

Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación



IIC2115 - Programación como Herramienta para la Ingeniería

Fundamentos de Programación Orientada a Objetos (OOP)

Profesora: Francesca Luchini
Prof. Coordinador: Hans Löbel

Partamos con un ejemplo conceptual

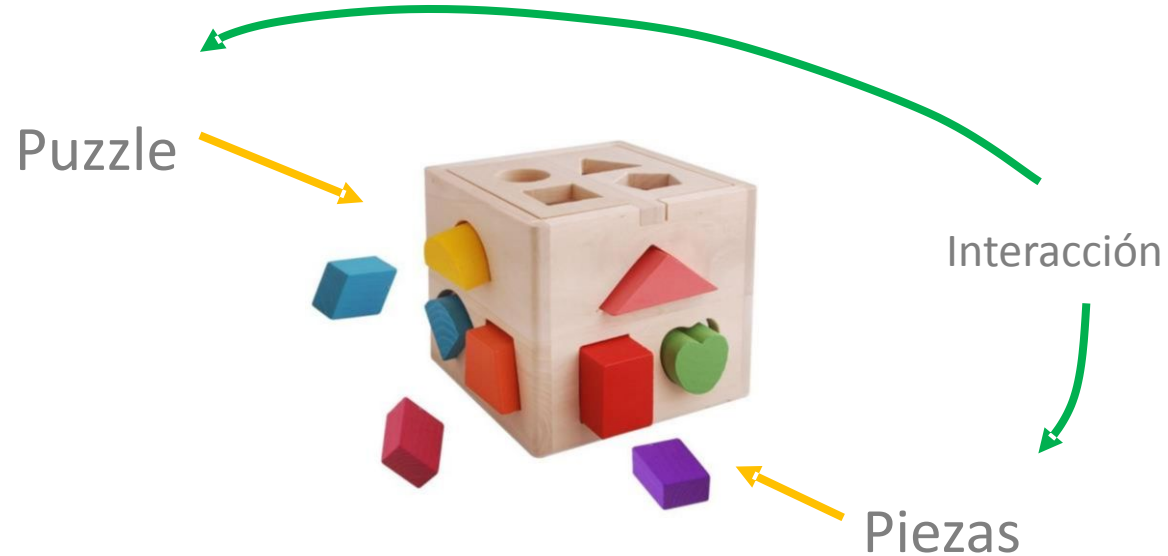
Cuando hablamos de un curso, ¿en qué estamos pensando?



Todas estas maneras de “modelarlo” representan distintas abstracciones del concepto curso, cada una más o menos adecuadas para distintas tareas.

Esto cambia un poco cuando tenemos elementos físicos

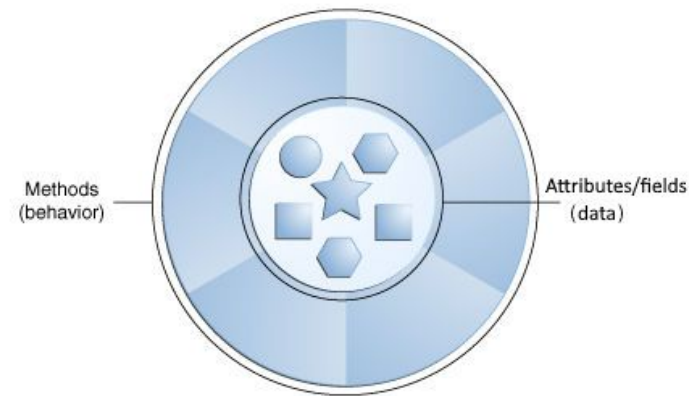
Los objetos físicos **ya están modelados**, por lo que traen definido su todo y sus partes constituyentes, pero además, una manera de **interactuar** con ellos



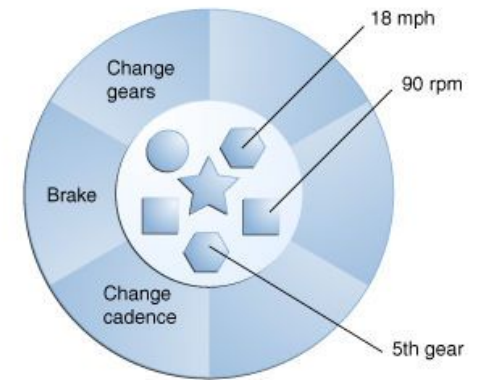
Objetos de software combinan ambas ideas

En el área de desarrollo de software, un objeto es una colección de **datos** que además tiene asociado **comportamientos**.

