

P:/iutAcy/logo\_IUT\_Annecy\_CMJN.jpg

# PROGRAMMATION ORIENTÉE OBJET

Cours

v0

*IUT d'Annecy, 9 rue de l'Arc en Ciel, 74940 Annecy*

## POO SUR CODESYS

Après un rapide rappel sur les notions de base de la programmation orientée objet, ce cours s'intéresse à la problématique suivante :



### Problématique

Comment mettre en place une programmation orientée objet sous CodeSys ?

## Sommaire

### 1 Rappels sur la programmation orientée objet

#### 1.1 Quelques définitions

La programmation orientée objet est une technique de programmation qui consiste à découper un programme en plusieurs objets qui interagissent entre eux. Chaque objet est une instance d'une classe. Une classe est un ensemble d'attributs et de méthodes. Les attributs sont des variables propres à chaque objet. Les méthodes sont des fonctions propres à chaque objet.

Il s'agit d'une façon de penser la programmation. Si elle peut paraître plus complexe au début, elle permet de structurer le code, de le rendre plus lisible et plus facilement modifiable.

La définition des objets (création de classes) peut être complexe, mais l'utilisation des objets est largement simplifiée. C'est d'ailleurs un avantage majeur de la programmation orientée objet : l'utilisateur d'un objet n'a pas besoin de connaître sa définition pour l'utiliser. Il lui suffit de connaître les méthodes et les attributs de l'objet.

Un objet est une instance d'une classe. Cela veut simplement dire qu'une classe définit une structure (attributs et méthodes) et qu'un objet est un cas particulier de cette structure. Par analogie, on peut parler des plans d'une maison qui définissent ce qu'est une maison (nombre de pièces, surface, etc.) et des maisons qui sont des instances de ces plans.

Chaque maison pourra avoir un nombre de pièces différentes, une couleur de toit différente, mais toutes les maisons auront les mêmes caractéristiques de base (elles auront toutes un toit, des pièces, des murs, des fenêtres, etc.).

D'un point de vue extérieur, un objet est une "boîte noire" dont les détails d'implémentation sont cachés. On ne peut pas savoir comment est codé un objet, on interagit avec lui par l'intermédiaire des fonctions d'interfaces. C'est le principe d'encapsulation.



## Principe d'encapsulation

L'encapsulation est un principe de la programmation orientée objet qui consiste à cacher les détails d'implémentation d'un objet. L'accès aux données de l'objet ne peut alors se faire que par l'intermédiaire des services proposées.

Cela a pour objectif d'en simplifier l'utilisation mais également de **garantir l'intégrité des données** contenues dans l'objet. En effet, les fonctions d'interfaces mises en places contrôlent la bonne utilisation des données.

### Exemple : Etape d'un grafcet

On peut définir une classe *CStep* qui représente une étape d'un grafcet. Cette classe peut contenir, entre autres, les attributs et méthodes suivants :

#### Attributs

- **m\_name** : nom de l'étape
- **m\_xActivityBit** : bit d'activité (actif ou non)
- **m\_tActivationDuration** : temps d'activation de l'étape

#### Méthodes

- **MInit** : Initialise l'étape
- **MActivate** : active l'étape
- **MDeactivate** : désactive l'étape

Chaque étape du grafcet sera alors un objet de la classe *CStep*.



## La notion d'Héritage

L'héritage est un principe de la programmation orientée objet qui consiste à créer une nouvelle classe à partir d'une classe existante. La nouvelle classe **hérite** alors des attributs et méthodes de la classe existante.

Par exemple, on peut créer une classe *Forme* qui contient les attributs et méthodes communs à toutes les formes géométriques fermées (comme son aire). On peut ensuite créer une classe *Rectangle* qui hérite de la classe *Forme* et qui contient, en plus, les attributs et méthodes spécifiques aux rectangles. On peut ensuite créer une classe *Cercle* qui hérite de la classe *Forme* et qui contient les attributs et méthodes spécifiques aux cercles.



## Le Polymorphisme

Le polymorphisme est un principe de la programmation orientée objet qui consiste à utiliser un objet de manière différente en fonction du contexte.

Plus spécifiquement, le polymorphisme autorise à utiliser un même nom pour des méthodes différentes selon si elle s'applique sur un contexte ou un autre.

