5. Fortran – Variables

A variable is nothing but a name given to a storage area that our programs can manipulate. Each variable should have a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable.

The name of a variable can be composed of letters, digits, and the underscore character. A name in Fortran must follow the following rules:

- It cannot be longer than 31 characters.
- It must be composed of alphanumeric characters (all the letters of the alphabet, and the digits 0 to 9) and underscores (_).
- First character of a name must be a letter.
- Names are case-insensitive.

Based on the basic types explained in previous chapter, following are the variable types:

Type	Description
Integer	It can hold only integer values.
Real	It stores the floating point numbers.
Complex	It is used for storing complex numbers.
Logical	It stores logical Boolean values.
Character	It stores characters or strings.

Variable Declaration

Variables are declared at the beginning of a program (or subprogram) in a type declaration statement.

Syntax for variable declaration is as follows:

```
type-specifier :: variable_name
```

For example,

```
integer :: total
real :: average
complex :: cx
logical :: done
character(len=80) :: message ! a string of 80 characters
```

Later you can assign values to these variables, like,

```
total = 20000
average = 1666.67
done = .true.
message = “A big Hello from Tutorials Point”
cx = (3.0, 5.0) ! cx = 3.0 + 5.0i
```

You can also use the intrinsic function **cmplx,** to assign values to a complex variable:

```
cx = cmplx (1.0/2.0, -7.0) ! cx = 0.5 – 7.0i
cx = cmplx (x, y) ! cx = x + yi
```

Example

The following example demonstrates variable declaration, assignment and display on screen:

```
program variableTesting
implicit none

   ! declaring variables
   integer :: total
   real :: average
   complex :: cx
   logical :: done
   character(len=80) :: message ! a string of 80 characters

   !assigning values
   total = 20000
   average = 1666.67
   done = .true.
   message = "A big Hello from Tutorials Point"
   cx = (3.0, 5.0) ! cx = 3.0 + 5.0i

   Print *, total
   Print *, average
   Print *, cx
   Print *, done
   Print *, message

end program variableTesting
```

When the above code is compiled and executed, it produces the following result:

```
20000
1666.67004
(3.00000000, 5.00000000 )
T
A big Hello from Tutorials Point
```

6. Fortran – Constants

The constants refer to the fixed values that the program cannot alter during its execution. These fixed values are also called **literals**.

Constants can be of any of the basic data types like an integer constant, a floating constant, a character constant, a complex constant, or a string literal. There are only two logical constants : **.true.** and **.false.**

The constants are treated just like regular variables, except that their values cannot be modified after their definition.

Named Constants and Literals

There are two types of constants:

- Literal constants
- Named constants

A literal constant have a value, but no name.

For example, following are the literal constants:

Type	Example
Integer constants	0 1 -1 300 123456789
Real constants	0.0 1.0 -1.0 123.456 7.1E+10 -52.715E-30
Complex constants	(0.0, 0.0) (-123.456E+30, 987.654E-29)
Logical constants	.true. .false.
Character constants	"PQR" "a" "123'abc$%#@!" " a quote "" " 'PQR' 'a' '123"abc$%#@!' ' an apostrophe '' '

A named constant has a value as well as a name.

Named constants should be declared at the beginning of a program or procedure, just like a variable type declaration, indicating its name and type. Named constants are declared with the parameter attribute. For example,

```
real, parameter :: pi = 3.1415927
```

Example

The following program calculates the displacement due to vertical motion under gravity.

```
program gravitationalDisp

! this program calculates vertical motion under gravity
implicit none

   ! gravitational acceleration
   real, parameter :: g = 9.81

   ! variable declaration
   real :: s ! displacement
   real :: t ! time
   real :: u ! initial speed

   ! assigning values
   t = 5.0
   u = 50

   ! displacement
   s = u * t - g * (t**2) / 2

   ! output
   print *, "Time = ", t
   print *, 'Displacement = ',s

end program gravitationalDisp
```

When the above code is compiled and executed, it produces the following result:

```
Time = 5.00000000
Displacement = 127.374992
```

7. Fortran – Operators

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. Fortran provides the following types of operators:

- Arithmetic Operators
- Relational Operators
- Logical Operators

Let us look at all these types of operators one by one.

