ADA

By Team Eda

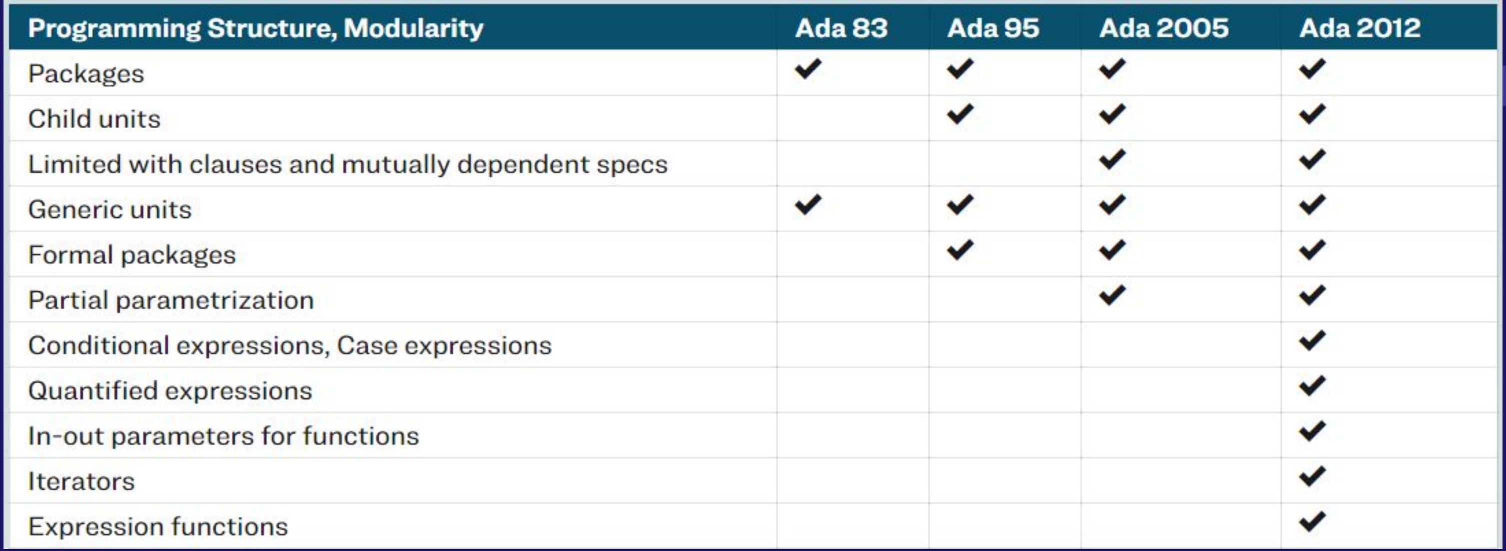
Batucan, Bohol, Bolabola, Canoy

HISTORY

Ada is a winning proposal submitted by Jean Ichbiah from CII Honeywell-Bull on the United State Department of Defense on 1970s.

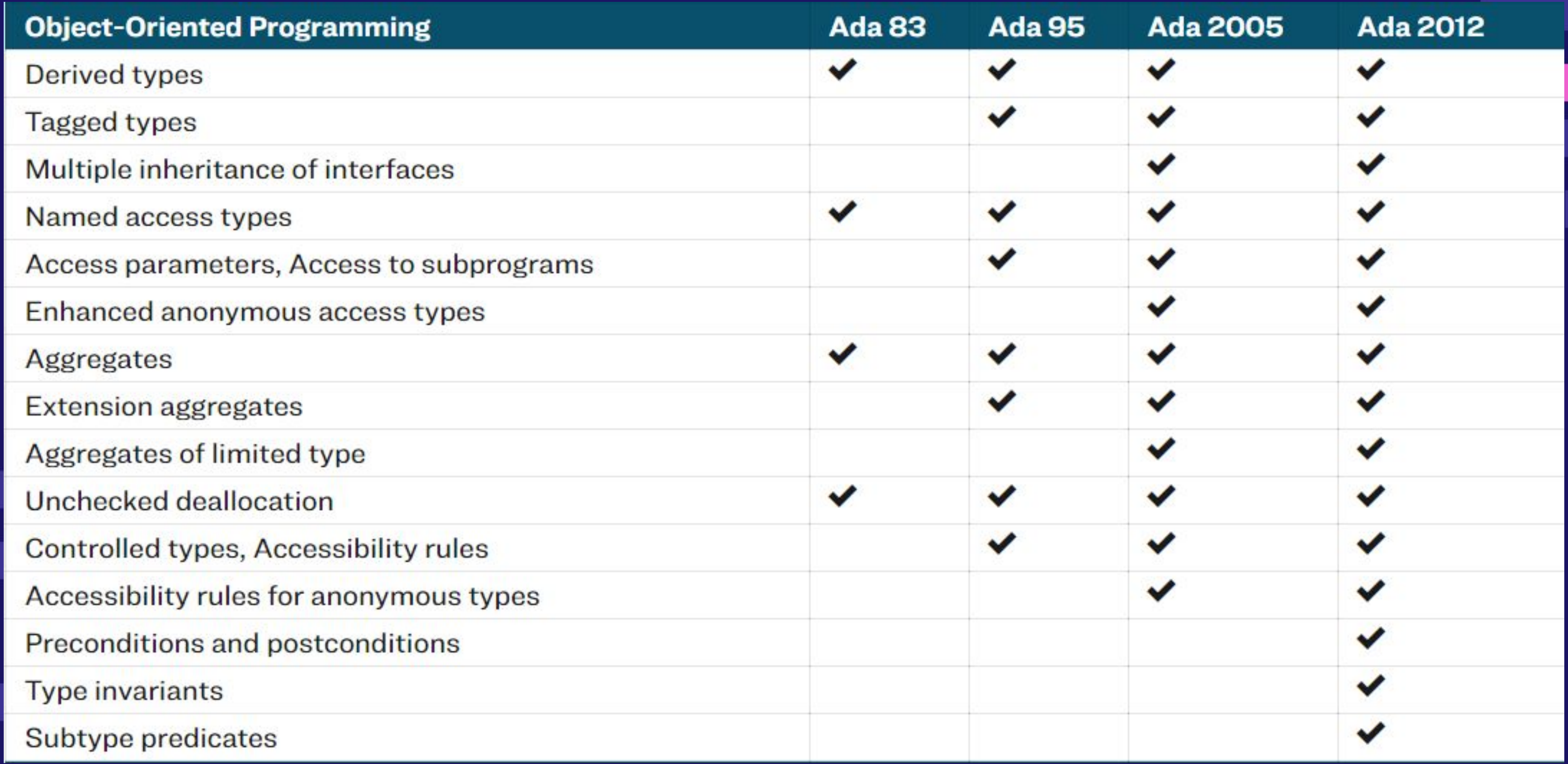
It was issued in 1983, subsequently revised and enhanced in 1995, 2005 and 2012, with each revision bringing useful new features.

HISTORY



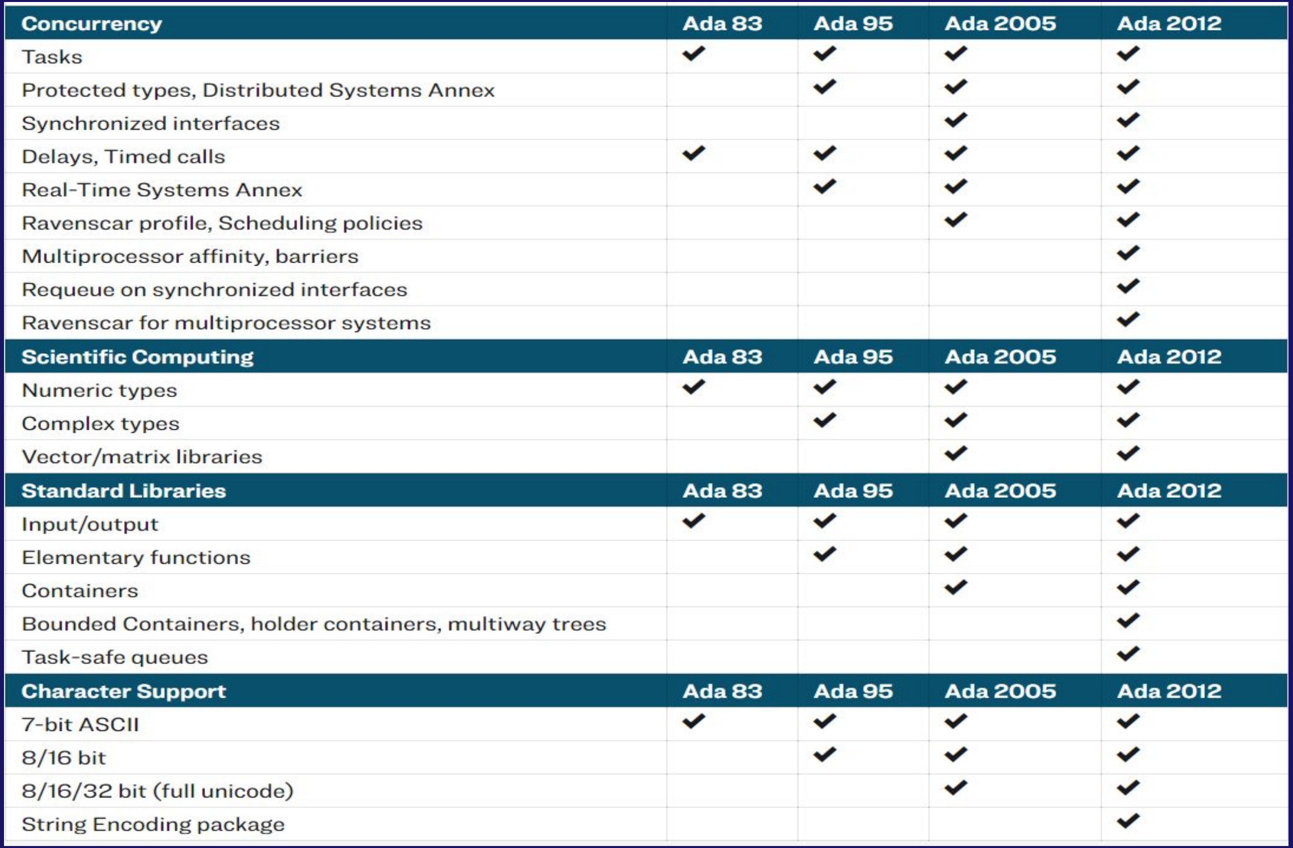
Programming Structure, Modularity	Ada 83	Ada 95	Ada 2005	Ada 2012
Packages	✓	✓	✓	✓
Child units		✓	✓	✓
Limited with clauses and mutually dependent specs			✓	✓
Generic units	✓	✓	✓	✓
Formal packages		✓	✓	✓
Partial parametrization			✓	✓
Conditional expressions, Case expressions				✓
Quantified expressions				✓
In-out parameters for functions				✓
Iterators				✓
Expression functions				✓

