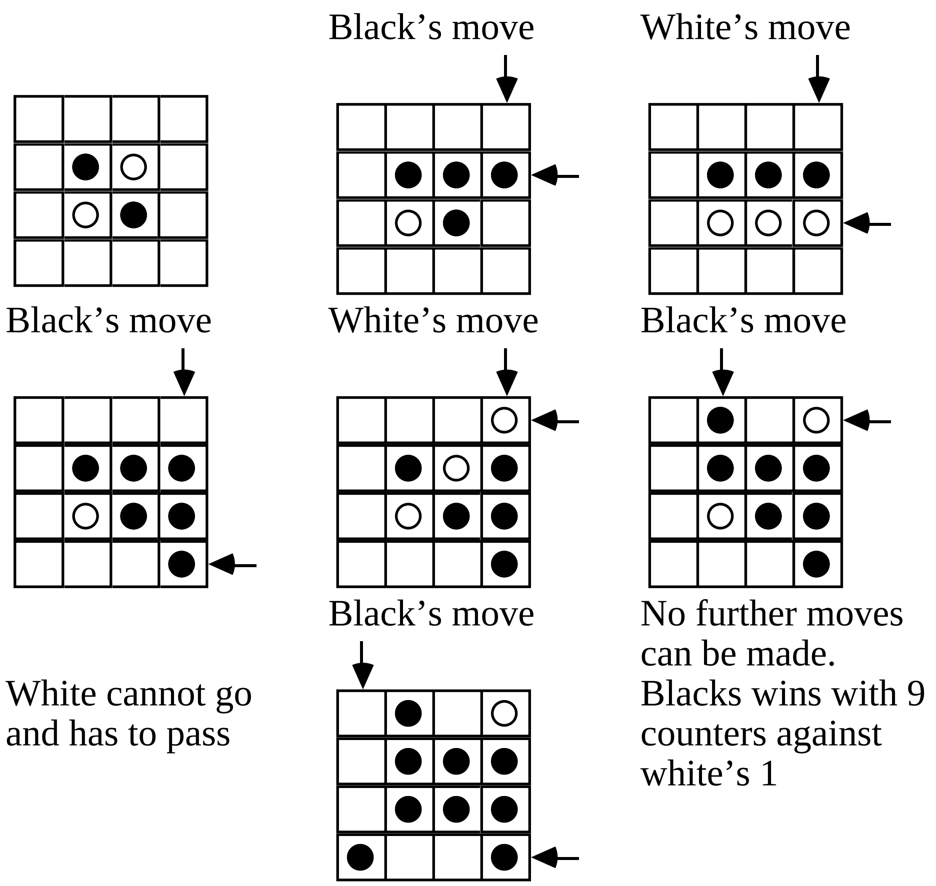
```
mada m'i madam
```

Reversi

In the game of reversi two players take it in turn to add counters to a board of 8-by-8 cells. Each player has a stack of counters black one side and white the other. One player plays his counters white side up whilst the other player plays his counters black side up. The object of the game is to capture all your opponent's counters. You do this by adding one of your counters to the board so that your opponent's counter(s) are flanked by two of your counters. When you do this, the counters you have captured are flipped over to become your counters. If you can't capture any of your opponent's counters during you turn, you must pass and let your opponent go.

The game is won when you have captured all your opponent's counters. If neither player can add a counter to the board, then the player with the most counters wins. If the number of counters for each player is equal, then the game is a draw.

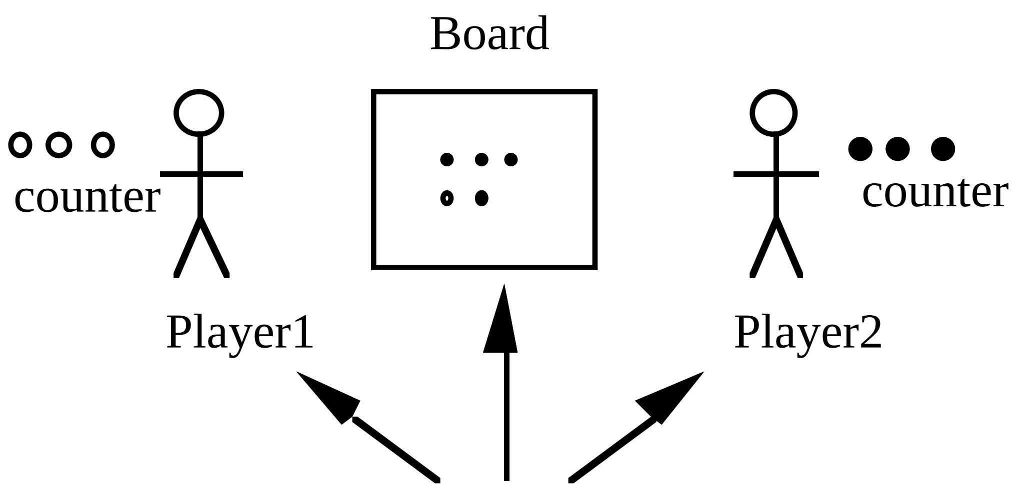
The initial starting position is set so that the 4 centre squares in the 8-by-8 board of cells is as follows:



In the game of reversi two **players** take it in turn to ***add*** **counters** to a **board** of 8-by-8 **cells**. Each **player** has a stack of **counters** black one side and white the other. One **player** plays his **counters** white side up whilst the other **player** plays his counters black side up. The object of the game is to capture all your opponent's **counters**. You do this by adding one of your counters to the **board** so that your opponent's **counter(s)** are flanked by two of your **counters**. When you do this, the **counters** you have captured are ***flipped*** over to become your **counters**. If you can't capture any of your opponent's **counters** during you turn, you must pass and let your opponent go.

The game is won when you have captured all your opponent's counters. If neither **player** can add a counter to the **board**, then the **player** with the most **counters** wins. If the number of counters for each **player** is equal, then the game is a draw.

A controller of the **game** (games master) ***asks*** each player in turn for a move. When a move is received from a **player** the **board** is asked to ***validate*** the move. If this is a valid move the **counter** of the current **player** is ***added*** to the **board**. The new state of the **board** is then ***displayed***. This process is repeated until either the **board** is filled or neither **player** can make a move. The **player** making the last move is asked to ***announce*** the result of the **game**.



Objects (nouns)	**System actions (verbs)**
board game cell counter player	add announce ask evaluated display validate

The following messages are sent to individual objects:

`board`
  Display a representation of the board.
  Add a counter to the board.
  Evaluate the current state of the board.
  Validate a proposed move.

`player`
  Announce the result of the game.
  Ask for the next move.

`cell`
  Add a counter into a cell on the board.

`counter`
  Display a representation of the counter.

Play
  Play the game

Refined

Class	Message	Responsibility of method
Board	`Add`	Add a counter into the board..
	`Check_Move`	Check if a player can drop a counter into a column.
	`Contents`	Return the contents of a cell.
	`Display`	Display a representation of the board.
	`Now_playing`	Say who is now playing on the board.
	`Set_Up`	Populate the board with the initial contents.
	`Status`	Evaluate the current state of the board.
Player	`Announce`	Announcing that the player has either won or drawn the game.
	`Get_Move`	Get the next move from the player.
	`My_Counter`	Return the counter that the player plays with.
	`Set`	Set a player to play with a particular counter.
Cell	`Add`	Add a counter to a cell
	`Display`	Display the contents of a cell.
	`Flip`	Flip the contents of a cell.
	`Holds`	Return the contents of a cell.
	`Initialize`	Initialize a cell
Counter	`Display`	Display a counter
	`Flip`	Flip a counter
	`Rep`	Return the colour of a counter.
	`Set`	Set a counter to be black/white.
Game	`Play`	Play the game

Specification of the Ada classes

The Ada class specifications for the above classes are implemented as follows:

Class	Ada specification

Game	`package Class_Game is`
	`   type Game        is private;`
	`   procedure Play( The:in Game );`
	`private`
	
	`end Class Counter;`

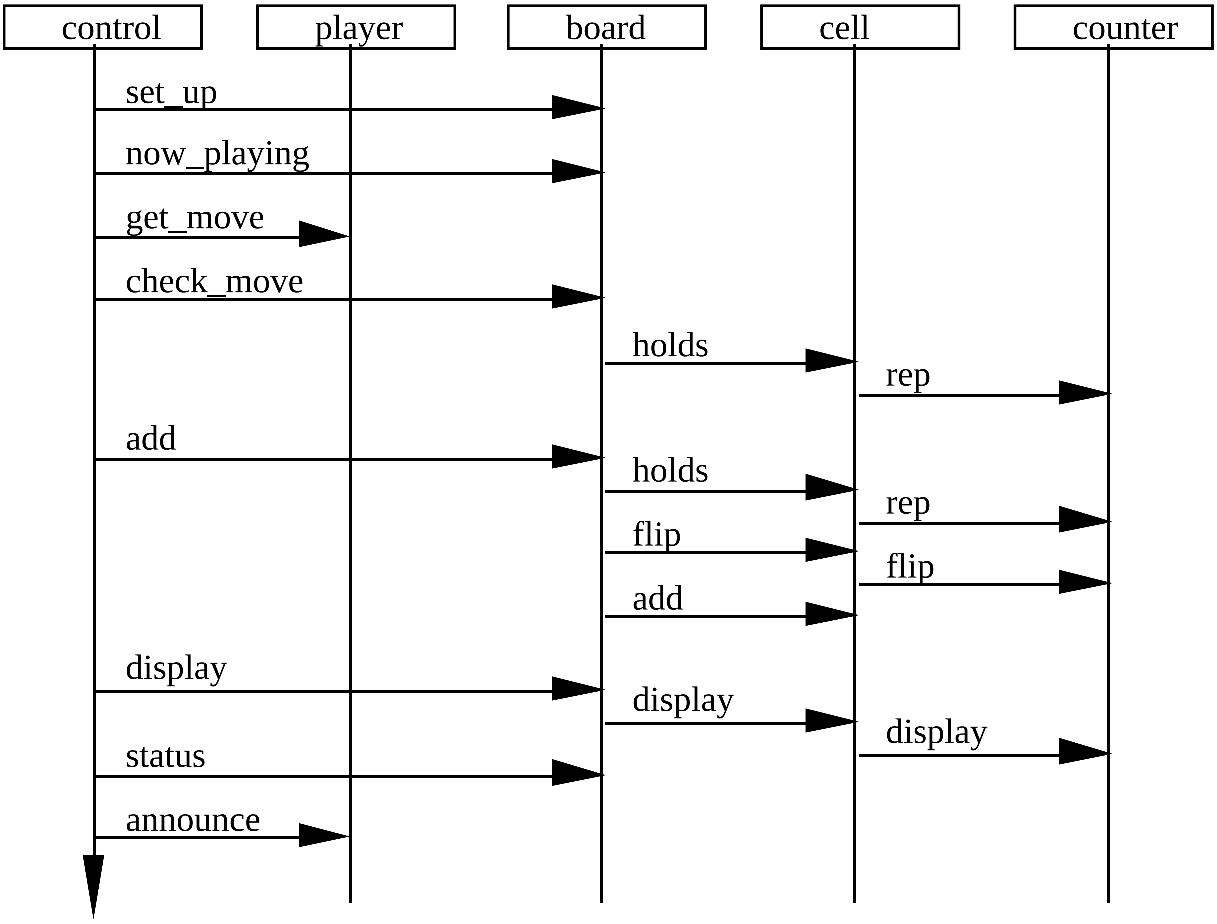
Counter	`package Class_Counter is`
	`   type Counter         is private;`
	`   type Counter_Colour is ( Black, White );`
	`   procedure Set( The:in out Counter; Rep:in Counter_Colour );`
	`   procedure Display( The:in Counter );`
	`   procedure Display_None( The:in Counter );`
	`   procedure Flip( The:in out Counter );`
	`   function  Rep( The:in Counter ) return Counter_Colour;`
	`private`
	
	`end Class Counter;`

Player	`package Class_Player is`
	`   type Player is private;`
	
	`   procedure Set( The:in out Player; C:in Counter_Colour );`
	`   procedure Get_Move(The:in Player; Row,Column:out Integer);`
	`   function  My_Counter( The:in Player ) return Counter;`
	`   procedure Announce( The:in Player; What:in State_Of_Game );`
	`private`
	
	`end Class Player;`

Cell	`package Class_Cell is`
	`   type Cell is private;`
	`   type Cell_Holds is ( C_White, C_Black, Empty );`
	
	`   procedure Initialize( The:in out Cell );`
	`   function  Holds( The:in Cell ) return Cell_Holds;`
	`   procedure Add( The:in out Cell; Players_Counter:in Counter );`
	`   procedure Display( The:in Cell );`
	`   procedure Flip( The:in out Cell );`
	`   function To_Colour( C:in Cell_Holds ) return Counter_Colour;`
	`private`
	
	`end Class Cell;`

Board

```
package Class_Board is

   type Board          is private;
   type State_Of_Game  is ( Play, Win, Draw, Lose );
   type Move_Status    is ( Ok, Invalid, Pass );

   procedure Set_Up( The:in out Board );
   procedure Add( The:in out Board; X,Y:in Integer;
                  Move_Is:in Move_Status );
   procedure Now_Playing( The:in out Board; C:in Counter_Colour );
   procedure Display( The:in Board );
   function  Check_Move( The:in Board; X,Y:in Integer )
                      return Move_Status;
   function  Status( The:in Board ) return State_Of_Game;
   function  Contents( The:in Board; X,Y:in Integer )
                      return Cell_Holds;
private

end Class_Board;
```


control
player
board
cell
counter
set_up
now_playing
get_move
check_move
holds
rep
add
holds
rep
flip
flip
add
display
display
display
status
announce

```ada
with Class_Board, Class_Player, Class_Counter;
use  Class_Board, Class_Player, Class_Counter;
package Class_Game is
  type Game is private;
  procedure play( The:in out Game );
private
  type Player_Array is array(Counter_Colour) of Player;
  type Game is record
    Reversi     : Board;                        --The playing board
    Contestant : Player_Array;
  end record;
end Class_Game;
```

```ada
package body Class_Game is

procedure Play( The:in out Game ) is          --Play reversi
  Current_State : State_Of_Game;              --State of game
  Person        : Counter_Colour;             --Current player
  X, Y          : Integer;                    --Move
  Move_Is       : Move_Status;                --Last move is
begin
  Set_Up( The.Reversi );                      --Set up board
  Set( The.Contestant(Black), Black );        --Set player black
  Set( The.Contestant(White), White );        --Set player white

  Current_State := Play;  Person := Black;    --Black starts

  Display( The.Reversi );                     --Initial board

  while Current_State = Play loop             --Playable game
    Now_Playing( The.Reversi, Person );       --set player

    loop                                      --Get move
      Get_Move(The.Contestant(Person), X, Y);
      Move_Is:=Check_Move(The.Reversi, X, Y);--Validate
      exit when Move_Is=Ok or Move_Is=Pass;   --OK
    end loop;

    Add( The.Reversi, X, Y, Move_Is );        --Add move to board

    Display( The.Reversi );                   --Display new board
    Current_State := Status( The.Reversi );   --State of play is

    if Current_State = Play then              --Is still playable
      case Person is                          --next player
        when Black  => Person := White;
        when White  => Person := Black;
      end case;
```

```
    end if;

  end loop;                                    --Next move

  Announce( The.Contestant(Person), Current_State );  --Result

end Play;

end Class_Game;
```

```
with Class_Game;
use  Class_Game;
procedure Main is
  A_Game : Game;
begin
  Play( A_Game );
end Main;
```

```
---------------------------------
|   |   |   |   |   |   |   |   |
---------------------------------
|   |   |   |   |   |   |   |   |
---------------------------------
|   |   |   |   |   |   |   |   |
---------------------------------
|   |   |   | X | O |   |   |   |
---------------------------------
|   |   |   | O | X |   |   |   |
---------------------------------
|   |   |   |   |   |   |   |   |
---------------------------------
|   |   |   |   |   |   |   |   |
---------------------------------
|   |   |   |   |   |   |   |   |
---------------------------------

Player X has 2 counters -
Player O has 2 counters
Please enter move X row column: 4 6
```

```
---------------------------------
|   |   |   |   |   |   |   |   |
---------------------------------
|   |   |   |   |   |   |   |   |
---------------------------------
|   |   |   |   |   |   |   |   |
---------------------------------
|   |   |   | X | X | X |   |   |
---------------------------------
|   |   |   | O | X |   |   |   |
---------------------------------
|   |   |   |   |   |   |   |   |
---------------------------------
|   |   |   |   |   |   |   |   |
---------------------------------
|   |   |   |   |   |   |   |   |
---------------------------------

Player X has 4 counters -
Player O has 1 counters
Please enter move O row column: 5 6
```

```
---------------------------------
|   |   |   |   |   |   |   |   |
---------------------------------
|   |   |   |   |   |   |   |   |
---------------------------------
|   |   |   |   |   |   |   |   |
---------------------------------
|   |   |   | X | X | X |   |   |
---------------------------------
|   |   |   | O | O | O |   |   |
---------------------------------
|   |   |   |   |   |   |   |   |
---------------------------------
|   |   |   |   |   |   |   |   |
---------------------------------
|   |   |   |   |   |   |   |   |
---------------------------------

Player X has 3 counters -
Player O has 3 counters
Please enter move O row column: 6 6
```

```
---------------------------------
|   |   |   |   |   |   |   |   |
---------------------------------
|   |   |   |   |   |   |   |   |
---------------------------------
|   |   |   |   |   |   |   |   |
---------------------------------
|   |   |   | X | X | X |   |   |
---------------------------------
|   |   |   | O | X | X |   |   |
---------------------------------
|   |   |   |   |   | X |   |   |
---------------------------------
|   |   |   |   |   |   |   |   |
---------------------------------
|   |   |   |   |   |   |   |   |
---------------------------------

Player X has 6 counters -
Player O has 1 counters
Please enter move O row column: 0 0
```

```
---------------------------------
|   |   |   |   |   |   |   |   |
---------------------------------
|   |   |   |   |   |   |   |   |
---------------------------------
|   |   |   |   |   |   |   |   |
---------------------------------
|   |   |   | X | X | X |   |   |
---------------------------------
|   |   |   | O | X | X |   |   |
---------------------------------
|   |   |   |   |   | X |   |   |
---------------------------------
|   |   |   |   |   |   |   |   |
---------------------------------
|   |   |   |   |   |   |   |   |
---------------------------------

Player X has 6 counters -
Player O has 1 counters
Please enter move X row column: 6 4
```

```
---------------------------------
|   |   |   |   |   |   |   |   |
---------------------------------
|   |   |   |   |   |   |   |   |
---------------------------------
|   |   |   |   |   |   |   |   |
---------------------------------
|   |   |   | X | X | X |   |   |
---------------------------------
|   |   |   | X | X | X |   |   |
---------------------------------
|   |   |   | X |   | X |   |   |
---------------------------------
|   |   |   |   |   |   |   |   |
---------------------------------
|   |   |   |   |   |   |   |   |
---------------------------------

Player X has 8 counters -
Player O has 0 counters
Player X has won
```

```
package Pack_Screen is
  procedure Screen_Clear;                       --Home clear screen
  procedure Screen_Home;                        --Home no clear screen
private
  Esc: constant Character := Character'Val(27);
end Pack_Screen;
```

```
with Text_Io; use Text_Io;
package body Pack_Screen is                    --Terminal dependent I/O
  procedure Screen_Clear is                    --Clear screen
  begin
    Put( Esc & "[2J" );                        --Escape sequence
  end Screen_Clear;
  procedure Screen_Home is                     --Home
  begin
    Put( Esc & "[0;0H");                       --Escape sequence
  end Screen_Home;
end Pack_Screen;
```

```
package Class_Counter is
  type Counter          is private;
  type Counter_Colour is ( Black, White );
  procedure Set( The:in out Counter; Rep:in Counter_Colour );
  procedure Display( The:in Counter );
  procedure Display_None( The:in Counter );
  procedure Flip( The:in out Counter );
  function  Rep( The:in Counter ) return Counter_Colour;
private
  type Counter is record
    Colour: Counter_Colour;            --Colour of counter
  end record;
end Class_Counter;
```

```
with Ada.Text_Io;
use  Ada.Text_Io;
package body Class_Counter is
  procedure Set( The:in out Counter; Rep:in Counter_Colour ) is
  begin
    The.Colour := Rep;
  end Set;
```

```
  procedure Display( The:in Counter ) is
  begin
    case The.Colour is
      when Black  => Put('X'); --Representation of a black piece
      when White  => Put('O'); --Representation of a white piece
    end case;
  end Display;


  procedure Display_None( The:in Counter ) is
  begin
    Put(' ');                 --Representation of NO piece
  end Display_None;
```

```
  procedure Flip( The:in out Counter ) is
  begin
    case The.Colour is
      when Black  => The.Colour := White;      --Flip to White
      when White  => The.Colour := Black;      --Flip to Black
    end case;
  end Flip;

  function  Rep( The:in Counter ) return Counter_Colour is
  begin
    return The.Colour;  --Representation of the counter colour
  end Rep;
end Class_Counter;
```

Counter

```
with Class_Counter;
use  Class_Counter;
package Class_Cell is
  type Cell is private;
  type Cell_Holds is ( C_White, C_Black, Empty );

  procedure Initialize( The:in out Cell );
  function  Holds( The:in Cell ) return Cell_Holds;
  procedure Add( The:in out Cell; Players_Counter:in Counter );
  procedure Display( The:in Cell );
  procedure Flip( The:in out Cell );
  function To_Colour( C:in Cell_Holds ) return Counter_Colour;
private
  type Cell_Is is ( Empty_Cell, Not_Empty_Cell );
  type Cell is record
    Contents: Cell_Is := Empty_Cell;
    Item    : Counter;                 --The counter
  end record;
end Class_Cell;
```

```
package body Class_Cell is
  procedure Initialize( The:in out Cell ) is
  begin
    The.Contents := Empty_Cell;    --Initialize cell to empty
  end Initialize;
```

```
function  Holds( The:in Cell ) return Cell_Holds is
begin
  case The.Contents is
    when Empty_Cell     =>              --Empty
      return Empty;                     -- No counter
    when Not_Empty_Cell =>              --Counter
      case Rep( The.Item ) is
        when White => return C_White;   -- white counter
        when Black => return C_Black;   -- black counter
      end case;
  end case;
end Holds;
```

```
procedure Add(The:in out Cell; Players_Counter:in Counter) is
begin
  The := (Not_Empty_Cell,Players_Counter);
end Add;

procedure Display( The:in Cell ) is
begin
  if The.Contents = Not_Empty_Cell then
    Display( The.Item );            --Display the counter
  else
    Display_None( The.Item );       --No counter
  end if;
end Display;

procedure Flip( The:in out Cell ) is
begin
  Flip( The.Item );                 --Flip counter
end Flip;
```

```
  function To_Colour(C:in Cell_Holds) return Counter_Colour is
  begin
    case C is                       --Conversion of enum.
      when C_White => return White;
      when C_Black => return Black;
      when others   => raise Constraint_Error;
    end case;
  end To_Colour;

end Class_Cell;
```

Board

```
with Class_Counter, Class_Cell;
use  Class_Counter, Class_Cell;
package Class_Board is

  type Board          is private;
  type State_Of_Game is ( Play, Win, Draw, Lose );
  type Move_Status   is ( Ok, Invalid, Pass );

  procedure Set_Up( The:in out Board );
  procedure Add( The:in out Board; X,Y:in Integer;
                 Move_Is:in Move_Status );
  procedure Now_Playing( The:in out Board; C:in Counter_Colour );
  procedure Display( The:in Board );
  function  Check_Move( The:in Board; X,Y:in Integer )
                      return Move_Status;
  function  Status( The:in Board ) return State_Of_Game;
  function  Contents( The:in Board; X,Y:in Integer )
                    return Cell_Holds;

private
  Size: constant := 8;                              --8 * 8 Board
  subtype Board_Index is Integer range 1 .. Size; --

  type Board_Array is array (Board_Index, Board_Index) of Cell;
  type Score_Array is array (Counter_Colour) of Natural;
  type Move_Array  is array (Counter_Colour) of Move_Status;

  type Board is record
    Sqrs      : Board_Array;                  --Game board
    Player    : Counter_Colour;               --Current Player
    Opponent  : Counter_Colour;               --Opponent
    Score     : Score_Array;                  --Running score
    Last_Move: Move_Array;                    --Last move is
  end record;
end Class_Board;
```

