OOP WITH WINDEV

How to start

Maxime Keltsma

Table des matières

1 Abstract classes and interfaces

These two concepts are very close one of the other and even swappable. Reminder of these two concepts according to the OOP standards:

Abstract class :

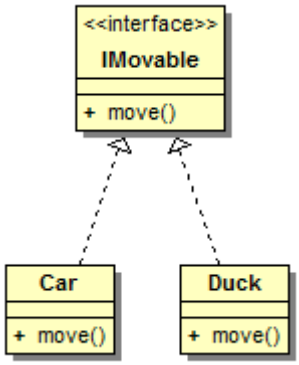
- It is a class defined with the **abstract** keyword.
- Such a class cannot instantiate objects. It can only be used to derive other classes.
- Can define methods implemented or not implemented. In the latter case the method is said to be **abstract**.
- A class must be abstract if it has at least one abstract method.
- The abstract methods are implemented by sub classes.

Interface :

- An interface is defined with the keyword **interface**.
- This is a set of methods that only the signature is provided (the method name, input parameters types and return value type).
- Classes that implement an interface are required to implement the methods defined in this interface.
- A class can implement multiple interfaces.

In the current version of Windev (version 18) the concepts of abstract class and interface does not exist. It will be necessary to get something that can play the role of an abstract class or an interface.

Suppose we have to implement the following class diagram:



This is classically a method (move()), whose signature is defined in an interface, and which must be implemented within two classes (Car and Duck). Of course each subclass will have a different implementation of move():

The Car class will return the string "I drive".

The Duck class will return the string "I fly".

1.1 Implementation with Windev

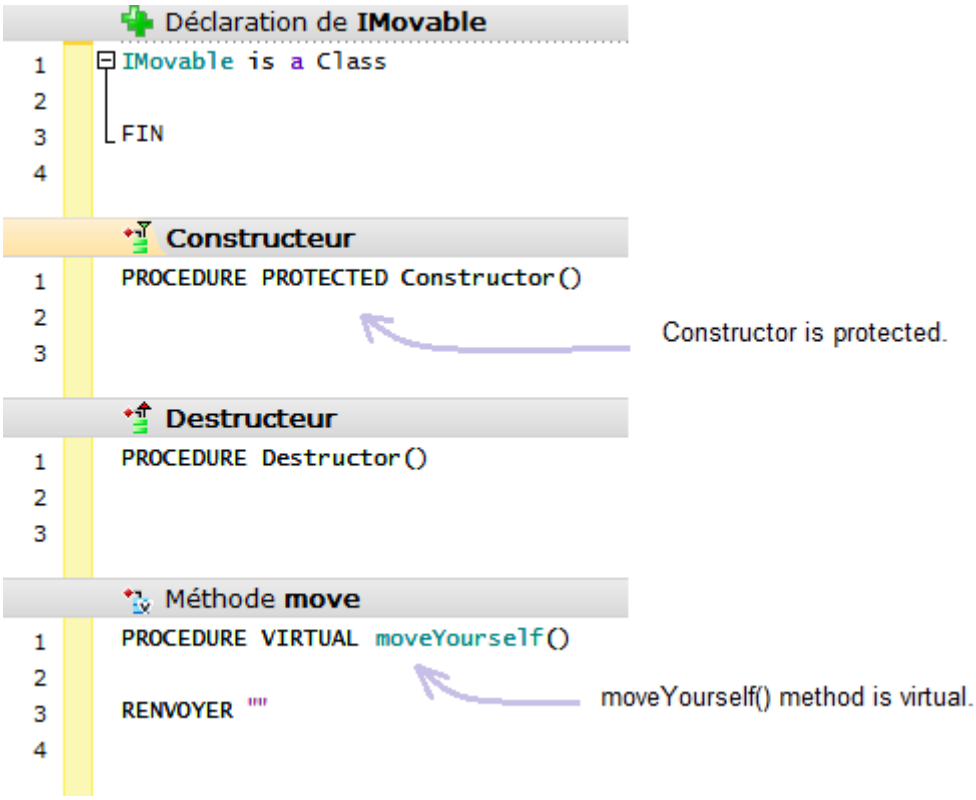
With Windev (version 18), as previously indicated, we cannot create explicitly interfaces nor true abstract class we will therefore create a class and make a few changes that will make it a kind of abstract class / interface.