HISTORY



Object-Oriented Programming	Ada 83	Ada 95	Ada 2005	Ada 2012
Derived types	✓	✓	✓	✓
Tagged types		✓	✓	✓
Multiple inheritance of interfaces			✓	✓
Named access types	✓	✓	✓	✓
Access parameters, Access to subprograms		✓	✓	✓
Enhanced anonymous access types			✓	✓
Aggregates	✓	✓	✓	✓
Extension aggregates		✓	✓	✓
Aggregates of limited type			✓	✓
Unchecked deallocation	✓	✓	✓	✓
Controlled types, Accessibility rules		✓	✓	✓
Accessibility rules for anonymous types			✓	✓
Preconditions and postconditions				✓
Type invariants				✓
Subtype predicates				✓

HISTORY



Concurrency	Ada 83	Ada 95	Ada 2005	Ada 2012
Tasks	✓	✓	✓	✓
Protected types, Distributed Systems Annex		✓	✓	✓
Synchronized interfaces			✓	✓
Delays, Timed calls	✓	✓	✓	✓
Real-Time Systems Annex		✓	✓	✓
Ravenscar profile, Scheduling policies			✓	✓
Multiprocessor affinity, barriers				✓
Requeue on synchronized interfaces				✓
Ravenscar for multiprocessor systems				✓

Scientific Computing	Ada 83	Ada 95	Ada 2005	Ada 2012
Numeric types	✓	✓	✓	✓
Complex types		✓	✓	✓
Vector/matrix libraries			✓	✓

Standard Libraries	Ada 83	Ada 95	Ada 2005	Ada 2012
Input/output	✓	✓	✓	✓
Elementary functions		✓	✓	✓
Containers			✓	✓
Bounded Containers, holder containers, multiway trees				✓
Task-safe queues				✓

Character Support	Ada 83	Ada 95	Ada 2005	Ada 2012
7-bit ASCII	✓	✓	✓	✓
8/16 bit		✓	✓	✓
8/16/32 bit (full unicode)			✓	✓
String Encoding package				✓

HISTORY

- ➔ Embedded systems with low memory requirements (no garbage collector allowed).
- ➔ Direct interfacing with hardware.
- ➔ Soft or hard real-time systems.
- ➔ Low-level systems programming

DESIGN PRINCIPLES

Readability

Keywords are preferred than symbols and no keyword is an abbreviation, etc.

DESIGN PRINCIPLES

Very strong typing

It is very easy to introduce new types in ADA, with the benefit of preventing data usage errors.

DESIGN PRINCIPLES

Explicit is better than implicit

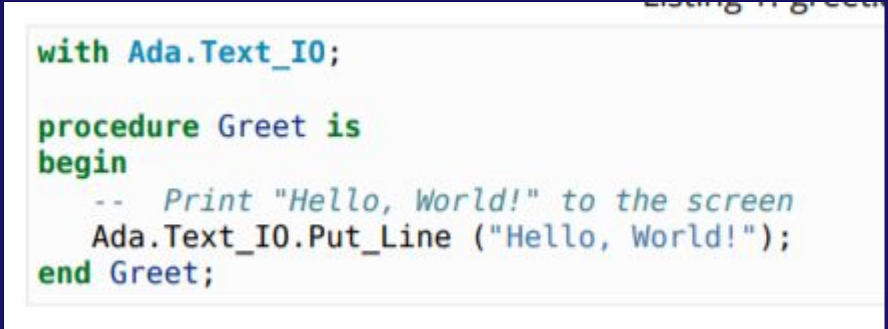
- Mostly no structural typing
- Mostly no type of inference
- Semantics are very well defined
- The programmer can give a lot of information about what their program means to the compiler

IMPERATIVE LANGUAGE

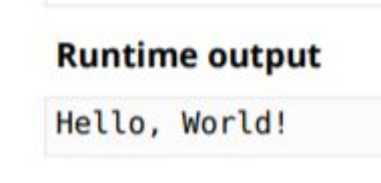
ADA is a multi-paradigm language with support for object orientation and some elements of functional programming

Its core is a simple, coherent procedural/ imperative language akin to C or Pascal.

A very simple imperative Ada program



```ada
with Ada.Text_IO;

procedure Greet is
begin
   --  Print "Hello, World!" to the screen
   Ada.Text_IO.Put_Line ("Hello, World!");
end Greet;
```



Runtime output

```
Hello, World!
```

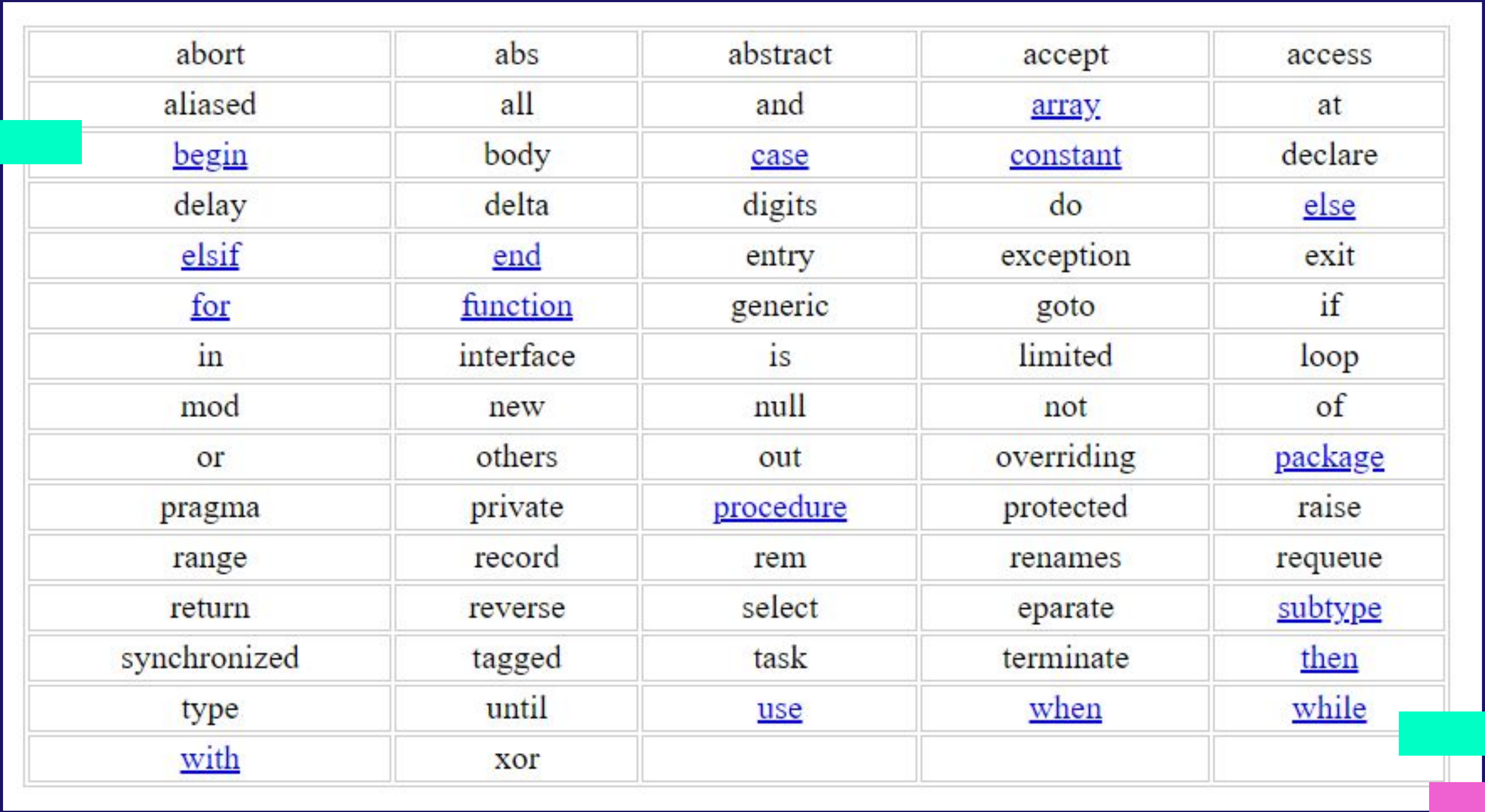
DECLARING A VARIABLE

```
VariableName : DataType;
```

