**Programmazione Orientata**

**agli Oggetti OOP IEC61131-3**

**Corso Youtube by Runtimevic**

**Programmazione Orientata agli Oggetti OOP IEC61131-3 PLC Corso Youtube**

**by Runtimevic.**

*runtimevic*

*Copyright © 2023 Víctor Durán.*

Table of contents

# Table of contents

1. Requisiti 4

|  |  |
| --- | --- |
| 2. Introduzione | 5 |
| 3. Tipi di paradigma | 7 |
| 4. Classe e Oggetti | 9 |
| 4.1 Classe e Oggetti | 9 |
| 4.2 Blocchi Funzione | 11 |
| 4.3 Oggetto Metodo | 16 |
| 4.4 Oggetto Propietà | 21 |
| 4.5 Eredità | 23 |
| 4.6 Puntatore THIS | 28 |
| 4.7 Puntatore SUPER | 31 |
| 4.8 Interfacce | 33 |
| 4.9 puntatore e riferimento | 35 |
| 4.10 FB astrazione di una interfaccia | 44 |
| 5. Tabella di Modificatori di accesso | 46 |
| 6. Tipi di variabili e variabili speciali | 47 |
| 7. Tipi di Dati | 51 |
| 7.1 Dichiarazione di una Variabile: | 51 |
| 8. Interfaccia fluida | 53 |
| 9. Principi OOP | 54 |
| 9.1 4 Pilastri | 54 |
| 9.2 Astrazione | 55 |
| 9.3 Incapsulamento | 56 |
| 9.4 Eredità | 57 |
| 9.5 Polimorfismo | 58 |
| 10. Relazioni | 59 |
| 11. SOLID | 60 |
| 11.1 SOLID | 60 |
| 11.2 Principio di Responsabilita Unica | 61 |
| 11.3 Principio di Aperto/Chiuso | 62 |
| 11.4 Principio di Sostituzione di Liskov | 63 |
| 11.5 Principio di Segregazione de Interfaccia | 64 |
| 11.6 Principio di Inversione de Dipendenza | 65 |
| 12. UML | 66 |
| 12.1 UML | 66 |

Table of contents

12.2 Class UML 67

|  |  |
| --- | --- |
| 12.3 StateChart UML | 68 |
| 13. Tipi di modelli per programmazione di PLC | 69 |
| 14. Modelli di progettazione | 70 |
| 14.1 Modelli di progettazione | 70 |
| 14.2 Modello di strategia | 76 |
| 15. Links | 77 |

1. Requisitos

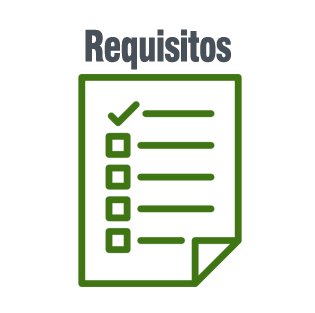
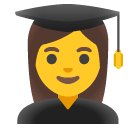
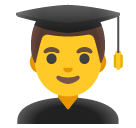
# 1. Requisiti

👨‍🎓

Requisiti

👩‍🎓

:



I requisiti necessari per seguire questo corso, sono l’utilizzo dei seguenti software:

* 🔗 [Beckhoff TwinCAT 3 XAE](https://www.beckhoff.com/en-en/products/automation/twincat/texxxx-twincat-3-engineering/te1000.html) o IDE di 🔗 [Codesys.](https://store.codesys.com/de/)
* Utente per accesso a 🔗 [GitHub.](https://github.com/)
* conoscere il minimo di Git o affidarsi a strumenti visivi come:
* 🔗 [GitHub Desktop.](https://desktop.github.com/)
* 🔗 [sourcetree](https://www.sourcetreeapp.com/)
* 🔗 [tortoiseGit](https://tortoisegit.org/), etc...
* Sarebbe consigliato avere una base di conoscenza di teoria di OOP, anche se derivata da altri linguaggi di programmazione, dato che questo corso si focalizerà proprio sul paradigma OOP IEC61131-3 per PLC.

1.0.1 Prima di iniziare:

Clonare il reposority di

[🔗](https://github.com/runtimevic/OOP-IEC61131-3--Curso-Youtube.git)



[GitHub](https://github.com/runtimevic/OOP-IEC61131-3--Curso-Youtube.git)

[:](https://github.com/runtimevic/OOP-IEC61131-3--Curso-Youtube.git)

•

$ git clone https://github.com/runtimevic/OOP-IEC61131-3--Curso- Youtube.git o utilizzare GitHub Desktop per clonare il reposority di GitHub...

* Cartelle all’interno del reposority:
* 🔗 [TC3\_OOP:](https://github.com/runtimevic/OOP-IEC61131-3--Curso-Youtube/tree/master/TC3_OOP) Qui dentro potete trovare il progetto di TwinCAT3, con tutto quello che viene spiegato nei vari video tutorial di youtube...

🔗 [Ficheros\_PLCOpen\_XML:](https://github.com/runtimevic/OOP-IEC61131-3--Curso-Youtube/tree/master/Ficheros_PLCOpen_XML) All'interno di questa cartella troveremo i file esportati in formato PLCOpen XML in modo che possano essere importati in TwinCAT3 o in Codesys, il tutto viene spiegato nei vari video su YouTube, poiché è il formato PLCOpen standardizzato, può essere esportato/importato in diverse piattaforme PLC che lavoranocon lo standard PLCOpen..., ma è consigliabile provare a fare quanto spiegato da zero per esercitarsi e assumere i concetti...

• La creazione di questa pagina web SSG, (Static Site Generator) verrà modificata man mano che avanziamo in questo corso OOP IEC-61131-3 PLC...

2. Introduccion

# 2. Introduzione



2.0.1 Cosa significa OOP?

* E’ un paradigma che fa uso di oggetti per la costruzione dei software.

. Che è un paradigma?

* A diverse interpretazioni, può essere un **modello**, **esempio** o **patrón.**
* E’ una forma di programmare.
* • cerca di tradurre la realtà nel codice.

2.0.2 Come pensare in Oggetti?

* Concentrati su qualcosa di reale**.**
* Descrive in dettaglio i suoi attributi, (proprietà)

2.0.3 Links: • Descrive in dettaglio i loro comportamenti (metodi)

2.0.3 Links:

🔗



[Codesys ammete OO](https://help.codesys.com/api-content/2/codesys/3.5.14.0/en/_cds_f_object_oriented_programming/#cd69b36a931eb47c0a8640e01dc4fc9-id-8938ae329606a7abc0a8640e00eabd77)P

🔗



[Beckhoff TwinCAT 3 ammete OO](https://infosys.beckhoff.com/english.php?content=../content/1033/tc3_plc_intro/2527303819.html&id=)P

•

•

3. Tipos de paradigmas

# 3. Tipi di paradigma

3.0.1 Tipi di paradigma:

• Imperativo -- (Istruzioni da seguire per risolvere un problema).

• Dichiarativo -- (Si concentra sul problema da risolvere).

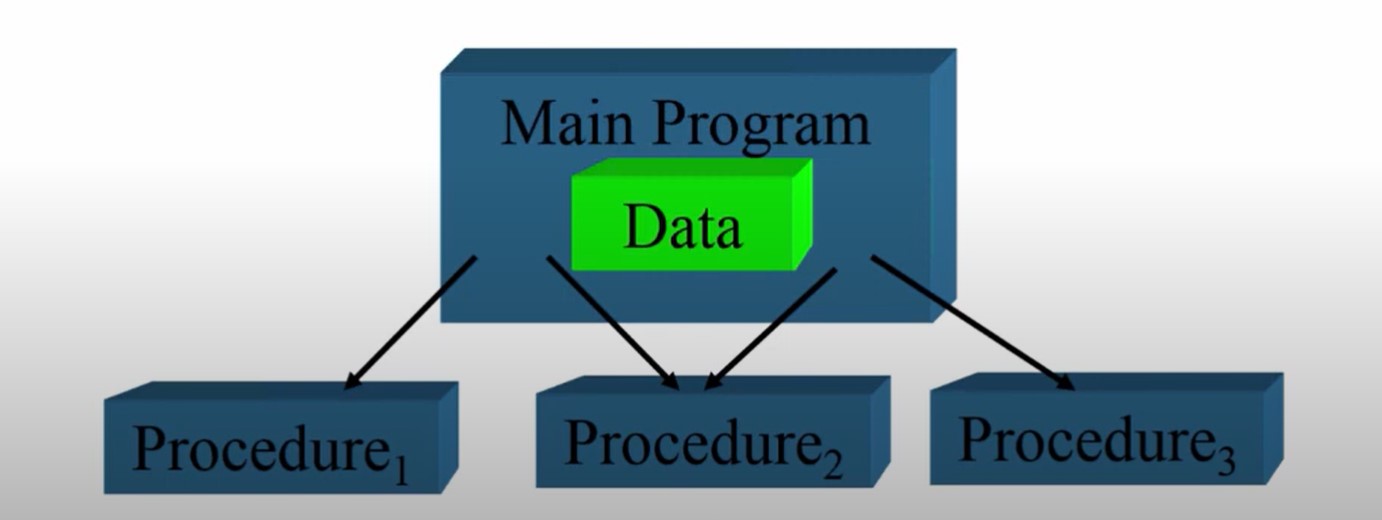
• Strutturato -- (La soluzione a un problema segue una sequenza dall'inizio alla fine).

• Funzionale -- (Divide il problema in diverse soluzioni che verranno eseguite dalle funzioni dichiarate). La programmazione procedurale o la programmazione procedurale è un paradigma di programmazione. Molte volte è applicabile sia nei linguaggi di programmazione di basso livello che nei linguaggi di alto livello. Nel caso in cui questa tecnica venga applicata in linguaggi di alto livello, riceverà il nome di programmazione funzionale.

• routine separate vengono chiamate dal programma principale

• dati per lo più globali -> nessuna protezione.

• le procedure generalmente non possono essere indipendenti -> scarso riutilizzo del codice.



* Orientada a oggetti – Costruire soluzioni basate su oggetti

1

2

3

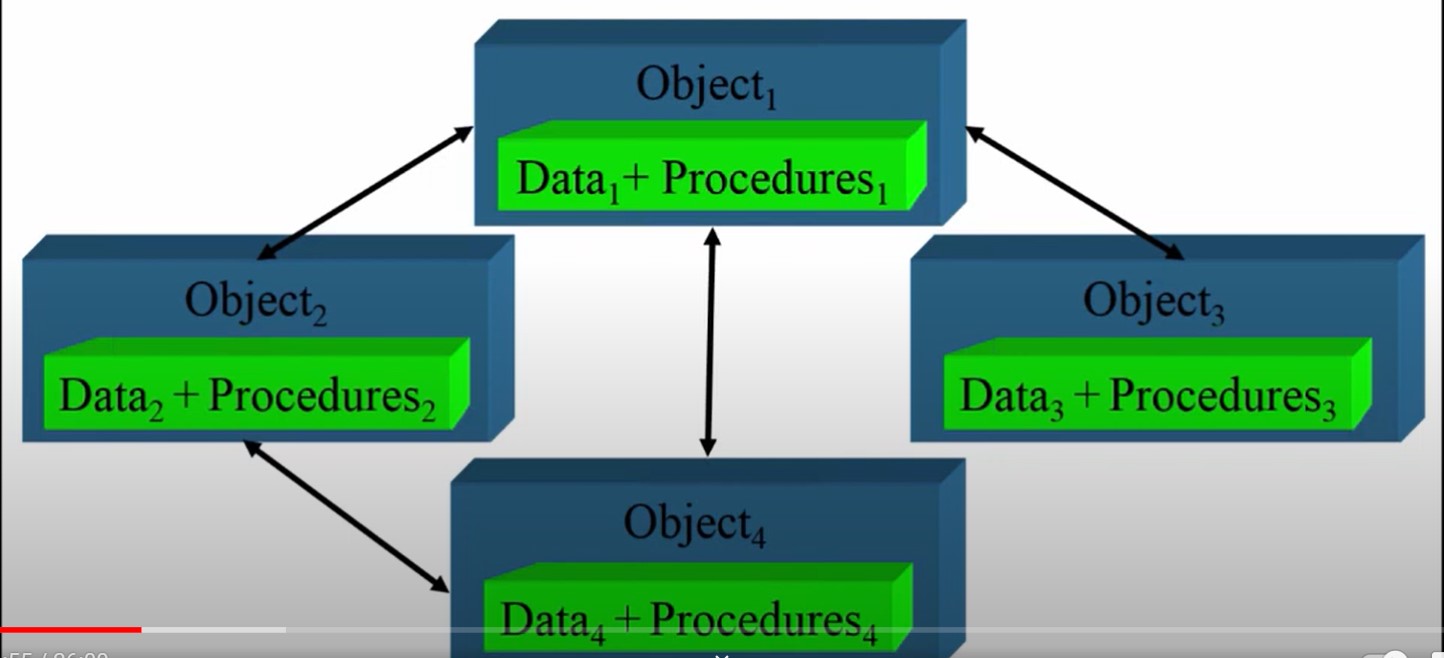
4

wikipedia:

La programación orientada a objetos es un paradigma de programación

basado en el concepto de "objetos", que pueden contener datos y código.

Los datos están en forma de campos y el código está en forma de procedimientos.



3.0.2 Vantaggi della programmazione OOP:

• routine e dati sono combinati in un oggetto -> Incapsulazione.

• metodi/Proprietà -> interfacce definite per le chiamate e l'accesso ai dati.

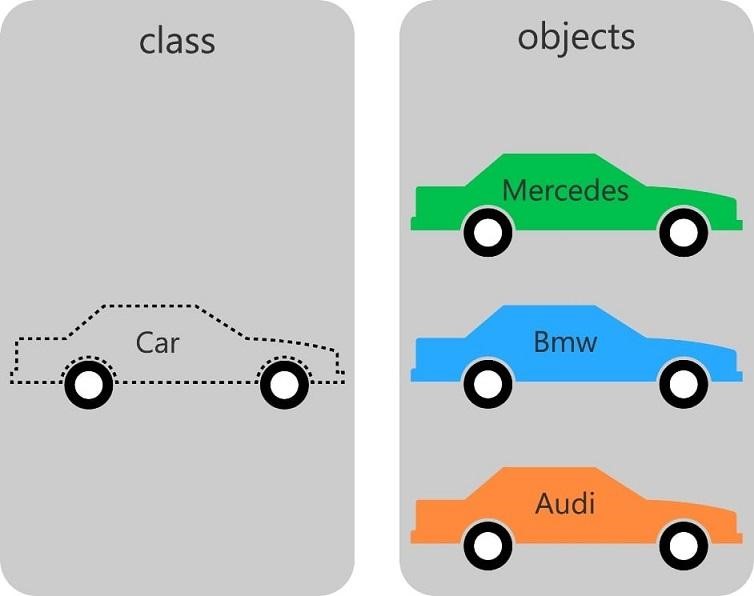
4. Clases y Objetos

# 4. Classi e Oggetti

## 4.1 Classi e Oggetti

**Classi y Oggetti:**

* Una Classe è una **modello**.
* Un Oggetto è una **istanza di una Classe**.



1

2

3

En este Ejemplo Nos encontramos la Clase Coche,

y hemos instanciado esta Clase para tener los Objetos de Coches

Mercedes, Bmw y Audi...

4.1 Classe y Oggetto

**Rappresentazione di una classe Auto in STL OOP IEC 61131-3**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

FUNCTION\_BLOCK Auto

VAR\_INPUT

END\_VAR

VAR\_OUTPUT

END\_VAR

VAR

\_Marca : STRING;

\_Colore : STRING;

\_Azione : STRING;

END\_VAR

----------------------------------------------------------------

METHOD PUBLIC Accellerare

azione := 'accellerare';

----------------------------------------------------------------

METHOD PUBLIC Guidare

azione := 'guidare';

----------------------------------------------------------------

METHOD PUBLIC Frenare

azione := 'frenare';

----------------------------------------------------------------

PROPERTY PUBLIC Colore : STRING

Get

Colore := \_Colore;

Set

\_Colore := Colore;

----------------------------------------------------------------

PROPERTY PUBLIC Marca : STRING

Get

Marca := \_Marca;

Set

\_Marca := Marca;

Istanza della classe negli Oggetti: Mercedes, BMW e Audi e richiami ai loro metodi e proprietà...

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

PROGRAM \_01\_Clase\_y\_Objetos

VAR

// tenemos la Clase Coche y la instanciamos y obtenemos los Objetos: Mercedes, Bmw y Audi.

Mercedes : Coche;

Bmw : Coche;

Audi: Coche;

Color : STRING;

Marca : STRING;

Acelerar : BOOL;

Conducir: BOOL;

Frenar : BOOL;

END\_VAR

//Objeto Mercedes

//llamadas a sus métodos.

IF Acelerar THEN

Mercedes.Acelerar();

Acelerar := FALSE;

END\_IF

IF Conducir THEN

Mercedes.Conducir();

Conducir := FALSE;

END\_IF

IF Frenar THEN

Mercedes.Frenar();

Frenar := FALSE;

END\_IF

//llamadas a sus propiedades.

Mercedes.Marca := 'Mercedes';

Mercedes.Color := 'Negro';

Color := Mercedes.Color;

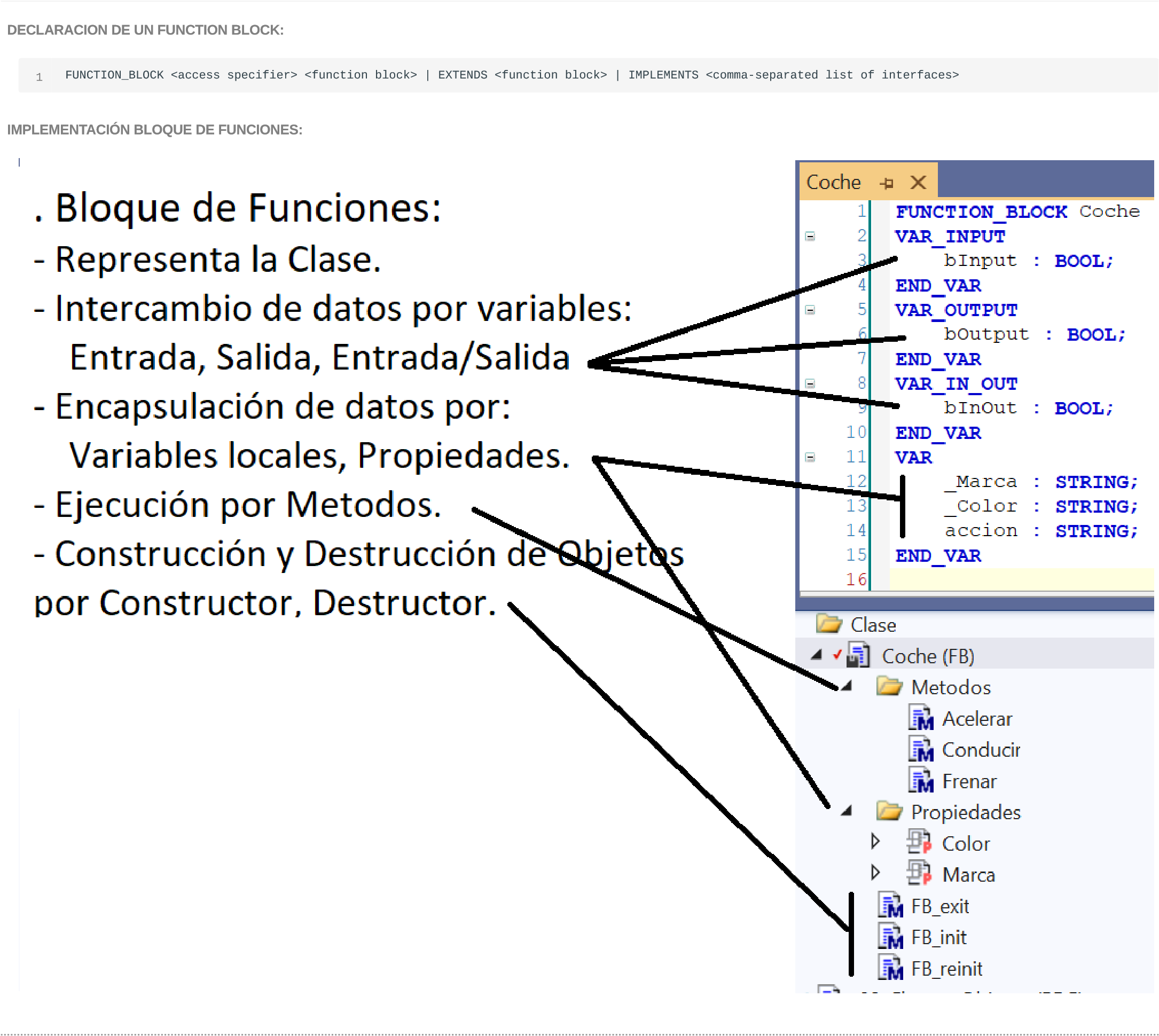
**Links:**

* 🔗 [methods-properties-and-inheritance (stefanhenneken)](https://stefanhenneken.net/2017/04/23/iec-61131-3-methods-properties-and-inheritance/)

4.2 Bloque de Funciones

## 4.2 Blocchi Funzione

### 4.2.1 Blocchi Funzione



EXTENDS: - Se nella dichiarazione di un FUNCTION\_BLOCK aggiungiamo la parola EXTENDS seguita dal nome dell'FB da cui vogliamo ereditare, significa che ne ereditiamo tutti i metodi e le proprietà (principio di ereditarietà) - Un FB può solo ereditare da una classe FB.

