Advanced Ada

"*C was designed to be written; Ada was designed to be read.*"
Jean Ichbiah

This unit will cover the following topics:

1. Exceptions
2. Simple Tasking
3. Overloading
4. Files
5. goto
6. Generics
7. Records

1. Exceptions

During the execution of a program an unusual or exceptional circumstance may arise. This may be due to some logical error in the algorithm. When this situation arises in Ada, we say "an exception is raised". This can be dealt with with an explicit raise statement, for example:

```
raise constraint_error;
```

The exceptions **constraint_error**, **numeric_error**, **program_error**, and **storage_error** are part of Ada.

1.1 Predefined exceptions

A **constraint_error** is concerned with attempts to violate a range of index constraint at run-time. For example:

```
list : array(1 .. 20) of integer;
```

and an attempt is made to access the component list(21) then a constraint_error exception will be raised. Another example is divide-by-zero:

```
    x, y : float := 0.0;
begin
    ...
    x := 1.0;
    x := x / y;
    ...
end;
```

Here an exception will be raised when an attempt is made to divide by zero.

numeric_error - when a numeric operation cannot give a correct result. e.g div by 0
program_error - attempt to leave a function other than by a return statement.
storage_error - an infinite series of recursive calls that eventually runs out of storage space will raise this.

1.2 Making exceptions

It is possible to create an exception variable:

```
function percentage(a, b: integer) return float is
    zero_divide : exception;
begin
    if b = 0 then
```

```
        raise zero_divide;
    else
        return (float(a)/float(b)*100.0);
    end if;
end percentage;
```

1.3 Handling exceptions

We can recover from the system or let the system handle the exception for itself. In the case of the percentage function, it can be modified to include its own exception handler:

```
function percentage(a, b: integer) return float is
    zero_divide : exception;
begin
    if b = 0 then
        raise zero_divide;
    else
        return (float(a)/float(b)*100.0);
    end if;
    exception
        when zero_divide =>
            put("percent: Attempt to divide by zero");
            return 0.0;
end percentage;
```

When the divide-by-zero condition is detected (y=0), the *zero-divide* exception is raised. Raising the exception causes execution to be suspended, and a search is made for the handler. If there is, execution is continued with the named handler, otherwise it is "propagated" forward.

For example:

```
subtype small_pos is positive range 1 .. 15;
```

Two possible errors can occur when a data item is read: the item may not be in the correct range, or it may not be a whole number. -> **data_error** exception will be raised.

The following procedure contains a **data_error** exception handler within a loop. This allows further attempts to be made if an initial attempt to read a small positive integer fails.

```
procedure get_small(num : out small_pos) is
begin
    loop
```

```
            begin -- inner frame
                get(num);
                return;
            exception
                when data_error =>
                  skip_line;
                  put_line("small positive integer expected");
            end;  -- inner frame
            put_line("try again");
        end loop;
    end get_small;
```

The **data_error** exception handler is associated with the two statements:

```
    get(num);
    return;
```

which occur between the reserved words begin and exception in the inner frame. When **get_small** is executed, we enter the loop, and then the inner frame. The statement **get (num)** is executed and if successful the return statement is executed and the procedure exited. The loop and exception handler have had no effect.

If the statement is not handled correctly, due to an attempt to read erroneous data, a **data_error** exception is raised. When this happens, normal execution of the statements in the inner frame is abandoned, and control is transferred to the exception handler. After the statements in the exception handler have been executed we leave the frame. The **put_line** following the frame is executed and control is transferred back to the beginning of the loop.

Here is an example of the procedure in use:

```
with ada.text_io; use ada.text_io;
with ada.integer_text_io; use ada.integer_text_io;
procedure small_excp is
    subtype small_pos is positive range 1..15;
    small : small_pos;

    procedure get_small(num : out small_pos) is
    begin
        loop
            begin
                get(num);
                return;
            exception
                when data_error =>
                    skip_line;
                    put_line("small positive integer expected");
```

```
            end;
            put_line("try again");
        end loop;
    end get_small;

begin
    get_small(small);
    put(small);
end small_excp;
```

Running the program with an input of **19** gives:

```
raised CONSTRAINT_ERROR : small_excp.adb:11 range check failed
```