- The name of a variable must be in one word
- It must start with a letter
- It can include a combination of letters, digits and underscores
- It must not contain special characters

DECLARING MULTIPLE VARIABLES

```
VariableName1, VariableName2 : DataType1;
VariableName1 : DataType1;
VariableName2: DataType1;
```

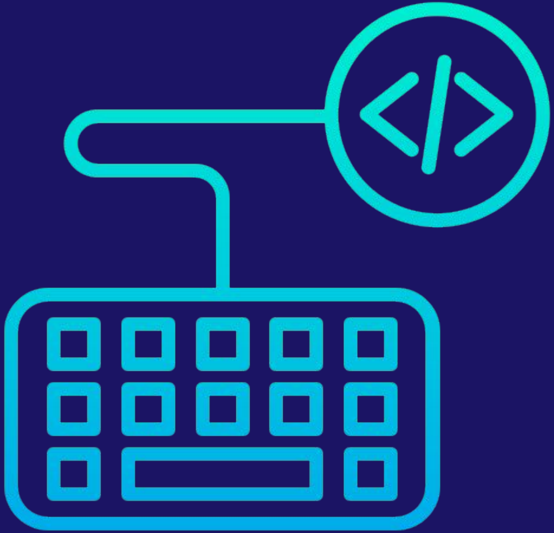


abort	abs	abstract	accept	access
aliased	all	and	array	at
begin	body	case	constant	declare
delay	delta	digits	do	else
elsif	end	entry	exception	exit
for	function	generic	goto	if
in	interface	is	limited	loop
mod	new	null	not	of
or	others	out	overriding	package
pragma	private	procedure	protected	raise
range	record	rem	renames	requeue
return	reverse	select	eparate	subtype
synchronized	tagged	task	terminate	then
type	until	use	when	while
with	xor			

INITIALIZING A VARIABLE

```
VariableName : Datatype := Value;
VariableName : DataType
  begin
     VariableName := Value;
  end
```

Data Types

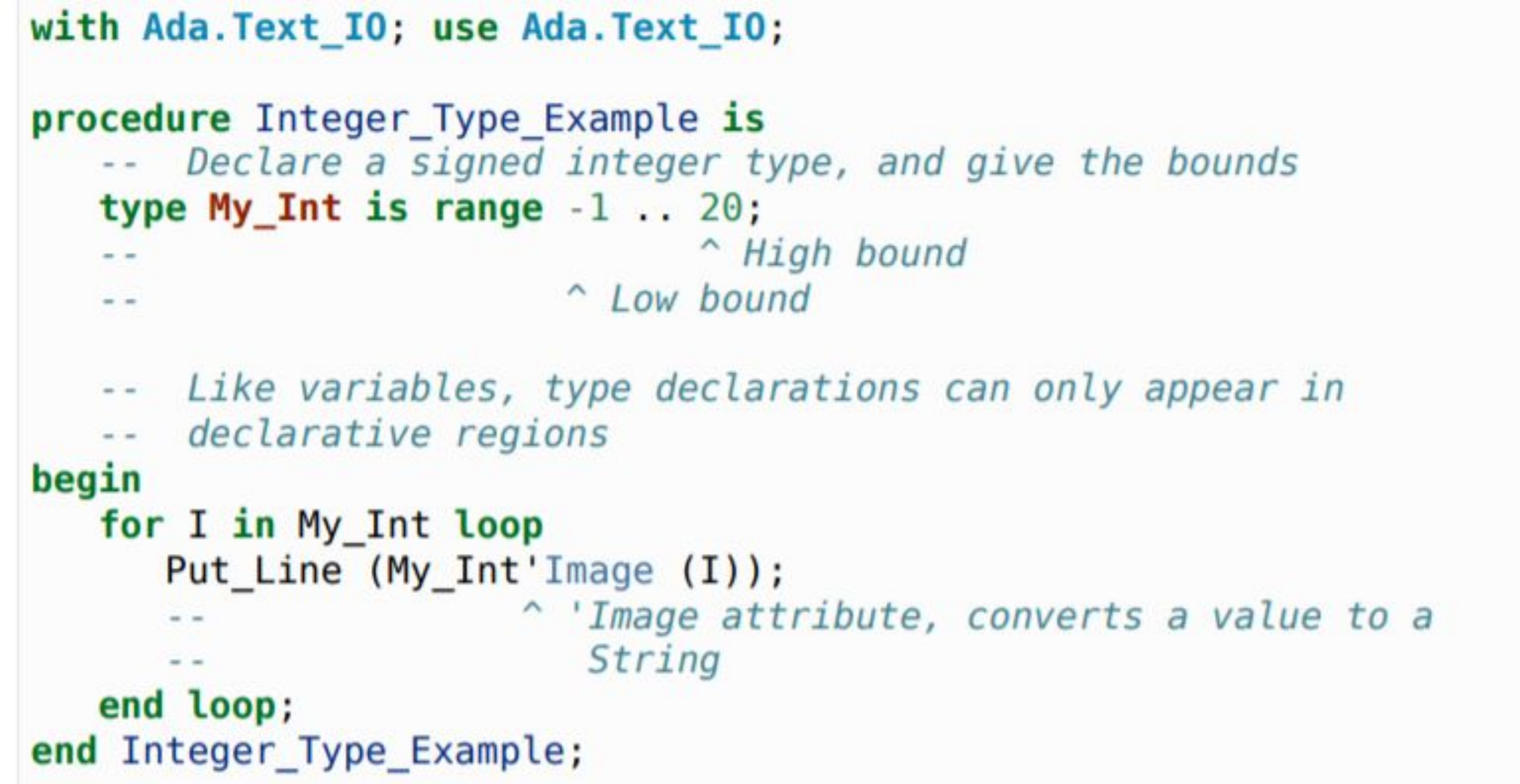


Integer

- is a numeric value for a natural number.

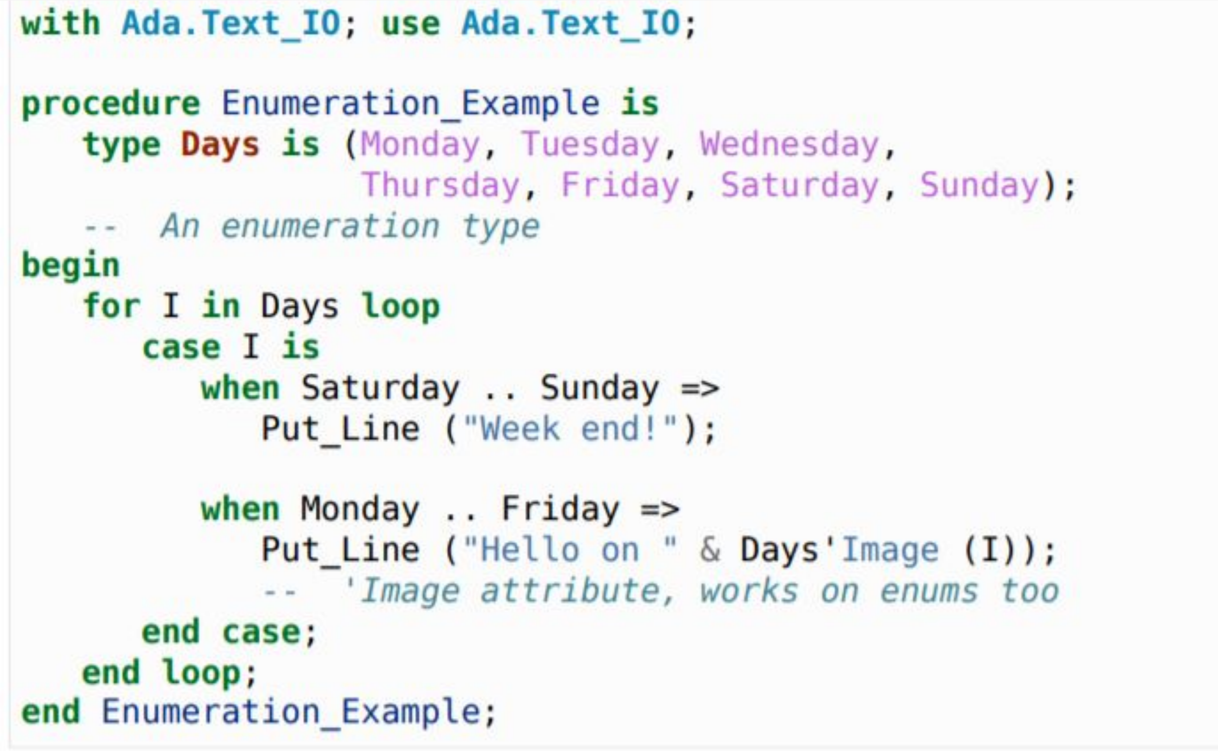
```
procedure Exercise is
    number : integer := 214685;
begin
end Exercise;
```