Arithmetic Operators

Following table shows all the arithmetic operators supported by Fortran. Assume variable **A**holds 5 and variable **B** holds 3 then:

Operator	Description	Example
+	Addition Operator, adds two operands.	A + B will give 8
-	Subtraction Operator, subtracts second operand from the first.	A - B will give 2
*	Multiplication Operator, multiplies both operands.	A * B will give 15
/	Division Operator, divides numerator by de-numerator.	A / B will give 1
**	Exponentiation Operator, raises one operand to the power of the other.	A ** B will give 125

Example

Try the following example to understand all the arithmetic operators available in Fortran:

```
program arithmeticOp

! this program performs arithmetic calculation
implicit none

   ! variable declaration
```

```
    integer :: a, b, c

    ! assigning values
    a = 5
    b = 3

    ! Exponentiation
    c = a ** b

    ! output
    print *, "c = ", c

    ! Multiplication
    c = a * b

    ! output
    print *, "c = ", c

    ! Division
    c = a / b

    ! output
    print *, "c = ", c

    ! Addition
    c = a + b

    ! output
    print *, "c = ", c

    ! Subtraction
    c = a - b

    ! output
    print *, "c = ", c
```

```
end program arithmeticOp
```

When you compile and execute the above program, it produces the following result:

```
c = 125
c = 15
c = 1
c = 8
c = 2
```

Relational Operators

Following table shows all the relational operators supported by Fortran. Assume variable **A**holds 10 and variable **B** holds 20, then:

Operator	Equivalent	Description	Example
==	.eq.	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A == B) is not true.
/=	.ne.	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(A /= B) is true.
>	.gt.	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
<	.lt.	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>=	.ge.	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<=	.le.	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true.

Example

Try the following example to understand all the logical operators available in Fortran:

```
program logicalOp

! this program checks logical operators
implicit none

   ! variable declaration
   logical :: a, b

   ! assigning values
   a = .true.
   b = .false.

   if (a .and. b) then
      print *, "Line 1 - Condition is true"
   else
      print *, "Line 1 - Condition is false"
   end if

   if (a .or. b) then
      print *, "Line 2 - Condition is true"
   else
      print *, "Line 2 - Condition is false"
   end if

   ! changing values
   a = .false.
   b = .true.

   if (.not.(a .and. b)) then
      print *, "Line 3 - Condition is true"
   else
      print *, "Line 3 - Condition is false"
   end if

   if (b .neqv. a) then
```

```
      print *, "Line 4 - Condition is true"
   else
      print *, "Line 4 - Condition is false"
   end if

   if (b .eqv. a) then
      print *, "Line 5 - Condition is true"
   else
      print *, "Line 5 - Condition is false"
   end if

end program logicalOp
```

When you compile and execute the above program it produces the following result:

```
Line 1 - Condition is false
Line 2 - Condition is true
Line 3 - Condition is true
Line 4 - Condition is true
Line 5 - Condition is false
```

Logical Operators

Logical operators in Fortran work only on logical   values .true. and .false.

The following table shows all the logical operators supported by Fortran. Assume variable A holds **.true.** and variable B holds **.false.** , then:

Operator	Description	Example
.and.	Called Logical AND operator. If both the operands are non-zero, then condition becomes true.	(A .and. B) is false.
.or.	Called Logical OR Operator. If any of the two operands is non-zero, then condition becomes true.	(A .or. B) is true.
.not.	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	!(A .not. B) is true.
.eqv.	Called Logical EQUIVALENT Operator. Used to check equivalence of two logical values.	(A .eqv. B) is false.

.neqv.	Called Logical NON-EQUIVALENT Operator. Used to check non-equivalence of two logical values.	(A .neqv. B) is true.

