

INICIACIÓN PYTHON

Anexo I

Clases y Herencias

Curso 2022

Germán Alonso Lascurain

germanalonso@opendeusto.es

galonso@datavaluemanagement.es



CLASES Y HERENCIAS

- Algunas definiciones:

¿Qué es un Objeto?

Los objetos son la clave para entender la Programación Orientada a Objetos. Podemos entender los objetos en programación como los objetos de la vida cotidiana.

En Python, un objeto es cualquier instancia que tenga un tipo, un conjunto de procedimientos para interactuar con él y una representación interna de los datos.

Todo en Python es un objeto y tiene un tipo.

¿Qué es un Atributo?

Los atributos o propiedades de los objetos son las características que puede tener un objeto: Si el objeto fuera perro, los atributos podrían ser: tamaño, edad, color, raza,...

CLASES Y HERENCIAS

¿Qué es un Método?

Los métodos son la acción o función que realiza un objeto. Si nuestro objeto es Perro, los métodos pueden ser: caminar, ladrar, saltar, dormir, etc...

¿Qué es una Clase?

Con todos los conceptos anteriores explicados, se puede decir que una clase es una plantilla genérica de un objeto. La clase proporciona variables iniciales de estado (donde se guardan los atributos) e implementaciones de comportamiento (métodos).

Si tuviéramos una máquina para “producir” objetos perro, crearía un nuevo perro, con sus atributos (color, raza, ...) y sus métodos (lo que hace, ladrar, ...).

CLASES Y HERENCIAS

¿Qué es una Instancia?

Ya sabemos que una clase es una estructura general del objeto.

→ P.ej, podemos decir que la clase Mascota necesita tener un nombre y una especie, pero no nos va a decir cual es el nombre y cual es la especie, es aquí donde entran las instancias. Una instancia es una copia específica de la clase con todo su contenido.

a = 1234, a es una instancia de un entero

b = "hola" → b es una instancia de una cadena

c = [1,2,3,4,5] → c es una instancia de una lista.

- Por tanto las instancias son un tipo particular de objetos.

CLASES Y HERENCIAS

- Las clases en Python son una mezcla de las clases de C++ y Modula-3 y ofrecen las características estándar.
- Las clases de Python proveen las características normales de la programación orientada a objetos → Herencia de clases → Una clase derivada puede sobrescribir cualquier método de su/s clase/s base.
- Hemos visto que Python soporta diferentes tipos de datos (enteros, cadenas, listas, ...)

CLASES Y HERENCIAS

- **Programación orientada a objetos (OOP).**

- En Python todo son objetos y tienen un tipo.

- Los objetos son una abstracción de los datos que:

- Capturan su representación interna a través de los atributos de los datos.
 - Pej, las listas, necesitan un tipo de “representación interna” para que Python pueda manejar esos datos (cómo organizar los datos de esa lista). En el caso de los enteros, esa representación interna queda reducida a la mínima expresión.
- Tienen una interfaz para interactuar con ellos → Métodos.
Definen el comportamiento del objeto, pero ocultan su implementación.

CLASES Y HERENCIAS

- **Programación orientada a objetos (POO).**
 - Los objetos, pueden crear nuevas instancias de objetos
 - Los objetos, pueden destruir objetos.

CLASES Y HERENCIAS

- **Programación orientada a objetos (OOP).**

→ Ej. `L = [1,2,3,4]` es un tipo **lista** y una instancia de la clase lista.

➤ Representación interna: Conjunto enlazado de datos.



La representación interna no nos importa. No necesitamos conocerla. No debemos actuar directamente contra ella, debemos usar las interfaces provistas por el objeto, al efecto.

➤ Interactuar con listas.

`L[i]`, `L[i:j]`, `L.append()`, `L.extend()`...

CLASES Y HERENCIAS

- **Creación y uso de nuestros propios objetos con clases.**

- Necesitaremos definir su estructura interna.
- Definir la interfaz con la que interactuar con dicho objeto.

- Diferencias entre crear una clase y usar una clase

- 1) Crear** una clase implica:

- Definir su nombre
 - Definir sus atributos

- 2) Usar** una clase implica:

- Crear nuevas instancias del objeto
 - Realizar operaciones sobre las instancias

- Pej: `L = [1,2]` y `L.append(3)`

CLASES Y HERENCIAS

•Ventajas del uso de POO.

- Permiten “empaquetar” datos en paquetes junto con procedimientos que podemos aplicar a través de una interfaz (ABSTRACCIÓN).
- Permiten el uso de “divide y vencerás”
 - ✓ Podemos implementar y testear el comportamiento de cada clase por separado.
 - ✓ La modularidad reduce la complejidad.

CLASES Y HERENCIAS

•Ventajas del uso de POO.

→ Permiten reutilizar el código

- ✓ Muchos módulos en Python crean nuevas clases.
- ✓ Cada clase tiene su propio entorno separado del resto.
 - Puedo nombrar un método de la misma manera para diferentes clases ya que cada método, se ejecuta dentro del entorno de su propia clase.
- ✓ La herencia permite a las subclases redefinir o extender una selección de comportamientos de sus clases “padre”.
 - No hace falta reescribir código que haga lo mismo para las subclases, se toma el código de la clase padre.

CLASES Y HERENCIAS

- **Definición de nuevos tipos.**

- La definición de tipos en Python tiene la siguiente estructura:

```
class <nombre de la clase> (object):  
    <definición de los atributos>
```

Clase
"padre"

Nombre
de la clase

Definición
de la clase

- Muy similar a la definición de funciones. El código bajo class, debe estar indentado para funcionar.
- Object es la clase "padre" de <nombre de la clase> de la que hereda sus atributos.

CLASES Y HERENCIAS

- **¿Qué son los atributos?**

- Son datos y procedimientos que pertenecen a dicha clase.
 - Ej. Dato (objeto) de coordenadas gps. Está compuesto de 2 elementos (x,y) que son sus atributos y puedo utilizar. Son 2 números, sí, pero tienen sentido dentro del objeto coordenadas gps.