En reprenant l'exemple précédent, on peut créer une méthode *MComputeArea* dans la classe *Forme* qui calcule l'aire de la forme. Cette méthode sera différente dans la classe *Rectangle* et dans la classe *Cercle* car l'aire d'un rectangle et l'aire d'un cercle ne se calculent pas de la même manière. En fonction de l'objet utilisé, la méthode *MComputeArea* sera différente.

## 2 La programmation orientée objet sous CodeSys



### Application : Etape d'un grafcet

Afin de faciliter la compréhension de cette partie, nous allons construire, tout au long de ce cours, une classe *CStep* qui représente une étape d'un grafcet. Cette classe contiendra les attributs et méthodes suivants :

**Attributs**

- **m\_xActivityBit** : bit d'activité (actif ou non)
  - ◇ Accessible en lecture uniquement
- **m\_tActivationDuration** : Stocke le temps d'activation de l'étape
  - ◇ Accessible en lecture uniquement
- **m\_tTaskPeriod** : période de la tâche
  - ◇ Accessible en lecture et en écriture

**Méthodes**

- **MInit** : Méthode appelée à l'initialisation du programme
- **MActivate** : active l'étape
- **MDeactivate** : désactive l'étape

## 2.1 Les classes

On l'a vu, la notion de classe nous demande de pouvoir créer des attributs et des méthodes. Les attributs sont des variables internes aux classes qui devront être conservées entre deux utilisations d'un objet de cette classe. L'utilisation de fonctions (POU FUNCTION) ne serait pas satisfaisante, car la durée de vie d'une variable interne à une fonction est limitée à l'exécution de cette fonction.

A l'inverse, un bloc fonctionnel s'utilise après en avoir créé une instance et ses variables internes sont conservées entre d'une utilisation à l'autre. Les POU FB semblent donc adaptées à la création de classes.



### À retenir

Nous utiliserons les blocs fonctions pour implémenter la notion de classe sous CodeSys.



### Déclaration d'une classe sous CodeSys

Définir une classe sous CodeSys revient à définir un bloc fonctionnel. Afin de différencier les variables privées (inaccessibles depuis l'extérieur) des attributs publics (accessibles depuis l'extérieur), on utilisera la convention suivante :

- Les variables publiques commenceront par **m\_**
- les variables privées n'ont pas de préfixe particulier

Ainsi, la déclaration d'une classe <CName> se fera de la manière suivante :

```

1  FUNCTION_BLOCK <CName> (* définition de la classe <CName> *)
2  VAR
3      <type><Name> : <type>; (* variable locale non accessible de l'extérieur *)
4      m_<type><Name> : <type>; (* membre de l'objet accessible de l'extérieur sous certaines conditions *)
5  END_VAR
6

```



### Attributs de la classe CStep

Puisqu'elle ne possède que des attributs accessibles, la déclaration de notre classe *CStep* se fera de la manière suivante :

```

1  FUNCTION_BLOCK FB_CStep
2  VAR
3      m_xActivityBit : BOOL; (* bit d'activité (actif ou non) *)
4      m_tActivationDuration : TIME; (* Stocke le temps d'activation de l'étape *)
5      m_tTaskPeriod : TIME;
6  END_VAR

```

L'instanciation d'un objet de cette classe pourra alors s'écrire `Etape1 : FB_CStep;`

Pour le moment, ces attributs ont été déclarés dans notre classe *CStep* mais ils ne sont pas accessibles depuis l'extérieur (Il est impossible d'évaluer l'expression `Etape1.m_xActivityBit`). Ces variables sont "coincées" à l'intérieur de notre bloc fonction. Pour les rendre accessibles, nous allons utiliser l'objet **PROPERTY** proposé par CodeSys.

## 2.2 L'objet PROPERTY (propriété)

L'objet **PROPERTY** est un outil proposé par CodeSys pour la programmation orientée objet. Il permet de définir des attributs (variables) propres à un contexte avec un niveau d'accessibilité paramétrable (publique, privée, protégée ou interne).

Cela signifie que l'on peut définir des variables à l'intérieur d'une classe qui pourront être accessibles depuis l'extérieur de la classe par l'intermédiaire de méthodes appelées *accesseurs*.



### Les accesseurs

Il existe deux types d'accesseurs : GET et SET.

