

## Ada Basics

- ☞ Ada is case insensitive — **begin BEGIN Begin** are all the same.
- ☞ The use of ' the tick — the tick is used to access attributes for an object. For instance the following code is used to assign to value s the size in bits of an integer.

```
int a = sizeof(int) * 8;
```

```
a : Integer := Integer'Size;
```

### Operators

Operator	C/C++	Ada
Assignment	=	:=
Equality	==	=
NonEquality	!=	/=
PlusEquals	+=	
SubtractEquals	-=	
MultiplyEquals	*=	
DivisionEquals	/=	
OrEquals	=	
AndEquals	&=	
Modulus	%	mod
Remainder		rem
AbsoluteValue		abs
Exponentiation		**
Range		..

## C/C++ types to Ada types

Objects are defined in reverse order to C/C++, the object name is first, then the object type, as in C/C++ you can declare lists of objects by separating them with commas.

```
int i;  
int a, b, c;  
int j = 0;  
int k, l = 1;
```

```
i : Integer;  
a, b, c : Integer;  
j : Integer := 0;  
k, l : Integer := 1;
```

Another difference is in defining constants.

```
const int days_per_week = 7;
```

```
days_per_week : constant Integer := 7;  
days_per_week : constant := 7;
```

In the Ada example it is possible to define a constant without type, the compiler then chooses the most appropriate type to represent it.

## Declaring new types and subtypes

Ada is a strongly typed language, in fact possibly the strongest.

```
typedef int INT;  
INT a;  
int b;
```

```
a = b; // works, no problem
```

The compiler knows that they are both ints.

```
type INT is new Integer;  
a : INT;  
b : Integer;  
  
a := b; -- fails.
```

The important keyword is **new**, which really sums up the way Ada is treating that line, it can be read as "a new type INT has been created from the type Integer", whereas the C line may be interpreted as "a new name INT has been introduced as a synonym for int".

Ada also provides you with a feature for reducing the distance between the new type and its parent.

```
subtype INT is Integer;  
a : INT;  
b : Integer;  
  
a := b; -- works.
```

The most important feature of the subtype is to constrain the parent type in some way, for example to place an upper or lower boundary for an integer value.

## Simple types, Integers and Characters

Besides Integer type, there are a few more with Ada.

☞ Integer, Long\_Integer etc.

Any Ada compiler must provide the Integer type, this is a signed integer, and of implementation defined size. The compiler is also at liberty to provide Long\_Integer, Short\_Integer, Long\_Long\_Integer etc as needed.

☞ Unsigned Integers

Ada does not have a defined unsigned integer, so this can be synthesised by a range type, and Ada-95 has a defined package, System.Unsigned\_Types which provide such a set of types.

☞ Ada-95 has added a **modular** type which specifies the maximum value, and also the feature that arithmetic is cyclic (underflow/overflow cannot occur).

```
type BYTE is mod 256;  
type BYTE is mod 2**8;
```

☞ Character

This is very similar to the C char type, and holds the ASCII character set. However it is actually defined in the package Standard {A.I} as an enumerated type (see section 1.1.5). There is an Ada equivalent of the C set of functions in ctype.h which is the package Ada.Characters.Handling.

Ada Also defines a Wide\_Character type for handling non ASCII character sets.

☞ Boolean

This is also defined in the package Standard as an enumerated type (see below) as (FALSE, TRUE).

## Strings

- ☞ A predefined String type (defined again in Standard).
- ☞ A & operator for string concatenation.
- ☞ As in C the basis for the string is an array of characters.
- ☞ Strings as unbounded objects are not allowed.

```
type A_Record is
  record
    illegal : String;
    legal   : String(1 .. 20);
  end record;

procedure check(legal : in String);
```

Note that the lower bound of the size must be greater than or equal to 1, the C/C++ array[4] which defines a range 0 .. 3 cannot be used in Ada, 1 .. 4 must be used.