- Para poder hacer uso de dichos atributos, necesitamos de **métodos.**

- Podemos pensar en los métodos como funciones que SOLO funcionan dentro de su clase.

- Para el caso de las coordenadas gps, podríamos definir un método que calcule la distancia entre 2 objetos de coordenadas. Este método no tendría sentido para calcular la distancia entre listas.

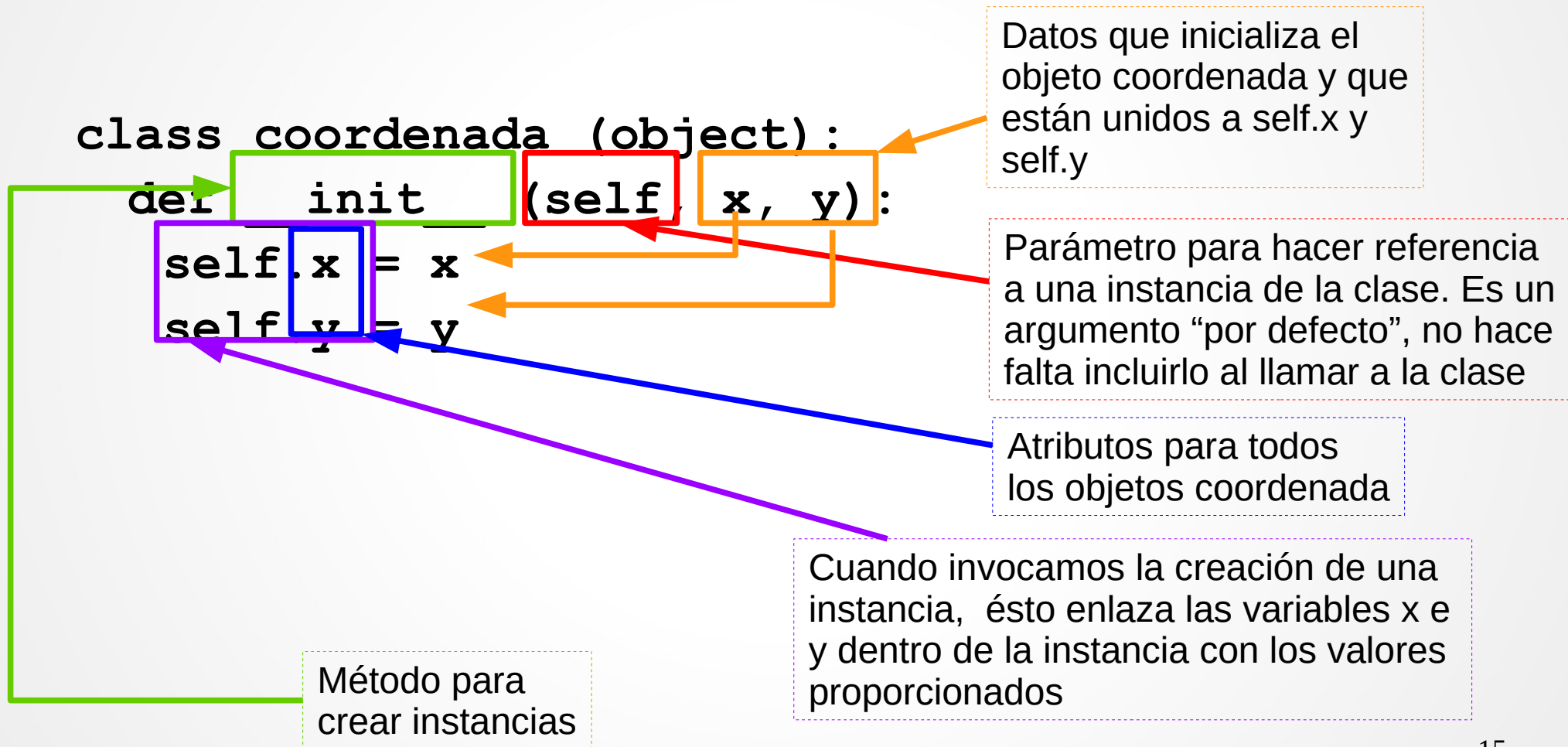
CLASES Y HERENCIAS

- **Creación de una instancia de una clase**
- Supongamos que vamos a definir una clase llamada coordenada

```
class coordenada (object):  
    def __init__ (self, x, y):  
        self.x = x  
        self.y = y
```

CLASES Y HERENCIAS

• Creación de una instancia de una clase



CLASES Y HERENCIAS

• Creación de una instancia de una clase

Ej: Si ejecutamos el código anterior...

```
c = coordenada (3,4)
origen = coordenada (0,0)
print (c.x)
3
print (origen.x)
0
```

Cada vez que llamo a `coordenada` se crean nuevas instancias (`c` y `origen`) del objeto `coordenada`

Uso del **punto** para acceder al atributo de una instancia

- Los atributos de los datos de una instancia son las variables de la instancia.

- No es necesario suministrar el argumento `self`, Python lo incluye automáticamente.

CLASES Y HERENCIAS

- **Creación de una instancia de una clase**

→ Podemos ver la creación de instancias (c y origen) como si fueran un nuevo “marco de trabajo”, donde se encuentran todas las variables definidas (x, y). Similar en concepto a los entornos de trabajo que se crean por las funciones.

CLASES Y HERENCIAS

- **¿Qué es un método?**

- Es un procedimiento, que solo funciona para esa clase (similar a las funciones).
- Utiliza el operador “.” para acceder:
 - ✓ A los atributos de un objeto.
 - ✓ A los métodos de un objeto.

CLASES Y HERENCIAS

- ¿Qué es un método?