IMPLEMENTS: - Se nella dichiarazione di un FUNCTION\_BLOCK aggiungiamo la parola IMPLEMENTS seguita dal nome dell'interfaccia o delle interfacce separate da virgole. - Se nell'FB è implementata un'interfaccia, è obbligatorio creare nell'FB la programmazione dei metodi e delle proprietà dell'interfaccia implementata.

4.2.1 Blocchi Funzione



• Esempio dichiarazione di un FUNCTION\_BLOCK:

4.2.2 Modificatori di accesso ai blocchi funzione

### 4.2.2 Modificatori di accesso ai blocchi funzione

**Modificatori di accesso ai blocchi funzione:**

* Possiamo avere 2 modificatori di accesso per il blocco funzione:
* PUBLIC:

• Non ci sono restrizioni, puoi chiamare da qualsiasi luogo.

• Se non mettiamo nulla quando dichiariamo l'FB di default viene impostato PUBLIC.

• Chiunque può chiamare o creare un'istanza dell'FB.

• Può essere utilizzato per l'ereditarietà essendo pubblico.

• Sono accessibili dopo aver istanziato la classe.

• Corrisponde alla specifica del modificatore di accesso illimitato.

INTERNAL:

* È possibile accedere all'FB solo dallo stesso spazio dei nomi.
* Ciò consente all'FB di essere disponibile solo all'interno di una determinata libreria. La configurazione predefinita in cui non è definito alcun identificatore di accesso è PUBLIC .

• L'accesso è limitato allo spazio dei nomi (la libreria).

Possiamo avere altri 2 modificatori di accesso per il blocco funzione: -

FINAL: - (In TwinCAT 3 non deve essere selezionato per impostazione predefinita durante la creazione di un FB, ma può essere aggiunto successivamente dopo averlo creato...) - L'FB non può fungere da blocco funzione principale. - I metodi e le proprietà di questo POU non possono essere ereditati. - FINAL è consentito solo per POU di tipo FUNCTION\_BLOCK. - Non è consentita la sovrascrittura, in una derivata del blocco funzione. - Ciò significa che non può essere sovrascritto/esteso su una sottoclasse eventualmente esistente.

* ABSTRACT: blocchi funzione astratti

1

FUNCTION\_BLOCK PUBLIC ABSTRACT FB\_Foo

* Non appena un metodo o una proprietà viene dichiarato astratto, anche il blocco funzione deve essere dichiarato astratto. - Non è possibile creare istanze da FB astratti. Gli FB astratti possono essere utilizzati solo come FB di base se ereditati. - Tutti i metodi astratti e tutte le proprietà astratte devono essere sovrascritti per creare un FB non astratto. Un metodo astratto o una proprietà astratta viene convertito in un metodo non astratto o in una proprietà non astratta durante l'override. - I blocchi funzione astratti possono anche contenere metodi non astratti e/o proprietà non astratte. - Se durante l'ereditarietà non vengono sovrascritti tutti i metodi astratti o tutte le proprietà astratte, l'FB ereditato può essere solo un FB astratto (implementazione passo passo). - Sono consentiti puntatori o riferimenti di tipo FB astratto. Tuttavia, questi possono fare riferimento a FB non astratti e quindi chiamare i loro metodi o proprietà (polimorfismo).

4.2.3 Dichiarazione delle variabili del blocco funzione

### 4.2.3 Dichiarazione delle variabili del blocco funzione

**TIPI DI VARIABILI CHE POSSONO ESSERE DICHIARATI IN UN BLOCCO\_FUNZIONE:**

🔗



[Local Variables - VA](https://infosys.beckhoff.com/content/1033/tc3_plc_intro/2528755083.html)

[R](https://infosys.beckhoff.com/content/1033/tc3_plc_intro/2528755083.html)

🔗



[Input Variables - VAR\_INPU](https://infosys.beckhoff.com/content/1033/tc3_plc_intro/2528760459.html)

[T](https://infosys.beckhoff.com/content/1033/tc3_plc_intro/2528760459.html)

🔗



[T](https://infosys.beckhoff.com/content/1033/tc3_plc_intro/2528765835.html)

[Output Variables - VAR\_OUTPU](https://infosys.beckhoff.com/content/1033/tc3_plc_intro/2528765835.html)

🔗



[T](https://infosys.beckhoff.com/content/1033/tc3_plc_intro/2528771211.html)

[Input/Output Variables - VAR\_IN\_OUT, VAR\_IN\_OUT CONSTAN](https://infosys.beckhoff.com/content/1033/tc3_plc_intro/2528771211.html)

🔗



[P](https://infosys.beckhoff.com/content/1033/tc3_plc_intro/2528781963.html)

[Temporary Variable - VAR\_TEM](https://infosys.beckhoff.com/content/1033/tc3_plc_intro/2528781963.html)

🔗



[T](https://infosys.beckhoff.com/content/1033/tc3_plc_intro/2528787339.html)

[Static Variables - VAR\_STA](https://infosys.beckhoff.com/content/1033/tc3_plc_intro/2528787339.html)

🔗



[External Variables - VAR\_EXTERNA](https://infosys.beckhoff.com/content/1033/tc3_plc_intro/2528792715.html)

[L](https://infosys.beckhoff.com/content/1033/tc3_plc_intro/2528792715.html)

🔗



[~~Instance Variables - VAR\_IN~~](https://infosys.beckhoff.com/content/1033/tc3_plc_intro/2528798091.html)

[~~S~~](https://infosys.beckhoff.com/content/1033/tc3_plc_intro/2528798091.html)

[~~T~~](https://infosys.beckhoff.com/content/1033/tc3_plc_intro/2528798091.html)

🔗



[Remanent Variables - PERSISTENT, RETAI](https://infosys.beckhoff.com/content/1033/tc3_plc_intro/2528803467.html)

[N](https://infosys.beckhoff.com/content/1033/tc3_plc_intro/2528803467.html)

🔗



[SUPE](https://infosys.beckhoff.com/content/1033/tc3_plc_intro/2528837771.html)

[R](https://infosys.beckhoff.com/content/1033/tc3_plc_intro/2528837771.html)

🔗



[THI](https://infosys.beckhoff.com/content/1033/tc3_plc_intro/2528843147.html)

[S](https://infosys.beckhoff.com/content/1033/tc3_plc_intro/2528843147.html)

🔗



[Variable types - attribute keyword](https://infosys.beckhoff.com/content/1033/tc3_plc_intro/2528848523.html)

[s](https://infosys.beckhoff.com/content/1033/tc3_plc_intro/2528848523.html)

🔗



[N](https://infosys.beckhoff.com/content/1033/tc3_plc_intro/2528803467.html)

[RETAIN: for remanent variables of type RETAI](https://infosys.beckhoff.com/content/1033/tc3_plc_intro/2528803467.html)

🔗



[PERSISTENT: for remanent variables of type PERSISTEN](https://infosys.beckhoff.com/content/1033/tc3_plc_intro/2528803467.html)

[T](https://infosys.beckhoff.com/content/1033/tc3_plc_intro/2528803467.html)

🔗



[CONSTANT: for constant](https://infosys.beckhoff.com/content/1033/tc3_plc_intro/2529284235.html#2529371275)

[s](https://infosys.beckhoff.com/content/1033/tc3_plc_intro/2529284235.html#2529371275)

•

•

•

•

•

•

•

•

•

•

•

•

•

•

•

• Questi tipi di variabili che sono dichiarabili all'interno dell'FB, possono ripetersi all'interno dell'FB, questo potrebbe essere valido per differenziare variabili dello stesso tipo nella zona di dichiarazione, sarebbe puramente indicativo...

• Esempio di dichiarazione di variabili in un FUNCTION\_BLOCK:

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

FUNCTION\_BLOCK fb\_tipos\_de\_datos

VAR\_INPUT

binput

:

BOOL

;

END\_VAR

VAR\_INPUT

binput2

:

BOOL

;

END\_VAR

VAR\_OUTPUT

output1

:

REAL

;

END\_VAR

VAR\_IN\_OUT

in\_out1

:

LINT

;

END\_VAR

VAR\_IN\_OUT CONSTANT

in\_out\_constant1

:

DINT

;

END\_VAR

VAR

var1

:

STRING

;

END\_VAR

VAR\_TEMP

temp1

:

ULINT

;

END\_VAR

VAR\_STAT

nVarStat1

:

INT

;

END\_VAR

VAR\_EXTERNAL

nVarExt1

:

INT

;

// 1st external variable

END\_VAR

VAR PERSISTENT

nVarPers1

:

DINT

;

(\*

1.

Persistent variable

\*)

bVarPers2

:

BOOL

;

(\*

2.

Persistent variable

\*)

END\_VAR

VAR RETAIN

nRem1

:

INT

;

END\_VAR

VAR CONSTANT

n

:

INT

:=

10

;

END\_VAR

4.2.4 Constructor y Destructor

### 4.2.4 Costruttore e Distruttore

**MÉTODOS 'FB\_INIT', 'FB\_REINIT' Y 'FB\_EXIT':**

**FB\_INIT:**

• A seconda dell'attività, i blocchi funzione potrebbero dover richiedere parametri monouso per le attività di inizializzazione. Un modo possibile per passarli elegantemente è usare il metodo FB\_init(). Questo metodo viene eseguito implicitamente una volta prima dell'avvio del task PLC e può essere utilizzato per eseguire task di inizializzazione.

• È anche possibile sovrascrivere FB\_init(). In questo caso, le stesse variabili di ingresso devono esistere nello stesso ordine e avere lo stesso tipo di dati dell'FB di base. Tuttavia, è possibile aggiungere più variabili di ingresso in modo che il blocco funzione derivato riceva parametri aggiuntivi.

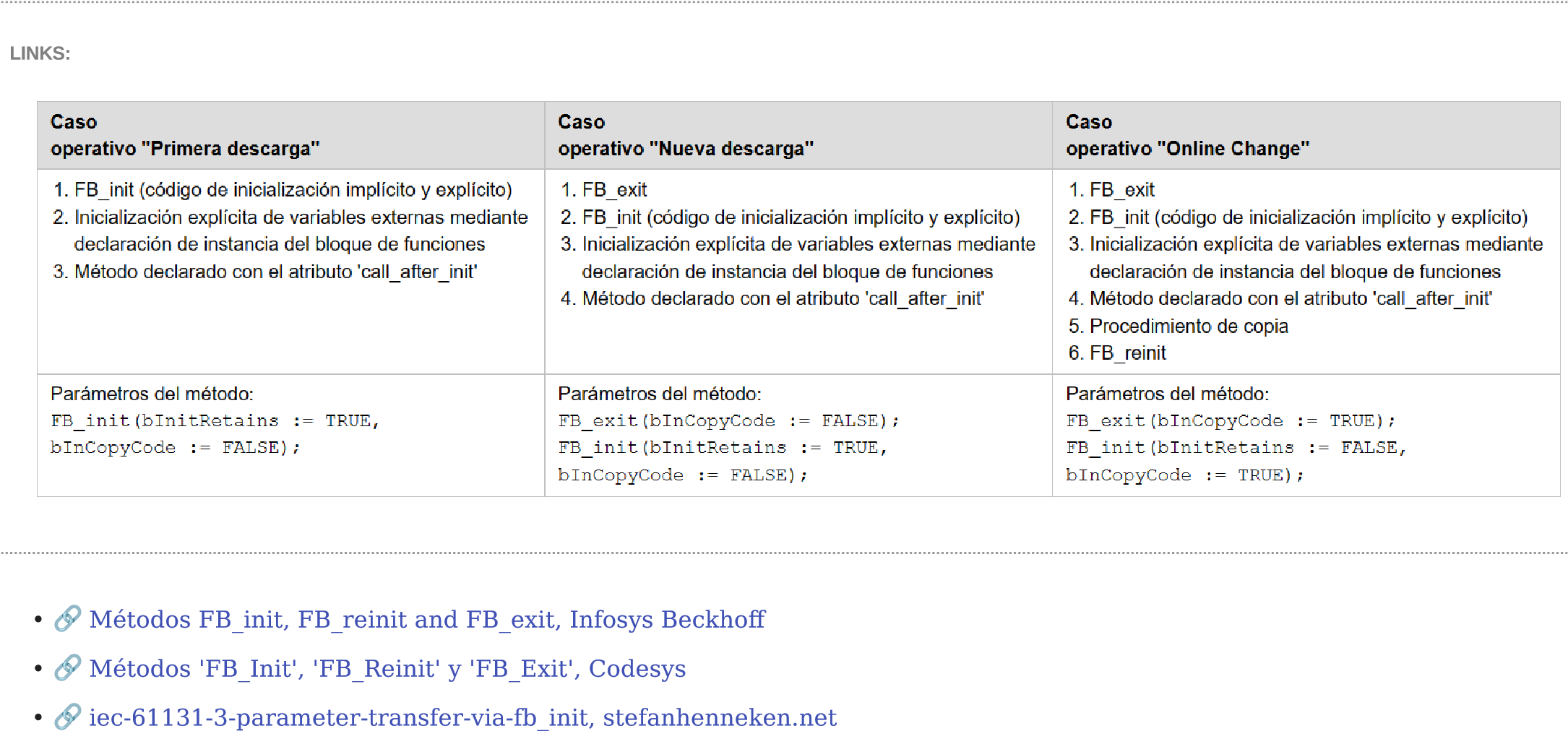
• Passando i parametri tramite FB\_init(), non possono essere letti dall'esterno o modificati in fase di esecuzione. L'unica eccezione sarebbe il richiamo esplicito di FB\_init() dal task PLC. Tuttavia, questo dovrebbe essere principalmente evitato, poiché in questo caso tutte le variabili locali dell'istanza verranno reinizializzate. Tuttavia, se l'accesso dovesse essere ancora possibile, è possibile creare proprietà appropriate per i parametri.

**FB\_REINIT:**

Se necessario, è necessario implementare esplicitamente FB\_reinit. Se questo metodo è presente, viene chiamato automaticamente dopo che l'istanza del blocco funzione corrispondente è stata copiata (chiamata implicita). Ciò si verifica durante una modifica in linea dopo le modifiche alla dichiarazione del blocco funzione (modifica della firma) per reinizializzare il nuovo blocco di istanza. Questo metodo viene chiamato dopo l'operazione di copia e deve impostare valori definiti per le variabili di istanza. Ad esempio, è possibile inizializzare le variabili in base alla nuova posizione in memoria o notificare ad altre parti dell'applicazione la nuova posizione di specifiche variabili in memoria. Progettare l'implementazione indipendentemente dal cambiamento online. Il metodo può anche essere richiamato dall'applicazione in qualsiasi momento per reimpostare un'istanza del blocco funzione al suo stato originale.

**FB\_EXIT:**

Se necessario, devi implementare esplicitamente FB\_exit. Se questo metodo è presente, viene chiamato automaticamente

(implicitamente) prima che il driver rimuova il codice dall'istanza del blocco funzione (ad esempio, anche se TwinCAT passa dalla modalità Run alla modalità di configurazione).

4.3 Oggetto Metodo

## 4.3 Oggetto Metodo

### 4.3.1 Metodo

**METHOD:**

I metodi suddividono la classe (blocco funzione) in funzioni più piccole che possono essere eseguite su chiamata. Funzioneranno solo con i dati di cui hanno bisogno e ignoreranno eventuali dati ridondanti che potrebbero esistere in un determinato blocco funzione.

I metodi possono accedere e manipolare le variabili interne della classe principale, ma possono anche utilizzare le proprie variabili a cui la classe principale non può accedere (a meno che non producano la variabile).

Inoltre, i metodi sono un modo molto più efficiente per eseguire un programma perché suddividendo una funzione in più metodi, l'utente evita di eseguire l'intero POU ogni volta, eseguendo solo piccoli pezzi di codice ogni volta che devono essere chiamati.

Questo è un ottimo modo per evitare errori e danneggiamento dei dati. I metodi sono anche denominati, il che significa che questi pezzi di codice possono essere identificati dal loro scopo piuttosto che dalle variabili che manipolano, migliorando così la lettura, la comprensione e la risoluzione dei problemi del codice.

L'astrazione gioca un ruolo importante qui, se i programmatori vogliono implementare il codice, devono solo chiamare il metodo.

Anche la risoluzione dei problemi diventa più semplice: quindi il programmatore non ha bisogno di cercare ogni istanza del codice, deve solo controllare il metodo corrispondente. A differenza della classe base, i metodi utilizzano la memoria temporanea del controller: i dati sono volatili, poiché le variabili manterranno i loro valori solo finché il metodo è in esecuzione. Se si suppone che i valori vengano mantenuti tra i cicli di esecuzione, la variabile deve essere memorizzata nella classe base o in qualche altro posto che manterrà i valori da un ciclo all'altro (come l'elenco delle variabili globali - GVL), oppure puoi usare la variabile di tipo VAR\_INST.