One way to specify the size is by initialisation, for example:

```
Name : String := "Simon";
```

is the same as defining Name as a String(1 .. 5) and assigning it the value "Simon" separately.

- ☞ For parameter types unconstrained types are allowed, similar to passing int array[] in C.

To overcome the constraint problem for strings Ada has a predefined package Ada.Strings.Unbounded which implements a variable length string type.

## Enumerations and Ranges

```
type Hours is new Integer range 1 .. 12;
type Hours24 is range 0 .. 23;
type Minutes is range 1 .. 60;
```

```
type Hours24 is new range 0 .. 23;
subtype Hours is Hours24 range 1 .. 12;
```

This limits the range even further, and as you might expect a subtype cannot extend the range beyond its parent, so range 0 .. 25 would have been illegal.

```
type All_Days is (Monday, Tuesday, Wednesday,
Thursday, Friday, Saturday, Sunday);
subtype Week_Days is All_Days range Monday ..
Friday;
subtype Weekend is All_Days range Saturday ..
Sunday;
```

We can now take a Day, and see if we want to go to work:

```
Day : All_Days := Today;

if Day in Week_Days then
  go_to_work;
end if;
```

Or you could use the form

```
if Day in range Monday .. Friday
```

and we would not need the extra types.

## Floating and Fixed point

- ☞ Two non-integer numeric types: the floating point and fixed point types.
- ☞ The predefined floating point type is Float and compilers may add Long\_Float, etc. A new Float type may be defined in one of two ways:

```
type FloatingPoint1 is new Float;
type FloatingPoint2 is digits 5;
```

The first simply makes a new floating point type, from the standard Float, with the precision and size of that type, regardless of what it is.

The second line defines a floating point type "of some kind" with a minimum of 5 digits of precision.

- ☞ You can get the number of digits which are actually used by the type by the attribute 'Digits.

```
num_of_dig : Integer := FloatingPoint2'Digits;
```

- ☞ Fixed point types are unusual, there is no predefined type 'Fixed' and such type must be declared in the long form:

```
type Fixed is delta 0.1 range -1.0 .. 1.0;
```

There is a specific form of fixed point types (added by Ada-95) called decimal types. These add a clause **digits**, and the **range** clause becomes optional.

```
type Decimal is delta 0.01 digits 10;
```

This specifies a fixed point type of 10 digits with two decimal places. The number of digits includes the decimal part and so the maximum range of values becomes -99,999,999.99 ... +99,999,999.99

## Attributes for enumeration type handling

(note these are used slightly differently than many other attributes as they are applied to the type, not the object)

- ☞ Succ

This attribute supplies the 'successor' to the current value, so the 'Succ value of an object containing Monday is Tuesday.

*Note:* If the value of the object is Sunday then an exception is raised, you cannot Succ past the end of the enumeration.

- ☞ Pred

This attribute provides the 'predecessor' of a given value, so the 'Pred value of an object containing Tuesday is Monday.

*Note:* the rule above still applies 'Pred of Monday' is an error.

- ☞ Val

This gives you the value (as a member of the enumeration) of element n in the enumeration. Thus Val(2) is Wednesday.

*Note:* the rule above still applies, and note also that 'Val(0)' is the same as 'First.

- ☞ Pos

This gives you the position in the enumeration of the given element name. Thus 'Pos(Wednesday) is 2.

*Note:* the range rules still apply, also that 'Last will work, and return Sunday.

```
All_Days'Succ(Monday) = Tuesday
All_Days'Pred(Tuesday) = Monday
All_Days'Val(0) = Monday
All_Days'First = Monday
All_Days'Val(2) = Wednesday
All_Days'Last = Sunday
All_Days'Succ(All_Days'Pred(Tuesday)) = Tuesday
```

## Attributes for range types

### ☞ First

This provides the value of the first item in a range. Considering the range 0 .. 100 then 'First is 0.