- Datos: **describen** el estado y/o composición de los objetos. Se les conoce como **atributos** o **campos** del objeto.
- Comportamientos: **representan acciones** que realiza el objeto, o realizan sobre él, que pueden generar cambios en su estado. Se les conoce como **métodos** del objeto.



(a) A software object



(b) Bicycle modelled as a software object

Ejemplo: datos y comportamiento

Clase: Auto	
Datos	Comportamiento
Marca	Calcular próxima mantención
Modelo	Calcular distancia a alguna dirección
Color	Pintar de otro color
Año	Realizar nueva mantención
Motor	
Kilometraje	
Ubicación actual	

¿Qué es entonces OOP?

La programación orientada a objetos (OOP) implica que los **programas** estarán orientados a **modelar** las funcionalidades a través de la **interacción entre objetos** por medio de sus **datos** y **comportamiento**.

Para definir un objeto, creamos una plantilla llamada **clase**

Cada objeto es una **instancia** de la clase Auto

Objeto 1



Objeto 2



Objeto 3



Clase **Auto**

```
1 class Departamento:
2     def __init__(self, _id, mts2, valor, num_dorms, num_banos):
3         self._id = _id
4         self.mts2 = mts2
5         self.valor = valor
6         self.num_dorms = num_dorms
7         self.num_banos = num_banos
8         self.vendido = False
9
10    def vender(self):
11        if not self.vendido:
12            self.vendido = True
13        else:
14            print("Departamento {} ya se vendió".format(self._id))
```



```
1 d1 = Departamento(_id=1, mts2=100, valor=5000, num_dorms=3, num_banos=2)
2 print(d1.vendido)
3 d1.vender()
4 print(d1.vendido)
5 d1.vender()
```

False

True

Departamento 1 ya se vendió

```
1 d2 = Departamento(_id=2, mts2=185, valor=4000, num_dorms=2, num_banos=1)
2 d3 = Departamento(_id=1, mts2=100, valor=5000, num_dorms=3, num_banos=2)
3 d3.vender()
4 d4 = d1
5
6 print(d1 == d2)
7 print(d1 == d3)
8 print(d1 == d4)
9
10 d4.vendido = False
11 print(d1.vendido == d4.vendido)
```

```
1 d2 = Departamento(_id=2, mts2=185, valor=4000, num_dorms=2, num_banos=1)
2 d3 = Departamento(_id=1, mts2=100, valor=5000, num_dorms=3, num_banos=2)
3 d3.vender()
4 d4 = d1
5
6 print(d1 == d2)
7 print(d1 == d3)
8 print(d1 == d4)
9
10 d4.vendido = False
11 print(d1.vendido == d4.vendido)
```

False

False

True

True

Un concepto fundamental es el de **interfaz** de un objeto

Existen atributos de los objetos que no necesitan ser visualizados ni accedidos por los otros objetos con que se interactúa.



Un concepto fundamental es el de **interfaz** de un objeto



Interface
Turn on Turn off Volume up Volume down Switch to next channel Switch to previous channel
Current channel Volume level