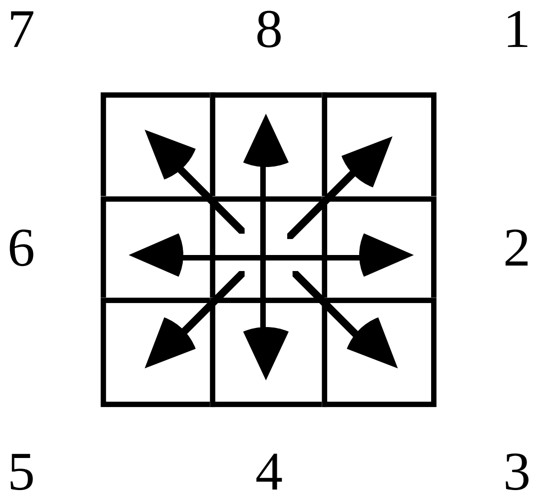
```
with Ada.Text_Io, Ada.Integer_Text_Io, Pack_Screen;
use  Ada.Text_Io, Ada.Integer_Text_Io, Pack_Screen;
package body Class_Board is

  procedure Next( The:in Board; X_Co,Y_Co:in out Board_Index;
                  Dir:in Natural; Res:out Boolean);
  function  Find_Turned( The:in Board; X,Y: in Board_Index )
                        return Natural;
  procedure Turn_Counters(The: in out Board; X,Y: in Board_Index;
                          Total: out Natural );
  function  No_Turned(The:in Board; O_X,O_Y:in Board_Index;
                      Dir:in Natural;
                      N:in Natural := 0 ) return Natural;
  procedure Capture(The:in out Board; X_Co, Y_Co:in Board_Index;
                    Dir:in Natural );
```

```
  procedure Set_Up( The:in out Board ) is
    Black_Counter: Counter;                   --A black counter
    White_Counter: Counter;                   --A white counter
  begin
    Set( Black_Counter, Black );              --Set black
    Set( White_Counter, White );              --Set white
    for X in The.Sqrs'range(1) loop
      for Y in The.Sqrs'range(2) loop
        Initialize( The.Sqrs(X,Y) );          --To empty
      end loop;
    end loop;
    Add( The.Sqrs( Size/2,   Size/2 ),   Black_Counter );
    Add( The.Sqrs( Size/2,   Size/2+1 ), White_Counter );
    Add( The.Sqrs( Size/2+1, Size/2 ),   White_Counter );
    Add( The.Sqrs( Size/2+1, Size/2+1 ), Black_Counter );
    The.Score( Black ) := 2; The.Score( White ) := 2;
  end Set_Up;
```

```
  procedure Now_Playing(The:in out Board; C:in Counter_Colour) is
  begin
    The.Player   := C;                           --Player
    case C is                                    --Opponent
      when White => The.Opponent := Black;
      when Black => The.Opponent := White;
    end case;
  end Now_Playing;
```

```
procedure Display( The:in Board ) is
  Dashes: String( 1 .. The.Sqrs'Length*4+1 ) := (others=>'-');
begin
  Screen_Clear;                                --Clear screen
  Put( Dashes ); New_Line;                     --Top
  for X in The.Sqrs'range(1) loop
    Put("|");                                  --Cells on line
    for Y in The.Sqrs'range(2) loop
      Put(" "); Display( The.Sqrs(X,Y) ); Put(" |");
    end loop;
    New_Line; Put( Dashes ); New_Line;         --Bottom lines
  end loop;
  New_Line;
  Put( "Player X has " );
  Put( Integer(The.Score(Black)), Width=>2 );
  Put( " counters" ); New_Line;
  Put( "Player O has " );
  Put( Integer(The.Score(White)), Width=>2 );
  Put( " counters" ); New_Line;
end Display;
```

```
function Check_Move( The:in Board; X,Y:in Integer )
    return Move_Status is
begin
  if X = 0 and then Y = 0 then
    return Pass;
  elsif X in Board_Index and then Y in Board_Index then
    if Holds( The.Sqrs( X, Y ) ) = Empty then
      if Find_Turned(The, X, Y) > 0 then
        return Ok;
      end if;
    end if;
  end if;
  return Invalid;
end Check_Move;
```




```
function Find_Turned( The:in Board; X,Y: in Board_Index )
    return Natural is
  Sum     : Natural := 0;          --Total stones turned
begin
  if Holds( The.Sqrs( X, Y ) ) = Empty then
    for Dir in 1 .. 8 loop         --The  8 possible directions
      Sum := Sum + No_Turned( The, X, Y, Dir );
    end loop;
  end if;
  return Sum;                      --return total
end Find_Turned;
```

```
function  No_Turned(The:in Board; O_X,O_Y:in Board_Index;
    Dir:in Natural;
    N:in Natural := 0 ) return Natural is
  Ok : Boolean;                          --Result from next
  Nxt: Cell_Holds;                       --Next in line is
  Col: Counter_Colour;                   --Counter colour
  X  : Board_Index := O_X;               --Local copy
  Y  : Board_Index := O_Y;               --Local copy
begin
  Next( The, X,Y, Dir, Ok );             --Next cell
  if Ok then                             --On the board
    Nxt := Holds( The.Sqrs(X,Y) );       --Contents are
    if Nxt = Empty then                  --End of line
      return 0;
    else
      Col := To_Colour( Nxt );           --Colour
      if Col = The.Opponent then         --Opponents counter
        return No_Turned(The, X,Y, Dir, N+1); --Try next cell
      elsif Col = The.Player then        --End of counters
        return N;                        --Counters turned
      end if;
    end if;
  else
    return 0;                            --No line
  end if;
end No_Turned;
```

```
procedure Next( The:in Board; X_Co,Y_Co:in out Board_Index;
    Dir:in Natural; Res:out Boolean) is
  X, Y   : Natural;
begin
  X := X_Co; Y := Y_Co;          --May go outside Board_range
  case Dir is
    when 1 =>         Y:=Y+1;    -- Direction to move
    when 2 => X:=X+1; Y:=Y+1;    --      8   1   2
    when 3 => X:=X+1;            --
    when 4 => X:=X+1; Y:=Y-1;    --      7   *   3
    when 5 =>         Y:=Y-1;    --
    when 6 => X:=X-1; Y:=Y-1;    --      6   5   4
    when 7 => X:=X-1;            --
    when 8 => X:=X-1; Y:=Y+1;    --
    when others => raise Constraint_Error;
  end case;
  if X in Board_Index and then Y in Board_Index then
    X_Co :=  X; Y_Co := Y;       --
    Res := True;                 --Found a next cell
  else
    Res := False;                --No next cell
  end if;
end Next;
```

```
procedure Add( The:in out Board; X,Y:in Integer;
               Move_Is:in Move_Status ) is
  Plays_With: Counter;          --Current player's counter
  Turned    : Natural;          --Number counters turned
begin
  Set( Plays_With, The.Player );              --Set current colour
  The.Last_Move( The.Player ) := Move_Is;     --Last move is
  if Move_Is = Ok then                        --Not Pass
    Turn_Counters(The, X,Y, Turned);          --and flip
    Add( The.Sqrs( X, Y ), Plays_With );      --to board
    The.Score( The.Player ) :=
      The.Score( The.Player ) + Turned + 1;
    The.Score( The.Opponent ):=
      The.Score( The.Opponent ) - Turned;
  end if;
end Add;
```

```
procedure Turn_Counters(The: in out Board; X,Y: in Board_Index;
                        Total: out Natural ) is
  Num_Cap  : Natural := 0;
  Captured : Natural;
begin
  if Holds( The.Sqrs( X, Y ) ) = Empty then
    for Dir in 1 .. 8 loop
      Captured := No_Turned( The, X, Y, Dir );
      if Captured > 0 then
        Capture( The, X, Y, Dir );
        Num_Cap := Num_Cap + Captured;
      end if;
    end loop;
  end if;
  Total := Num_Cap;
end Turn_Counters;
```

```
procedure Capture(The:in out Board; X_Co, Y_Co:in Board_Index;
                  Dir:in Natural ) is
  Ok   : Boolean;                    --There is a next cell
  X, Y : Board_Index;                --Coordinates of cell
  Nxt  : Cell_Holds;                 --Next in line is
begin
  X := X_Co; Y := Y_Co;
  Next( The, X, Y, Dir, Ok );        --Calculate pos next cell
  if Ok then                         --Cell exists (Must)
    Nxt := Holds( The.Sqrs(X,Y) );
    if To_Colour( Nxt ) = The.Opponent then
      Flip( The.Sqrs(X, Y) );        --Capture
      Capture(The, X, Y, Dir );      --Implement capture
    else
      return;                        --End of line
    end if;
  else
    raise Constraint_Error;          --Will never occur
  end if;
end Capture;
```

```
function  Status ( The:in Board ) return State_Of_Game is
begin
  if The.Score( The.Opponent ) = 0 then
    return Win;
  end if;
  if (The.Sqrs'Length(1) * The.Sqrs'Length(2) =
      The.Score(The.Opponent)+The.Score(The.Player)) or
      (The.Last_Move(Black)=Pass and The.Last_Move(White)=Pass)
      then
    if The.Score(The.Opponent) = The.Score(The.Player)
        then return Draw;
    end if;
    if The.Score(The.Opponent) < The.Score(The.Player)
        then return Win;
    else
      return Lose;
    end if;
  end if;
  return Play;
end;
```

```
function Contents( The:in Board; X,Y:in Integer )
                   return Cell_Holds is
begin
  return Holds( The.Sqrs( X, Y ) );
end Contents;

end Class_Board;
```

```
with Class_Counter, Class_Board;
use  Class_Counter, Class_Board;
package Class_Player is
  type Player is private;

  procedure Set( The:in out Player; C:in Counter_Colour );
  procedure Get_Move(The:in Player; Row,Column:out Integer);
  function  My_Counter( The:in Player ) return Counter;
  procedure Announce( The:in Player; What:in State_Of_Game );
private
  type Player is record
    Plays_With : Counter;         --Player's counter
  end record;
end Class_Player;
```

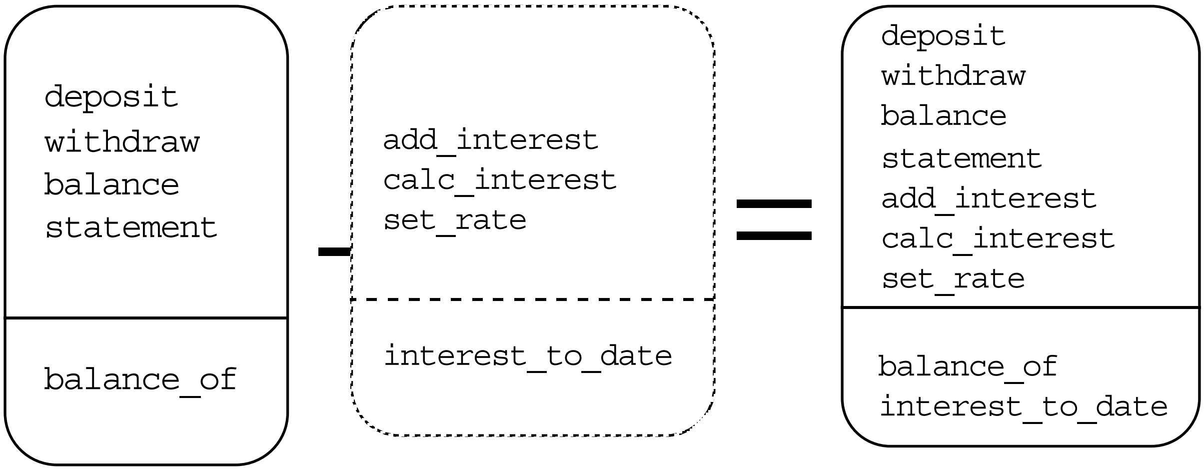
```
with Ada.Text_Io, Ada.Integer_Text_Io;
use  Ada.Text_Io, Ada.Integer_Text_Io;
package body Class_Player is
  procedure Set(The:in out Player; C:in Counter_Colour ) is
    A_Counter : Counter;
  begin
    Set( A_Counter, C );          --Set colour
    The.Plays_With := A_Counter;  --Player is playing with
  end Set;
```

```
  procedure Get_Move(The:in Player; Row,Column:out Integer) is
    Valid_Move : Boolean := False;
  begin
    while not Valid_Move loop
      begin
        Put("Please enter move "); Display( The.Plays_With );
        Put(" row column : "); Get( Row ); Get( Column );
        Valid_Move := True;
      exception
        when Data_Error =>
          Row := -1; Column := -1; Skip_Line;
        when End_Error =>
          Row := 0; Column := 0;
          return;
      end;
    end loop;
  end Get_Move;
```

```
  function  My_Counter( The:in Player ) return Counter is
  begin
    return The.Plays_With;
  end My_Counter;
```

```
  procedure Announce(The:in Player; What:in State_Of_Game) is
  begin
    case What is
      when Win    =>
        Put("Player "); Display( The.Plays_With );
        Put(" has won");
      when Lose   =>
        Put("Player "); Display( The.Plays_With );
        Put(" has lost");
      when Draw   =>
        Put("It's a draw");
      when others =>
        raise Constraint_Error;
    end case;
    New_Line;
  end Announce;

end Class_Player;
```

Inheritance

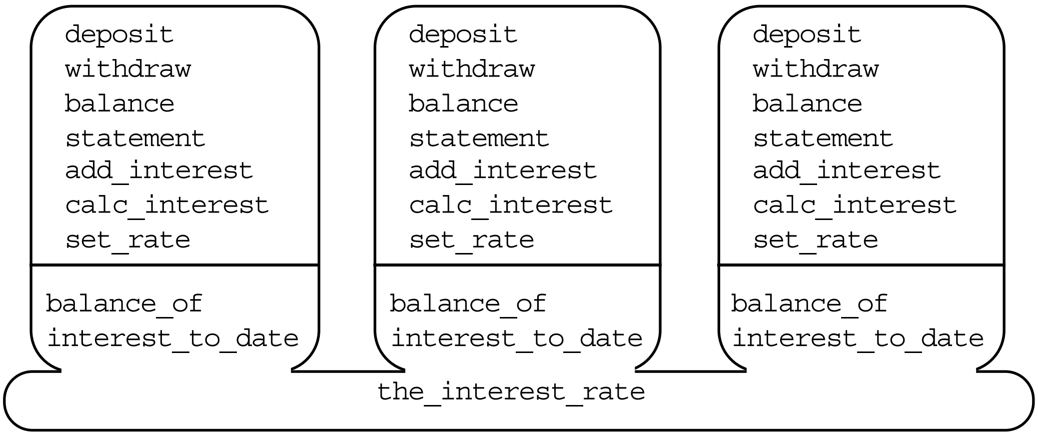



```
package Class_Account is
  type Account is tagged private;
  subtype Money  is Float;
  subtype Pmoney is Float range 0.0 .. Float'Last;

procedure Deposit( The:in out Account; Amount:in Pmoney );
  procedure Withdraw( The:in out Account;
                      Amount:in Pmoney; Get:out Pmoney );
  function  Balance( The:in Account ) return Money;
private
  type Account is tagged record
    Balance_Of : Money := 0.00;           --Amount on deposit
  end record;
end Class_Account;
```

base class	A class from which other classes are derived from.
derived class	A new class that specialises an existing class

Method	**Responsibility**
calc_interest	Calculate at the end of the day the interest due on the balance of the account. This will be accumulated and credited to the account at the end of the accounting period.
add_interest	Credit the account with the accumulated interest for the accounting period.
set_rate	Set the interest rate for all instances of the class.

Three instances of the class `Interest_account` sharing the same class global variable `the_interest_rate`.




```
with Class_Account;
use  Class_Account;
package Class_Interest_Account is

  type Interest_Account is new Account with private;

  procedure Set_Rate( Rate:in Float );
  procedure Calc_Interest( The:in out Interest_Account );
  procedure Add_Interest( The:in out Interest_Account );
private
  Daily_Interest_Rate: constant Float := 0.00026116; --10%
  type Interest_Account is new Account with record
    Accumulated_Interest : Money := 0.00;          --To date
  end record;
  The_Interest_Rate      : Float := Daily_Interest_Rate;
end Class_Interest_Account;
```

The class `Interest_account` contains:

- The following methods:

Defined in Class_account	Defined in Class_interest_account
deposit	calc_interest
withdraw	add_interest
balance	set_rate
statement	

- The following data members:

Defined in Class_account	Defined in Class_interest_account
balance_of	accumulated_interest
	interest_rate

```
package body Class_Interest_Account is

  procedure Set_Rate( Rate:in Float ) is
  begin
    The_Interest_Rate := Rate;
  end Set_Rate;

  procedure Calc_Interest( The:in out Interest_Account ) is
  begin
    The.Accumulated_Interest := The.Accumulated_Interest +
      Balance(The) * The_Interest_Rate;
  end Calc_Interest;

  procedure Add_Interest( The:in out Interest_Account ) is
  begin
    Deposit( The, The.Accumulated_Interest );
    The.Accumulated_Interest := 0.00;
  end Add_Interest;

end Class_Interest_Account;
```

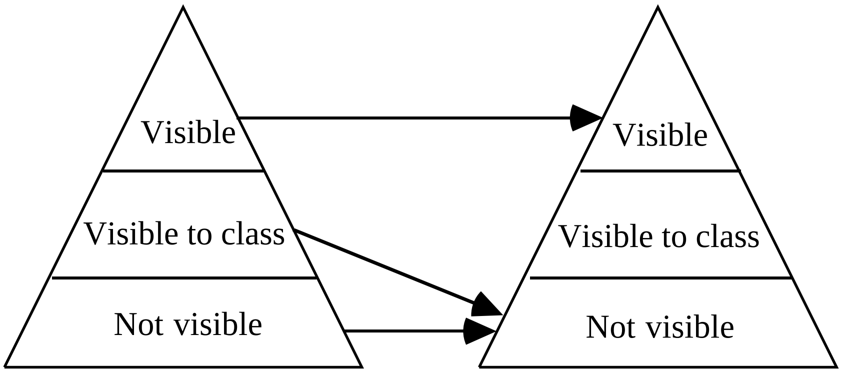
```
with Ada.Text_Io, Ada.Float_Text_Io, Class_Account;
use  Ada.Text_Io, Ada.Float_Text_Io, Class_Account;
procedure Statement( An_Account:in Account ) is
begin
  Put("Mini statement: The amount on deposit is £" );
  Put( Balance(An_Account), Aft=>2, Exp=>0 );
  New_Line(2);
end Statement;
```

```
with Ada.Text_io,
     Class_Interest_Account, Class_Account, Statement;
use  Ada.Text_io,
     Class_Interest_Account, Class_Account;
procedure Main is
  Mike    :Account;            --Normal Account
  Corinna :Interest_Account;   --Interest bering account
  Obtained:Money;
begin
  Set_Rate( 0.00026116 );      --For all instances of
                               --interest bering accounts
  Statement( Mike );

  Put("Deposit £50.00 into Mike's account"); New_Line;
  Deposit( Mike, 50.00 );
  Statement( Mike );

  Put("Withdraw £80.00 from Mike's account"); New_Line;
  Withdraw( Mike, 80.00, Obtained );
  Statement( Mike );

  Put("Deposit £500.00 into Corinna's account"); New_Line;
  Deposit( Corinna, 500.00 );
  Statement( Account(Corinna) );

  Put("Add interest to Corinna's account"); New_Line;
  Calc_Interest( Corinna );
  Add_Interest( Corinna );
  Statement( Account(Corinna) );
end Main;
```

```
Mini statement: The amount on deposit is £ 0.00

Deposit £50.00 into Mike's account
Mini statement: The amount on deposit is £50.00

Withdraw £80.00 from Mike's account
Mini statement: The amount on deposit is £50.00

Deposit £500.00 into Corinna's account
Mini statement: The amount on deposit is £500.00

Add interest to Corinna's account
Mini statement: The amount on deposit is £500.13
```

Visibility

Key	Base class visibility	Derived class visibility
Visible to class and client.		
Visible to this class only		
Not visible to class or client		



Converting a derived class to a base class

```
with Class_Interest_Account, Class_Account, Statement;
use  Class_Interest_Account, Class_Account;
procedure Main is
  Corinna : Interest_Account;
  New_Acc : Account;
begin
  Deposit( Corinna, 100.00 );
  New_Acc := Account(Corinna);    --derived -> base conversion
  Statement( Account(Corinna) ); --Interest_account
  Statement( New_Acc );          --Account
end Main;
```

Abstract Class

```
package Class_Abstract_Account is

  type Abstract_Account is abstract tagged null record;
  subtype Money  is Float;
  subtype Pmoney is Float range 0.0 .. Float'Last;

  procedure Deposit  ( The:in out Abstract_Account;
                       Amount:in Pmoney ) is abstract;
  procedure Withdraw ( The:in out Abstract_Account;
                       Amount:in Pmoney;
                       Get:out Pmoney ) is abstract;
  function  Balance  ( The:in Abstract_Account )
                       return Money is abstract;
end Class_Abstract_Account;
```

```
with Class_Abstract_Account;
use  Class_Abstract_Account;
package Class_Account is

  type Account is new Abstract_Account with private;
  subtype Money  is Class_Abstract_Account.Money;
  subtype Pmoney is Class_Abstract_Account.Pmoney;

  procedure Deposit  ( The:in out Account; Amount:in Pmoney );
  procedure Withdraw ( The:in out Account; Amount:in Pmoney;
                       Get:out Pmoney );
  function  Balance  ( The:in Account ) return Money;
private
  type Account is new Abstract_Account with record
    Balance_Of : Money := 0.00;       --Amount in account
  end record;
end Class_Account;
```

Limiting the withdrawals

```
with Class_Account;
use  Class_Account;
package Class_Account_Ltd is

  type Account_Ltd is new Account with private;

  procedure Withdraw ( The:in out Account_Ltd;
                       Amount:in Pmoney; Get:out Pmoney );
  procedure Reset( The:in out Account_Ltd );
private
  Withdrawals_In_A_Week : Natural := 3;
  type Account_Ltd is new Account with record
    Withdrawals : Natural := Withdrawals_In_A_Week;
  end record;
end Class_Account_Ltd;
```

```
package body Class_Account_Ltd is
```

```
  procedure Withdraw ( The:in out Account_Ltd;
      Amount:in Pmoney; Get:out Pmoney ) is
  begin
    if The.Withdrawals > 0 then                --Not limit
      The.Withdrawals := The.Withdrawals - 1;
      Withdraw( Account(The), Amount, Get );   --In Account
    else
      Get := 0.00;                              --Sorry
    end if;
  end Withdraw;
```

```
  procedure Reset( The:in out Account_Ltd ) is
  begin
    The.Withdrawals := Withdrawals_In_A_Week;
  end Reset;

end Class_Account_Ltd;
```

```
with Class_Account, Class_Account_ltd, Statement;
use  Class_Account, Class_Account_ltd;
procedure Main is
  Mike   : Account_Ltd;
  Obtain: Money;
begin
  Deposit( Mike, 300.00 );          --In credit
  Statement( Account(Mike) );
  Withdraw( Mike, 100.00, Obtain ); --Withdraw some money
  Withdraw( Mike,  10.00, Obtain ); --Withdraw some money
  Withdraw( Mike,  10.00, Obtain ); --Withdraw some money
  Withdraw( Mike,  20.00, Obtain ); --Withdraw some money
  Statement( Account(Mike) );
end Main;
```

```
Mini statement: The amount on deposit is £300.00
Mini statement: The amount on deposit is £180.00
```

```
  Withdraw( Account(Mike),  20.00, Obtain ); --Cheat
```

```
with Class_Account, Class_Account_Ltd, Statement;
use  Class_Account;
procedure Main is
  Mike   : Account_Ltd;
  Obtain: Class_Account.Money;
begin
  Deposit( Mike, 300.00 );          --In credit
  Statement( Account(Mike) );
  Withdraw( Mike, 100.00, Obtain ); --Withdraw some money
  Withdraw( Mike,  10.00, Obtain ); --Withdraw some money
  Withdraw( Mike,  10.00, Obtain ); --Withdraw some money
  Withdraw( Mike,  20.00, Obtain ); --Withdraw some money
  Statement( Account(Mike) );
end Main;
```

Initialization & Finalization

Method	Responsibility
`Withdraw`	Withdraw money from the account and write an audit trail record.
`Deposit`	Deposit money into the account and write an audit trail record.
`Balance`	Return the amount in the account and write an audit trail record.
`Initialize`	If this is the only active instance of the class `Account` then open the audit trail file.
`Finalization`	If this is the last active instance of the class `Account` then close the audit trail file.

```
with Ada.Text_Io, Ada.Finalization;
use  Ada.Finalization;
package Class_Account_At is

  type Account_At is new Limited_Controlled with private;
  subtype Money  is Float;
  subtype Pmoney is Float range 0.0 .. Float'Last;

  procedure Initialize( The:in out Account_At );
  procedure Finalize  ( The:in out Account_At );

  procedure Deposit( The:in out Account_At; Amount:in Pmoney );
  procedure Withdraw( The:in out Account_At;
                      Amount:in Pmoney; Get:out Pmoney );
  function  Balance( The:Account_At ) return Money;
private
  type Account_At is new Limited_Controlled with record
    Balance_Of : Money   := 0.00;        --Amount on deposit
    Number     : Natural := 0;
  end record;
  The_Audit_Trail: Ada.Text_Io.File_Type;  --File handle
  The_Active     : Natural := 0;           --No of accounts
end Class_Account_At;
```

```
with Ada.Text_Io, Ada.Integer_Text_Io, Ada.Float_Text_Io;
use  Ada.Text_Io, Ada.Integer_Text_Io, Ada.Float_Text_Io;
package body Class_Account_At is

  procedure Initialize( The:in out Account_At ) is
  begin
    The_Active := The_Active + 1;      --Another object
    if The_Active = 1 then             -- first time for class
      Open( File=>The_Audit_Trail,
        Mode=>Append_File, Name=>"log.txt" );
    end if;
  end Initialize;
```

```
  procedure Finalize( The:in out Account_At ) is
  begin
    if The_Active = 1 then Close( The_Audit_Trail ); end if;
    The_Active:=The_Active-1;
  end Finalize;
```

```
  procedure Deposit( The:in out Account_At; Amount:in Pmoney ) is
  begin
    The.Balance_Of := The.Balance_Of + Amount;
    Audit_Trail( The, " Deposit    : Amount   = ", Amount );
  end Deposit;
```

```
  procedure Withdraw( The:in out Account_At;
                      Amount:in Pmoney; Get:out Pmoney ) is
  begin
    if The.Balance_Of >= Amount then
      The.Balance_Of := The.Balance_Of - Amount;
      Get := Amount;
    else
      Get := 0.00;
    end if;
    Audit_Trail( The, " Withdraw  : Amount  = ", Get );
  end Withdraw;

  function  Balance( The:in Account_At ) return Money is
  begin
    Audit_Trail( The, " Balance   : Balance = ", The.Balance_Of );
    return The.Balance_Of;
  end Balance;

end Class_Account_At;
```

```
with Class_Account_At, Statement;
use  Class_Account_At;
procedure Main is
  Bank   : array ( 1 .. 10 ) of Account_At;
  Obtain: Money;
begin
  Deposit( Bank(1), 100.00 );        --Deposit 100.00
  Withdraw( Bank(1), 80.00, Obtain );--Withdraw 80.00
  Deposit( Bank(2), 200.00 );        --Deposit 200.00
end Main;
```

```
Deposit   : 100.00
Withdraw  : 80.00
Deposit   : 200.00
```

Type in Ada.Finalization	Properties
`Controlled`	Allow user defined initialization and finalization for inheriting types. Instances of these types may be assigned.
`Limited_Controlled.`	Allow user defined initialization and finalization for inheriting types. Instances of these types may not be assigned.

Hiding the base class methods

```
with Class_Account;
use  Class_Account;
package Class_Restricted_Account is

  type Restricted_Account is private;
  subtype Money  is Class_Account.Money;
  subtype Pmoney is Class_Account.Pmoney;

  procedure Deposit( The:in out Restricted_Account;
                     Amount:in Pmoney );
private
  type Restricted_Account is new Account with record
    null;
  end record;
end Class_Restricted_Account;
```

```
package body Class_Restricted_Account is

  procedure Deposit( The:in out Restricted_Account;
      Amount:in Pmoney ) is
  begin
    Deposit( Account(The), Amount );
  end Deposit;

end Class_Restricted_Account;
```

Child libraries

```
package Class_Interest_Account.Inspect_Interest is
  function Interest_Is( The:in Interest_Account )
    return Money;
end Class_Interest_Account.Inspect_Interest;
```

```
package body Class_Interest_Account.Inspect_Interest is

  function Interest_Is( The:in Interest_Account )
      return Money is
  begin
    return The.Accumulated_Interest;
  end Interest_Is;

end Class_Interest_Account.Inspect_Interest;
```

```
with Ada.Text_Io, Ada.Float_Text_Io, Class_Account,
     Class_Interest_Account,
     Class_Interest_Account.Inspect_Interest, Statement;
use  Ada.Text_Io, Ada.Float_Text_Io, Class_Account,
     Class_Interest_Account,
     Class_Interest_Account.Inspect_Interest;
procedure Main is
  My_Account: Interest_Account;
  Obtained   : Money;
begin
  Statement( My_Account );
  Put("Deposit 100.00 into account"); New_Line;
  Deposit( My_Account, 100.00 );           --Day 1
  Calc_Interest( My_Account );             --End of day 1
  Calc_Interest( My_Account );             --End of day 2
  Statement( My_Account );                 --Day 3
  Obtained := Interest_Is( My_Account );   --How much interest
  Put("Interest accrued so far : £" );
  Put( Obtained, Aft=>2, Exp=>0 ); New_Line;
end Main;
```

Defining new operators

```
  function "+" ( F:in Integer; S:in Integer ) return Integer is
  begin
    Put("[Performing "); Put(F, Width=>1 );
    Put(" + "); Put(S, Width=>1 ); Put("]");
    return Standard."+"( F, S );
  end "+";
```

```
with Ada.Text_Io, Ada.Integer_Text_Io;
use  Ada.Text_Io, Ada.Integer_Text_Io;
procedure Main is
  function "+" ( F:in Integer; S:in Integer ) return Integer is
  begin
    Put("[Performing "); Put(F, Width=>1 );
    Put(" + "); Put(S, Width=>1 ); Put("]");
    return Standard."+"( F, S );
  end "+";
begin
  Put("The sum of 1 + 2 is: "); Put ( 1+2 ); New_Line;
  Put("The sum of 1 + 2 is: ");
  Put( Standard."+"(1,2), Width=>1 ); New_Line;
  Put("The sum of 1 + 2 is: ");
  Put( "+"(1,2), Width=>1 ); New_Line;
end Main;
```

which when compiled and run will deliver the following results:

```
The sum of 1 + 2 is: [Performing 1 + 2]3
The sum of 1 + 2 is: 3
The sum of 1 + 2 is: [Performing 1 + 2]3
```