### ☞ Last

This provides the value of the last item in a range, and so considering above, 'Last is 100.

### ☞ Length

This provides the number of items in a range, so 'Length is actually 101.

### ☞ Range

## Arrays

Arrays in Ada make use of the range syntax to define their bounds and can be arrays of any type, and can even be declared as unknown size.

Some example:

```
char name[31];
int track[3];
int dbla[3][10];
int init[3] = { 0, 1, 2 };
typedef char[31] name_type;
track[2] = 1;
dbla[0][3] = 2;
```

```
Name : array (0 .. 30) of Character; -- OR
Name : String (1 .. 30);
Track : array (0 .. 2) of Integer;
DblA : array (0 .. 2) of array (0 .. 9) of
Integer; -- OR
DblA : array (0 .. 2, 0 .. 9) of Integer;
Init : array (0 .. 2) of Integer := (0, 1, 2);
type Name_Type is array (0 .. 30) of Character;
track(2) := 1;
dbla(0,3) := 2;

-- Note try this in C.
a, b : Name_Type;
a := b; -- will copy all elements of b into a.
```

## Arrays

2

### ☞ non-zero based ranges

Because Ada uses ranges to specify the bounds of an array then you can easily set the lower bound to anything you want, for example:

Example : **array** (-10 .. 10) **of** Integer;

### ☞ non-integer ranges

In the examples above we have used the common abbreviation for range specifiers. The ranges above are all integer ranges, and so we did not need to use the correct form which is:

```
array(type range low .. high)
```

which would make Example above:

```
array(Integer range -10 .. 10).
```

Taking an enumerated type, All\_Days and you can define:

```
Hours_Worked : array (All_Days range Monday ..
Friday);
```

### ☞ unbounded array types

Ada allows you to define unbounded array types. An unbounded type can be used as a parameter type, but you cannot simply define a variable of such a type.

```
type Vector is array (Integer range <>) of Float;
procedure sort_vector(sort_this : in out Vector);
Illegal_Variable : Vector;
Legal_Variable : Vector(1..5);
subtype SmallVector is Vector(0..1);
Another_Legal : SmallVector;
```

### ☞ array range attributes

Example : **array** (1 .. 10) **of** Integer;

```
for i in Example'First .. Example'Last loop
for i in Example'Range loop
```

Use the notation Array\_Name(dimension) 'attribute for multi-dimensional arrays.

## Arrays

3

### ☞ Initialisation by range (Aggregates)

When initialising an array one can initialise a range of elements in one go:

```
Init : array (0 .. 3) of Integer := (0 .. 3 => 1);
Init : array (0 .. 3) of Integer := (0 => 1,
others => 0);
```

The keyword **others** sets any elements not explicitly handled.

### ☞ Slicing

Array slicing is something usually done with memcpy in C/C++. Take a section out of one array and assign it into another.

```
Large : array (0 .. 100) of Integer;
Small : array (0 .. 3) of Integer;
```

```
-- extract section from one array into another.
Small(0 .. 3) := Large(10 .. 13);
```

```
-- swap top and bottom halves of an array.
Large := Large(51 .. 100) & Large(1..50);
```

*Note:* Both sides of the assignment must be of the same type, that is the same dimensions with each element the same. The following is illegal.

```
-- extract section from one array into another.
Small(0 .. 3) := Large(10 .. 33);
-- ^^^^^^^^ range too big.
```

## Records

☞ Almost direct mapping from C/C++ to Ada for simple structures.

```
struct _device {
    int    major_number;
    int    minor_number;
    char   name[20];
};
typedef struct _device Device;
```

```
type struct_device is
    record
        major_number : Integer;
        minor_number : Integer;
        name          : String(1 .. 19);
    end record;
type Device is new struct_device;
```

☞ Initialisation of record

```
Device lp1 = {1, 2, "lp1"};
lp1 : Device := (1, 2, "lp1");
lp2 : Device := (major_number => 1,
                 minor_number => 3,
                 name          => "lp2");
tmp : Device := (major_number => 255,
                 name          => "tmp");
```

When initialising a record we use an *aggregate*.

```
tmp : Device;
-- some processing
tmp := (major_number => 255, name => "tmp");
```

```
type struct_device is
    record
        major_number : Integer := 0;
        minor_number : Integer := 0;
        name          : String(1 .. 19) := "unknown";
    end record;
```

## Access types (pointers)

☞ In C/C++ the value of a pointer is the real memory address, in Ada it is not. It is a type used to access the data.

