

# IMPLEMENTATION

---

## Chapter 8

**Software Engineering**

Computer Science School

DSIC – UPV




# Goals

- Discuss aspects related with the implementation of OO applications
- Discuss the foundations of software testing
- Describe tools for testing automation

# Contents (Block I)

- Polymorphism in programming languages
- Dynamic and Static Binding
- Constructors Implementation
- Software Reuse

# References

-  Sommerville, I. Ingeniería del Software. (8ª ed.). Addison-Wesley, 2008
-  Presman, R.S., Ingeniería del Software: un enfoque práctico (6ª ed.), McGraw-Hill, 2005
-  Cardelli, Luca and Wegner, peter. On Understanding Types, Data Abstraction, and Polymorphism *Computing Surveys*, Vol 17 n. 4, pp 471-522, December 1985  
(<http://lucacardelli.name/Papers/OnUnderstanding.A4.pdf>)

---

# Polymorphism in programming languages

# Polymorphism

- A characteristic of an entity that lets it adopt different forms:
  - Polymorphic Variable: it may contain values of different types
  - Polymorphic Function: a function acting on polymorphic variables and it may return a polymorphic result

# Polymorphism in Programming Languages

Ad-Hoc	Universal
Overloading	<b>Inclusive/Inheritance driven</b>
Cohercion	Parametric (Genericity)

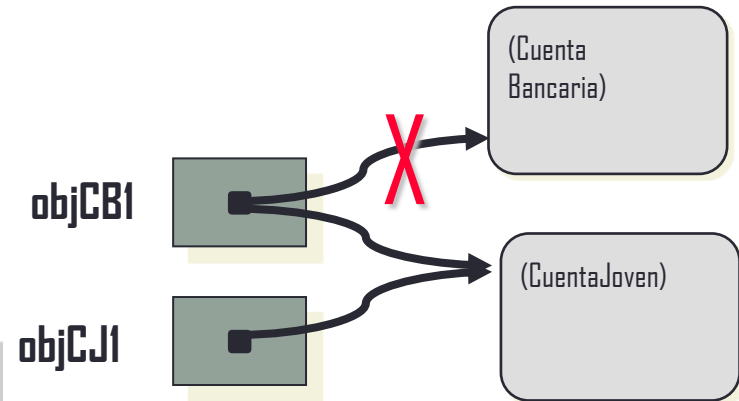
# Inheritance Driven Polymorphism

Variables of a given type may refer  
To instances of descendant classes

```
class CBancaria
{
    ...
}
class CJoven extends CBancaria
{
    ...
}
```

```
. . .
CBancaria objCB1 = new CBancaria();
CJoven objCJ1 = new CJoven();
```

```
objCB1=objCJ1 ;
```





# Inheritance Driven Polymorphism

```
class CBancaria
{
    ...
    public void f() {...}
    ...
}
class CJoven extends CBancaria
{
    ...
    public void g() {...}
}
```

Objects of descendant classes are seen as objects of the parent class

```
. . .
CBancaria objCB1 = new CBancaria();
CJoven objCJ1 = new CJoven();

objCB1=objCJ1;
objCB1.f(); 😊
objCB1.g(); 😞 //compilation error
. . .
```

# Inheritance Driven Polymorphism

- Advantages:
  - More expressive power (meaningful “is-a” relationship)
  - Heterogeneous collections can be created
- OO languages implement inheritance driven polymorphism
- However... somethingelse is needed

---

# Dynamic and Static Binding

# Definitions

- ❑ **Identifier or variable:** name used by a coder to denote entities that must be manipulated
- ❑ **Value:** real content of the computer's memory associated to a variable

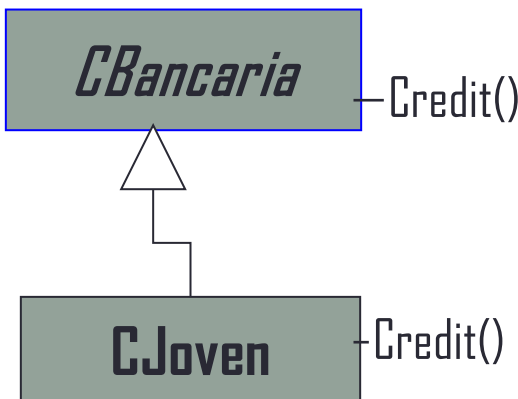
# Static and Dynamic Type

In typed languages:

- ❑ **Static type assignment:** types are associated with variables or identifiers by means of explicit declarations
  - ❑ In languages with static type assignment the name (variable) of an object has both a **static** and a **dynamic type**.
    - ❑ The static type is determined at compilation time by inspecting the declaration of the variable.
    - ❑ The dynamic type may change at run time. It is determined by the type of the value referenced by the variable at a given time.
- ❑ **Dynamic type assignment:** types are bound to values

# Dynamic/Static Binding

**Binding:** association between a message passing expression and the associated code execution at the receiver



```
CBancaria objCB1;
CJoven objCJ1;
objCB1 = new CBancaria();
objCJ1 = new CJoven();
objCB1 = objCJ1;
objCB1.Credit();
...
```

*What method is executed in response to objCB1.Credit()?*

We need to know the type of binding for this method

# Dynamic/Static Binding

- The result will be different depending on the binding:
  - ***Static Binding.*** The executed method is determined based on the static type of the variable `objCB1`
    - `Credit()` from `Cbancaria` would be executed
  - ***Dynamic Binding.*** The executed method is determined based on the dynamic type of variable `objCB1`
    - `Credit()` from `Cjoven` would be executed

# Binding in main OO languages

- Java, SmallTalk, Eiffel, PHP, Perl: every non-static method has dynamic binding
- C++, C#, Object Pascal...: binding user-defined
  - By default: static
  - Dynamic binding: "virtual/override"



# Example of Dynamic Binding in C#

```
class Token
{
    ...
    public int LineNumber( )
    { ...
    }
    public virtual string Name( ) { ... }
}

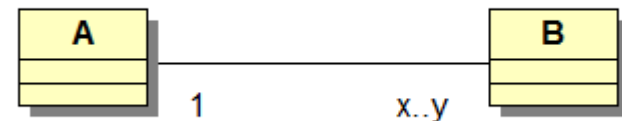
class CommentToken: Token
{
    ...
    public override string Name( ) { ... }
}
```

---

# Constructors Implementation

# Considerations about constructors (1/2)

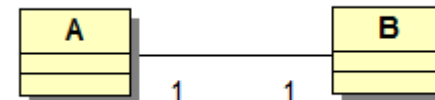
- Initializing an object results in giving values not only to attributes but also to links with objects of other classes.
- The minimum cardinality of associations/aggregations determines how the initialization is done.



x	y	Declaration	Constructor	Initialization
0	1	B the_B	A(...)	null
1	1		A(..., B the_B, ...)	this.the_B=the_B
0	*	ArrayList<B>the_Bs	A(...)	the_Bs=new ArrayList<B>
1	*		A(..., B the_B, ...)	the_Bs=new ArrayList<B>; the_Bs.add(the_B)

# Considerations about constructors (2/2)

- Case 1/1
  - When both sides have a minimum cardinality of 1, a circular dependency is created that cannot be resolved in one step.
  - An initialization must be performed in several steps



```
Class A {
String a1;
B the_B;
Public A(String a)
{
a1=a;
}
...
}
```

```
Class B {
String b1;
A the_A;
Public B(String b, A the_A)
{
A1=a;
this.the_A=the_A;
} ...
}
```

```
...
//inicialization;
A an_A=new A("an A");
B a_B = new B("a B", an_A);
an_A.setThe_B(a_B);
...
```

---

## Code Reuse

# Code Reuse

- Sometimes several relevant code segments appear in many classes. To reduce duplicated code two reuse strategies may be used
  - Inheritance: the new component inherits all the behavior of the existing one and it adds new behavior ("the new one *is-an* old one")
  - Composition: the new component is based in another existing one but the former is really a different entity ("the new one *has -an* old one")

# Code Reuse: Example

Let us assume we have the following class to implement collections on integer numbers...

```
class List_integers
{
    public void insert(int i){...}
    public boolean included(int i) {...}
    public boolean remove(int i){...}
    public int first_element{...}
    . . .
    //Implementation
}
```

... and we want to implement a set of integers by reusing as much as possible of the List class

# Code Reuse (Option 1): Inheritance

- The set is defined as a subclass of the list
- The insert method is redefined:

```
class Set_integers extends List_integers
{
    //redefinition to remove repeated elements:
    public void insert(int i)

        {
            ...
        }
}
```



## Code Reuse (Option 2): Composition

- The set is constructed over a list
- The methods of the set are implemented in terms of those of the list (delegation)

```
class Set_integers
{
    private List_integers data;

    public void insert(int i){ if (!data.included(i))
                                data.insert(i); }
    public boolean included(int i){return data.included(i);}
    public boolean remove(int i){return data.remove(i);}

    // first_element() method is hidden
    . . .
}
```