- Definición de un método para coordenada que calcule la distancia euclídea para un objeto.

```
class coordenada (object):  
    def __init__ (self, x, y):  
        self.x = x  
        self.y = y  
  
    def distancia (self, otra):  
        x_diff_cuadr = (self.x - otra.x)**2  
        y_diff_cuadr = (self.y - otra.y)**2  
        return (x_diff_cuadr + y_diff_cuadr)**0.5
```

Se refiere a la instancia creada por la clase

Parámetro del método. En este caso, sería la otra coordenada contra la que quiero calcular la distancia

→ Aparte de self y la notación ".", los métodos se comportan como funciones¹⁹ (reciben parámetros, ejecutan operaciones y devuelven un resultado).

CLASES Y HERENCIAS

•¿Qué es un método?

Para acceder a los datos de la instancia creada, tenemos que llamarla OBLIGATORIAMENTE mediante self.

```
def distancia (self, otra):  
    x_diff_cuadr = (self.x - otra.x)**2  
    y_diff_cuadr = (self.y - otra.y)**2  
    return (x_diff_cuadr + y_diff_cuadr)**0.5
```

Mismo comportamiento para otra. Tenemos que hacer referencia al nombre de la instancia.

CLASES Y HERENCIAS

•¿Qué es un método?

•Ej.

```
c = coordenada (3,4)
origen = coordenada (0,0)
print (c.distancia(origen))
```

Objeto sobre el
que aplicar el
método

Parámetros sin incluir
self → Python entiende
que **self** en este caso
es **c**

Nombre
del método

Nos devuelve
5.0

•En este caso **c**, está heredando **distancia** de la definición de clase y automáticamente usa **c** como primer argumento.

CLASES Y HERENCIAS

- **¿Qué es un método?**

- También podemos llamar a una clase y método con la siguiente notación:

`coordenada.distancia(c, origen)`

→ Ambos equivalentes.

- Podemos considerar que `coordenada` tiene su propio marco de trabajo, donde se almacenan sus métodos.
- Al utilizar el **punto** estamos accediendo al método `distancia` existente dentro de ese marco.
- Ese método (`distancia`) tiene asociados 2 argumentos.

CLASES Y HERENCIAS

•¿Qué es un método?

Forma convencional	Forma equivalente
c = coordenada (3,4)	c = coordenada (3,4)
origen = coordenada (0,0)	origen = coordenada (0,0)
print (c.distancia(origen))	print (coordenada.distance(c,origen))

¡OJO! Porque aunque ambos print, imprimen lo mismo en el primer caso solo debemos suministrar 1 argumento, frente a los 2 del segundo print.

CLASES Y HERENCIAS

- **¿Qué es un método?**

- Ahora que tenemos nuestra clase y métodos, si hacemos:

```
print(c)
```

```
Out[5]: <__main__.coordenada at 0x7fd1e4131978>
```

- Podemos definir a través de otro método, lo que vuelca en pantalla una clase.

- Para ello usaremos `__str__method` dentro de la clase.

- Python llama a `__str__method` cuando hacemos `print`

CLASES Y HERENCIAS

- ¿Qué es un método?

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

- Este método funciona con cadenas de texto y nos permite devolver información sobre objeto, de manera más clara.

- Consulta del tipo de la clase.

```
print(type(c)) → <class '__main__.coordenada'>
```

- Conocer si un objeto es una instancia de otro objeto.

```
isinstance()
```

```
print (isinstance(c, coordenada))
```

```
True
```

CLASES Y HERENCIAS

- **¿Qué es un método?**

- Aislar métodos dentro de clases, me permite variar muy fácilmente el comportamiento de los métodos, basta con variar un método en una clase, para que éste se actualice en cascada.

CLASES Y HERENCIAS

•Operadores especiales

- Los básicos como `+`, `-`, `==`, `<`, `>`, `len()`, `print()`, ...

<https://docs.python.org/3/reference/datamodel.html#basic-customization>

- De la misma manera que hemos hecho con las cadenas de texto, definiendo un método que devuelva la información sobre el objeto de una manera más clara, podemos definir otros métodos:

- Suma de 2 instancias: `__add__(self, other)`
- Resta de 2 instancias: `__sub__(self, other)`
- Igualdad entre 2 instancias (`==`): `__eq__(self, other)`
- `self < otra` (less than=`lt`): `__lt__(self, other)`
- `len(self)` `__len__(self)`
- `print (self)` `__str__(self)`
- ...

CLASES Y HERENCIAS

•Operadores especiales

- Para el ejemplo que estamos desarrollando, podría añadir un método a coordenada que me calcule la resta de 2 instancias.

```
class coordenada (object):  
    def __init__ (self, x, y):  
        self.x = x  
        self.y = y  
  
    def distancia (self, otra):  
        x_diff_cuadr = (self.x - otra.x)**2  
        y_diff_cuadr = (self.y - otra.y)**2  
        return (x_diff_cuadr + y_diff_cuadr)**0.5  
    def __str__(self):  
        return "<" + str(self.x) + "," + str(self.y) + ">"  
    def __sub__(self, otra):  
        return coordenada(self.x-otra.x, self.y-otra.y)
```

CLASES Y HERENCIAS

- **Operadores especiales**

- Supongamos que queremos calcular la diferencia entre coordenadas.

`c - origen` → Devuelve

`<__main__.coordenada at 0x7ff8b02938d0>`

Lo que nos dice que la clase se ha ejecutado correctamente.

Aprovechamos el método que acabamos de añadir para mostrar el resultado.

`print (coordenada - origen)` → Devuelve

`<3,4>`

`c = coordenada (7,3), origen = coordenada (1,8)`

`print (c-origen)` → Devuelve

CLASES Y HERENCIAS

•Ejemplo. POO. Aplicación para organizar información de personas.

1)Definimos la clase Person.

Cada Persona tendrá unos atributos(nombre y año de nacimiento). Veremos cómo obtener el nombre, apellido, ordenar por grupos de gente, y obtener la edad. Básicamente acciones que nos permiten el acceso a los métodos de esta clase.

→ Abrir el script Personas.py

CLASES Y HERENCIAS

- **Ejemplo. POO. Aplicación para organizar información de personas.**

2) Definimos la clase MITPerson.