A rational arithmetic package

Method	Responsibility
+	Delivers the sum of two rational numbers as a rational number.
-	Delivers the difference of two rational numbers as a rational number.
*	Delivers the product of two rational numbers as a rational number.
/	Delivers the division of two rational numbers as a rational number.
`Rat_Const`	Creates a rational number from two `Integer` numbers
`Image`	Returns a string image of a rational number in the canonical form '`a b/c`'. For example: Put(Image(Rat_Const(3,2))); would print 1 1/2

```
package Class_Rational is
  type Rational is private;

  function "+" ( F:in Rational; S:in Rational ) return Rational;
  function "-" ( F:in Rational; S:in Rational ) return Rational;
  function "*" ( F:in Rational; S:in Rational ) return Rational;
  function "/" ( F:in Rational; S:in Rational ) return Rational;

  function Rat_Const( F:in Integer;
                      S:in Integer:=1 ) return Rational;
  function  Image( The:in Rational ) return String;
private
  type Rational is record
    Above : Integer := 0;      --Numerator
    Below : Integer := 1;      --Denominator
  end record;
end Class_Rational;
```

```
with Ada.Text_Io, Class_Rational;
use  Ada.Text_Io, Class_Rational;
procedure Main is
  A,B : Rational;
begin
  A := Rat_Const( 1, 2 );
  B := Rat_Const( 1, 3 );

  Put( "a     = " ); Put( Image(A) );   New_Line;
  Put( "b     = " ); Put( Image(B) );   New_Line;
  Put( "a + b = " ); Put( Image(A+B) ); New_Line;
  Put( "a - b = " ); Put( Image(A-B) ); New_Line;
  Put( "b - a = " ); Put( Image(B-A) ); New_Line;
  Put( "a * b = " ); Put( Image(A*B) ); New_Line;
  Put( "a / b = " ); Put( Image(A/B) ); New_Line;
end Main;
```

```
a     = 1/2
b     = 1/3
a + b = 5/6
a - b = 1/6
b - a = -1/6
a * b = 1/6
a / b = 1 1/2
```

```
package body Class_Rational is

function Sign( The:in Rational ) return Rational is
begin
  if The.Below >= 0 then                --   -a/b or a/b
    return The;
  else                                  --   a/-b or -a/-b
    return Rational'( -The.Above, -The.Below );
  end if;
end Sign;
```

```
function Simplify( The:in Rational ) return Rational is
  Res: Rational := The;
  D  : Positive;                        --Divisor to reduce with
begin
  if Res.Below = 0 then                 --Invalid treat as 0
    Res.Above := 0; Res.Below := 1;
  end if;
  D := 2;                               --Divide by 2, 3, 4 ...
  while D < Res.Below loop
    while Res.Below rem D = 0 and then Res.Above rem D = 0 loop
      Res.Above := Res.Above / D;
      Res.Below := Res.Below / D;
    end loop;
    D := D + 1;
  end loop;
  return Res;
end Simplify;
```

```
function "+" (F:in Rational; S:in Rational) return Rational is
  Res : Rational;
begin
  Res.Below := F.Below * S.Below;
  Res.Above := F.Above * S.Below + S.Above * F.Below;
  return Simplify(Res);
end "+";

function "-" (F:in Rational; S:in Rational) return Rational is
  Res : Rational;
begin
  Res.Below := F.Below * S.Below;
  Res.Above := F.Above * S.Below - S.Above * F.Below;
  return Simplify(Res);
end "-";

function "*" (F:in Rational; S:in Rational) return Rational is
  Res : Rational;
begin
  Res.Above := F.Above * S.Above;
  Res.Below := F.Below * S.Below;
  return Simplify(Res);
end "*";

function "/" (F:in Rational; S:in Rational) return Rational is
  Res : Rational;
begin
  Res.Above := F.Above * S.Below;
  Res.Below := F.Below * S.Above;
  return Simplify(Res);
end "/";
```

```
function Image( The:in Rational ) return String is
    Above : Integer := The.Above;
    Below : Integer := The.Below;

    function Trim( Str:in String ) return String is
    begin
      return Str( Str'First+1 .. Str'Last );
    end Trim;

    function To_String( Above, Below : in Integer )
             return String is
    begin
      if Above = 0 then              --No fraction
        return "";
      elsif Above >= Below then      --Whole number
        return Trim( Integer'Image(Above/Below) ) & " " &
               To_String( Above rem below, Below );
      else
        return Trim( Integer'Image( Above ) ) & "/" &
               Trim( Integer'Image( Below ) );
      end if;
    end To_String;

  begin
   if Above = 0 then
     return "0";                                  --Zero
   elsif Above < 0 then
     return "-" & To_String( abs Above, Below ); ---ve
   else
     return To_String( Above, Below );            --+ve
   end if;
  end Image;

end Class_Rational;
```

```
with Ada.Text_Io, Class_Rational;
procedure Main is
  function "+" ( F:in Class_Rational.Rational;
                 S:in Class_Rational.Rational )
           return Class_Rational.Rational
             renames Class_Rational."+";
  function "-" ( F:in Class_Rational.Rational;
                 S:in Class_Rational.Rational )
           return Class_Rational.Rational
             renames Class_Rational."-";
  -- etc

  A,B : Class_Rational.Rational;
begin
  A := Class_Rational.Rat_Const( 1, 2 );
  B := Class_Rational.Rat_Const( 1, 3 );

  put( "A+B   = " );
  put( Class_Rational.Image(A+B) );
  Ada.Text_Io.New_Line;

  -- Etc

end Main;
```

Use Type

```
with Ada.Text_Io, Class_Rational;
use  type Class_Rational.Rational;
procedure Main is
  A,B : Class_Rational.Rational;
begin
  A := Class_Rational.Rat_Const( 1, 2 );
  B := Class_Rational.Rat_Const( 1, 3 );

  put( "A+B   = " );
  put( Class_Rational.Image(A+B) );
  Ada.Text_Io.New_Line;

  -- Etc

end Main;
```

A bounded string class

This has the following responsabilities

Method	**Responsibility**
operator: `&`	Concatenate an Ada string or a `Bounded_string` to a `Bounded_string`.
operators: `> >= < <= =`	Compare Bounded_string's
`to_string`	Convert an instance of a `Bounded_string` to an Ada string.
`to_bounded_string`	Convert an Ada string to an instance of a `Bounded_string`.
`slice`	Deliver a slice of a `Bounded_string`

```
package Class_Bounded_String is
  type Bounded_String is private;

  function To_Bounded_String(Str:in String)
    return Bounded_String;

  function To_String(The:in Bounded_String) return String;

  function "&" (F:in Bounded_String; S:in Bounded_String)
    return Bounded_String;
  function "&" (F:in Bounded_String; S:in String)
    return Bounded_String;
  function "&" (F:in String; S:in Bounded_String)
    return Bounded_String;

  function Slice( The:in Bounded_String;
                  Low:in Positive; High:in Natural )
    return String;

  function "="  ( F:in Bounded_String; S:in Bounded_String )
    return Boolean;

  function ">"  ( F:in Bounded_String; S:in Bounded_String )
    return Boolean;
  function ">=" ( F:in Bounded_String; S:in Bounded_String )
    return Boolean;
  function "<"  ( F:in Bounded_String; S:in Bounded_String )
    return Boolean;
  function "<=" ( F:in Bounded_String; S:in Bounded_String )
    return Boolean;
private
  Max_String: constant := 80;
  subtype Str_Range is Natural range 0 .. Max_String;
  type A_Bounded_String( Length: Str_Range := 0 ) is record
    Chrs: String( 1 .. Length );  --Stored string
  end record;
  type Bounded_String is record
    V_Str : A_Bounded_String;
  end record;
end Class_Bounded_String;
```

```
package body Class_Bounded_String is

  function To_Bounded_String( Str:in String )
      return Bounded_String is
  begin
    return (V_Str=>(Str'Length, Str));
  end To_Bounded_String;
```

```
  function To_String(The:in Bounded_String) return String is
  begin
    return The.V_Str.Chrs( 1 .. The.V_Str.Length );
  end To_String;
```

```
  function Slice( The:in Bounded_String;
                  Low:in Positive; High:in Natural)
    return String is
  begin
    if Low <= High and then High <= The.V_Str.Length then
      return The.V_Str.Chrs( Low .. High );
    end if;
    return "";
  end Slice;
```

```
function "&" ( F:in Bounded_String; S:in Bounded_String )
    return Bounded_String is
begin
  return (V_Str=>(F.V_Str.Chrs'Length + S.V_Str.Chrs'Length,
      F.V_Str.Chrs & S.V_Str.Chrs));
end "&";

function "&" ( F:in Bounded_String; S:in String )
    return Bounded_String is
begin
  return (V_Str=>(F.V_Str.Chrs'Length + S'Length,
      F.V_Str.Chrs & S ) );
end "&";

function "&" ( F:in String; S:in Bounded_String )
    return Bounded_String is
begin
  return ( V_Str=>(F'Length + S.V_Str.Chrs'Length,
      F & S.V_Str.Chrs ) );
end "&";
```

```
function ">"  ( F:in Bounded_String; S:in Bounded_String )
    return Boolean is
begin
  return F.V_Str.Chrs > S.V_Str.Chrs;
end ">";

function ">=" ( F:in Bounded_String; S:in Bounded_String )
    return Boolean is
begin
  return F.V_Str.Chrs >= S.V_Str.Chrs;
end ">=";

function "<"  ( F:in Bounded_String; S:in Bounded_String )
    return Boolean is
begin
  return F.V_Str.Chrs < S.V_Str.Chrs;
end "<";

function "<=" ( F:in Bounded_String; S:in Bounded_String )
    return Boolean is
begin
  return F.V_Str.Chrs <= S.V_Str.Chrs;
end "<=";
```

```
  function "="  ( F:in Bounded_String; S:in Bounded_String )
      return Boolean is
  begin
    return F.V_Str.Chrs = S.V_Str.Chrs;
  end "=";
end Class_bounded_string;
```

```
procedure Main is
  Town, County, Address : Bounded_String;
begin
  Town   := To_Bounded_String( "Brighton" );
  County := To_Bounded_String( "East Sussex" );

  Address := Town & " " & County;

  Put( To_String(Address) ); New_Line;
  Put( Slice( County & " UK", 6, 14 ) );
  New_Line;

end Main;
```

```
Brighton East Sussex
Sussex UK
```

Use Type

```
with Ada.Text_io, Class_bounded_string;
procedure main is
  function "&" (f,s:in Class_bounded_string:Bounded_string)
    return Class_bounded_string: Bounded_string
      renames Class_bounded_string."&";

  function "&" (f:in Class_bounded_string:Bounded_string;
              s:in String)
    return Class_bounded_string: Bounded_string
      renames Class_bounded_string."&";

  -- etc

  Town   : Class_Bounded_String.Bounded_String :=
    Class_Bounded_String.To_Bounded_String("Brighton");
  County: Class_Bounded_String.Bounded_String :=
    Class_Bounded_String.To_Bounded_String("E Sussex");
begin
  Ada.Text_Io.Put(
    Class_Bounded_String.To_String( Town & " " & County )
    );
end main;
```

```
with Ada.Text_Io, Class_Bounded_String;
use  type Class_Bounded_String.Bounded_String;
procedure Main is
  Town   : Class_Bounded_String.Bounded_String :=
    Class_Bounded_String.To_Bounded_String("Brighton");
  County: Class_Bounded_String.Bounded_String :=
    Class_Bounded_String.To_Bounded_String("E Sussex");
begin
  Ada.Text_Io.Put(
    Class_Bounded_String.To_String( Town & " " & County )
    );
end Main;
```

Exceptions

Exception	Explanation
Constraint_error	An invalid value has been supplied.
Data_error	The data item read is not of the expected type.
End_error	During a read operation the end of file was detected.

```
with Ada.Text_Io, Ada.Integer_Text_Io;
use  Ada.Text_Io, Ada.Integer_Text_Io;
procedure Main is
  Number : Integer;          --Number read in
  Ch     : Character;        --As a character
begin
  loop
    begin
      Put("Enter character code : ");          --Ask for number
      exit when End_Of_File;                   --EOF ?
      Get( Number ); Skip_Line;                --Read number
      Put("Represents the character [");       --Valid number
      Put( Character'Val(Number) );
      Put("]");                                --Valid character
      New_Line;
    exception
      when Data_Error =>
        Put("Not a valid Number"); Skip_Line;     --Exception
        New_Line;
      when Constraint_Error =>
        Put("Not representable as a Character]"); --Exception
        New_Line;
      when End_Error =>
        Put("Unexpected end of data"); New_Line;  --Exception
        exit;
    end;
  end loop;
end Main;
```

```
Enter character code : 96
Represents the character [`]
Enter character code : Invalid
Not a valid Number
Enter character code : 999
Represents the character [Not representable]
Enter character code : ^D
```

Exception	Explanation
Name_error	File does not exist.
Status_error	File is all ready open.

The unix program cat

```
with Ada.Text_Io, Ada.Command_Line;
use  Ada.Text_Io, Ada.Command_Line;
procedure Cat is
  Fd  : Ada.Text_Io.File_Type;             --File descriptor
  Ch  : Character;                         --Current character
begin
  if Argument_Count >= 1 then
    for I in 1 .. Argument_Count loop      --Repeat for each file
      begin
        Open( File=>Fd, Mode=>In_File,     --Open file
          Name=>Argument(I) );
        while not End_Of_File(Fd) loop     --For each Line
          while not End_Of_Line(Fd) loop--For each character
            Get(Fd,Ch); Put(Ch);           --Read / Write character
          end loop;
          Skip_Line(Fd); New_Line;         --Next line / new line
        end loop;
        Close(Fd);                         --Close file
      exception
        when Name_Error =>
          Put("cat: " & Argument(I) & " no such file" );
          New_Line;
        when Status_Error =>
          Put("cat: " & Argument(I) & " all ready open" );
          New_Line;
      end;
    end loop;
  else
    Put("Usage: cat file1 ... "); New_Line;
  end if;
end Cat;
```

Raising an exception

```
raise Constraint_Error;
```

```
Unexpected_Condition : Exception;
```

```
raise Unexpected_Condition;
```

Handling any exception

```
  when others =>
    Put("Exception caught"); New_Line;
```

```
 when The_Event: others =>
    Put("Unexpected exception is ");
    Put( Exception_Name( The_Event ) ); New_Line;
```

Package `Ada.Exceptions`

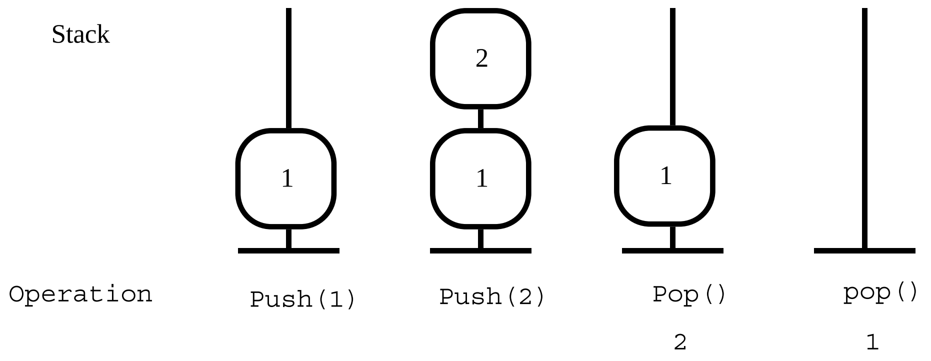
```
event : Exception_ocurrence;
```

Function (Defined in `Ada.Exceptions`)	**Returns as a string:**
`exception_name(event)`	In upper case the exception name starting with the root library unit.
`exception_information(event)`	Detailed information about the exception.
`exception_message(event)`	A short explanation of the exception.

Function / procedure	**Action**
`reraise_occurrence(event)`	A procedure which re-raises the exception event.
`raise_exception(e,"Mess")`	A procedure which raises exception `e` with the message `"Mess"`.

Data structures

A stack



Method	Responsibility
push	Push the current item onto the stack. The exception Stack_error will be raised if this cannot be done.
pop	Return the top item on the stack, the item is removed from the stack. The exception Stack_error will be raised if this cannot be done.
reset	Resets the stack to an initial state of empty.

```
package Class_Stack is
  type Stack is private;                   --Copying allowed
  Stack_Error: exception;                  --When error

  procedure Reset( The:in out Stack);
  procedure Push( The:in out Stack; Item:in Integer );
  procedure Pop(The:in out Stack; Item:out Integer );
private

  Max_Stack: constant := 3;
  type    Stack_Index is range 0 .. Max_Stack;
  subtype Stack_Range is Stack_Index range 1 .. Max_Stack;
  type    Stack_Array is array ( Stack_Range ) of Integer;

  type Stack is record
    Elements: Stack_Array;             --Array of elements
    Tos     : Stack_Index := 0;        --Index
  end record;

end Class_Stack;
```

```
with Simple_io, Class_stack;
with Ada.Text_Io, Ada.Integer_Text_Io, Class_Stack;
use  Ada.Text_Io, Ada.Integer_Text_Io, Class_Stack;
procedure Main is
  Number_Stack : Stack;                   --Stack of numbers
  Action       : Character;               --Action
  Number       : Integer;                 --Number processed
begin
  while not End_Of_File loop
    while not End_Of_Line loop
      begin
        Get( Action );
        case Action is                    --Process action
          when '+' =>
            Get( Number ); Push(Number_Stack,Number);
            Put("push number = "); Put(Number); New_Line;
          when '-' =>
            Pop(Number_Stack,Number);
            Put("Pop number  = "); Put(Number); New_Line;
          when others =>
            Put("Invalid action"); New_Line;
        end case;
      exception
        when Stack_Error =>
          Put("Stack_error"); New_Line;
        when Data_Error  =>
          Put("Not a number"); New_Line; Skip_Line;
        when End_Error   =>
          Put("Unexpected end of file"); New_Line; exit;
      end;
    end loop;
    Skip_Line;
  end loop;
  Reset( Number_Stack );
end Main;
```

```
+1+2+3+4----
```

```
push number =       1
push number =       2
push number =       3
Stack_error
Pop number  =       3
Pop number  =       2
Pop number  =       1
Stack_error
```

```
package body Class_Stack is

  procedure Reset( The:in out Stack ) is
  begin
    The.Tos := 0;  --Set TOS to 0 (Non existing element)
  end Reset;
```

```
  procedure Push( The:in out Stack; Item:in Integer ) is
  begin
    if The.Tos /= Max_Stack then
      The.Tos := The.Tos + 1;                 --Next element
      The.Elements( The.Tos ) := Item;        --Move in
    else
      raise Stack_Error;                      --Failed
    end if;
  end Push;
```

```
 procedure Pop( The:in out Stack; Item :out Integer ) is
  begin
    if The.Tos > 0 then
      Item := The.Elements( The.Tos );        --Top element
      The.Tos := The.Tos - 1;                 --Move down
    else
      raise Stack_Error;                      --Failed
    end if;
  end Pop;

end Class_Stack;
```

Generics

Procedures / functions

Specification

```
generic                                        --Specification
  type T is ( <> );                            --Any discrete type
procedure G_Order( A,B:in out T );             --Prototype ord
```

Implementation

```
procedure G_Order( A,B:in out T ) is           --Implementation ord
  Tmp : T;                                     --Temporary
begin
  if A > B then                                --Compare
    Tmp := A; A := B; B := Tmp;                -- Swap
  end if;
end G_Order;                                   --
```

Instantiation

```
  procedure Order is new G_Order( Natural );  --Instantiate order
```

Nested generic procedures/functions

Specification

```
generic                                         --Specification
  type T is ( <> );                             --Any discrete type
procedure G_3Order( A,B,C:in out T );           --Prototype ord
```

Implementation

```
with G_Order;
procedure G_3Order( A,B,C:in out T ) is   --Implementation ord
  procedure Order is new G_Order( T );    --Instantiate order
begin
  Order( A, B );                          --S L -
  Order( B, C );                          --? ? L
  Order( A, B );                          --S M L
end G_3Order;                             --
```

Example of use

```
with Ada.Text_Io, Ada.Integer_Text_Io, G_3Order;
use  Ada.Text_Io, Ada.Integer_Text_Io;
procedure Main is
  procedure Order is new G_3Order( Natural );      --Instantiate
  Room1 : Natural := 30; --30 Square meters
  Room2 : Natural := 25; --25 Square meters
  Room3 : Natural := 20; --20 Square meters
begin
  Order( Room1, Room2, Room3 );
  Put("Rooms in ascending order of size are "); New_Line;
  Put( Room1 ); New_Line;
  Put( Room2 ); New_Line;
  Put( Room3 ); New_Line;
end Main;
```

```
Rooms in ascending order of size are
    20
    25
    30
```

Overview of componants

The formal type specifies the catergory of types to which the generic parameter can belong.

Specification of generic procedure

```
generic
  type T is ( <> );
procedure ord_2( a,b:in out T );
```

Implementation of generic procedure

```
procedure ord_2(a,b:in out T) is
begin
  -- Body of ord_2
end ord_2;
```

Advantages

- Facilitate re-use, by allowing an implementor to write procedures or functions which process objects of a type that is determined by the user of the procedure or function.

Disadvantages

- Extra care must be exercised in writing the procedure or function. This will undoubtedly result in a greater cost to the originator.
- The implementation of the generic procedure, function or package may not be as efficient as a direct implementation.

Formal type specification `type T`	**In Ada 83**	**Actual parameter can belong to the following types**
`is private`	✔	Any non limited type.
`is limited private`	✔	Any type.
`is tagged`	✗	Any non limited tagged type.
`is limited tagged`	✗	Any tagged type.
`is (<>)`	✔	Any discrete type, constrained type.
`(<>) is private`	✗	Any discrete or indefinite non limited type.
`(<>) is limited private`	✗	Any discrete or indefinite type.
`is mod <>`	✗	Any modular type.
`is range <>`	✔	Any integer type.
`is digits <>`	✔	Any float type.
`is delta <>`	✔	Any fixed ordinary type.
`is delta <> digits <>`	✗	Any fixed decimal type.
`is access`	✔	Any access type.
`with procedure ...`	✔	procedure matching the signiture.
`with package ...`	✗	package matching the signiture.

Generic packages

Specification of a stack

```
generic
  type T is private;            --Can specify any type
  Max_Stack:in Positive := 3; --Has to be typed / not const
package Class_Stack is
  type Stack is tagged private;
  Stack_Error: exception;

  procedure Reset( The:in out Stack);
  procedure Push( The:in out Stack; Item:in T );
  procedure Pop( The:in out Stack; Item:out T );
private

  type    Stack_Index is new Integer range 0 .. Max_Stack;
  subtype Stack_Range is Stack_Index
    range 1 .. Stack_Index(Max_Stack);
  type    Stack_Array is array ( Stack_Range ) of T;

  type Stack is tagged record
    Elements: Stack_Array;          --Array of elements
    Tos     : Stack_Index := 0;     --Index
  end record;

end Class_Stack;
```

Implementation of a stack

```
package body Class_Stack is

  procedure Push( The:in out Stack; Item:in T ) is
  begin
    if The.Tos /= Stack_Index(Max_Stack) then
      The.Tos := The.Tos + 1;             --Next element
      The.Elements( The.Tos ) := Item;    --Move in
    else
      raise Stack_Error;                  --Failed
    end if;
  end Push;
```

```
  procedure Pop( The:in out Stack; Item: out T ) is
  begin
    if The.Tos > 0 then
      Item := The.Elements( The.Tos );      --Top element
      The.Tos := The.Tos - 1;               --Move down
    else
      raise Stack_Error;                    --Failed
    end if;
  end Pop;
```

```
  procedure Reset( The:in out Stack ) is
  begin
    The.Tos := 0;  --Set TOS to 0 (Non existing element)
  end Reset;

end Class_Stack;
```

Instantiation

```
with Class_Stack;
pragma Elaborate_All( Class_Stack );
package Class_Stack_Int is new Class_Stack(Integer);
```

Example of use

```
with Ada.Text_Io, Ada.Integer_Text_Io, Class_Stack_Int;
use  Ada.Text_Io, Ada.Integer_Text_Io, Class_Stack_Int;
procedure Main is
  Number_Stack : Stack;              --Stack of numbers
  Action       : Character;          --Action
  Number       : Integer;            --Number processed
begin
  while not End_Of_File loop
    while not End_Of_Line loop
      begin
        Get( Action );
        case Action is               --Process action
          when '+' =>
            Get( Number ); Push(Number_Stack,Number);
            Put("push number = "); Put(Number); New_Line;
          when '-' =>
            Pop(Number_Stack,Number);
            Put("Pop number  = "); Put(Number); New_Line;
          when others =>
            Put("Invalid action"); New_Line;
        end case;
      exception
        when Stack_Error =>
          Put("Stack error"); New_Line;
        when Data_Error  =>
          Put("Not a number"); New_Line; Skip_Line;
        when End_Error   =>
          Put("Unexpected end of file"); New_Line; exit;
      end;
    end loop;
    Skip_Line;
  end loop;

  Reset( Number_Stack );
end Main;
```

```
+1+2+3+4----
```

```
push number = 1
push number = 2
push number = 3
Pop: Exception Stack_error
Pop number = 3
Pop number = 2
Pop number = 1
Pop: Exception Stack_error
```

Generic formal subprograms

Consider

```
if Instance_Of_Formal_Type > Another_Instance_Of_Formal_type then
  ...
end if;
```

Specification

```
generic                                     --Specification
  type T is private;                        --Any non limited type
  with function ">" ( A, B:in T )
       return Boolean is <>;                --Need def for >
procedure G_Order( A,B:in out T );          --Prototype G_Order
```

Implementation

```
procedure G_Order( A,B:in out T ) is        --Implementation G_Order
  Tmp : T;
begin
  if A > B then                             --Compare
    Tmp := A; A := B; B := Tmp;             --Swap
  end if;
end G_Order;                                --
```

Need to provide a definition for >

```
  with function ">" ( A, B:in T )
       return Boolean is <>;                --Need def for >
```

Instantiation

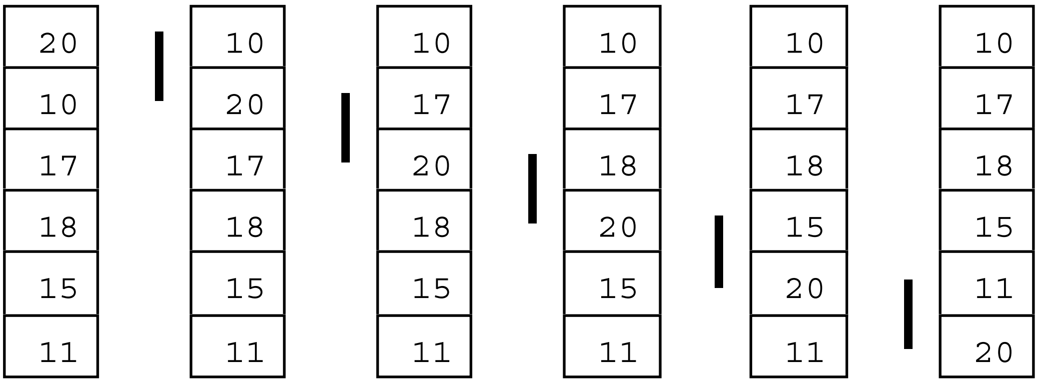
```
with G_Order;
  procedure Order is new G_Order( Natural );       --Instantiate
```

```
with G_Order;
  procedure Order is new G_Order( Natural, ">" );  --Instantiate
```

Bubble sort: Overview

20	10	17	18	15	11

The first pass of the bubble sort compares consecutive pairs of numbers and orders each pair into ascending order. This is illustrated below.



List of numbers	Commentary
20 10 17 18 15 11	The original list.
10 17 18 15 11 20	After the 1st pass through the list.
10 17 15 11 18 20	After the 2nd pass through the list.
10 15 11 17 18 20	After the 3rd pass through the list.
10 11 15 17 18 20	After the 4th pass through the list.

Specification: Generic sort

```
generic
  type T         is private;          --Any non limited type
  type Vec_Range is (<>);             --Any discrete type
  type Vec       is array( Vec_Range ) of T;
  with function  ">"( First, Second:in T ) return Boolean is <>;
procedure Sort( Items:in out Vec );
```

The generic formal parameters for the procedure `sort` are:

Formal parameter	Description
type T **is private**;	The type of data item to be sorted.
type Vec_Range **is** (<>);	The type of the index to the array.
type Vec **is array**(Vec_Range) **of** T;	The type of the array to be sorted.
with function ">"(First,Second:**in** T) **return** Boolean **is** <>;	A function that the user of the generic procedure provides to compare pairs of data items.