**GET** : Permet de lire la valeur d'une variable

**SET** : Permet d'écrire la valeur d'une variable

```
1 C<ClassName>.<propertyName>.Get
2   <propertyName> := THIS^.m_<
3   localName>;
```

```
1 C<ClassName>.<propertyName>.Set
2   THIS^.m_<localName> := <
3   propertyName>;
```

Il est possible de supprimer un des accesseurs pour rendre la propriété en lecture seule ou en écriture seule.

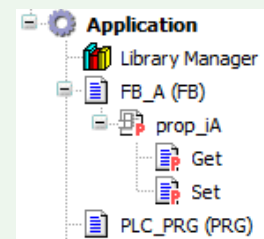
Lors d'une dérivation, les propriétés de la classe sont héritées et leurs accesseurs peuvent être surchargés.

Une propriété est donc un objet que l'on associe à une POU. On décrit ensuite ses accesseurs pour définir comment lire ou écrire la valeur de la propriété. Ce sont elles qui permettent d'obtenir une encapsulation des données. En effet, elles permettent de contrôler l'accès aux données de l'objet (variables internes, d'un POU, d'un BF ou d'une variable globale – GVL). Elles agissent comme un filtre ou une surcharge.



### Ajout d'un propriété sous CodeSys

Sous CodeSys, les propriétés et les méthodes sont ajoutés à la POU en utilisant l'outil *Add Object*. Il suffit alors de sélectionner le type de l'objet à ajouter (propriété ou méthode) et de le nommer.



### Propriétés de la classe CStep

Par exemple, si notre classe *CStep* possède un attribut *m\_xActivityBit*, nous allons ajouter une propriété *xActivityBit* à notre bloc fonctionnel *FB\_CStep*. Cette propriété, en lecture seule, n'aura pas d'accesseur *SET*. Son accesseur *GET* sera défini de la manière suivante :

```
1 FB_CStep.xActivityBit.Get
2   xActivityBit := THIS^.m_xActivityBit;
3
```



### Les attributs d'accessibilité

Les méthodes et les attributs sont associées à un attribut d'accessibilité. Cela permet de contrôler l'accès aux méthodes et aux attributs de la classe :

**PRIVATE** : Accessibilité réduite à l'entité à laquelle elle est associée.

**PROTECTED** : Accessibilité réduite à l'entité à laquelle elle est associée et à ses dérivées.

**PUBLIC** : Accessibilité à toutes les entités.

**INTERNAL** : Accessibilité réduite à l'espace de noms de la bibliothèque

## 2.3 Méthodes

L'objet **METHOD** est un outil proposé par CodeSys pour la programmation orientée objet. Il permet de définir des méthodes (fonctions) propres à un contexte avec un niveau d'accessibilité paramétrable (publique, privée, protégée ou interne).

Les avantages d'utiliser des méthodes au sein d'une classe sont les suivants :

- La modularité des blocs fonctionnels est favorisée
  - ◊ On pourra décomposer le corps d'un bloc fonctionnel en plusieurs appels de méthodes
- La lisibilité du code est améliorée
- L'intégrité des données est favorisée par l'utilisation judicieuse d'attributs d'accessibilité

Comme pour une fonction, on peut définir :

- Des variables d'Entrées, de sorties ou d'entrées-sorties
- Des variables locales à la méthode
  - ◊ La durée de vie de ces variables locales est alors limitée à l'exécution de la méthode

Par ailleurs, **une méthode a accès au contexte de l'instance à laquelle elle est associée**. Cela signifie qu'elle a accès aux variables et méthodes de l'objet dont elle dépend.

Enfin, CodeSys autorise la déclaration de **METHODS** dans une interface. Elle devra alors être implémentée dans les classes qui implémentent cette interface.



### À retenir

Les méthodes des classes que nous définirons seront des méthodes associées au bloc fonction concerné.

Leur ligne de déclaration est la suivante : **METHOD** <Attribut> <Method\_Name> | : <return\_data\_type>



### Les méthodes de la classe CStep

Les méthodes de la classe *CStep* seront définies dans le bloc fonctionnel *FB\_CStep*. Elles seront donc des méthodes associées à ce bloc fonctionnel.