Añadimos una nueva clase a nuestra aplicación. En este caso define los atributos con la información relativa a una persona que trabaja en el MIT. Esta clase hereda los métodos de la clase Person.

→ Abrir el script MITPerson.py

CLASES Y HERENCIAS

•Visualización de las herencias.

Clase Person

```
Objeto Person:  
    __init__  
    GetLastName  
    __lt__
```

Cuando llamo a la clase MITPerson, esta llama al procedimiento `__init__` de su clase.

Pero esa llamada, a su vez llama a `__init__` en la clase Person.

De esta manera, en los objetos MITPerson, tenemos también su fecha de nacimiento y Apellido.

Clase MITPerson

```
Objeto MITPerson:  
    NextIDNum:  
    __init__  
    __lt__
```


CLASES Y HERENCIAS

•Ejemplo. POO. Aplicación para organizar información de personas.

3)Definimos las clases UG y Grad.

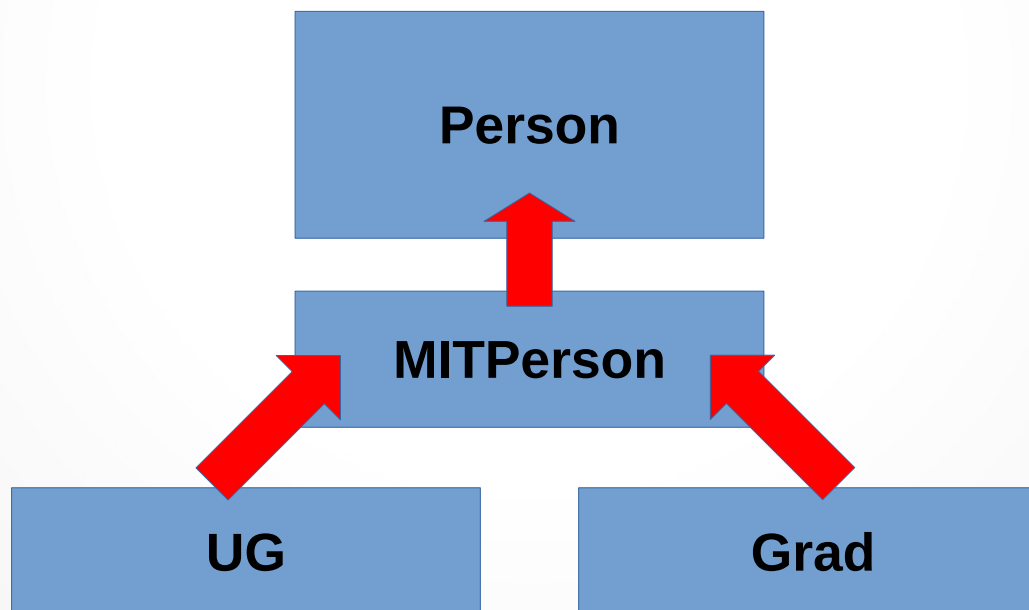
Con información relativa a estudiantes. Los estudiantes son personas del MIT por lo que heredarán los métodos de MITPerson que a su vez hereda de Person. Además incluirán atributos específicos de su clase. Definimos UG, Grad, que heredarán de la clase MITPerson

→ Abrir el script Alumnos.py

Nota: UG, undergraduate y hace referencia a los alumnos que no han terminado sus estudios todavía.

CLASES Y HERENCIAS

- **Ejemplo. POO. Aplicación para organizar información de personas.**
- Hasta ahora tenemos las siguientes clases y sus herencias.

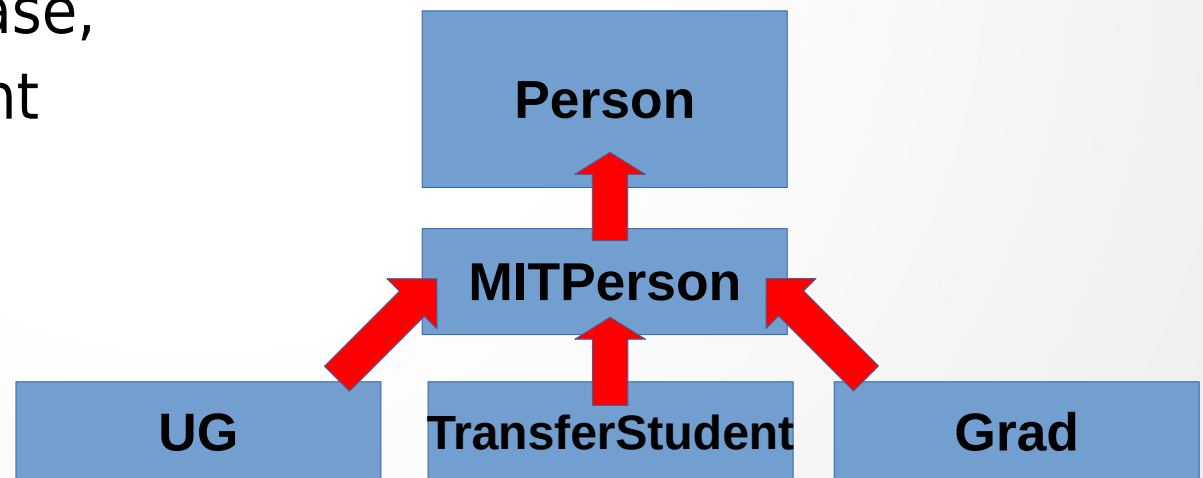


CLASES Y HERENCIAS

- **Ejemplo. POO. Aplicación para organizar información de personas.**

Supongamos que vemos que en nuestro esquema, faltan p.ej. los alumnos que han venido transferidos de otras facultades. Estos alumnos no los podríamos catalogar ni dentro de la clase UG, ni Grad.

Creamos una nueva clase, llamada TransferStudent



CLASES Y HERENCIAS

•Ejemplo. POO. Aplicación para organizar información de personas.

¿Qué implicaciones tiene añadir una nueva clase?

