

# **Tema 1 – Sessió 4**

## **Programació orientada a objecte**

---

# Encapsulament de dades

## Exemple

```
import complex
c = complex.NumeroComplex()

c.real = float(input())
c.img = float(input())

conj = c.conjugat()

print (conj.real, conj.img) ???
```

```
class NumeroComplex:
    def __init__(self, pReal, pImg):
        self.real = pReal
        self.img = pImg

    def conjugat(self):
    def __add__(self, c):
    def __sub__(self, c):
    def __mul__(self, c):
```

Què passa si volem canviar  
la representació de les dades  
del número complex?

```
class NumeroComplex:
    def __init__(self, pReal, pImg):
        self.coord = [pReal, pImg]

    def conjugat(self):
    def __add__(self, c):
    def __sub__(self, c):
    def __mul__(self, c):
```

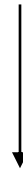
# Encapsulament de dades

## Exemple

```
c.real = float(input())  
c.img = float(input())  
  
conj = c.conjugat()  
print (conj.real, conj.img) ???
```

```
class NumeroComplex:  
    def __init__(self, pReal, pImg):  
        self.coord = [pReal, pImg]  
  
    def conjugat(self):  
    def __add__(self, c):  
    def __sub__(self, c):  
    def __mul__(self, c):
```

- Els clients (o usuaris) d'una classe no han de tenir accés directe a la representació interna (propietats o atributs) de la classe.
- Qualsevol canvi o consulta a l'estat intern de la classe s'ha de fer utilitzant els mètodes (o accions) definides sobre la classe.



Distingir entre part privada i part pública de la classe

# Encapsulament de dades: part privada i pública

La majoria de llenguatges de programació orientats a objecte permeten distingir entre **membres públics** i **privats** en una classe

```
conjugat  
__add__  
__sub__  
__mul__
```

```
real  
img
```

## Part pública

- Accessible des de fora de la classe (mètodes d'altres classes i funcions globals) i també pels mètodes de la pròpia classe
- Normalment conté només mètodes
- Defineix la interfície de la classe: conjunt d'accions o operacions que podem fer amb els objectes de la classe



```
c.real = float(input())  
c.img = float(input())
```



```
conj = c.conjugat()
```



```
print(conj.real, conj.img)
```

## Part privada

- Només és accessible dins dels mètodes de la pròpia classe
- No és visible ni accessible fora de la classe (mètodes d'altres classes i funcions globals)
- Normalment conté tots els atributs i també els mètodes que només es criden des d'altres mètodes de la classe

# Encapsulament de dades: part privada i pública

... però **Python no té cap mecanisme** per **distingir** explícitament entre membres **públics** i **privats** en una classe.

## ALTERNATIVA

```
class NumeroComplex:
    def __init__(self, pReal = 0, pImg = 0):
        self._real = pReal
        self._img = pImg

    def conjugat(self):
        return NumeroComplex(self._real, -self._img)
```

- Declarar els atributs amb un caràcter de subratllat '\_' al principi del nom
- **Convenció implícita:** els atributs declarats així no s'haurien d'utilitzar fora de l'àmbit de la classe. **No és una obligació, només una recomanació.**

Provem...

```
c = complex.NumeroComplex()
c._real = float(input())
c._img = float(input())
conj = c.conjugat()
print (conj._real, conj._img)
```

Funcionament correcte ...

... però un bon programador en Python no utilitzarà mai directament un atribut precedit per \_

... Com podem modificar o recuperar el valor dels atributs precedits per \_?