#### Méthodes

- **MInit** : Méthode appelée à l'initialisation du programme
- **MActivate** : active l'étape
- **MDeactivate** : désactive l'étape

```

1 METHOD PUBLIC MInit : BOOL
2   VAR_INPUT
3   END_VAR
4   VAR_OUTPUT
5   END_VAR
6   VAR
7   END_VAR
8   // Code de la méthode
9   // ...
10  // Fin du code de la méthode
11 END_METHOD
12
```

## 2.4 Les interfaces

Une interface est un ensemble de méthodes et de propriétés qui définissent un contrat. Une classe qui implémente une interface doit alors implémenter toutes les méthodes et propriétés de cette interface. Cela permet de définir une structure très générique qui sera utilisée, ensuite, par les classes que nous définirons.



### À retenir

Les interfaces permettent de définir un contrat que les classes qui les implémentent doivent respecter. La déclaration d'une interface se fait de manière similaire à la déclaration d'une classe, mais aucune méthode n'est implémentée.

### Activité 1: Exemple d'interface

**Question 1** Donner un exemple dans lequel l'utilisation d'une interface est pertinente, par exemple dans le monde du jeu vidéo. Expliquer

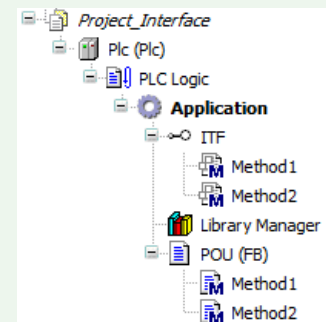


### Les interfaces sous CodeSys

Sous CodeSys, une interface s'ajoute dans l'arborescence du projet en utilisant l'outil *Add Object*. Il suffit alors de sélectionner le type de l'objet à ajouter (interface) et de le nommer. Elle apparaît alors ainsi que les méthodes et propriétés qui lui sont associées.

Le mot clef utilisé sous CodeSys est **INTERFACE**.

Le mot clef **IMPLEMENT** est utilisé lors de la déclaration d'une classe pour indiquer que cette classe implémente une interface.



**Dans notre contexte** les interfaces pourront définir, pour un ensemble de blocs fonctionnels :

- Des propriétés (**PROPERTY**)
  - ◊ Nom
  - ◊ Type
  - ◊ Accesseurs
- Des méthodes (**METHOD**)
  - ◊ Nom
  - ◊ Paramètres d'entrées et de sorties (**VAR\_INPUT**, **VAR\_OUTPUT**, **VAR\_IN\_OUT**)



### Plusieurs interfaces

Une classe peut implémenter plusieurs interfaces. Elle devra alors implémenter toutes les méthodes et propriétés de ces interfaces.

```
1 | FUNCTION_BLOCK <fb_name> IMPLEMENTS <ITF_name_0> |, <ITF_name_1> |, <ITF_name_n>
2 |
```

## 2.5 Les pointeurs spéciaux

CodeSys définit deux vecteurs spéciaux **THIS** et **SUPER**. Leur utilisation est réservée au contexte de la POO, dans le corps d'un bloc fonctionnel ou dans les méthodes associées.

### 2.5.1 le pointeur THIS

Le pointeur `this` est un pointeur spécial qui pointe vers l'objet auquel appartient la méthode. Il est automatiquement disponible dans tous les blocs fonctions.

Les utilisations principales de `THIS` sont données ci-dessous, les exemples sont extraits de la documentation en ligne de CodeSys.

**Démasquage d'une propriété ou d'une méthode :** Par exemple, lorsqu'une variable locale a le même nom qu'une propriété de l'objet, il est nécessaire de démasquer l'objet pour accéder à la propriété.

```

1 FUNCTION_BLOCK fbA
2 VAR_INPUT
3   iVarA: INT;
4 END_VAR
5 iVarA := 1;
6
7 FUNCTION_BLOCK fbB EXTENDS fbA
8 VAR_INPUT
9   iVarB: INT := 0;
10 END_VAR
11 iVarA := 11;
12 iVarB := 2;
13
14 METHOD DoIt : BOOL
15 VAR_INPUT

```

```

16 END_VAR
17 VAR
18     iVarB: INT;
19 END_VAR
20 iVarB := 22;           // The local variable iVarB is set.
21 THIS~.iVarB := 222;    // The function block variable
                        // iVarB is set even though iVarB is obscured.
22
23 PROGRAM PLC_PRG
24 VAR
25     MyfbB: fbB;
26 END_VAR
27
28 MyfbB(iVarA:=0, iVarB:= 0);
29 MyfbB.DoIt();