☞ The most common use of access types is in dynamic programming, for example in linked lists.

```
struct _device_event {
    int    major_number;
    int    minor_number;
    int    event_ident;

    struct _device_event* next;
};
```

```
type Device_Event;
type Device_Event_Access is access Device_Event;
type Device_Event is
    record
        major_number : Integer := 0;
        minor_number : Integer := 0;
        event_ident   : Integer := 0;

        next          : Device_Event_Access := null;
        -- Note:the assign. to null is not required,
        -- Ada initialises access types to
        -- null if no other value is specified.
    end record;
```

☞ Ada allocator syntax is much closer to C++ than to C.

```
Event_1 := new Device_Event;
Event_1.next := new Device_Event'(1, 2,
                                   EV_Paper_Low, null);
```

## Access types (pointers)

2

☞ The syntax, we can say directly that we want a new *thing*, none of this malloc rubbish.

☞ There is no difference in syntax between access of elements of a statically allocated record and a dynamically allocated one. We use the `record.element` syntax for both.

☞ We can initialise the values as we create the object, the tick is used again, not as an attribute, but with parentheses in order to form a qualified expression.

Ada allows you to assign between access types, and as you would expect it only changes what the access type points to, not the contents of what it points to. One thing to note again, Ada allows you to assign one structure to another if they are of the same type, and so a syntax is required to assign the contents of an access type, its easier to read than write, so:

```
dev1, dev2 : Device_Event;
pdv1, pdv2 : Device_Event_Access;

dev1 := dev2; -- all elements copied.
pdv1 := pdv2; -- pdv1 now points to contents of
pdv2.
pdv1.all := pdv2.all; -- !!
```

## C/C++ statements to Ada

*Note:* All Ada statements can be qualified by a name, this be discussed further in the section on Ada looping constructs, however it can be used anywhere to improve readability, for example:

```
begin
    Init_Code:
        begin
            Some_Code;
        end Init_Code;
    Main_Loop:
        loop
            if Some_Value then
                exit loop Main_Loop;
            end if;
        end loop Main_Loop;
    Term_Code:
        begin
            Some_Code;
        end Term_Code;
end A_Block;
```

## Compound Statement

A compound statement ( a block) and in C allows you to define variables local to that block. In Ada they must be declared as part of the block, but must appear in the declare part just before the block starts.

```
{
    declarations
    statements
}
```

```
declare
    declarations
begin
    statement
end;
```

## if Statement

If statements are the primary selection tool available to programmers. The Ada if statement also has the 'elsif' construct (which can be used more than once in any if statement), very useful for large complex selections where a switch/case statement is not possible.

*Note:* Ada does not require brackets around the expressions used in if, case or loop statements.

```
if (expression)
{
    statement
} else {

    statement
}
```

```
if expression then
    statement
elsif expression then
    statement
else
    statement
end if;
```

## switch Statement

The switch or case statement is a very useful tool where the number of possible values is large, and the selection expression is of a constant scalar type.

```
switch (expression)
{
    case value: statement
    default:    statement
}
```

```
case expression is
    when value => statement
    when others => statement
end case;
```

In C the end of the statement block between case statements is a break statement, otherwise we drop through into the next case. In Ada this does not happen, the end of the statement is the next case.