Podría parecer que ninguna, ya que TransferStudent también depende de MITPerson como UG y Grad.

Sin embargo, en el script Alumnos.py tenemos el siguiente código:

```
# creamos una función que nos diga si una
# instancia pertenece o bien a la clase UG
# o a la clase Grad.
def isStudent(obj):
    return isinstance(obj, UG) or isinstance(obj, Grad)
```

CLASES Y HERENCIAS

- **Ejemplo. POO. Aplicación para organizar información de personas.**

`isStudent`, evalúa si un objeto es estudiante, si pertenece a la clase `UG` ó `Grad`. No tiene en cuenta la nueva clase `TransferStudent` que hemos creado.

Podemos solucionar el problema, modificando la función `isStudent(obj)`

```
def isStudent(obj):  
    return (isinstance(obj,UG) or  
            isinstance(obj,Grad) or isinstance(obj,  
            TransferStudent))
```

CLASES Y HERENCIAS

- **Ejemplo. POO. Aplicación para organizar información de personas.**

Problema: Si bien la modificación previa soluciona el problema, en el momento que dentro de la definición de estudiante, tengamos otro tipo (alumnos becados, alumnos extranjeros, de doctorado, ...) tendríamos que modificar de nuevo nuestra función **isStudent**.

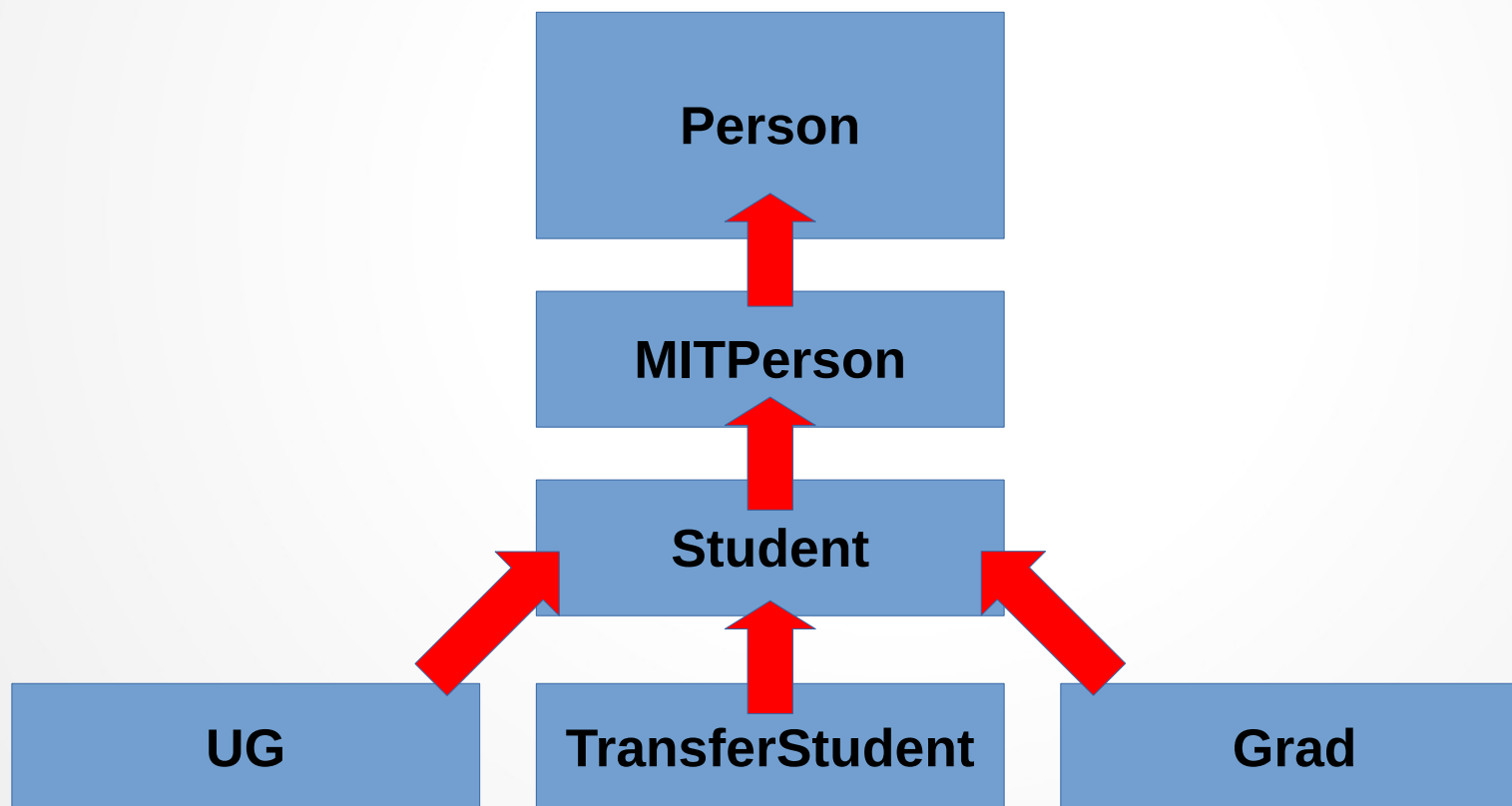
CLASES Y HERENCIAS

- **Ejemplo. POO. Aplicación para organizar información de personas.**

Solución: Definir una nueva clase sobre **UG**, **Grad** y **TransferStudent** (que llamaremos **Student**) que englobe a todos los posibles tipos de estudiante futuros. Modificar la función **isStudent** de manera que apunte no a cada una de las diferentes clases de estudiantes, sino a una que los englobe a todos.

CLASES Y HERENCIAS

- **Ejemplo. POO. Aplicación para organizar información de personas.**
- Hasta ahora tenemos las siguientes clases y sus herencias.



CLASES Y HERENCIAS

•Ejemplo. POO. Aplicación para organizar información de personas.

4)Definimos la clase Student, dependiente de MITTransfer y padre de UG, TransferStudent y Grad.