```

**Désigner l'objet tout entier** lors d'un passage d'objet en paramètre d'une méthode.

```

1 FUNCTION funA
2   VAR_INPUT
3     pFB: fbA; //La fonction A prend en parametre d'entree
4               //un objet issu de la classe fbA
5   END_VAR
6   ...;
7
8 FUNCTION_BLOCK fbA
9   VAR_INPUT
10    iVarA: INT;
11  END_VAR
12  ...;
13
14 FUNCTION_BLOCK fbB EXTENDS fbA
15   VAR_INPUT
16    iVarB: INT := 0;
17  END_VAR
18  iVarA := 11;

```

```

19 | iVarB := 2;
20 |
21 | METHOD DoIt : BOOL
22 | VAR_INPUT
23 | END_VAR
24 | VAR
25 | iVarB: INT;
26 | END_VAR
27 | iVarB := 22;
28 | funA(pFB := THIS~); //On donne l'objet appelant en
    parametre
29 |                               //de la fonction funA
30 | PROGRAM PLC_PRG
31 | VAR
32 | MyfbB: fbB;
33 | END_VAR
34 | MyfbB(iVarA:=0 , iVarB:= 0);
35 | MyfbB.DoIt();

```

### 2.5.2 Le pointeur SUPER

Le pointeur `SUPER` est un pointeur spécial qui pointe vers l'objet parent de l'objet auquel appartient la méthode. Il est automatiquement disponible dans tous les blocs fonctions. Par définition, son utilisation est réservée aux blocs fonctionnels obtenus par dérivation.

Ce pointeur permet d'accéder aux méthodes et propriétés de l'objet parent, notamment quand celles-ci ont été surchargées dans les blocs fonctionnels dérivés.

Le code suivant, issu de la documentation en ligne de CodeSys, montre quelques exemples d'utilisation des pointeurs `SUPER` et `THIS`.

```
1 FUNCTION_BLOCK FB_1 EXTENDS FB_Base
2 VAR_OUTPUT
3     iBase : INT;
4 END_VAR
5
6 THIS^.METHOD_DoIt(); //Call of the methods of FB_1
7 THIS^.METHOD_DoAlso();
```

```

9  SUPER^.METH_DoIt();    //Call of the methods of FB_Base
10 SUPER^.METH_DoAlso();
11 iBase := SUPER^.iCnt;
12
13 METHOD METH_DoIt : BOOL
14 iCnt := 1111;
15 METH_DoIt := TRUE;
16

```

```

17 PROGRAM PLC_PRG
18 VAR
19     myBase : FB_Base;
20     myFB_1 : FB_1;
21     iTHIS : INT;
22     iBase : INT;
23 END_VAR
24 myBase();
25 iBase := myBase.iCnt;
26 myFB_1();
27 iTHIS := myFB_1.iCnt;
28
29 FUNCTION_BLOCK FB_Base
30 VAR_OUTPUT
31     iCnt : INT;
32 END_VAR
33
34 METHOD METH_DoIt : BOOL
35     iCnt := -1;
36
37 METHOD METH_DoAlso : BOOL
38     METH_DoAlso := TRUE;
39
40 FUNCTION_BLOCK FB_1 EXTENDS FB_Base
41 VAR_OUTPUT
42     iBase : INT;

```

```

43 END_VAR
44
45 THIS^.METH_DoIt(); //Call of the methods of FB_1
46 THIS^.METH_DoAlso();
47
48 SUPER^.METH_DoIt(); //Call of the methods of FB_Base
49 SUPER^.METH_DoAlso();
50 iBase := SUPER.iCnt;
51
52 METHOD METH_DoIt : BOOL
53     iCnt := 1111;
54     METH_DoIt := TRUE;
55
56 PROGRAM PLC_PRG
57 VAR
58     myBase : FB_Base;
59     myFB_1 : FB_1;
60     iTHIS : INT;
61     iBase : INT;
62 END_VAR
63
64 myBase();
65 iBase := myBase.iCnt;
66 myFB_1();
67 iTHIS := myFB_1.iCnt;

```

## 2.6 Construction des objets

La construction d'un objet consiste à lui donner des valeurs spécifiques d'initialisation pour chacun de ses membres. Lors de l'instanciation d'un bloc fonction, CodeSys appelle automatiquement, et en premier lieu, la méthode `FBInit` de l'objet. Cette méthode est créée implicitement et peut être surchargée pour définir une initialisation particulière.