This leads to a slight problem, it is not uncommon to find a switch statement in C which looks like this:

```
switch (integer_value) {
case 1:
case 2:
case 3:
    value_ok = 1;
    break;
case 4:
case 5:
    break;
}
```

Ada also allows you to or values together:

```
case integer_value is
    when 1 .. 4    => value_ok := 1;
    when 5 | 6 | 7 => null;
end case;
```

## Ada loops

All Ada loops are built around the simple **loop ... end** construct

```
loop
    statement
end loop;
```

### while Loop

The while loop is common in code and has a very direct Ada equivalent.

```
while (expression)
{
    statement
}
```

```
while expression loop
    statement
end loop;
```

### do Loop

The do loop has no direct Ada equivalent.

```
do
{
    statement
} while (expression)

-- no direct Ada equivalent.
```

### for Loop

Ada has no direct equivalent to the C/C++ for loop (the most frighteningly overloaded statement in almost any language) but does allow you to iterate over a range, allowing you access to the most common usage of the for loop, iterating over an array.

```
for (init-statement; expression; loop-statement)
{
    statement
}
```

```
for ident in range loop
    statement
end loop;
```

However Ada adds some nice touches to this simple statement.

Firstly, the variable ident is actually declared by its appearance in the loop, it is a new variable which exists for the scope of the loop only and takes the correct type according to the specified range.

Secondly you will have noticed that to loop for 1 to 10 you can write the following Ada code:

```
for i in 1 .. 10 loop
    null;
end loop;
```

In Ada you cannot specify a range of 10 .. 1 as this is defined as a 'null range'. Passing a null range to a for loop causes it to exit immediately. The code to iterate over a null range such as this is:

```
for i in reverse 1 .. 10 loop
    null;
end loop;
```

## break and continue

In C and C++ we have two useful statements break and continue which may be used to add fine control to loops. Consider the following C code:

```
while (expression) {
    if (expression1) {
        continue;
    }
    if (expression2) {
        break;
    }
}
```

In Ada there is no continue, and break is now exit.

```
while expression loop
    if expression2 then
        exit;
    end if;
end loop;
```

The Ada exit statement however can combine the expression used to decide that it is required, and so the code below is often found.

```
while expression loop
    exit when expression2;
end loop;
```

This leads us onto the do loop, which can now be coded as:

```
loop
    statement
    exit when expression;
end loop;
```

## return

Here again a direct Ada equivalent, you want to return a value, then return a value,

```
return value; // C++ return
```

```
return value; -- Ada return
```

## labels and goto

Declare a label and jump to it.

```
label:
    goto label;
```

```
<<label>>
    goto label;
```

## Breaking nested loops

A useful feature which C and C++ lack is the ability to 'break' out of nested loops

```
while ((!feof(file_handle) && (!percent_found))){
    for (char_index=0; buffer[char_index]!='\n';
        char_index++) {
        if (buffer[char_index] == '%') {
            percent_found = 1;
            break;
        }
        // some other code, incl. get next line.
    }
}
```

Main Loop:

```
while not End_Of_File(File_Handle) loop
    for Char_Index in Buffer'Range loop
        exit when Buffer(Char_Index) = NEW_LINE;
        exit Main_Loop when Buffer(Char_Index) =
            PERCENT;
    end loop;
end loop Main_Loop;
```

## Exception handling

Exception handling consists of three components, the exception, raising the exception and handling the exception.

In C++ there is no exception type, when you raise an exception you pass out any sort of type, and selection of the exception is done on its type. In Ada there is a 'psuedo-type' for exceptions and they are then selected by name.