→ Abrir el script Alumnos2.py

Añadir Student a la la jerarquía, conseguimos más flexibilidad en nuestro árbol sin modificar o cambiar el comportamiento de las herencias desde MITPerson

CLASES Y HERENCIAS

- **Ejemplo. POO. Aplicación para organizar información de personas.**

- Empezamos a acumular métodos y clases.
- Para comprobar las clases con las que estamos trabajando.

```
import sys, inspect
classes = inspect.getmembers(sys.modules[__name__], inspect.isclass)
print (classes)
```

- Para comprobar las clases de las que hereda una subclase
- ```
herencias = inspect.getmro(UG)
print(herencias)
```

# CLASES Y HERENCIAS

- **Ejemplo. POO. Aplicación para organizar información de personas.**

- ¿Qué variables tenemos definidas?

**dir()** → Devuelve las variables definidas en el entorno de trabajo actual.

**globals()** → Devuelve un diccionario con TODAS las variables (incluidas las que no pertenecen al entorno de trabajo actual).

**locals()** → Devuelve un diccionario con todas la variables del entorno local.

# CLASES Y HERENCIAS

## •Ejemplo. POO. Aplicación para organizar información de personas.

5)Definimos la clase Professor.

Esta clase, heredará de la clase MITPerson

→ Abrir el script Profesores.py

6)Supongamos que ahora queremos que al llamar al método speak, en vez de mostrar el apellido únicamente, queremos que se muestre el nombre completo. Podríamos cambiar individualmente el método en cada clase, ó modificarlo en la clase de las que dependen el resto.

→ Abrir el script PersonalMIT2.py

# CLASES Y HERENCIAS

- **Ejemplo. POO. Aplicación para organizar información de personas.**

De la modificación previa se desprende que debemos tener mucho cuidado al anular métodos de clases superiores con los métodos de clases inferiores.

Como norma general, todos los métodos y atributos importantes de la clase principal deben estar soportados por las clases inferiores.

# CLASES Y HERENCIAS

## •Ejemplo. POO. Aplicación para organizar información de personas.

### 6) Añadir una nueva clase Grades (notas).

Usaremos esta clase para definir nuevas instancias de cuadernos de notas de los alumnos.

Crearemos una clase que incluya instancias de otras clases, dentro de ella.

El objetivo es crear una estructura que almacene los Grados de los estudiantes (aquellos que se han graduado) de manera que podamos reunir en una sola estructura datos y procedimientos. Ello permitiría manipular la estructura sin conocer sus detalles internos.

# CLASES Y HERENCIAS

- **Ejemplo. POO. Aplicación para organizar información de personas.**

Por tanto, dentro de la clase Grades o de cualquier instancia de Grades, tendremos una representación de estudiantes (Students) y de sus cuadernos notas.

→ Abrir el script Notas.py

# CLASES Y HERENCIAS

## •Ejemplo. POO. Aplicación para organizar información de personas.

```
class Grades(object):
 """Inicializamos la creación de cuadernos de grados"""
 def __init__(self):
 """Generamos un cuaderno de grados vacío"""
 self.students = [] # lista de objetos estudiantes
 self.grades = {} # me interesa poder obtener
 # las grados de un alumno
 # a través de su idNum
 self.isSorted = True # True si self.students está
```

ordenado

```
 def addStudent(self, student):
```

```
 """Asume: student es un tipo Student
 Añade student al cuaderno de grados"""
```

```
 if student in self.students:
```

```
 raise ValueError('Estudiante Repetido')
```

```
 self.students.append(student)
```

Si el alumno  
existe  
no añadirlo



# CLASES Y HERENCIAS

## •Ejemplo. POO. Aplicación para organizar información de personas.

### Método Añadir Estudiante

```
def addStudent(self, student):
 """Asume: student es un tipo Student
 Añade student al cuaderno de grados"""
 if student in self.students:
 raise ValueError('Estudiante Repetido')
 self.students.append(student)
 self.grades[student.getIdNum()] = []
 self.isSorted = False
```

Grades es un diccionario. Cuando añadimos un estudiante, cogemos la instancia correspondiente a dicho alumno y le aplicamos el método getIdNum() para encontrar su idNum. Este idNum lo usaremos como key en el diccionario.

Aunque la lista de estudiantes está vacía y por tanto no puede estar ordenada, le añado un semáforo que diga que lo está. Vemos la razón un poco más adelante.

# CLASES Y HERENCIAS

## •Ejemplo. POO. Aplicación para organizar información de personas.

### Método Añadir Notas

```
def addGrade(self, student, grade):
 """Asume: grade es coma flotante
 Añade grade a la lista de grados del estudiante"""
 try:
 self.grades[student.getIdNum()].append(grade)
 except KeyError:
 raise ValueError('El estudiante no está en el
cuaderno de grados')
```

Añadimos las notas a la lista de grados, tomando como valor key `student.getIdNum()`.