Son prototype est le suivant :

```

1 METHOD FB_Init : BOOL
2 VAR_INPUT
3     bInitRetains : BOOL;
4     bInCopyCode : BOOL;
5 END_VAR

```



### À retenir

- La méthode `FBInit` est définie implicitement lors de la déclaration d'un bloc fonction.
- Elle peut être surchargée pour définir une initialisation particulière.
- Elle est appelée la première et automatiquement lors de l'instanciation d'un bloc fonction.

## 2.7 Mise en oeuvre

Cette section propose un exemple de mise en oeuvre de la programmation orientée objet sous CodeSys. Nous allons construire une classe *CStep* qui représente une étape d'un grafset. Cette classe implémentera une interface *ITF\_ObjStep* qui définira les méthodes et attributs communs à toutes les étapes :

- **Attributs**
  - ◊ **m\_xActivityBit** : bit d'activité (actif ou non)
    - Accessible en lecture uniquement
  - ◊ **m\_tActivationDuration** : Stocke le temps d'activation de l'étape
    - Accessible en lecture uniquement
  - ◊ **m\_tTaskPeriod** : période de la tâche
    - Accessible en lecture et en écriture
- **Méthodes**
  - ◊ **MInit** : Méthode appelée à l'initialisation du programme
  - ◊ **MActivate** : active l'étape



◇ **MDeactivate** : désactive l'étape

### Activité 2: L'interface ITF\_ObjStep

**Question 2** Compléter la liste des propriété et méthodes de l'interface `ITF_ObjStep`.

```
INTERFACE ITF_ObjStep
  PROPERTY xActivityBit : BOOL (*Accesseur Get*)
  PROPERTY .....
  PROPERTY .....
  ...
  METHOD MInit;
  METHOD .....
  METHOD .....
```

**Activité 3: Le bloc fonctionnel CStep**

**Question 3** Compléter la déclaration du bloc fonctionnel `cstep` qui implémente l'interface `ITF_ObjStep`. Elle contiendra les variables locales associées aux propriétés de l'interface.

```

1 FUNCTION_BLOCK .....
2 VAR
3     .....
4     .....
5     .....
6 END_VAR

```

**Question 4** Ajouter des lignes permettant la mise à jour de la durée d'activation lorsque l'étape est active – Si l'étape est active, la durée d'activation augmente de la période de la tâche.

```

1 .....
2 .....
3 .....
4 .....

```

**Question 5** Définir les accesseurs pour les propriétés de l'interface.


```

CStep.xActivityBit.Get
.....
CStep.....
.....
CStep.....
.....
CStep.Set (*Limiter les valeurs admissible a [ T#1ms, T#10s]*)
.....
.....
.....
.....

```



**Activité 4**

**Question 6** Surcharger le constructeur de la classe `CStep` pour initialiser la période de la tâche à `T#1ms`, le bit d'activité à `FALSE` et la durée d'activation à `T#0s`.

```
 METHOD .....  
.....  
.....  
.....  
.....  
.....  
.....  
.....  
.....  
.....
```

**Activité 5**

**Question 7** Définir les méthodes `MInit`, `MActivate` et `MDeactivate` de la classe `CStep`.

```
 METHOD CStep.MInit  
.....  
.....  
.....  
 METHOD CStep.MActivate  
.....  
.....  
.....  
.....  
.....  
 METHOD CStep.MDeactivate  
.....
```

**Activité 6: Utilisation**

**Question 8** Compléter le programme suivant pour utiliser la classe `cstep`. Le programme activera l'étape au premier cycle et la désactivera au bout de 30 s.

```
PROGRAM UserCase
```

```
VAR
```

```
oStep : CStep := (tTaskPeriod := T#10ms);
```

```
itfStep : ITF_ObjStep := oStep;
```

```
pStep : POINTER TO CStep := ADR(oStep);
```

```
xFirstCycleRun : BOOL := TRUE;
```

```
END_VAR
```

```
IF (xFirstCycleRun) THEN
```

```
.....  
    xFirstCycleRun := FALSE;
```

```
END_IF
```

```
.....  
.....  
.....  
.....
```