# Encapsulament de dades: getters i setters

```
class NumeroComplex:
    def __init__(self, pReal = 0, pImg = 0):
        self._real = pReal
        self._img = pImg
    def getReal(self):
        return self._real
    def getImg(self):
        return self._img
    def setReal(self, real):
        self._real = real
    def setImg(self, img):
        self._img = img
```

getters: mètodes per recuperar el valor dels atributs

setters: mètodes per modificar el valor dels atributs

Apart de recuperar/modificar el valor dels atributs poden incloure altres accions per controlar errors, validar el rang dels valors, etc.

```
c = complex.NumeroComplex()
c.setReal(float(input()))
c.setImg(float(input()))
conj = c.conjugat()
print (conj.getReal(), conj.getImg())
```

Accés públic a les dades a partir de crides als getters i setters


# Exercici

- Feu tots els canvis necessaris a la classe NumeroComplex i, si cal al script de prova, per poder canviar la representació interna del número complex tal com s'indica

```
c = complex.NumeroComplex()
c.setReal(float(input()))
c.setImg(float(input()))
conj = c.conjugat()
print (conj.getReal(), conj.getImg())
```

```
class NumeroComplex:
    def __init__(self, pReal, pImg):
        self._real = pReal
        self._img = pImg

    def conjugat(self):
    def __add__(self, c):
    def __sub__(self, c):
    def __mul__(self, c):
```



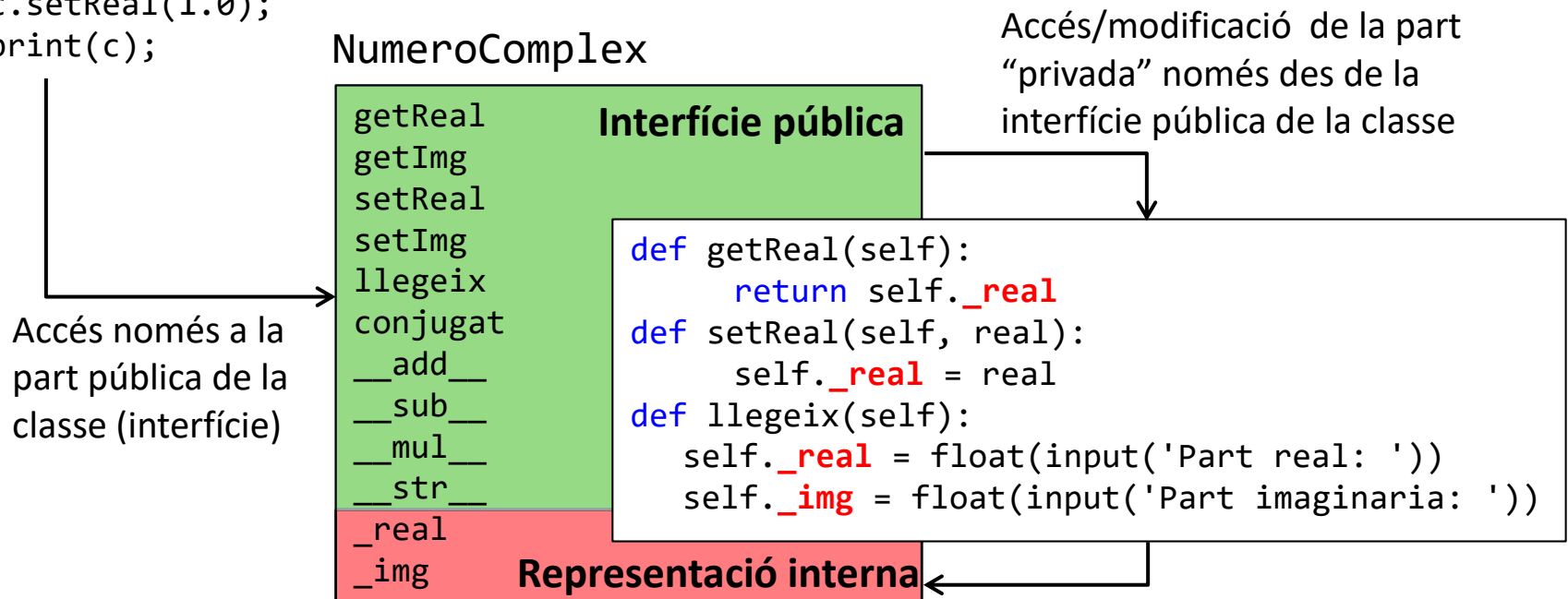
```
class NumeroComplex:
    def __init__(self, pReal, pImg):
        self._coord = [pReal, pImg]

    def conjugat(self):
    def __add__(self, c):
    def __sub__(self, c):
    def __mul__(self, c):
```

