Agenda

ID	Topic
00	Introduction to the workshop
01	Introduction to AX Code IDE
02	Get started with your first AX Project
03	Introduction to ST Programming
04	Loading and Debugging
05	**OOP Elements of ST**
06	Unit Testing
07	Tools for commissioning
08	Package management

Prerequisites

Basic ST programming knowledge

What will you learn in this chapter ?

NOTE: This chapter is intended for users who want to dive into more advanced programming topics. You can skip this chapter for now, if you just want to get started with the basics of AX and come back later to learn more about how OOP helps you to structure your code.

- Classes
- Encapsulation
- Inheritance
- Accessors
- Methods
- Polymorphism
- Interfaces
- Abstract

OOP

Object oriented programming

Introduction

Some facts about OOP

Object-oriented programming (OOP) is a programming paradigm based on the concept of "objects", which can contain **data** and **code**: data in the form of fields (often known as **attributes** or **properties**), and code, in the form of **methods**.
A feature of objects is that an object's own **procedures** can **access** and often **modify** the data fields of itself (objects have a notion of this or self).
In OOP, computer programs are designed by making them out of objects that **interact** with one another.

History

The basics of OOP were developed in the 60ties and 70ties during the software crisis.
In this time, the software costs exceeded the hardware costs. In the 80ties, OOP became popular in computer programming.

Main concepts of OOP

- Encapsulation
- Inheritance
- Polymorphism

How to facilitate the understanding of these properties ?

Seeing directly the structure on which programmed objects are based.

Let's start ➡

Class

What is a class ?

A class is an extensible program-code-template for creating objects.

- It has properties (attributes or fields)
- It has a behavior (methods as code)
- The created **object** of a class is called **object instance**

Difference to FUNCTION BLOCK

A FUNCTION BLOCK is similar to a CLASS with one method

	Function Block	Class
can be instantiated	yes	yes
occupy memory	yes	yes
method supported	no	yes
access modifier[1]	no (just private)	yes
can be inherited	no	yes

[1] in ST they're private. In SCL on TIA Portal they're public

Class declaration in ST

```
CLASS Valve
    // Variable declaration
    VAR // Optional access modifier
        // member variables
    END_VAR
    VAR CONSTANT
        // constants
    END_VAR

    // Implementation of Methods
    METHOD PUBLIC Open
        ;
    END_METHOD
    METHOD PUBLIC Close
        ;
    END_METHOD
END_CLASS
```

Methods and THIS-Operator

```
CLASS Valve
    // Implementation of Methods
    METHOD PUBLIC Open
        THIS.CalcSomething();  // call a private method
    END_METHOD

    METHOD PRIVATE CalcSomething : REAL
        CalcSomething := 0.1231;
    END_METHOD
END_CLASS
```

Methods can be called within their class. Then they have to be called with the THIS-Operator

Instantiate a class in ST

Define an object instance in **VAR_GLOBAL** full qualified access:

```
CONFIGURATION
    VAR_GLOBAL
        v : Valve;
    END_VAR
END_CONFIGURATION
```

Initializer

```
CLASS Tank
    VAR PUBLIC
        volume : LREAL;
    END_VAR
END_CLASS

CONFIGURATION
    VAR_GLOBAL
        t : Tank := (volume := 100.0);
    END_VAR
END_CONFIGURATION
```

Public variables can be initialized while declaration.

Method Call

```
PROGRAM
    VAR_EXTERNAL
        v : Valve;
    END_VAR

    v.Open();
END_PROGRAM
```

Remarks

Classes can not instantiated in:

- **VAR_TEMP** section
- **VAR_INPUT** section
- **VAR_OUTPUT** section

- In ST, classes can only be instantiated during compile time.
- In opposite to other languages, ST can not create instances during runtime.
- There are no constructors for classes in ST

Encapsulation

What is encapsulation ?

One of the main concepts of OOP is encapsulation

- It refers to the bundling of data (variables) and methods (functions) that operate on the data into a single unit, typically a class.
- In a properly designed object, the state can be changed. This is achieved by providing public methods that operate on the values of private member variables.
- It also involves restricting direct access to some of an object's components, which is a means of preventing unintended interference and misuse of the data.
- Encapsulation ensures that private information
    - is not exposed
    - cannot be modified except through calls of methods

Key benefits

1. **Control of Data:** By controlling access to the data, the class can ensure that it is used only in intended ways.
2. **Maintainability:** Changes to the implementation details of a class can be made without affecting other parts of the program that uses the class.
3. **Reusability:** Encapsulated classes can be more easily reused across different programs.
4. **Modularity:** Encapsulation leads to better organization of code, making it more modular.