Pertanto, una dichiarazione di metodo ha la seguente struttura:

1

METHOD

<

Access specifier

>

<

Name

>

:

<

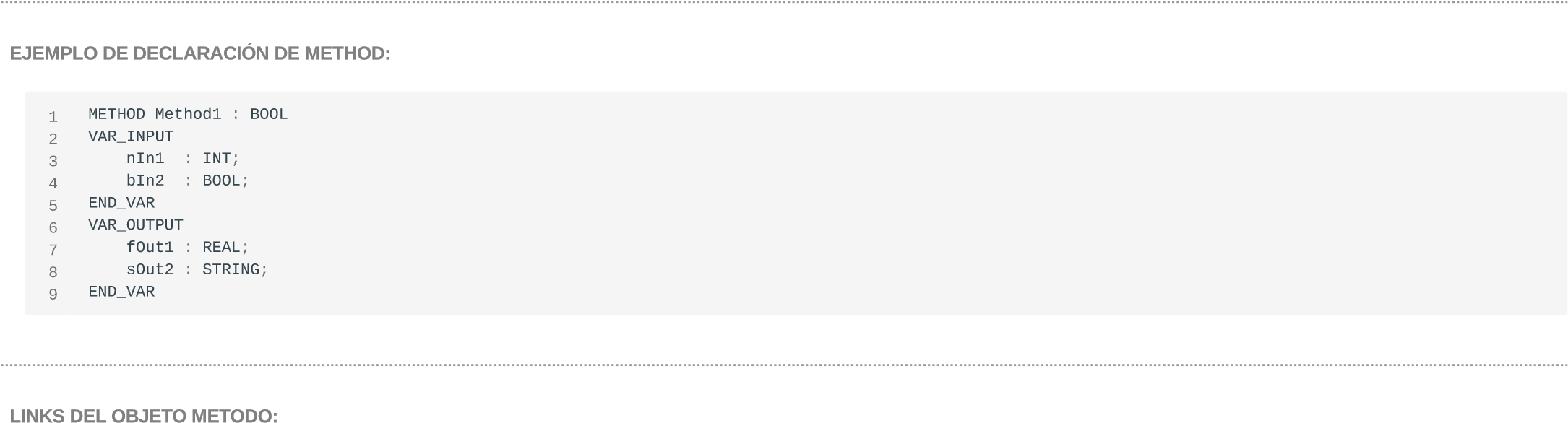
Datatype

return

value

>

Non è obbligatorio che un Metodo debba restituire un valore...



* 🔗 [Documentazione Codesys dell’oggetto](https://help.codesys.com/api-content/2/codesys/3.5.14.0/en/_cds_obj_method/#e4507ebe4233ac0c0a8640e00a37b12-id-3375759d0dd23b38c0a864630d4cd159) Metodo
* 🔗 [Documentazione Beckhoff dell’oggetto Metodo](https://infosys.beckhoff.com/english.php?content=../content/1033/tc3_plc_intro/2530307467.html&id=)
* 🔗 [TC08.Beckhoff TwinCAT3 Function Block-Part3 Method.JP](https://www.youtube.com/watch?v=csr11XOva_w)

4.3.2 Metodo Modificatori di accesso

### 4.3.2 Metodo -> Modificatori di accesso

**SPECIFICATORI DI ACCESSO AL METODO::**

La dichiarazione del metodo può includere uno specificatore di accesso facoltativo. Ciò limita l'accesso al metodo.

**TIPI DI MODIFICATORI DI ACCESSO PER IL METODO:**

* PUBLIC:
* Chiunque può chiamare il metodo, non ci sono restrizioni.
* PRIVATE:

• Il metodo è disponibile solo all'interno del POU. Non è possibile chiamare dall'esterno del POU.

• Sono accessibili all'interno della classe.

• L'accesso è limitato rispettivamente al blocco funzione o al programma.

PROTECTED:

• Solo la propria POU o le POU da essa derivate (ereditate) possono accedere al metodo. La derivazione è discussa di seguito.

• Sono accessibili per ereditarietà.

• L'accesso è limitato rispettivamente al programma o al blocco funzione e ai suoi derivati.

INTERNAL:

* È possibile accedere al metodo solo dallo stesso spazio dei nomi. Ciò consente ai metodi di essere disponibili solo all'interno di una determinata libreria, ad esempio.
* L'accesso è limitato allo spazio dei nomi (la libreria).

La configurazione predefinita in cui non è definito alcun identificatore di accesso è PUBLIC .

* FINAL:(se può aggiungere accompagnato con uno qualsiasi dei precedenti)

• Il metodo non può essere sostituito da un altro metodo. L'override del metodo è descritto di seguito.

• La sovrascrittura non è consentita, in una derivata del blocco funzione.

• Ciò significa che non può essere sovrascritto/esteso su una possibile sottoclasse esistente.

4.3.3 Metodo di dichiarazione delle variabili

### 4.3.3 Metodo Dichiarazioni variabili:

**TIPO DI VARIABILI CHE SI POSSONO DICHIARARE IN UN METODO:**

🔗



[Local Variables - VA](https://infosys.beckhoff.com/content/1033/tc3_plc_intro/2528755083.html)R

🔗



[Input Variables - VAR\_INPU](https://infosys.beckhoff.com/content/1033/tc3_plc_intro/2528760459.html)T

🔗



[Output Variables - VAR\_OUTPU](https://infosys.beckhoff.com/content/1033/tc3_plc_intro/2528765835.html)T

🔗



[Input/Output Variables - VAR\_IN\_OUT, VAR\_IN\_OUT CONSTAN](https://infosys.beckhoff.com/content/1033/tc3_plc_intro/2528771211.html)T

🔗



[Temporary Variable - VAR\_TE](https://infosys.beckhoff.com/content/1033/tc3_plc_intro/2528781963.html)MP

🔗



[Static Variables - VAR\_STA](https://infosys.beckhoff.com/content/1033/tc3_plc_intro/2528787339.html)T

🔗



[External Variables - VAR\_EXTERNA](https://infosys.beckhoff.com/content/1033/tc3_plc_intro/2528792715.html)L

🔗



[Instance Variables - VAR\_INS](https://infosys.beckhoff.com/content/1033/tc3_plc_intro/2528798091.html)T

🔗



[~~Remanent Variables - PERSISTENT, RETA~~](https://infosys.beckhoff.com/content/1033/tc3_plc_intro/2528803467.html)~~IN~~

🔗



[SUPE](https://infosys.beckhoff.com/content/1033/tc3_plc_intro/2528837771.html)

[R](https://infosys.beckhoff.com/content/1033/tc3_plc_intro/2528837771.html)

🔗



[THI](https://infosys.beckhoff.com/content/1033/tc3_plc_intro/2528843147.html)

[S](https://infosys.beckhoff.com/content/1033/tc3_plc_intro/2528843147.html)

🔗



[Variable types - attribute keyword](https://infosys.beckhoff.com/content/1033/tc3_plc_intro/2528848523.html)s

[~~RETAIN: for remanent variables of type RETA~~](https://infosys.beckhoff.com/content/1033/tc3_plc_intro/2528803467.html)~~IN~~

•

•

•

•

•

•

•

•

•

•

•

•

•

* [~~PERSISTENT: for remanent variables of type PERSISTENT~~](https://infosys.beckhoff.com/content/1033/tc3_plc_intro/2528803467.html)
* [CONSTANT: for constants](https://infosys.beckhoff.com/content/1033/tc3_plc_intro/2529284235.html#2529371275)
* Esempio di dichiarazioni variabili in un **METHOD**:

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

METHOD metodo0\_Dichiarazioni\_Variabili

VAR\_INPUT

binput

:

BOOL

;

END\_VAR

VAR\_INPUT

binput2

:

BOOL

;

END\_VAR

VAR\_OUTPUT

output1

:

REAL

;

END\_VAR

VAR\_IN\_OUT

in\_out1

:

LINT

;

END\_VAR

VAR\_IN\_OUT CONSTANT

in\_out\_constant1

:

DINT

;

END\_VAR

VAR

var1

:

STRING

;

END\_VAR

//Non si possono dichiarare variabili TEMPO all’interno delle dichiarazioni variabili del metodo!!!

//VAR\_TEMP

// temp1 : ULINT;

//END\_VAR

VAR\_INST

counter

:

INT

;

END\_VAR

VAR\_STAT

nVarStat1

:

INT

;

aarray

:

ARRAY

[

1.

.

n

]

OF INT

;

END\_VAR

VAR\_EXTERNAL

nVarExt1

:

INT

;

// 1st external variable

END\_VAR

//Non si possono dichiarare variabili PERSISTENT e/o RETAIN all’interno delle dichiarazioni variabili del metodo!!!

//VAR PERSISTENT

// nVarPers1 : DINT; (\* 1. Persistent variable \*)

// bVarPers2 : BOOL; (\* 2. Persistent variable \*)

//END\_VAR

//VAR RETAIN

// nRem1 : INT;

//END\_VAR

VAR CONSTANT

n

:

INT

:=

10

;

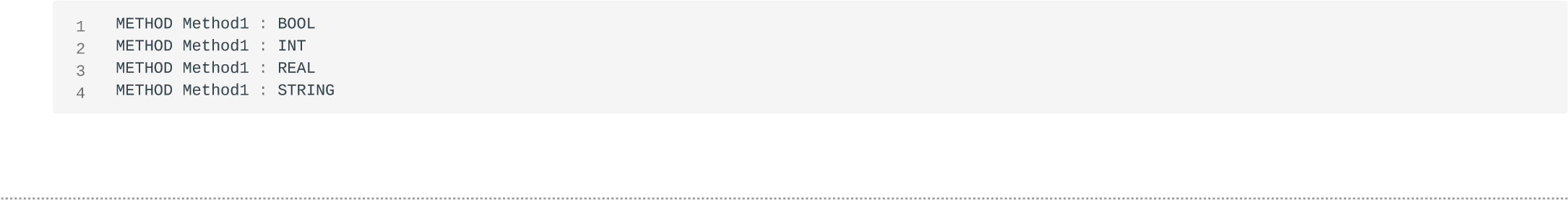
END\_VAR

4.3.4 Metodo tipo di variabili di ritorno

### 4.3.4 Metodo tipi di variabili di ritorno

**TIPI DI VARIABILI DI RITORNO:**

• Non è obbligatorio nel metodo restituire un tipo di variabile.

• Esempi di dichiarazioni di metodo che restituiscono una variabile di tipi diversi:

**RITORNO PER STRUCT:**

Accesso a un singolo elemento di un tipo restituito strutturato durante la chiamata al metodo/funzione/proprietà.

La seguente implementazione può essere utilizzata per accedere direttamente a un singolo elemento del tipo di dati strutturati che il metodo/funzione/proprietà restituisce quando viene chiamato un metodo, una funzione o una proprietà.

Un tipo di dati strutturato è, ad esempio, una struttura o un blocco funzionale.

Il tipo restituito del metodo/funzione/proprietà è definito come:

1

2

3

REFERENCE TO

<

structured type

>

//en lugar de simplemente

<

structured type

>

Tenere presente che con questo tipo di ritorno, se ad es. deve essere restituita un'istanza FB locale del tipo di dati strutturati, si dovrebbe utilizzare l'operatore di riferimento REF= al posto del "normale" operatore di assegnazione := .

Le dichiarazioni e l'esempio in questa sezione fanno riferimento alla chiamata di una proprietà. Tuttavia, sono ugualmente trasferibili ad altre chiamate che offrono valori di ritorno (ad esempio metodi o funzioni).

ESEMPIO:

Dichiarazione della struttura ST\_Sample (STRUCTURE):

1

2

3

4

5

6

TYPE ST\_Sample

:

STRUCT

bVar

:

BOOL

;

nVar

:

INT

;

END\_STRUCT

END\_TYPE

Dichiarazione del blocco funzione FB\_Sample:

1

2

3

4

FUNCTION\_BLOCK FB\_Sample

VAR

stLocal

:

ST\_Sample

;

END\_VAR

Dichiarazione della proprietà FB\_Sample.MyProp con il tipo di ritorno "REFERENCE TO ST\_Sample":

1

PROPERTY MyProp

:

REFERENCE TO ST\_Sample

Implementazione del metodo Get della proprietà FB\_Sample.MyProp:

1

MyProp REF

=

stLocal

;

Implementazione del metodo Set della proprietà FB\_Sample.MyProp: 4.3.4 Metodo tipo di variabili di ritorno

Chiamare i metodi Get e Set nel programma principale MAIN:

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

PROGRAM MAIN

VAR

fbSample

:

FB\_Sample

;

nSingleGet

:

INT

;

stGet

:

ST\_Sample

;

bSet

:

BOOL

;

stSet

:

ST\_Sample

;

END\_VAR

// Get - single member and complete structure possible

nSingleGet

:=

fbSample

.

MyProp

.

nVar

;

stGet

:=

fbSample

.

MyProp

;

// Set - only complete structure possible

IF bSet THEN

fbSample

.

MyProp REF

=

stSet

;

bSet

:=

FALSE

;

END\_IF

Dichiarando il tipo restituito della proprietà MyProp come "REFERENCE TO ST\_Sample" e utilizzando l'operatore di riferimento REF= nel metodo Get di questa proprietà, è possibile accedere direttamente a un singolo elemento del tipo di dati strutturato restituito durante la chiamata alla proprietà.

1

2

3

4

5

VAR

fbSample

:

FB\_Sample

;

nSingleGet

:

INT

;

END\_VAR

nSingleGet

:=

fbSample

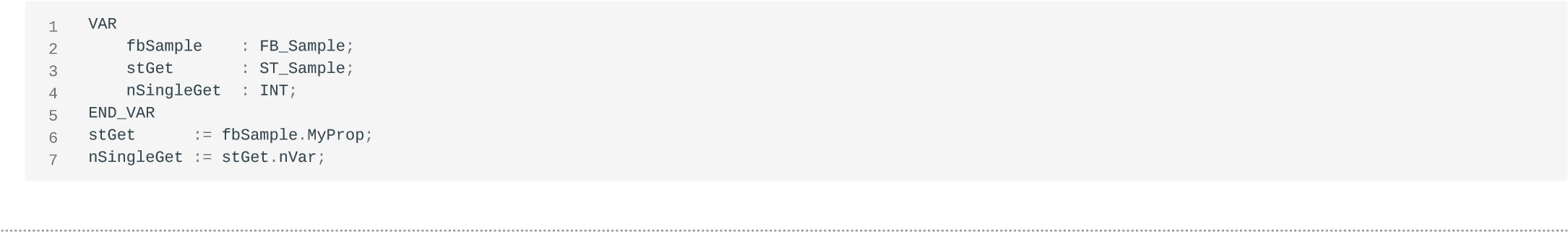
.

MyProp

.

nVar

;

Se il tipo restituito fosse dichiarato solo come "ST\_Sample", la struttura restituita dalla proprietà dovrebbe essere assegnata prima a un'istanza di struttura locale. I singoli elementi della struttura potrebbero essere interrogati sulla base dell'istanza della struttura locale.

**RITORNO PER INTERFACE:**

Esempio di una dichiarazione di metodo che restituisce una variabile di tipo INTERFACE.

**RITORNO PER FUNCTION\_BLOCK:**

Esempio di dichiarazione di metodo che restituisce una variabile di tipo FUNCTION\_BLOCK.

1

METHOD Method1

:

FB1

4.4 Oggetto Proprietà

## 4.4 Oggetto Proprietà

**Proprietà:**



Le proprietà sono le variabili principali di una classe. Possono essere utilizzati in alternativa alla normale classe o al blocco funzione I/O. Le proprietà hanno metodi Get e Set che consentono di accedere e/o modificare le variabili:

• Get - Metodo che restituisce il valore di una variabile.

• Set - Metodo che imposta il valore di una variabile.

Rimuovendo il metodo "Get" o "Set", un programmatore può rendere rispettivamente le proprietà "sola scrittura" o "sola lettura". Poiché si tratta di metodi, significa che le proprietà possono:

• Avere le proprie variabili interne.

• Eseguire operazioni prima di restituire il loro valore.

• Non è necessario allegare la variabile restituita a un particolare ingresso o uscita (o variabile interna) del POU, è possibile restituire un valore basato su una determinata combinazione delle proprie variabili.

• Essere accessibili per evento invece di essere controllati in ogni ciclo di esecuzione.



**Propietà: Getters & Setters:**

per la modifica diretta delle nostre proprietà, vengono utilizzati i metodi Getters e Setter, che variano a seconda della lingua ma il concetto è lo stesso.



Struttura di esempio per la dichiarazione delle proprietà:

1

PROPERTY

<

Access specifier

>

<

Name

>

:

<

Datatype

>

Nell'Oggetto Proprietà è obbligatorio che restituisca un valore.



**Indicatori di visibilità:**

Come per i metodi, anche le proprietà possono accettare i seguenti identificatori di accesso: PUBLIC ,

PRIVATO, PROTETTO, INTERNO e FINALE. Quando non è definito alcun identificatore di accesso, la proprietà è PUBLIC . Inoltre, è possibile specificare uno specificatore di accesso per ogni setter e getter. Questo ha la precedenza sull'identificatore di accesso della proprietà.

Le proprietà sono riconoscibili dalle seguenti caratteristiche:

4.4 Oggetto Proprietà

Indicatori di visibilità:

* PUBLIC:
* Corrisponde alla specifica del modificatore di non accesso.
* PRIVATE:
* L’accesso alla proprietà è limitato solo al blocco funzione.
* PROTECTED:
* L'accesso alle proprietà è limitato al programma o al blocco funzione e ai suoi derivati.
* INTERNAL:
* L'accesso alla proprietà è limitato al namespace, cioè alla libreria
* FINAL:
* Non è consentito sovrascrivere la proprietà in un derivato del blocco funzione. Ciò significa che la proprietà non può essere sovrascritta o estesa in una possibile sottoclasse esistente.
* Le proprietà possono essere astratte, il che significa che una proprietà non ha un'implementazione iniziale e l'effettiva implementazione è fornita nel blocco funzione derivato.

I pragma sono molto utili per monitorare le proprietà in modalità online. Per questo, scrivili all'inizio delle dichiarazioni di proprietà (attributo 'monitoraggio'):

{attribute 'monitoring := 'variable'}: Quando si accede a una proprietà, TwinCAT memorizza il valore effettivo in una variabile e visualizza il valore di quest'ultima. Questo valore può diventare obsoleto se il codice non accede più alla proprietà.

{attribute 'monitoring' := 'call'}: ogni volta che viene visualizzato il valore, TwinCAT chiama il codice di accesso Get.

Qualsiasi effetto collaterale, causato da quel codice, può essere visualizzato nella traccia.



**Links di Oggetti proprietà:**

* 🔗 [Documentación de Codesys del Objeto propiedad](https://help.codesys.com/api-content/2/codesys/3.5.12.0/en/_cds_obj_property/#b08bdbd0d86c0a8640e00400511-id-5dbd3039bdbd0d85c0a8640e003d7982)
* 🔗 [Documentación de Beckhoff del Objeto propiedad](https://infosys.beckhoff.com/content/1033/tc3_plc_intro/2530335371.html?id=7678313929092874522)
* 🔗 [utilizing-properties,twincontrols.com](https://twincontrols.com/community/twincat-troubleshooting/utilizing-properties/#post-76)
* 🔗 [object-oriented-programming-in-programmable-logic-controllers-plc-whats-really-new,en.grse.de](https://en.grse.de/blog/object-oriented-programming-in-programmable-logic-controllers-plc-whats-really-new/)
* 🔗 [TC07.Beckhoff TwinCAT3 Function Block-Part2 Property.JP- DUT](https://www.youtube.com/watch?v=zbnb8K15YdI)

4.5 Herencia

## 4.5 Eredità

### 4.5.1 Eredità blocco funzione

**Eredità blocco funzione:**

I blocchi funzione sono un ottimo modo per mantenere le sezioni del programma separate l'una dall'altra. Ciò migliora la struttura del software e semplifica notevolmente il riutilizzo. In precedenza, l'estensione della funzionalità di un blocco funzione esistente era sempre un compito delicato. Ciò significava modificare il codice o programmare un nuovo blocco funzione attorno al blocco esistente (ovvero, il blocco funzione esistente era effettivamente incorporato in un nuovo blocco funzione). In quest'ultimo caso, è stato necessario creare nuovamente tutte le variabili di ingresso e assegnarle alle variabili di ingresso per il blocco funzione esistente. Lo stesso era richiesto, al contrario, per le variabili di output.

