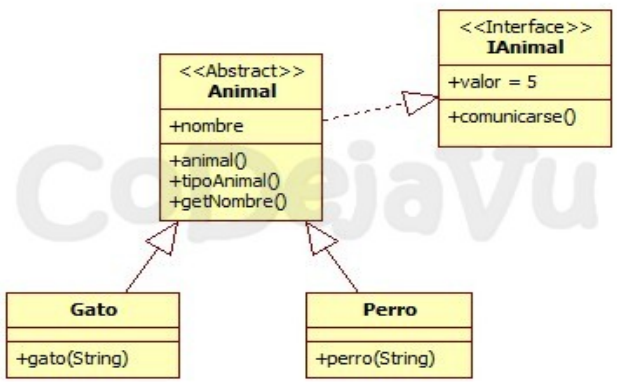
UNIVERSIDAD DE MURCIA

Clases Abstractas, interfaces

Polimorfirmo

L. Daniel Hernández < *ldaniel@um.es* >

Dpto. Ingeniería de la Información y las Comunicaciones
Universidad de Murcia
14 de noviembre de 2023



[1]

[1] Imagen: Codejavu

Índice de Contenidos

Desarrollo

Clases Abstractas

- La **abstracción** es el proceso por el que captamos las características esenciales de un objeto tanto en atributos como en comportamiento.
- **Las clases abstractas codifican altos niveles de abstracción**
- Una **clase abstracta representa** a objetos para los que **conocemos algunas de las variables** que definen su estado y **conocemos algunas de sus funcionalidades pero no sabemos establecerlas** sin concretar el objeto.
- Un **método abstracto** es aquel que tiene declaración pero no está definido.
- Una **clase abstracta** es la que tiene (al menos) un método abstracto
- Una clase abstracta es una clase que (1) **no se puede instanciar** (no representan algo específico), (2) **se usa únicamente para definir subclases**
- Una **subclase puede ser abstracta** si la clase padre lo es.
- Existirán **subclases** de una clase abstracta que **implementarán** los métodos abstractos.
- Por tanto, tendremos **un método** (abstracto) que **se ejecutará de forma diferente** en cada subclase: **Polimorfismo de inclusión o subtipado**.
- Una clase abstracta puede tener constructores:
  - Pero no sirven para construir instancias (por definición de clase abstracta).
  - Las subclases no abstractas deberá de sobreescribir el constructor **explícito** con los mismos parámetros y deben llamar al constructor del padre (abstracto) con `super()`.

Ejemplo de Clase Abstracta

Ejemplo:

- La clase **Animal** tiene los métodos `caminar()` o `comer()` pero no se sabe cuáles son las acciones concretas que deben realizarse.
- Por tanto, la clase **Animal** **debe de ser** **abstracta**.
- Las clase **Ave** y **Mamífero** son subclases de Animal.
- **Ave** y **Mamífero** heredan el método `caminar()` que lo harán de forma diferente,
  - Una **ave** camina con **dos** patas pero un **mamífero** camina con **cuatro** patas.

```
>>> from abc import ABC, \
...         abstractmethod
>>> class Animal(ABC):
...   @abstractmethod
...   def caminar(self):
...     pass
...
>>> class Mamifero(Animal):
...   def caminar(self):
...     print("... con 4 patas")
...
>>> class Ave(Animal):
...   pass
...
```

