Introduction to Systematic Programming
Unit 14 - Subtypes; Other Forms of Repetition

14.1 Subtypes

Sometimes we want to use variables which cannot possibly take on all the values defined for one of the pre-defined or enumeration types. For example, suppose we wished to process the examination and coursework marks of a student, where the examination mark should be a value between 0 and 100 (inclusive) and the coursework mark a value between 0 and 20. How do we specify that a variable `ExamMark` (say) can only hold an integer value between 0 and 100, and that a variable `CwkMark` (say) can only hold an integer value between 0 and 20. We need these two variables have all the normal properties of `Integer` variables (eg. be able to add them together using the operator +, or be able to input and output their values using the `Get` and `Put` procedures from `CS_Int_IO`, etc.), but we can't simply declare them to be of type `Integer`, since they could then hold any `Integer` value. The answer here is not to declare `Integer` variables, but of a **subtype** of the type `Integer`. We could define suitable subtypes as follows:

```
SUBTYPE Percentage IS Integer RANGE 0..100;
SUBTYPE OutOf20    IS Integer RANGE 0..20;
```

which defines two new subtypes called `Percentage` and `OutOf20`. We can see from the above **subtype declaration** that subtypes have a **range constraint** attached to the **base type** (from which they are derived) which limits the range of values that can be stored in variables of these subtypes. We could then declare suitable variables as follows:

```
ExamMark : Percentage;
CwkMark  : OutOf20;
```

which declares `ExamMark` to be a variable of the subtype `Percentage`, ie. capable of holding only those integer values in the range `0` to `100`, and `CwkMark` to be a variable of subtype `OutOf20`, ie. capable of holding only those integer values in the range `0` to `20`. The variables `ExamMark` and `CwkMark` are often referred to as **integer subtype** (or **subrange**) **variables** since the values they are permitted to hold form a subrange of all the possible integer values.

The general form of a subtype declaration with a range constraint is as follows:

SUBTYPE *Subtype_name* IS *Type_name* RANGE *First..Last*;

where the *Type_name* is the name of a discrete type (ie. `Integer`, `Character`, `Boolean` or an enumeration type), and where *First..Last* indicates the range of consecutive values (from the ordered list of values) of the discrete type which are permitted in the subtype.

Since subtypes are derived from an existing base type, subtype variables 'inherit' all the properties of the base type. This means that all the previously defined operations for the base type are also available for manipulating subtype variables. For example we could write:

a)	`Get(Item => CwkMark);`	since the procedure `Get` is defined for an `Integer` AP
b)	`Total := ExamMark + CwkMark + 10;`	since the operator + is defined for operands of type `Integer`
c)	`IF ExamMark < CwkMark THEN`	since the relational operator < is defined for `Integer` operands
d)	`Ratio := Float(CwkMark) / Float(ExamMark);`	since the type converter `Float` is defined for `Integer` values
e)	`Put(Item => ExamMark, Width => 8);`	since the procedure `Put` is defined for an `Integer` AP

and since `Integer` is the base type, from which these subrange variables have been derived.

The only restriction is that a subtype variable may only contain a value which lies in the specified range, and attempting to assign a value outside that specified range to a subtype variable will result in a run-time error.

Why do we want to use subtype variables? One reason for using subtype variables is to *improve the clarity of the program*; when reading a program it is plain from the declarations of the subtype variables what values the program is designed to process. The use of subtype variables also provides *an additional check on the correctness of our programs and/or data.* For the above example, our algorithm might only work correctly for input values of `ExamMark` from 0 to 100. Thus it would be reasonable to prevent anyone who uses the program from attempting to execute it with values for `ExamMark` which are outside this range. One way to do this is to declare `ExamMark` as an `Integer` variable (capable of holding any integer value), and then test the value of `ExamMark` which is read in before using this value in the algorithm, eg:

```
ExamMark : Integer;
...................
Get(ExamMark);
IF ExamMark < 0 OR ExamMark > 100 THEN
   Put("Incorrect value input");
ELSE .......
```

Alternatively, we could declare `ExamMark` as a subtype variable, and then just write instead:

```
ExamMark : Percentage;
...................
Get(ExamMark);
...................
```

The computer will then do all the checking for us, since an error will be reported if the user attempts to input (into the variable `ExamMark`) a value which outside the specified range.