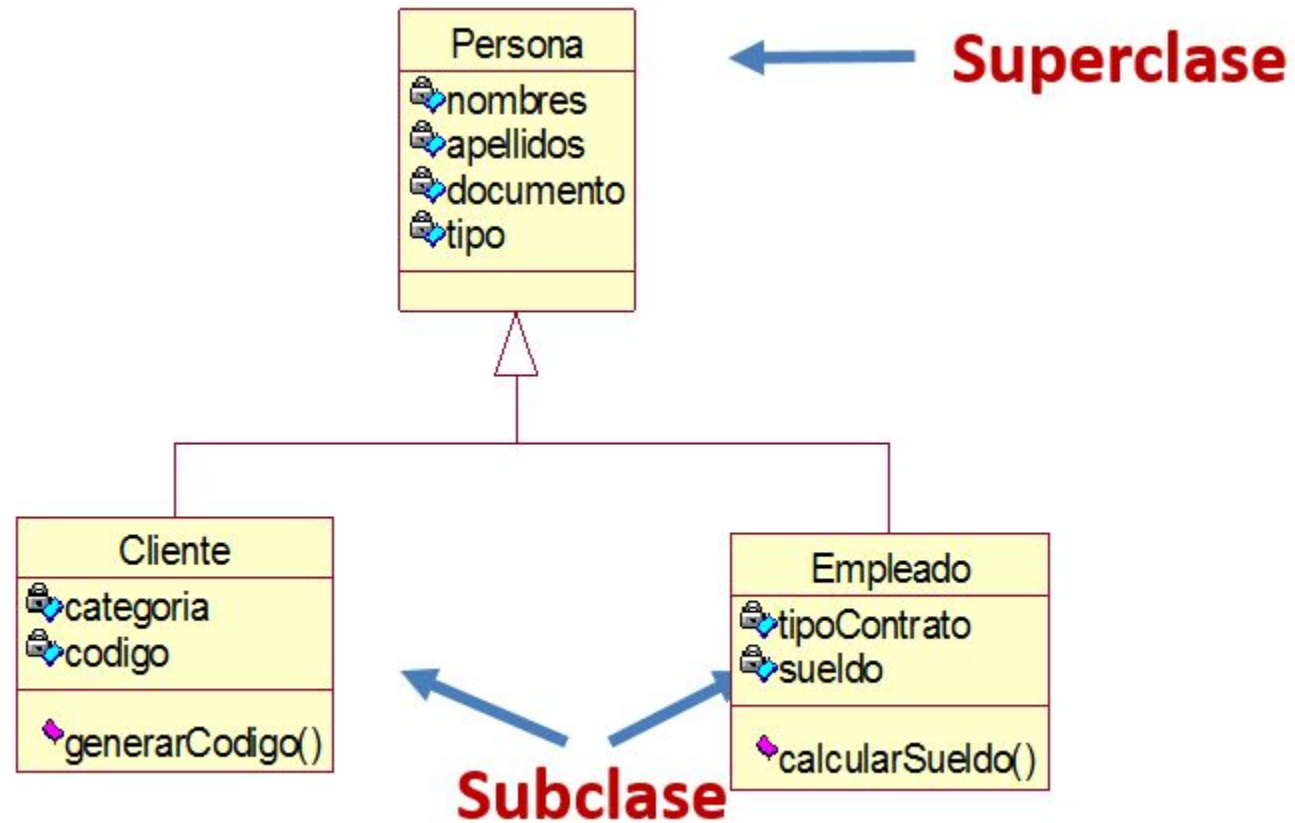
```
1 class Televisor:
2     ''' Clase que modela un televisor.
3     '''
4
5     def __init__(self, pulgadas, marca):
6         self.pulgadas = pulgadas
7         self.marca = marca
8         self.encendido = False
9         self.canal_actual = 0
10
11     def encender(self):
12         self.encendido = True
13
14     def apagar(self):
15         self.encendido = False
16
17     def cambiar_canal(self, nuevo_canal):
18         self._codificar_imagen()
19         self.canal_actual = nuevo_canal
20
21     def __codificar_imagen(self):
22         print("Estoy convirtiendo una señal eléctrica en la imagen que estás viendo.")
```


El nivel de detalle de la interfaz define nuestra **abstracción**



Vendedor
Nombre
Nº autos vendidos
Comisión asignada

Herencia nos permite modelar **clases** similares sin reescribir todo de nuevo




```
1 import numpy as np
2
3 class Variable:
4     def __init__(self, data):
5         self.data = np.array(data)
6
7     def representante(self):
8         pass
9
10 class Ingresos(Variable):
11     def representante(self):
12         return np.mean(self.data)
13
14 class Comuna(Variable):
15     def representante(self):
16         ind = np.argmax([np.sum(self.data == c) for c in self.data]) # el que mas se repite
17         return self.data[ind]
18
19 class Puesto(Variable):
20     categorias = {'Gerente': 1, 'SubGerente': 2, 'Analista': 3,
21                  'Alumno en Practica': 4} # class (or static) variable
22
23     def representante(self):
24         return self.data[np.argmin([Puesto.categorias[c] for c in self.data])]#la categoria mas alta acorde con el diccionario
```