Integer



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

procedure Integer_Type_Example is
   --  Declare a signed integer type, and give the bounds
   type My_Int is range -1 .. 20;
   --                         ^ High bound
   --                   ^ Low bound

   --  Like variables, type declarations can only appear in
   --  declarative regions
begin
   for I in My_Int loop
      Put_Line (My_Int'Image (I));
      --              ^ 'Image attribute, converts a value to a
      --                 String
   end loop;
end Integer_Type_Example;
```

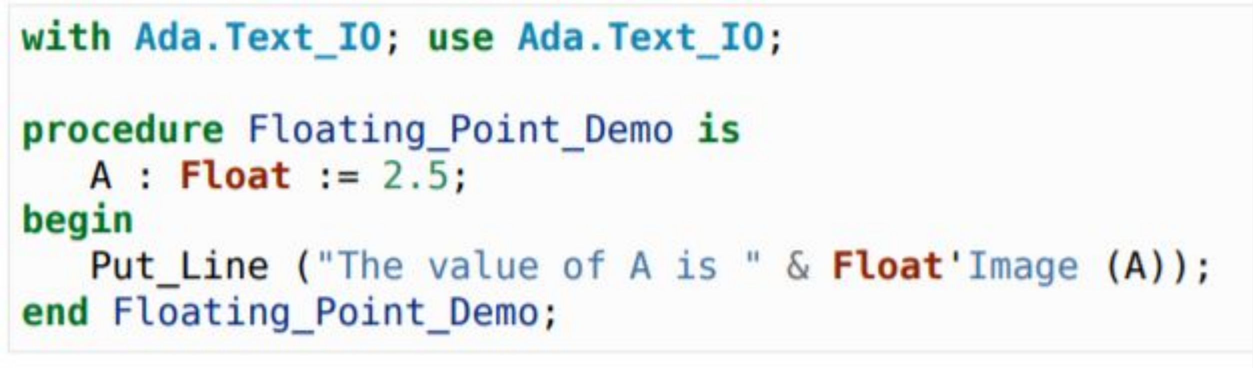
Enumerations



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

procedure Enumeration_Example is
   type Days is (Monday, Tuesday, Wednesday,
                 Thursday, Friday, Saturday, Sunday);
   --  An enumeration type
begin
   for I in Days loop
      case I is
         when Saturday .. Sunday =>
            Put_Line ("Week end!");

         when Monday .. Friday =>
            Put_Line ("Hello on " & Days'Image (I));
            --  'Image attribute, works on enums too
      end case;
   end loop;
end Enumeration_Example;
```

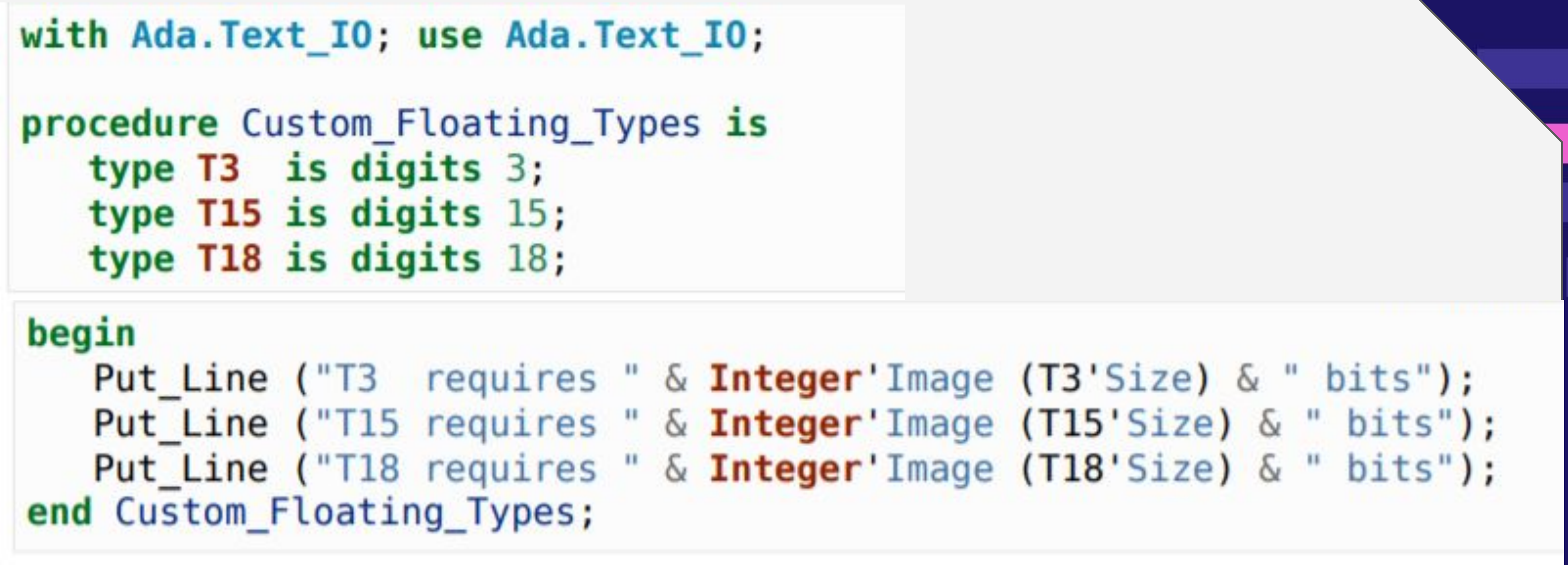
Floating Point Numbers

- use the float keyword

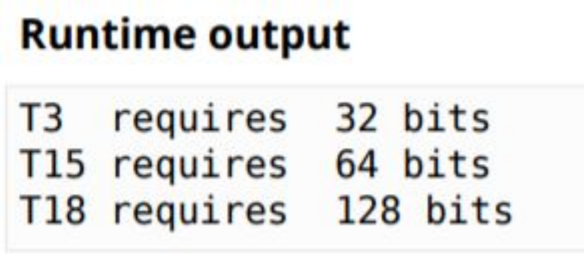


```
with Ada.Text_IO; use Ada.Text_IO;

procedure Floating_Point_Demo is
   A : Float := 2.5;
begin
   Put_Line ("The value of A is " & Float'Image (A));
end Floating_Point_Demo;
```

Precision of floating-point types



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

procedure Custom_Floating_Types is
   type T3  is digits 3;
   type T15 is digits 15;
   type T18 is digits 18;
begin
   Put_Line ("T3  requires " & Integer'Image (T3'Size) & " bits");
   Put_Line ("T15 requires " & Integer'Image (T15'Size) & " bits");
   Put_Line ("T18 requires " & Integer'Image (T18'Size) & " bits");
end Custom_Floating_Types;
```



Runtime output

```
T3  requires  32 bits
T15 requires  64 bits
T18 requires  128 bits
```

Character

- is a letter, a symbol, or a digit

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

   procedure Welcome is
      gender : character := ‘M’;
   begin
      Put_Line(“Gender = “&character’image(gender));
   end Welcome;
```

String

– a combination of characters. To represent strings, ADA uses the String data type.