Example

Try the following example to understand all the logical operators available in Fortran:

```
program logicalOp
! this program checks logical operators
implicit none
   ! variable declaration
   logical :: a, b
   ! assigning values
   a = .true.
   b = .false.
   if (a .and. b) then
      print *, "Line 1 - Condition is true"
   else
      print *, "Line 1 - Condition is false"
   end if

    if (a .or. b) then
      print *, "Line 2 - Condition is true"
   else
      print *, "Line 2 - Condition is false"
   end if

   ! changing values
   a = .false.
   b = .true.

   if (.not.(a .and. b)) then
      print *, "Line 3 - Condition is true"
   else
      print *, "Line 3 - Condition is false"
   end if
   if (b .neqv. a) then
      print *, "Line 4 - Condition is true"
   else
      print *, "Line 4 - Condition is false"
   end if
   if (b .eqv. a) then
      print *, "Line 5 - Condition is true"
   else
      print *, "Line 5 - Condition is false"
   end if

end program logicalOp
```

When you compile and execute the above program it produces the following result:

```
Line 1 - Condition is false
Line 2 - Condition is true
Line 3 - Condition is true
Line 4 - Condition is true
Line 5 - Condition is false
```

Operators Precedence in Fortran

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator.

For example, x = 7 + 3 * 2; here, x is assigned 13, not 20 because operator * has higher precedence than +, so it first gets multiplied with 3*2 and then adds into 7.

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

Category	Operator	Associativity
Logical NOT and negative sign	.not. (-)	Left to right

Exponentiation	**	Left to right
Multiplicative	* /	Left to right
Additive	+ -	Left to right
Relational	< <= > >=	Left to right
Equality	== !=	Left to right
Logical AND	.and.	Left to right
Logical OR	.or.	Left to right
Assignment	=	Right to left

Example

Try the following example to understand the operator precedence in Fortran:

```
program precedenceOp
! this program checks logical operators

implicit none

   ! variable declaration
   integer :: a, b, c, d, e

   ! assigning values
   a = 20
   b = 10
   c = 15
   d = 5

   e = (a + b) * c / d      ! ( 30 * 15 ) / 5
   print *, "Value of (a + b) * c / d is :    ",  e
   e = ((a + b) * c) / d    ! (30 * 15 ) / 5
   print *, "Value of ((a + b) * c) / d is  : ",  e

   e = (a + b) * (c / d);   ! (30) * (15/5)
   print *, "Value of (a + b) * (c / d) is  : ",  e

   e = a + (b * c) / d;     !  20 + (150/5)
   print *, "Value of a + (b * c) / d is  :   " ,  e

end program precedenceOp
```

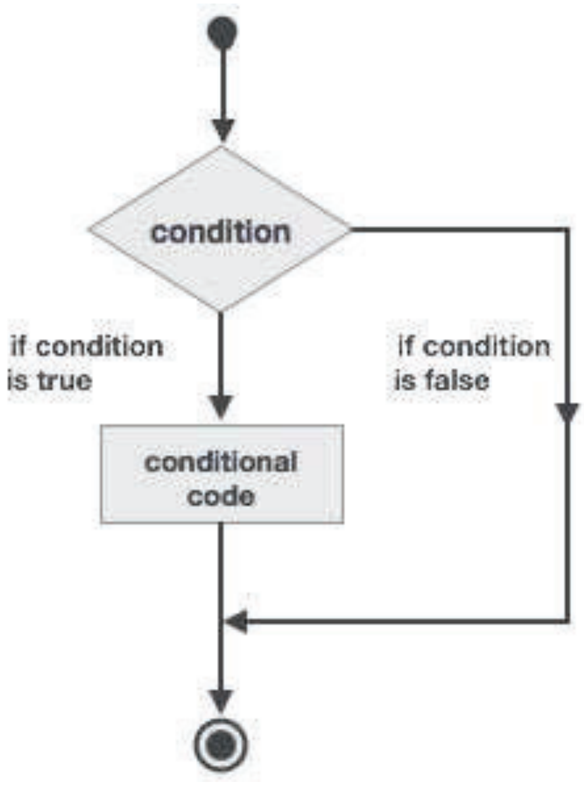
When you compile and execute the above program it produces the following result:

```
Value of (a + b) * c / d is : 90
Value of ((a + b) * c) / d is : 90
Value of (a + b) * (c / d) is : 90
Value of a + (b * c) / d is : 50
```

8. Fortran – Decisions

Decision making structures require that the programmer specify one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed, if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

Following is the general form of a typical decision making structure found in most of the programming languages:



Fortran provides the following types of decision making constructs.