# Abstracció

- Definició d'una interfície pública amb les accions i/o operacions (mètodes) que es poden utilitzar per interactuar amb els objectes de la classe.
- La interfície pública amaga (abstrau) la representació interna de la classe als programes/classes (clients) que la utilitzen.
- Els clients no necessiten conèixer la representació interna de la classe per poder-la utilitzar.

```
NumeroComplex c;  
c.llegeix();  
c.setReal(1.0);  
print(c);
```





# Encapsulament de dades: property

```
class NumeroComplex:
    def __init__(self, pReal = 0, pImg = 0):
        self._real = pReal
        self._img = pImg
    def getReal(self):
        return self._real
    def getImg(self):
        return self._img
    def setReal(self, real):
        self._real = real
    def setImg(self, img):
        self._img = img
```

property: crea un nou atribut definit com a propietat

- Agrupa un getter i setter per accedir i modificar un dels atributs de la classe
- S'utilitza igual que si fos un atribut
- Cridarà al getter o setter corresponent segons s'estigui consultant o modificant el valor

```
real = property(getter, setter)
img = property(getImg, setImg)
```

```
c = complex.NumeroComplex()
c.real = (float(input()))
c.img = (float(input()))
conj = c.conjugat()
print (conj.real, conj.img)
```

Crida al setter vinculat a la propietat  
`c.setReal(float(input()))`

Crida al getter vinculat a la propietat  
`print (cont.getReal(), ...)`

# Encapsulament de dades: property i decorators

```
class NumeroComplex:
    def __init__(self, pReal = 0, pImg = 0):
        self._real = pReal
        self._img = pImg
```

**@property**

```
def real(self):
    return self._real
```

Definició del nom de la propietat (real), i de la funció que fa de getter

**@real.setter**

```
def real(self, real):
    self._real = real
```

Definició de la funció setter associada a la propietat anomenada real

**@property**

```
def img(self):
    return self._img
```

**@img.setter**

```
def img(self, img):
    self._img = img
```

Utilització de les propietats als mètodes de la classe


```
def conjugat(self):
    return NumeroComplex(self.real, -self.img)
```

# Exercici

- Torneu a fer canvis necessaris a la classe `NumeroComplex` per poder canviar la representació interna del número complex tal com s'indica, utilitzant propietats i decoradors

```
class NumeroComplex:
    def __init__(self, pReal, pImg):
        self._real = pReal
        self._img = pImg

    def conjugat(self):
    def __add__(self, c):
    def __sub__(self, c):
    def __mul__(self, c):
```



```
class NumeroComplex:
    def __init__(self, pReal, pImg):
        self._coord = [pReal, pImg]

    def conjugat(self):
    def __add__(self, c):
    def __sub__(self, c):
    def __mul__(self, c):
```

# Encapsulament de dades: property

```
class NumeroComplex:
    def __init__(self, pReal = 0, pImg = 0):
        self._real = pReal
        self._img = pImg
    def getReal(self):
        return self._real
    def getImg(self):
        return self._img
    def setReal(self, real):
        self._real = real
    def setImg(self, img):
        self._img = img
```

property: crea un nou atribut definit com a propietat

- Agrupa un getter i setter per accedir i modificar un dels atributs de la classe
- S'utilitza igual que si fos un atribut
- Cridarà al getter o setter corresponent segons s'estigui consultant o modificant el valor

```
real = property(getter, setter)
img = property(getImg, setImg)
```

```
c = complex.NumeroComplex()
c.real = (float(input()))
c.img = (float(input()))
conj = c.conjugat()
print (conj.real, conj.img)
```