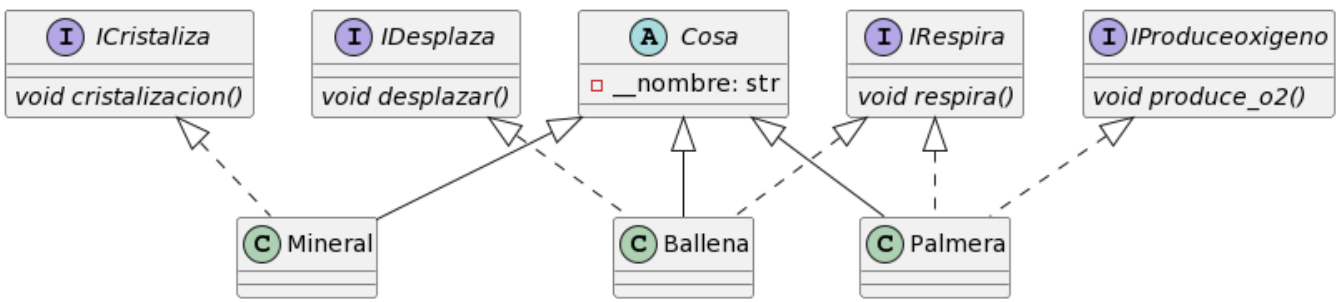
```
>>> perro = Mamifero()
>>> perro.caminar()
... con 4 patas
>>> gallo = Ave()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class Ave w
```

Si una clase hija **no** define el método abstracto sigue siendo una clase abstracta.

Desarrollo

Interfaces

- Una **interfaz** es una abstracción **basada solo** en la funcionalidad y definen un **tipo**.
- Es solo un conjunto de **métodos**. NO tienen atributos de instancia.
- Son métodos **necesarios** en algunos objetos para que cumplan sus **requisitos mínimos de funcionalidad**.
- Dichos objetos son del **tipo** que diga el nombre de la interface (**subtipado estructural**).
- **Polimorfismo de subtipado**: Añaden **especialización**/extensión a los objetos.
- Cada objeto implementan la especialización de forma diferente:
  - **Son métodos abstractos y polimórficos**



En el mundo de la Cosa solo se conoce el nombre de las cosas. Se tienen las siguientes subclases directas: mineral, ballena y palmera. a sus objetos, se quieren dotar de las funcionalidades: cristaliza, desplaza, respira y produce oxígeno.

Interfaces en Python

- **Python** sigue la premisa **"Duck Typing"**.
- Para **Python** todo los objetos que tengan ciertos métodos definen un tipo de objetos.
  Ejemplo: Todo lo que implemente `__iter__()` y `__next__()` es un iterador.
- En **Python** una clase debe **definir todos** los métodos requeridos para que se reconozca "como un pato" (un objeto de la interface).
- En **Python se debe registrar una interface** para que cualquier objeto que implemente los métodos **sea reconocido** como objeto del tipo de la interface.
- Se distinguen
  - **Interfaces Informales**:
    Aquellas que **no** fuerzan a la sobreescritura de todos los métodos.
  - **Interfaces Formales**:
    Aquellas que **sí** fuerzan a la sobreescritura de todos los métodos.

Registro de una Interfaces en Python

- **Python** es demasiado versátil para trabajar con registros de interfaces. Puede consultar p.e. https://realpython.com/python-interface/
- Una forma "sencilla" es sobreescribir el métododo de clase **__subclasshook__()**. Se requiere el paquete `abc` (Abstract Base Classes).

```
>>> import abc

>>> class Interface(metaclass=abc.ABCMeta): # Es mejor poner abc.ABC
...     """Esta es la interface usada. Se registra
...    definiendo __subclasshook__ (con todos los métodos)
...     """
...     @classmethod
...     def __subclasshook__(cls, subclass): # Aquí añade los métodos
...         if cls is Interface:
...             return (hasattr(subclass, 'metodo1') and
...                     callable(subclass.metodo1) and
...                     hasattr(subclass, 'metodo2') and
...                     callable(subclass.metodo2))
...         return NotImplemented
...
```

- Lectura: Si la subclase tiene el atributo `metodo1` y se puede invocar y tiene el atributo `metodo2` y se puede invocar, entonces es `True` que es de la clase `IMovimiento`

Crear Interfaces Informales en Python

1. Crear el registro del interface con la siguiente sintaxis:

```
>>> import abc

>>> class IMovible(metaclass=abc.ABCMeta):
...     """Esta es la interface usada. Se registra
...    definiendo __subclasshook__ (con todos los métodos)
...     """
...     @classmethod
...     def __subclasshook__(cls, subclass): # Aquí añade los métodos
...         if cls is IMovible:
...             return (hasattr(subclass, 'movimiento') and
...                     callable(subclass.movimiento) and
...                     hasattr(subclass, 'lo_que_mueve') and
...                     callable(subclass.lo_que_mueve))
...         return NotImplemented
...
```