Multiherencia es poco común, pero puede ser útil

```
1 class Investigador:
2     def __init__(self, area):
3         self.area = area
4
5 class Docente:
6     def __init__(self, departamento):
7         self.departamento = departamento
8
9 class Academico(Docente, Investigador):
10     def __init__(self, nombre, area_investigacion, departamento):
11         #esto no es del todo correcto, coming soon...
12         Investigador.__init__(self, area_investigacion)
13         Docente.__init__(self, departamento)
14         self.nombre = nombre
15
16 p1 = Academico("Juan Perez", "Inteligencia de Máquina", "Ciencia De La Computación")
17 print(p1.nombre)
18 print(p1.area)
19 print(p1.departamento)
```

Juan Perez
Inteligencia de Máquina
Ciencia De La Computación

Clases abstractas permiten formalizar interfaz

```
1 from abc import ABCMeta, abstractmethod
2
3 class Base(metaclass=ABCMeta):
4     @abstractmethod
5     def func_1(self):
6         pass
7
8     @abstractmethod
9     def func_2(self):
10         pass
11
12 class SubClase(Base):
13     def func_1(self):
14         pass
15
16     # Nuevamente olvidamos declarar el método func_2
17
18 print('Es subclase: {}'.format(issubclass(SubClase, Base)))
19 print('Es instancia: {}'.format(isinstance(SubClase(), Base)))
```

Es subclase: True

```
-----
TypeError                                Traceback (most recent call last)
<ipython-input-3-657925633d3a> in <module>()
    17
    18 print('Es subclase: {}'.format(issubclass(SubClase, Base)))
--> 19 print('Es instancia: {}'.format(isinstance(SubClase(), Base)))
```

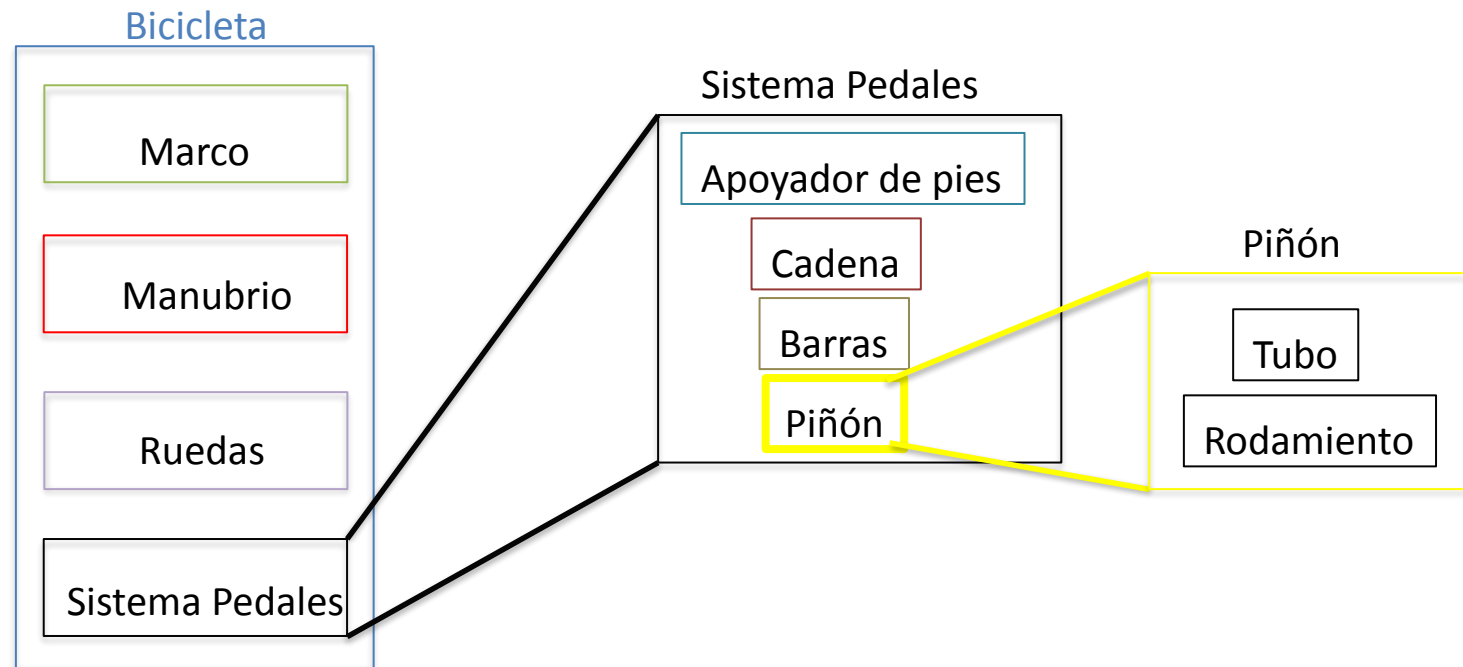
TypeError: Can't instantiate abstract class SubClase with abstract methods func_2