Statement	Description
If… then construct	An if… then… end if statement consists of a logical expression followed by one or more statements.
If… then...else construct	An if… then statement can be followed by an optional else statement, which executes when the logical expression is false.

nested if construct	You can use one if or else if statement inside another if or else if statement(s).
select case construct	A select case statement allows a variable to be tested for equality against a list of values.
nested select case construct	You can use one select case statement inside another select case statement(s).

If…then Construct

An **if… then** statement consists of a logical expression followed by one or more statements and terminated by an **end if** statement.

Syntax

The basic syntax of an **if… then** statement is:

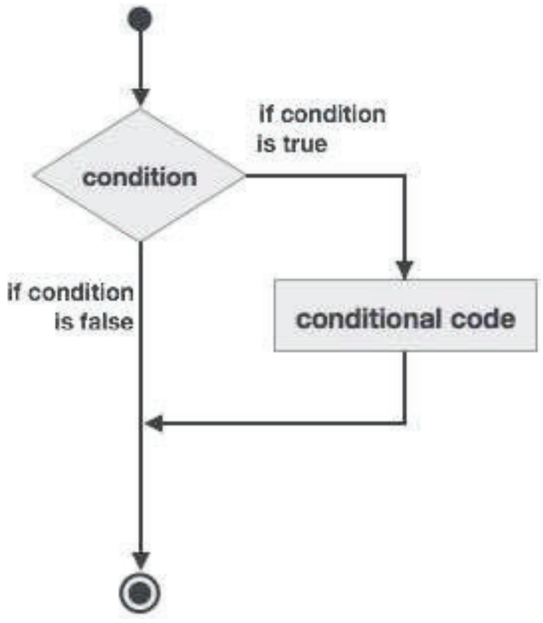
```
if (logical expression) then
   statement
end if
```

However, you can give a name to the **if** block, then the syntax of the named **if** statement would be, like:

```
[name:] if (logical expression) then
   ! various statements
   . . .
end if [name]
```

If the logical expression evaluates to **true**, then the block of code inside the **if**…**then** statement will be executed. If logical expression evaluates to **false**, then the first set of code after the **end if** statement will be executed.

Flow Diagram



Example 1

```
program ifProg
implicit none
   ! local variable declaration
   integer :: a = 10

   ! check the logical condition using if statement
   if (a < 20 ) then

       ! if condition is true then print the following
       print*, "a is less than 20"
   end if

   print*, "value of a is ", a

end program ifProg
```

When the above code is compiled and executed, it produces the following result:

```
a is less than 20
 value of a is     10
```

Example 2

This example demonstrates a named **if** block:

```
program markGradeA
implicit none
  real :: marks
  ! assign marks
  marks = 90.4
  ! use an if statement to give grade

  gr: if (marks > 90.0) then
  print *, " Grade A"
  end if gr
 end program markGradeA
```

When the above code is compiled and executed, it produces the following result:

```
Grade A
```

If… then… else Construct

An **if… then** statement can be followed by an optional **else statement**, which executes when the logical expression is false.

Syntax

The basic syntax of an **if… then… else** statement is:

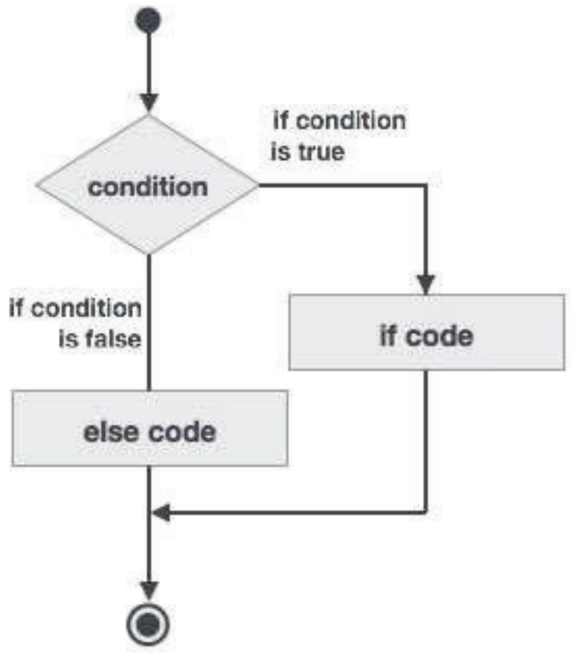
```
if (logical expression) then
   statement(s)
else
   other_statement(s)
end if
```

However, if you give a name to the **if** block, then the syntax of the named **if-else** statement would be, like:

```
[name:] if (logical expression) then
      ! various statements
   . . .
   else
      !other statement(s)
   . . .
end if [name]
```

If the logical expression evaluates to **true**, then the block of code inside the **if**…**then** statement will be executed, otherwise the block of code inside the **else** block will be executed.