The class can have:

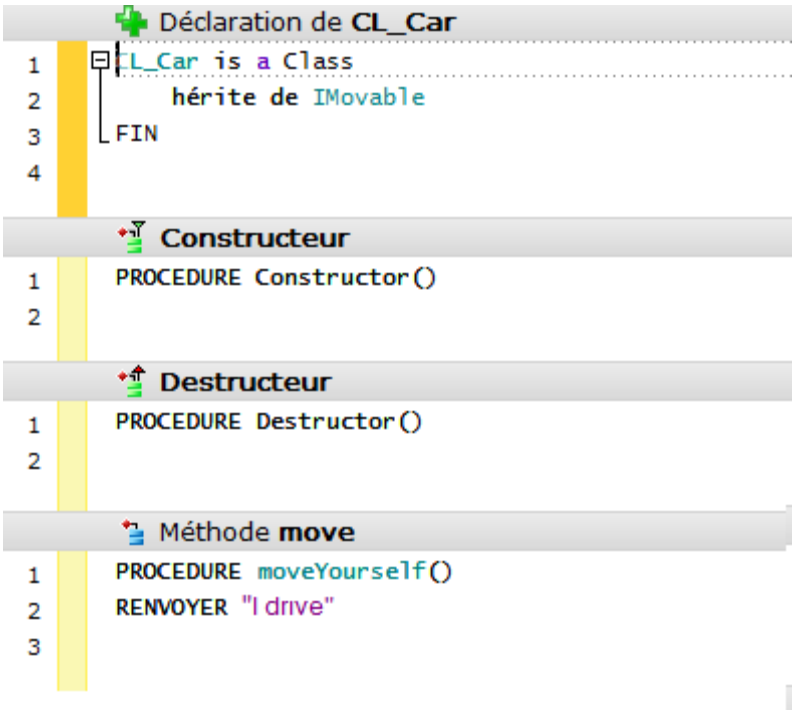
- only not implemented methods as an interface.
- implemented or not implemented methods, as an abstract class.

Create the IMovable class:

- The constructor should be PRIVATE to prohibit any instantiation, but Windev does not accept it. So we can make it PROTECTED so that the sub classes, and only it, could instantiate IMovable objects. It is a lesser evil.
- The moveYourself() method must be virtual to be redefined in sub classes. It must also define the return value, which explains that it returns an empty string on line 3.



We just have to create the sub classes:



Déclaration de **CL_Duck**

```
1  CL_Duck is a Class
2      hérite de IMovable
3  FIN
4
```

Constructeur

```
1  PROCEDURE Constructor()
2
```

Destructeur

```
1  PROCEDURE Destructor()
2
```

Méthode **move**

```
1  PROCEDURE  moveYourself()
2    RENVOYER "I fly."
3
```

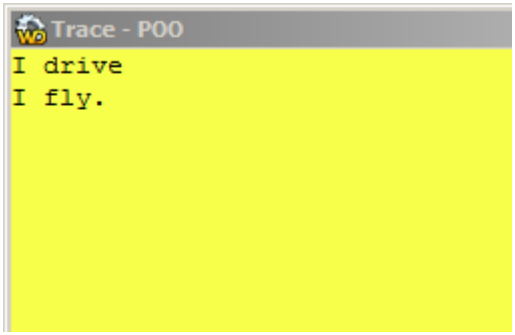
Now we can use our two classes. We instantiate an object of each class and we call the moveYourself() method:

```
pclMyCar is IMovable dynamic = new CL_Car
pclMyDuck is IMovable dynamic = new CL_Duck

Trace(pclMyCar.moveYourself())
Trace(pclMyDuck.moveYourself())
```

References are type IMovable, and objects are from IMovable subclasses.