Implementation: Generic sort

```
procedure Sort( Items:in out Vec ) is
  Swaps : Boolean := True;
  Tmp   : T;
begin
  while Swaps loop
    Swaps := False;
    for I in Items'First .. Vec_Range'Pred(Items'Last) loop
      if Items( I ) > Items( Vec_Range'Succ(I) ) then
        Swaps := True;
        Tmp := Items( Vec_Range'Succ(I) );
        Items( Vec_Range'Succ(I) ) := Items( I );
        Items( I ) := Tmp;
      end if;
    end loop;
  end loop;
end Sort;
```

```
with Ada.Text_Io, Sort;
use  Ada.Text_Io;
procedure Main is

  type Chs_Range is range 1 .. 6;
  type Chs       is array( Chs_Range ) of Character;

  procedure Sort_Chs is new Sort (
    T         => Character,
    Vec_Range => Chs_Range,
    Vec       => Chs,
    ">"       => ">" );
  Some_Characters : Chs := ( 'q', 'w', 'e', 'r', 't', 'y' );
begin
  Sort_Chs( Some_Characters );
  for I in Chs_Range loop
    Put( Some_Characters( I ) ); Put( " " );
  end loop;
  New_Line;
end Main;
```

```
e q r t w y
```

```
with Ada.Text_Io, Sort;
use  Ada.Text_Io;
procedure Main is
  Max_Chs : constant := 7;
  type Height_Cm is range 0 .. 300;
  type Person is record
    Name   : String( 1 .. Max_Chs );  --Name as a String
    Height : Height_Cm := 0;          --Height in cm.
  end record;
  type People_Range is (First, Second, Third, Forth );
  type People       is array( People_Range ) of Person;
```

Then the declaration of two functions: The function `cmp_height` that returns true if the first person is taller than the second and the second function `cmp_name` that returns true if the first person's name collates later in the alphabet than the second.

```
 function Cmp_Height(First, Second:in Person) return Boolean is
  begin
    return First.Height > Second.Height;
  end Cmp_Height;

  function Cmp_Name( First, Second:in Person ) return Boolean is
  begin
    return First.Name > Second.Name;
  end Cmp_Name;
```

Two instantiations of the generic procedure `sort` are made, the first to sort people into ascending height order, the second to sort people into ascending name order.

```
 procedure Sort_People_Height is new Sort (
    T         => Person,
    Vec_Range => People_Range,
    Vec       => People,
    ">"       => Cmp_Height );

  procedure Sort_People_Name is new Sort (
    T         => Person,
    Vec_Range => People_Range,
    Vec       => People,
    ">"       => Cmp_Name );
```

The body of the program which orders the friends into ascending height and name order is:

```
  Friends : People := ( ("Paul    ", 146 ), ("Carol   ", 147 ),
                        ("Mike    ", 183 ), ("Corinna", 171 ) );
begin
  Sort_People_Name( Friends );                        --Name order
  Put( "The first in ascending name order is   " );
  Put( Friends( First ).Name ); New_Line;
  Sort_People_Height( Friends );                      --Height order
  Put( "The first in ascending height order is " );
  Put( Friends( First ).Name ); New_Line;
end Main;
```

Which when run will print:

```
The first in ascending name order is   Carol
The first in ascending height order is Paul
```

Generic child library

Method	Responsibility
top	Return the top item of the stack without removing it from the stack.
items	Return the current numbers of items in the stack.

```
generic
package Class_Stack.Additions is
  function Top( The:in Stack ) return T;
  function Items( The:in Stack ) return Natural;
private
end Class_Stack.Additions;
```

```
package body Class_Stack.Additions is

  function Top( The:in Stack ) return T is
  begin
    return The.Elements( The.Tos );
  end Top;

  function Items( The:in Stack ) return Natural is
  begin
    return Natural(The.Tos);
  end Items;

end Class_Stack.Additions;
```

```
with Class_Stack;
  pragma Elaborate_All( Class_Stack );
  package Class_Stack_Pos is new Class_Stack(Positive,10);

with Class_Stack_Pos, Class_Stack.Additions;
  pragma Elaborate_All( Class_Stack_Pos, Class_Stack.Additions );
  package Class_Stack_Pos_Additions is
          new Class_Stack_Pos.Additions;
```

A generic child of a package is considered to be declared within the generic parent. Thus, to instantiate an instance of the parent and child the following code is used:

```
with Class_Stack;
  pragma Elaborate_All( Class_Stack );
  package Class_Stack_Pos is new Class_Stack(Positive,10);

with Class_Stack_Pos, Class_Stack.Additions;
  pragma Elaborate_All( Class_Stack_Pos, Class_Stack.Additions );
  package Class_Stack_Pos_Additions is
          new Class_Stack_Pos.Additions;
```

```
with Ada.Text_Io, Ada.Integer_Text_Io,
     Class_Stack_Pos, Class_Stack_Pos_Additions;
use  Ada.Text_Io, Ada.Integer_Text_Io,
     Class_Stack_Pos, Class_Stack_Pos_Additions;
procedure Main is
  Numbers : Stack;
begin
  Push( Numbers, 10 );
  Push( Numbers, 20 );
  Put("Top item "); Put( Top( Numbers ) ); New_Line;
  Put("Items    "); Put( Items( Numbers ) ); New_Line;
end Main;
```

```
Top item      20
Items          2
```

Example using the generic child package

```
with Simple_io, Class_stack_pos, Class_stack_pos_additions;
use  Simple_io, Class_stack_pos, Class_stack_pos_additions;
procedure main is
  Numbers : Stack;
begin
  push( numbers, 10 );
  push( numbers, 20 );
  put("Top item "); put( top( numbers ) ); new_line;
  put("Items    "); put( items( numbers ) ); new_line;
end main;
```

which when run gives these results:

```
Top item     20
Items         2
```

Inheriting from a generic class

Method	Responsibility
depth	Return the maximum depth that the stack reached.

```
with Class_Stack, Class_Stack.Additions;
generic
  type T is private;
  Max_Stack:in Positive := 3; --Has to be typed / not const
package Class_Better_Stack is
  package Class_Stack_T is new Class_Stack(T,Max_Stack);
  package Class_Stack_T_Additions is new Class_Stack_T.Additions;

  type Better_Stack is new Class_Stack_T.Stack with private;

  procedure Push( The:in out Better_Stack; Item:in T );
  function  Max_Depth( The:in Better_Stack ) return Natural;
private
  type Better_Stack is new Class_Stack_T.Stack with record
    Depth : Natural := 0;
  end record;
end Class_Better_Stack;
```

```
package body Class_Better_Stack is

  procedure Push( The:in out Better_Stack; Item:in T ) is
    D : Natural;
  begin
    Class_Stack_T.Push( Class_Stack_T.Stack(The), Item );
    D := Class_Stack_T_Additions.Items(Class_Stack_T.Stack(The) );
    if D > The.Depth then
      The.Depth := The.Depth + 1;
    end if;
  end Push;

  function Max_Depth( The:in Better_Stack ) return Natural is
  begin
    return The.Depth;
  end Max_Depth;

end Class_Better_Stack;
```

```
with Class_Better_Stack;
  pragma Elaborate_All( Class_Better_Stack );
  package Class_Better_Stack_Pos is
          new Class_Better_Stack(Positive,10);
```

```
with Ada.Text_Io, Ada.Integer_Text_Io, Class_Better_Stack_Pos;
use  Ada.Text_Io, Ada.Integer_Text_Io, Class_Better_Stack_Pos;
procedure Main is
  Numbers : Better_Stack;
  Res     : Positive;
begin
  Put("Max depth  "); Put( Max_Depth( Numbers ) ); New_Line;
  Push( Numbers, 10 );
  Push( Numbers, 20 );
  Put("Max depth  "); Put( Max_Depth( Numbers ) ); New_Line;
  Push( Numbers, 20 );
  Put("Max depth  "); Put( Max_Depth( Numbers ) ); New_Line;
  Pop( Numbers, Res );
  Put("Max depth  "); Put( Max_Depth( Numbers ) ); New_Line;
  null;
end Main;
```

```
Max depth   0
Max depth   2
Max depth   3
Max depth   3
```

Dynamic memory allocation

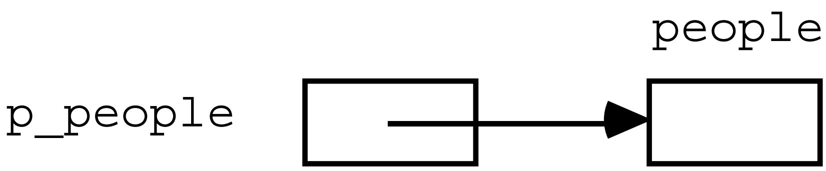
Access types

```
People    : aliased Integer;
```

```
People    := 24;
```

```
type P_Integer is access all Integer;

P_People : P_Integer;
```

```
P_People := People'Access;        --Access value for people
```

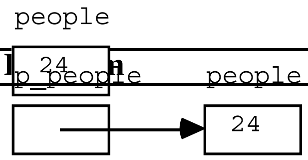
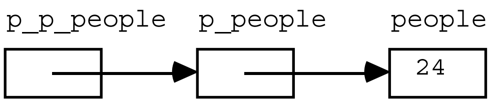



```
with Ada.Text_Io, Ada.Integer_Text_Io;
use  Ada.Text_Io, Ada.Integer_Text_Io;
procedure Main is
  type P_Integer is access all Integer;
  People    : aliased Integer;
  P_People : P_Integer;
begin
  People    := 24;
  P_People := People'Access;        --Access value for people
  Put("The number of people is : "); Put( P_People.all );
  New_Line;
End Main;
```

```
The number of people is :    24
```

```
declare
  type P_Integer   is access all Integer;
  type P_P_Integer is access all P_Integer;
  P_P_People : P_P_Integer;
  P_People   : aliased P_Integer;
  People     : aliased Integer;
begin
  People     := 42;
  P_People   := People'Access;
  P_P_People := P_People'Access;

end;
```

the following expressions will deliver the contents of the object `people`
fix

Expression	Explanation
`People`	
`P_People.all`	
`P_P_People.all.all`	

In a similar way the following statements will assign 42 to the object `people`.

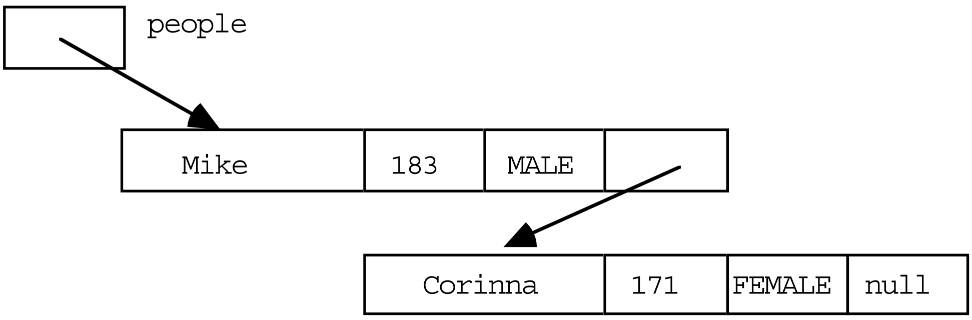
Statement	Explanation
`People := 42;`	Straight-forward assignment.
`P_People.all := 42;`	Single level of indirection.
`P_P_People.all.all := 42;`	Double level of indirection.

Allocating memory dynamically

```
declare
  Max_Chs : constant := 7;
  type Gender    is ( Female, Male );
  type Height_Cm is range 0 .. 300;
  type Person is record
    Name    : String( 1 .. Max_Chs );   --Name as a String
    Height  : Height_Cm := 0;           --Height in cm.
    Sex     : Gender;                   --Gender of person
  end record;
  type P_Person is access Person;      --Access type
  P_Mike  : P_Person;
begin
  P_Mike    := new Person'("Mike   ", 183, Male);
end;
```

Building a linked list

```
declare
  Max_Chs : constant := 7;
  type Gender    is ( Female, Male );
  type Height_Cm is range 0 .. 300;
  type Person;                          --Incomplete declaration
  type P_Person is access Person;       --Access type
  type Person is record
    Name   : String( 1 .. Max_Chs );    --Name as a String
    Height : Height_Cm := 0;            --Height in cm.
    Sex    : Gender;                    --Gender of person
    Next   : P_Person;
  end record;
  People : P_Person;
begin
  People := new Person'("Mike   ", 183, Male, null );
  People.Next := new Person'("Corinna", 171, Female, null);

end;
```



Navigating a linked list

```
procedure Put( Crowd: in P_Person ) is
  Cur : P_Person := Crowd;
begin
  while Cur /= null loop
    Put( Cur.Name ); Put(" is ");
    Put( Integer(Cur.Height), Width=>3 ); Put("cm and is ");
    if Cur.Sex = Female then
      Put("female");
    else
      Put("male");
    end if;
    New_Line;
    Cur := Cur.Next;
  end loop;
end Put;
```

```
procedure Put( Cur: in P_Person ) is
begin
  if Cur /= null then
    Put( Cur.Name ); Put(" is ");
    Put( Integer(Cur.Height), Width=>3 ); Put("cm and is ");
    if Cur.Sex = Female then
      Put("female");
    else
      Put("male");
    end if;
    New_Line;
    Put( Cur.Next );
  end if;
end Put;
```

```
Mike    is 183cm and is male
Corinna is 171cm and is female
```

Note	Declaration (T is an `Integer` type)	Example of use

1	`type P_T is access all T;`
	`a_t : aliased T;`
	`a_pt: P_T;`
		`a_pt     := a_t'Access;`
		`a_pt.all := 2;`

2	`type P_T is access constant T;`
	`a_t : aliased constant T := 0;`
	`a_pt: P_T;`
		`a_pt     := a_t'Access;`
		`Put( a_pt.all );`

3	`type P_T is access T;` `a_pt: P_T;`	`a_pt     := new T;` `a_pt.all := 2;`

Note 1: *Used when it is required to have both read and write access to* `a_t` *using the access value held in* `a_pt`*. The storage described by* `a_t` *may also be dynamically created using an allocator.*

Note 2: *Used when it is required to have only read access to* `a_t` *using the access value held in* `a_pt`*. The storage described by* `a_t` *may also be dynamically created using an allocator.*

Note 3: *Used when the storage for an instance of a* `T` *is allocated dynamically. Access to an instance of* `T` *can be read or written to using the access value obtained from* **`new`**.
*This form may only be used when an access value is created with an allocator (***`new`** `T`*).*

Problem	Result
Memory leak	The storage that is allocated is not always returned to the system. For a program which executes for a long time, this can result in eventual out of memory error messages.
Accidentally using the same storage twice for different data items.	This will result in corrupt data in the program and probably a crash which is difficult to understand.
Corruption of the chained data structure holding the data.	Most likely a program crash will occur some time after the corruption of the data structure.
Time taken to allocate and de-allocate storage is not always constant.	There may be unpredictable delays in a real-time system. However a worst case Figure can usually be calculated.

Process	Advantages	Disadvantages
Storage reclamation implicitly managed by the system.	No problem about de-allocating active storage.	May result in a program consuming large amounts of storage even though its actual use of storage is small. In extreme cases this may prevent a program from continuing to run.
Storage de-allocation explicitly initiated by a programmer.	Prevents inactive storage consuming program address space.	If the programmer makes an error in the de-allocation then this may be very difficult to track down.

Generic stack using a linked list

```
generic
  type Stack_Element is private;          --

package Class_Stack is
  type Stack is limited private;          --NO copying
  Stack_Error : exception;

  procedure Push( The:in out Stack; Item:in Stack_Element );
  procedure Pop(The:in out Stack; Item :out Stack_Element );
  procedure Reset( The:in out Stack );
private

  type Node;                              --Mutually recursive def
  type P_Node is access Node;             --Pointer to a Node
  pragma Controlled( P_Node );            --We do deallocation

  type Node is record                     --Node holds the data
    Item   : Stack_Element;               --The stored item
    P_Next : P_Node;                      --Next in list
  end record;

  type Stack is record
    P_Head : P_Node := null;              --First node in list
  end record;
end Class_Stack;
```

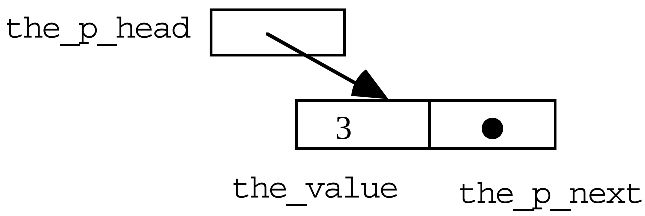
```
with Unchecked_Deallocation;
pragma Elaborate_All( Unchecked_Deallocation );
package body Class_Stack is

  procedure Dispose is
    new Unchecked_Deallocation( Node, P_Node );
```

FIX

Linked list

An Empty list

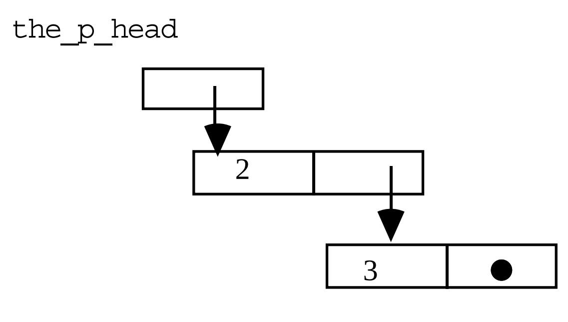
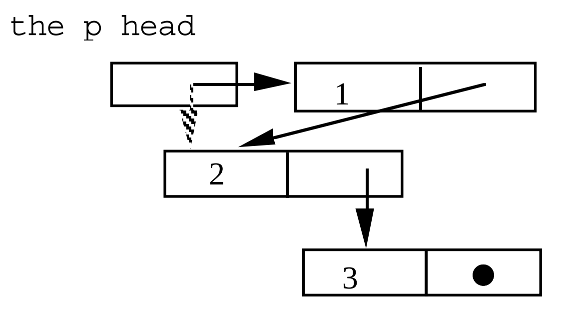
A list of 1 item



Accessing a data value

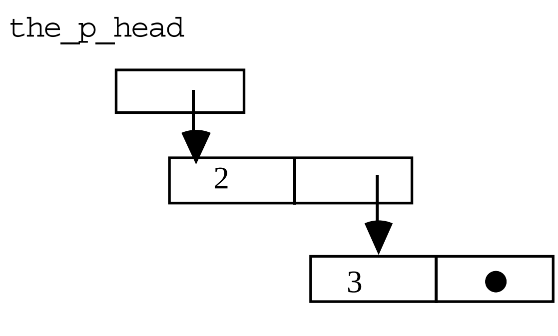
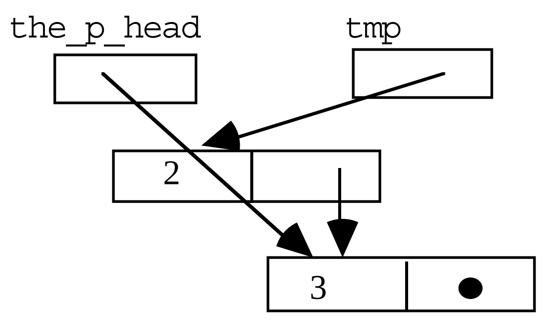
```
P_Head.Item = 3;
```

Adding an item to a linked list

The chain of elements after adding 3 and 2 to an instance of a `Integer Stack`	The chain of elements after pushing 1 on to an instance of a `Integer Stack`
	

```
procedure Push( The:in out Stack; Item:in Stack_Element ) is
  Tmp : P_Node;                         --Allocated node
begin
  Tmp := new Node'(Item=>Item, P_Next=>The.P_Head);
  The.P_Head := Tmp;
end Push;
```

Removing an item from a linked list

The chain of elements after adding 3 and 2 to an instance of the `Stack`	The chain of elements after popping the top element
	

```ada
procedure Pop( The:in out Stack; Item :out Stack_Element ) is
  Tmp : P_Node;                              --Free node
begin
  if The.P_Head /= null then                 --if item then
    Tmp := The.P_Head;                       --isolate top node
    Item := The.P_Head.Item;                 --extract item stored
    The.P_Head := The.P_Head.P_Next;         --Relink
    Dispose( Tmp );                          --return storage
  else
    raise Stack_Error;                       --Failure
  end if;
end Pop;
```

Clening up the stack

```ada
procedure Reset( The:in out Stack ) is
  Tmp : Stack_Element;
begin
  while The.P_Head /= null loop              --Re-initialize stack
    Pop( The, Tmp );
  end loop;
end Reset;

end Class_Stack;
```

Testing the stack

```
with Ada.Text_Io, Ada.Integer_Text_Io, Class_Stack_Int;
use  Ada.Text_Io, Ada.Integer_Text_Io, Class_Stack_Int;
procedure Main is
  Number_Stack : Stack;                  --Stack of numbers
  Action       : Character;              --Action
  Number       : Integer;                --Number processed
begin
  Reset( Number_Stack );                 --Reset stack to empty
  while not End_Of_File loop
    while not End_Of_Line loop
      begin
        Get( Action );
        case Action is                   --Process action
          when '+' =>
            Get( Number ); Push(Number_Stack,Number);
            Put("push number = "); Put(Number); New_Line;
          when '-' =>
            Pop(Number_Stack,Number);
            Put("Pop number  = "); Put(Number); New_Line;
          when others =>
            Put("Invalid action"); New_Line;
        end case;
      exception
        when Stack_Error =>
          Put("Stack_error"); New_Line;
        when Data_Error  =>
          Put("Not a number"); New_Line;
        when End_Error   =>
          Put("Unexpected end of file"); New_Line; exit;
      end;
    end loop;
    Skip_Line;
  end loop;

  Reset( Number_Stack );
end Main;
```

```
push number = 1
push number = 2
push number = 3
push number = 4
Pop number = 4
Pop number = 3
Pop number = 2
Pop number = 1
Pop: Exception Stack_error
```

Hiding the structure of an object (opaque type)

```
with Ada.Finalization;
use  Ada.Finalization;
package Class_Account is
type Account is new Limited_Controlled with private;
  subtype Money  is Float;
  subtype Pmoney is Float range 0.0 .. Float'Last;
  procedure Initialize( The:in out Account );
  procedure Finalize ( The:in out Account );
  procedure Deposit  ( The:in out Account; Amount:in Pmoney );
  procedure Withdraw ( The:in out Account; Amount:in Pmoney;
                       Get:out Pmoney );
  function  Balance  ( The:in Account ) return Money;
private
  type Actual_Account;                        --Details In body
  type P_Actual_Account is access all Actual_Account;
  type Account is new Limited_Controlled with record
    Acc : P_Actual_Account;                   --Hidden in body
  end record;
end Class_Account;
```

```
with Unchecked_Deallocation;
package body Class_Account is

  pragma Controlled( P_Actual_Account );  -- We do deallocation
  type Actual_Account is record           --Hidden declaration
    Balance_Of : Money := 0.00;          --Amount in account
  end record;
```

```
  procedure Dispose is
    new Unchecked_Deallocation(Actual_Account, P_Actual_Account);

  procedure Initialize( The:in out Account ) is
  begin
    The.Acc := new Actual_Account;       --Allocate storage
  end Initialize;

  procedure Finalize ( The:in out Account ) is
  begin
    if The.Acc /= null then              --Release storage
      Dispose(The.Acc); The.Acc:= null;  --Note can be called
    end if;                              -- more than once
  end Finalize;
```

```
  procedure Deposit ( The:in out Account; Amount:in Pmoney ) is
  begin
    The.Acc.Balance_Of := The.Acc.Balance_Of + Amount;
  end Deposit;

  procedure Withdraw( The:in out Account; Amount:in Pmoney;
                      Get:out Pmoney ) is
  begin
    if The.Acc.Balance_Of >= Amount then
      The.Acc.Balance_Of := The.Acc.Balance_Of - Amount;
      Get := Amount;
    else
      Get := 0.00;
    end if;
  end Withdraw;

function  Balance( The:in Account ) return Money is
  begin
    return The.Acc.Balance_Of;
  end Balance;

end Class_Account;
```

Opaque type: Example

```
with Ada.Text_Io, Class_Account, Statement;
use  Ada.Text_Io, Class_Account;
procedure Main is
  My_Account:Account;
  Obtain     :Money;
begin
  Statement( My_Account );

  Put("Deposit £100.00 into account"); New_Line;
  Deposit( My_Account, 100.00 );
  Statement( My_Account );

  Put("Withdraw £80.00 from account"); New_Line;
  Withdraw( My_Account, 80.00, Obtain );
  Statement( My_Account );

  Put("Deposit £200.00 into account"); New_Line;
  Deposit( My_Account, 200.00 );
  Statement( My_Account );
end Main;
```

```
Mini statement: The amount on deposit is £ 0.00

Deposit £100.00 into account
Mini statement: The amount on deposit is £100.00

Withdraw £80.00 from account
Mini statement: The amount on deposit is £20.00
```

Access value of a function

```
type P_Fun is access function(Item:in Float) return Float;
type Vector is array ( Integer range <> ) of Float;

procedure Apply( F:in P_Fun; To:in out Vector ) is
begin
  for I in To'range loop
    To(I) := F( To(I) );
  end loop;
end Apply;
```

```
function Square( F:in Float ) return Float is
begin
  return F * F;
end Square;

function Cube( F:in Float ) return Float is
begin
  return F * F * F;
end Cube;
```

```
with Ada.Text_Io, Ada.Float_Text_Io;
use  Ada.Text_Io, Ada.Float_Text_Io;
procedure Main is
  type P_Fun is access function(Item:in Float) return Float;
  type Vector is array ( Integer range <> ) of Float;

  -- Body of the procedures apply, square and float

  procedure Put( Items:in Vector ) is
  begin
    for I in Items'Range loop
      Put( Items(I), Fore=>4, Exp=>0, Aft=>2 ); Put(" ");
    end loop;
  end Put;
begin
  Numbers := (1.0, 2.0, 3.0, 4.0, 5.0);
  Put("Square list :");
  Apply( Square'access, Numbers );
  Put( Numbers ); New_Line;
  Numbers := (1.0, 2.0, 3.0, 4.0, 5.0);
  Put("cube list   :");
  Apply( Cube'access, Numbers );
  Put( Numbers ); New_Line;
end Ex2;
```

```
Square list :   1.00    4.00    9.00   16.00   25.00
cube list   :   1.00    8.00   27.00   64.00  125.00
```

Attributes 'Access and 'Unchecked_Access

```
procedure Main is
  Max_Chs : constant := 7;
  type Height_Cm is range 0 .. 300;
  type Person is record
    Name   : String( 1 .. Max_Chs );  --Name as a String
    Height : Height_Cm := 0;          --Height in cm.
  end record;
  Mike   : aliased Person := Person'("Mike   ",156);
begin
  declare
    type P_Person is access all Person; --Access type
    P_Human: P_Person;
  begin
    P_Human:= Mike'access;              --OK
    declare
      Clive : aliased Person := Person'("Clive  ", 160 );
    begin
      P_Human := Clive'Access;
    end;
    Put( P_Human.Name ); New_Line;      --Clive no loner exists
    P_Human := Mike'access;             --Change to Mike
  end;
end Main;
```

a compile time error message is generated for the line:

```
P_Human := Clive'Access;          -- Compile time error
```

as the object `clive` does not exist for all the scope of the type `P_Person`. In fact, there is a serious error in the program, as when the line:

```
Put( P_Human.Name ); New_Line;      -- Clive no longer exists
```

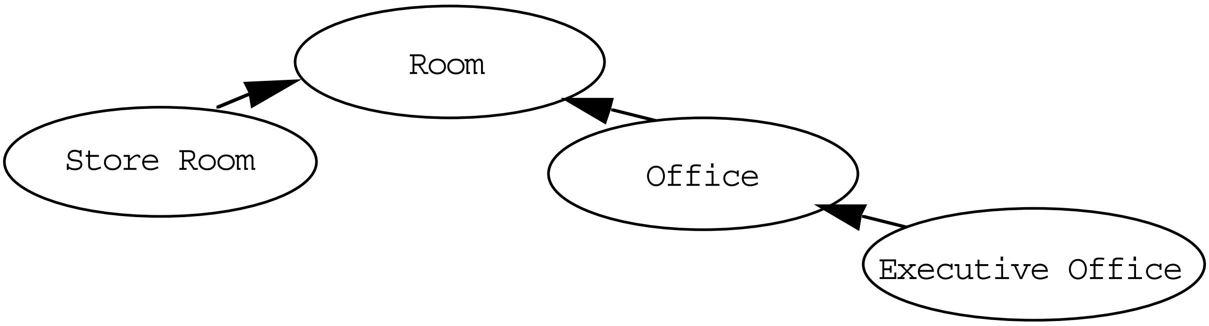
is executed, the storage that `p_human` points to does not exist. Remember the scope of `clive` is the **`declare`** block.

In some circumstances the access value of an object declared in an inner block to the access type declaration is required. If this is so, then the compiler checking can be subverted or overridden by the use of `'Unchecked_Access`.