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

   procedure Exercise is
      sentence : String := “Welcome to the wonderful
world of Ada programming!”;
   begin
      Put_Line(sentence);
   end Exercise;
```

Constant

- use the constant keyword

```
procedure Welcome is
   value1 : constant integer := 605;
begin
   Put_Line("Value 1 = " & integer'image(value1));
end Welcome;
```

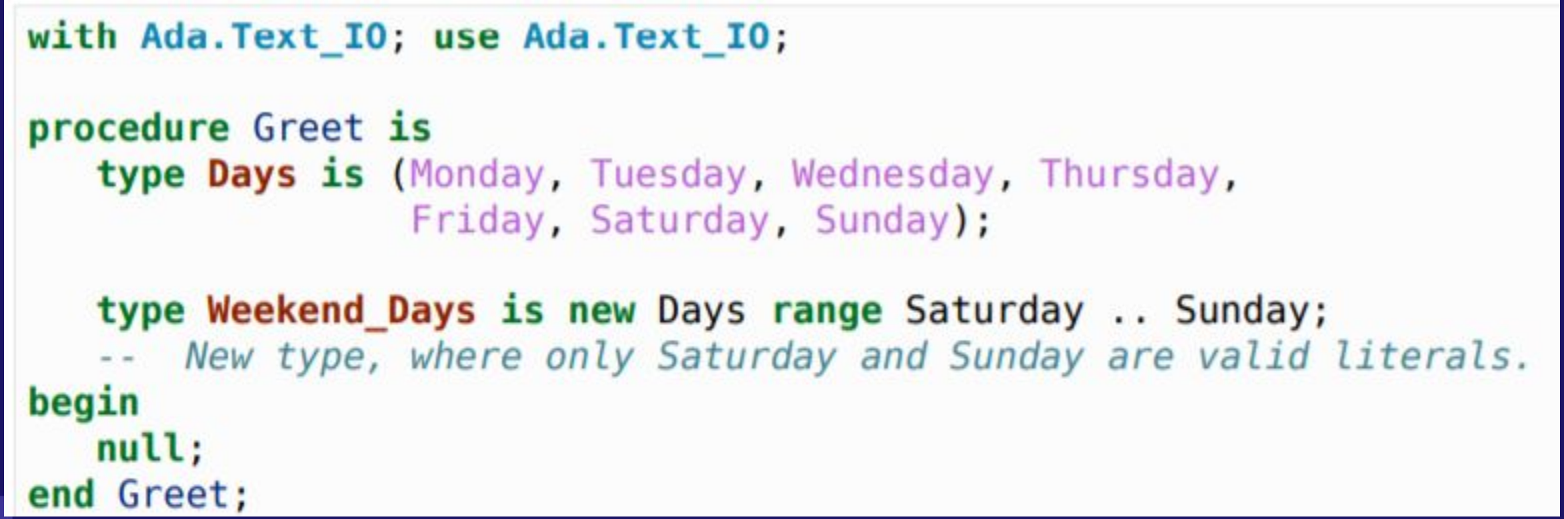
Boolean

- use the Boolean keyword (Default is False)

```
procedure Exercise is
   IsDrunk : Boolean;

begin
 ..
end Exercise;
```

Derived types

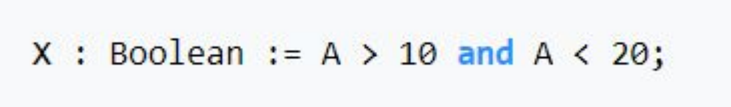
- create new types based on existing ones



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

procedure Greet is
   type Days is (Monday, Tuesday, Wednesday, Thursday,
                 Friday, Saturday, Sunday);

   type Weekend_Days is new Days range Saturday .. Sunday;
   --  New type, where only Saturday and Sunday are valid literals.
begin
   null;
end Greet;
```

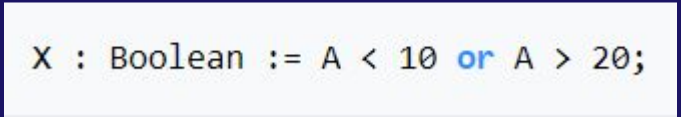
Logical Operators

AND



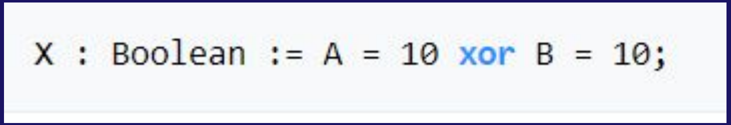
```
X : Boolean := A > 10 and A < 20;
```

OR

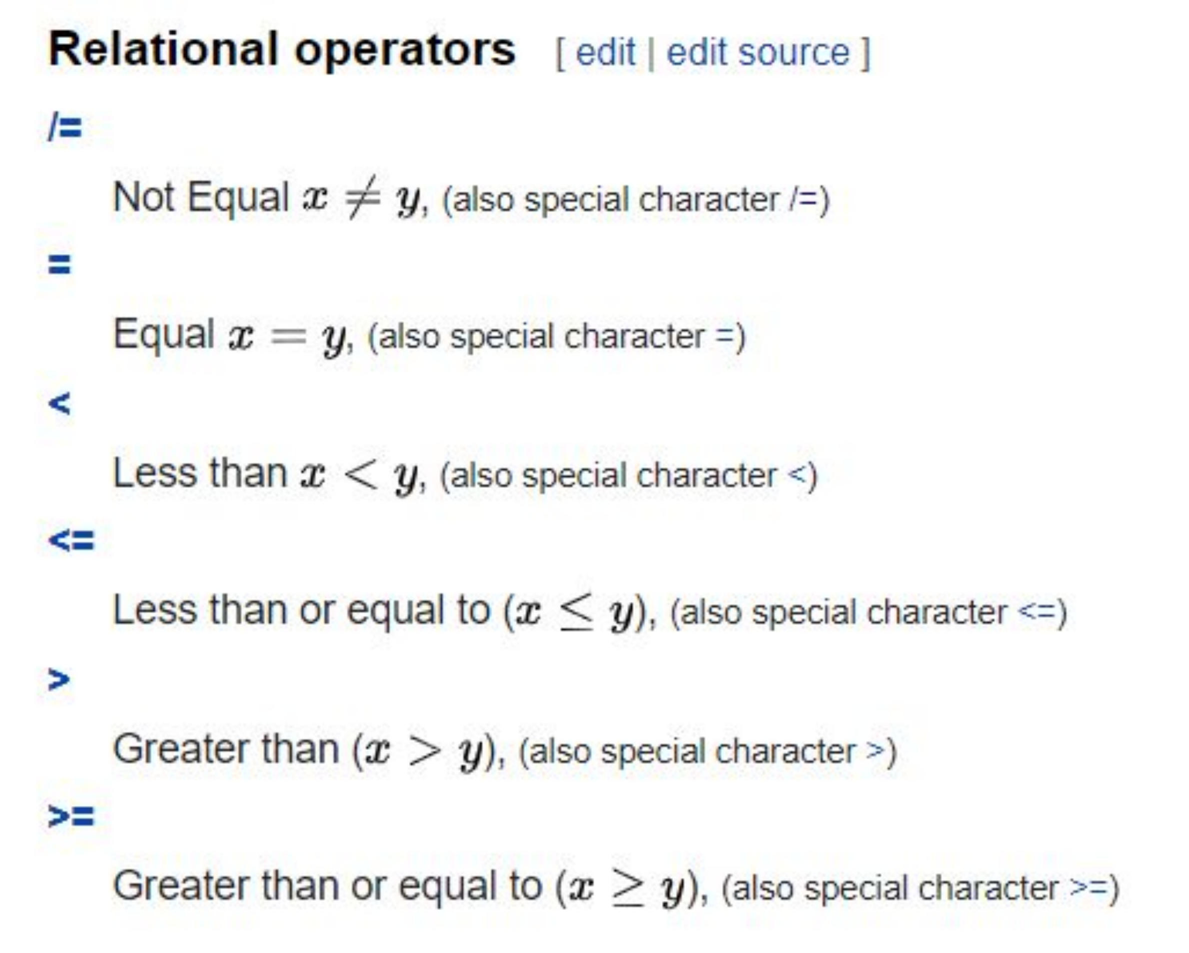


```
X : Boolean := A < 10 or A > 20;
```

XOR



```
X : Boolean := A = 10 xor B = 10;
```



Relational operators [edit | edit source]

/=

Not Equal $x \neq y$, (also special character /=)

=

Equal $x = y$, (also special character =)

<

Less than $x < y$, (also special character <)

<=

Less than or equal to $(x \leq y)$, (also special character <=)

>

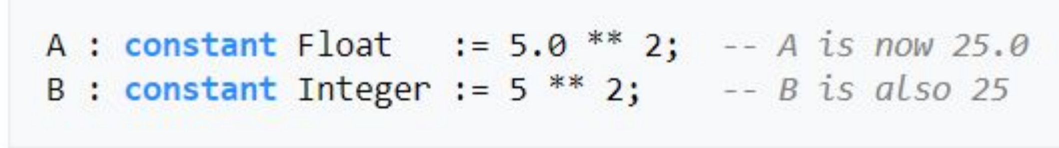
Greater than $(x > y)$, (also special character >)

>=

Greater than or equal to $(x \geq y)$, (also special character >=)