TwinCAT 3 e Codesys (IEC61131-3) introducono il concetto di ereditarietà. L'ereditarietà è uno dei principi fondamentali della programmazione orientata agli oggetti. L'ereditarietà comporta la derivazione di un nuovo blocco funzione da un blocco funzione esistente. Il nuovo blocco può quindi essere esteso. Nella misura consentita dagli specificatori di accesso del blocco funzione padre, il nuovo blocco funzione eredita tutte le proprietà ei metodi del blocco funzione padre. Ogni blocco funzione può avere un numero qualsiasi di blocchi funzione figli, ma solo un blocco funzione padre. La derivazione di un blocco funzione avviene nella nuova dichiarazione del blocco funzione. Il nome del nuovo blocco funzione è seguito dalla parola chiave EXTENDS seguita dal nome del blocco funzione principale. Per esempio:

1

FUNCTION\_BLOCK PUBLIC FB\_NewEngine EXTENDS FB\_Engine

Il nuovo blocco funzione derivato ( FB\_NewEngine ) ha tutte le proprietà ei metodi del suo genitore ( FB\_Engine ). Tuttavia, i metodi e le proprietà vengono ereditati solo quando lo specificatore di accesso lo consente.

Il blocco funzione figlio eredita anche tutte le variabili locali, VAR\_INPUT , VAR\_OUTPUT e VAR\_IN\_OUT dal blocco funzione padre. Questo comportamento non può essere modificato dagli specificatori di accesso.

Se i metodi o le proprietà del blocco funzione padre sono stati dichiarati PROTECTED, è possibile accedervi dal blocco funzione figlio ( FB\_NewEngine ), ma non dall'esterno di FB\_NewEngine .

L'ereditarietà si applica solo ai POU di tipo FUNCTION\_BLOCK.

**Indicatori di Visibilità:**

Le dichiarazioni FUNCTION\_BLOCK , FUNCTION o PROGRAM possono includere uno specificatore di accesso. Ciò limita l'accesso e, ove applicabile, la possibilità di ereditare.

* PUBLIC:

Chiunque può chiamare o istanziare il POU. Inoltre, se il POU è un FUNCTION\_BLOCK , può essere utilizzato per l'ereditarietà. Non si applicano restrizioni.

* INTERN:

Il POU può essere utilizzato solo all'interno del proprio spazio dei nomi. Ciò consente, ad esempio, che le POU siano disponibili solo all'interno di una determinata libreria.

* FINAL:

FUNCTION\_BLOCK non può fungere da blocco funzione principale. I metodi e le proprietà di questo POU non possono essere ereditati. FINAL è consentito solo per POU di tipo FUNCTION\_BLOCK .

La configurazione predefinita in cui non è definito alcun identificatore di accesso è PUBLIC. Gli specificatori di accesso PRIVATE e PROTECTED non sono consentiti nelle dichiarazioni POU.

4.5.1 Eredità Blocco Funzione:

Se si prevede di utilizzare l'ereditarietà, la dichiarazione del blocco funzione avrà la seguente struttura:

1

FUNCTION\_BLOCK

<

Access specifier

>

<

Name

>

EXTENDS

<

Name basic

function

block

>

**METODI DI SOVRASCRITTURA:**

Il nuovo FUNCTION\_BLOCK FB\_NewEngine , derivato da FB\_Engine , può contenere proprietà e metodi aggiuntivi.

Ad esempio, possiamo aggiungere la proprietà Gear. Questa proprietà può essere utilizzata per interrogare e modificare la marcia corrente.

Devi configurare get e set per questa proprietà.

Tuttavia, dobbiamo anche assicurarci che il parametro nGear del metodo Start() venga passato a questa proprietà. Poiché il blocco funzione principale FB\_Engine non ha accesso a questa nuova proprietà, è necessario creare un nuovo metodo con esattamente gli stessi parametri in FB\_NewEngine . Copiamo il codice esistente nel nuovo metodo e aggiungiamo nuovo codice in modo che il parametro nGear venga passato alla proprietà Gear.

1

2

3

4

5

6

7

8

9

10

11

12

METHOD PUBLIC Start

VAR\_INPUT

nGear

:

INT

:=

2

;

fVelocity

:

LREAL

:=

8.0

;

END\_VAR

IF

(

fVelocity

<

MaxVelocity

)

THEN

velocityInternal

:=

fVelocity

;

ELSE

velocityInternal

:=

MaxVelocity

;

END\_IF

Gear

:=

nGear

;

// new

La linea 12 copia il parametro nGear nella proprietà Gear.

Quando un metodo o una proprietà che è già presente nel blocco funzione padre viene ridefinito all'interno del blocco funzione figlio, si parla di override. Il blocco funzione FB\_NewEngine sovrascrive il metodo Start().

Pertanto, FB\_NewEngine ha la nuova proprietà Gear e sovrascrive il metodo Start().

Eredità

1

fbNewEngine

.

Start

(

1

,

7.5

)

;

chiama il metodo Start() su FB\_NewEngine, poiché questo metodo è stato ridefinito (sovrascritto) su FB\_NewEngine .

Chiata STOP:

1

fbNewEngine

.

Stop

()

;

chiama il metodo Stop() da FB\_Engine . Il metodo Stop() è stato ereditato da FB\_NewEngine da FB\_Engine



**LINKS:**

* 🔗 [stefanhenneken.net,iec-61131-3-methods-properties-and-inheritance](https://stefanhenneken.net/2017/04/23/iec-61131-3-methods-properties-and-inheritance/)
* 🔗 [Simple Codesys OOP - Inheritance](https://www.youtube.com/watch?v=ehwCe9C4gWQ)

4.5.2 Herencia Estructura

### 4.5.2 Struttura dell’Eredità:

**Struttura dell’eredità:**

Come i blocchi funzione, le strutture possono essere estese. La struttura ottiene quindi le variabili della struttura di base oltre alle proprie variabili.

Crea una struttura che estende un'altra struttura:

1

2

3

4

5

6

7

TYPE ST\_Base1

:

STRUCT

bBool1

:

BOOL

;

iINT

:

INT

;

rReal

:

REAL

;

END\_STRUCT

END\_TYPE

1

2

3

4

5

6

TYPE ST\_Sub1 EXTENDS ST\_Base1

:

STRUCT

ttime

:

TIME

;

tton

:

TON

;

END\_STRUCT

END\_TYPE

1

2

3

4

5

TYPE ST\_Sub2 EXTENDS ST\_Sub1

:

STRUCT

bBool2

:

BOOL

;

// No se podria llamar la variable bBool1 porque la tenemos declarada en la estructura ST\_Base1

END\_STRUCT

END\_TYPE

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

PROGRAM MAIN

VAR

stestructura1

:

ST\_Sub1

;

stestructura2

:

ST\_Sub2

;

END\_VAR

//Extensión de Estructura:

stestructura1

.

bBool1

;

stestructura1

.

iINT

;

stestructura1

.

rReal

;

stestructura1

.

ttime

;

stestructura1

.

tton

(

in

:=

TRUE

,

pt

:=

T#

1

S

)

;

stestructura2

.

bBool1

;

stestructura2

.

iINT

;

stestructura2

.

rReal

;

stestructura2

.

ttime

;

stestructura2

.

tton

(

in

:=

TRUE

,

pt

:=

T#

1

S

)

;

stestructura2

.

bBool2

;

* In questo modo di estendere una Struttura per Ereditarietà, non è possibile ripetere lo stesso nome di variabile dichiarato con le strutture estese.
* Inoltre, senza utilizzare EXTENDS per la struttura, potremmo farlo come segue:

4.5.2 Struttura dell’eredità:

1

2

3

4

5

TYPE ST\_2

:

STRUCT

bBool

:

BOOL

;

END\_STRUCT

END\_TYPE

1

2

3

4

5

6

TYPE ST\_1

:

STRUCT

sStruct

:

ST\_2

;

sstring

:

STRING

(

80

)

;

END\_STRUCT

END\_TYPE

1

2

3

4

5

6

7

PROGRAM MAIN

VAR

stestructura11

:

ST\_1

;

END\_VAR

stestructura11

.

sstring

;

stestructura11

.

sStruct

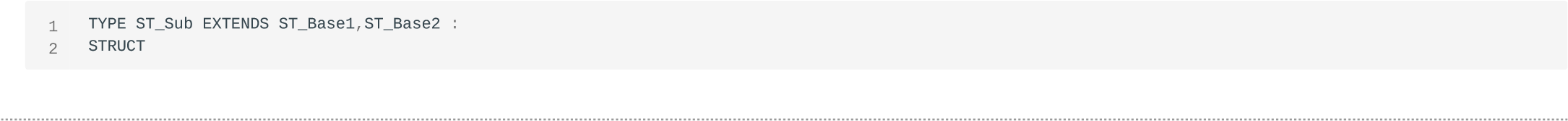
.

bBool

;

//el resultado es que queda mas anidado

* In questo modo è possibile dichiarare lo stesso nome della variabile in Strutture diverse, in quanto il problema precedente non sussiste in quanto sono annidate.
* L'ereditarietà multipla non è consentita come segue:



**LINKS:**

* 🔗 [infosys.beckhoff.com, Extends Structure](https://infosys.beckhoff.com/content/1033/tc3_plc_intro/3468091787.html?id=592001323464924565)
* 🔗 [help.codesys.com, Structure](https://help.codesys.com/webapp/_cds_datatype_structure;product=codesys;version=3.5.17.0)
* 🔗 [help.codesys.com, Structure](https://help.codesys.com/api-content/2/codesys/3.5.14.0/en/_cds_obj_dut/)
* 🔗 [help.codesys.com, Structure](https://help.codesys.com/api-content/2/codesys/3.5.14.0/en/_cds_datatype_structure/#b2e3e6da93f532b0c0a8640e011c7a1d-3s-structures)

4.5.3 Herencia Interface

### 4.5.3 Eredità Interfaccia

**Eredità Interfaccia:**

Come i blocchi funzione, le interfacce possono essere estese. L'interfaccia ottiene quindi i metodi di interfaccia e le proprietà dell'interfaccia di base, oltre alle proprie.

Crea un'interfaccia che estende un'altra interfaccia utilizzando l'estensione:

1

INTERFACE I\_Sub1 EXTENDS I\_Base1

,

I\_Base2

* L'ereditarietà multipla è consentita attraverso l'estensione delle interfacce:

1

INTERFACE I\_Sub2 EXTENDS I\_Sub1

* L'ereditarietà multipla è consentita per le interfacce. È possibile che un'interfaccia si estenda su più di un'interfaccia.



**LINKS:**

* 🔗 [infosys.beckhoff.com, Extends Interface](https://infosys.beckhoff.com/content/1033/tc3_plc_intro/2527343499.html?id=365591094627259992)
* 🔗 [help.codesys.com, Extends Interface](https://help.codesys.com/api-content/2/codesys/3.5.13.0/en/_cds_extending_interface/)

## 4.6 Puntatore THIS

**Puntatore THIS:**

Il puntatore THIS^ viene utilizzato per fare riferimento all'istanza corrente di una classe in un programma orientato agli oggetti. In altre parole, quando viene creato un oggetto di una classe, il puntatore THIS^ viene utilizzato per accedere agli attributi e ai metodi di quell'oggetto specifico. Ad esempio, se abbiamo una classe chiamata "Motor" con un attributo "speed" e un metodo "speed up", quando creiamo un oggetto di classe Motor, possiamo usare il puntatore THIS^ per fare riferimento a quell'oggetto e cambiarne la velocità o accelerare.

Il puntatore THIS^ è disponibile per tutti i blocchi funzione e punta all'istanza corrente del blocco funzione. Questo puntatore è richiesto ogni volta che un metodo contiene una variabile locale che nasconde una variabile nel blocco funzione.

Una dichiarazione di assegnazione all'interno del metodo imposta il valore della variabile locale. Se vogliamo che il metodo imposti il ​​valore della variabile locale nel blocco funzione, dobbiamo usare il puntatore THIS^ per accedervi.

Come per il puntatore SUPER, anche il puntatore THIS deve essere sempre maiuscolo.

1

THIS

^.

METH\_DoIt

()

;

**Esempio:**

* La variabile del blocco funzione nVarB è impostata anche se nVarB è nascosto.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

FUNCTION\_BLOCK FB\_A

VAR\_INPUT

nVarA

:

INT

;

END\_VAR

nVarA

:=

1

;

FUNCTION\_BLOCK FB\_B EXTENDS FB\_A

VAR\_INPUT

nVarB

:

INT

:=

0

;

END\_VAR

nVarA

:=

11

;

nVarB

:=

2

;

METHOD DoIt

:

BOOL

VAR\_INPUT

END\_VAR

VAR

nVarB

:

INT

;

END\_VAR

nVarB

:=

22

;

// Se establece la variable local nVarB.

THIS

^.

nVarB

:=

222

;

// La variable del bloque de funciones nVarB se establece aunque nVarB está oculta.

PROGRAM MAIN

VAR

fbMyfbB

:

FB\_B

;

END\_VAR

fbMyfbB

(

nVarA

:=

0

,

nVarB

:=

0

)

;

fbMyfbB

.

DoIt

()

;

* Una chiamata di funzione richiede il riferimento all'istanza FB stessa.



**Links THIS^ pointer:**

* 🔗 [THIS puntero Infosys Beckhoff](https://infosys.beckhoff.com/content/1033/tc3_plc_intro/2528843147.html?id=1252534934601716110)
* 🔗 [help.codesys.com, THIS](https://help.codesys.com/api-content/2/codesys/3.5.14.0/en/_cds_pointer_this/#efa671ce4f92ff0c0a8640e009d26eb-id-939859e9e4f92fefc0a8640e00938466)
* 🔗 [stefanhenneken.net,iec-61131-3-methods-properties-and-inheritance](https://stefanhenneken.net/2017/04/23/iec-61131-3-methods-properties-and-inheritance/)

4.7 SUPER puntero

## 4.7 Puntatore SUPER

**Puntatore SUPER^:**

Nella programmazione orientata agli oggetti (OOP) nei PLC, il puntatore SUPER^ viene utilizzato per fare riferimento all'oggetto o all'istanza di una classe superiore o genitore. Supponiamo di avere una classe chiamata "Sensor" e un'altra classe chiamata "Temperature\_Sensor", che eredita dalla prima. La classe "Sensor" è la classe genitore o superiore e la classe "Temperature\_Sensor" è la classe figlio o inferiore. Se stai programmando nella classe "Temperature\_Sensor" e hai bisogno di accedere a un metodo o proprietà della classe "Sensor", puoi utilizzare il puntatore SUPER^ per fare riferimento all'istanza della classe "Sensor" a cui appartiene l'oggetto corrente.

Ad esempio, se vuoi accedere al metodo "get\_value()" della classe "Sensor", puoi farlo in questo modo: SUPER^.get\_value(). Ciò indica che si desidera chiamare il metodo "get\_value()" dell'istanza della classe "Sensor" a cui appartiene l'oggetto corrente.

Ogni blocco funzione derivato da un altro blocco funzione ha accesso a un puntatore chiamato SUPER^. Questo può essere utilizzato per accedere agli elementi (metodi, proprietà, variabili locali, ecc.) dal blocco funzione principale.

Invece di copiare il codice dal blocco funzione principale al nuovo metodo, è possibile utilizzare il puntatore SUPER^ per chiamare il metodo dal blocco funzione. Ciò elimina la necessità di copiare il codice.

1

2

SUPER

^();

// Llamada del cuerpo FB de la clase base.

SUPER

^.

METH\_DoIt

()

;

// Llamada del método METH\_DoIt que se implementa en la clase base.

**Esempio:**

* Utilizzo puntatori SUPER y THIS:

Blocco Funzione -- FB\_Base:

1

2

3

4

FUNCTION\_BLOCK FB\_Base

VAR\_OUTPUT

nCnt

:

INT

;

END\_VAR

Metodo -- FB\_Base.METH\_DoIt:

1

2

METHOD METH\_DoIt

:

BOOL

nCnt

:=

-

1

;

Metodo -- FB\_Base.METH\_DoAlso:

1

2

METHOD METH\_DoAlso

:

BOOL

METH\_DoAlso

:=

TRUE

;

Blocco Funzione -- FB\_1:

1

2

3

4

5

6

7

8

9

10

FUNCTION\_BLOCK FB\_1 EXTENDS FB\_Base

VAR\_OUTPUT

nBase

:

INT

;

END\_VAR

THIS

^.

METH\_DoIt

()

;

// llamada al metodo METH\_DoIt del FB\_1.

THIS

^.

METH\_DoAlso

()

;

SUPER

^.

METH\_DoIt

()

;

// llamada al metodo METH\_DoIt del FB\_Base.

SUPER

^.

METH\_DoAlso

()

;

nBase

:=

SUPER

^.

nCnt

;

Metodo -- FB\_1.METH\_DoIt:

1

2

3

METHOD METH\_DoIt

:

BOOL

nCnt

:=

1111

;

METH\_DoIt

:=

TRUE

;

4.7 SUPER puntero

Metodo -- FB\_1.METH\_DoAlso:

1

2

3

METHOD METH\_DoAlso

:

BOOL

nCnt

:=

123

;

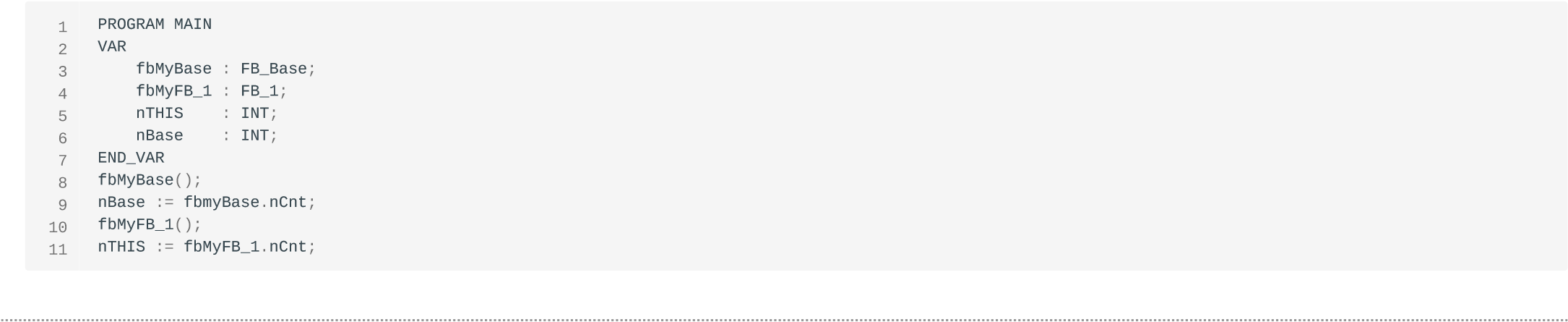
METH\_DoAlso

:=

FALSE

;

Programma MAIN:



**Links SUPER^ pointer:**

* 🔗 [SUPER puntero Infosys Beckhoff](https://infosys.beckhoff.com/content/1033/tc3_plc_intro/2528837771.html?id=5132996865500332085)
* 🔗 [help.codesys.com, SUPER](https://help.codesys.com/api-content/2/codesys/3.5.14.0/en/_cds_pointer_super/)
* 🔗 [stefanhenneken.net,iec-61131-3-methods-properties-and-inheritance](https://stefanhenneken.net/2017/04/23/iec-61131-3-methods-properties-and-inheritance/)

4.8 Interfacce

## 4.8 Interfacce

**Interfacce:**

Nella programmazione orientata agli oggetti (OOP) nei PLC, un'interfaccia è un tipo di struttura che definisce un insieme di metodi e proprietà che una classe deve implementare. In altre parole, un'interfaccia definisce un contratto tra le diverse parti del codice per garantire il rispetto di determinati requisiti e il mantenimento di una struttura coerente. In termini pratici, ciò significa che quando si crea una classe che implementa un'interfaccia, tale classe deve fornire i metodi e le proprietà definiti nell'interfaccia. Ciò consente a classi diverse di condividere un insieme comune di metodi e proprietà e di comunicare tra loro in modo coerente. Ad esempio, se hai un'interfaccia "I\_Sensor" con i metodi:

"ReadValue","Calibrate" e "Uncalibrate" e le Proprietà: "Temperature", "Setpoint" e "Calibrated" ogni classe che implementa questa interfaccia deve fornire questi tre metodi e le tre proprietà. Ciò garantisce che qualsiasi altra parte del codice che funziona con quella classe possa fidarsi che quei metodi e proprietà saranno disponibili.