Example

Let's illustrate encapsulation with a simple example of a BankAccount class:

- An account has a balance and a number that identifies the account
- How can I change the state of the balance of a specific account?

Private Attributes:

accountNumber and balance are private attributes of the BankAccount class and can not be accessed from the outside of the class instance (e.g. can not be changed by any other method than the methods of the same BankAccound class instance)

```
// Declaration of the BankAccount class
Class BankAccount
VAR
// Private attributes
    accountNumber : INT;
    balance : REAL;
END_VAR
```

Public Methods:

GetBalance method returns the current balance.

Deposit method adds a valid amount to the balance and returns TRUE if the deposit was successful, otherwise FALSE.

```
// Public methods
METHOD GetBalance : REAL
VAR_OUTPUT
    GetBalance : REAL;
END_VAR
    GetBalance := balance;
END_METHOD

METHOD Deposit : BOOL
VAR_INPUT
    amount : REAL;
END_VAR
VAR_OUTPUT
    Deposit : BOOL;
END_VAR
    IF amount > 0 THEN
        balance := balance + amount;
        Deposit := TRUE;
    ELSE
        Deposit := FALSE;
    END_IF;
END_METHOD
```

Withdraw method subtracts a valid amount from the balance and returns TRUE if the withdrawal was successful, otherwise FALSE.

```
METHOD Withdraw : BOOL
VAR_INPUT
    amount : REAL;
END_VAR
VAR_OUTPUT
    Withdraw : BOOL;
END_VAR
    IF (amount > 0) AND (amount <= balance) THEN
        balance := balance - amount;
        Withdraw := TRUE;
    ELSE
        Withdraw := FALSE;
    END_IF;
END_METHOD
```

Initialization:

The Init method is used to set up the initial values of accountNumber and balance when an instance of the BankAccount class is created. This method can act as a substitute for a constructor.

```
// Initialization method to set up initial values
METHOD Init
VAR_INPUT
    initAccountNumber : INT;
    initBalance : REAL;
END_VAR
    accountNumber := initAccountNumber;
    balance := initBalance;
END_METHOD
END_CLASS
```

Benefits

Controlled Access:

- The Deposit method ensures only positive amounts are added to the balance.
- The Withdraw method ensures only valid amounts are subtracted, and only if there are sufficient funds.

Accessors

For methods and properties

Accessor	These attributes can be accessed
PRIVATE	only inside the class
PROTECTED	by derived classes
PUBLIC	by everybody

Remark: In ST the accessor for methods and properties by default is always PROTECTED

Examples for Accessors

```
NAMESPACE ContolModules
    CLASS Valve
        VAR PRIVATE
            _anyPrivate : BOOL;
        END_VAR
        VAR PUBLIC
            _anyPublic : BOOL;
        END_VAR
        VAR PROTECTED
            _anyProtected : BOOL;
        END_VAR
        METHOD PRIVATE Check
            ;
        END_METHOD
        METHOD PUBLIC Open
            ;
        END_METHOD
        METHOD PROTECTED Close
            ;
        END_METHOD
    END_CLASS
END_NAMESPACE
```

Hands On (OOP01)

Goal

Using the introduction to ST exercises as starting point:

Transform Valve and Tank Function Blocks into classes

Step	Description
1	Small change: In the tank class, control the valves independently: Open inlet/outlet valve separately (different methods)
2	Then, create the object that instance the class in configuration.
3	Call your object instance in an example program.
4	Compile and download it to the PLC

Method	Functionality
Open()	Open the valve
Close()	Close the valve
GetState : ValveState	returns the state Undefined, Open, Close (Hint: Enumeration)
WriteCyclic(ctrlOpen : BOOL)	for the activation of the digital output. true when valve opened, false when valve is closed

Advice: do not change the function block. Choose a different namespace instead

Example strucuter for the class and Enumeration Type

```
NAMESPACE FluidHandlingClass
    CLASS ValveClass
        //VARIABLES
        _ctrlOpen : BOOL;
        _state : ValveState;

        //METHODS
        Open();
        Close();
        GetState() : ValveState;
        WriteCyclic(VAR_OUTPUT: (ctrlOpen : BOOL));
    END_CLASS

    //ENUM
    TYPE ValveState
       STATES: Open, Closed, Error, Undefined
    END_TYPE
END_NAMESPACE
```

Create the Class Tank

Transform the function block Tank into a class

```
NAMESPACE FluidHandlingClass
    CLASS Tank
        inletValve : Valve
        outletValve : Valve
        Fill()
        Emptying()
        Flush()
        Close()
    END_CLASS
END_NAMESPACE
```

Method	InletValve	OutletValve
OpenInlet()	opened	unchanged
CloseInlet()	closed	unchanged
OpenOutlet()	unchanged	opened

Interfaces

What is an interface ?