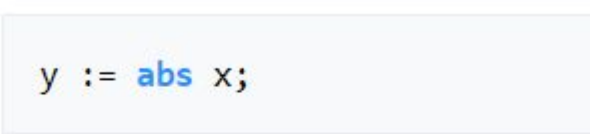
Highest precedence operator

Power **

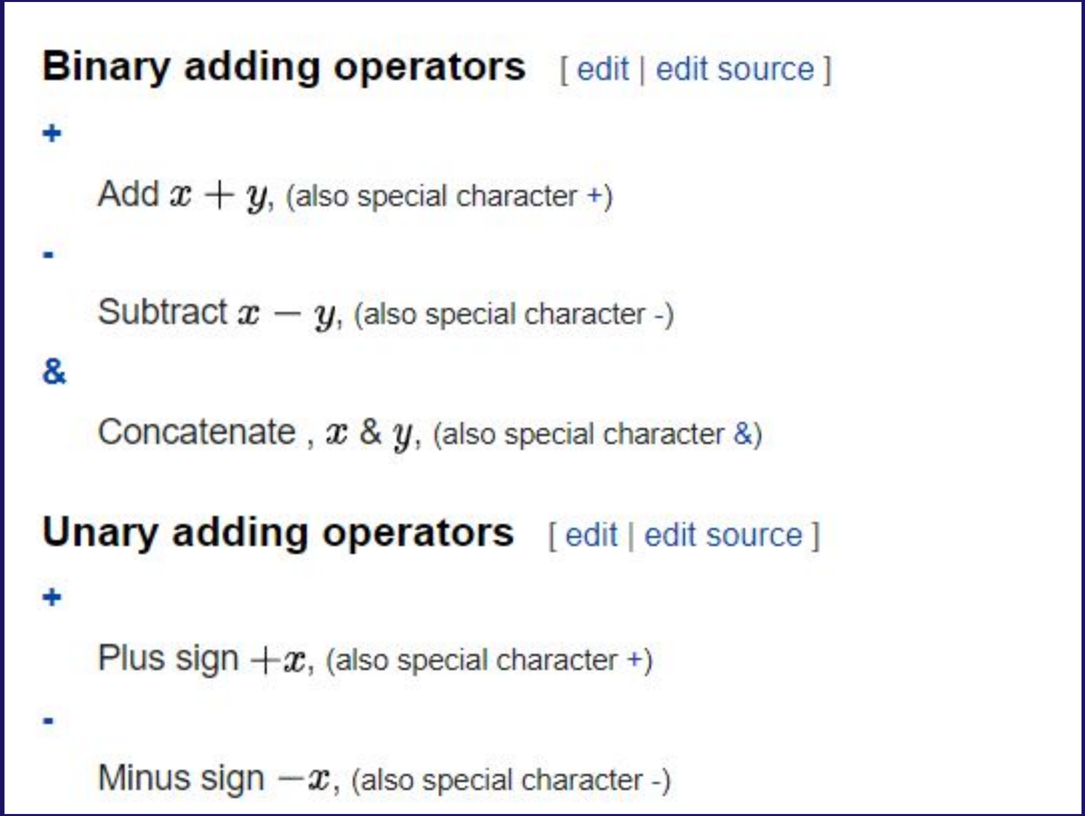


```
A : constant Float   := 5.0 ** 2;  -- A is now 25.0
B : constant Integer := 5 ** 2;    -- B is also 25
```

Absolute



```
y := abs x;
```



Binary adding operators [edit | edit source]

+

Add $x + y$, (also special character +)

-

Subtract $x - y$, (also special character -)

&

Concatenate , $x \& y$, (also special character &)

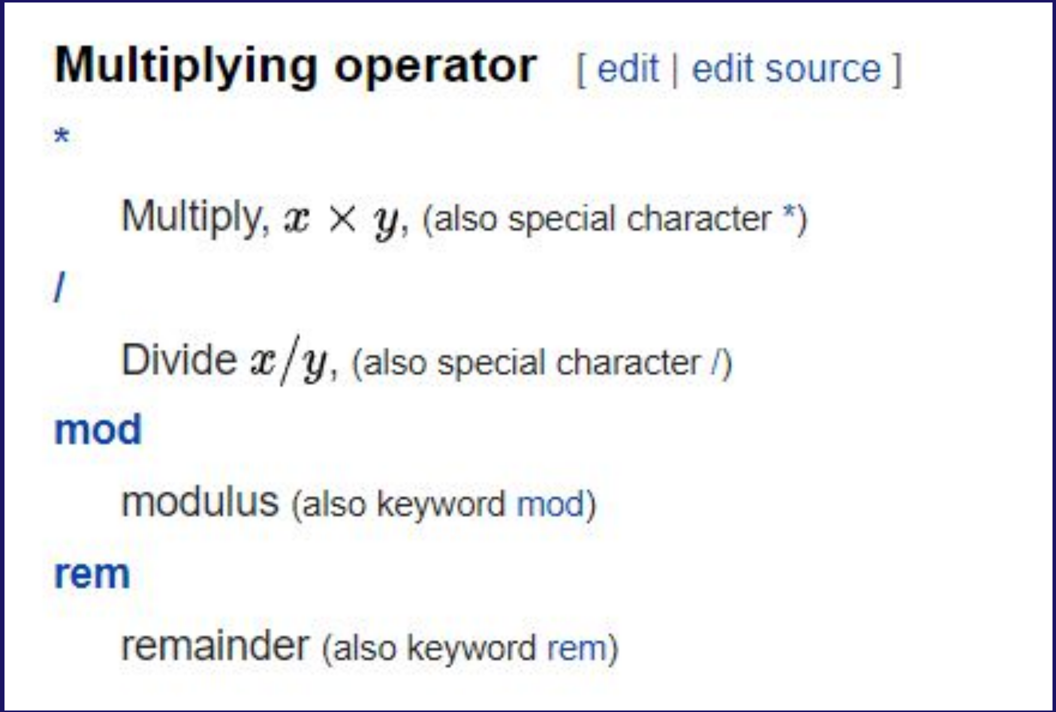
Unary adding operators [edit | edit source]

+

Plus sign $+x$, (also special character +)

-

Minus sign $-x$, (also special character -)



Multiplying operator [edit | edit source]

*

Multiply, $x \times y$, (also special character *)

/

Divide x/y, (also special character /)

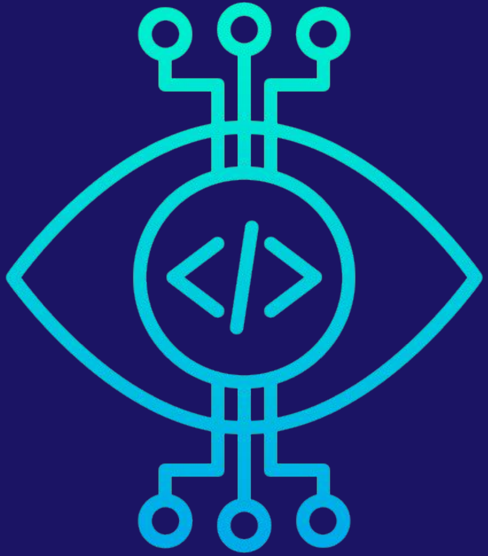
mod

modulus (also keyword mod)

rem

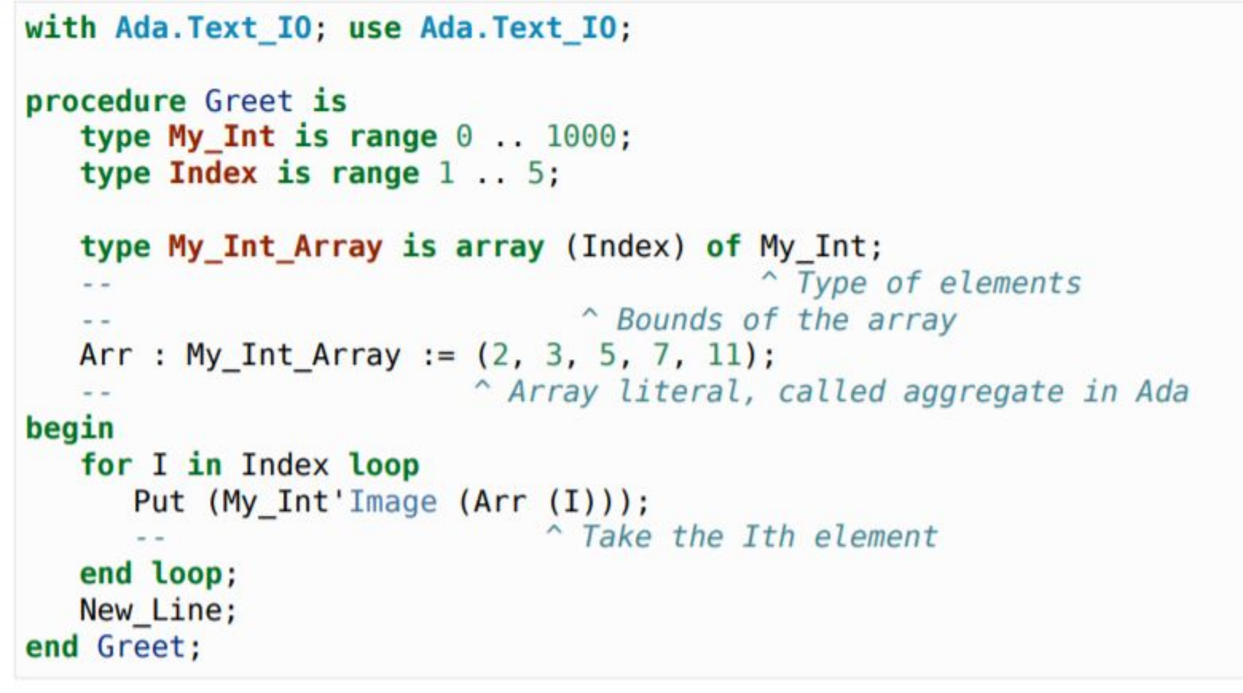
remainder (also keyword rem)