Crida al setter vinculat a la propietat  
`c.setReal(float(input()))`

Crida al getter vinculat a la propietat  
`print (cont.getReal(), ...)`

# Encapsulament de dades: property i decorators

```
class NumeroComplex:
    def __init__(self, pReal = 0, pImg = 0):
        self._real = pReal
        self._img = pImg
```

```
@property
def real(self):
    return self._real
```

Definició del nom de la propietat (real), i de la funció que fa de getter

```
@real.setter
def real(self, real):
    self._real = real
```

Definició de la funció setter associada a la propietat anomenada real

```
@property
def img(self):
    return self._img
```

```
@img.setter
def img(self, img):
    self._img = img
```

Utilització de les propietats als mètodes de la classe


```
def conjugat(self):
    return NumeroComplex(self.real, -self.img)
```

# Exercici

- Torneu a fer canvis necessaris a la classe `NumeroComplex` per poder canviar la representació interna del número complex tal com s'indica, utilitzant propietats i decoradors

```
class NumeroComplex:
    def __init__(self, pReal, pImg):
        self._real = pReal
        self._img = pImg

    def conjugat(self):
    def __add__(self, c):
    def __sub__(self, c):
    def __mul__(self, c):
```



```
class NumeroComplex:
    def __init__(self, pReal, pImg):
        self._coord = [pReal, pImg]

    def conjugat(self):
    def __add__(self, c):
    def __sub__(self, c):
    def __mul__(self, c):
```

# Exercici

Imaginem que volem fer una aplicació de missatgeria de l'estil de WhatsApp. Volem crear un conjunt de classes que ens serveixin per gestionar un grup de conversa.

De moment començarem creant una **classe Missatge** que serveixi per guardar les dades d'un dels missatges que s'envien al grup de conversa. Les dades que haurem de guardar a cada missatge són el **nom de l'emissor** del missatge, el **text del missatge** i la **data d'enviament**.

A més a més d'aquestes dades, volem que la classe tingui un **mètode** que es digui **llegeix** a la seva interfície pública que permeti llegir les dades d'un missatge per teclat.

La classe també ha de tenir les **propietats** necessàries per poder modificar i recuperar les dades internes del missatges (nom de l'emissor, text del missatge i data d'enviament). En el cas de la **data**, sempre que la modifiquem haurem de comprovar que segueix el **format "dd/mm/yyyy"**. Si no segueix aquest format aixecarem una excepció.

## Exercici

1. Feu la **implementació** completa de la **classe** amb els **atributs i mètodes**, distingint entre el que ha d'estar declarat com a **privat** i el que ha de formar part de la **interfície pública** de la classe. Declareu  **propietats** per donar accés a la part privada de la classe.
2. **Implementeu** una funció **mostraMissatgesEmissor** que rebi com a paràmetres el nom d'una persona i una llista amb objectes de la classe **Missatge** i mostri per pantalla tots els missatges (text i data) que ha enviat aquesta persona.
3. Implementeu un **programa principal** que declari una llista per guardar objectes de la classe **Missatge**, llegeixi per teclat un conjunt de missatges i els guardi a la llista, i després llegeixi el nom d'un emissor i cridi a la funció **mostraMissatgesEmissor** per mostrar tots els missatges d'aquesta persona.



# Exercici

A partir de la definició d'aquesta classe `Point` volem crear una nova classe `Poligon` que permeti guardar tots els vèrtexs d'un polígon i també les coordenades de la cantonada superior esquerra i de la cantonada inferior dreta del rectangle mínim que engloba al polígon

```
class Point:
    def __init__(self, x = 0, y = 0):
        self._x = x
        self._y = y

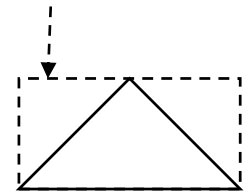
    @property
    def x(self):
        return self._x
    @x.setter
    def x(self, valor):
        self._x = valor

    def __sub__(self, p2):
        return math.sqrt((self.x - p2.x)**2 + (self.y - p2.y)**2)

    def __str__(self):
        return "(" + str(self.x) + ", " + str(self.y) + ")"
```

```
@property
def y(self):
    return self._y
@y.setter
def y(self, valor):
    self._y = valor
```