And here is the spectacular result:



This approach allows to simulate something that can play the role of abstract class / interface.

There is still a drawback: IMovable constructor is not **private** but simply **protected,** the sub classes might theoretically instantiate IMovable objects. The responsibility of do not instantiate is based exclusively on the developer.

Note that version 19 of Windev would support real abstract classes. This is to be confirmed.

2 Static and dynamic instantiation

Suppose we have a CL_Boat class.

2.1 Static instantiation

Features:

- The object is instantiated immediately.
- The name given to the object is treated as a variable by the language.

Syntax:

< ObjectName > is [object] < class name > ([< parameters >])

Therefore multiple syntaxes are allowed:

- myBlueBoat is object CL_Boat
- myBlueBoat is CL_ Boat

For do not instantiate the object immediately, it must be set to NULL:

myBlueBoat is a CL_ Boat = NULL

In this case, the assignment of an object to the static reference, is done with the operator "<-".

- Either by instantiating an object at the same time:

  myBlueBoat <- new CL_Boat

- Either by setting an already existing object:

  B1 is a CL_Boat

  myBlueBoat <-B1

Windev official documentation indicates that static instantiation does not support polymorphism. However I have seen that the parametric polymorphism as well as inclusion polymorphism work with static instantiations.

Here is the description of my tests:

Parametric polymorphism:

- Create a class with an accelerate() method that returns a string.
- In the same class, override the accelerate() method (speed is integer). It also returns a string.
- There are therefore two signatures: one takes no parameters, the other takes an integer.
- In the code of a window, instantiate statically an object of the class containing the accelerate() method, and then call the two methods: parametric polymorphism works with objects instantiated statically.

Inclusion polymorphism:

- Create a A class with a start() method without parameters and returning a string.
- Create a B class derives from class A, redefining the start() method and returning a string but different from the A class method returned string.
- Statically instantiating an object from A class and an object from B class.
- Invoke the start() method on the two objects: each returns its string. Inclusion polymorphism works with static objects.

2.2 Dynamic instantiation

Dynamic instantiation is done in two steps:

- Creation of the reference using the **dynamic** keyword.
- Instantiating the object and setting it to the reference.

A common technique is to declare the references in the declarative part of a class, and instantiate objects in the constructor, or even in one of the methods.

Syntax for reference creation:

```
<Object Name> is [objet] <Class Name> dynamic
```

Example:

```
B1 is CL_Boat dynamic
```

Syntax for instantiating the object:

```
<Object Name> = new < Class Name > ([<Paramètres>])
```

Example:

```
B1 = new  CL_Boat
```

It is of course possible to perform two steps (reference creation and object instantiating) in a single line of code:

```
B1 is CL_Boat dynamic = new CL_Boat
```

3 Reference taking and object copy

Taking reference, as its name implies, allows to create a reference and associate it with an existing object.

As for object copy, it creates a clone of an object and associates it with a reference.

Depending on static or dynamic objects, three operators are available to deal with all situations:

Operator "="	Object	Dynamic object
Object	Copy	Copy
Dynamic object	Take reference	Take reference

