<codoacodo/>

Agencia de Aprendizaje a lo largo de la vida

DJANGO
Reunión 4

Python – Herencia, encapsulamiento y excepciones

BA

Les damos la bienvenida

Vamos a comenzar a grabar la clase

Reunión 05

Python – Herencia

- Clases y objetos, constructores, variables de instancia y de clase
- Visibilidad de atributos (público y privado)
- Generalización, herencia simple y múltiple
- Polimorfismo
- Clase abstractas

Reunión 06

Python – Excepciones

- Manejo de excepciones
- Árbol de herencia de las excepciones
- Excepciones personalizadas
- Lanzando excepciones
- Buenas prácticas en el manejo de excepciones

¿Qué es la Herencia?

Es una técnica de los lenguajes de programación para construir una clase a partir de una o varias clases, compartiendo atributos y operaciones. Básicamente, la **herencia** es la **implementación** de la **generalización** en un **lenguaje de programación**

Es una relación entre clases en la que una clase comparte la estructura y/o el comportamiento definidos en una (herencia simple) o más clases (herencia múltiple)

¿Qué es el Encapsulamiento?

Es la característica que permite asegurar que el contenido de la información de un objeto está oculto al “mundo exterior”

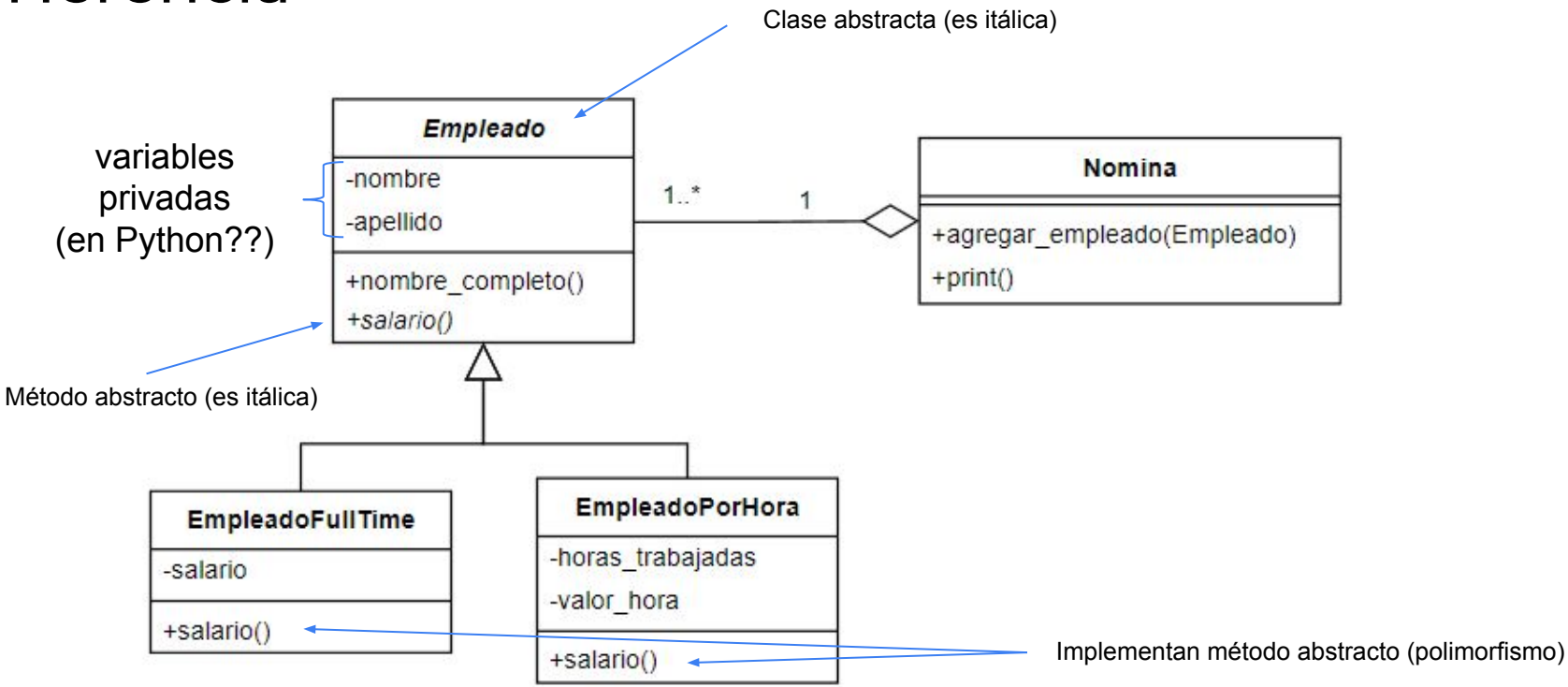
El encapsulamiento, al separar el comportamiento de un objeto de su implementación interna, permite la modificación de este sin que se tengan que modificar las aplicaciones que lo utilizan.