Flow Diagram



Example

```
program ifElseProg
implicit none
   ! local variable declaration
   integer :: a = 100
   ! check the logical condition using if statement
   if (a < 20 ) then

      ! if condition is true then print the following
      print*, "a is less than 20"
   else
      print*, "a is not less than 20"
   end if
   print*, "value of a is ", a
end program ifElseProg
```

When the above code is compiled and executed, it produces the following result:

```
a is not less than 20
value of a is      100
```

if...else if...else Statement

An **if** statement construct can have one or more optional **else-if** constructs. When the **if**condition fails, the immediately followed **else-if** is executed. When the **else-if** also fails, its successor **else-if** statement (if any) is executed, and so on.

The optional else is placed at the end and it is executed when none of the above conditions hold true.

- All else statements (else-if and else) are optional.
- **else-if** can be used one or more times
- **else** must always be placed at the end of construct and should appear only once.

Syntax

The syntax of an **if...else if...else** statement is:

```
[name:]
if (logical expression 1) then
   ! block 1
else if (logical expression 2) then
   ! block 2
else if (logical expression 3) then
   ! block 3
else
```

```
   ! block 4
end if [name]
```

Example

```
program ifElseIfElseProg
implicit none

   ! local variable declaration
   integer :: a = 100

   ! check the logical condition using if statement
   if( a == 10 ) then

      ! if condition is true then print the following
      print*, "Value of a is 10"

   else if( a == 20 ) then

      ! if else if condition is true
      print*, "Value of a is 20"

   else if( a == 30 ) then

      ! if else if condition is true
      print*, "Value of a is 30"
   else
      ! if none of the conditions is true
      print*, "None of the values is matching"
   end if
   print*, "exact value of a is ", a
end program ifElseIfElseProg
```

When the above code is compiled and executed, it produces the following result:

```
None of the values is matching
exact value of a is 100
```

Nested If Construct

You can use one **if** or **else if** statement inside another **if** or **else if** statement(s).

Syntax

The syntax for a nested **if** statement is as follows:

```
if ( logical_expression 1) then
   !Executes when the boolean expression 1 is true
   …
   if(logical_expression 2)then
      ! Executes when the boolean expression 2 is true
        …
   end if
end if
```

Example

```
program nestedIfProg
implicit none
   ! local variable declaration
   integer :: a = 100, b= 200

   ! check the logical condition using if statement
   if( a == 100 ) then

       ! if condition is true then check the following
       if( b == 200 ) then
         ! if inner if condition is true
         print*, "Value of a is 100 and b is 200"
        end if
   end if
   print*, "exact value of a is ", a
   print*, "exact value of b is ", b
end program nestedIfProg
```

When the above code is compiled and executed, it produces the following result:

```
Value of a is 100 and b is 200
exact value of a is          100
exact value of b is          200
```

Select Case Construct

A **select case** statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being selected on is checked for each **select case**.