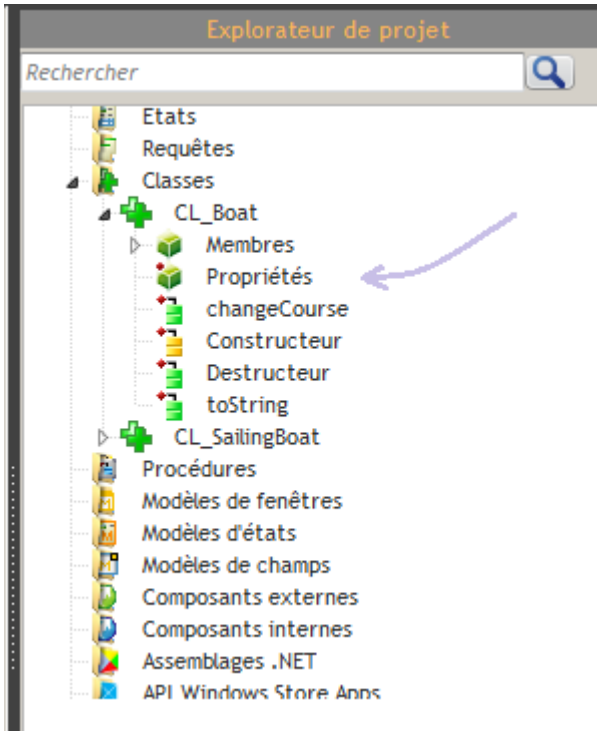
Operator <=	Object	Dynamic object
Object	Copy	Copy
Dynamic object	Copy	Copy

Operator <-	Object	Dynamic object
Object	Take reference	Take reference
Dynamic object	Take reference	Take reference

4 Properties (accessors)

In OOP a property is a combination of an attribute with two methods for accessing this attribute. One to read, the other to write. The 3 elements in the same class.

WinDev supports properties:



To create a property:

- In the project tree, select the relevant class and right click on «properties».
- Enter a name for the property.
- Windev generates two methods, unimplemented, bearing the name of the property:
  - One with no parameter: it will allow to read the attribute.
  - The other takes a parameter: it will allow to change the attribute value.

The relevant attribute must be created manually.

You have to implement the two methods.

It is of course possible to get the same result without using properties:

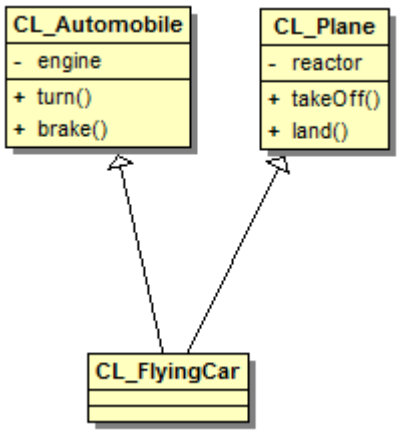
- Create the relevant attribute.
- Create manually two methods, one that returns the value of the attribute, the other takes a parameter and changes the attribute value.

5 Multiple inheritance

With multiple inheritance, a class can directly inherit from multiple classes. This feature tends to disappear in recent object-oriented languages, as source of confusion and difficulty of maintenance.

Languages that do not support multiple inheritance, propose instead a mechanism based on **the single-inheritance**, associated with the technique of the **interfaces** (see relevant chapter).

WinDev supports multiple inheritance. Suppose we have to implement the class diagram below:



Here's the implementation of the classes:

Déclaration de **CL_Automobile**

```
CL_Automobile is a Class
    m_engine is a string
FIN
```

Constructeur

```
PROCEDURE Constructor()
m_engine = "Ford"
```

Destructeur

```
PROCEDURE Destructor()
```

Méthode **turn**

```
PROCEDURE turn()
RENVOYER "I turn."
```

Méthode **brake**

```
PROCEDURE brake()
RENVOYER "I brake"
```

Méthode **getEngine**

```
PROCEDURE getEngine()
RENVOYER m_engine
```

Déclaration de **CL_Plane**

```
CL_Plane is a Class
    m_reactor is a string
FIN
```