4.8 Interface



• Un'interfaccia è una classe che contiene metodi e proprietà senza implementazione.

• L'interfaccia può essere implementata in qualsiasi classe, ma quella classe deve implementare tutti i suoi metodi. e proprietà.

• Sebbene l'ereditarietà sia una relazione "è un", le interfacce possono essere descritte come "si comporta come" o "ha una" relazione.

• Le interfacce sono oggetti che consentono a diverse classi di avere qualcosa in comune con meno dipendenze. Classi e blocchi funzione possono implementare diverse interfacce. Si può pensare ai metodi e alle proprietà dell'interfaccia come azioni che significano cose diverse a seconda di chi le sta eseguendo. Ad esempio, la parola "correre" significa "muoversi a una velocità maggiore di una camminata" per un essere umano, ma significa "correre" per i computer.

• Classi o blocchi funzione che non condividono somiglianze possono implementare la stessa interfaccia. In questo caso, l'implementazione dei metodi in ciascuna classe può essere totalmente diversa. Questo apre molti potenti approcci di programmazione:

• I POU possono chiamare un'interfaccia per eseguire un metodo o accedere a una proprietà, senza sapere con quale classe o FB ha a che fare o come eseguirà l'operazione. L'interfaccia punta quindi a una classe o un blocco funzione, che implementa l'interfaccia e l'operazione che viene eseguita.

• I programmatori possono creare switch box facilmente personalizzabili utilizzando il polimorfismo.



**Links Interface:**

* 🔗 [Codesys Comando 'Implementar interfaces'](https://help.codesys.com/api-content/2/codesys/3.5.12.0/en/_cds_cmd_implement_interfaces/)
* 🔗 [Codesys Objeto Interface](https://help.codesys.com/api-content/2/codesys/3.5.13.0/en/_cds_obj_interface/)
* 🔗 [Codesys Implementando Interfaces](https://help.codesys.com/api-content/2/codesys/3.5.12.0/en/_cds_implementing_interface/)
* 🔗 [Beckhoff Objeto Interface](https://infosys.beckhoff.com/english.php?content=../content/1033/tc3_plc_intro/4256428299.html&id=)
* 🔗 [Beckhoff Implementando Interfaces](https://infosys.beckhoff.com/english.php?content=../content/1033/tc3_plc_intro/4262436875.html&id=)
* 🔗 [Extender Interfaces, Infosys Beckhoff](https://infosys.beckhoff.com/content/1033/tc3_plc_intro/2527343499.html?id=365591094627259992)
* 🔗 [TC09.Beckhoff TwinCAT3 Function Block-Part4 Interface.JP](https://www.youtube.com/watch?v=SAGchtGi3-o)

## 4.9 Puntatore e Riferimento

**Puntatore e Riferimento:**

Nella programmazione orientata agli oggetti (OOP) nei PLC IEC 61131-3, puntatori e riferimenti sono due concetti importanti utilizzati per accedere ai dati e ai metodi di un oggetto. Un puntatore è una variabile che memorizza l'indirizzo di memoria di un'altra variabile. Un riferimento è una variabile utilizzata per accedere a un'altra variabile senza dover conoscere il suo indirizzo di memoria.

**. Cos’è un puntatore**

* È un dato che punta o punta a un indirizzo di memoria.
* È una variabile che contiene l'indirizzo di memoria dove la variabile “vive”.
* Con l'uso dei puntatori si accede direttamente alla memoria, quindi è una buona tecnica per ridurre il tempo di esecuzione di un programma e molte altre funzionalità.

**. Tipi di puntatori:**

* Esiste un tipo di puntatore per ogni tipo di dato, programma, blocco funzione, funzione, ecc.
* A seconda dell'“oggetto” a cui si desidera accedere, è necessario un puntatore di un tipo o di un altro.

**. Dichiarazione del puntatore:**

Il compilatore deve conoscere tutti i puntatori che verranno utilizzati nel progetto, quindi devono essere dichiarati, come qualsiasi altra variabile. Il codice mostra lo script necessario per la dichiarazione di vari tipi di puntatori:

1. // Un puntero no deja de ser una variable, la diferencia está en que su contenido no es un valor determinado sino que es la dirección
2. // de memoria donde se ubica la variable de la que se quiere leer o escribir su valor. Y al igual que hay que declarar todas las 3 // variables del tipo correspondiente. también hay que declarar todas las variables -punteros- que contendrán esas direcciones de 4 // memoria y su correspondiente tipo.
3. VAR
4. stTest1 : stTipo1; //Declara una estructura de datos del tipo stTipo1.

7

1. pin01 : POINTER TO INT; //Declara un puntero para acceder a variables del tipo INT.
2. ps20 : POINTER TO STRING[20]; //Declara un puntero para acceder a variables del tipo STRING de 20 caracteres.
3. pa20 : POINTER TO ARRAY [1..20] OF INT; //Declara un puntero para acceder a variables del tipo ARRAY de 20 elementos del tipo INT.
4. pDword : POINTER TO DWORD; //Declara un puntero para acceder a variables del tipo DWORD.
5. past1 : POINTER TO stTipo1; //Declara un puntero para acceder a variables del tipo stTipo1.
6. pReal : POINTER TO REAL; //Declara un puntero para acceder a variables del tipo REAL. 14 END\_VAR

**. Come sapere quale indirizzo assegnare al puntatore:**

• Per poter accedere ad una variabile tramite un puntatore è necessario conoscerne l'indirizzo di memoria, per fare questo esiste un operatore chiamato ADR che assegna al puntatore l'indirizzo della variabile desiderata.

• È opportuno verificare che il valore del puntatore non sia zero, prima di utilizzarlo. D'altra parte, per poter leggere/scrivere il valore della variabile, a cui punta il puntatore, è disponibile l'operatore di contenuto ^. Quando ci si riferisce al contenuto dell'indirizzo di memoria puntato, si parla di dereferenziare il puntatore. Il codice seguente mostra un esempio:

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

PROGRAM SR\_Main\_02

VAR

in01

:

INT

;

//Declaración de la variable in01 de tipo entero.

in02

:

INT

:=

123

;

//Declaración e inicialización de la variable in02 de tipo entero.

in03

:

INT

;

//Declaración de la variable in03 de tipo entero.

pint

:

POINTER TO INT

;

//Declaración de un puntero para acceder a variables del tipo entero.

END\_VAR

// Ejemplo de uso básico de los operadores ADR y del operador de contenido ^

// Se muestra como asignar a un puntero la dirección de memoria de una variable y como leer/escribir

// así como un ejemplo de acceso a variables locales de otros programas.

pint

:=

ADR

(

in01

)

;

//Asignamos al puntero la dirección de memoria donde se ubica la variable in01.

pint

^

:=

44

;

//A la posición de memoria indicada por el puntero, le asignamos el valor 44

//Por tanto a la variable in01 se le ha escrito el valor 44.

in02

:=

in01

;

// in02 será igual a 44.

pint

:=

ADR

(

in02

)

;

//Cambiamos la dirección para acceder a la dirección de la variable in02.

in03

:=

pint

^;

// in03 tomara el valor del contenido de la posición de memoria contenida en el

// que hemos asignado la dirección de in02, por tanto in03= 123.

pint

:=

ADR

(

SR\_Main\_01

.

inLocalAway

)

;

//Cargamos la dirección de memoria de una variable local de

// otro programa, la que sería inaccesible por otros medios.

pint

^

:=

240

;

// La varible local del programa SR\_Main\_01.inLocalAway tomará el valor 240

**. Cos’è un accesso indiretto?**

Prima di tutto, dì che non ha nulla a che fare con un puntatore. Un accesso indiretto permette di scegliere un numero di elemento all'interno di un array, esiste una variabile, chiamata indice, che contiene il numero dell'elemento dell'array a cui si vuole accedere. In questo caso, non è possibile accedere a nessun'altra variabile oltre agli elementi dell'array, insisto sul fatto che non ha nulla a che fare con i puntatori. Con un puntatore è possibile accedere a qualsiasi dato o oggetto presente nella memoria del controllo. Con un accesso indiretto è possibile accedere solo agli elementi di un array. Il codice seguente mostra alcuni esempi di accesso indiretto a un array:

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

PROGRAM SR\_Main\_01

VAR

aR20

:

ARRAY

[

1..20

]

OF REAL

;

//Declara un array de 20 elementos del tipo REAL.

inIndex

:

INT

;

//Declara la variable de indice del array para el acceso indirecto

xNewVal

:

BOOL

;

//Indica que hay una nueva lectura del sensor de fuerza.

rFuerza

:

REAL

;

//Valor de fuerza del sensor.

END\_VAR

// Ejemplo01: Se asigna valores del 1 al 20 a cada elemento del array mediante un bucle.

FOR inIndex

:=

1

TO

20

BY

1

DO

// Se empieza por el valor de la variable indice a 1, hasta 20

aR20

[

inIndex

]

:=

inIndex

;

// Al elemento aR20[inIndex] se le asigna el valor de inIndex

END\_FOR

;

// Se incrementa inIndex y se repite el proceso.

// Ejemplo02: Creamos un FIFO en el que guardamos un valor analógico de fuera a cada impulso de la señal xNewVal.

IF xNewVal THEN

// Si hay un nuevo valor de fuerza realizamos el codigo.

xNewVal

:=

FALSE

;

// Reset de la señal xNewVal.

FOR inIndex

:=

20

TO

2

BY

-

1

DO

// Variable indice a 20, hasta 2.

aR20

[

inIndex

]

:=

aR20

[

inIndex

-

1

]

;

//Desplazamiento de los valores en el FIFO --->

END\_FOR

;

// Se decrementa inIndex y se repite el proceso.

aR20

[

1

]

:=

rFuerza

;

// Entrada del valor de fuerza en el primer elemento del FIFO.

END\_IF

A questo stesso array si può accedere usando un puntatore, come si vedrà in seguito, che è più veloce nei tempi di esecuzione, ma non così chiaro per chi non usa abitualmente i puntatori.

**. Accesso a una struttura dati tramite i puntatori:**

Il processo è lo stesso già visto per accedere ad una variabile di tipo INT, ma dovrai dichiarare un puntatore del tipo appropriato, che corrisponda al tipo di struttura a cui vuoi accedere, vediamolo nel codice seguente:

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

PROGRAM SR\_Main\_03

VAR

stMotor\_01

:

stMotorCtrl

;

// Estructura de control del motor 1

stMotor\_02

:

stMotorCtrl

;

// Estructura de control del motor 2

stMotor\_03

:

stMotorCtrl

;

// Estructura de control del motor 3

pstMotorCtrl

:

POINTER TO stMotorCtrl

;

// Puntero para acceder a estructuras del tipo stMotorCtrl.

xMarcha

:

BOOL

;

// Pulsador marcha motores.

END\_VAR

// Ejemplo básico de como acceder a estructuras de datos mediante punteros.

// La estructura de datos empleada es una llamada a stMotorCtrl, que coincide un bit de marcha, otro de paro y

// valores de velocidad en Rpms y tiempo de aceleración/deceleración.

// Asignamos valores a la estructura para el control del motor 1.

stMotor\_01

.

rTpoAcelDecel

:=

5.4

;

// Tiempo para acelerar/decelerar hasta alcanazar la velocidad.

stMotor\_01

.

rVelRpm

:=

1436.2

;

// Velocidad en RPM.

stMotor\_01

.

xMotorOff

:=

TRUE

;

// Bit de paro ON.

stMotor\_01

.

xMotorOn

:=

FALSE

;

// Bit de marcha OFF.

pstMotorCtrl

:=

ADR

(

stMotor\_01

)

;

// Cargamos la dirección de memoria de la estructura del motor 1

stMotor\_02

:=

pstMotorCtrl

^;

// Copia el contenido de la zona de memoria apuntada a la

// estructura del motor 2, en este caso el resultado es el mismo

// que se obtendría con stMotor\_02:= stMotor\_01;

stMotor\_03

:=

stMotor\_02

;

// Copia los mismos valores al motor 3;

IF xMarcha THEN

// Si se pulsa marcha máquina

pstMotorCtrl

^.

xMotorOn

:=

TRUE

;

// Se activa el bit de marcha al que apunta el puntero (stMotor\_01).

pstMotorCtrl

^.

xMotorOff

:=

FALSE

;

// Se desactiva el bit de paro al que apunta el puntero (stMotor\_01)

END\_IF

**. Accesso a un array tramite puntatori:**

Il processo è lo stesso di quello già visto per accedere a una variabile di tipo INT, ma dovrai dichiarare un puntatore ad un array con il numero di elementi e il tipo di dati appropriato, vediamolo nel codice seguente:

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

PROGRAM SR\_Main\_03

VAR

aintFIFO

:

ARRAY

[

1..20

]

OF INT

;

// Array de 20 enteros.

aintFIFO2

:

ARRAY

[

1..20

]

OF INT

;

// Array de 20 enteros.

paint

:

POINTER TO ARRAY

[

1..20

]

OF INT

;

// Puntero al array.

pint

:

POINTER TO INT

;

// Puntero a un entero.

END\_VAR

// Ejemplo basico de como acceder a arrays mediante punteros:

paint

:=

ADR

(

aintFIFO

)

;

// \_Asignamos la dirección del array al puntero.

paint

^[

3

]

:=

4

;

// Dentro del array podemos acceder a un elemento en concreto

aintFIFO2

:=

paint

^;

// O copiar el array apuntado entero, sobre otro array

pint

:=

paint

+

(

4

\*

SIZEOF

(

INT

))

;

// Tambien se puede crear un puntero a un INT para acceder a uno de los

// elementos del array. Tomamos la dirección inicial del array y le

// sumampos un offeset de tantos bytes como se necesitan para el tipo de

// datos INT y lo multiplicamos por el indice del array al que queremos

// acceder. SIZEOF (TYPE) retorna el número de bytes según el tipo de datos.

pint

^

:=

5

;

// Asignamos el valor de 5, aintFIFO[5]:=5 sería lo mismo.

**. Accesso ai dati tramite riferimenti:**

L'accesso ai riferimenti è ancora un accesso tramite puntatore, ma in questo caso l'indirizzo di un riferimento è lo stesso dell'indirizzo dell'oggetto a cui punta. Un puntatore ha il proprio indirizzo, e questo contiene l'indirizzo dell'oggetto a cui vuoi fare riferimento. I riferimenti vengono inizializzati all'inizio del programma e non possono essere modificati durante la sua esecuzione. Un puntatore può avere il suo indirizzo modificato quanto necessario durante l'esecuzione del programma. Un altro modo di intendere i riferimenti è come se fossero un altro modo di riferirsi allo stesso oggetto/variabile, come se fosse un alias. Rispetto ai puntatori, i riferimenti hanno i seguenti vantaggi: - 1) Facilità d'uso. - 2) Sintassi più semplice quando si passano i parametri alle funzioni. - 3) Ridurre al minimo gli errori nella scrittura del codice.

Il riassunto di tutto ciò, che può portare a molta confusione, è che, come vedremo in seguito, il grande valore dei riferimenti è quando si passano grandi quantità di dati come parametri di input alle funzioni.

**. Vari modi di passare i parametri alle funzioni:**

Normalmente una funzione esegue alcune operazioni con alcuni parametri di input e restituisce uno o più valori come risultato. Nell'esempio che vedremo di seguito, è una funzione per calcolare l'area di un rettangolo, a cui passeremo i valori del lato A e del lato B in modo che restituisca il risultato dell'area.

Per prima cosa definiremo un tipo di dato [stRectangle] che conterrà il lato A, il B e l'area.

Creeremo tre rettangoli, [stRectangle01], [stRectangle02] e [stRectangle03].

Insieme a tre varianti della funzione di calcolo dell'area: - [Fc\_AreaCalcVal] - passa per valori - - [Fc\_AreaCalcPoint] - passa per puntatore - - [Fc\_AreaCalcRef] - passa per riferimento - Di seguito è riportato il codice per le tre funzioni:

**Passaggio di valori:**

1

2

3

4

5

6

7

8

9

10

// Función para calcular el area de un Rectangulo, pasando los valores de los lados del Rectangulo

// la función retorna el resultado del area calculado

FUNCTION Fc\_AreaCalcVal

:

REAL

// La función retona un número real

VAR\_INPUT

i\_rASide

:

REAL

;

// Parámetro de entrada que contiene el lado A del rectangulo.

i\_rBSide

:

REAL

;

// Parámetro de entrada que contiene el lado B del rectangulo.

END\_VAR

Fc\_AreaCalcVal

:=

i\_rASide

\*

i\_rBSide

;

// Retorna el resultado de multiplicar el lado A por el lado B.

**Passaggio per puntatori:**

1

2

3

4

5

6

7

8

9

10

11

12

13

// Función para calcular el area de un Rectangulo, con los valores contenidos en una estructura de datos del tipo stRectangulo

// La estructura se pasa mediante un puntero a la estructura stRectangulo deseada y la función retorna el resultado a la

// misma estructura.

FUNCTION Fc\_AreaCalcPoint

:

REAL

VAR\_INPUT

i\_ptstRect

:

POINTER TO st\_Rectangulo

;

// Puntero de entrada con la dirección de la estructura.

END\_VAR

// El valor del area, de la estructura indicada por la dirección del puntero es igual al

// valor del lado A de la estructura indicada por la dirección del puntero por

// el valor del lado B de la estructura indicada por la dirección del puntero

i\_ptstRect

^.

rArea

:=

i\_ptstRect

^.

rASide

\*

i\_ptstRect

^.

rBSide

;

**Passaggio per roferimento:**

1

2

3

4

5

6

7

8

9

// Función para calcular el area de un Rectangulo, con los valores contenidos en una estructura de datos del tipo stRectangulo

// La estructura se pasa por referencia.

FUNCTION Fc\_AreaCalcRef

:

REAL

VAR\_INPUT

i\_Ref

:

REFERENCE TO st\_Rectangulo

;

END\_VAR

i\_Ref

.

rArea

:=

i\_Ref

.

rASide

\*

i\_Ref

.

rBSide

;

**Esempio di codice di chiamate di funzione:**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

PROGRAM SR\_Main\_01

VAR

inLocalAway

:

INT

;

// Variable integer local de SR\_Main\_01 para ser accedida externamente

stRectangulo1

:

st\_Rectangulo

;

// Estructura que contiene los datos del rectangulo1 A, B y su area

stRectangulo2

:

st\_Rectangulo

;

// Estructura que contiene los datos del rectangulo2 A, B y su area

stRectangulo3

:

st\_Rectangulo

;

// Estructura que contiene los datos del rectangulo3 A, B y su area

refRectangulo

:

REFERENCE TO st\_Rectangulo

:=

stRectangulo3

;

// Hace Referencia a stRectangulo3

END\_VAR

// Asignación de valores a los lados de los tres rectángulos.

// Asignación de valores de los lados del rectángulo 1

stRectangulo1

.

rAside

:=

44

;

//Valor del lado A.

stRectangulo1

.

rBside

:=

32

;

//Valor del lado B.

// Asignación de valores de los lados del rectángulo 2

stRectangulo2

.

rAside

:=

12.8

;

//Valor del lado A.

stRectangulo2

.

rBside

:=

320.4

;

//Valor del lado B.