The code below shows the basic structure used to protect statement1, and execute statement2 on detection of the specified exception.

```
try {
    statement1
} catch (declaration) {
    statement2
}
```

```
begin
    statement1
exception
    when ident => statement2
    when others => statement2
end;
```

## Exception handling

2

Let us now consider an example, we will call a function which we know may raise a particular exception, but it may raise some we don't know about, so we must pass anything else back up to whoever called us.

```
try {
    function_call();
} catch (const char* string_exception) {
    if (!strcmp(string_exception,
                "the_one_we_want")) {
        handle_it();
    } else {
        throw;
    }
} catch (...) {
    throw;
}
```

```
begin
    function_call;
exception
    when the_one_we_want => handle_it;
    when others          => raise;
end;
```

This shows how much safer the Ada version is, we know exactly what we are waiting for and can immediately process it. In the C++ case all we know is that an exception of type 'const char\*' has been raised, we must then check it still further before we can handle it.

## Exception handling

3

You will also notice the similarity between the Ada exception catching code and the Ada case statement, this also extends to the fact that the when statement can catch multiple exceptions. Ranges of exceptions are not possible, however you can *or* exceptions, to get:

```
begin
    function_call;
exception
    when the_one_we_want |
        another_possibility => handle_it;
    when others            => raise;
end;
```

This also shows the basic form for raising an exception, the **throw** statement in C++ and the **raise** statement in Ada. Both normally raise a given exception, but both when invoked with no exception reraise the last one. To raise the exception above consider:

```
throw (const char*)"the_one_we_want";
```

```
raise the_one_we_want;
```

## Sub-programs

☞ Functions

```
return_type func_name(parameters);
return_type func_name(parameters)
{
    declarations
    statement
}
```

```
function func_name(parameters) return return_type;
function func_name(parameters) return return_type is
    declarations
begin
    statement
end func_name
```

☞ Procedures

```
void func_name(parameters);
```

```
procedure func_name(parameters);
```

## Argument passing

```
void func1(int by_value);
void func2(int* by_address);
void func3(int& by_reference); // C++ only.
```

```
type int      is new Integer;
type int_star is access int;
procedure func1(by_value      : in      int);
procedure func2(by_address    : in out int_star);
procedure func3(by_reference  : in out int);
```

A procedure or function which takes no parameters can be written in two ways in C/C++, though only one is Ada.

```
void func_name();
void func_name(void);
int func_name(void);
```

```
procedure func_name;
function func_name return Integer;
```

## Overloading

Ada allows more than one function/procedure with the same name as long as they can be uniquely identified by their signature (a combination of their parameter and return types).

```
function Day return All_Days;  
function Day(a_date : in Date_Type) return  
All_Days;
```

The first returns you the day of week, of today, the second the day of week from a given date. They are both allowed, and both visible. The compiler decides which one to use by looking at the types given to it when you call it.

## Operator overloading

As in C++ you can redefine the standard operators in Ada, unlike C++ you can do this outside a class, and for any operator, with any types. The syntax for this is to replace the name of the function (operators are always functions) with the operator name in quotes, ie:

```
function "+"(Left, Right : in Integer) return  
Integer;
```

Available operators are:

=	<	<=	>	>=
+	-	&	abs	not
*	/	mod	rem	**
and	or	xor		

## Default parameters

Ada (and C++) allow you to declare default values for parameters, this means that when you call the function you can leave such a parameter off the call as the compiler knows what value to use.

```
procedure Create  
  (File : in out File_Type;  
   Mode : in      File_Mode := Inout_File;  
   Name : in      String    := "");  
  Form : in      String    := "");
```

The simplest invocation of Create is

```
Create(File_Handle);
```

If we wish to provide the name of the file

```
Create(File_Handle, Inout_File, "text.file");
```

By using designators we could use the form:

```
Create(File => File_Handle,  
       Name => "text.file");
```

and we can leave the mode to pick up its default. This skipping of parameters is a uniquely Ada feature.