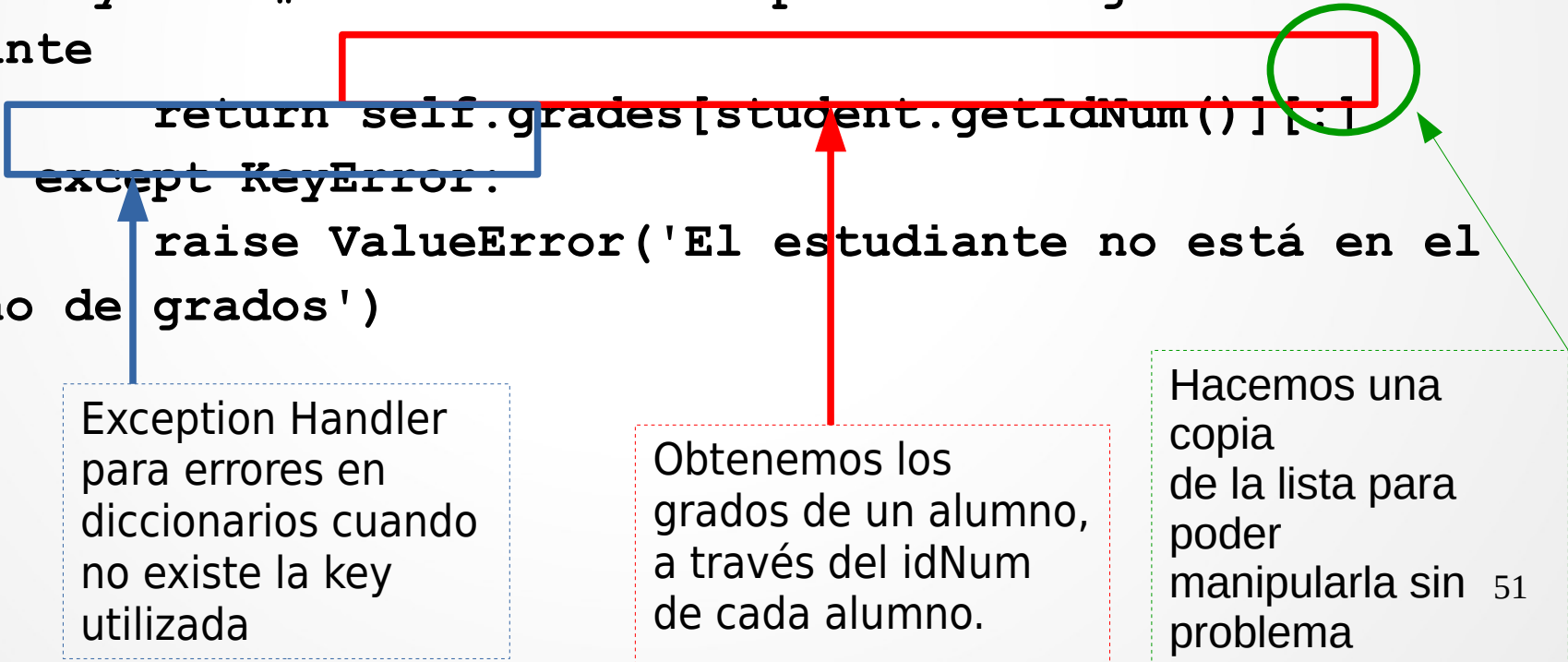
Mensaje de error en caso de intentar añadir notas a un estudiante que no esté dado de alta.

# CLASES Y HERENCIAS

## •Ejemplo. POO. Aplicación para organizar información de personas.

### Método Obtener Notas

```
def getGrades(self, student):
 """Devuelve las grados de un estudiante"""
 try: # devuelve una copia de los grados del
estudiante
 return self.grades[student.getIdNum()][:]
 except KeyError:
 raise ValueError('El estudiante no está en el
cuaderno de grados')
```



Exception Handler  
para errores en  
diccionarios cuando  
no existe la key  
utilizada

Obtenemos los  
grados de un alumno,  
a través del idNum  
de cada alumno.

Hacemos una  
copia  
de la lista para  
poder  
manipularla sin  
problema 51

# CLASES Y HERENCIAS

## •Ejemplo. POO. Aplicación para organizar información de personas.

Método Lista de estudiantes

```
def allStudents(self):
```

```
 """Devuelve una lista con los estudiantes del
 libro de grados"""
```

```
 if not self.isSorted:
 self.students.sort()
 self.isSorted = True
 return self.students[:]
```

```
 #devuelve una copia de la lista de estudiantes
```

Si el semáforo, definido inicialmente es FALSE, entonces y solo entonces, ordena la lista y nos devuelve una copia de la lista de estudiantes. Una vez ordenada la lista, vuelve a poner el semáforo a TRUE

# CLASES Y HERENCIAS

## •Ejemplo. POO. Aplicación para organizar información de personas.

### Método Informe de Grados

```
def gradeReport(course):
 """Supone: Course es un tipo grades"""
 report = []
 for s in course.allStudents():
 tot = 0.0
 numGrades = 0
 for g in course.getGrades(s):
 tot += g
 numGrades += 1
 try:
 average = tot/numGrades
 report.append(str(s) + '\ 's mean grade is '
 + str(average))
 except ZeroDivisionError:
 report.append(str(s) + ' has no grades')
```

Aplicamos el método/procedimiento allStudents() para obtener un iterable sobre el que hacer el for. Usamos un copia de la lista estudiantes.

Iteramos sobre todos los grados que tiene cada alumno.

# CLASES Y HERENCIAS

## •Ejemplo. POO. Aplicación para organizar información de personas.

### Método Informe de Grados

```
def gradeReport(course):
 """Supone: course es un tipo grades"""
 report = []
 for s in course.allStudents():
 tot = 0.0
 numGrades = 0
 for g in course.getGrades(s):
 tot += g
 numGrades += 1
 try:
 average = tot/numGrades
 report.append(str(s) + '\n's mean grade is '
 + str(average))
 except ZeroDivisionError:
 report.append(str(s) + 'has no grades')
```