2. Crear las clases que quieran implementar la interface

```
>>> class Vaca:
...     def movimiento(self):
...         print("Se desplaza a 4 patas")
...     def lo_que_mueve(self):
...         print("Muevo la cola")
...
```

Ejemplo de Interfaces Informales en Python

- Solo los objetos de las clases que implementen la interfaces serán reconocidos como instancias de la interface.

```
>>> class Vaca:
...     def movimiento(self):
...         print("Se desplaza a 4 patas")
...     def lo_que_mueve(self):
...         print("Muevo la cola")
...
>>> v = Vaca(); isinstance(v, IMovible)  # El objeto es de su clase
True
>>> issubclass(Vaca, IMovible) # Están los dos métodos
True

>>> class Serpiente:
...     def movimiento(self):
...         print("Repto como una serpiente")
...
>>> s = Serpiente(); isinstance(s, IMovible) # Falta el método queMueve()
False
>>> issubclass(Serpiente, IMovible) # Falta el método queMueve()
False
```

Observa que en `Vaca` no se ha indicado ningún tipo de herencia sobre `IMovimiento` para que Python la reconozca como una subclase.

Creación de Interfaces Formales en Python - I

1. Crear el registro de la interface y **añadir** los métodos como **métodos abstractos** con la siguiente sintaxis:

```
>>> import abc

>>> class IMovible(abc.ABC):
...     """Esta es la interface.
...     Es necesario definir __subclasshook__ con todos los métodos abstractos.
...     """
...     @classmethod
...     def __subclasshook__(cls, subclass):
...         if cls is IMovible:
...             return (hasattr(subclass, 'movimiento') and
...                     callable(subclass.movimiento) and
...                     hasattr(subclass, 'lo_que_mueve') and
...                     callable(subclass.lo_que_mueve))
...         return NotImplemented
...     @abc.abstractmethod
...     def movimiento(self):  # Esto lo diferencia de las informales
...         pass
...     @abc.abstractmethod
...     def lo_que_mueve(self):
...         pass
...
```

Interfaces Formales en Python - II

2. Parar **forzar** su implementación deberá **indicar que la interface es una superclase**.

```
>>> class Vaca:      # Esta clase, si quiere no implementa la interface
...     def movimiento(self):
...         print("Se desplaza a 4 patas")
...     def lo_que_mueve(self):
...         print("Muevo la cola")
...

>>> class Serpiente(IMovible):  # Esta clase está forzada a implementar la interface
...     def movimiento(self):
...         print("Repto como una serpiente")
...
>>> v = Vaca();
>>> print(isinstance(v, IMovible))  # El objeto es de su clase
True
>>> print(issubclass(Vaca, IMovible)) # Están los dos métodos
True


>>> s = Serpiente(); # Falta el método lo_que_mueve()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class Serpiente with abstract method lo_que_mu
```

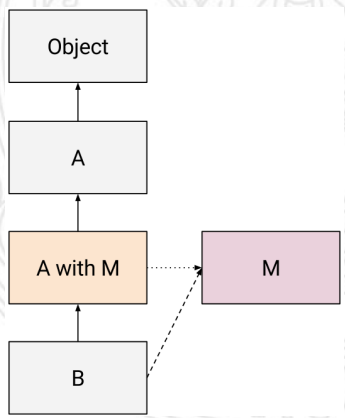
Observa que en `Vaca` no se ha indicado ningún tipo de herencia sobre `IMovimiento` para que Python la reconozca como una subclase (definición informal).