14.1.1 What types can subtypes be derived from?

It is possible to have subtypes of any of the discrete types. We have seen examples of subtypes of the base type `Integer` above, so let us consider some examples of variables of `Character` subtypes:

```
SUBTYPE DigitChar     IS Character RANGE '0'..'9';
SUBTYPE CapitalLetter IS Character RANGE 'A'..'Z';
SUBTYPE GradeLetter   IS Character RANGE 'A'..'F';
```

Following these subtype declarations we could declare the variables:

```
Digit     : DigitChar;
Initial   : CapitalLetter;
ExamGrade : GradeLetter;
```

Here the specified range of values which define the subtypes are defined according to the ordering of the consecutive values in the Latin-1 character set. Thus the variable `Digit` is capable of holding one of the character values `'0'`, `'1'`, ...,`'9'`, the variable `Initial` one of the values `'A'`, `'B'`, ...,`'Z'`, and the variable `ExamGrade` one of the values `'A'`, `'B'`, `'C'`, `'D'`, `'E'` or `'F'`.

The other important types from which it is possible to derive subtypes are enumeration types. For example, if we defined an enumeration type `DayOfWeek` (as it was defined in [13.1]), it would then be possible to declare the subtypes:

```
TYPE DayOfWeek IS (Mon, Tues, Wed, Thurs, Fri, Sat, Sun);
SUBTYPE WeekDay    IS DayOfWeek RANGE Mon..Fri;
SUBTYPE WorkingDay IS DayOfWeek RANGE Mon..Sat;
```

Hence the declarations:

```
Today   : WeekDay;
WorkDay : WorkingDay;
```

would define a variable `Today` capable of holding one of the enumeration values `Mon`, `Tues`, `Wed`, `Thurs` or `Fri`, and a variable `WorkDay` capable of holding one of the values `Mon`, `Tues`, `Wed`, `Thurs`, `Fri` or `Sat`.

14.2 Pre-defined subtypes

There are in fact two pre-defined integer subtypes in Ada which are often useful, these are the subtypes `Positive` and `Natural`. The first defines positive integer values which are greater than or equal to 1, the second defines integer values which are greater than or equal to 0. Thus if we are using a counting variable `Number` (say), to count over the values 0, 1, 2, ..., etc, rather than declare it of type `Integer`, it would be more meaningful to declare it (and initialise its value) using:

```
Number : Natural := 0;
```

Alternatively, if we were to count over the values 1, 2, 3, ..., etc, it would be more meaningful to declare it (and initialise its value) using:

```
Number : Positive := 1;
```

14.2.1 Subtypes of subtypes

It is possible in Ada to have subtypes which are derived from other subtypes. So for example we could have defined the subtype `Percentage` using:

```
SUBTYPE Percentage IS Natural RANGE 0..100;
```

or the subtypes `WorkingDay` and `WeekDay` using:

```
SUBTYPE WorkingDay IS DayOfWeek  RANGE Mon..Sat;
SUBTYPE WeekDay    IS WorkingDay RANGE Mon..Fri;
```

14.3 Procedure parameters and subtypes

What happens when we use subtypes as parameters in a procedure? It is possible to have subtypes as formal and actual parameters, provided that the corresponding FP's and AP's are *of the same base type*. However, a run-time error will occur if there is any attempt to assign an 'out of range' value to:

i) a subtype FP on entry to the procedure (as the AP value is copied into the FP), or
ii) a subtype AP on exit from the procedure (as the FP value is copied into the AP).

An example of (i) would be if we were to define a procedure with subtype formal parameters as follows:

```
PROCEDURE DetermineGrade (Exam : IN Percentage;
                          Grade : OUT GradeLetter) IS
    -- To determine the appropriate Grade letter for
    -- this percentage Exam mark.
BEGIN
    CASE Exam IS
        WHEN 0..39   => Grade := 'F';
        WHEN 40..44  => Grade := 'E';
        WHEN 45..49  => Grade := 'D';
        WHEN 50..59  => Grade := 'C';
        WHEN 60..69  => Grade := 'B';
        WHEN 70..100 => Grade := 'A';
    END CASE;
END DetermineGrade;
```

and then call the procedure with actual parameters which are variables of the base types, eg:

```
DetermineGrade (Exam => TotalMark, Grade => Level);
```

where `TotalMark` is a variable of type `Integer` (holding some suitable value), and `Level` is a variable of type `Character`. Thus a run-time error will occur on entry to the procedure if there is an attempt to initialise the subtype FP `Exam` with a value of `TotalMark` outside the range 0 to 100. Alternatively, if `TotalMark` holds a value in the range 0 to 100, this value can be used to initialise the subtype FP `Exam`, which can then be used inside the procedure body. The steps in the procedure body will then assign a `Character` value in the range `'A'` to `'F'` to the subtype FP `Grade`, whose value will then be copied to the `Character` AP `Level` on exit.