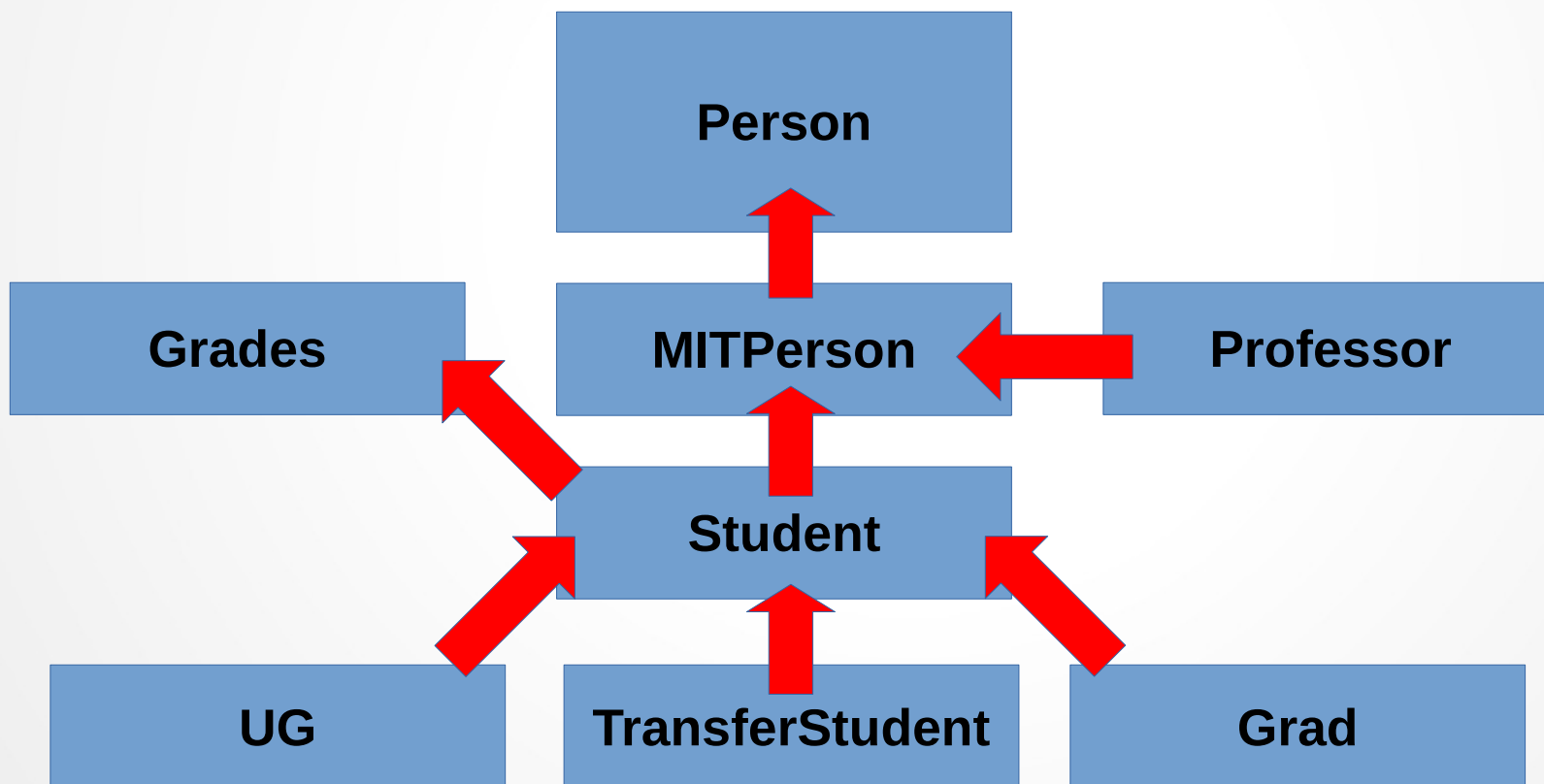
Si no hay error  
(numGrades = 0),  
calculamos la media  
del total de grados.

Añade al informe, la  
media de grados para  
ese alumno.

Añade retorno de carro a cada registro.  
has no grades

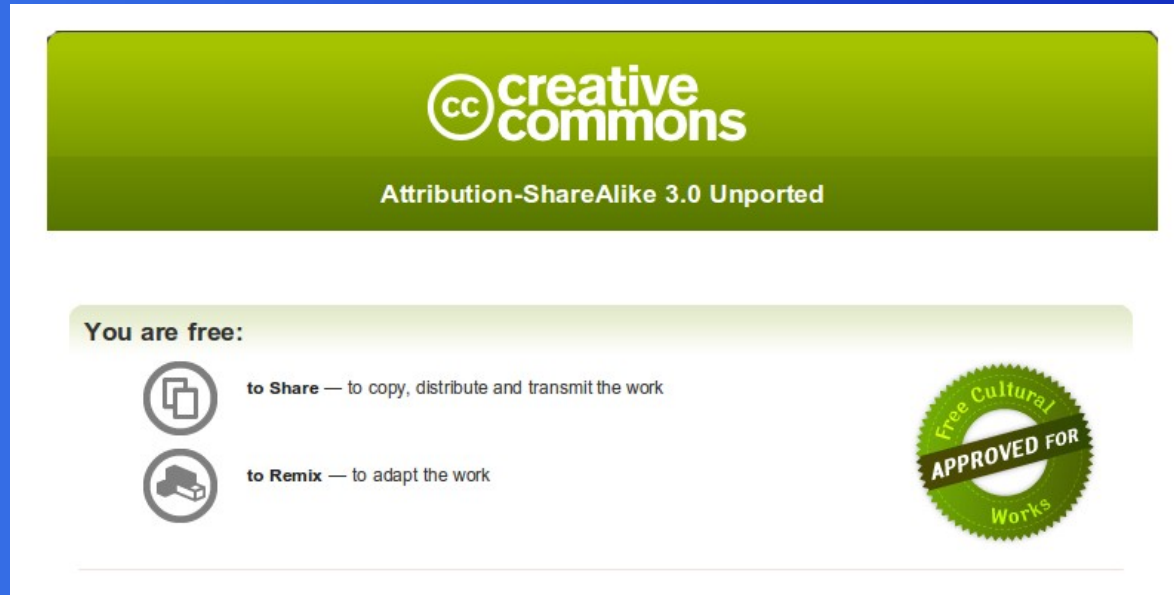
# CLASES Y HERENCIAS

- **Ejemplo. POO. Aplicación para organizar información de personas.**
- Ahora tenemos las siguientes clases y sus herencias.



## Copyright (c) 2022 Germán Alonso Lascurain

This work (but the quoted images, whose rights are reserved to their owners\*) is licensed under the Creative Commons “Attribution-ShareAlike” License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-sa/3.0/>



# Germán Alonso Lascurain

[germanalonso@opendeusto.es](mailto:germanalonso@opendeusto.es)

[galonso@datavaluemanagement.es](mailto:galonso@datavaluemanagement.es)