// Asignación de valores de los lados del rectángulo 3

stRectangulo3

.

rAside

:=

1024.2

;

//Valor del lado A.

stRectangulo3

.

rBside

:=

2048.4

;

//Valor del lado B.

// Cálculo del área del rectángulo pasando valores a la función

stRectangulo1

.

rArea

:=

Fc\_AreaCalcVal

(

i\_rAside

:=

stRectangulo1

.

rAside

,

i\_rBside

:=

stRectangulo1

.

rBside

)

;

// Cálculo del área del rectángulo pasando un puntero a la función

Fc\_AreaCalcPoint

(

ADR

(

stRectangulo2

))

;

// Cálculo del área del rectángulo pasando una referencia a la función

Fc\_AreaCalcRef

(

refRectangulo

)

;

In questo caso, le differenze possono sembrare insignificanti, poiché la quantità di dati che viene passata alla funzione è piccola. Ma poi vedremo un esempio con un numero maggiore di parametri di input per poter apprezzare i vantaggi del passaggio dei parametri soprattutto per riferimento e anche per puntatore.

**.** **Caso di passaggio di grandi quantità di dati a funzioni:**

Quando è necessario passare strutture con una grande quantità di dati a funzioni o FB, passare parametri per valori non è il metodo più appropriato in quanto è richiesto un numero elevato di parametri di input, ogni parametro prevede la creazione di una nuova variabile locale del funzione, o dell'FB, che presuppone un dispendio di memoria e tempo di esecuzione nella copia dei dati. Nel caso di strutture dati di diversi Kbyte, o array di centinaia o migliaia di elementi, questo metodo è impensabile. Nel caso di dover passare grandi quantità di dati, la soluzione è l'uso di puntatori, o meglio ancora, passando i dati per riferimento. Di seguito un esempio di funzione per calcolare il valore medio di un array di 20 elementi, passando i valori alla funzione e passando i valori tramite un riferimento.

**Codice funzione Fc\_AverageValues ​​per il passaggio di valori:**

**Fc\_AverageReferencia codice funzione per passaggio per riferimento:**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

// Esta función calcula la media de un buffer de 20 elementos. Solo a modo de ejemplo comparativo

// pasando valores por referencia.

FUNCTION Fc\_AverageReferencia

:

REAL

VAR\_INPUT

i\_Ref

:

REFERENCE TO ARRAY

[

1..20

]

OF REAL

;

END\_VAR

VAR

intIdx

:

INT

;

// Variable indice para el bucle.

rVAcum

:

REAL

:=

0

;

// Valor acumulado.

END\_VAR

// Retorna la suma de todos los valores divida del número de valores que son 20.

FOR intIdx

:=

1

TO

20

BY

1

DO

rVAcum

:=

rVAcum

+

i\_Ref

[

intIdx

]

;

END\_FOR

;

Fc\_AverageReferencia

:=

rVAcum

/

20.0

;

**Esempio di codice che chiama entrambe le funzioni:**

Come si può notare, la chiamata di funzione che passa i valori per riferimento è la migliore. E in questo esempio si è ipotizzato un esempio con solo 20 dati di input, ma è normale trovare applicazioni con strutture dati di diversi Kbyte.

• Un puntatore di tipo T indica un oggetto di tipo T (T = tipo di dati di base o definito dall'utente).

• Un puntatore contiene l'indirizzo dell'oggetto a cui punta.

• L'operazione fondamentale con un puntatore si chiama "dereferenziazione". La dereferenziazione in CODESYS viene eseguita con il simbolo "^"

• Un puntatore può puntare a un oggetto diverso in un momento diverso.

• Prima di dereferenziare un puntatore e assegnargli un valore, dovresti sempre controllare se un puntatore punta a un oggetto. (puntatore = 0)?

• Un riferimento di tipo T "punta" a un oggetto di tipo T (T = tipo di dati di base o definito dall'utente).

• Un riferimento deve essere inizializzato con un oggetto e il suo "puntamento" a questo oggetto in tutto il programma.

• Un riferimento non deve essere dereferenziato come puntatore e può essere utilizzato con la stessa sintassi dell'oggetto.

• Un'altra parola di riferimento è "Alias" (un altro nome) uno pseudonimo per l'oggetto.

• Il riferimento non ha un indirizzo appropriato e un puntatore sì. La direzione del riferimento è la stessa dell'oggetto "appuntito".

Il miglior uso di puntatori e riferimenti è quando si desidera passare o restituire un oggetto di qualche tipo a una funzione o blocco funzione per "riferimento" perché l'oggetto è troppo grande o si desidera manipolare l'oggetto passato all'interno della funzione/blocco funzione . Assicurati che il lettore del tuo codice sappia che cambierai il valore dell'oggetto all'interno della funzione/blocco funzione se questo è ciò che intendi fare quando lo passi come argomento.

**. Riassunto / Conclusioni:**

• La memoria contiene migliaia e persino milioni di celle o byte, in cui si trovano il codice del programma e tutti i dati/variabili. Ogni cella ha il suo numero, che si chiama indirizzo di memoria ed è solitamente espresso in esadecimale 16#FA1204 -ad esempio-

• Un puntatore è una variabile, che invece di contenere un valore contiene un indirizzo di memoria, in cui “vive” la variabile a cui vogliamo realmente accedere.

• Come qualsiasi altra variabile, i puntatori devono essere dichiarati in modo che il compilatore possa localizzarli in memoria. Ricorda che un puntatore è una variabile, ma il suo contenuto è un indirizzo di memoria.

• Per ogni tipo di variabile, è richiesto il tipo di puntatore corrispondente. Non è possibile accedere a una variabile INT con un puntatore destinato ad accedere a una struttura dati.

• Niente ha a che fare con l'accesso indiretto ad un array attraverso una variabile indice, con un puntatore. In questo caso l'accesso è limitato all'array stesso, con il puntatore si può accedere a qualsiasi indirizzo di memoria.

• Con i puntatori puoi accedere a tutti i tipi di dati, in una singola riga di codice puoi copiare un'intera struttura di diversi Kbyte di dati. Che è molto più veloce.

• Un riferimento è molto simile a un puntatore, per semplificare potremmo dire che è un "alias" di un oggetto e che è un po' meno critico dei puntatori, la sua principale utilità è quella di passare un gran numero di parametri alle funzioni, in molto semplice e veloce.

• Il passaggio di parametri ad una funzione può avvenire in vari modi, per valori, per puntatori o per riferimento, il programmatore deve scegliere quello più appropriato per ogni applicazione.

• Quando si ha a che fare con grandi quantità di dati, sarà opportuno passare i parametri per riferimento o per puntatori.

**Links Puntatori e Riferimenti:**

* 🔗 [Perre Garriga,Pointer&Reference](https://www.infoplc.net/blogs-automatizacion/item/112457-punteros-programacion-plcs-basico-que-hay-que-saber)
* 🔗 [Control and use of Pointers In Codesys](https://www.youtube.com/watch?v=I_wdSAyJ6LE)
* 🔗 [help.codesys.com, Pointers](https://help.codesys.com/api-content/2/codesys/3.5.14.0/en/_cds_datatype_pointer/#ca488150a6a89ac0a8640e013d12b7-id-2afac6cf50a6a899c0a8640e001ca61a)
* 🔗 [AT&U, CODESYS - Difference between pointer and reference](https://www.youtube.com/watch?v=CAeWLPaUeuM)
* 🔗 [AT&U, CODESYS -Differente between pass by vale and pass by Reference](https://www.youtube.com/watch?v=rUfDD5FTgE0)

4.10 FB abstracto frente a interfaz

## 4.10 FB astratto rispetto all'interfaccia

**FB astratto rispetto all’interfaccia:**

La differenza tra l'utilizzo di un blocco funzione astratto e un'interfaccia è che l'FB astratto è un tipo di modello che definisce un insieme di variabili e parametri di ingresso/uscita da utilizzare in diverse parti del programma.

D'altra parte, un'interfaccia definisce un insieme di metodi e attributi (proprietà) che devono essere implementati da qualsiasi classe che la implementa.

In breve, i blocchi funzione astratti sono utili quando è necessario riutilizzare il codice in diverse parti del programma, mentre le interfacce sono utili quando si desidera garantire che determinate classi implementino determinati metodi.

Immagina di avere un programma che controlla diversi tipi di motori, come motori elettrici, motori a benzina e motori diesel. Per creare una struttura modulare e riutilizzabile, è possibile creare un blocco funzione astratto denominato "Motor Controller" che dispone di ingressi per tipo di motore, velocità e direzione. Questo blocco funzione astratto può quindi essere utilizzato in diverse parti del programma per controllare diversi motori. Il blocco funzione astratto definisce un modello comune utilizzato in diverse parti del programma. D'altra parte, se si desidera assicurarsi che tutte le classi che controllano i motori implementino determinati metodi (ad esempio, un metodo per accendere il motore e un altro per spegnerlo), è possibile creare un'interfaccia chiamata "Motor Controller" che definisce questi metodi. Pertanto, qualsiasi classe che implementa questa interfaccia deve implementare questi metodi. In breve, i blocchi funzione astratti sono utili quando è necessario riutilizzare il codice in diverse parti del programma, mentre le interfacce sono utili quando si desidera garantire che determinate classi implementino determinati metodi.

• Blocchi funzione, metodi e proprietà possono essere contrassegnati come astratti. "da TwinCAT V3.1 build 4024".

• Gli FB astratti possono essere utilizzati solo come FB di base per l'ereditarietà.

• La creazione diretta di FB astratti non è possibile, pertanto gli FB astratti presentano alcune somiglianze con le interfacce.

Ora la domanda è in quale caso dovrebbe essere utilizzata un'interfaccia e in quale caso un FB astratto. metodi astratti:

1

METHOD PUBLIC ABSTRACT DoSomething

:

LREAL

- consistono esclusivamente nella dichiarazione e non contengono alcuna implementazione. Il corpo del metodo è vuoto. - Può essere pubblico, protetto o interno. Il modificatore di accesso privato non è consentito. - non può essere ulteriormente dichiarato definitivo. proprietà astratte:

1

PROPERTY PUBLIC ABSTRACT nAnyValue

:

UINT

- può contenere getter, setter o entrambi. - getter e setter consistono solo nella dichiarazione e non contengono alcuna implementazione. - può essere pubblico, protetto o interno. Il modificatore di accesso privato non è consentito. - non può essere ulteriormente dichiarato definitivo. blocchi funzione astratti:

1

FUNCTION\_BLOCK PUBLIC ABSTRACT FB\_Foo

- Non appena un metodo o una proprietà viene dichiarato astratto, anche il blocco funzione deve essere dichiarato astratto. - Non è possibile creare istanze da FB astratti. Gli FB astratti possono essere utilizzati solo come FB di base se ereditati. - Tutti i metodi astratti e tutte le proprietà astratte devono essere sovrascritti per creare un FB non astratto. Un metodo astratto o una proprietà astratta viene convertito in un metodo non astratto o in una proprietà non astratta da sovrascrivere.

- I blocchi funzione astratti possono anche contenere metodi non astratti e/o proprietà non astratte. - Se durante l'ereditarietà non vengono sovrascritti tutti i metodi astratti o tutte le proprietà astratte, l'FB ereditato può essere solo un FB astratto (implementazione passo passo). - Sono consentiti puntatori o riferimenti di tipo FB astratto. Tuttavia, questi possono fare riferimento a FB non astratti e quindi chiamare i loro metodi o proprietà (polimorfismo).

**Differenze tra un FB astratto e un'interfaccia:**

Se un blocco funzione è costituito esclusivamente da metodi astratti e proprietà astratte, allora non contiene alcuna implementazione e quindi presenta alcune somiglianze con le interfacce. Tuttavia, ci sono alcune caratteristiche speciali da considerare in dettaglio. | | Interfaccia | FB Estratto | |:------------ |:----------------| :-------| | supporta l'ereditarietà multipla | + | - | | può contenere variabili locali | - | + | | può contenere metodi non astratti | - | + | | può contenere proprietà non astratte | - | + | | supporta più modificatori di accesso rispetto a public | - | + | | applicabile con matrice | + | solo tramite PUNTATORE |

La tabella può dare l'impressione che le interfacce possano essere quasi completamente sostituite da FB astratti. Tuttavia, le interfacce offrono una maggiore flessibilità perché possono essere utilizzate in diverse gerarchie di ereditarietà.

Quindi, come sviluppatore, vuoi sapere quando utilizzare un'interfaccia e quando utilizzare un FB astratto. La risposta semplice è preferibilmente entrambe allo stesso tempo. Ciò fornisce un'implementazione standard nell'FB di base astratto, facilitando la derivazione. Tuttavia, ogni sviluppatore è libero di implementare direttamente l'interfaccia.**Ejemplo:**

I blocchi funzione dovrebbero essere progettati per la gestione dei dati dei dipendenti. Viene fatta una distinzione tra dipendenti a tempo indeterminato ( FB\_FullTimeEmployee ) e dipendenti a contratto ( FB\_ContractEmployee ). Ogni dipendente è identificato dal proprio nome ( sFirstName ), cognome ( sLastName ) e numero di matricola ( nPersonnelNumber ). A tale scopo vengono fornite le proprietà corrispondenti. Inoltre, è necessario un metodo che restituisca il nome completo, incluso il numero del personale, come una stringa formattata ( GetFullName() ). Il calcolo del reddito mensile viene effettuato utilizzando il metodo GetMonthlySalary().

Lo risolveremo in 3 modi diversi: - 1. Approccio risolutivo: abstract FB!!! metti l'UML!!! - 2. Approccio risolutivo:

Interfaccia!!! metti l'UML!!! - 3. Approccio alla soluzione: combinazione di FB astratto e interfaccia !!! metti l'UML!!!

**Riassunto e conclusioni:**

* Se l'utente non deve creare la propria istanza dell'FB (perché questo non sembra essere utile), allora gli FB astratti o le interfacce sono utili.
* Se si desidera generalizzare a più di un tipo di base, è necessario utilizzare un'interfaccia.
* Se un FB può essere configurato senza implementare metodi o proprietà, si dovrebbe preferire un'interfaccia a un FB astratto.

**Collegamenti FB vs interfaccia astratti:**

* 🔗 [FB abastracto frente a interfaz, stefanhenneken.net](https://stefanhenneken.net/2020/12/13/iec-61131-3-abstract-fb-vs-interface/)
* 🔗 [The ABSTRACT keyword, www.plccoder.com](https://www.plccoder.com/abstract/)
* 🔗 [ABSTRACT concept, infosys.beckhoff.com](https://infosys.beckhoff.com/english.php?content=../content/1033/tc3_plc_intro/6413748235.html&id=)

5. Tabla de Modificadores de acceso

# 5. Tabella modificatori di accesso

|  |  |  |  |
| --- | --- | --- | --- |
| **Modificadores de acceso** | **FUNCTION\_BLOCK - FB** | **METODO** | **PROPRIETA’** |
| **PUBLIC** | Si | Si | Si |
| **INTERNAL** | Si | Si | Si |
| **FINAL** | Si | Si | Si |
| **ABSTRACT** | Si | Si | Si |
| **PRIVATE** | No | Si | Si |
| **PROTECTED** | No | Si | Si |

6. Tipos de variables y variables especiales

# 6. Tipi di variabili e variabili speciali

6.0.1 Variable types and special variables:

The variable type defines how and where you can use the variable. The variable type is defined during the variable declaration.

6.0.2 Further Information:

* [Local Variables - VAR](https://infosys.beckhoff.com/content/1033/tc3_plc_intro/2528755083.html)

• Le variabili locali vengono dichiarate nella parte dichiarativa degli oggetti di programmazione tra le parole chiave VAR e END\_VAR.

• È possibile estendere le variabili locali con una parola chiave attributo.

• È possibile accedere alle variabili locali da leggere dall'esterno degli oggetti di programmazione tramite il percorso dell'istanza. L'accesso in scrittura dall'esterno dell'oggetto pianificazione non è possibile; Questo verrà visualizzato dal compilatore come un errore.

• Per mantenere l'incapsulamento dei dati previsto, si consiglia vivamente di non accedere alle variabili locali di una POU dall'esterno della POU, né in modalità di lettura né in modalità di scrittura. (Altri compilatori di linguaggi di alto livello generano anche operazioni di accesso in lettura su variabili locali come errori.) Inoltre, con i blocchi funzione della libreria non è possibile garantire che le variabili locali di un blocco funzione rimangano invariate durante i successivi aggiornamenti. Ciò significa che il progetto dell'applicazione potrebbe non essere più in grado di essere compilato correttamente dopo l'aggiornamento della libreria.

• Si noti anche qui la regola dell'analisi statica SA0102, che determina l'accesso alle variabili locali per la lettura dall'esterno.

* [Input Variables - VAR\_INPUT](https://infosys.beckhoff.com/content/1033/tc3_plc_intro/2528760459.html)

• Le variabili di ingresso sono variabili di ingresso per un blocco funzione.

• Le variabili VAR\_INPUT vengono dichiarate nella parte dichiarativa degli oggetti di programmazione tra le parole chiave VAR\_INPUT e END\_VAR.

• È possibile estendere le variabili di input con una parola chiave attributo.

* [Output Variables - VAR\_OUTPUT](https://infosys.beckhoff.com/content/1033/tc3_plc_intro/2528765835.html)

• Le variabili di uscita sono variabili di uscita di un blocco funzione.

• Le variabili VAR\_OUTPUT vengono dichiarate nella parte dichiarativa degli oggetti di programmazione tra le parole chiave VAR\_OUTPUT e END\_VAR. TwinCAT restituisce i valori di queste variabili al blocco funzione chiamante. Lì puoi rivedere i valori e continuare a usarli.

• È possibile estendere le variabili di output con una parola chiave attributo.

* [Input/Output Variables - VAR\_IN\_OUT, VAR\_IN\_OUT CONSTANT](https://infosys.beckhoff.com/content/1033/tc3_plc_intro/2528771211.html)
* [Global Variables - VAR\_GLOBAL](https://infosys.beckhoff.com/content/1033/tc3_plc_intro/2528776587.html)

• La sua dichiarazione è possibile solo in GVL (Global Variable List)

* [Temporary Variable - VAR\_TEMP](https://infosys.beckhoff.com/content/1033/tc3_plc_intro/2528781963.html)

• Questa funzionalità è un'estensione dello standard IEC 61131-3.

• Le variabili temporanee vengono dichiarate localmente tra le parole chiave VAR\_TEMP e END\_VAR.

• Le dichiarazioni VAR\_TEMP sono possibili solo nei programmi e nei blocchi funzione.

• TwinCAT reinizializza le variabili temporanee ogni volta che viene richiamato il blocco funzione.

• L'applicazione può accedere solo alle variabili temporanee nella parte di implementazione di un programma o di un blocco funzione.

6.0.3 Variables persistentes

* [Static Variables - VAR\_STAT](https://infosys.beckhoff.com/content/1033/tc3_plc_intro/2528787339.html)

• Questa funzionalità è un'estensione dello standard IEC 61131-3.

• Le variabili statiche vengono dichiarate localmente tra le parole chiave VAR\_STAT e END\_VAR. TwinCAT inizializza le variabili statiche quando il rispettivo blocco funzione viene richiamato per la prima volta.

• È possibile accedere alle variabili statiche solo all'interno dello spazio dei nomi in cui le variabili sono dichiarate (come nel caso delle variabili statiche in C). Tuttavia, le variabili statiche mantengono il proprio valore quando l'applicazione esce dal blocco funzione. È possibile utilizzare variabili statiche, ad esempio come contatori per le chiamate di funzioni.

• È possibile estendere le variabili statiche con una parola chiave attributo.