Constructeur

```
PROCEDURE Constructor()
  m_reactor = "Rolls Royce"
```

Destructeur

```
PROCEDURE Destructor()
```

Méthode **takeOff**

```
PROCEDURE takeOff()
  RENVOYER "I take off."
```

Méthode **land**

```
PROCEDURE land()
  RENVOYER "I land."
```

Déclaration de **CL_FlyingCar**

```
CL_FlyingCar est une Classe
    hérite de CL_Automobile
    hérite de CL_Plane
FIN
```

Constructeur

```
PROCEDURE Constructor()
```

Destructeur

```
PROCEDURE Destructor()
```

Here is a piece of code using our classes:

```
Procédure locale héritage_multi
1  PROCEDURE héritage_multi()
2
3    clMyFlyingCar est un CL_FlyingCar
4
5    Trace(clMyFlyingCar.turn())
6    Trace(clMyFlyingCar.brake())
7    Trace(clMyFlyingCar.takeOff())
8    Trace(clMyFlyingCar.land())
9
```

And here is the stunning result:

```
Trace - POO
I turn.
I brake
I take off.
I land.
```

Note that in design patterns world, multiple inheritance is banned.

6 Class methods and class attributes

The **global** keyword is used to define class attributes and class methods. That mean that this attribute or this method will exist only in a single place in the class and will not be represented in objects instantiated from this class.

This is of interest when we want to factorize a resource (method or attribute) so that it is common to all objects of the class.

Go to auction: we create a class to handle buyers bidding. But we hope that the amount of auctions is represented only once and is common to all buyers. We just have to declare the amount of auctions **global**.

Here's the class:

Déclaration de **CL_Buyer** Si Erreur

```
1  CL_Buyer is a Class
2      GLOBAL
3          m_currentAuction is entier
4  FIN
5
```

Constructeur

```
1  PROCEDURE Constructor()
2
```

Destructeur

```
1  PROCEDURE Destructor()
2
```

Méthode **bid**

```
1  PROCEDURE bid(nAmount est un entier)
2    m_currentAuction += nAmount  // we increment.
3    RENVOYER getAuction()        // and we return the new auction amount.
4
```

Méthode **getAuction**

```
1  PROCEDURE getAuction()
2    RENVOYER m_currentAuction
3
```

Here is a piece of code that uses our CL_Buyer class:

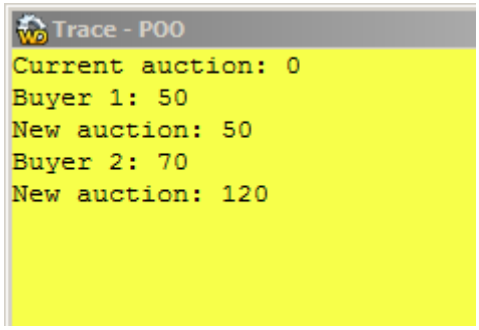
```
clBuyer1 is CL_Buyer
clBuyer2 is CL_Buyer

Trace("Current auction: " + clBuyer1.getAuction())

Trace("Buyer 1: 50")
Trace("New auction: " + clBuyer1.bid(50))

Trace("Buyer 2: 70")
Trace("New auction: " + clBuyer2.bid(70))
```

Here is the result for execution: the amount of auctions is well shared by all buyers.



In this example we used a **global attribute** . You can do the same with a method: just declare it **global**.

7 Miscalenous

7.1 Call a parent class method

```
Ancestor:method_name()
```

The **ancestor** keyword refers to the parent class.

It is also possible to reference a class ancestor by its name:

```
class_name:method_name()
```

7.2 Determine whether or not a reference point to an object

```
IF <my_reference> = NULL ...
IF <my_reference> <> NULL ...
```

7.3 The "This" reference

The equivalent of the 'this' of Java is 'object' with Windev: in this example it instantiates a CL_Boat object by passing in the reference parameter of the object running this line of code.

MyBoat is a CL_Boat = new CL_Boat (object)

7.4 Declare a structure in a class

The structure must be declared at the very beginning of the class before the line:

```
< MyClass > is a class
```

End of document