Use of `Unchecked_Access`

```
procedure Main is
  -- Declaration of Person, P_Person, Mike etc.
begin
    P_Human:= Mike'Access;                            -- OK
    declare
      Clive : aliased Person := Person'("Clive  ", 160 );
    begin
      P_Human := Clive'Unchecked_Access;
      Put( P_Human.Name ); New_Line;                  -- Clive
    end;
    P_Human:= Mike'Access;                            --Change to Mike
    Put( P_Human.Name ); New_Line;                    --Mike
end Main;
```

Polymorphism



Class wide type	Can describe an instance of the following types
Room'Class	Room, Office, Executive_Office or Store_room
Office'Class	Office or Executive_Office
Executive_Office'Class	Executive_Office
Store_Room'Class	Store_Room

```
  if W422'Tag = W414'Tag then
    Put("Areas are the same type of accommodation");
    New_Line;
  end if;
```

Method	Responsibility
Initialize	Store a description of the room.
Describe	Deliver a string containing a description of the room.
Where	Deliver the room's number.

```
with Ada.Strings.Bounded;
use  Ada.Strings.Bounded;
package B_String is new Generic_Bounded_Length( 80 );
```

```
with B_String; use B_String;
package Class_Room is
  type Room   is tagged private;

  procedure Initialize( The:in out Room; No:in Positive;
                        Mes:in String );
  function  Where( The:in Room ) return Positive;
  function  Describe( The:in Room ) return String;
private
  type Room is tagged record
    Desc  : Bounded_String;      --Description of room
    Number: Positive;            --Room number
  end record;
end Class_Room;
```

```
with Ada.Integer_Text_Io;
use  Ada.Integer_Text_Io;
package body Class_Room is

  procedure Initialize( The:in out Room;
                        No:in Positive; Mes:in String ) is
  begin
    The.Desc := To_Bounded_String( Mes );
    The.Number := No;
  end Initialize;

  function  Where( The:in Room ) return Positive is
  begin
    return The.Number;
  end Where;

  function  Describe( The:in Room ) return String is
    Num : String( 1 .. 4 );    --Room number as string
  begin
    Put( Num, The.Number );
    return Num & " " & To_String(The.Desc);
  end Describe;

end Class_Room;
```

Method	Responsibility
Initialize	Store a description of the office plus the number of occupants
Describe	Returns a String describing an office.
No_Of_People	Return the number of people who occupy the room.

```
with Class_Room; use  Class_Room;
package Class_Office is
  type Office is new Room with private;

  procedure Initialize( The:in out Office; No:in Positive;
                        Desc:in String; People:in Natural );
  function  Deliver_No_Of_People(The:in Office) return Natural;
  function  Describe( The:in Office ) return String;
private
  type Office is new Room with record
    People : Natural := 0;          --Occupants
  end record;
end Class_Office;
```

```
with Ada.Integer_Text_Io;
use  Ada.Integer_Text_Io;
package body Class_Office is

  procedure Initialize( The:in out Office; No:in Positive;
                        Desc:in String; People:in Natural ) is
  begin
    Initialize( The, No, Desc );
    The.People := People;
  end Initialize;

  function Deliver_No_Of_People( The:in Office ) return Natural is
  begin
    return The.People;
  end Deliver_No_Of_People;

  function  Describe( The:in Office ) return String is
    No : String( 1 .. 4 );    --the.people as string
  begin
    Put( No, The.People );
    return Describe( Room(The) ) &
                   " occupied by" & No & " people";
  end Describe;

end Class_Office;
```

Polymorphism: Example

```
with Ada.Text_Io, Class_room, Class_Office;
use  Ada.Text_Io, Class_room, Class_Office;
procedure Main is
  W422 : Room;
  W414 : Office;
```

```
  procedure About( Place:in Room'Class ) is
  begin
    Put( "The place is" ); New_Line;
    Put( "   " & Describe( Place ) ) ;  --Run time dispatch
    New_Line;
  end About;
```

```
begin
  Initialize( W414, 414, "4th Floor west wing", 2 );
  Initialize( W422, 422, "4th Floor east wing" );

  About( W422 );                        --Call with a room
  About( W414 );                        --Call with an Office

end Main;
```

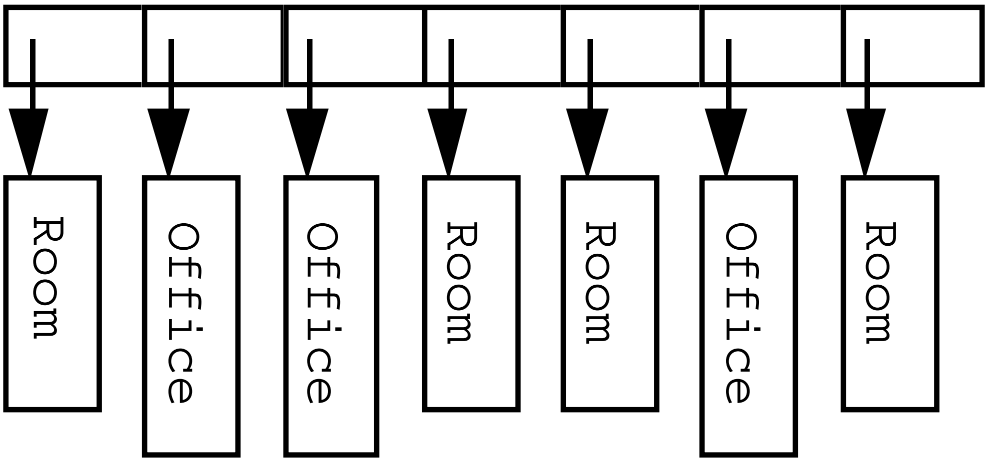
```
The place is
   422 4th Floor east wing
The place is
   414 4th Floor west wing occupied by   2 people
```

Run time dispatch

In class Room	**function** Describe(The:**in** Room) **return** String;
In class Office	**function** Describe(The:**in** Office) **return** String;

In class Room	**In class Office**
Initialize(Room,String)	Initialize(Office,String)
Describe(Room) -> String	Describe(Office) -> String
	No_Of_People(Room) -> Integer
	Initialize(Office,String,Natural)

Heterogeneous collections of objects




```
type P_Room      is access all Room'Class;
type Rooms_Array is array (1 .. 15) of P_Room;
```

```
declare
  P           : P_Room;
  Accommodation: Rooms_Array;
begin
  P := new Room;
  Initialize( P.all, 422, "4th Floor east wing" );
  Accommodation(1) := P;
end;
```

Additions to the class Office and Room

Method	Responsibility
Build_Room	Deliver an access value to a dynamically created Room.
Build_Office	Deliver an access value to a dynamically created Office.

```
package Class_Room.Build is
  type P_Room is access all Room'Class;

  function  Build_Room( No:in Positive;
                        Desc:in String ) return P_Room;
end Class_Room.Build;
```

```
package body Class_Room.Build is

  function  Build_Room( No:in Positive;
                        Desc:in String ) return P_Room is
    P : P_Room;
  begin
    P := new Room; Initialize( P.all, No, Desc );
    return P;
  end Build_Room;

end Class_Room.Build;
```

```
with Class_Room, Class_Room.Build;
use  Class_Room, Class_Room.Build;
package Class_Office.Build is

  function  Build_Office( No:in Positive; Desc:in String;
                          People:in Natural ) return P_Room;
end Class_Office.Build;
```

```
package body Class_Office.Build is

  type P_Office is access all Office;

  function  Build_Office( No:in Positive; Desc:in String;
                          People:in Natural ) return P_Room is
    P : P_Office;
  begin
    P := new Office; Initialize( P.all, No, Desc, People );
    return P.all'access;
  end Build_Office;
end Class_Office.Build;
```

Method	Responsibility
add	Add a description of a room
about	Return a description of a specific room

```
with Class_Room, Class_Room.Build;
use  Class_Room, Class_Room.Build;
package Class_Building is

  type Building is tagged private;

  procedure Add( The:in out Building; Desc:in P_Room );
  function About(The:in Building; No:in Positive) return String;

private
  Max_Rooms : constant := 15;
  type    Rooms_Index is range 0 .. Max_Rooms;
  subtype Rooms_Range is Rooms_Index range 1 .. Max_Rooms;
  type    Rooms_Array is array (Rooms_Range) of P_Room;

  type Building is tagged record
    Last        : Rooms_Index := 0;  --Last slot allocated
    Description : Rooms_Array;       --Rooms in building
  end record;
end Class_Building;
```

```
package body Class_Building is

  procedure Add( The:in out Building; Desc:in P_Room ) is
  begin
    if The.Last < Max_Rooms then
      The.Last := The.Last + 1;
      The.Description( The.Last ) := Desc;
    else
      raise Constraint_Error;
    end if;
  end Add;
```

```
  function About(The:in Building; No:in Positive) return String is
  begin
    for I in 1 .. The.Last loop
      if Where(The.Description(I).all) = No then
        return Describe(The.Description(I).all);
      end if;
    end loop;
    return "Sorry room not known";
  end About;
end Class_Building;
```

Putting it all together

```
with Ada.Text_Io,Ada.Integer_Text_Io,Class_Room,Class_Room.Build,
     Class_Office, Class_Office.Build, Class_Building;
use  Ada.Text_Io,Ada.Integer_Text_Io,Class_Room,Class_Room.Build,
     Class_Office, Class_Office.Build, Class_Building;
procedure Set_Up( Watts:in out Building ) is
begin
  Add( Watts, Build_Office( 414, "4th Floor west wing", 2 ) );
  Add( Watts, Build_Room  ( 422, "4th Floor east wing" ) );
end Set_Up;
```

```
with Ada.Text_Io, Ada.Integer_Text_Io, Class_Building, Set_Up;
use  Ada.Text_Io, Ada.Integer_Text_Io, Class_Building;
procedure Main is
  Watts    : Building;                         --Watts Building
  Room_No  : Positive;                         --Queried room
begin
```

```
  Set_Up( Watts );                             --Populate building
  loop
    begin
      Put( "Inquiry about room: " );           --Ask
      exit when End_Of_File;
      Get( Room_No ); Skip_Line;               --User response
      Put( About( Watts, Room_No ) );
      New_Line;                                --Display answer
    exception
      when Data_Error =>
        Put("Please retype the number");       --Ask again
        New_Line; Skip_Line;
    end;
  end loop;
end Main;
```

```
Inquiry about room: 414
414 4th Floor west wing occupied by   2 people
Inquiry about room: 422
422 4th Floor east wing
Inquiry about room: 999
Sorry room not known
^D
```

Fully qualified names and polymorphism

```
with Class_Room, Class_Room.Build;
package Class_Building is

  type Building is tagged private;

  procedure Add( The:in out Building;
                 Desc:in Class_Room.Build.P_Room );
  function About(The:in Building; No:in Positive) return String;

private
  Max_Rooms : constant := 15;
  type    Rooms_Index is range 0 .. Max_Rooms;
  subtype Rooms_Range is Rooms_Index range 1 .. Max_Rooms;
  type    Rooms_Array is array (Rooms_Range) of
                          Class_Room.Build.P_Room;

  type Building is tagged record
    Last        : Rooms_Index := 0;  --Last slot allocated
    Description : Rooms_Array;       --Rooms in building
  end record;
end Class_Building;
```

```
package body Class_Building is

  procedure Add( The:in out Building;
                 Desc:in Class_Room.Build.P_Room ) is
  begin
    if The.Last < Max_Rooms then
      The.Last := The.Last + 1;
      The.Description( The.Last ) := Desc;
    else
      raise Constraint_Error;
    end if;
  end Add;

  function About(The:in Building; No:in Positive) return String is
  begin
    for I in 1 .. The.Last loop
      if Class_Room.Where(The.Description(I).all) = No then
        return Class_Room.Describe(The.Description(I).all);
      end if;
    end loop;
    return "Sorry room not known";
  end About;
end Class_Building;
```

Downcasting

```
with Ada.Text_Io,Ada.Integer_Text_Io,Class_Room, Class_Room.Build,
     Class_Office, Class_Office.Build, Ada.Tags;
use  Ada.Text_Io,Ada.Integer_Text_Io,Class_Room, Class_Room.Build,
     Class_Office, Class_Office.Build, Ada.Tags;
procedure Main is
  Max_Rooms : constant := 3;
  type     Rooms_Index  is range 0 .. Max_Rooms;
  subtype  Rooms_Range  is Rooms_Index range 1 .. Max_Rooms;
  type     Rooms_Array  is array ( Rooms_Range ) of P_Room;
  type     Office_Array is array ( Rooms_Range ) of Office;
  Accommodation : Rooms_Array;    --Rooms and Offices
  Offices       : Office_Array;   --Offices only
  No_Offices    : Rooms_Index;
begin
  Accommodation(1):=Build_Office(414, "4th Floor west wing", 2);
  Accommodation(2):=Build_Room  (518, "5th Floor east wing");
  Accommodation(3):=Build_Office(403, "4th Floor east wing", 1);

  No_Offices := 0;
  for I in Rooms_Array'range loop
    if Accommodation(I).all'Tag = Office'Tag then
      No_Offices := No_Offices + 1;
      Offices(No_Offices) := Office(Accommodation(I).all);  --
    end if;
  end loop;

  Put("The offices are:" ); New_Line;
  for I in 1 .. No_Offices loop
    Put( Describe( Offices(I) ) ); New_Line;
  end loop;

end Main;
```

```
The offices are:
 414 4th Floor west wing occupied by   2 people
 403 4th Floor east wing occupied by   1 people
```

Converting a base class to a derived class

```
with Ada.Tags;
use  Ada.Tags;
procedure Main is
  Withdrawals_In_A_Week : constant Natural := 3;
  subtype Money     is Float;
  type Account      is tagged record
    Balance_Of : Money := 0.00;      --Amount in account
  end record;
  type Account_Ltd is new Account with record
    Withdrawals : Natural := Withdrawals_In_A_Week;
  end record;
  Normal      : Account;
  Restricted : Account_Ltd;
begin
  Normal      := ( Balance_Of => 20.0 );
  Restricted := ( Normal with 4 );
  Restricted := ( Normal with Withdrawals => 4 );
end Main;
```

The Observers responsibilities

Method	Responsibility
Update	Display the state of the observed object.

```
type Observer     is tagged private;

procedure Update( The:in Observer; What:in Observable'Class );
```

The responsibilities of the observable object

Method	Responsibility
Add_Observer	Add an observer to the observable object.
Delete_Observer	Removes an observer.
Notify_Observers	If the object has changed, tells all observers to update their view of the object..
Set_Changed	Sets a flag to indicate that the object has changed.

```
type Observable  is tagged private;

type P_Observer  is access all Observer'Class;

procedure Add_Observer    ( The:in out Observable;
                            O:in P_Observer );
procedure Delete_Observer( The:in out Observable;
                            O:in P_Observer );
procedure Notify_Observers( The:in out Observable'Class );
procedure Set_Changed( The:in out Observable );
```

```
package Class_Observe_Observer is
  type Observable  is tagged private;
  type Observer    is tagged private;

  type P_Observer  is access all Observer'Class;

  procedure Add_Observer    ( The:in out Observable;
                              O:in P_Observer );
  procedure Delete_Observer( The:in out Observable;
                              O:in P_Observer );
  procedure Notify_Observers( The:in out Observable'Class );
  procedure Set_Changed( The:in out Observable );

  procedure Update( The:in Observer; What:in Observable'Class );
private
  Max_Observers : constant := 10;
  subtype Viewers_Range is Integer       range 0 .. Max_Observers;
  subtype Viewers_Index is Viewers_Range range 1 .. Max_Observers;
  type    Viewers_Array is array( Viewers_Index ) of P_Observer;
  type Observable is tagged record
    Viewers       : Viewers_Array := ( Others => null );
    Last          : Viewers_Range := 0;
    State_Changed : Boolean := True;
  end record;

  type Observer is tagged null record;

end Class_Observe_Observer;
```

```
package body Class_Observe_Observer is
```

```
  procedure Add_Observer( The:in out Observable;
                          O:in P_Observer ) is
  begin
    for I in 1 .. The.Last loop             --Check for empty slot
      if The.Viewers( I ) = null then
        The.Viewers( I ) := O;              --Populate
        return;
      end if;
    end loop;
    if The.Last >= Viewers_Index'Last then --Extend
      raise Constraint_Error;               -- Not enough room
    else
      The.Last := The.Last + 1;             -- Populate
      The.Viewers( The.Last ) := O;
    end if;
  end Add_Observer;
```

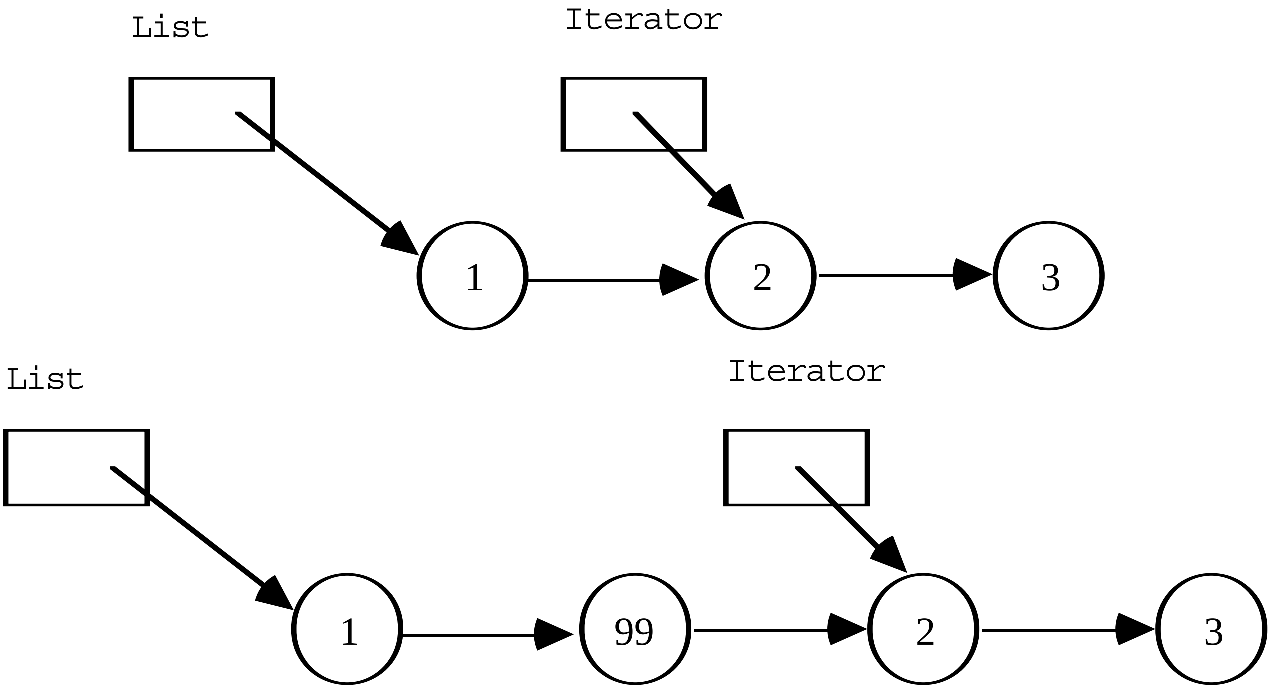
```
  procedure Delete_Observer( The:in out Observable;
                             O:in P_Observer ) is
  begin
    for I in 1 .. The.Last loop             --For each observer
      if The.Viewers( I ) = O then          --Check if to be removed
        The.Viewers( I ) := null;
      end if;
    end loop;
  end Delete_Observer;
```

```
  procedure Notify_Observers( The:in out Observable'Class ) is
  begin
    for I in 1 .. The.Last loop              --For each observer
      if The.Viewers( I ) /= null then       -- call it's
        Update( The.Viewers( I ).all, The );-- update method
      end if;
    end loop;
    The.State_Changed := True;               --
  end Notify_Observers;
```

```
procedure Set_Changed( The:in out Observable ) is
begin
  The.State_Changed := True;
end Set_Changed;
```

```
procedure Update( The:in Observer; What:in Observable'Class ) is
begin
  null;                                     --Should be overridden
end Update;
```

Containers (List)



Criteria	List	Array
The number of items held can be increased at run-time.	✔	✗
Deletion of an item leaves no gap when the items are iterated through.	✔	✗
Random access is very efficient.	✗	✔

```
with Class_List;
 pragma Elaborate_All( Class_List );
 package Class_List_Nat is new Class_List(Natural);

with Class_List_Nat, Class_List.Iterator;
 pragma Elaborate_All( Class_List_Nat, Class_List.Iterator );
 package Class_List_Nat_Iterator is new Class_List_Nat.Iterator;

with Ada.Text_Io, Ada.Integer_Text_Io, Class_List_Nat,
     Class_List_Nat_Iterator;
use  Ada.Text_Io, Ada.Integer_Text_Io,
     Class_List_Nat, Class_List_Nat_Iterator;
procedure Main is
  Numbers    : List;
  Numbers_It : List_Iter;
  Value      : Integer;
begin
  Value := 1;
  While Value <= 10 loop
    Last( Numbers_It, Numbers );            --Set iterator Last
    Next( Numbers_It );                     --Move beyond last
    Insert( Numbers_It, Value );            --Insert before
    value := Value + 1;                     --Increment
  end loop;
  First(Numbers_It,Numbers);                --Set to start
  while not Is_End( Numbers_It ) loop       --Not end of list
    Put( Deliver(Numbers_It) , Width=>3);   -- Print
    Next( Numbers_It );                     --Next item
  end loop;
  New_Line;
end Main;
```

List

Method	Responsibility
`Initialize`	Initialize the container.
`Finalize`	Finish using the container object.
`Adjust`	Used to facilitate a deep copy.
`=`	Comparison of a list for equality.

List Iterator

Method	Responsibility
`Initialize`	Initialize the iterator.
`Finalize`	Finish using the iterator object.
`Deliver`	Deliver the object held at the position indicated by the iterator.
`First`	Set the current position of the iterator to the first object in the list.
`Last`	Set the current position of the iterator to the last object in the list.
`Insert`	Insert into the list an object before the current position of the iterator.
`Delete`	Remove and dispose of the object in the list which is specified by the current position of the iterator.
`Is_end`	Deliver true if the iteration on the container has reached the end.
`Next`	Move to the next item in the container and make that the current position.
`Prev`	Move to the previous item in the container and make that the current position.

Example of use

```
with Class_list;
  package Class_list_nat is new Class_list(Natural);

with Class_list_nat, Class_list.Iterator;
  package Class_list_nat_iterator is new Class_list_nat.Iterator;

with Class_List;
  pragma Elaborate_All( Class_List );
  package Class_List_Nat is new Class_List(Natural);

with Class_List_Nat, Class_List.Iterator;
  pragma Elaborate_All( Class_List_Nat, Class_List.Iterator );
  package Class_List_Nat_Iterator is new Class_List_Nat.Iterator;
```

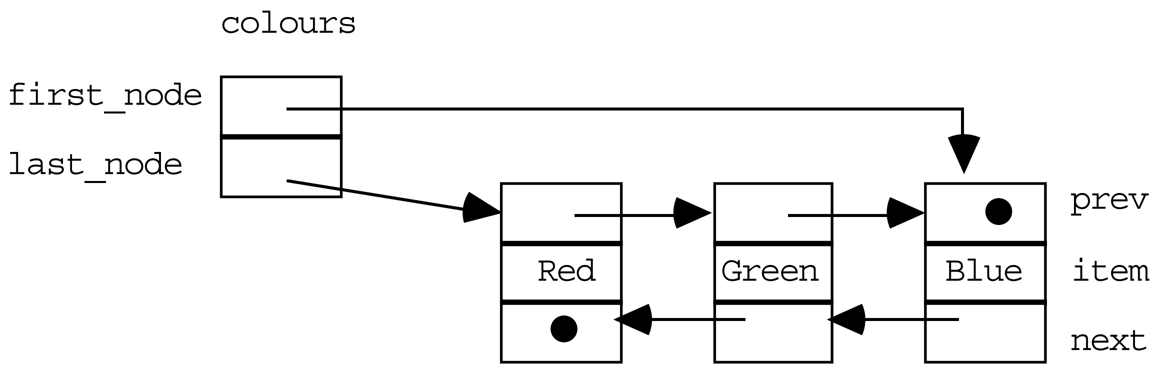
```
with Ada.Text_Io, Ada.Integer_Text_Io, Class_List_Nat,
     Class_List_Nat_Iterator;
use  Ada.Text_Io, Ada.Integer_Text_Io,
     Class_List_Nat, Class_List_Nat_Iterator;
procedure Main is
  Numbers     : List;
  Numbers_It : List_Iter;
  Num,In_List: Natural;
begin
  First( Numbers_It, Numbers );              --Setup iterator

  while not End_Of_File loop                 --While data
    while not End_Of_Line loop
      Get(Num);                              --Read number
      First(Numbers_It,Numbers);             --Iterator at start
      while not Is_End( Numbers_It ) loop    --scan through list
        In_List := Deliver(Numbers_It);
        exit when In_List > Num;             --Exit when larger no.
        Next( Numbers_It );                  --Next item
      end loop;
      Insert( Numbers_It, Num );             -- before curent number
    end loop;
    Skip_Line;                               --Next line
  end loop;
  Put("Numbers sorted are: ");
  First(Numbers_It,Numbers);                  --Set at start
  while not Is_End( Numbers_It ) loop
    In_List := Deliver( Numbers_It );         --Current number
    Put( In_List ); Put(" ");                 -- Print
    Next( Numbers_It );                       --Next number
  end loop;
  New_Line;
end Main;
```

```
1 8 6 2 4
```

```
Numbers sorted are:      2      4      6      8     10
```

```
with Ada.Finalization, Unchecked_Deallocation;
use  Ada.Finalization;
generic
  type T is private;                     --Any type
package Class_List is
  type List is new Controlled with private;

  procedure Initialize( The:in out List );
  procedure Initialize( The:in out List; Data:in T );
  procedure Finalize( The:in out List );
  procedure Adjust( The:in out List );
  function "="  ( F:in List; S:in List ) return Boolean;
private
  type Node;                           --Tentative declaration
  type P_Node is access all Node;      --Pointer to Node

  type Node is record
    Prev    : P_Node;                  --Previous Node
    Item    : T;                       --The physical item
    Next    : P_Node;                  --Next Node
  end record;

  type List is new Controlled with record
    First_Node : aliased P_Node := null;   --First item in list
    Last_Node  : aliased P_Node := null;   --First item in list
  end record;

end Class_List;
```

Data Structure (list)



Assignment of controlled objects	**Actions that take place on assignment**

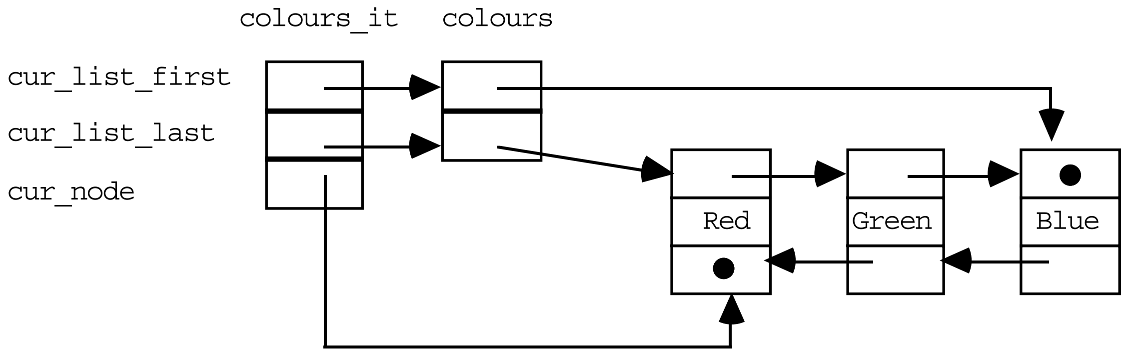
`A := B;`	`Anon := B;`
	`  Adjust( Anon );`
	`  Finalize( A );`
	`A := Anon;`
	`  Adjust( a );`
	`  Finalize( Anon );`

Action on assignment	**Commentary**
`Anon := B`	Make a temporary anonymous copy `anon`.
`Adjust(Anon);`	Adjustments required to be made after copying the direct storage of the source object `B` to `Anon`.
`Finalize(A);`	Finalize the target of the assignment.
`A := Anon;`	Perform the physical assignment of the direct components of the `anon` object.
`Adjust(A);`	Adjustments required to be made after copying the direct storage of the `Anon` object.
`Finalize(Anon);`	Finalize the anonymous object `Anon`.