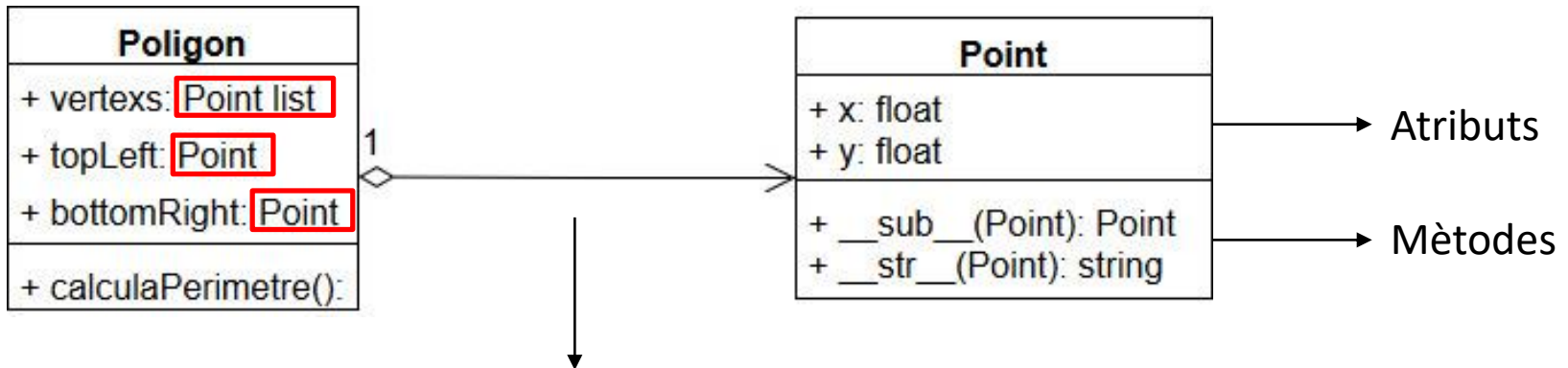
Rectangle mínim



# Composició de classes

- Utilització d'una classe ja existent (Point) com a atribut dins d'una altra classe (Poligon): *un polígon està compost per punts*

**Diagrama UML:** Representació de l'estructura de les classes



**Relació de composició:**

- Un polígon està compost per N punts

# Composició de classes

```
import punt
```

Importem el mòdul amb la definició de la classe Point

```
class Poligon:
```

```
    maxim = 1000
```

```
    def __init__(self):
```

```
        self._vertexs = []
```

```
        self._topLeft = punt.Point(Poligon.maxim, Poligon.maxim)
```

```
        self._bottomRight = punt.Point()
```

Inicialització dels objectes de la classe Point

```
    def afageixVertex(self, pt):
```

```
        self._vertexs.append(pt)
```

```
        if pt.x < self._topLeft.x:
```

```
            self._topLeft.x = pt.x
```

```
        else:
```

```
            if pt.x > self._bottomRight.x:
```

```
                self._bottomRight.x = pt.x
```

```
            if pt.y < self._topLeft.y:
```

```
                self._topLeft.y = pt.y
```

```
            else:
```

```
                if pt.y > self._bottomRight.y:
```

```
                    self._bottomRight.y = pt.y
```

Afegim objectes de la classe Point a la llista

- Utilització dels mètodes i propietats de la classe Point.
- Només hauríem d'utilitzar mètodes i propietats "*públics*", no els atributs "*privats*".

## Exercici

Completar la definició de la classe `Pòligon`:

- Afegir propietats per recuperar els punts de la cantonada superior esquerra i de la cantonada inferior dreta del rectangle mínim
- Afegir un mètode que retorni el perímetre del polígon

Fer un programa que llegeixi tots els vèrtexs d'un polígon i després calculi i mostri per pantalla el perímetre del polígon i mostri per pantalla els punts de la cantonada superior esquerra i de la cantonada inferior dreta del rectangle mínim.