An interface is a contract that defines a set of methods **without implementing them**. Classes that implement the interface **must** provide the method implementation.

Main features

Contractual framework: Specifies methods that must be implemented by any class that uses the interface.

Abstraction: Provides a way to define capabilities without specifying how they should be implemented.

Multiple inheritcance: Allows a class to inherit from multiple interfaces, enabling more flexible design.

Decoupling: Promotes loose coupling between classes by separating the definition of methods from the implementation.

Testability: Facilitates easier unit testing by allowing mock implementations of interfaces.

Reusability: Promotes code reuse as different classes can implement the same interface in various ways.

NOTE

Some of the concepts mentioned in the interface features will be seen on next points/chapters

Benefits

Standardization: Ensures a consistent method signature across different classes.
Interoperability: Enables different classes to work together through a common interface.
Flexibility: Allows changes in implementation without affecting the code that relies on the interface.
Maintainability: Simplifies the maintenance of large codebases by providing clear method contracts.

Example: Interface definition

```
INTERFACE IGeometricCalculation
    METHOD Perimeter : LREAL
    END_METHOD

    METHOD Area : LREAL
    END_METHOD
END_INTERFACE
```

Example: Interface implementation

```
CLASS Rectangle IMPLEMENTS IGeometricCalculation
    VAR PUBLIC
        side1 : LREAL;
        side2 : LREAL;
        END_VAR
    METHOD PUBLIC Perimeter : LREAL


        Perimeter := 2 * side1 + 2 * side2;
    END_METHOD

    METHOD PUBLIC Area : LREAL
        Area := side1 * side2;
    END_METHOD
END_CLASS

CLASS Triangle IMPLEMENTS IGeometricCalculation
    VAR PUBLIC
        side1 : LREAL;
        side2 : LREAL;
        side3 : LREAL;
        height : LREAL;
    END_VAR
    METHOD PUBLIC Perimeter : LREAL
        Perimeter := side1 + side2 + side3;
    END_METHOD

    METHOD PUBLIC Area : LREAL

        Area := (side1 * height) / 2;
    END_METHOD
END_CLASS
```

Instanciating the classes

```
CONFIGURATION MyConfiguration
    TASK Main(Interval := T#1000ms, Priority := 1);
    PROGRAM P1 WITH Main: MyProgram;
    VAR_GLOBAL
        tri : Triangle :=(side1 := 3, side2 := 3, side3 := 2);
        rect : Rectangle :=(side1 := 4, side2 := 5);
    END_VAR
END_CONFIGURATION
```

Rules when using interfaces

Method Signature:
The method signature in the class must exactly match the method signature in the interface.
Parameter Matching:
If the interface defines a method with parameters, the implementation in the class must have the same parameters in type, order, and number.
No Additional Parameters:
You cannot add additional parameters in the implementation of a method that are not specified in the interface.

Hands On (OOP02)

Goal

Using the solution for the HandsOn1 as the starting point:

- Create two interfaces, one for the tanks and the other for the valves.
- This interface will work as a "contract" that the classes must implement.
- In this case, we are going to include all the methods implemented in the HandsOn1 in the interface.

Valve Interface

Method

Open()

Close()

GetState : ValveState

WriteCyclic(ctrlOpen : BOOL)

Advice: remember that the interface does not include any implementation, only the definition.

Tank Interface

Method

OpenInlet()

OpenOutlet()

CloseInlet()

CloseOutlet()

Fill()

Flush()

Close()

Emptying()

WriteCyclic(Capacity : REAL)

Advice: remember that the interface does not include any implementation, only the definition.

Hands On (OOP03)

Goal

Developing program skills is achieved by practicing and trying to solve problems yourself. The next challenge is to apply what you have learned since the beginning of this training.

The proposal is to **develop a program that calculates the volume of a tank**.

Here you will read physical entries, write on outputs, use timers, evaluate cases...

Using the solution for the HandsOn2 as the starting point:

- **1:** Create a TYPE for the Tank State as the Valve one.
- **2:** Create a second class: TankWithVolume that implements also the interface ITank created in the HandsOn2.
- **3:** Create a program that calculates and displays as an output the current volume of the tank .