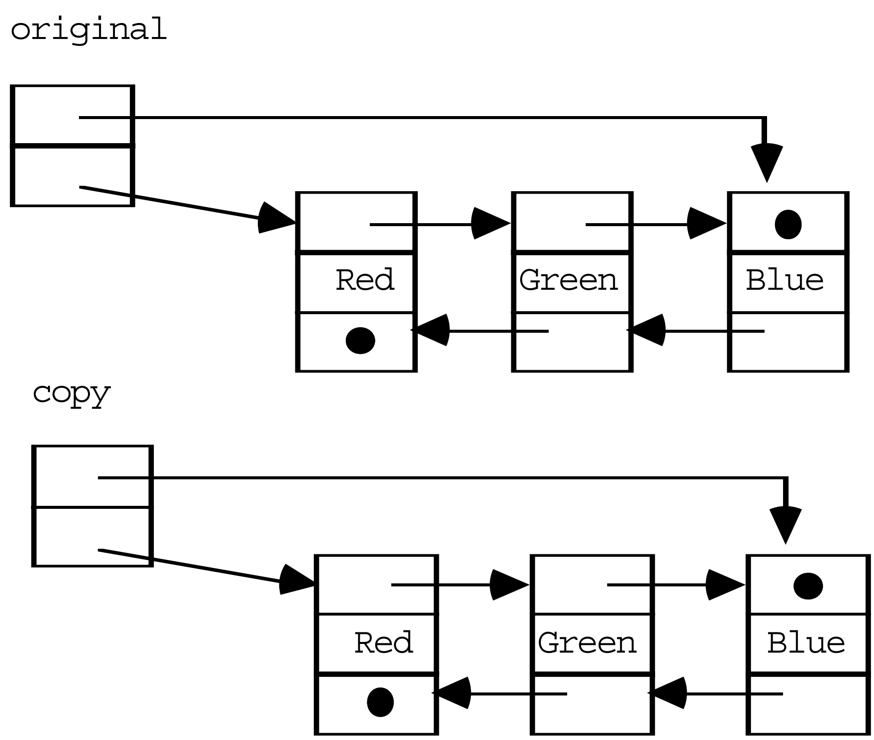
Iterator

```
generic
package Class_List.Iterator is

  type List_Iter is limited private;

  procedure First( The:in out List_Iter; L:in out List );
  procedure Last( The:in out List_Iter; L:in out List );

  function  Deliver( The:in List_Iter) return T;
  procedure Insert( The:in out List_Iter; Data:in T );
  procedure Delete( The:in out List_Iter );
  function  Is_End( The:in List_Iter ) return Boolean;
  procedure Next( The:in out List_Iter );
  procedure Prev( The:in out List_Iter );
private
  type P_P_Node is access all P_Node;
  type List_Iter is record
    Cur_List_First: P_P_Node := null;     --First in chain
    Cur_List_Last : P_P_Node := null;     --Last in chain
    Cur_Node      : P_Node   := null;     --Current item
  end record;
end Class_List.Iterator;
```

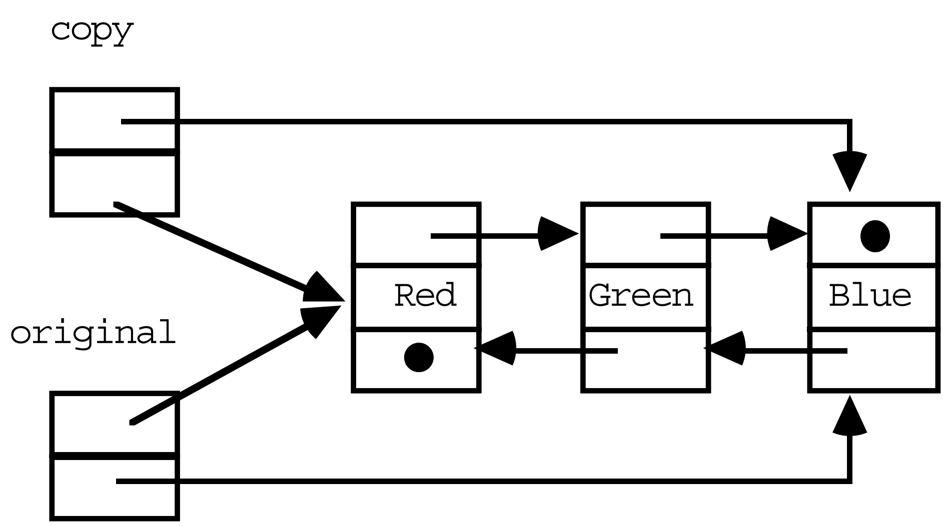
Iterator data structure



Deep copy



Shallow copy



A set

Method	Responsibility
Put	Display the contents of the set.
+	Form the union of two sets
Set_Const	Return a set with a single member.
Members	Return the numbers of items in the set.

```
with Class_List, Class_List.Iterator;
pragma Elaborate_All( Class_List, Class_List.Iterator );
generic
   type T is private;
   with procedure Put( Item:in T ) is <>;
   with function  ">" (First,Second:in T ) return Boolean is <>;
   with function  "<" (First,Second:in T ) return Boolean is <>;
package Class_Set is
   type Set is private;
   procedure Put( The:in Set );
   function "+"( F:in Set; S:in Set ) return Set;
   function Set_Const( Item: in T )   return Set;
   function Members( The:in Set )     return Positive;
private
   package Class_List_T          is new Class_List(T);
   package Class_List_T_Iterator is new Class_List_T.Iterator;
   type Set is new Class_List_T.List with record
     Elements : Natural := 0;               --Elements in set
   end record;
end Class_Set;
```

Example of Use

```
package Pack_Types is
  type Filling is ( Cheese, Onion, Ham, Tomato );
end Pack_Types;

with Ada.Text_Io, Pack_Types;
use  Ada.Text_Io, Pack_Types;
procedure Put_Filling( C:in Filling ) is
begin
  Put( Filling'Image( C ) );
end Put_Filling;

with Pack_Types, Class_Set, Put_Filling;
use  Pack_Types;
pragma Elaborate_All( Class_Set );
package Class_Set_Sandwich is
  new Class_Set( T => Pack_Types.Filling, Put => Put_Filling );

with Pack_Types,Ada.Text_Io,Ada.Integer_Text_Io,Class_Set_Sandwich;
use  Pack_Types,Ada.Text_Io,Ada.Integer_Text_Io,Class_Set_Sandwich;
procedure Main is
  Sandwich : Class_Set_Sandwich.Set;
begin
  Sandwich := Sandwich + Set_Const(Cheese);
  Sandwich := Sandwich + Set_Const(Onion) ;
  Put("Contents of sandwich are : ");
  Put( Sandwich ); New_Line;
  Put("Number of ingredients is : ");
  Put( Members(Sandwich) ); New_Line;
  null;
end Main;
```

```
Contents of sandwich are : (CHEESE,ONION)
Number of ingredients is :           2
```

Input & Output

```
with Ada.Text_Io;
use  Ada.Text_Io;
procedure Main is
  Max_Mem      : constant := 4096;                      --Mb
  Max_Mips     : constant := 12000.0;                   --Mips
  Max_Clock    : constant := 4000.0;                    --Clock
  type Memory is range 0 .. Max_Mem;                    --Integer
  type Cpu    is (I64, I32, PowerPc);                   --Enum
  type Mips   is digits 8    range 0.0 .. Max_Mips;     --Float
  type Clock  is delta 0.01  range 0.0 .. Max_Clock;    --Fixed
  Mc_Mem       : Memory;          --Main memory
  Mc_Cpu       : Cpu;             --Type of CPU
  Mc_Mips      : Mips;            --Raw MIPS
  Mc_Clock     : Clock;           --Clock frequency

  package Class_Mem_Io   is new Ada.Text_Io.Integer_Io(Memory);
  package Class_Cpu_Io   is new Ada.Text_Io.Enumeration_Io(Cpu);
  package Class_Mips_Io  is new Ada.Text_Io.Float_Io(Mips);
  package Class_Clock_Io is new Ada.Text_Io.Fixed_Io(Clock);
begin
  declare
    use Class_Mem_Io, Class_Mips_Io, Class_Clock_Io, Class_Cpu_Io;
  begin
    Mc_Mem  := 512;       Mc_Cpu   := I64;
    Mc_Mips := 3000.0;    Mc_Clock := 1000.0;
    Put("Memory:"); Put( Mc_Mem );   New_Line;
    Put("CPU   :"); Put( Mc_Cpu );   New_Line;
    Put("Mips  :"); Put( Mc_Mips );  New_Line;
    Put("Clock :"); Put( Mc_Clock ); New_Line;
    Put("Memory:"); Put( Mc_Mem, Width=>3); New_Line;
    Put("CPU   :"); Put( Mc_Cpu, Width=>7, Set=>Upper_Case );
    New_Line;
    Put("Mips  :"); Put( Mc_Mips, Fore=>3, Aft=>2, Exp=>0 );
    New_Line;
    Put("Clock :"); Put( Mc_Clock,Fore=>3, Aft=>2, Exp=>0 );
    New_Line;
  end;
end Main;
```

```
Memory:  512
CPU   :I64
Mips  : 3.0000000E+03
Clock : 1000.00
Memory:512
CPU   :I64
Mips  :3000.00
Clock :1000.00
```

Reading & writing to files

```
with Text_Io;
use  Text_Io;
procedure Main is
  Fd        : Text_Io.File_Type;            --File descriptor
  File_Name: constant String:= "file.txt";--Name
  Ch        : Character;                    --Character read
begin
  Create( File=>Fd, Mode=>Out_File, Name=>File_Name );
  while not End_Of_File loop              --For each Line
    while not End_Of_Line loop            --For each character
      Get(Ch); Put(Fd, Ch);               --Read / Write character
    end loop;
    Skip_Line; New_line(Fd);              --Next line / new line
  end loop;
  Close(Fd);
exception
  when Name_Error =>
    Put("Can not create " & File_Name ); New_Line;
end Main;
```

```
with Text_Io;
use  Text_Io;
procedure Main is
  Fd        : Text_Io.File_Type;            --File descriptor
  File_Name: constant String:= "file.txt";--Name
  Ch        : Character;                    --Character read
begin
  Open( File=>Fd, Mode=>In_File, Name=>File_Name );
  while not End_Of_File(Fd) loop          --For each Line
    while not End_Of_Line(Fd) loop        --For each character
      Get(Fd, Ch); Put(Ch);               --Read / Write character
    end loop;
    Skip_Line(Fd); New_Line;              --Next line / new line
  end loop;
  Close(Fd);
exception
  when Name_Error =>
    Put("Can not open " & File_Name ); New_Line;
end Main;
```

```
with Ada.Text_io; use Ada.Text_io;
procedure main is
  type Number is Range 1 .. 10;
  fd       : Ada.Text_io.File_type;       -- File descriptor
  file_name: constant String:= "file.txt";-- Name
  package Pack_number_io is new
    Ada.Text_io.Integer_io( Number );
begin
  create( File=>fd, Mode=>APPEND_FILE, Name=>file_name );
  for i in Number loop
    Pack_number_io.put( fd, i ); new_line( fd );
  end loop;
  close(fd);
exception
  when Name_Error =>
    put("Cannot append to " & file_name ); new_line;
end main;
```

Reading and writing binary data

```
package Pack_Types is
  Max_Chs : constant := 10;
  type Gender    is ( Female, Male );
  type Height_Cm is range 0 .. 300;

  type Person is record
    Name    : String( 1 .. Max_Chs );  --Name as a String
    Height  : Height_Cm := 0;          --Height in cm.
    Sex     : Gender;                  --Gender of person
  end record;

  type    Person_Index is range 1 .. 3;
  subtype Person_Range is Person_Index;
  type    Person_Array is array ( Person_Range ) of Person;
end Pack_Types;
```

```
with Text_Io, Pack_Types, Sequential_Io;
use  Text_Io, Pack_Types;
procedure Main is
  File_Name: constant String:= "people.txt";--Name
  People    : Person_Array;
  package Io is new Sequential_Io( Person );
begin
  declare
    Fd        : Io.File_Type;             --File descriptor
  begin
    People(1) := (Name=>"Mike      ", Sex=>Male,   Height=>183);
    People(2) := (Name=>"Corinna   ", Sex=>Female, Height=>171);
    People(3) := (Name=>"Miranda   ", Sex=>Female, Height=> 74);
    Io.Create( File=>Fd, Mode=>Io.Out_File, Name=>File_Name );
    for I in Person_Range loop
      Io.Write( Fd, People(I) );
    end loop;
    Io.Close(Fd);
  exception
    when Name_Error =>
      Put("Can not create " & File_Name ); New_Line;
  end;
end Main;
```

Switching the default input and output streams

Procedure	Sets the default
Set_Input (File:**in** File_Type)	Input file descriptor.
Set_OutPut(File:**in** File_Type)	Output file descriptor.
Set_Error (File:**in** File_Type)	Error file descriptor.

Function	Returns the access
Standard_Input **return** File_Access;	Value of the input file descriptor.
Standard_Output **return** File_Access;	Value of the output file descriptor.
Standard_Error **return** File_Access;	Value of the error file descriptor.

```
with Ada.Text_Io;
use  Ada.Text_Io;
procedure Main is
  Fd      : Ada.Text_Io.File_Type;    --File descriptor
  P_St_Fd : Ada.Text_Io.File_Access;  --Access value of Standard
  Ch      : Character;                --Current character
begin
  P_St_Fd := Standard_Input;        --Acess value of standard fd
  Open( File=>Fd, Mode=>In_File, Name=>"file.txt" );
  Set_Input( Fd );
  while not End_Of_File loop        --For each Line
    while not End_Of_Line loop      -- For each character
      Get(Ch); Put(Ch);             --   Read / Write character
    end loop;
    Skip_Line; New_Line;            --  Next line / new line
  end loop;
  Close(Fd);                        --Close file
  Set_Input( P_St_Fd.all );
  while not End_Of_File loop        --For each Line
    while not End_Of_Line loop      --For each character
      Get(Ch); Put(Ch);             --Read / Write character
    end loop;
    Skip_Line; New_Line;            --Next line / new line
  end loop;
end Main;
```

A persistent indexed collection

Method	Responsibility
`Initialize`	Initialize the object. When the object is initialized with an identity, the state of the named persistent object is restored into the object.
`Finalize`	If the object has an identity, save the state of the object under this name.
`Add`	Add a new data item to the object.
`Extract`	Extract the data associated with an index.
`Update`	Update the data associated with an index.
`Set_Name`	Set the identity of the object.
`Get_Name`	Return the identity of the object.

```
package Pack_Types is
 subtype  Country is String(1 .. 12); --Country
 subtype  Idc     is String(1 .. 6); --International Dialling Code
end Pack_Types;
```

```
with Class_Pic, Pack_Types;
use  Pack_Types;
  pragma Elaborate_All( Class_Pic );
  package Class_Tel_List is new Class_Pic( Country, Idc, ">" );
```

```
with Ada.Text_Io, Pack_Types, Class_Tel_List;
use  Ada.Text_Io, Pack_Types, Class_Tel_List;
procedure Main is
  Tel_List : Pic;
  Action   : Character;
  Name     : Country;
  Tel      : Idc;
begin
  Initialize( Tel_List, "tel_list.per" );
  while not End_Of_File loop
    begin
      Get( Action );                          --Action to perform
      case Action is
        when '+' =>                           --Add
          Get( Name ); Get( Tel );
          Add( Tel_List, Name, Tel );
        when '=' =>                           --Extract
          Get( Name );
          Extract( Tel_List, Name, Tel );
          Put( "IDC for " ); Put( Name );
          Put( " is "); Put( Tel ); New_Line;
        when '*' =>                           --Update
          Get( Name ); Get( Tel );
          Update( Tel_List, Name, Tel );
        when others =>                        --Invalid action
          null;
      end case;
    exception
      when Not_There =>                       --Not there
        Put("Name not in directory"); New_Line;
      when Mainists =>                         --Exists
        Put("Name already in directory"); New_Line;
    end;
    Skip_Line;
  end loop;
end Main;
```

```
with Ada.Text_Io, Class_Tel_List;
use  Ada.Text_Io, Class_Tel_List;
procedure Main is
  Tel_List : Pic;
begin
  Put("Creating Telephone list"); New_Line;
  Set_Name( Tel_List, "tel_list.per" );
  Add( Tel_List, "Canada      ", "+1   " );
  Add( Tel_List, "USA         ", "+1   " );
  Add( Tel_List, "Netherlands ", "+31  " );
  Add( Tel_List, "Belgium     ", "+32  " );
  Add( Tel_List, "France      ", "+33  " );
  Add( Tel_List, "Gibraltar   ", "+350 " );
  Add( Tel_List, "Ireland     ", "+353 " );
  Add( Tel_List, "Switzerland ", "+41  " );
  Add( Tel_List, "UK          ", "+44  " );
  Add( Tel_List, "Denmark     ", "+45  " );
  Add( Tel_List, "Norway      ", "+47  " );
  Add( Tel_List, "Germany     ", "+49  " );
  Add( Tel_List, "Australia   ", "+61  " );
  Add( Tel_List, "Japan       ", "+81  " );
end Main;
```

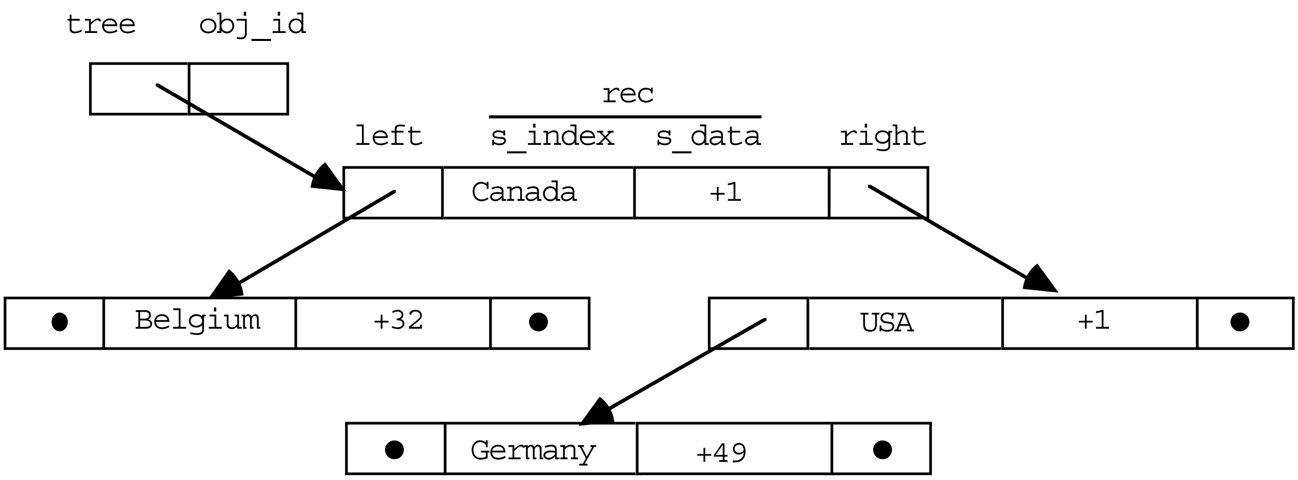
```
with Ada.Strings.Unbounded, Ada.Finalization;
use  Ada.Strings.Unbounded, Ada.Finalization;
generic
  type Index is private;                    --Index for record
  type Data  is private;                    --Data for record
  with function  ">"( F:in Index; S:in Index ) return Boolean;
package Class_Pic is
  Not_There, Mainists, Per_Error : exception; --Raised Exceptions
  type Pic  is new Limited_Controlled with private;
  procedure Initialize( The:in out Pic );
  procedure Initialize( The:in out Pic; Id:in String );
  procedure Finalize( The:in out Pic );
  procedure Discard( The:in out Pic );
  procedure Set_Name( The:in out Pic; Id:in String );
  function  Get_Name( The:in Pic ) return String;

  procedure Add( The:in out Pic; I:in Index; D:in Data );
  procedure Extract( The:in out Pic; I:in Index; D:in out Data );
  procedure Update( The:in out Pic; I:in Index; D:in out Data );
private
  type Leaf;                       --Index + Data
  type Subtree is access Leaf;     --
  type Pic is new Limited_Controlled with record
    Tree    : Subtree := null;     -- Storage
    Obj_Id : Unbounded_String;      -- Name of object
  end record;

  function  Find( The:in Subtree; I:in Index) return Subtree;
  procedure Release_Storage( The:in out Subtree );

end Class_Pic;
```

```
with Unchecked_Deallocation, Sequential_Io;
package body Class_Pic is

  type Element is record      --
    S_Index: Index;           --The Index
    S_Data : Data;            --The Data
  end record;

  type Leaf is record         --
    Left   : Subtree;         --Possible left node
    Rec    : Element;         --Index + data
    Right  : Subtree;         --Possible right node;
  end record;
```

For example, after the following data is added to the data structure:

Country	**IDC**
Canada	+1
USA	+1
Belgium	+32
Germany	+49

the resultant tree would be as:



```
package Io is new Sequential_Io( Element );
```

```
procedure Initialize( The:in out Pic ) is
begin
  The.Tree := null;    --No storage
end Initialize;
```

```
procedure Initialize( The:in out Pic; Id:in String ) is
  Per : Io.File_Type;      --File descriptor
  Cur : Element;           --Persistent data record element
begin
  Set_Name( The, Id );                         --Name object
  Io.Open( Per, Io.In_File, Id );              --Open saved state
  while not Io.End_Of_File( Per ) loop         --Restore saved state
    Io.Read( Per, Cur );
    Add( The, Cur.S_Index, Cur.S_Data );
  end loop;
  Io.Close( Per );
exception                                      --Return real exception
  when others => raise Per_Error;             -- as sub code
end Initialize;
```

```
procedure Finalize( The:in out Pic ) is
  Per : Io.File_Type;    --File descriptor
  procedure Rec_Finalize( The:in Subtree ) is --Save state
  begin
    if The /= null then                      --Subtree save as
      Io.Write( Per, The.Rec );              -- Item
      Rec_Finalize( The.Left );              -- LHS
      Rec_Finalize( The.Right );             -- RHS
    end if;
  end Rec_Finalize;
begin
  if To_String(The.Obj_Id) /= "" then        --If save state
    Io.Create( Per, Io.Out_File,
      To_String( The.Obj_Id ) );
    Rec_Finalize( The.Tree );
    Io.Close( Per );
  end if;
  Release_Storage( The.Tree );
exception                                 --Return real exception
  when others => raise Per_Error;        -- as sub code
end Finalize;
```

```
procedure Discard egin
  Set_Name( The, "" );                    --No name
  Release_Storage( The.Tree );            --Release storage
end Discard;

procedure Set_Name( The:in out Pic; Id:in String ) is
begin
  The.Obj_Id := To_Unbounded_String(Id); --Set object name
end Set_Name;

function  Get_Name( The:in Pic ) return String is
begin
  return To_String( The.Obj_Id );        --Name of object
end Get_Name;
```

```
procedure Add( The:in out Pic; I:in Index; D:in Data ) is
  procedure Add_S(The:in out Subtree; I:in Index; D:in Data) is
  begin
    if The = null then
      The := new Leaf'( null, Element'(I,D), null );
    else
      if I = The.Rec.S_Index then       --Index all ready exists
        raise Mainists;
      elsif I > The.Rec.S_Index then    --Try on RHS
        Add_S( The.Right, I, D );
      else                              --LHS
        Add_S( The.Left, I, D );
      end if;
    end if;
  end Add_S;
begin
  Add_S( The.Tree, I, D );
end Add;
```

```
procedure Extract(The:in out Pic; I:in Index; D:in out Data) is
  Node_Is : Subtree;
begin
  Node_Is := Find( The.Tree, I );       --Find node with iey
  D := Node_Is.Rec.S_Data;              --return data
end Extract;

procedure Update(The:in out Pic; I:in Index; D:in out Data) is
  Node_Is : Subtree;
begin
  Node_Is := Find( The.Tree, I );       --Find node with iey
  Node_Is.Rec.S_Data := D;              --Update data
end Update;
```

```
function Find( The:in Subtree; I:in Index) return Subtree is
begin
  if The = null then raise Not_There; end if;
  if I = The.Rec.S_Index then
    return The;                              --Found
  else
    if I > The.Rec.S_Index
      then return Find( The.Right, I );   --Try RHS
    else return Find( The.Left,  I );   --Try LHS
    end if;
  end if;
end Find;
```

```
  procedure Dispose is
    new Unchecked_Deallocation( Leaf, Subtree );

  procedure Release_Storage( The:in out Subtree ) is
  begin
    if The /= null then                  --Not empty
      Release_Storage( The.Left );  --Free LHS
      Release_Storage( The.Right ); --Free RHS
      Dispose( The );                    --Dispose of item
    end if;
    The := null;                         --Subtree root NULL
  end Release_Storage;

end Class_Pic;
```

Concurrency

```
package Pack_Factorial is
  task type Task_Factorial is                       --Specification
    entry Start( F:in Positive );                   --Rendezvous
    entry Finish( Result:out Positive );            --Rendezvous
  end Task_Factorial;
end Pack_Factorial;
```

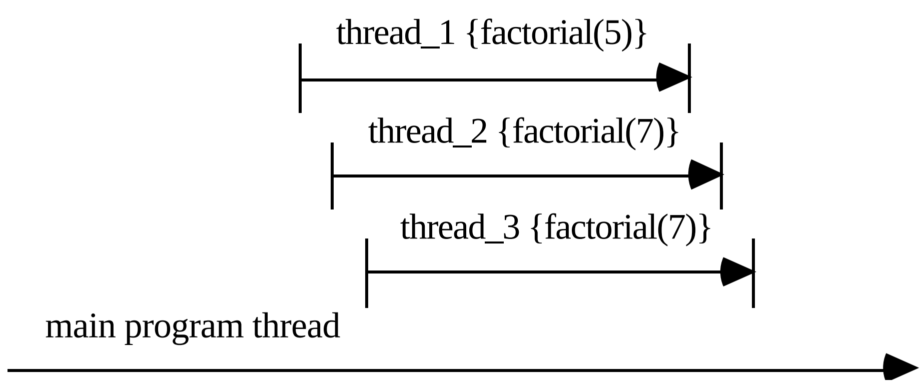
```
package Pack_Is_A_Prime is
  task type Task_Is_Prime is                        --Specification
    entry Start( P:in Positive );                   --Rendezvous
    entry Finish( Result:out Boolean );             --Rendezvous
  end Task_Is_Prime;
end Pack_Is_A_Prime;
```

```
  Thread_1 : Task_Factorial;
```

```
 Thread_1.Start(5);              --Start factorial calculation
```

```
with Ada.Text_Io, Ada.Integer_Text_Io,
     Pack_Factorial, Pack_Is_A_Prime;
use  Ada.Text_Io, Ada.Integer_Text_Io,
     Pack_Factorial, Pack_Is_A_Prime;
procedure Main is
  Thread_1 : Task_Factorial;
  Thread_2 : Task_Factorial;
  Thread_3 : Task_Is_Prime;
  Factorial: Positive;
  Prime    : Boolean;
begin
  Thread_1.Start(5);           --Start factorial calculation
  Thread_2.Start(7);           --Start factorial calculation
  Thread_3.Start(97);          --Start is_prime  calculation

  Put("Factorial  5 is ");
  Thread_1.Finish( Factorial ); --Obtain result
  Put( Factorial ); New_Line;

  Put("Factorial  8 is ");
  Thread_2.Finish( Factorial ); --Obtain result
  Put( Factorial ); New_Line;

  Put("97 is a prime is ");
  Thread_3.Finish( Prime );     --Obtain result
  if Prime then Put("True"); else  Put("False"); end if;
  New_Line;
```

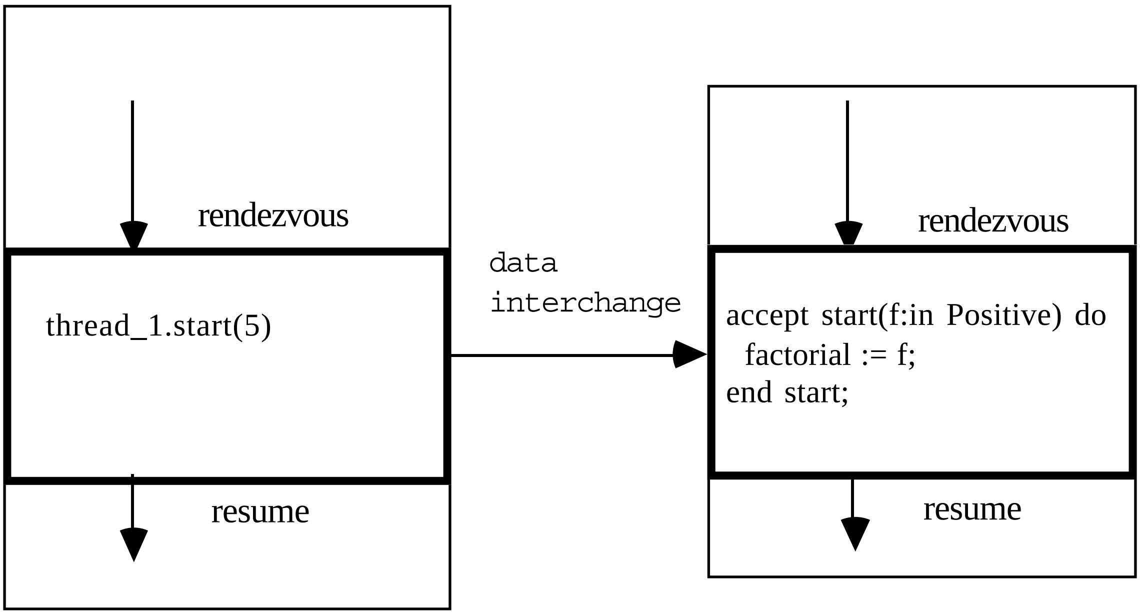
```
Factorial  5 is        120
Factorial  7 is       5040
97 is a prime is True
```



Task rendezvous

Main program (client)	**Body of task** Thread_1 (server)

`Thread_1.Start(5);`	`accept Start(F:in Positive) do`
	`Factorial := F;`
	`end Start;`