• Le variabili statiche esistono solo una volta. Ciò vale anche per le variabili statiche in un blocco funzione o metodo del blocco funzione, anche se il blocco funzione viene istanziato più volte.

* [External Variables - VAR\_EXTERNAL](https://infosys.beckhoff.com/content/1033/tc3_plc_intro/2528792715.html)

• Le variabili esterne sono variabili globali che vengono "importate" in un blocco funzione.

• È possibile dichiarare variabili tra le parole chiave VAR\_EXTERNAL e END\_VAR. Se la variabile globale non esiste, viene emesso un messaggio di errore.

• Nel PLC TwinCAT 3 non è necessario dichiarare le variabili come esterne. La parola chiave esiste per mantenere la compatibilità con IEC 61131-3.

• Se, tuttavia, si utilizzano variabili esterne, accertarsi di indirizzare le variabili assegnate (con AT %I o AT %Q) solo nell'elenco delle variabili globali. L'indirizzamento aggiuntivo di istanze di variabili locali porterebbe alla duplicazione nell'immagine di processo.

• Queste variabili dichiarate devono essere dichiarate anche la stessa variabile con lo stesso nome in un GVL (List of

variabili globali)

* [Instance Variables - VAR\_INST](https://infosys.beckhoff.com/content/1033/tc3_plc_intro/2528798091.html)

• TwinCAT crea una variabile VAR\_INST di un metodo non sullo stack del metodo come le variabili VAR, ma sullo stack dell'istanza del blocco funzione. Ciò significa che la variabile VAR\_INST si comporta come altre variabili di istanza del blocco funzione e non viene reinizializzata ogni volta che viene chiamato il metodo.

• Le variabili VAR\_INST sono consentite solo nei metodi di un blocco funzione e l'accesso a tale variabile è disponibile solo all'interno del metodo. È possibile monitorare i valori delle variabili di istanza nella parte dichiarativa del metodo.

• Le variabili di istanza non possono essere estese con una parola chiave di attributo.

* [Remanent Variables - PERSISTENT, RETAIN](https://infosys.beckhoff.com/content/1033/tc3_plc_intro/2528803467.html)

• Le variabili ritentive possono mantenere i propri valori oltre il normale tempo di esecuzione del programma. Le variabili ritentive possono essere dichiarate come variabili RETAIN o anche più strettamente come variabili PERSISTENT nel progetto PLC.

Un presupposto per la piena funzionalità delle variabili RETAIN è un'area di memoria corrispondente nel controller (NovRam). Le variabili persistenti vengono scritte solo quando TwinCAT viene chiuso. Ciò richiede generalmente un UPS corrispondente. Eccezione: le variabili persistenti possono essere scritte anche con il blocco funzione FB\_WritePersistentData.

Se l'area di memoria corrispondente non esiste, i valori delle variabili RETAIN e PERSISTENT vengono persi durante un'interruzione di corrente.

L'istruzione AT non deve essere utilizzata in combinazione con VAR RETAIN o VAR PERSISTENT.

## 6.0.3 Variabili persistenti

È possibile dichiarare variabili persistenti aggiungendo la parola chiave PERSISTENT dopo la parola chiave per il tipo di variabile (VAR, VAR\_GLOBAL, ecc.) nella parte di dichiarazione degli oggetti di programmazione.

I tag PERSISTENT mantengono il loro valore dopo un'interruzione incontrollata, un reset a freddo o un nuovo download del progetto PLC. Quando il programma viene riavviato, il sistema continua a funzionare con i valori memorizzati. In questo caso, TwinCAT reinizializza le variabili "normali" con i loro valori iniziali esplicitamente specificati o con le inizializzazioni predefinite. In altre parole, TwinCAT reimposta solo le variabili PERSISTENT durante un Reset source.

Un esempio di applicazione per le variabili persistenti è un contatore delle ore di funzionamento, che dovrebbe continuare a contare dopo un'interruzione di corrente e quando il progetto PLC viene nuovamente scaricato.

Evitare di utilizzare il tipo di dati POINTER TO negli elenchi di variabili persistenti, poiché i valori degli indirizzi potrebbero cambiare quando il progetto PLC viene nuovamente scaricato. TwinCAT emette i corrispondenti avvertimenti del compilatore. Dichiarare una variabile locale come PERSISTENT in una funzione non ha alcun effetto. La persistenza dei dati non può essere utilizzata in questo modo. Il comportamento durante un ripristino a freddo può essere influenzato dal pragma 'TcInitOnReset'

## 6.0.4 Variabili RETAIN

## È possibile dichiarare variabili RETAIN aggiungendo la parola chiave RETAIN dopo la parola chiave per il tipo di variabile (VAR, VAR\_GLOBAL, ecc.) nella parte di dichiarazione degli oggetti di programmazione.

## Le variabili dichiarate come RETAIN dipendono dal sistema di destinazione, ma sono tipicamente gestite in un'area di memoria separata che deve essere protetta da interruzioni di corrente. Il cosiddetto controller Retain assicura che le variabili RETAIN vengano scritte alla fine di un ciclo PLC e solo nell'area corrispondente della NovRam. La gestione dell'Holding Handle è descritta nel capitolo "Preserting Data" della documentazione C/C++.

## Le variabili RETAIN conservano il loro valore dopo una terminazione incontrollata (mancanza di corrente). Quando il programma viene riavviato, il sistema continua a funzionare con i valori memorizzati. In questo caso TwinCAT reinizializza le variabili

## "normali" con i loro valori iniziali specificati in modo esplicito o con inizializzazioni predefinite. TwinCAT reinizializza le variabili RETAIN su una sorgente di ripristino.

## Una possibile applicazione è un contapezzi in un impianto di produzione, che deve continuare a contare dopo un'interruzione di corrente.

## Se si dichiara una variabile locale come RETAIN in un programma o in un blocco funzione, TwinCAT memorizza questa variabile specifica nell'area di conservazione (come variabile globale RETEAIN). Se dichiari una variabile locale in una funzione come RETAIN, questo non ha effetto. TwinCAT non memorizza la variabile nell'area di attesa.6.0.5 Cuadro general completo

Il grado di ritenzione delle variabili RETAIN viene automaticamente incluso in quello delle variabili PERSISTENT.

|  |  |  |  |
| --- | --- | --- | --- |
| **Dopo il comando in linea** | **VAR** | **VAR RETAIN** | **VAR PERSISTENT** |
| Ripristino a freddo | I valori vengono ripristinati | I valori sono mantenuti | I valori sono mantenuti |
| Ripristino valore origine | I valori vengono ripristinati | I valori vengono ripristinati | I valori vengono ripristinati |
| Scaricare | I valori vengono ripristinati | I valori sono mantenuti | I valori sono mantenuti |
| Cambio online | I valori sono mantenuti | I valori sono mantenuti | I valori sono mantenuti |



* [SUPER](https://infosys.beckhoff.com/content/1033/tc3_plc_intro/2528837771.html)
* [THIS](https://infosys.beckhoff.com/content/1033/tc3_plc_intro/2528843147.html)
* [Variable types - attribute keywords](https://infosys.beckhoff.com/content/1033/tc3_plc_intro/2528848523.html)
* [RETAIN: for remanent variables of type RETAIN](https://infosys.beckhoff.com/content/1033/tc3_plc_intro/2528803467.html)
* [PERSISTENT: for remanent variables of type PERSISTENT](https://infosys.beckhoff.com/content/1033/tc3_plc_intro/2528803467.html)
* [CONSTANT: for constants](https://infosys.beckhoff.com/content/1033/tc3_plc_intro/2529284235.html#2529371275)



6.0.6 Links:

6.0.6 Links:

🔗



[Local Variables - VAR, infosys.beckhoff.com](https://infosys.beckhoff.com/english.php?content=../content/1033/tc3_plc_intro/2528755083.html&id=)

[/](https://infosys.beckhoff.com/english.php?content=../content/1033/tc3_plc_intro/2528755083.html&id=)

🔗



[Instance Variables - VAR\_INST, infosys.beckhoff.com](https://infosys.beckhoff.com/english.php?content=../content/1033/tc3_plc_intro/2528798091.html&id=)

[/](https://infosys.beckhoff.com/english.php?content=../content/1033/tc3_plc_intro/2528798091.html&id=)

🔗



[www.plccoder.com/instance-variables-with-var\_ins](https://www.plccoder.com/instance-variables-with-var_inst/)

[t](https://www.plccoder.com/instance-variables-with-var_inst/)

🔗



[www.plccoder.com/var\_temp-var\_stat-and-var\_cons](https://www.plccoder.com/var_temp-var_stat-and-var_const/)

[t](https://www.plccoder.com/var_temp-var_stat-and-var_const/)

🔗



[Tipos de variables y variables especiale](https://infosys.beckhoff.com/content/1033/tc3_plc_intro/2528749707.html?id=3782076432056683724)

[s](https://infosys.beckhoff.com/content/1033/tc3_plc_intro/2528749707.html?id=3782076432056683724)

•

•

•

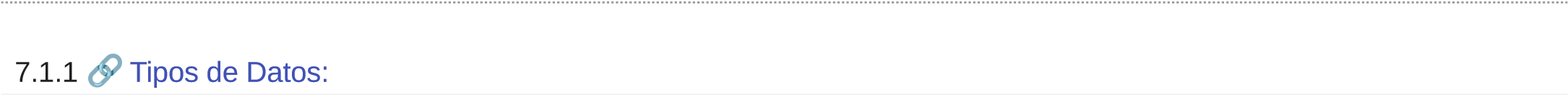
•

•

7. Tipos de Datos

# 7. Tipi di Dati

7.1 Dichiarazione di una variabile:

La dichiarazione delle variabili in CODESYS o TwinCAT includerà: - Un nome di variabile - Due punti - Un tipo di dati - Un valore iniziale facoltativo - Un punto e virgola - Un commento facoltativo

7.1.2 I vantaggi delle strutture dati.

• Il contributo principale delle strutture di dati e dei tipi di dati creati dall'utente è la chiarezza e l'ordine del codice risultante.



* Strutra dati: (STRUCT)
* 🔗 [Extender una Estructura, Infosys Beckhoff](https://infosys.beckhoff.com/content/1033/tc3_plc_intro/3468091787.html?id=592001323464924565)

• Dati utente: UDT (User Data Type) Gli UDT (User Data Type) sono tipi di dati che l'utente crea in base alle esigenze di ogni progetto.



When programming in TwinCAT, you can use different data types or instances of function blocks. You assign a data type to each identifier. The data type determines how much memory space is allocated and how these values are interpreted.

The following groups of data types are available:

## 7.1.3 Standard data types

TwinCAT supports all data types described in the IEC 61131-3 standard.

* BOOL
* Integer Data Types
* REAL / LREAL
* STRING
* WSTRING
* Time, date and time data types
* LTIME

## 7.1.4 Extensions of the IEC 61131-3 standard

* BIT
* ANY and ANY\_
* Special data types XINT, UXINT, XWORD and PVOID
* REFERENCE
* UNION
* POINTER
* Data type \_\_SYSTEM.ExceptionCode

7.1.5 User-defined data types

## 7.1.5 User-defined data types

Note the recommendations for naming objects.

* POINTER
* REFERENCE • ARRAY
* Subrange Types User-defined data types that you create as DUT object in the TwinCAT PLC project tree:
* Structure
* Enumerations
* Alias
* UNION

## 7.1.6 Further Information

* BOOL
* Integer Data Types
* Subrange Types
* BIT
* REAL / LREAL
* STRING
* WSTRING
* Time, date and time data types
* ANY and ANY\_
* Special data types XINT, UXINT, XWORD and PVOID
* POINTER
* Data type \_\_SYSTEM.ExceptionCode
* Interface pointer / INTERFACE
* REFERENCE https://infosys.beckhoff.com/english.php?content=../content/1033/tc3\_plc\_intro/2529458827.html&id=
* ARRAY
* Structure
* Enumerations
* Alias
* UNION



7.1.7 Links Tipos de Datos:

🔗



[12](https://www.youtube.com/watch?v=qh2cC6eOhKw)

[. TwinCAT 3: Standard data type](https://www.youtube.com/watch?v=qh2cC6eOhKw)

[s](https://www.youtube.com/watch?v=qh2cC6eOhKw)

🔗



[help.codesys.com, Tipos de dato](https://help.codesys.com/api-content/2/codesys/3.5.14.0/en/_cds_struct_reference_datatypes/#c2bdb4ccec0a8640e0157fbed-id-b8c8a0ca2bdb4ccdc0a8640e00e8ce32)

[s](https://help.codesys.com/api-content/2/codesys/3.5.14.0/en/_cds_struct_reference_datatypes/#c2bdb4ccec0a8640e0157fbed-id-b8c8a0ca2bdb4ccdc0a8640e00e8ce32)

🔗



[www.infoplc.net, codesys-variable](https://www.infoplc.net/descargas/42-codesys/3418-codesys-variables-globales-persistentes)

[s](https://www.infoplc.net/descargas/42-codesys/3418-codesys-variables-globales-persistentes)

🔗



TC10.Beckhoff TwinCAT3 DUT

.J

[P](https://www.youtube.com/watch?v=UBmHhbZf12s)

•

•

•

•

8. Interfaz fluida

# 8. Interfaz fluida

8.0.1 Links:

🔗



[fluent-code, www.plccoder.co](https://www.plccoder.com/fluent-code/)

[m](https://www.plccoder.com/fluent-code/)

🔗



[fluent-interface-and-method-chaining-in-twincat-](https://twincontrols.com/community/twincat-knowledgebase/fluent-interface-and-method-chaining-in-twincat-3/#post-278)

[3](https://twincontrols.com/community/twincat-knowledgebase/fluent-interface-and-method-chaining-in-twincat-3/#post-278)

•

•

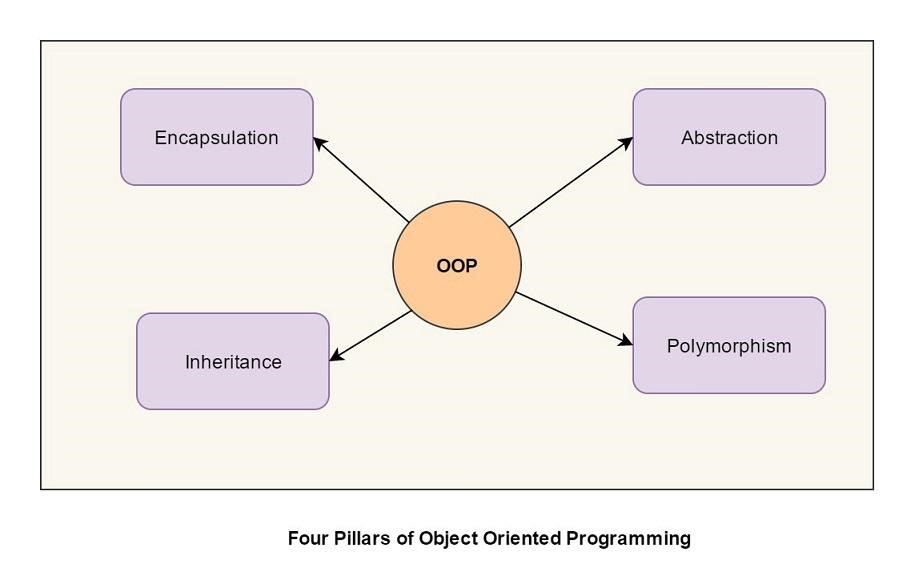
9. Principios OOP

# 9. Principi della OOP

## 9.1 4 Pilastri

Principi OOP: (4 pilastri)

* Astrazione -- Il modo di tradurre qualcosa nel codice per concentrarsi sul suo utilizzo. Non concentrarti tanto su cosa c'è dietro il codice ma sull'uso che ne fa.
* Incapsulmento -- Non tutte le informazioni del nostro oggetto sono rilevanti e/o accessibili all'utente.
* Eredità -- È la qualità di ereditare caratteristiche da un'altra classe. (EXTENDS)
* Polimorfismo -- I molteplici modi in cui un oggetto può ottenere se condivide la stessa classe o interfaccia. (IMPLEMENTS)



9.2 Abstracción

## 9.2 Astrazione

**astrazione**

L'astrazione è il processo di nascondere informazioni importanti, mostrando solo le informazioni più essenziali. Riduce la complessità del codice e isola l'impatto delle modifiche. L'astrazione può essere compresa da un esempio di vita reale: accendere un televisore deve solo richiedere di fare clic su un pulsante, poiché le persone non hanno bisogno di sapere o il processo che attraversa. Anche se tale processo può essere complesso e importante, non è necessario che l'utente sappia come viene implementato. Le informazioni importanti che non sono richieste sono nascoste all'utente, riducendo la complessità del codice, migliorando l'occultamento e la riusabilità dei dati, rendendo così i blocchi funzione più facili da implementare e modificare.

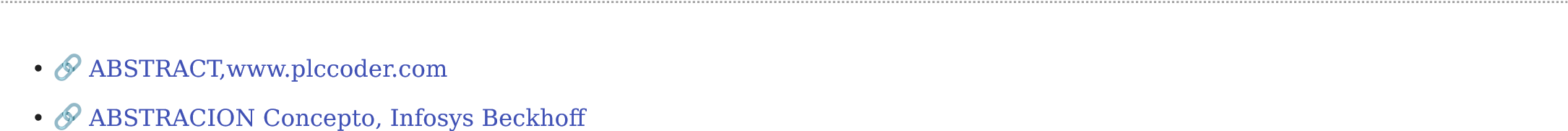
La parola chiave ABSTRACT è disponibile per blocchi funzione, metodi e proprietà. Consente l'implementazione di un progetto PLC con livelli di astrazione.

L'astrazione è un concetto chiave della programmazione orientata agli oggetti. I diversi livelli di astrazione contengono aspetti implementativi generali o specifici..

**Applicazione dell'astrazione:**

È utile implementare funzioni di base o punti in comune di classi diverse in una classe base astratta. Aspetti specifici sono implementati in sottoclassi non astratte. Il principio è simile all'utilizzo di un'interfaccia. Le interfacce corrispondono a classi puramente astratte che contengono solo metodi e proprietà astratti. Una classe astratta può contenere anche metodi e proprietà non astratti.

Regole per l'utilizzo della parola chiave ABSTRACT - I blocchi funzione Abstract non possono essere istanziati. - I blocchi funzione astratti possono contenere metodi e proprietà astratti e non astratti. - I metodi astratti o le proprietà non contengono alcuna implementazione (solo la dichiarazione). - Se un blocco funzione contiene un metodo o una proprietà astratta, deve essere astratta. - I blocchi funzione astratti devono essere estesi per implementare i metodi o le proprietà astratte. - Quindi: un FB derivato deve implementare i metodi/proprietà del suo FB base oppure deve essere definito anche astratto.



9.3 Encapsulamiento

## 9.3 Incapsulamento

**incapsulamento**

L'incapsulamento viene utilizzato per raggruppare i dati con i metodi che operano su di essi e per nascondere i dati al loro interno. una classe, impedendo a persone non autorizzate di accedervi direttamente. Riduci la complessità del codice e aumenta la riusabilità. La separazione del codice consente la creazione di routine che possono essere riutilizzate invece di copiare e incollare il codice, riducendo la complessità del programma principale.

• https://www.plccoder.com/encapsulation/

9.4 Herencia

## 9.4 Eredità

**eredità**

L'ereditarietà consente all'utente di creare classi basate su altre classi. Le classi ereditate possono utilizzare le funzionalità della classe base così come alcune funzionalità aggiuntive che l'utente può definire. Elimina il codice ridondante, impedisce il copia e incolla e facilita l'espansione. Questo è molto utile perché consente alle classi di essere estese o modificate (override) senza modificare l'implementazione del codice della classe base. Cosa hanno in comune un vecchio telefono fisso e uno smartphone? Entrambi possono essere classificati come telefoni. Dovrebbero essere classificati come oggetti? No, poiché definiscono anche le proprietà e i comportamenti di un gruppo di oggetti. Uno smartphone funziona proprio come un normale telefono, ma è anche in grado di scattare foto, navigare in Internet e fare molte altre cose. Quindi, il vecchio telefono fisso e lo smartphone sono classi figlie che estendono la classe del telefono genitore.