Syntax

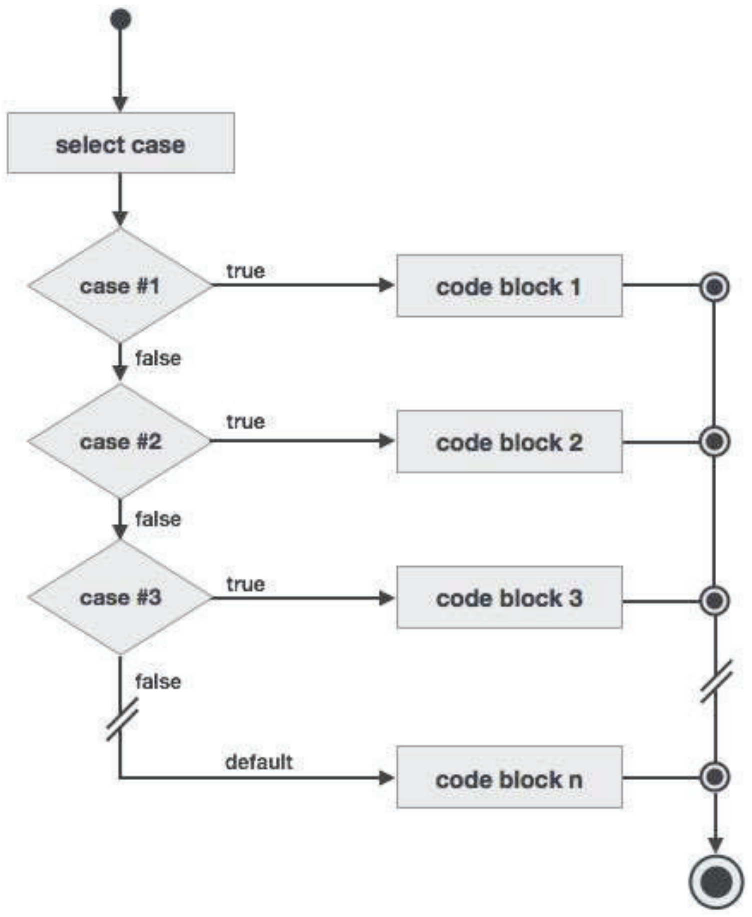
The syntax for the **select case** construct is as follows:

```
[name:] select case (expression)
      case (selector1)
       ! some statements
       ...       case (selector2)
      ! other statements
       ...
      case default
     ! more statements
       ...
end select [name]
```

The following rules apply to a **select** statement:

- The logical expression used in a select statement could be logical, character, or integer (but not real) expression.
- You can have any number of case statements within a select. Each case is followed by the value to be compared to and could be logical, character, or integer (but not real) expression and determines which statements are executed.
- The constant-expression for a case, must be the same data type as the variable in the select, and it must be a constant or a literal.
- When the variable being selected on, is equal to a case, the statements following that case will execute until the next case statement is reached.
- The case default block is executed if the expression in select case (expression) does not match any of the selectors.

Flow Diagram



Example 1

```
program selectCaseProg
implicit none
   ! local variable declaration
   character :: grade = 'B'

   select case (grade)

   case ('A')
      print*, "Excellent!"

   case ('B')
   case ('C')
      print*, "Well done"

   case ('D')
      print*, "You passed"

   case ('F')
      print*, "Better try again"

   case default
      print*, "Invalid grade"
   end select
   print*, "Your grade is ", grade

end program selectCaseProg
```

When the above code is compiled and executed, it produces the following result:

```
Your grade is B
```

Specifying a Range for the Selector

You can specify a range for the selector, by specifying a lower and upper limit separated by a colon:

```
case (low:high)
```

The following example demonstrates this:

Example 2

```
program selectCaseProg
implicit none
   ! local variable declaration
   integer :: marks = 78

   select case (marks)

   case (91:100)
      print*, "Excellent!"
   case (81:90)
      print*, "Very good!"

   case (71:80)
      print*, "Well done!"

   case (61:70)
      print*, "Not bad!"

   case (41:60)
      print*, "You passed!"

   case (:40)
      print*, "Better try again!"

   case default
      print*, "Invalid marks"
   end select
   print*, "Your marks is ", marks

end program selectCaseProg
```

When the above code is compiled and executed, it produces the following result:

```
Well done!
Your marks is           78
```

Nested Select Case Construct

You can use one **select case** statement inside another **select case** statement(s).

Syntax

```
select case(a)
      case (100)
         print*, "This is part of outer switch", a
         select case(b)
```

```
            case (200)
               print*, "This is part of inner switch", a
         end select
   end select
```

Example

```
program nestedSelectCase
   ! local variable definition
   integer :: a = 100
   integer :: b = 200

   select case(a)
      case (100)
         print*, "This is part of outer switch", a
         select case(b)
            case (200)
               print*, "This is part of inner switch", a
         end select
   end select
   print*, "Exact value of a is : ", a
   print*, "Exact value of b is : ", b

end program nestedSelectCase
```

When the above code is compiled and executed, it produces the following result:

```
This is part of outer switch         100
This is part of inner switch         100
Exact value of a is :          100
Exact value of b is :          200
```