Variation	Client	Server task
No information passed	`Thread_1.Start;`	`accept Start;`

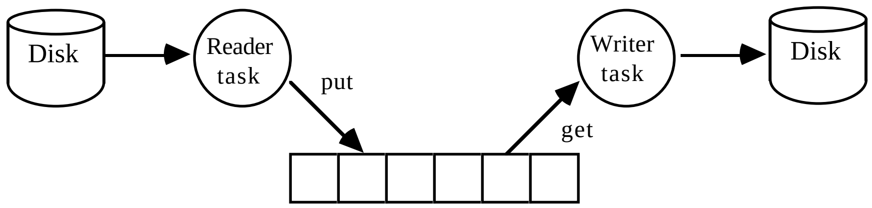
No information passed but Thread_1 executes statements during the rendezvous.	`Thread_1.Start;`	`accept Start do`
		`Statements;`
		`end Start;`

The tasks implementation

```
package body Pack_Factorial is
  task body Task_Factorial is                        --Implementation
    Factorial : Positive;
    Answer    : Positive := 1;
  begin
    accept Start( F:in Positive ) do                 --Factorial
      Factorial := F;
    end Start;
    for I in 2 .. Factorial loop                     --Calculate
      Answer := Answer * I;
    end loop;
    accept Finish( Result:out Positive ) do --Return answer
      Result := Answer;
    end Finish;
  end Task_Factorial;
end Pack_Factorial;
```

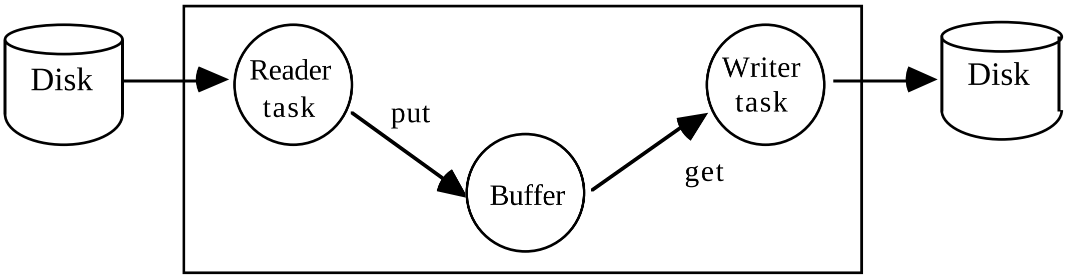
```
package body Pack_Is_A_Prime is
  task body Task_Is_Prime is                         --Implementation
    Prime : Positive;
    Answer: Boolean := True;
  begin
    accept Start( P:in Positive ) do                 --Factorial
      Prime := P;
    end Start;
    for I in 2 .. Prime-1 loop                       --Calculate
      if Prime rem I = 0 then
        Answer := False; exit;
      end if;
    end loop;
    accept Finish( Result:out Boolean ) do  --Return answer
      Result := Answer;
    end Finish;
  end Task_Is_Prime;
end Pack_Is_A_Prime;
```

Mutual exclusion and critical sections



Protected type

Unit	Commentary	Access (1)
`procedure`	A procedure will only execute when no other units are being executed. If necessary the procedure will wait until the currently executing unit(s) have finished.	Read and write
`function`	A function may execute simultaneously with other executing functions. However, a function cannot execute if a procedure is currently executing.	Read only.
`entry`	Like a procedure but may also have a barrier condition associated with the entry. If the barrier condition is false the entry is queued until the barrier becomes true.	Read and write



Name	Object is	Responsibilities
`Task_Reader`	`Task`	Read data from the file and then pass the data to the buffer. Note: The task will block if the buffer is full.
`Task_Writer`	`Task`	Take data from the buffer task and write the data to the file. Note: The task will block if there is no data in the buffer.
`PT_Buffer`	`Protected type`	Serialize the storing and retrieving of data to and from a buffer.

```
with Ada.Text_Io;
use  Ada.Text_Io;
package Pack_Types is
  type P_File_Type is access all Ada.Text_Io.File_Type;
  Eot   : constant Character := Character'Val(0);
  Cr    : constant Character := Character'Val(15);
  Queue_Size     : constant := 3;

  type     Queue_No     is new Integer   range 0 .. Queue_Size;
  type     Queue_Index is mod Queue_Size;
  subtype Queue_Range is Queue_Index;
  type     Queue_Array is array ( Queue_Range ) of Character;
end Pack_Types;
```

```
with Pack_Types;
use  Pack_Types;
package Pack_Threads is
  protected type PT_Buffer  is           --Task type specification
    entry Put( Ch:in Character; No_More:in Boolean );
    entry Get( Ch:in out Character; No_More:out Boolean);
  private
    Elements     : Queue_Array;              --Array of elements
    Head         : Queue_Index := 0;         --Index
    Tail         : Queue_Index := 0;         --Index
    No_In_Queue : Queue_No     := 0;         --Number in queue
    Fin          : Boolean     := False;     --Finish;
  end PT_Buffer ;

  type P_PT_Buffer  is access all PT_Buffer ;

  task type Task_Read( P_Buffer:P_PT_Buffer ;
                       Fd_In:P_File_Type) is
    entry Finish;
  end Task_Read;

  task type Task_Write( P_Buffer:P_PT_Buffer ;
                        Fd_Out:P_File_Type) is
    entry Finish;
  end Task_Write;
end Pack_Threads;
```

```
with Ada.Text_Io, Pack_Threads, Pack_Types;
use  Ada.Text_Io, Pack_Threads, Pack_Types;
procedure Do_Copy(From:in String; To:in String) is
  type State is ( Open_File, Create_File );
  Fd_In    : P_File_Type := new Ada.Text_Io.File_Type;
  Fd_Out   : P_File_Type := new Ada.Text_Io.File_Type;
  Mode     : State := Open_File;
begin
  Open(  File=>Fd_In.all,  Mode=>In_File,  Name=>From);
  Mode := Create_File;
  Create(File=>Fd_Out.all, Mode=>Out_File, Name=>To);
  declare
    Buffers : P_PT_Buffer  := new PT_Buffer ;
    Reader  : Task_Read( Buffers, Fd_In );
    Writer  : Task_Write( Buffers, Fd_Out );
  begin
    Reader.Finish;  Close( Fd_In.all );   --Finish reader task
    Writer.Finish;  Close( Fd_Out.all );  --Finish writer task
  end;
exception
  when Name_Error =>
    case Mode is
      when Open_File =>
        Put("Problem opening file " & From ); New_Line;
      when Create_File =>
        Put("Problem creating file " & To ); New_Line;
    end case;
  when Tasking_Error =>
    Put("Task error in main program"); New_Line;
end Do_Copy;
```

```
with Ada.Text_Io, Ada.Command_Line, Do_Copy;
use  Ada.Text_Io, Ada.Command_Line;
procedure Copy is
begin
  if Argument_Count = 2 then
    Do_Copy ( Argument(1), Argument(2) );
  else
    Put("Usage: copy from to"); New_Line;
  end if;
end Copy;
```

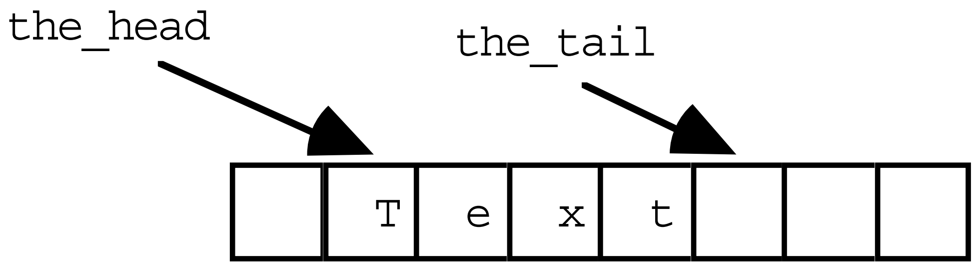
```
with Ada.Text_Io;
use  Ada.Text_Io;
package body Pack_Threads is

  task body Task_Read is                          --Task implementation
    Ch        : Character;
  begin
    while not End_Of_File( Fd_In.all ) loop
      while not End_Of_Line( Fd_In.all ) loop
        Get( Fd_In.all, Ch);                      --Get character
        P_Buffer.Put( Ch, False );                --Add to buffer
      end loop;
      Skip_Line( Fd_In.all );                     --Next line
      P_Buffer.Put( Cr, False );                  --New line
    end loop;
    P_Buffer.Put( Eot, True );                    --End of characters

    accept Finish;
  exception
    when Tasking_Error =>
      Put("Exception in Task read"); New_Line;
  end Task_Read;
```

```
  task body Task_Write is                         --Task implementation
    Last       : Boolean := False;                --No more data
    Ch         : Character;                       --Character read
  begin
    loop
      P_Buffer.Get( Ch, Last );                   --From buffer
      exit when Last;                             --No more characters
      if Ch = Cr then
        New_Line( Fd_Out.all );                   --New line
      else
        Put( Fd_Out.all, Ch );                    --Character
      end if;
    end loop;

    accept Finish;                                --Finished
  exception
    when Tasking_Error =>
      Put("Exception in Task write"); New_Line;
  end Task_Write;
```

Barrrier condition entry

```
entry Put( Ch:in Character; No_More:in Boolean )
   when No_In_Queue < Queue_Size  is
```

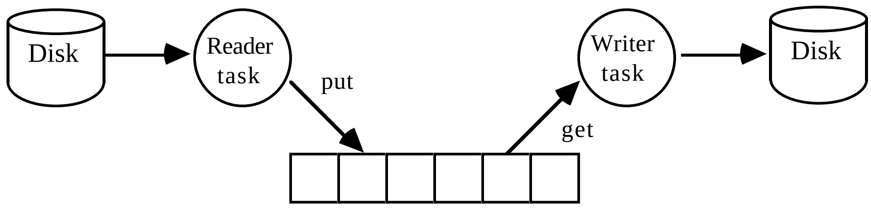
```
protected body PT_Buffer  is
```



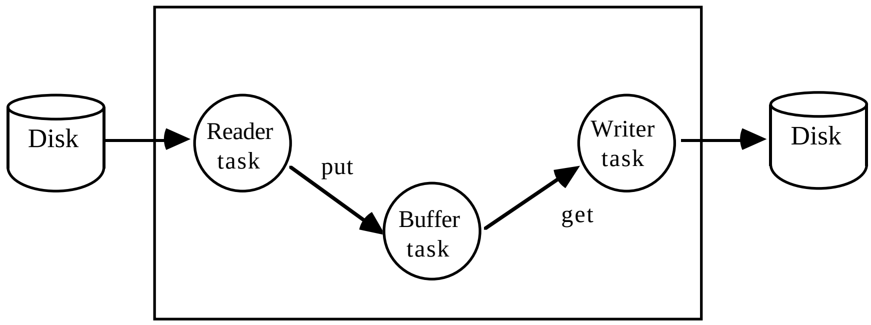

```
entry Put( Ch:in Character; No_More:in Boolean )
   when No_In_Queue < Queue_Size  is
begin
  if No_More then                       --Last
    Fin := True;                        --Set flag
  else
    Elements( Tail ) := Ch;             --Add to queue
    Tail := Tail+1;                     --Next position
    No_In_Queue := No_In_Queue + 1;     --
  end if;
end;
```

```
entry Get(Ch:in out Character; No_More:out Boolean)
   when No_In_Queue > 0 or else Fin is
begin
  if No_In_Queue > 0 then                --Item available
    Ch := Elements( Head );              --Get item
    Head := Head+1;                      --Next position
    No_In_Queue := No_In_Queue - 1;      --
    No_More := False;                    --Not end
  else
    No_More := True;                     --End of items
  end if;
end;

 end PT_Buffer;
end Pack_Threads;
```

Mutual exclusion and critical sections
Using a task for serialisation of requests




```
select                          -- Choice of accepts
  accept Put do
    ...
  end;
or
  accept Get do
    ...
  end;

end select;
```



Task	Responsibilities
`Task_reader`	Read data from the file and then pass the data to the buffer. Note: The task will block if the buffer is full.
`Task_writer`	Take data from the buffer task and write the data to the file. Note: The task will block if there is no data in the queue.
`Task_buffer`	Store and deliver data in a queue.


```
with Text_io; use  Text_io;
package Pack_types is
  type P_File_type is access Text_io.File_type;
end Pack_types;

package body Pack_types is
end Pack_types;
```

```
with Text_io, Pack_types; use Text_io, Pack_types;
package Class_threads is
  task type Task_buffer is          -- Task type specification
    entry put( ch:in Character);
    entry get( ch:in out Character; eof:out Boolean);
    entry finish;
  end;
  type P_Task_buffer is access Task_buffer;

  task type Task_read is            -- Task type specification
    entry start( b:in P_Task_buffer; fd:in P_File_type );
    entry finish;
  end;

  task type Task_write is           -- Task type specification
    entry start( b:in P_Task_buffer; fd:in P_File_type);
    entry finish;
  end;
end Class_threads;
```

```
with Text_io, Class_threads, Pack_types;
use  Text_io, Class_threads, Pack_types;
procedure execute_threads is
buffers : P_Task_buffer;
fd_in   : P_File_Type := new Text_io.File_type; -- File handle
fd_out  : P_File_type := new Text_io.File_type; -- File handle
reader  : Task_read;
writer  : Task_write;
begin
  open(   File=>fd_in.all,  Mode=>IN_FILE,  Name=>"text" );
  create( File=>fd_out.all, Mode=>OUT_FILE, Name=>"text_cpy" );

  buffers := new Task_buffer;            -- Start buffer task
  reader.start( buffers, fd_in );        -- Start reader task
  writer.start( buffers, fd_out );       -- Start writer task

  reader.finish;  close( fd_in.all );
  writer.finish;  close( fd_out.all );
  buffers.finish;
exception
  when Name_error =>
    put("Problem opening / creating files"); new_line;
  when Tasking_error =>
    put("Task error in main program"); new_line;
end execute_threads;
```

```
buffers := new Task_buffer;            -- Start buffer task
```

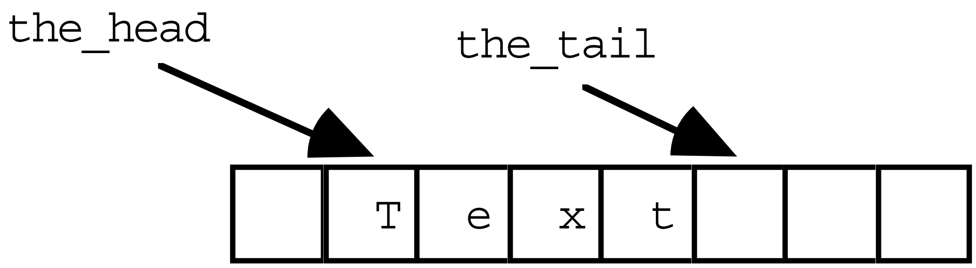
```
with Text_io; use  Text_io;
package body Class_threads is
  task body Task_read is                        -- Task implementation
  p_buffer: P_Task_buffer;                      -- Buffer task
  fd_in    : P_File_type;
  ch       : Character;
  begin
    accept start(b:in P_Task_buffer; fd:in P_File_type) do
      p_buffer := b;                            -- buffer task
      fd_in := fd;                              -- fd for read
    end;
    while not end_of_file( fd_in.all ) loop
      while not end_of_line( fd_in.all ) loop
        get( fd_in.all, ch);                    -- Get character
        p_buffer.put( ch );                     -- Add to buffer
      end loop;
      p_buffer.put( Ascii.cr );                 -- New line ch
      skip_line( fd_in.all );                   -- Next line
    end loop;
    p_buffer.finish;                            -- Copy end
    accept finish;
  exception
    when Tasking_error =>
      put("Exception in Task read"); new_line;
  end Task_read;
```

```
  task body Task_write is                       -- Task implementation
  fd_out    : P_File_type;                      -- File handle
  p_buffer  : P_Task_buffer;                    -- Buffer task
  eof       : Boolean;                          -- No more data
  ch        : Character;                        -- Character read
  begin
    accept start(b:in P_Task_buffer; fd:in P_File_type) do
      p_buffer := b;                            -- Buffer task
      fd_out   := fd;                           -- fd for write
    end;
    loop
      p_buffer.get( ch, eof );                  -- From buffer
      exit when eof;
      if ch = Ascii.cr then
        new_line( fd_out.all );                 -- New line
      else
        put( fd_out.all, ch );                  -- Character
      end if;
    end loop;
    accept finish;                              -- Finished
  exception
    when Tasking_error =>
      put("Exception in Task write"); new_line;
  end Task_write;
```

Guarded accepts

```
when the_no_of < QUEUE_SIZE  =>
  accept put( ch: in Character ) do ... end;
```

```
  task body Task_buffer is
  QUEUE_SIZE     : constant := 3;

  type    Queue_no     is new Integer   range 0 .. QUEUE_SIZE;
  type    Queue_index is new Queue_no;
  subtype Queue_range is Queue_index   range 0 .. QUEUE_SIZE-1;
  type    Queue_array is array ( Queue_range ) of Character;

  the_elements: Queue_array;              -- Array of elements
  the_head    : Queue_index := 0;         -- Index
  the_tail    : Queue_index := 0;         -- Index
  the_no_of   : Queue_no    := 0;         -- Number in queue
  the_end     : Boolean     := FALSE;     -- Finish;
```




```
  begin
    loop
      select                              -- Store in buffer
        when the_no_of < QUEUE_SIZE  =>
          accept put( ch:in Character ) do
            the_elements( the_tail ) := ch;
            the_tail := (the_tail+1) rem QUEUE_SIZE;
            the_no_of := the_no_of + 1;
          end;
```

```
      or                                      -- Get from buffer
        when the_no_of > 0 or else the_end =>
          accept get(ch:in out Character; eof:out Boolean) do
            if the_no_of > 0 then
              ch := the_elements( the_head );
              the_head := (the_head+1) rem QUEUE_SIZE;
              the_no_of := the_no_of - 1;
              eof := FALSE;
            else
              eof := TRUE;
            end if;
          end;
          exit when the_end and then the_no_of = 0;
```

```
     or                                       -- Inform no more data
          accept finish do
            the_end := TRUE;
          end;
      end select;
    end loop;
```

```
    accept finish;                            -- Finished
  exception
    when Tasking_error =>
      put("Exception in Buffers"); new_line;
  end Task_buffer;
end Class_threads;
```

Delay

```
delay 2.5;
```

```
delay until Time_Of(2010,1,1,0.0);     -- Until 1 Jan 2010
```

Choice of accepts

```
select                          -- Choice of accepts
  accept option1 do
    ...
  end;
or
  accept option2 do
    ...
  end;

end select;
```

```
loop
  select                        -- Serialization of code
    accept option1 do ... end;
  or
    accept option2 do ... end;
  end select;
end loop;
```

Accept alternative

```
select
  accept  ...


else
  statements;          -- Only executed if no call on an
                       --  accept immediately satisfied
end select;
```

Accept time-out

```
select
  accept  ...


or
  delay TIME;             -- Time out delay in seconds
  statements;             -- Only executed if no call on an
                          --  accept within TIME seconds
end select;
```

```
package Pack_Watchdog is
  task type Task_Watchdog is                    --Specification
    entry Poll;                                 --Rendezvous
    entry Finish;                               --Rendezvous
  end Task_Watchdog;
end Pack_Watchdog;
```

```
with Ada.Text_Io;
use  Ada.Text_Io;
package body Pack_Watchdog is
  task body Task_Watchdog is                    --Implementation
  begin
    loop
      select
        accept Poll;                            --Successful poll
      or
        accept Finish;                          --Terminate
        exit;
      or
        delay 0.2;                              --Time out
        Put("WARNING Watchdog failure");
        New_Line;
        exit;
      end select;
      delay 0.0001;                             --Cause task switch
    end loop;
  end Task_Watchdog;
end Pack_Watchdog;
```

System programming

Representation clause

```
type Country is  (USA, France, UK, Australia);
```

```
type Country is  (USA, France, UK, Australia);
for  Country use (USA=> 1, France=> 33, UK=> 44, Australia=> 61);
```

Expression	Delivers
Country'Succ(USA)	France
Country'Pred(Australia)	UK
Country'Pos(France)	1
Country'Val(2)	UK

```
type Country is  (USA, France, UK, Australia);
for  Colour'Size use Integer'Size;
for  Country use (USA=> 1, France=> 33, UK=> 44, Australia=> 61);
```

```
with System, System.Storage_Elements;
use  System, System.Storage_Elements;
procedure Main is
  type Country is  (USA, France, UK, Australia);
  for  Colour'Size use Integer'Size;
  for  Country use (USA=>1, France=>33, UK=>44, Australia=>61);

  function Idc is new Unchecked_Conversion( Country, Integer );
begin
  Put( "International dialling code for France is ");
  Put( Idc(France) );
  New_Line;
end Main;
```

```
International dialling code for France is 33
```

```
function Canada return Country renames USA;
```

Binding of a variable to a specific address

Location (hexadecimal)	Contents
046E - 046F	The time of day in hours.
046C - 046D	The ticks past the current hour. Each tick is 5/91 seconds.

```
Time_High_Address : constant Address := To_Address( 16#046C# );

type Time      is range 0 .. 65365;          --Unsigned
for  Time'Size use 16;                       -- in 2 bytes

Time_High: Time;
  for Time_High'Address use Time_Low_Address;
```

```
with System, System.Storage_Elements,
     Ada.Text_Io, Ada.Integer_Text_Io;
use  System, System.Storage_Elements,
     Ada.Text_Io, Ada.Integer_Text_Io;
procedure Main is
  Time_High_Address : constant Address := To_Address( 16#046C# );
  Time_Low_Address  : constant Address := To_Address( 16#046E# );
  type Seconds_T is range 0 .. 1_000_000_000;  --up to 65k * 5
  type Time      is range 0 .. 65365;          --Unsigned
  for  Time'Size use 16;                       -- in 2 bytes
  Time_Low : Time;
    for Time_Low'Address  use Time_High_Address;
  Time_High: Time;
    for Time_High'Address use Time_Low_Address;
  Seconds   : Seconds_T;
begin
  Put("Time is ");
  Put( Time'Image(Time_High) ); Put(" :");      --Hour
  Seconds := (Seconds_T(Time_Low) * 5) / 91;
  Put(Seconds_T'Image(Seconds/60)); Put(" :"); --Mins
  Put(Seconds_T'Image(Seconds rem 60));        --Seconds
  New_Line;
end Main;
```

```
Time is  17 : 54 : 57
```

Access to individual bits

Most significant		Bit position				Least significant	
7	6	5	4	3	2	1	0
Insert	Caps lock	Number lock	Scroll lock				

```
with System, System.Storage_Elements,
     Ada.Text_Io, Ada.Integer_Text_Io;
use  System, System.Storage_Elements,
     Ada.Text_Io, Ada.Integer_Text_Io;
procedure Main is
  Keyboard_Address : constant Address := To_Address( 16#417# );
  type Status      is  ( Not_Active, Active );
  for  Status      use ( Not_Active => 0, Active => 1 );
  for  Status'Size use 1;
```

```
  type Keyboard_Status is
  record
    Scroll_Lock : Status;          --Scroll lock status
    Num_Lock    : Status;          --Num lock status
    Caps_Lock   : Status;          --Caps lock status
    Insert      : Status;          --Insert status
  end record;

  for Keyboard_Status use
    record
    Scroll_Lock at 0 range 4..4; --Storage unit 0 Bit 4
    Num_Lock    at 0 range 5..5; --Storage unit 0 Bit 5
    Caps_Lock   at 0 range 6..6; --Storage unit 0 Bit 6
    Insert      at 0 range 7..7; --Storage unit 0 Bit 7
  end record;
  Keyboardstatus_Byte : Keyboard_Status;
  for Keyboardstatus_Byte'Address use Keyboard_Address;
```

```ada
begin
  if Keyboardstatus_Byte.Insert = Active then
    Put("Insert mode set"); New_Line;
  else
    Put("Insert mode not set"); New_Line;
  end if;
  if Keyboardstatus_Byte.Caps_Lock = Active then
    Put("Caps   lock set"); New_Line;
  else
    Put("Caps   lock not set"); New_Line;
  end if;
  if Keyboardstatus_Byte.Num_Lock = Active then
    Put("Number lock set"); New_Line;
  else
    Put("Number lock not set"); New_Line;
  end if;
end Main;
```

```
Insert mode not set
Caps   lock not set
Number lock not set
```

Mixed language programming

```
with Interfaces.C;
use  Interfaces.C;
function Triple( N:in Integer ) return Integer is
  function C_Triple(N:in Int) return Int;
  pragma Import (C, C_Triple, "c_triple");
begin
  return Integer( C_Triple( Int(N) ) );
end Triple;

with Ada.Text_Io, Ada.Integer_Text_Io, Triple;
use  Ada.Text_Io, Ada.Integer_Text_Io;
procedure Main is
begin
  Put("3 Tripled is "); Put( Triple(3) ); New_Line;
end Main;
```

```
int c_triple( int n )           /* function to double argument */
{
  return n + n + n;
}
```

```
3 Tripled is           9
```

```
with Interfaces.C, Ada.Text_Io;
use  Interfaces.C, Ada.Text_Io;
procedure Main is

  function Strlen( Str:in String ) return Integer is
    function C_Strlen( C_Str:in char_array ) return Int;
    pragma Import (C, C_Strlen, "strlen");
  begin
    return Integer( C_Strlen( To_C( Str, Append_Nul => True ) ) );
  end Strlen;

  place : constant String := "Brighton";
begin
  Put("The length of the string [" & place & "] is " &
       Integer'Image( strlen( place ) ) & " characters long");
  New_Line;
end Main;
```

```
The length of the string [Brighton] is  8 characters long
```

```
package raw_io is
  procedure get_immediate( ch:out Character );
  procedure put( ch:in Character );
  procedure put( str:in String );
end raw_io;
```

```
with Interfaces.C;
use  Interfaces.C;
package body raw_io is

procedure get_immediate( ch:out Character ) is
   function c_get_char return Char;
   pragma import (C, c_get_char, "c_get_char");
begin
   ch := to_ada( c_get_char );
end get_immediate;

procedure put( ch:in Character ) is
   procedure c_put_char( ch:in Char );
   pragma import (C, c_put_char, "c_put_char");
begin
   c_put_char( to_c( ch ) );
end put;

procedure put( str:in String ) is
   procedure c_put_str( str:in Char_array );
   pragma import (C, c_put_str, "c_put_str");
begin
   c_put_str( to_c( str, append_nul=>TRUE ) );
end put;
```

```
char c_get_char()
{
  int c = getchar();
  return (char) (c & 0xFF);            /* Ordinary character */
}

void c_put_char( char ch )
{
  fputc(ch, stdout); fflush( stdout );  /* Output ch */
}

void c_put_str( char *str )
{
  while (*str) fputc(*str++, stdout);  /* Output String */
  fflush( stdout );                    /* Flush buffer */
}
```

Ada 95 : a summary

Simple object declarations

```
ch : Character;     -- An 8 bit character
i  : Integer;       -- A whole number
f  : Float;         -- A floating point number
```

Array declaration

```
  Computers_In_Room : array ( 1 .. 10 ) of Natural;
```

Type and subtype declarations

```
  type    Money  is delta 0.01 digits 8;           --
  subtype Pmoney is Money range 0.0 .. Money'Last; --+ve Money
```

Enumeration declaration

```
  type    Colour      is (Red, Green, Blue);
```

Simple statements

```
  Sum := 2 + 3;
  Deposit( Mine, 100.00 );
```

Block

```
  declare
    Ch : Character;
  begin
    Ch := 'A'; Put( Ch );
  end;
```

Class declaration and implementation

```
package Class_Account is
  type Account is tagged private;

  type    Money  is delta 0.01 digits 8;          --
  subtype Pmoney is Money range 0.0 .. Money'Last; --+ve Money

  procedure Deposit( The:in out Account; Amount:in Pmoney );
  procedure Withdraw( The:in out Account;
                      Amount:in Pmoney; Get:out Pmoney );
  function  Balance( The:in Account ) return Money;
private
  type Account is tagged record          --Instance variables
    Balance_Of        : Money := 0.00;   --Amount on deposit
    Min_Balance       : Money := 0.00;   --Minimum Balance
  end record;
end Class_Account;
```

```
package body Class_Account is

  procedure Deposit( The:in out Account; Amount:in Pmoney ) is
  begin
    The.Balance_Of := The.Balance_Of + Amount;
  end Deposit;

  -- Procedures withdraw and balance

end Class_Account;
```