DATA STRUCTURES



Array

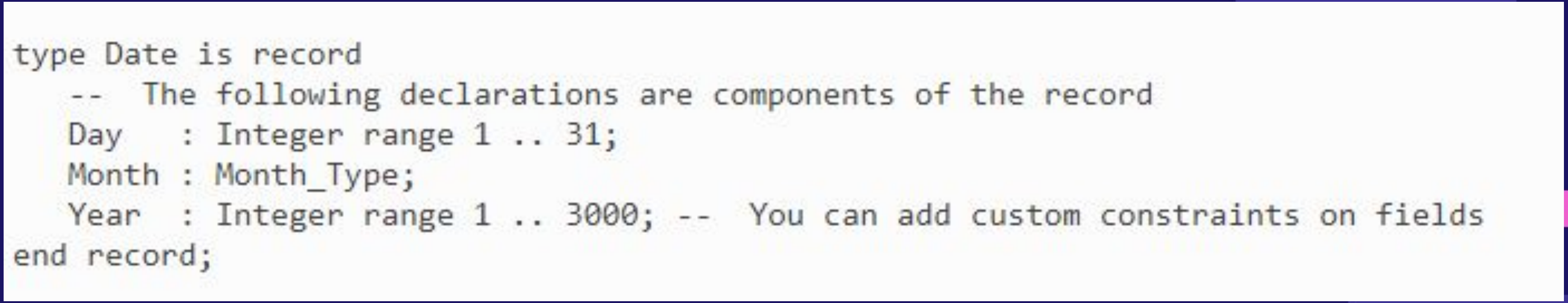
- to define contiguous collections of elements that can be selected by indexing



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

procedure Greet is
   type My_Int is range 0 .. 1000;
   type Index is range 1 .. 5;

   type My_Int_Array is array (Index) of My_Int;
   --                                    ^ Type of elements
   --                          ^ Bounds of the array
   Arr : My_Int_Array := (2, 3, 5, 7, 11);
   --                    ^ Array literal, called aggregate in Ada
begin
   for I in Index loop
      Put (My_Int'Image (Arr (I)));
      --                     ^ Take the Ith element
   end loop;
   New_Line;
end Greet;
```

Records

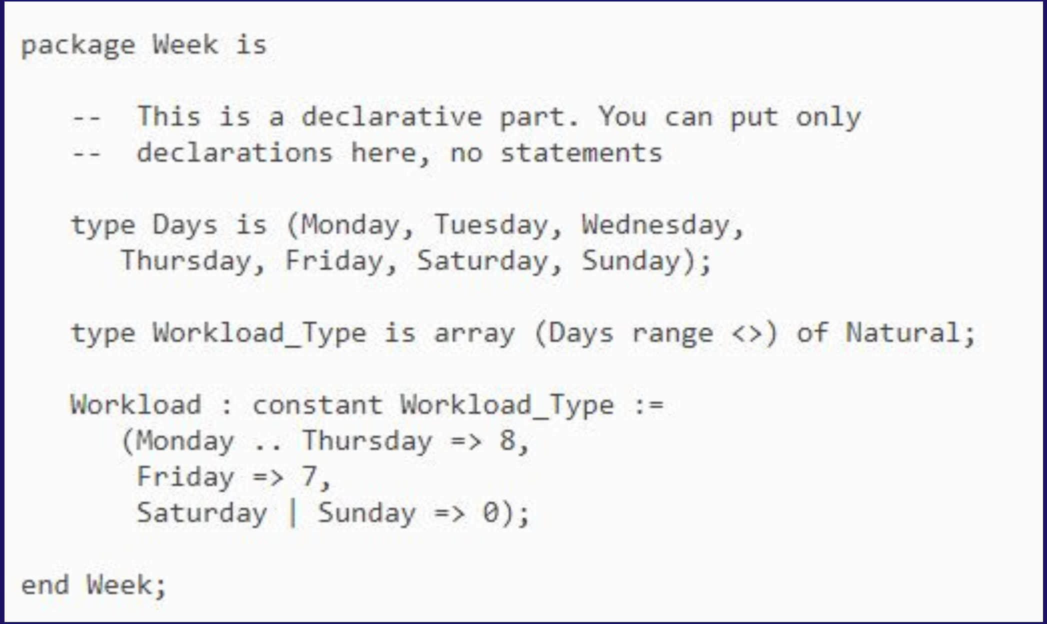
- a way to piece together several instances of other types



```
type Date is record
   --  The following declarations are components of the record
   Day    : Integer range 1 .. 31;
   Month  : Month_Type;
   Year   : Integer range 1 .. 3000; --  You can add custom constraints on fields
end record;
```

Packages

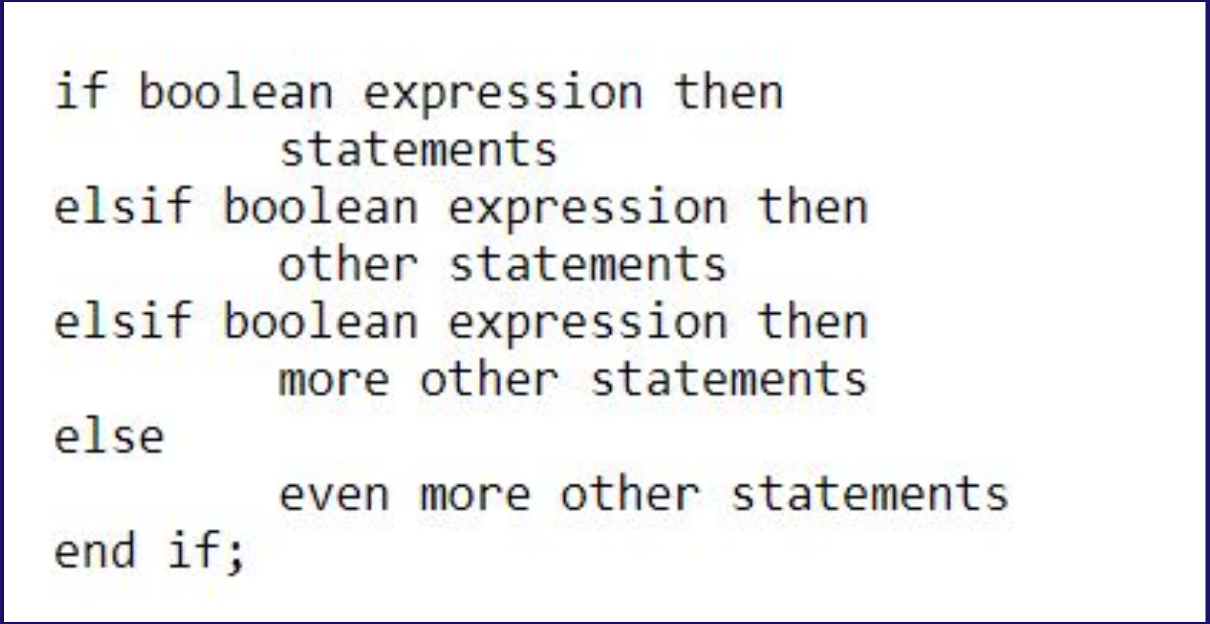
– a way to make your code modular



```
package Week is

   --  This is a declarative part. You can put only
   --  declarations here, no statements

   type Days is (Monday, Tuesday, Wednesday,
      Thursday, Friday, Saturday, Sunday);

   type Workload_Type is array (Days range <>) of Natural;