An example of (ii) would be if we were to use the 'swap' procedure:

```
PROCEDURE Swap (First, Second : IN OUT Integer) IS
    -- To swap the values in First and Second.
    Temp : Integer;
BEGIN
    Temp   := First;
    First  := Second;
    Second := Temp;
END Swap;
```

with a call which includes actual parameters which are subtype variables, eg:

```
Swap (First => ExamMark, Second => CwkMark);
```

On entry to the procedure the values of the two AP's `ExamMark` and `CwkMark` are copied into the two FP's `First` and `Second`, which are then manipulated by the steps of the procedure body. On exit from the procedure, the computer attempts to copy the values of the two FP's `First` and `Second` back into the two AP's `ExamMark` and `CwkMark`. If `Second` holds a value outside the range 0 to 20, a run-time error will occur. Alternatively, if `Second` holds a value in the range 0 to 20, this value is copied into `CwkMark`. We may note that `First` will hold a value in the range 0 to 100 (since its value was previously stored in `CwkMark`, and thus must be in the range 0 to 20), and hence its value will be copied into `ExamMark` without the possibility of an error.

14.4 Subtype attributes

A subtype 'inherits' all the attribute functions defined for its base type, thus:

`GradeLetter'Succ('C')`	yields the value `'D'`
`GradeLetter'Pred('B')`	yields the value `'A'`
`GradeLetter'First`	yields the value `'A'` (the first value of the subtype)
`GradeLetter'Last`	yields the value `'F'` (the last value of the subtype)

However:

`GradeLetter'Pos('A')`	yields the value `65`
`GradeLetter'Val(66)`	yields the value `'B'`

since the `Pos` and `Val` functions refer to *positions in the base type* (ie. the underlying internal numerical codes for the type `Character` in these examples).

14.5 Using a type or subtype name

Since subtypes 'inherit' all the operations permitted for the base types, we would expect to be able to subscript arrays, and be able to 'count' in a `FOR` repetition using a subtype, and this is indeed the case. In order to make the meaning of these two features even clearer, Ada allows us to write a type or subtype name instead of expressing a range of values using *`First..Last`*. Thus for example, instead of defining array types using:

```
TYPE LetterArray   IS ARRAY ('A'..'Z') OF Integer;
TYPE SunsetTime    IS ARRAY (Mon..Sun) OF Integer;
TYPE WorkedPerWeek IS ARRAY (Mon..Fri) OF Float;
```

we could have written the equivalent declarations:

```
TYPE LetterArray   IS ARRAY (CapitalLetter) OF Integer;
TYPE SunsetTime    IS ARRAY (DayOfWeek) OF Integer;
TYPE WorkedPerWeek IS ARRAY (WeekDay) OF Float;
```

where in the second of these we use a type name `DayOfWeek`, and in the first and third we use the subtype names `CapitalLetter` and `WeekDay` to define the subscript range of the array types. Similarly, instead of writing `FOR` repetitions using:

```
FOR ThisChar IN 'A'..'Z' LOOP ...
```

or

```
FOR ThisDay IN Mon..Fri LOOP ...
```

we could have written the equivalent steps:

```
FOR ThisChar IN CapitalLetter LOOP ....
```
or
```
FOR ThisDay IN WeekDay LOOP ...
```

Notice how these forms of array declaration and FOR repetition can be used to improve program clarity.

14.5.1 Testing for membership of a subtype

It is often necessary to test whether a value belong to a particular range of values. For example we might wish to test whether the value stored in a variable Ch of type Character is a capital letter. Ada allows us to do this by using the boolean-valued expression of the form:

Variable IN *Range*

This expression has the value True if the value of *Variable* is contained within the specified *Range,* and False otherwise. Thus it can be used anywhere in an Ada program that a boolean value could be used. Hence for the example mentioned above, instead of writing a compound comparison:

```
IF Ch >= 'A' AND Ch <= 'Z' THEN ....
```

which is elaborate and somewhat error prone, we could write the equivalent condition using IN as:

```
IF Ch IN 'A'..'Z' THEN ....
```

or better still, using the subtype CapitalLetter defined in [14.1.1], as:

```
IF Ch IN CapitalLetter THEN ....
```

14.6 Other forms of repetition

Although the WHILE loop construction is sufficient for controlling all repetitions, it is not always the most convenient form. Of course, when the number of repetitions is known *before the loop commences* it is better to use a FOR loop. However, even when the number of repetitions is not known before the loop commences (so that we can't use a FOR loop), the WHILE loop does not always provide a very convenient solution since the controlling condition is evaluated at the beginning of the loop. For this reason a WHILE loop is sometimes referred to as a **pre-conditioned** loop.