TYPE for Tank State

Status

Filling

Emptying

Flushing

Closed

Advice: remember that the starting state is closed and you should include it on the definition.

Tank with Volume

This tank implements also the ITank interface created on the previous hands On.
It also has a new parameter **volume** given as an entry by the user with the tank volume as a REAL.
To handle the different status transitions, it will have entries to activate the different states: fill, empty, flush and close.
In addition, it has three new methods:
WriteCyclic(Capacity : REAL)
Capacity : REAL that returns a real with the percentage of capacity of the volume of the tank.
ReadCyclic(Fill : BOOL, Empty : BOOL, Flush : BOOL, Close : BOOL) to read the entries.

Program for calculating and displaying

Structure the program as:

Declaration of the external variables that you will use on the program: valves, tank, phyisical entries...

Then, declarate the complements that you will need to execute the program. For example, a timer, the filling and emptying rates...

Initialize the variables and read the entries that you need.

Evaluate the different tank phases and update the current volume in each phase. We define here that the filling/emptying is **5 L/s**. So every second we activate and reinitialize a timer and increment/decrement the fill value by 5 units.

Write on the outputs.

Inheritance

What is inheritance?

Inheritance is a fundamental principle where a new class, called the derived or child class, inherits properties and behaviors (methods) from an existing class, known as the base or parent class. This mechanism promotes code reuse and establishes a natural a hierarchy between classes.

Benefits of Inheritance

Reusabilty: Inheritance allows the reuse of existing code. Common functionality can be defined in a base class and inherited by multiple derived classes, reducing redundancy.

Maintainability: Changes made to a base a class are automatically inherited by derived classes, making the code easier to update and maintain.

Extensibility: New functionalities can be added to existing classes without modifying them. Derived classes can extend base classes by adding new methods and properties.

Abstraction: Simplifies complex systems by defining common behaviors in base classes, while derived classes provide specific implementations.

Polymorphism: Allows objects of different classes to be treated as objects of a common base class, enabling flexible and interchangeable code.

EXAMPLE

```
CLASS vehicle
    VAR
        Brand : STRING;
        Model : STRING;
        Speed : INT;
        FuelLevel : INT;
    END_VAR

    METHOD PUBLIC SetSpeed
        VAR_INPUT
            newSpeed : INT;
        END_VAR
        Speed := newSpeed;
    END_METHOD
END_CLASS

CLASS car EXTENDS VEHICLE
    VAR
        NumberOfDoors : INT;  //New propertie
    END_VAR

    METHOD PUBLIC SetFuelLevel  //New Method
        VAR_INPUT
            newFuelLevel : INT;
        END_VAR

        FuelLevel := newFuelLevel;
    END_METHOD
```

Hands On (OOP04)

Goal

Using the solution for the HandsOn3 as the starting point:

- Applying **inheritance** create a class: ComplexValve that has an entry: Regulator and instead of boolean the regulation can be adjusted from 0 to 100 for both states, open and closed.

For example: if the regulator for the entry valve is 30, the valve will be opened the 30%.

- Applying **inheritance** create a file: TankWithShape that has an two classes: CilindricTank and CubeTank. These classes extend the properties of the TankWithVolume but now the volume is calculated with a method called VolumeCalculator and the result stored in the propertie volume.
    - For the cilindric tank it is needed to provide to the method the properties: radium and height
    - For the cube tank only a single propertie (is a regular cube): side_length

- With this new valves, **modify the previous volume calculator**. To guide you through the process and facilitate the understanding:
    - You need to modify the filling/emptying rate by multiplying the rate plus the regulation value. This regulation value is a value that is needed to read from an entry of the valve in the first program lines after the declaration of variables.

- To use the new classes tankCube/tankCilinder, choose your favourite and provide the properties in the declaration. Then, in the intialization call the method to Calculate the tank volume that writes on the tank propertie volume.

For example: If the regulation is 50% on the entry valve. In this case, the filling rate will be: 5 * 50/100 = 2.5 L/s. So here, every second that the timer is executed you add to the current_volume := current_volume + filling_rate * MAX_FILLING_RATE;

- **HINT:** Now for the **flushing state** you must take into account that the filling and emptying rates can be different. So in this state the tank can be filled or emptied too.

What did you learn

In this section you learned about...

- what OOP and why it is widely used in software development.
- about OOP features in Structured Test
- how to structure your code to follow the OOP paradigm