Inheritance

```
with Class_account use Class_account;
package Class_interest_account is
  type Interest_account is tagged private;
  procedure calc_interest( the:in out Account );
  procedure add_interest( the:in out Account );
private
  type Interest_account is new Account with record
     accumulated_interest : Float := 0.00;
  end record;
end Class_account;
```

```
package body Class_interest_account is
  procedure add_interest( the:in out Account ) is
  begin
    deposit( Account(the), the.accumulated_interest );
    the.accumulated_interest := 0;
  end add_interest;

  -- Procedure calc_interest

end Class_account;
```

Selection statements

```
if temp < 15 then put("Cool"); end if;

if temp < 15 then put("Cool"); else put("Warm"); end if;

case number is
  when 2+3    => put("Is 5");
  when 7      => put("Is 7");
  when others => put("Not 5 or 7");
end case;
```

Looping statements

```
while raining loop
  work;
end loop;

loop
  play;
  exit when sunny;
end loop;

for i in 1 .. 10 loop
  put(i); new_line;
end loop;
```

Arithmetic operators

```
res := a + b;   -- plus
res := a - b;   -- minus
res := a * b;   -- multiplication
res := a / b;   -- division
res := a mod b; -- modulus
res := a rem b; -- remainder
```

Conditional expressions

```
if a = b  then -- Equal to
if a > b  then -- Greater
if a < b  then -- Less than
if a /= b then -- Not equal
if a >= b then -- Greater
               --  or equal
if a <= b then -- Less
               --  or equal
```

```
if wet and jan then   -- and
if dry or  feb then   -- or

if wet and then jan then --
if dry or else  feb then --
```

```
if temp > 15 and dry then play; end if;
```

Note: When using **`and then`** *or* **`or else`** *the conditional expression will only be evaluated as far as necessary to produce the result of the condition. Thus in the `if` statement:*

```
if fun_one or else fun_two then
```

`fun_two` *will not be called if* `fun_one` *delivered true.*

Exits from loops

```
loop play; exit when sunny; end loop;
```

Program delay

delay n.m seconds	delay until a_time;

`delay n.m;`	`declare`
	`use Ada.Calendar;`
	`begin`
	`delay until`
	`time_of(2000,1,1,0.0);`
	`-- Until 21st century`
	`end;`

Task

```
task type Task_factorial is                    -- Specification
  entry start( f:in Positive );                -- Rendezvous
  entry finish( result:out Positive );         -- Rendezvous
end Task_factorial;
```

```
task body Task_factorial is
begin
  accept start( f:in Positive )         do   end start;
  accept finish( result:out Positive )  do   end finish;
end Task_factorial;
```

Communication with a task

```
procedure main is
  thread    : Task_factorial;
  factorial: Positive;
begin
  thread_1.start(5);                -- Start factorial calculation
  thread_1.finish( factorial );     -- Obtain result
end execute_threads;
```

Rendezvous

select statement	select with else	select with delay

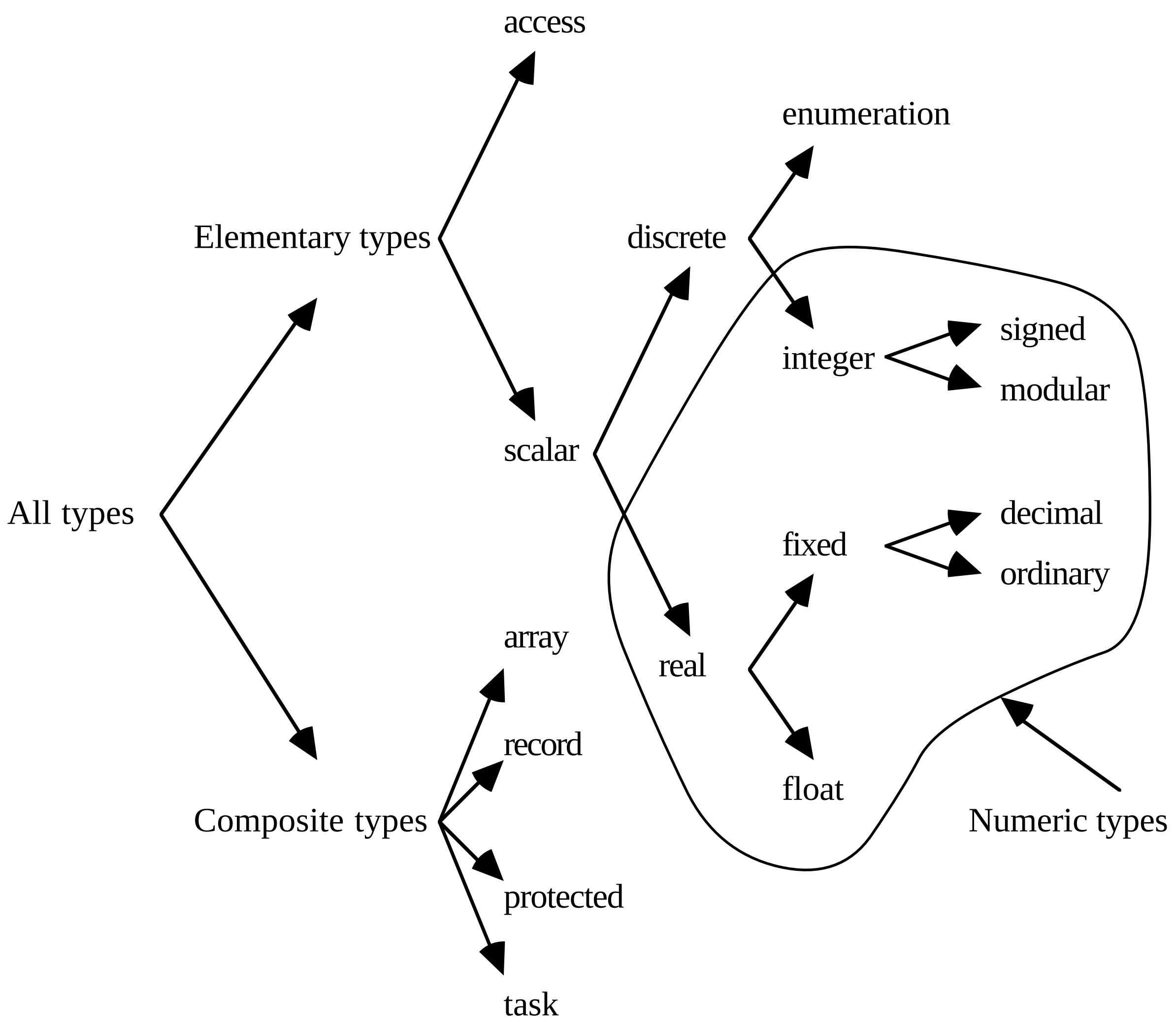
select
  accept option1 do
    ...
  end;
or
  accept option2 do
    ...
  end;
 	
end select;
	select
	  accept ...
	
	else
	  statements;
	
	end select;
		select
		  accept ...
		
		or
		  delay n.m;
		  statements;
		
		end select;

Protected type

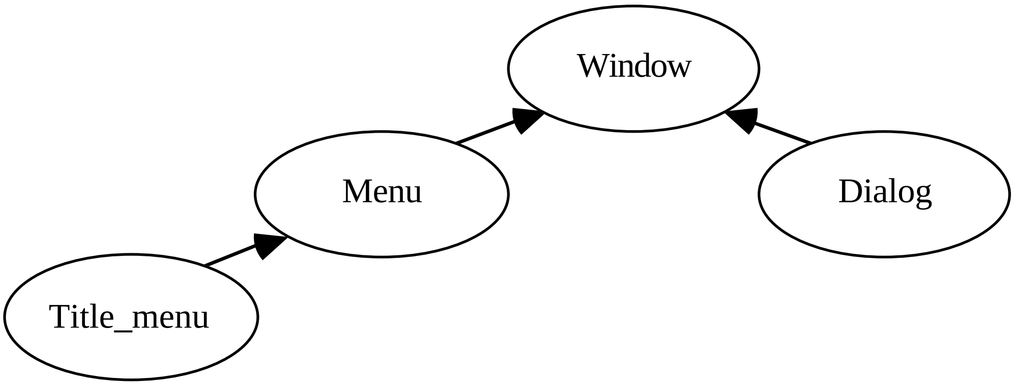
```
protected type PT_ex is
  entry put(i:in T);
  entry get(i:out T);
private
  -- variables which cannot be simultaneous accessed
end PT_ex;
```

```
  protected body PT_ex is
  begin
    entry put(i:in T)  is begin          end put;
    entry get(i:out T) is begin          end put;
  end PT_ex;
```

Type hierarchy



Text Windows



function / procedure	Note
`window_prolog;`	Set up the environment for the TUI. This must be called outside the block in which windows are elaborated.
`window_start;`	After initializing any program-generated windows, start the application by allowing a user to interact with the program.
`window_epilog;`	Close down the window system. This must be called outside the block in which the windows are elaborated.

```
procedure main is
begin
  window_prolog;                -- Set-up window system
  declare
                                -- Declaration of windows used in
                                --  the program
  begin
                                -- Initialization of windows
                                --  used in program
    window_start;               -- Start the user interaction
  end;
  window_epilog;                -- Close window system
end main;
```

```
win : Window;
```

Notes	Function / procedure
1	`procedure framework( the:in out Window; abs_x_crd, abs_y_crd: Positive; max_x_crd, max_y_crd: Positive; cb:in P cbf := null );`
2	`procedure put( w:in out Window; mes:in String );`
2	`procedure put( w:in out Window; ch:in Character );`
2	`procedure put( w:in out Window; n:in Integer );`
3	`procedure position( w:in out Window; x,y:in Positive );`
4	`procedure clear( w:in out Window );`
5	`procedure new_line( w:in out Window );`
6	`procedure make_window( w:in out Window; mo:in mode );`

Notes:

1 Sets the absolute position and size of the window on the screen.
The top left hand corner position is at:
`(abs_x_crd, abs_y_crd)`
The bottom left corner position is at:
`(abs_x_crd+max_x_crd-1, abs_y_crd+max_y_crd-1)`

2 Displays information in a window. These functions are modelled after the procedures in `Ada.Text_io`.

3 Sets the current output position in the window.

4 Clears the window to all spaces.

5 Writes a newline to the window. This will cause the information in the window to scroll up if the current position is at the last line of the window.

6 Makes the displayed window visible or invisible.

Dialog API calls

```
diag : Dialog;
```

Note	Function / procedure
1	`procedure framework ( the:in out Dialog; abs_x, abs_y:in Positive; max_x: in Positive; name:in String; cb:in P cbf );`

Note:

1 *Sets the absolute position of the window on the screen. The size of the window is set with `max_x`. The call-back function `cb` will be called after the user has constructed a message in the dialog box . This is initiated by the user entering the Enter character (return key). When the Enter character is received the menu window calls the call-back function with a string parameter containing the user's entered text. The signature of the call-back function is:*

```
function cb(mes:in String) return String
```

where `mes` is the message typed by the user.

User interaction with the TUI

Switch character	Description
TAB	Swaps the focus for user input to another window on the VDU screen. The active window is indicated by a # in the top left hand corner.
ESC	Activates the menu system. The menu system is described in detail in section 21.4.
^E	Terminates the TUI session. All windows will be closed and the user returned to the environment which initiated the program.

Classes used

API for a	Contained in the package	Notes
Window	Class_window	-
Dialog	Class_dialog	Plus the API inherited from a Window.
Menu	Class_menu	Plus the API inherited from a Window.
Menu_title	Class_menu_title	Plus the API inherited from a Menu
TUI set up	Class_input_manager	Controls the input sent to the TUI.

An example program using the TUI

```
with Class_window;
use  Class_window;
package Pack_globals is
  p_result : P_Window;
end Pack_globals;
```

```
with Simple_io, Class_window, Class_dialog, Pack_globals;
use  Simple_io, Class_window, Class_dialog, Pack_globals;
function user_input( cb_mes:in String ) return String is
  miles   : Float;              -- Miles input by user
  last    : Positive;           --
  str_kms: String( 1 .. 10 ); -- As a string in Kms
  str_mls: String( 1 .. 10 ); -- As a string in Miles
begin
  begin
    get( cb_mes & ".", miles, last );
    put( str_kms, miles * 1.609_344, aft=>2, exp=>0 );
    put( str_mls, miles, aft=>2, exp=>0 );
    put( p_result.all, "Distance in miles = " );
    put( p_result.all, str_mls ); new_line( p_result.all );
    put( p_result.all, "Distance in Kms   = " );
    put( p_result.all, str_kms ); new_line( p_result.all );
  exception
    when Data_Error =>
      put( p_result.all, " Not a valid number" );
      new_line( p_result.all );
    when others =>
      put( p_result.all, " [Calculation error]" );
      new_line( p_result.all );
  end;
  return "";
end user_input;
```

```
with Class_input_manager, Class_window,
     Class_dialog, Pack_globals, user_input;
use  Class_input_manager, Class_window,
     Class_dialog, Pack_globals;
procedure main is
begin
  window_prolog;                    -- Setup window system
  declare
    result : aliased Window;        -- Result window
    input  : Dialog;                -- Input Window
    title  : Window;                -- title Window
```

```
  begin
    framework( title,   20,   1,   36, 5 );    -- Title Window
    framework( result, 30, 10,   36, 5 );    -- Result Window

    position( title, 8, 2 );
    put( title, "Miles to Kilometres" );
    framework( input, 5, 10, 22,              -- Input Window
               "Miles", user_input'Access );
```

```
    p_result := result'Unchecked_Access;
```

```
    window_start;                -- Start the user interaction
```

```
  end;
  window_epilog;                 -- Close window system
end main;
```

Putting it all together

```
framework( title,   20,   1,   36, 5 );

              (20,1)
               +-----------------------------------+
               |                                   |
               |       Miles to kilometres         |
               |                                   |
               +-----------------------------------+
                                            (20+36-1,1+5-1)

(5,20)                   (30,10)
  #--------------------+   +-----------------------------------+
  |Dialog| Miles       |   |                                   |
  |--------------------|   |                                   |
  |*                   |   |                                   |
  +--------------------+   +-----------------------------------+

                        framework( result, 30, 10, 36, 5 );

framework( input, 5, 10, 22, "Miles", user_input'Access );
```

```
                        +-----------------------------------+
                        |                                   |
                        |       Miles to kilometres         |
                        |                                   |
                        +-----------------------------------+


#---------------------+   +------------------------------------+
|Dialog| Miles        |   |Distance in miles =      50.00     |
|--------------------  |   |Distance in Kms   =      80.47     |
|50.0*                |   |                                    |
+---------------------+   +------------------------------------+
```

The menu system

Menu Component	Effect
About	Prints information about the program
Reset	Resets the program to an initial state

```
+----------------------+
|*About      | Reset   |
+----------------------+
```

Note	Function / procedure
1	`procedure framework(the:in out Menu'Class; m1:in String:=""; w1:in P_Menu:=null; cb1:in P_cbf:=null; m2:in String:=""; w2:in P_Menu:=null; cb2:in P_cbf:=null; m3:in String:=""; w3:in P_Menu:=null; cb3:in P_cbf:=null; m4:in String:=""; w4:in P_Menu:=null; cb4:in P_cbf:=null; m5:in String:=""; w5:in P_Menu:=null; cb5:in P_cbf:=null; m6:in String:=""; w6:in P_Menu:=null; cb6:in P_cbf:=null);`

Note 1 *This sets up a menu title bar or a menu title. The first parameter can be an instance of either a* `Menu` *or a* `Menu_title`*. Each menu item in the menu bar has three parameters:*

- *The displayed name of the menu item.*
- *A possible pointer to another menu bar.*
- *A possible pointer to a call-back function which is to be called when the menu is selected.*

The second and third parameter are mutually exclusive. Thus, you can have either another menu bar or a call-back function.

As the menu bar is always at the top of the screen its position is not selected. It would of course be an error to have a window overlapping the menu bar.

The type `P_cbf` *is defined as:*

```
type P_cbf is access function(str:in String) return
String;
```

```
with Class_input_manager, Class_window,
     Class_dialog, Class_menu, Class_menu_title,
     laser, ink_jet, about;
use  Class_input_manager, Class_window,
     Class_dialog, Class_menu, Class_menu_title;
procedure main is
begin
  window_prolog;
  declare
    menu_bar     : Menu_title;
    printer_type : aliased Menu;
  begin
    framework( printer_type,
               "Laser",   null, laser'Access,
               "Ink jet", null, ink_jet'Access );
    framework( menu_bar,
               "About",   null, about'Access,
               "Print",   printer_type'Unchecked_Access, null );
    window_start;
  end;
  window_epilog;
end main;
```

Main menu bar	Secondary menu bar

`+---------------------+`
`| About    |*Print   |`
`+---------------------+`
	`+---------------------+`
	`|*Laser    | Ink jet |`
	`+---------------------+`

Noughts and crosses program

Method	Responsibility
`add`	Add a piece to the board.
`reset`	Reset the board to empty.
`state`	Return the state of the board.
`update`	Update onto a window the state of the board.
`valid`	Check if the move is valid.

```
with Class_window;
use  Class_window;
package Class_board is
  type Board is private;

  type Game_state is ( WIN, PLAYABLE, DRAW );
  function  valid( the:in Board; pos:in Integer ) return Boolean;
  procedure add(the:in out Board; pos:in Integer;
                piece:in Character);
  function  state( the:in Board ) return Game_state;
  procedure display_board( the:in Board; win:in P_Window );
  procedure update( the:in Board; win:in P_Window );
  procedure reset( the:in out Board );

 private
  SIZE_TTT: constant := 9;                    -- Must be 9
  subtype Board_index is Integer range 1 .. SIZE_TTT;
  subtype Board_range is Board_index;
  type    Board_grid  is array( Board_range ) of Character;
  type Board is record
    sqrs  : Board_grid := ( others => ' ');      -- Initialize
    last  : Board_index := 1;                    -- Last move
    moves : Natural  := 0;
  end record;
end Class_board;
```

```
package body Class_board is

  function  valid(the:in Board; pos:in Integer) return Boolean is
  begin
    return pos in Board_range and then the.sqrs( pos ) = ' ';
  end valid;
```

```
  procedure add( the:in out Board; pos:in Integer;
                 piece:in Character ) is
  begin
    the.last := pos;
    the.sqrs( pos ) := piece;
    the.moves := the.moves + 1;
  end add;
```

```
  function  state( the:in Board ) return Game_state is
    type Win_line      is array( 1 .. 3 ) of Positive;
    type All_win_lines is range 1 .. 8;
    cells: constant array ( All_win_lines ) of Win_line :=
       ( (1,2,3), (4,5,6), (7,8,9), (1,4,7),
         (2,5,8), (3,6,9), (1,5,9), (3,5,7) ); -- All win lines
    first : Character;
  begin
    for pwl in All_win_lines loop          -- All Pos Win Lines
      first := the.sqrs( cells(pwl)(1) ); -- First cell in line
      if first /= ' ' then                 --  Looks promising
        if first = the.sqrs(cells(pwl)(2)) and then
           first = the.sqrs(cells(pwl)(3)) then return WIN;
        end if;
      end if;
    end loop;
    if the.moves >= 9                      -- Check for draw
      then return DRAW;                    --  Board full
      else return PLAYABLE;                --  Still playable
    end if;
  end state;
```

```
procedure reset( the:in out Board ) is
begin
  the.sqrs  := ( others => ' ');    -- All spaces
  the.last  := 1;                   -- Last move
  the.moves := 0;                   -- No of moves
end reset;
```

```
procedure display_board( the:in Board; win:in P_Window ) is
begin
  position( win.all, 1, 2 );
  put(win.all, " 7 | 8 | 9" ); new_line( win.all );
  put(win.all, " ---------" ); new_line( win.all );
  put(win.all, " 4 | 5 | 6" ); new_line( win.all );
  put(win.all, " ---------" ); new_line( win.all );
  put(win.all, " 1 | 2 | 3" ); new_line( win.all );
end display_board;
```

```
procedure update( the:in Board; win:in P_Window ) is
  type Co_ordinate is ( X , Y );
  type Cell_pos is array ( Co_ordinate ) of Positive;
  type Board    is array ( 1 .. SIZE_TTT ) of Cell_pos;
  pos: constant Board :=    ( (2,6), (6,6), (10,6),
                              (2,4), (6,4), (10,4),
                              (2,2), (6,2), (10,2) );

begin
  position( win.all, pos(the.last)(X), pos(the.last)(Y) );
  put( win.all, the.sqrs( the.last ) );    -- Display counter;
end update;

end Class_board;
```

```
with Class_board, Class_window;
use  Class_board, Class_window;
package Pack_globals is
  game      : Board;        -- The board
  p_win_brd : P_Window;     -- Window to display OXO board in
  p_win_bnr : P_Window;     -- Window to display Banner in
  p_win_r   : P_Window;     -- Window to display commentary in
  player    : Character;    -- Either 'X' or 'O'
end Pack_globals;
```

```
with Simple_io, Class_window, Class_board, Pack_globals;
use  Simple_io, Class_window, Class_board, Pack_globals;
function user_input( cb_mes:in String ) return String is
  move: Integer; last: Positive;
begin
  clear( p_win_r.all );                        -- Clear
  get( cb_mes, move, last );                   -- to int
  if valid( game, move ) then                  -- Valid
    add( game, move, player );                 -- to board
    update( game, p_win_brd );
```

```
    case state( game ) is                      -- Game is
      when WIN      =>
        put(p_win_r.all, " " & player & " wins");
      when PLAYABLE  =>
        case player is                         -- Next player
          when 'X'    => player := 'O';        --   'X' => 'O'
          when 'O'    => player := 'X';        --   'O' => 'X'
          when others => null;                 --
        end case;
        put( p_win_r.all, " Player " & player );
      when DRAW     =>
        put( p_win_r.all, " It's a draw ");
    end case;
  else
    put(p_win_r.all, " " & player & " Square invalid");
  end if;
  return "";
exception
  when others =>
    put(p_win_r.all, " " & player & " re-enter move");
    return "";
end user_input;
```

```
with Class_window, Class_board, Pack_globals;
use  Class_window, Class_board, Pack_globals;
procedure re_start( first_player:in Character ) is
begin
  player := first_player;                     -- Start with
  reset( game );                              -- Reset Board
  display_board(game, p_win_brd );            -- Display
  clear( p_win_r.all );                       -- Status info
  put( p_win_r.all, " Player " & player ); -- Player name
end re_start;

with re_start;
function reset_x( cb_mes:in String ) return String is
begin
  re_start('X'); return "";
end reset_x;

with re_start;
function reset_o( cb_mes:in String ) return String is
begin
  re_start('O'); return "";
end reset_o;
```

```
with Class_window, Pack_globals;
use  Class_window, Pack_globals;
function about( cb_mes:in String ) return String is
begin
  clear( p_win_bnr.all ); position( p_win_bnr.all, 17, 1 );
  put( p_win_bnr.all, "Written in Ada 95");
  return "";
end about;
```

```
with Class_input_manager, Class_board, Class_window,
     Class_dialog, Class_menu, Class_menu_title,
     Pack_globals, reset_x, reset_o, about, user_input;
use  Class_input_manager, Class_board, Class_window,
     Class_dialog, Class_menu, Class_menu_title, Pack_globals;
procedure play is
begin
  window_prolog;                   -- Setup window system
  declare
    win_brd   : aliased Window; -- Board Window
    win_r     : aliased Window; -- Result Window
    win_bnr   : aliased Window; -- title Window
    win_usr   : aliased Dialog; -- Input Window
    ttt_reset: aliased Menu;    -- Reset menu
    ttt_menu : Menu_title;      -- Title menu
```

```
  begin
    framework( win_bnr,  1,  4, 52, 3 );     -- Banner
    framework( win_brd, 32,  8, 13, 9 );     -- OXO board
    framework( win_r,    9, 14, 22, 3 );     -- Results
```

```
    framework( ttt_reset,
               "X start",  null,  reset_x'Access,
               "O start",  null,  reset_o'Access  );

    framework( ttt_menu,
               "About",    null,  about'Access,
               "Reset",    ttt_reset'Unchecked_Access,  null );
```

```
    position( win_bnr, 17, 1 );
    put( win_bnr, "Noughts and crosses" );

    framework( win_usr,  9, 8, 22,
               "Move (1-9)", user_input'Access );

    player := 'X';                            -- Set player
    p_win_brd := win_brd'Unchecked_Access;    -- OXO Board
    p_win_bnr := win_bnr'Unchecked_Access;    -- Banner
    p_win_r   := win_r'Unchecked_Access;      -- Commentary

    display_board( game, p_win_brd );         -- Empty board
    new_line( win_r );                        -- Clear
    put( win_r, " Player " & player );        -- Players turn is

    put( win_usr, "" );       -- Cursor
```

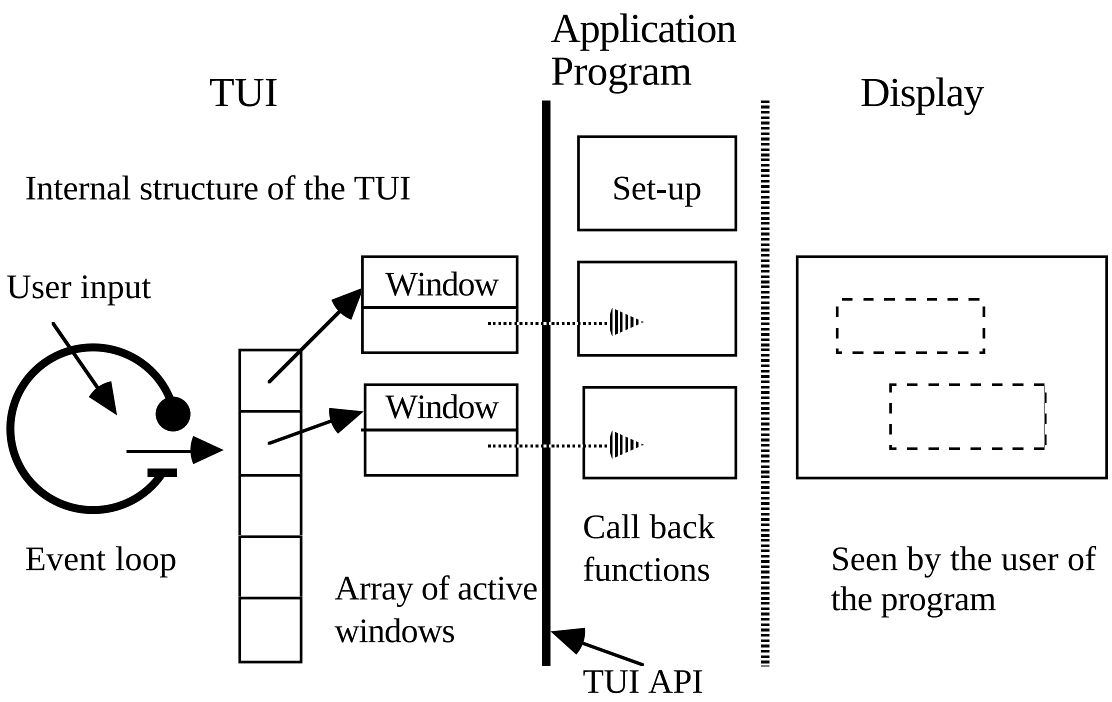
```
    window_start;             -- Start the user interaction
  end;
  window_epilog;              -- Close window system
end play;
```

Putting it all together

```
+----------------------+
|*About     | Reset    |
+----------------------+
+----------------------------------------------------+
|                Noughts and crosses                 |
+----------------------------------------------------+

        #--------------------+ +-----------+
        |Dialog| Move (1-9)  | |           |
        |--------------------| | 7 | 8 | 9 |
        |*                   | | --------- |
        +--------------------+ | 4 | 5 | 6 |
                               | --------- |
        +--------------------+ | 1 | 2 | 3 |
        | Player X           | |           |
        +--------------------+ +-----------+
```

X's move	Commentary	O's move	Commentary
1	Claim the centre square	2	Not the correct move
3	Setting up a win	4	Block the X's
5	Two win lines	6	Block one of them
7	Win with three X's		

```
+----------------------+
|*About     | Reset    |
+----------------------+
+----------------------------------------------------+
|                Noughts and crosses                 |
+----------------------------------------------------+

        #--------------------+ +-----------+
        |Dialog| Move (1-9)  | |           |
        |--------------------| | X | O | X |
        |*                   | | --------- |
        +--------------------+ | 4 | X | O |
                               | --------- |
        +--------------------+ | O | 2 | X |
        | X wins             | |           |
        +--------------------+ +-----------+
```



Application
Program
TUI
Display
Internal structure of the TUI
Set-up
User input
Window
Window
Event loop
Call back
functions
Seen by the user of
the program
Array of active
windows
TUI API