As an example, consider using a WHILE loop to find the first non-blank character in some input data. We would write the fragment:

```
Get(Ch);  -- Input the first character.
-- So long as the last character input was a blank,
-- keep inputting characters.
WHILE Ch = Blank LOOP -- Last character input was a blank.
    Get(Ch);          -- Input another character.
END LOOP;
```

where we have assumed the declarations:

```
Blank : CONSTANT Character := ' ';
Ch    : Character;
```

The problem here really requires performance of:

i) Input a character.
ii) If the input character is a blank, then repeat from step (i).

From this description we see that the controlling condition is at the end of the steps being repeated, ie. we really have a **post-conditioned** repetition process. However if we use a (pre-conditioned) WHILE loop, the Get step has to be written out twice; once outside the loop and again inside the loop.

Another illustration of this situation occurs when re-using the subalgorithm for reading a set a data values followed by an explicit terminator. For the program at the end of Unit 13 we had the following steps in the main program:

```
Get(Age);
WHILE Age /= Terminator LOOP
    Category := AgeRange(Age);
    Total(Category) := Total(Category) + 1;
    Get(Age);
END LOOP;
```

Here we really want to perform:

i) Input an age.
ii) If the input age is zero, then stop the repetition.
iii) Work out the age category.
iv) Update the appropriate count.
v) Repeat from step (i).

Thus we see that the controlling condition is in the middle of the steps being repeated. However the use of a WHILE loop, again necessitates the duplication of some program steps; in this case the Get step has to be written out twice.

14.7 The LOOP..END LOOP construction

Ada also provides a more general looping construct which allows a controlling condition to be tested anywhere within the loop. This form of loop has the structure:

```
LOOP
    Step(s)_to_be_repeated
END LOOP;
```

where the *Step(s)_to_be_repeated* part of the loop is repeated indefinitely. However the loop will normally contain an EXIT step, which may cause the repetition to stop. EXIT steps have one of the two forms:

```
EXIT;
```

or

```
EXIT WHEN Condition;
```

If the first of these is executed, it causes the enclosing loop to stop repeating, (and execution continues with the step following the END LOOP). When the second form of EXIT step is executed, again it causes the loop to stop repeating, but only if the specified *Condition* is True. If the value of the *Condition* is False, execution of the repetition continues (with the step following the EXIT step).

We could use this LOOP..END LOOP form to rewrite the above repetitions by placing the controlling condition wherever it would naturally occur in the repetition, ie:

```
LOOP
    Get(Ch);                -- Input a character.
    EXIT WHEN Ch /= Blank; -- Stop if character was not a blank.
END LOOP;
```

and:

```
LOOP
    Get(Age);
    EXIT WHEN Age = Terminator;
    Category := AgeRange(Age);
    Total(Category) := Total(Category) + 1;
END LOOP;
```

These examples could have been rewritten using the first form of EXIT step. For example the fragment to input the next blank character could have been written as:

```
LOOP
    Get(Ch);
    IF Ch /= Blank THEN
          EXIT;
    END IF;
END LOOP;
```

However, the second form of EXIT step often provides a more compact and clearer solution.

Notes a) If the *Step(s)_to_be_repeated* does not contain an EXIT step the loop will repeat forever.

b) If the first form of EXIT step is used, it should always form part of some selection step such as an IF or CASE, since otherwise any steps following this step in the loop will never be reached.

c) In a LOOP..END LOOP construct, the *Step(s)_to_be_repeated* part may contain any valid program steps including another LOOP..END LOOP construct. In this case when an EXIT step is encountered in a nested inner loop only the innermost loop is terminated.

d) Observe that the condition in an EXIT step stops the LOOP..END LOOP repetition when the *Condition* is True, but the condition in a WHILE..LOOP..END LOOP stops the repetition when the controlling condition is False.

e) Any number of EXIT steps may appear the *Step(s)_to_be_repeated* part of a loop. This allows us to write loops with more than one terminating condition. However this feature should not be overused - loops with many different exit points are often difficult to understand and debug.

14.8 When to use a LOOP..END LOOP?

We see from (e) above that it is possible to have a loop with several exit points. As an example, consider a procedure to input and store the sequence of non-blank characters which make up an English word. We could store the word in a suitable array of characters defined by:

```
MaxWordLength : CONSTANT Integer := 20;
TYPE WordType IS ARRAY (1..MaxWordLength) OF Character;
```

If we were extracting words from an English sentence of the form:
It was a bright cold day in April, and the clocks were striking thirteen.
then each word will terminated by either a blank, comma or full-stop character, or by the end of a line (for simplicity we will assume that no other punctuation characters occur in the data).