## Parameter passing modes

C++ allows three parameter passing modes, by value, by pointer and by reference (the default mode for Ada).

```
void func(int by_value, int* by_pointer, int&  
by_reference);
```

Ada provides two optional keywords to specify how parameters are passed, **in** and **out**. These are used like this:

```
procedure proc(Parameter : in      Integer);  
procedure proc(Parameter :      out Integer);  
procedure proc(Parameter : in out Integer);  
procedure proc(Parameter :      Integer);
```

If these keywords are used then the compiler can protect you even more, so if you have an **out** parameter it will warn you if you use it before it has been set, also it will warn you if you assign to an **in** parameter.

Note that you cannot mark parameters with **out** in functions as functions are used to return values, such *side affects* are disallowed.

## Nested procedures

You can define any number of procedures within the definition of another as long as they appear before the begin.

```
procedure Sort(Sort_This : in out An_Array) is  
  
  procedure Swap(Item_1, Item_2 : in out  
                                Array_Type) is  
  
    begin  
    end Swap;  
  
begin  
end Sort;
```

*Notes:* you can get in a mess with both C++ and Ada when mixing overloading and defaults. For example:

```
procedure increment(A_Value : A_Type);  
procedure increment  
  (A_Value : in out A_Type;  
   By      : in      Integer := 1);
```

If we call increment with one parameter which of the two above is called? Now the compiler will show such things up, but it does mean you have to think carefully and make sure you use defaults carefully.

---



## Ada Packages

Ada has one feature which many C/C++ programmers like to think they have an equivalent too — the package — they do not.

Any Ada package consists of two parts, the specification (header) and body (code). The specification however is a completely stand alone entity which can be compiled on its own and so must include specifications from other packages to do so. An Ada package body at compile time must refer to its package specification to ensure legal declarations, but in many Ada environments it would look up a compiled version of the specification.

Below is the skeleton of a package, spec and body.

```
--file example.ads, the package specification.
package example is
:
:
end example;
```

```
--file example.adb, the package body.
package body example is
:
:
end example;
```

## Include a package in another

Whereas a C file includes a header by simply inserting the text of the header into the current compilation stream with `#include "example.h"`, the Ada package specification has a two stage process.

```
-- Specification for package example
with Project_Specs;
package example is
    type My_Type is new Project_Spec.Their_Type;
end example;
```

```
-- Body for package example
with My_Specs;
package body example is
    type New_Type_1 is new My_Specs.Type_1;
    type New_Type_2 is new Project_Specs.Type_1;
end example;
```

To use an item, say a the type `Type_1` you must name it `My_Specs.Type_1`, in effect you have included the package name, not its contents. To get the same effect as the C `#include` you must also add another statement to make:

```
with My_Specs; use My_Specs;
package body example is
:
:
end example;
```

It is usual in Ada to put the *with* and the *use* on the same line, for clarity. There is a special form of the **use** statement which can simply include an element (types only) from a package, consider:

```
use type Ada.Calendar.Time;
```

## Package data hiding

Data encapsulation requires, for any level of safe reuse, a level of hiding. That is to say we need to defer the declaration of some data to a future point so that any client cannot depend on the structure of the data and allows the provider the ability to change that structure if the need arises.

In Ada this concept is formalised into the 'private part' of a package. This private part is used to define items which are forward declared as private.

```
package Our_List is
    type List_Rep is private;

    function Create return List_Rep;

private
    type List_Rep is
        record
            -- some data
        end record;
end Our_List;
```

As you can see the way the Ada private part is usually used the representation of `List_Rep` is exposed, but because it is a private type the only operations that the client may use are `=` and `/=`, all other operations must be provided by functions and procedures in the package. *Note:* we can even restrict use of `=` and `/=` by declaring the type as **limited private** when you wish to have no predefined operators available.

## Package data hiding

2

You may not in the public part of the package specification declare variables of the private type as the representation is not yet known, we can declare constants of the type, but you must declare them in both places, forward reference them in the public part with no value, and then again in the private part to provide a value:

```
package Example is
    type A is private;
    B : constant A;

private
    type A is new Integer;
    B : constant A := 0;
end Example;
```

```
package Our_List is
    type List_Access is limited private;

    function Create return List_Access;

private
    type List_Rep; -- opaque type
    type List_Access is access List_Rep;
end Our_List;
```

We now pass back to the client an access type, which points to a 'deferred incomplete type' whose representation is only required to be exposed in the package body.