Superclase: la classe le cui caratteristiche vengono ereditate è nota come superclasse (classe base o classe genitore).

Subclase: la classe che eredita dall'altra classe è nota come sottoclasse (o classe derivata, classe estesa o classe figlia).

• 🔗 [stefanhenneken.net,iec-61131-3-methods-properties-and-inheritance](https://stefanhenneken.net/2017/04/23/

9.5 Polimorfismo

## 9.5 Polimorfismo

**Polimorfismo**

Il concetto di polimorfismo deriva dalla combinazione di due parole: Poly (Many) e Morphism (Form). Effettua il refactoring di casi di commutazione/dichiarazioni di casi brutti e complessi. Programmazione PLC orientata agli oggetti 8 Il polimorfismo consente a un oggetto di cambiare il suo aspetto e le sue prestazioni a seconda della situazione pratica per essere in grado di svolgere un compito particolare [10]. Può essere statico o dinamico: il polimorfismo statico si verifica quando il tipo dell'oggetto è definito dal compilatore; il polimorfismo dinamico si verifica quando il tipo viene determinato durante l'esecuzione, consentendo a una stessa variabile di accedere a oggetti diversi mentre il programma è in esecuzione. Un buon esempio per spiegare il polimorfismo è un coltellino svizzero (Figura 2.4): Figura 2.4 - Coltellino svizzero Un coltellino svizzero è un singolo strumento che include una serie di risorse che possono essere utilizzate per risolvere diversi problemi. Selezionando lo strumento appropriato, un coltellino svizzero può essere utilizzato per eseguire in modo efficiente una serie di compiti preziosi. Nel modo duale, un semplice blocco sommatore che si adatta per far fronte, ad esempio, ai tipi di dati int, float, string e time è un esempio di risorsa di programmazione polimorfica



Come ottenere il polimorfismo?

Il polimorfismo può essere ottenuto grazie a Interfacce e/o Classi Astratte.

.Interface: (INTERFACE) - Sono un contratto che obbliga una classe ad implementare le proprietà e/oi metodi definiti. Sono un modello (nessuna logica).

.Classi astratte: (ABSTRACT) - Queste sono classi che non possono essere istanziate, possono essere implementate solo tramite ereditarietà.

* Differenze:

**Classi astratte Interfacce**

1.- Limitato a una singola implementazione. 1. Non ha limiti di implementazione.

|  |  |
| --- | --- |
| 2.- Possono definire il comportamento di base. | 2. Espone proprietà e metodi astratti (nessuna logica). |
| \*\*\* |  |
| ### Links Polimorfismo: |  |

* 🔗 [polymorphism, www.plccoder.com](https://www.plccoder.com/polymorphism/)
* 🔗 [abstract, www.plccoder.com](https://www.plccoder.com/abstract/)
* 🔗 [stefanhenneken.net,iec-61131-3-methods-properties-and-inheritance](https://stefanhenneken.net/2017/04/23/
* 🔗 [AT&U, CODESYS - Runtime polymorphism using inheritance (OOP)](https://www.youtube.com/watch?v=oxwkeLggtnM)
* 🔗 [AT&U,CODESYS - Runtime polymorphism using an ITF (OOP)](https://www.youtube.com/watch?v=dcSW0X4gM98)

10. Relaciones

# 10. Relazioni

.Relazioni:

Vediamo due tipi di relazioni:

* Associazione.
* Uno a uno: Una classe mantiene un'associazione uno a uno con un'altra classe..
* Uno a tanti: Una classe mantiene un'associazione con un'altra classe attraverso una raccolta..
* Tanti a tanti: L'associazione avviene su entrambi i lati attraverso una raccolta.
* Collaborazion.

• La collaborazione avviene attraverso il riferimento di una classe al fine di raggiungere un obiettivo.

11. SOLID

# 11. SOLID

11.1

SOLID



- Proposte da Robert C. Martin nel 2000. - Sono raccomandazioni per scrivere un codice sostenibile, manutenibile, scalabile e robusto. - Benefici:

- Alta coesione. Collaborazione tra le classi. - Accoppiamento basso. Impedire che una classe dipenda pesantemente da un'altra classe.

• Principio di responsabilità unica: una classe deve avere una ragione per esistere ma non per cambiare.

• Principio aperto/chiuso: le parti del software devono essere aperte per l'estensione ma chiuse per la modifica.

• Principio di sostituzione di Liskov: le classi dei sottotipi dovrebbero essere sostituibili dalle loro classi genitore.

• Principio della segregazione dell'interfaccia: diverse interfacce funzionano meglio di una.

• Principio di inversione delle dipendenze: le classi di alto livello non dovrebbero dipendere dalle classi di basso livello.



Oltre i principi SOLID, esistono altri principi come: - Keep It Simple, Stupid (KISS). - Don't Repeat Yourself (DRY). Law Of Demeter (LOD). - You Ain't Gonna Need It (YAGNI).

Tutti questi principi hanno l'obiettivo comune di migliorare la manutenibilità e la riusabilità del software.

**Links:**

* 🔗 [Cómo explicar conceptos de programación orientada a objetos a un niño de 6 años](https://www.freecodecamp.org/news/object-oriented-programming-concepts-21bb035f7260/#:~:text=The-20four-20principles-20of-20object,abstraction-2C-20inheritance-2C-20and-20polymorphism.)

11.2 Principio de Responsabilidad Única

## 11.2 Principio di Responsabilità Unica

**Principio di Responsabilità unica:**

**Links:**

• 🔗 [stefanhenneken.net,iec-61131-3-solid-the-single-responsibility-principle](https://stefanhenneken.net/2022/03/10/iec-61131-3-solid-the-single-responsibility-principle/)

11.3 Principio de Abierto/Cerrado

## 11.3 Principio Aperto/Chiuso

**Principio Aperto/Chiuso**

La definizione del principio aperto/chiuso Il principio aperto/chiuso (OCP) è stato formulato da Bertrand Meyer nel 1988 e afferma:

Un'entità software deve essere aperta alle estensioni, ma allo stesso tempo chiusa alle modifiche. Entità software:

Ciò significa una classe, un blocco funzione, un modulo, un metodo, un servizio, ...

Aperto: il comportamento dei moduli software deve essere estensibile.

Chiuso: l'espandibilità non dovrebbe essere raggiunta modificando il software esistente.

Quando Bertrand Meyer definì il principio aperto/chiuso (OCP) alla fine degli anni '80, l'attenzione era rivolta al linguaggio di programmazione C++. Utilizzava l'ereditarietà, ben nota nel mondo orientato agli oggetti. La disciplina orientata agli oggetti, che all'epoca era ancora giovane, prometteva grandi miglioramenti in termini di riusabilità e manutenibilità, consentendo l'utilizzo di classi concrete come classi base per nuove classi.

Quando Robert C. Martin ha rilevato il principio da Bertrand Meyer negli anni '90, lo ha implementato tecnicamente in modo diverso. Il C++ consente l'uso dell'ereditarietà multipla, mentre l'ereditarietà multipla si trova raramente nei linguaggi di programmazione più recenti. Per questo motivo, Robert C. Martin si è concentrato sull'uso delle interfacce. Maggiori informazioni su questo possono essere trovate nel libro (link pubblicitario Amazon \*) Clean Architecture: il manuale pratico per la progettazione di software professionale.

Tuttavia, l'adesione al principio aperto/chiuso (OCP) comporta il rischio di un'ingegnerizzazione eccessiva. L'opzione delle estensioni dovrebbe essere implementata solo dove specificamente necessario. Il software non può essere progettato in modo tale che tutte le estensioni immaginabili possano essere implementate senza apportare modifiche al codice sorgente.

!!! Questo link è in tedesco, metti il ​​link in inglese quando esce:!!!

**Links:**

* 🔗 [stefanhenneken.net, iec-61131-3-solid-das-open-closed-principle](https://stefanhenneken.net/2023/03/09/iec-61131-3-solid-das-open-closed-principle/)
* 🔗 [stefanhenneken.net, EC 61131-3: SOLID – The Open/Closed Principle](https://stefanhenneken.net/2023/04/06/iec-61131-3-solid-the-open-closed-principle/)

11.4 Principio de Sustitución de Liskov

## 11.4 Principio di sostituzione di Liskov

**Principio di sostituzione di Liskov**

**Links:**

• 🔗 [stefanhenneken.net,iec-61131-3-solid-the-liskov-substitution-principle](https://stefanhenneken.net/2022/09/27/iec-61131-3-solid-the-liskov-substitution-principle/)

11.5 Principio de Segregación de Interfaz

## 11.5 Principio di segregazione dell'interfaccia

**Principio di segregazione dell’interfaccia**

**Links:**

• 🔗 [stefanhenneken.net,iec-61131-3-solid-the-interface-segregation-principle](https://stefanhenneken.net/2023/02/25/iec-61131-3-solid-the-interface-segregation-principle/)

11.6 Principio de Inversión de Dependencia

## 11.6 Principio di inversione delle dipendenze

**Principio di inversione delle dipendenze:**

**Links:**

• 🔗 [stefanhenneken.net,iec-61131-3-solid-the-dependency-inversion-principle](https://stefanhenneken.net/2022/02/09/iec-61131-3-solid-the-dependency-inversion-principle/)

12. UML

# 12. UML

## 12.1 UML

**UML**

• https://www.plccoder.com/twincat-uml-class-diagram/

12.2 Class UML

## 12.2 Class UML

**Class UML**

La gerarchia di ereditarietà può essere rappresentata sotto forma di un diagramma. L'Unified Modeling Language (UML) è lo standard stabilito in questo settore. UML definisce diversi tipi di diagrammi che descrivono sia la struttura che il comportamento del software.

Un buon strumento per descrivere la gerarchia di ereditarietà dei blocchi funzione è il diagramma delle classi.

I diagrammi UML possono essere creati direttamente in TwinCAT 3. Le modifiche al diagramma UML hanno un effetto diretto sui POU. Pertanto, i blocchi funzione possono essere modificati e modificati tramite il diagramma UML.

Ciascun riquadro rappresenta un blocco funzione ed è sempre diviso in tre sezioni orizzontali. La sezione superiore mostra il nome del blocco funzione, la sezione centrale elenca le sue proprietà e la sezione inferiore elenca tutti i suoi metodi. In questo esempio, le frecce mostrano la direzione dell'ereditarietà e puntano sempre verso il blocco funzione padre.

**Links UML listado de referencias:**

* 🔗 [stefanhenneken.net, UML Class](https://stefanhenneken.net/2017/04/23/iec-61131-3-methods-properties-and-inheritance/)
* 🔗 [www.lucidchart.com/tutorial-de-diagrama-de-clases-uml](https://www.lucidchart.com/pages/es/tutorial-de-diagrama-de-clases-uml)
* 🔗 [www.edrawsoft.com/uml-class-diagram-explained](https://www.edrawsoft.com/es/article/uml-class-diagram-explained.html)
* 🔗 [blog.visual-paradigm.com/what-are-the-six-types-of-relationships-in-uml-class-diagrams](https://blog.visual-paradigm.com/es/what-are-the-six-types-of-relationships-in-uml-class-diagrams/)
* 🔗 [Ingeniería del Software: Fundamentos de UML usando Papyrus](https://www.udemy.com/course/ingenieria-del-software-fundamentos-de-uml-usando-papyrus/learn/lecture/30833780?start=11#overview)
* 🔗 [plantuml.com/class-diagram](https://plantuml.com/es/class-diagram)
* 🔗 [www.planttext.com](https://www.planttext.com/)
* 🔗 [UML Infosys Beckhoff](https://infosys.beckhoff.com/english.php?content=../content/1033/tf1910_tc3_uml/1510519307.html&id=)

12.3 StateChart UML

## 12.3 StateChart UML

12.3.1 state chart:

13. Tipos de Diseño para programacion de PLC

# 13. Tipi di design per la programmazione PLC

13.0.1 Tipi di design per la programmazione PLC:

Ingegneria di sviluppo per la programmazione OOP - Design by Component, Unit, Device, Object... - Gli oggetti sono le unità di base della programmazione orientata agli oggetti. - Un componente fornisce servizi, mentre un oggetto fornisce operazioni e metodi. Un componente può essere compreso da tutti, mentre un oggetto può essere compreso solo dagli sviluppatori. - Le unità sono i più piccoli gruppi di codice che possono essere mantenuti ed eseguiti in modo indipendente - Design by Unit Testing. - Progettazione in UML.

Units: (Esempio di unità): -FCAnalogSensor -FBGenericUnit !!!

Punti che si posso includere nel corso!!

* Basic of Structured Text programming Language
* Modular Design
* Classes
* Methods
* Properties
* Inheritance
* Polymorphism
* Access Specifiers
* Pointers and References
* Interfaces and Abstractions
* Advanced State Pattern
* Wrappers and Features
* Layered Design
* Final Project covering a real-world problem to be solved using OOP
* [Testo strutturato (ST), Testo strutturato esteso (ExST)](https://content.helpme-codesys.com/en/CODESYS-20Development-20System/_cds_f_programming_language_st.html)

14. Patrones de Diseño

# 14. Modelli di progettazione

## 14.1 Modelli di progettazione

**MODELLI DI PROGETTAZIONE:**

I modelli di progettazione sono soluzioni generali e riutilizzabili a problemi comuni riscontrati nella programmazione del software. Nella programmazione orientata agli oggetti, esistono molti modelli di progettazione che possono essere applicati per migliorare la modularità, la flessibilità e la manutenibilità del codice. Alcuni esempi di design pattern che possono essere applicati nella programmazione PLC includono il pattern Singleton, il pattern Factory Method, il pattern Observer e il pattern Strategy. Ad esempio, il modello Singleton viene utilizzato per garantire che esista una sola istanza di una determinata classe nell'intero programma. Ciò può essere utile nella programmazione PLC quando si desidera garantire che vi sia una sola istanza attiva dell'oggetto che controlla un determinato processo o dispositivo. Il modello Factory Method viene utilizzato per creare un'istanza di oggetti senza specificare in modo esplicito la classe concreta di cui creare un'istanza. Questo può essere utile nella programmazione PLC quando si vogliono creare oggetti secondo le specifiche esigenze del programma. Il modello Observer viene utilizzato per stabilire una relazione uno-a-molti tra gli oggetti, in modo che quando un oggetto cambia stato, tutti gli oggetti correlati vengono automaticamente notificati. Questo modello può essere molto utile nella programmazione PLC per stabilire relazioni tra diversi componenti del sistema, come sensori e attuatori. Il pattern Strategy viene utilizzato per definire un insieme di algoritmi intercambiabili e quindi incapsulare ciascuno di essi come un oggetto. Questo pattern può essere utile nella programmazione PLC quando si vuole modificare dinamicamente il comportamento del sistema in base alle condizioni ambientali. In breve, i design pattern sono uno strumento molto utile per migliorare la qualità del codice nella programmazione PLC e possono essere applicati con successo nella programmazione orientata agli oggetti per PLC.

1

2

3

4

5

6

“Los patrones de diseño son

descripciones de objetos y clases

conectadas que se personalizan para

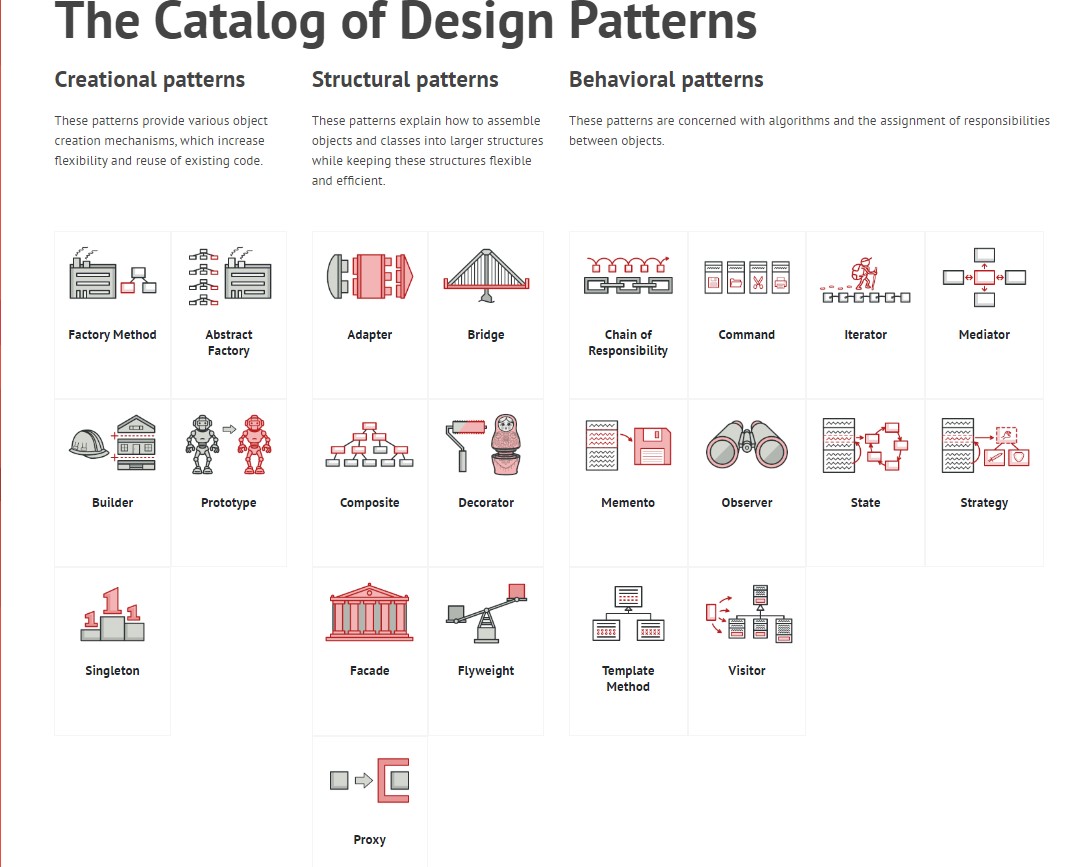
resolver un problema de diseño

general en un contexto particular”.

-

Gang of Four









Clasificación según su propósito: Los patrones de diseño se clasificaron originalmente en tres grupos:

* Creacionales.
* Estructurales.
* De comportamiento.

Clasificación según su ámbito:

* De clase: Basados en la herencia de clases.
* De objeto: Basados en la utilización dinámica de objetos.

Patrones creacionales:

* Builder
* Singleton
* Dependency Injection
* Service Locator
* Abstract Factory
* Factory Method

Patrones estructurales:

* Adapter
* Data Access Object (DAO)
* Query Object
* Decorator
* Bridge

Patrones de comportamiento:

* Command
* Chain of Responsibility
* Strategy
* Template Method
* Interpreter
* Observer
* State
* Visitor
* Iterator

**Links de Patrones de Diseño:**

* [IEC 61131-3: SOLID – The Interface Segregation Principle](https://stefanhenneken.net/2023/02/25/iec-61131-3-solid-the-interface-segregation-principle/#more-2505)

14.2 Patron de Estrategia

## 14.2 Patron de Estrategia

TwinCAT with Head First Design Patterns Ch.1 - IntroStrategy Pattern.docx

15. Links

# 15. Links

## 15.0.1 Links

15.0.2 Mención a la Fuentes Links empleadas para la realización de esta Documentación: