A Simple Console Application

The program below is a simple console application that you can compile and run from the command prompt:

```
program Greeting;

{$APPTYPE CONSOLE}

var
  MyMessage: string;

begin
  MyMessage := 'Hello world!';
  Writeln(MyMessage);
end.
```

The first line declares a program called Greeting. The `{$APPTYPE CONSOLE}` directive tells the compiler that this is a console application, to be run from the command line. The next line declares a variable called MyMessage, which holds a string. (Object Pascal has genuine string data types.) The program then assigns the string "Hello world!" to the variable MyMessage, and sends the contents of MyMessage to the standard output using the `Writeln`procedure. (`Writeln` is defined implicitly in the `System` unit, which the compiler automatically includes in every application.)

After you compile the program, the resulting executable prints the message `Hello world!`

Aside from its simplicity, this example differs in several important ways from programs that you are likely to write with Appmethod development tools. First, it is a console application. Appmethod development tools are most often used to write applications with graphical interfaces; hence, you would not ordinarily call `Writeln`. Moreover, the entire example program (save for `Writeln`) is in a single file. In a typical GUI application, the program heading and the first line of the example would be placed in a separate project file that would not contain any of the actual application logic, other than a few calls to routines defined in unit files.

A More Complicated Example

The next example shows a program that is divided into two files: a project file and a unit file. The project file, which you can save as `greeting.dpr`, looks like this:

```
program Greeting;

{$APPTYPE CONSOLE}

uses
  Unit1;

begin
  PrintMessage('Hello World!');
end.
```

The first line declares a program called `greeting`, which, once again, is a console application. The `uses Unit1;` clause tells the compiler that the program `greeting` depends on a unit called `Unit1`. Finally, the program calls the `PrintMessage` procedure, passing to it the string `Hello World!` The `PrintMessage` procedure is defined in `Unit1`. Here is the source code for `Unit1`, which must be saved in a file called `Unit1.pas`:

```
unit Unit1;
interface
procedure PrintMessage(msg: string);
implementation
procedure PrintMessage(msg: string);
begin
   Writeln(msg);
end;
end.
```

`Unit1` defines a procedure called `PrintMessage` that takes a single string as an argument and sends the string to the standard output. (In Object Pascal, routines that do not return a value are called procedures. Routines that return a value are called functions.)

Notice that `PrintMessage` is declared twice in `Unit1`. The first declaration, under the reserved word **interface**, makes `PrintMessage` available to other modules (such as `greeting`) that use `Unit1`. The second declaration, under the reserved word **implementation**, actually defines `PrintMessage`.

```
 When the compiler processes greeting.dpr, it automatically looks
for unit files that the greeting program depends on. The resulting
executable does the same thing as our first example: it prints the
message Hello world!
```

```
program greeting;
{$APPTYPE CONSOLE}
var MyMessage : string;
begin
MyMessage := 'Hello world!';
Writeln(MyMessage);
readln;
end.
```

```
program greeting;
{$APPTYPE CONSOLE}
var MyMessage : string;
begin
MyMessage := 'Hello world!';
Writeln(MyMessage);
readln;
end.
```

```
Program factorial;
{$APPTYPE CONSOLE}
var i, y, n :integer;
begin
    writeln('Enter a number n');
    readln(n);
    y:=1;
    for i:=1 to n do y:=y*i;
    writeln('Factorial of number  ',n,' is ',y);
  readln;
end.
```

```
program hi;
{$APPTYPE CONSOLE}
const ca = 5;
      cn = 'number 5';
var name : string;
   age : byte;
begin
    writeln('What is your name?');
    readln(name);
    writeln('Hi ' ,name, ', my name is ',cn, '.');
    writeln(' How old are you, ' , name, ' ?');
    readln(age);
    writeln(name, ', your ',age-ca,' years younger
computer , ',cn,', says hi to you!');
  readln;
end.
```

```
program numbers;
{$APPTYPE CONSOLE}
var number : integer;
begin
 writeln('Enter your special number');
 readln(number);
 case number of
  7: writeln('This is really a lucky number');
  13: writeln('No, this will bring you only bad luck');
  else writeln('Really boring number.');
 end;
 readln;
end.
```

```
program triangle;
{$APPTYPE CONSOLE}
var a,b,c : integer;
begin
Writeln('how long are the sides of your triangle?');
readln(a);
readln(b);
readln(c);
if (a+b)>c then begin
            if   (b+c)>a    then begin
                              if (a+c)>b then writeln('this triangle is real')
                                          else writeln('this triangle is not real');
                              end
                        else writeln('this triangle is not real');
             end
         else writeln('this triangle is not real');
readln;
end.
```

Declaration

```
This is the differnece:

When declaring variables:
var
    variablename: datatype;

When declaring constants:
const
    constantname = datatype;
```

Commands

Assignment

Function calls, because they return a value, can be used as expressions in assignments and operations. For example,

I := SomeFunction(X);

calls SomeFunction and assigns the result to I. Function calls cannot appear on the left side of an assignment statement.

I := x where x can be a value, formula or variable. I and x are of the same data type.

For example,
I := 10 I := a*b I := a

The For Loop

The **for loop** is a sort of repeat-until loop. The for loop, repeats a set of instructions for a number of times. The for loop is in the form:

- If used for only one action:

for *{variable}** := *{original value}* **to**/**downto** *{final value}* **do**

{code...(for one action)}

- If used for more than one action:

for *{variable}** := *{original value}* **to**/**downto** *{final value}* **do Begin**

{code...}

{code...}

End;

*Generally, this variable is called the **'loop counter'**.

Now, an example of the for loop is shown below, but firstly, you should have an idea of the usefulness of the **for loop**. Consider the following example:

using for

```
program usingit;
{$APPTYPE CONSOLE}
var sentence : string;
begin
sentence := 'Noooooooooo!';
Writeln(sentence);
Writeln(sentence);
Writeln(sentence);
readln;
end.
```

not using for

```
program notusingit;
{$APPTYPE CONSOLE}
var sentence : string;
n,i : byte;
begin
sentence := ' Noooooooooo!';
n := 3;
for i :=1 to n do
begin
Writeln(sentence);
i := i+1
end;
readln;
end.
```

While-Do Loop

This type of loop is executed ***while* the condition is true**. It is different from the 'Repeat-Until' loop since the loop might not be executed for at least one time. The code works like this:

While <condition is true> do the following:

instruction 1;

instruction 2;

instruction 3;

etc...

End; {If while-do loop starts with a begin statement}

The Simple Case Statement

In some cases the '**case statement**' is preferred to the if statement because it reduces some unnecessary code but the same meaning is retained. The case statement is very similar to the if statement, except in that the it does not accept literal conditional expressions (i.e.: strings) but surprisingly enough, it allows single character conditional expressions. Here is how it works:

Case {variable of type: integer or character ONLY} of

{input statement- within inverted commas if of type char} : {code..}

{input statement- within inverted commas if of type char} : {code..}

...

End; {End Case}

The Case-Else Statement

Again this is similar to the **if..then..else** statement.

The If Statement

The 'if statement' executes a the proceeding statement(s) conditionally. This means that if an action comes to be true, then the statement(s) proceeding the if statement are executed, else these statements are skipped. It works like this:

If *this happens*(action)*,* ***then*** *do this*(reaction, if action is true)*.*

OR:

If *this happens*(action)*,* ***then*** *do this*(reaction, if action is true)*,* ***else*** *do this*(reaction, if action is false)*.*

In Pascal, the 'if statement' should be written as follows:

If conditional expression **then** code ... ; *//if one action*

OR:

If conditional expression **then Begin** instructions ... **End**; *//if more than one action is required*

Note that you should not use an assignment statement in the '**if**' construct, otherwise the compiler will raise a **syntax error**. I.e.:

Wrong:

If x := 20 **then** x := x + 1; *//the underlined character must be excluded*

Correct:

If x = 20 **then** x := x + 1; //only an equal sign is used for comparison

If..Then..Else

In a normal if statement, the 'reaction' cannot be performed if the condition is not true. But in an **if..then..else** statement, there is at least one set of statements to be performed. Let's take a look at the example below:

```
writeln('Who has discovered the land of America?');
Readln(ans);
If (ans = 'Christopher Colombus') then
 score := score + 1                          //if this is false,
ELSE
 writeln('sorry, you''ve got it wrong!');     //then this is true
```

Note that if the 'else' term is included with an if statement, then there should be**no semi-colon** before the 'else' term; just as seen in the above example.

The Repeat-Until Loop

This loop is used to repeat the execution of a set of instructions for at least one time. It is repeated until the conditional expression is obeyed. The following example, shows the model of the 'repeat-until' loop:

Repeat

..(code)

..(code)

..(code)

Until *conditional statement**;***