We see that the body of the procedure will involve a loop which on each repetition attempts to input another character and store it in an array. This repetition should terminate when either a blank, comma or full-stop character is input, or if the end of a line is encountered. In addition, if more than 20 consecutive non-blank characters are input the repetition should also stop since the array is only defined to be capable of holding 20 characters. Thus a loop LOOP..END LOOP repetition might be considered as a suitable form of loop since it is possible to include several EXIT steps, giving:

```
PROCEDURE GetNextWord (Word : OUT WordType; Length : OUT Integer) IS
    -- To input the next sequence of non-blank characters
    -- (i.e. a word) into the Word array, and record the number
    -- of characters stored in Length.

    NextCh   : Character;      -- Next character input.
    Blank    : CONSTANT Character := ' ';
    Comma    : CONSTANT Character := ',';
    FullStop : CONSTANT Character := '.';
BEGIN
    Length := 0;   -- Initialise number of characters in the word.
    -- Now skip over spaces to the first non-blank character.
    LOOP
        Get(NextCh);
        EXIT WHEN NextCh /= Blank;
    END LOOP;
    -- Now store this character, and then input and store
    -- the remaining characters of the word.
    LOOP
        Length := Length + 1;
        Word(Length) := NextCh;
        EXIT WHEN End_Of_Line;
        Get(NextCh);
        CASE NextCh IS
            WHEN Blank|Comma|FullStop => EXIT;
            WHEN OTHERS => EXIT WHEN Length = MaxWordLength;
        END CASE;
    END LOOP;

END GetNextWord;
```

However it is often argued that loops with multiple exit points lack clarity, and hence are difficult to read and understand. The use of a LOOP..END LOOP repetition in the above example is acceptable, since

the *Step(s)_to_be_repeated* part is fairly short, and the exit points of the loop are relatively easy to identify. We could of course always rewrite such a repetition using a WHILE loop, but since this loop may stop repeating for several different reasons, how can we avoid writing a long, complicated (and hence unclear) controlling condition. The answer here is to use what is often referred to as a **state variable**, whose value controls the repetition process. For example, we could re-write the above procedure as:

```
PROCEDURE GetNextWord (Word   : OUT WordType;
                       Length : OUT Integer) IS
    -- To input the next sequence of non-blank characters
    -- (ie a word) into the Word array, and record the number
    -- of characters stored in Length.

    NextCh   : Character;     -- Next character input.
    Blank    : CONSTANT Character := ' ';
    Comma    : CONSTANT Character := ',';
    FullStop : CONSTANT Character := '.';
    TYPE State IS (Running, EOLReached, Terminated, WordTooLong);
    Status : State := Running;
BEGIN
    Length := 0;   -- Initialise number of characters in the word.
    -- First skip over spaces to the first non-blank character.
    LOOP
        Get(NextCh);
        EXIT WHEN NextCh /= Blank;
    END LOOP;
    -- Now store this character, and then input and store
    -- the remaining characters of the word.
    WHILE Status = Running LOOP
        Length := Length + 1;
        Word(Length) := NextCh;
        IF End_Of_Line THEN
            Status := EOLReached;
        ELSE
            Get(Item => NextCh);
            CASE NextCh IS
                WHEN Blank|Comma|FullStop => Status := Terminated;
                WHEN OTHERS => IF Length = MaxWordLength THEN
                                  Status := WordTooLong;
                               END IF;
            END CASE;
        END IF;
    END LOOP;

END GetNextWord;
```

In this version we have the WHILE loop controlled by a state variable Status, of enumeration type State. There is a single controlling condition Status = Running under which the repetition proceeds, but at various places within the loop the state variable can be assigned to one of the other values of the enumeration type (other than Running). This will cause the repetition to stop when the controlling condition is next evaluated. An advantage of the 'state variable' approach is that on exit from the repetition it is easy to determine what caused the repetition to stop. We could for example write (after the WHILE loop):

```
CASE Status IS
    WHEN EOLReached  => Put("Word terminated by end of line");
    WHEN Terminated  => Put("Word terminated by blank, comma");
                        Put(" or full-stop");
    WHEN WordTooLong => Put("Word was more than 20 characters");
    WHEN OTHERS      => NULL;
                        -- Can't happen, as loop would still repeat.
END CASE;
```

Notice that the last part of this CASE step includes the step NULL, which means 'do nothing'. This is needed because the CASE step must include all possible values of the selecting variable Status, and because Ada does not allow a sequence of steps to be left empty .