BAJO ACOPLAMIENTO

ALTA COHESIÓN

Herencia y encapsulamiento en Python

Herencia


Clase abstracta (es itálica)
variables privadas (en Python??)
Empleado
-nombre
-apellido
+nombre_completo()
+salario()
1..*
1
Nomina
+agregar_empleado(Empleado)
+print()
Método abstracto (es itálica)
EmpleadoFullTime
-salario
+salario()
EmpleadoPorHora
-horas_trabajadas
-valor_hora
+salario()
Implementan método abstracto (polimorfismo)

Herencia y Polimorfismo

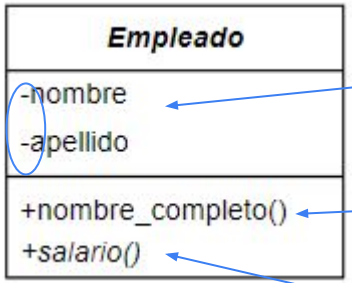
```python
class Empleado(ABC):
    def __init__(self, nombre, apellido):
        self.__nombre = nombre
        self.__apellido = apellido

    @property
    def nombre_completo(self):
        return f"{self.__nombre} {self.__apellido}"

    @property
    @abstractmethod
    def salario(self):
        pass
```

```python
class EmpleadoFullTime(Empleado):
    def __init__(self, nombre, apellido, salario):
        super().__init__(nombre, apellido)
        self.__salario = salario

    @property
    def salario(self):
        return self.__salario
```

```python
class EmpleadoPorHora(Empleado):
    def __init__(self, nombre, apellido, horas_trabajadas, valor_hora):
        super().__init__(nombre, apellido)
        self.__horas_trabajadas = horas_trabajadas
        self.__valor_hora = valor_hora

    @property
    def salario(self):
        return self.__horas_trabajadas * self.__valor_hora
```