When the stack handling procedures **push** and **pop** were created, we checked that no attempt was made to add an item to a full stack or remove an item from an empty stack. Instead of explicitly checking for these possibilities, we could use a constraint_error exception.

```
procedure pop(x : out character; st : in out stack) is
begin
    x := st.item(st.top);
    st.top = st.top - 1;
exception
  when constraint_error = >
    put_line("stack is empty");
    raise constraint_error;
end pop;
```

2. Simple Tasking

2.1 What are tasks?

Use tasking when you need some form of parallel operation. Unless the system is set up for multiple hardware processors, tasking occurs on only one processor, which means it's not true parallel processing. Tasking with one processor, shares the processor. Ada tasks have a number of states:

- running
- ready - unblocked waiting for a processor
- blocked - delayed or waiting to rendezvous
- completed - at the 'end'
- terminated - no longer active

2.2 Task syntax

Ada provides light-weight *tasks*, which are referred to as *threads* in some other languages. A task is composed of a *task specification* of the form:

```
task T is
    <entry_points>;
end T;
```

where **T** is the name of the task. If there are no entry points, then the specification can be shortened to:

```
task T;
```

Secondly, a *task body* is required, of the form:

```
task body T is
begin
...
end T;
```

Tasks may be declared at any program level. For example:

```
program task_example is
    task type A_type;
    task B;
    A,C: A_type;

    task body A_type is
```

```
      -- local declarations for task A and C
    begin
      -- statements for task A and C
    end A_Type;

    task body B is
      -- local declarations for task B
    begin
      -- statements for task B
    end B;

begin
   -- task A, C, and B start their execution
end task_example;
```

Here the task **A_type** has been declared as a *task type*, thereby allowing task units to be created dynamically, and incorporated into more complex structures.

2.3 A Simple Task

Consider the following simple task:

```
with ada.text_io; use ada.text_io;
procedure simpleTasks is
    task type Simple(message: character; howmany: integer);

    task body Simple is
    begin
        for count in 1..howmany loop
            put("hello from task " &message);
            new_line;
        end loop;
    end Simple;

    Task_A: Simple('A', 4);
    Task_B: Simple('B', 2);

begin -- simpleTasks
    null;
end simpleTasks;
```

This program will invoke either task A or task B, as soon as control reaches the start of the procedure **simpleTasks**, before any of the main programs statements are executed. Ada does not specify which task will start first.

3. Overloading

Ada allows operator overloading. This is helpful because Ada doesn't like mixed-type arithmetic. For example, the "+" operator has the following predefined specifications:

```
function "+"(x,y: integer) return integer;
function "+"(x,y: float) return float;
```

To allow mixed type arithmetic requires the "+" to be overloaded. For example:

```
function "+"(x: integer; y: float) return float is
begin
    return float(x) + y;
end

function "+"(x: float; y: integer) return float is
begin
    return x + float(y);
end
```

Now these can be used as one would normally use the "+" operator. For example:

```
i: integer := 6;
f: float := 4.2;

put (i + f);          => 8.7
put (f + i);          => 8.7
```

This can be extremely useful for user-defined types. For example overloading the "+" operator to add arrays together. Consider the following user-defined type:

```
type vector is array(integer range<>) of integer;
a,b: vector(1..4);
```

Normally to add **a** and **b** together would require the use of a loop. It is much more convenient to overload the "+" operator. For example:

```
function "+"(x,y: vector) return vector is
    z: vector(x'range);
begin
    for i in z'range loop
        z(i) := x(i) + y(i);
    end loop;
    return z;
end "+";
```

Now the following operation can be performed:

```
a := (2,4,4,3);
b := (3,9,1,7);
c := a + b;
```

All the predefined arithmetic operators can be overloaded, as can the relational operators <, <=, >, >=. The operator /= is implicitly overloaded when = is overloaded, however = should likely never be overloaded.

4. Files

4.1 Introduction to Files

There are two kinds of files in Ada: *internal* and *external*. An internal file is a declared object which has a name and a file-type associated with it. The following are file-types in Ada:

```
in_file, out_file, append_file
```

These provide read, write and read/write facilities respectively. An external file is where the information actually resides. Once the packages have been instantiated, the file "pointers" can be defined:

```
infp : file_type;
outfp : file_type;
```

4.2 File Open and Close

To read from a file requires the use of the **open** function.

```
open(infp,in_file,"data.txt");
```

The clause **in_file** specifies that the file is to be used for input, and **fp** is associated with the physical file "**data.txt**". The clause **out_file** can be used to specify a file for output. To close the file, use the function close:

```
close(infp);
```