## Hierarchical packages

Ada allows the nesting of packages within each other, this can be useful for a number of reasons. With Ada-83 this was possible by nesting package specs and bodies physically, thus:

```
package Outer is

    package Inner_1 is
    end Inner_1;

    package Inner_2 is
    end Inner_2;

private
end Outer;
```

Ada-95 has added to this the possibility to define child packages outside the physical scope of a package, thus:

```
package Outer is

    package Inner_1 is
    end Inner_1;

end Outer;
```

```
package Outer.Inner_2 is
end Outer.Inner_2;
```

As you can see `Inner_2` is still a child of `Outer` but can be created at some later date, by a different team.

## Renaming identifiers

This is not a package specific topic, and it is only introduced here as the using of packages is the most common place to find a `renames` clause.

Consider:

```
with Outer;
with Outer.Inner_1;
package New_Package is
    OI_1 renames Outer.Inner_1;

    type New_type is new OI_1.A_Type;
end New_Package;
```

The use of `OI_1` not only saves us a lot of typing, but if `Outer` were the package `Sorting_Algorithms`, and `Inner_1` was `Insertion_Sort`, then we could have `Sort renames Sorting_Algorithms.Insertion_Sort` and then at some later date if you decide that a quick sort is more appropriate then you simply change the `renames` clause, and the rest of the package spec stays exactly the same.

Similarly if you want to include 2 functions from two different package with the same name then, rather than relying on overloading, or to clarify your code text you could:

```
with Package1;
function Function1 return Integer renames
Package1.Function;
with Package2;
function Function2 return Integer renames
Package2.Function;
```

## Concurrency

Most modern operating systems contain constructs known either as *lightweight processes* or as *threads*. These allow programmers to have multiple threads of execution within the same address space.

Unlike C/C++ Ada defines a concurrency model as part of the language itself. Some languages (Modula-3) provide a concurrency model through the use of standard library packages, and of course some operating systems provide libraries to provide concurrency. In Ada there are two base components, the task which encapsulates a concurrent process and the protected type which is a data structure which provides guarded access to its data.

### Tasks as threads

For those who have not worked in a multi-threaded environment you might like to consider the advantages. In a non-multi-threaded UNIX (for example) the granularity of concurrency is the process. This process is an atomic entity to communicate with other processes you must use sockets, IPC etc. The only way to start a cooperating process is to initialise some global data and use the `fork` function to start a process which is a copy of the current process and so inherits these global variables. The problem with this model is that the global variables are now replicated in both processes, a change to one is not reflected in the other.

In a multi-threaded environment multiple concurrent processes are allowed within the same address space, that is they can share global data. Usually there are a set of API calls such as `StartThread`, `StopThread` etc which manage these processes.

*Note:* An Ada program with no tasks is really an Ada process with a single running task, the default code.

### Simple task

In the example below an Ada task is presented which will act like a thread found in a multi-threaded operating system such as OS/2, Windows-NT or Solaris.

```
task X is
end X;

task body X is
begin
    loop
        -- processing.
    end loop;
end X;
```

As with packages a task comes in two blocks, the specification and the body. Both of these are shown above, the task specification simply declares the name of the task and nothing more. The body of the task shows that it is a loop processing something. In many cases a task is simply a straight through block of code which is executed in parallel, or it may be, as in this case, modelled as a service loop.

### Task as types

Tasks can be defined as types, this means that you can define a task which can be used by any client. Once defined as a task objects of that type can be created in the usual way. Consider:

```
task type X is
end X;
```

```
Item : X;
Items : array (0 .. 9) of X;
```

*Note:* however that tasks are declared as constants, you cannot assign to them and you cannot test for equality.