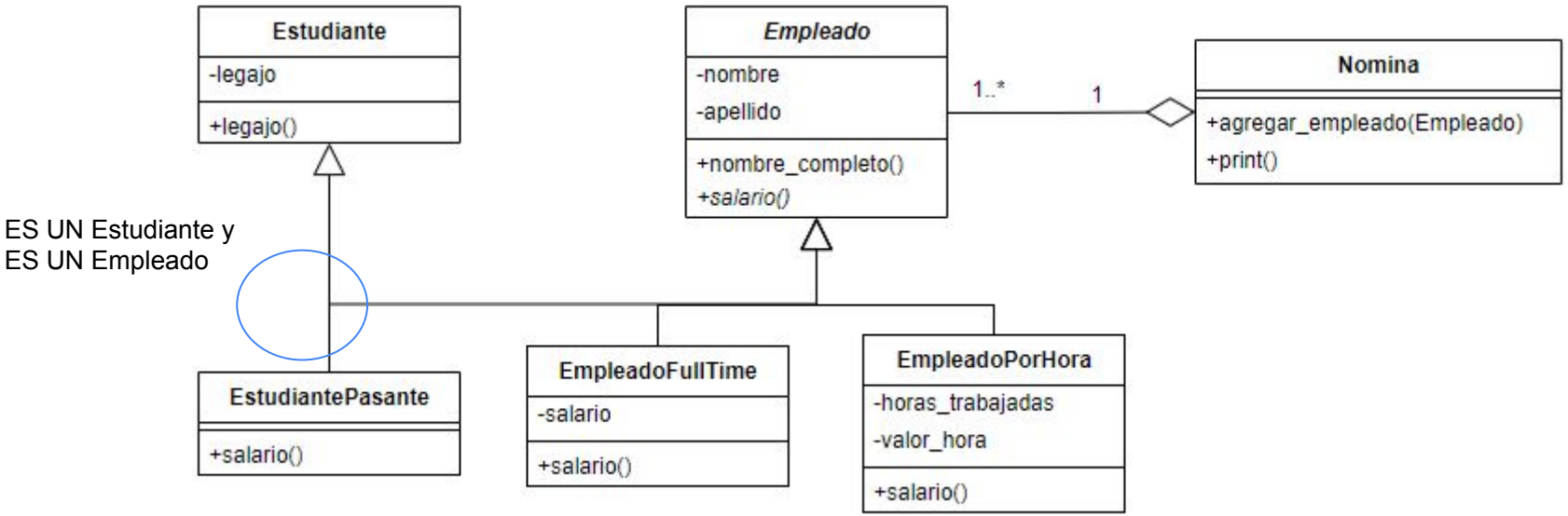
Encapsulamiento

NO HAY VARIABLES
PRIVADAS EN PYTHON..
CONVENCIÓN “_” o “__”




```python
from abc import ABC, abstractmethod

class Empleado(ABC):
    def __init__(self, nombre, apellido):
        self.__nombre = nombre
        self.__apellido = apellido

    @property
    def nombre_completo(self):
        return f"{self.__nombre} {self.__apellido}"

    @property
    @abstractmethod
    def salario(self):
        pass
```

Herencia Múltiple



Herencia Múltiple

```python
class Estudiante():
    def __init__(self, legajo):
        self.__legajo = legajo

    @property
    def legajo(self):
        return self.__legajo
```

El orden importa

Llamo a los constructores de las clases base

```python
class EstudiantePasante(Empleado, Estudiante):
    def __init__(self, nombre, apellido, legajo):
        Empleado.__init__(self, nombre, apellido)
        Estudiante.__init__(self, legajo)

    # Tengo que implementar la propiedad salario porque hereda de empleado
    @property
    def salario(self):
        return 0
```

¿Qué es una Excepción?

Los errores detectados durante la ejecución del programa se llaman EXCEPCIONES

Si una excepción ocurre en algún lugar de nuestro programa y no es capturada en ese punto, va subiendo (burbujeando)

Hasta que es capturada en alguna función que ha hecho la llamada. Si en toda la «pila» de llamadas no existe un control de la excepción el programa se parará

Excepciones en Python

try:

Ejecutar este código

except:

Ejecutar este código solo **SI** arriba ocurre alguna **excepción**

else:

Ejecutar este código solo si **NO** ocurre alguna **excepción**

finally:

Ejecutar **SIEMPRE** este código

Excepciones

```python
def mostrar_division_entera(dividendo, divisor):
    try:
        print("Intentando hacer la división")
        resultado = dividendo // divisor
        print(f"El resultado entero de la divisón es: {resultado}")
    except TypeError:
        print('Revisar los operandos hay un dato mal cargado...')
    except ZeroDivisionError:
        print('No se puede dividir por cero...')
    except Exception:
        print('Algo anduvo mal...')
    else:
        print("Este programa nunca falla..")
    finally:
        print('El super programa ha finalizado..')
```

Herencia de excepciones

Todas las excepciones en Python deben ser instancias de una clase que derive de BaseException

Si se definen nuevas excepciones, se recomienda a los programadores derivar de Exception

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- StopAsyncIteration
      +-- ArithmeticError
      |    +-- FloatingPointError
      |    +-- OverflowError
      |    +-- ZeroDivisionError
      +-- AssertionError
      +-- AttributeError
      +-- BufferError
      +-- EOFError
      +-- ImportError
      |    +-- ModuleNotFoundError
      +-- LookupError
      |    +-- IndexError
      |    +-- KeyError
      +-- MemoryError
      +-- NameError
      |    +-- UnboundLocalError
      +-- OSError
```

Solo son algunas

Exepciones personalizadas

```python
class DivisorNegativoError(Exception):
    """Excepción lanzada se divide por números negativos"""
    pass
```

Palabra reservada para lanzar una excepción

Debe estar definida antes de ser utilizada

```python
def mostrar_division_entera(dividendo, divisor):
    try:
        if divisor < 0:
            raise DivisorNegativoError("Mandaron un número negativo")
        print("Intentando hacer la división")
        resultado = dividendo // divisor
        print(f"El resultado entero de la divisón es: {resultado}")
    except TypeError:
        print('Revisar los operandos hay un dato mal cargado...')
    except ZeroDivisionError:
        print('No se puede dividir por cero...')
    except Exception as ex:
        print(f'Algo anduvo mal: {ex}')
    else:
        print("Este programa nunca falla..")
    finally:
        print('El super programa ha finalizado..')
```

Se suele crear un módulo específico para las excepciones de negocio

Aserciones

```python
def mostrar_division_entera(dividendo, divisor):
    try:
        assert divisor >= 0, "Mandaron un número negativo"
```

Si el divisor es menor a 0 lanza la excepción **AssertionError** con el mensaje “Mandaron un número negativo”

No te olvides de completar la asistencia y consultar dudas

Recordá:

- Revisar la Cartelera de Novedades.
- Hacer tus consultas en el Foro.

TODO EN EL AULA VIRTUAL