4.3 File creation

A call to the function **create** will cause a new external file to be created, and a link to be created to an internal file. This example shows how a file can be created using the function **create**:

```
create(outfp, out_file, "results.out");
```

The function **is_open** can be used to check to see if a file is open:

```
if is_open(outfp) then
    close(outfp);
end if;
```

4.4 File get and put

The functions **get**, **get_line** and **put**, **put_line, and new_line** can be used in the same ways as they are for standard I/O. For example for integers:

```
get(fp, num);
get(fp, num, n);
put(fp, num, n, base);
```

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

```
get(fp, num);
put(fp, num, fore, aft, exp);
```

where **num** is the variable to be input/output, **fore** is the number of places in front of the decimal point, **aft** is the number of places after the decimal point, and **exp** the number of places for the exponent.

4.5 File exceptions

Exceptions are raised if any of these file handling procedures are used incorrectly.

- **status_error** - raised if the internal file is already open (for open or create).
- **name_error** - raised if there is a problem creating a file, or opening a file that does not exist.
- **use_error** - raised if an attempt is made to open, create or delete a file for which there are no permissions.
- **mode_error** - raised if we try to read from a file which is not in **in_file** mode, or write to a file which is not in **out_file** mode.

4.6 External files

There are two further functions used to deal with external files: **name** and **delete**. The function **name** returns a string representing the external file. The function **delete**, removes an external file. A call to delete does not guarantee that the named file is immediately deleted, only that no further opens can be performed on the file. A **name_error** is raised if for some reason the file cannot be deleted.

4.7 Redirection

Output can be re-directed using the **set_output** function. For example:

```
set_output(outfp);
```

makes outfp the default output, so file pointers are not needed. To return to the standard I/O use:

```
set_output(standard_output);
```

4.8 Example with other file functions

The following example involves a procedure which reads in a series of integers from a file:

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

procedure fileio is
    infp : file_type;
    i, sum : integer;

begin
    sum := 0;
    open(infp,in_file,"rainfall.dat");
    loop
        exit when end_of_file(infp);
        get(infp,i);
        sum := sum + i;
        put(sum); new_line;
    end loop;

    close(infp);
end fileio;
```

The loop inside the program uses an **exit when** clause to break out of the loop when the *end-of-file* is encountered, which is tested using the function **end_of_file**. To reset a file back to the first character in the file, the **reset** function can be used:

```
reset(infp);
```

To check for end-of-line and end-of-page, **end_of_line** and **end_of_page** can be used respectively.

4.9 String I/O

The following example involves a procedure which reads in a series of strings from a file:

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

```
procedure stringio is
    infp : file_type;
    str : string(1..10);

begin
    open(infp,in_file,"fox.txt");
    loop
        exit when end_of_file(infp);
        get(infp,str);
        put(str); new_line;
    end loop;

    close(infp);
end stringio;
```

This program works perfectly if there are exact multiples of 10 characters in the file, the string being 10 characters in length. However were this not the case, for example if the input were:

```
The quick brown fox jumped over the lazy dog
```

Then an exception would be raised, of the form:

```
The quick
brown fox
jumped ove
r the lazy

raised ADA.IO_EXCEPTIONS.END_ERROR : a-textio.adb:514
```

The same is output if the file is organized in the following manner (**fox2.txt**):

```
The
quick
brown
fox
jumped
over
the
lazy
dog
```

This can be remedied using unbounded strings. For example:

```
with ada.Text_IO; use Ada.Text_IO;
with ada.Integer_Text_IO; use Ada.Integer_Text_IO;
with ada.strings.unbounded; use ada.strings.unbounded;
with ada.strings.unbounded.Text_IO; use ada.strings.unbounded.Text_IO;

procedure vstringio is

    s : unbounded_string;
    infp : file_type;

begin
    open(infp,in_file,"fox2.txt");
    loop
        exit when end_of_file(infp);
        get_line(infp,s);
        put(s);
        new_line;
    end loop;

    close(infp);

end vstringio;
```

This processes the file **fox2.txt**, which has each word on a separate line. An unbounded string, **s**, is used to read data from the file.

5. goto

Every language has its hidden "features", and Ada, like most other languages has a **goto** statement. Any statement could be labelled by an identifier enclosed in double angle brackets, << >>. For example:

```
<<gohere>> g = 12.3;
...
goto gohere;
```

However, the goto in Ada is somewhat better behaved. It cannot transfer control outside the current subprogram or package body; it cannot transfer control inside a structure (e.g. from **else** to **then** in an **if** statement); and it cannot transfer control from the outside of a structured statement into the body of a structured statement. Hence the following is not permitted:

```
if denom < 0 then
    result := 0;
    <<here>>
    put_line("error");
end if;
goto here;
```

6. Generics

Generics are used when the the same logical function is to be applied to a multiplicity of differing data types.

6.1 Generic subprograms

A generic subprogram is introduced via a generic subprogram declaration. For example, consider the following generic version of "swap".