   Workload : constant Workload_Type :=
      (Monday .. Thursday => 8,
       Friday => 7,
       Saturday | Sunday => 0);

end Week;
```

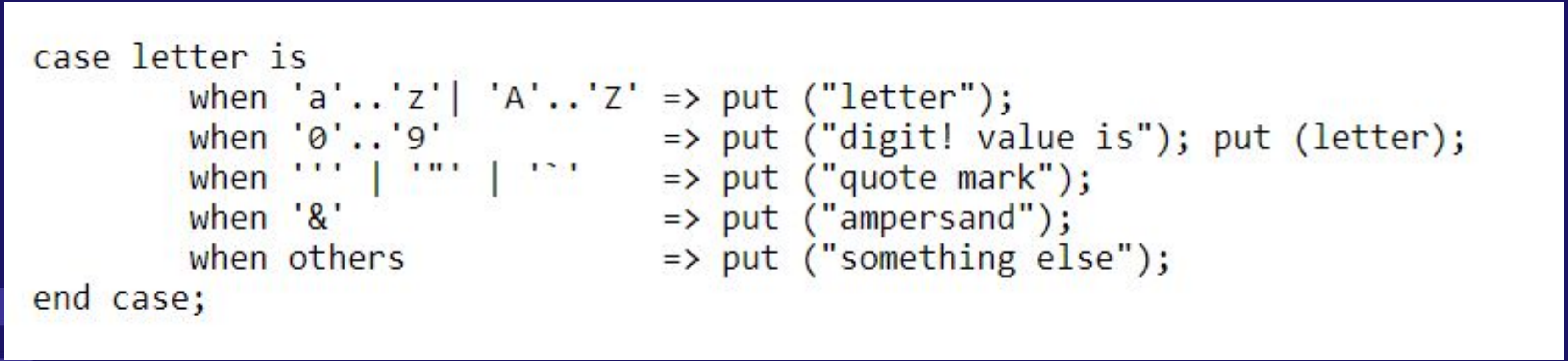
CONTROL STRUCTURES

If statements



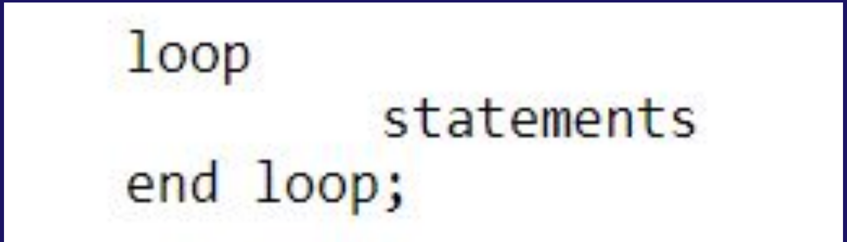
```
if boolean expression then
        statements
elsif boolean expression then
        other statements
elsif boolean expression then
        more other statements
else
        even more other statements
end if;
```

Case statements



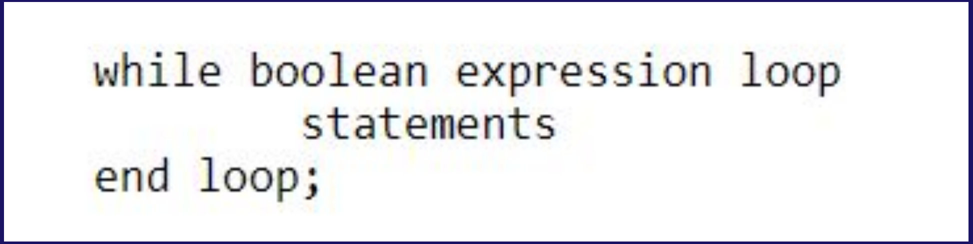
```
case letter is
        when 'a'..'z'| 'A'..'Z' => put ("letter");
        when '0'..'9'           => put ("digit! value is"); put (letter);
        when ''' | '"' | '`'    => put ("quote mark");
        when '&'                => put ("ampersand");
        when others             => put ("something else");
end case;
```

Simple Loops



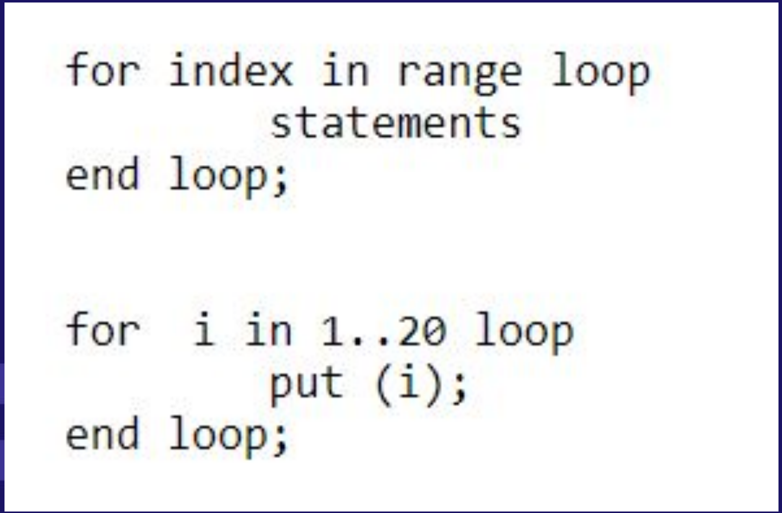
```
loop
            statements
end loop;
```

While Loops



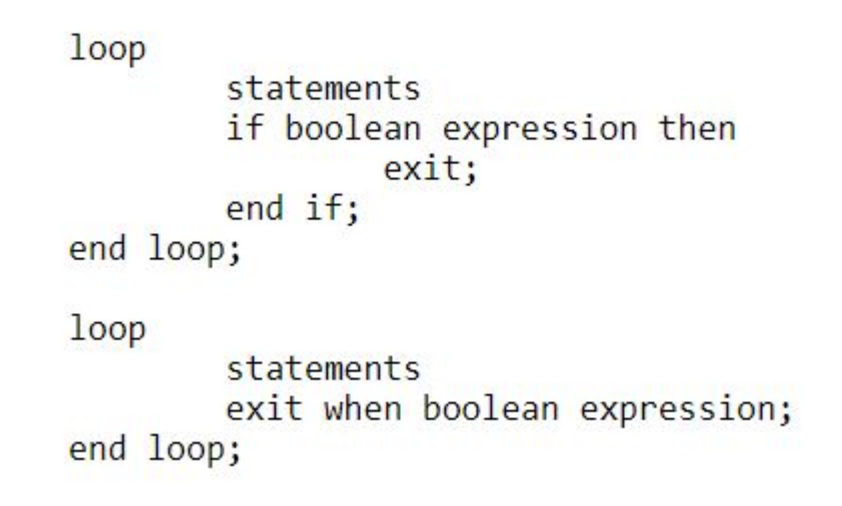
```
while boolean expression loop
       statements
end loop;
```

For Loops



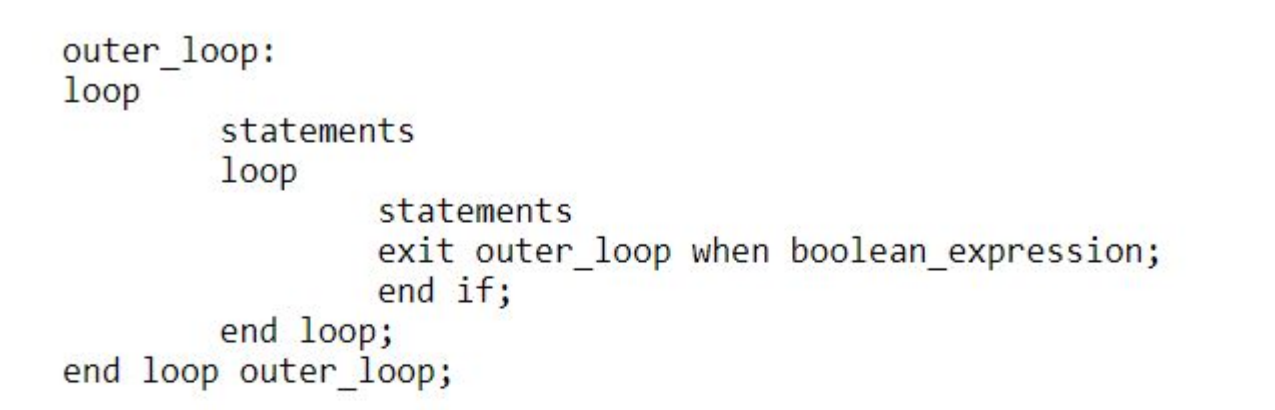
```
for index in range loop
        statements
end loop;


for  i in 1..20 loop
        put (i);
end loop;
```

Exit and exit when



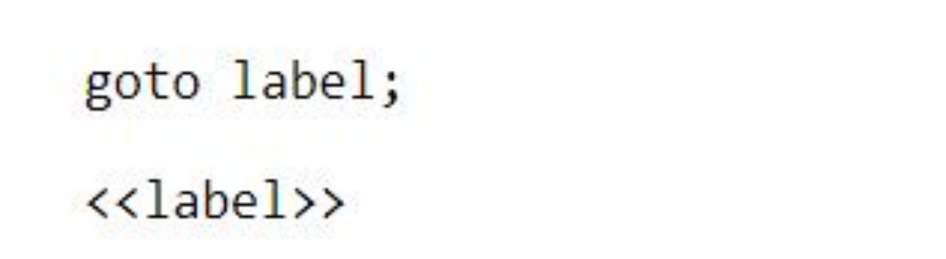
```
loop
        statements
        if boolean expression then
                exit;
        end if;
end loop;

loop
        statements
        exit when boolean expression;
end loop;
```

Labeled loops



```
outer_loop:
loop
        statements
        loop
                statements
                exit outer_loop when boolean_expression;
                end if;
        end loop;
end loop outer_loop;
```

Goto statement



```
goto label;

<<label>>
```

DEMO