Clases abstractas permiten formalizar interfaz

```
1 from abc import ABCMeta, abstractmethod
2
3 class Base(metaclass=ABCMeta):
4     @abstractmethod
5     def func_1(self):
6         pass
7
8     @abstractmethod
9     def func_2(self):
10        pass
11
12 class SubClase(Base):
13
14     def func_1(self):
15         pass
16
17     def func_2(self):
18         pass
19
20     # Nuevamente olvidamos declarar el método func_2
21
22 c = SubClase()
23 print('Subclass: {}'.format(issubclass(SubClase, Base)))
24 print('Instance: {}'.format(isinstance(SubClase(), Base)))
```

Subclass: True
Instance: True

También es posible modelar **objetos** como atributos de otros objetos, mediante **agregación y composición**



#Ejemplo: Programa para manejar postits en un panel mural

```
import datetime
```

```
class PostIt:
```

```
    ''' Representa un post it, contiene un mensaje, guarda un conjunto de tags
        y responde si hay match con ciertos tags
        Contiene ademas un id
    '''
```

```
    last_id = 0 #variable estática para manejar el ultimo id generado
```

```
    def __init__(self, mensaje, tags = ''):
```

```
        self.mensaje = mensaje
```

```
        self.tags = tags
```

```
        self.creation_date = datetime.date.today()
```

```
        self._id = PostIt.last_id #variable de la clase para manejar el ultimo id generado
```

```
        PostIt.last_id += 1
```

```
    def match(self, keyword):
```

```
        ''' determina si el mensaje de la nota contiene el keyword o no'''
```

```
        return keyword in self.mensaje or keyword in self.tags
```

```
class Panel:
    ''' Representa un panel con un conjunto de postits (con memos) pegados
        cada hoja ademas de un memo tiene tags, asi podemos buscar hojas
        tambien podemos modificarlas
    '''

    def __init__(self):
        self.postit1 = PostIt('')
        self.postit2 = PostIt('')
        self.postit3 = PostIt('')
```

Agregación/composición vs herencia

- ¡NO TIENEN NADA QUE VER!
- Si bien ambos son mecanismo para modelar, estructuralmente difieren de manera fundamental.
- **Herencia** busca facilitar la **especialización** de las clases, sin requerir repetir código.
- **Agregación y composición** buscan aumentar el nivel de **abstracción** de las clases, al permitir tipos de datos complejos (otras clases) como atributos.

Actividad de Hoy

Sobre los repositorios:

- Crear repositorios si no lo han hecho
- Registrar cuenta de Github con su correo UC

Actividad de hoy (nota participación)

- Ir a Syllabus/Material Clase/Capítulo 1/Parte a/Ejercicios/C1a.pdf
- Avanzar actividad durante horario de taller
- Entrega en repositorio personal hasta las 22:00 hrs
- No olvidar **revisar Notebooks de ejemplo!**
 - Syllabus-2022-1/Material de clases/Capítulo 1/Parte a/Notebooks Ejemplo/

Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación



IIC2115 - Programación como Herramienta para la Ingeniería

Fundamentos de Programación Orientada a Objetos (OOP)

Profesora: Francesca Luchini
Prof. Coordinador: Hans Löbel