```
generic
    type object_type is private;
    procedure swap0(a,b: in out object_type);
```

which is followed by a generic body:

```
procedure swap0(a,b: in out object_type) is
    temp : object_type;
begin
    temp := a;
    a := b;
    b := temp;
end swap0;
```

Now we can create a generic instantiation for the enumeration type '*day*':

```
procedure swapday0 is new swap0(day);
```

and an example of how it can be used:

```
type day is (sun,mon,tue,wed,thu,fri,sat);
x, y : day;
begin
    swapday0(x,y);
end
```

6.2 Generic packages

These are similar in concept to generic subprograms. It can be illustrated by modifying the stack package reviewed previously.

```
generic
    stack_size : integer range 2..integer'last;
    type object is private;
package stack is
    procedure push(x : in object);
    procedure pop(x : out object);
    function stack_is_empty return Boolean;
    function stack_top return object;
    procedure reset_stack;
end stack;

package body stack is
    type list is array(1 .. stack_size) of object;
    type ob_stack is
        record
            item : list;
            top : natural := 0;
        end record;
    st : ob_stack;

    procedure push(x : in object) is
    begin
        if s.top = 100 then
            put_line("stack is full");
        else
            st.top = st.top + 1;
            st.item(st.top) := x;
        end if;
    end push;

    procedure pop(x : out object) is
    begin
        if s.top = 0 then
            put_line("stack is empty");
        else
            x := st.item(st.top);
            st.top = st.top - 1;
        end if;
    end pop;
```

```
    function stack_is_empty return Boolean is
    begin
        return st.top = 0;
    end stack_is_empty;

    function stack_top return object is
    begin
        if st.top = 0 then
            put_line("stack is empty");
            return '';
        else
            return st.item(st.top);
        end if;
    end stack_top;

    procedure reset_stack is
        st.top := 0;
    end reset_stack;

end stack;
```

Now we can create two instances of the package:

```
package stackC is new stack(100,character);
use stackC;
a, b : character;

begin
    push(a);
    pop(b);
end;

package stackI is new stack(100,integer);
use stackI;
a, b : integer;

begin
    push(a);
    pop(b);
end;
```

7. Structured data: Records

7.1 Simple records

Records are composite structures composed of objects with differing types. For example:

```
type manufacturer is (Stanley, Sargent, MillersFalls);

type plane is
    record
        planetype : string(1..40 => ‘ ‘);
        maker     : manufacturer;
        year      : integer range 1800..2000;
        model     : string(1..40 => ‘ ‘);
    end record;
```

Now create a new object:

```
new_plane : plane;
```

and propagate it with data:

```
new_plane.planetype := “block”;
new_plane.maker := Sargent;
new_plane.year := 1897;
new_plane.model := “5063”;
```

or

```
new_plane := (“block”,”Sargent”,1897,”5063”);
```

7.2 Dynamic records

To generate records of a dynamic size, Ada provides *record discriminants*. For example:

```
type catalog(size : integer) is
    record
        items : array(1..size) of plane;
    end record;
```

Now create a new object, containing 100 different planes:

```
catalog59 : catalog(100);
```

7.3 Variant records

Ada allows the components of a record to vary. For example:

```
type planetype is (block, smoother, jack, joiner);

type plane (ptype : planetype) is
    record
        maker     : manufacturer;
        year      : integer range 1800..2000;
        modelno   : string(1..40 => ' ');
        length    : float;
        width     : float;
        case ptype is
            when block => adjmth : boolean;
            when others => null;
        end case;
    end record;
```

The record type will have a differing number of components depending on the **ptype** parameter. Now an object whose **ptype** is "block" can be created in the following manner:

```
p1 : plane(block);

p1 := (Sargent, 1910, "6402", 7.5, 1.8, true);
```