

## 1.1 Fables Enumerated Types

In Fables, enumerated types (*enums*) are data types consisting of a set of named constant elements. These identifiers are handled as named constants in the language.

Example:

```
enum Strategy { Cooperate, Defect, Abide } ;
```

The elements of an enumerated type are handled as distinct constants. By default, the first element represents the value of 0, and this default value is incremented by one for each new element. When one of the elements (or the enumerated type) is printed on the screen, their value can be verified.

Example:

```
enum Strategy { Cooperate, Defect, Abide } ;

startUp {
    println( Strategy ) ;
}
```

Result:

```
[ Cooperate (0), Defect (1), Abide (2) ]
```

The default assigned value can be altered during the evaluation. In this case, the following element gets the previous element's value increased by one.

Example:

```
enum Strategy { Cooperate, Defect = 5, Abide } ;

startUp {
    println( Strategy ) ;
}
```

Result:

```
[ Cooperate (0), Defect (5), Abide (6) ]
```

*Defect is defined with the value of 5, hence Abide get the next integer value by default, which is 6.*

## 1.2 Enum Definitions

The general form to declare an enumerated type is the following:

```
enum <EnumName> {
    <Element 1> [ = Value 1 ] , <Element 2> [ = Value 2 ] ,
    <Element 3> [ = Value 3 ] , ...
}
```

When defining enumerated types, please consider the following restrictions:

- The name of the enum elements must be unique identifiers.

```
enum MyEnum { A, B, A } ; // Error: duplicate identifier A
```

- Currently the assigned values must be integer constants. Neither function calls nor operators may be used in the initial assignment.

Example:

```
enum Strategy {
    Cooperate ,
```

```

    Defect = 5 , // OK
    Abide = sqrt( 5 * uniform(1,10) ) // Syntax Error
} ;

```

- The values assigned to enum elements must be unique, and form a strictly monotonic sequence.

Example 1:

```

enum Strategy {
    Cooperate , // Default value 0
    Defect , // Default value 1
    Abide = 0 // Error: duplicate 0 values in the same definition
} ;

```

Example 2:

```

enum Strategy {
    Cooperate,
    Defect = 10 ,
    Abide = 5 // Error: value is lower than previous constant
} ;

```

## 1.3 Using Enum Elements

Enumerated types have a twofold behaviour: they are an aggregate type (since they have other members), and on the other hand, they can be used as a collection of integer values.

Like in the case of classes, enum elements can be referred by using the member operator:

```

enum Strategy { Cooperate, Defect, Abide } ;

startUp {
    println( Strategy.Cooperate ) ; // Prints out the default value of Cooperate (0)
}

```

And enum types can be used in every situation where a collection of integer values may be used:

```

enum Strategy { Cooperate, Defect, Abide } ;

startUp {
    // Prints out the value of one of the elements
    println( discreteUniform( Strategy ) ) ;
}

```

As any other collections, they may be indexed as well:

```

enum Strategy { Cooperate, Defect, Abide } ;

startUp {
    println( Strategy(0) ) ; // Prints out the value of the first element
}

```

And, for instance, used as an iteration:

```
enum Strategy { Cooperate, Defect, Abide } ;

startUp {
    for each strategy in Strategy do
        println( strategy ) ;
}
```

Variables may also be defined having an enumerated type can be assigned any of the enum's elements. They are going to contain the value of the specified constant:

```
enum Strategy { Cooperate, Defect, Abide } ;
var currentStrategy:= discreteUniform( Strategy ) ;

startUp {
    println( currentStrategy ) ;
}
```

Enum elements may also be used everywhere where an integer expression is accepted:

```
enum Strategy { Cooperate = 10, Defect = 20, Abide = 30 } ;

var resources := 100 ;
updateResources( action ) = resources := resources - action ;

startUp {
    updateResources( Strategy.Cooperate ) ;
    println( resources ); // Prints out 100 - 10 = 90
}
```

## 1.4 Special Conditional Structures for Enum Types

To use enum types efficiently, there is a special syntax element to create conditional expressions in the following form:

```
if ( <action> )
    is ( <enum element 1> ) => <statement 1>
    is ( <enum element 2> ) => <statement 2>
    ...
    otherwise                => <statement N>
```

Example:

```
enum Strategy { Cooperate, Defect, Abide } ;
var strategy := discreteUniform( Strategy ) ;

startUp {
    if ( strategy )
        is Strategy.Cooperate => println( "Cooperate" )
        is Strategy.Defect    => println( "Defect" )
        is Strategy.Abide     => println( "Abide" ) ;
}
```

The branches of the statement must contain statements with the same return value (e.g. each of them must be a statement, or all of them must return a number, a String, etc.).

In the case the branches contain only statements, then any of the branches may be omitted. In the other case all of the branches have to be defined, or an `otherwise` case has to be declared.