Desarrollo

Mixin (Mezclas)

- Un **mixin** no es ni una clase abstracta ni una interface.
- Es una clase que
  - **Añade funcionalidad** a otras clases
    - Permite a un programador **inyectar código** en las clases que lo necesiten.
  - **No añade especialización**
    - Los métodos inyectados en la clase no se pueden considerar como una especialización de la clase.
  - **No tiene sentido** que tengan objetos o clases derivadas.
    - El mixin no se considera una clase base.
- Se usa siempre de la siguiente manera:

  la clase B extiende (A con Mixin)
- **Conceptualmente** **es como si** se creara la clase **A-con-M** donde
  - La clase padre de **A-con-M** es **A**
  - La clase padre de **B** es **A-con-M**
- Ejemplos que hay que conocer: collections.abc — Abstract Base Classes for Containers¶



Ejemplo de Mixin en Python

https://programmerclick.com/article/8047526022/

- En **Python** los **Mixins** se crean con **herencia múltiple**

```
class Displayer():
    def display(self, message):
        print(message)

# Este Mixin añade la funcionalidad de Mostrar + Registrar
# Para Mostrar se delega en el padre.
# Este Mixin solo lo pueden usar las clases cuyo padre tengan display()
class LoggerMixin():
    def log(self, message, filename='logfile.txt'):
        with open(filename, 'a') as fh:
            fh.write(message)
    def display(self, message):
        super().display(message) # << No todos los objetos lo tienen
        self.log(message)

class MySubClass(LoggerMixin, Displayer):
    def log(self, message):
        super().log(message, filename='subclasslog.txt')

subclass = MySubClass()
subclass.display("This string will be shown and logged in subclasslog.txt")
```

Desarrollo

Intefaces vs Clases Abstractas vs Mixin - I

https://codesitio.com/recursos-utiles-para-tu-web-o-blog/cursos/curso-de-java-clases-abstractas-e-interfaces/

- Una **clase abstracta** se caracteriza por:
  - **Constructores**: Tiene al menos uno (el implícito)
  - **Variables de Clase**: Se admiten
  - **Campos**: Pueden tener cualquier modificador de visualización.
  - **Métodos estáticos**: Tendrán **signatura y cuerpo**. No pueden ser abstractos.
  - **Métodos de instancia**:
    - Pueden tener **signatura y cuerpo**.
    - **Al menos uno** tendrá el modificador `@abstractmethod`: Tendrá **signatura pero no cuerpo**
  - **Sobreescritura de métodos:** está permitida
  - Los métodos pueden tener **cualquier modificador** de visualización.
  - **Admiten herencia**.

    En esto casos se habla de subclase y de superclase.

Intefaces vs Clases Abstractas vs Mixin - I

https://codesitio.com/recursos-utiles-para-tu-web-o-blog/cursos/curso-de-java-clases-abstractas-e-interfaces/

- Una **interface** se caracteriza por:
  - **Constructores**: NO tiene.
  - **Variables de Clase**: Se admiten
  - **Campos**: NO tiene. Algunos lenguajes relajan esto.
    Es decir, no existen variables de objeto y son solo de clase.
  - **Métodos estáticos**: Tendrán **signatura y cuerpo**.
  - **Métodos de instancia**: Contendrán **solo la signatura**, sin cuerpo.
    Tendrá el modificador `@abstractmethod` o sentencia similar.
  - **Sobreescritura de métodos: Todos** se deben **sobreescribir**.
  - Todos los métodos (estáticos y de instancia) serán **públicos**.
  - **Admiten herencia**.
    En esto casos se habla de subinterface y de superinterface.

Intefaces vs Clases Abstractas vs Mixin - II

- Una **Mixin** se caracteriza por:
  - **Constructores**: NO tiene.
  - **Variables de Clase**: Se admiten
  - **Campos**: Se admiten
  - **Métodos estáticos**: Tendrán **signatura y cuerpo**.
  - **Métodos de instancia**:
    - Deberían tener **signatura y cuerpo**.
    - Pero **se permite** que tengan el modificador `@abstractmethod`.
  - **Sobreescritura de métodos:** No se deberían de sobreescribir salvo que sean abstractos.
  - Todos los métodos (estáticos y de instancia) serán **públicos**.
  - No son auténticas clases. No tiene sentido crear subclases del Mixin. Evitar `@abstractmethod`.
  - Nota: serán heredados por las subclases de la clase Mezcla pero las subclases no deberían de modificarlos (no deberán crear especificidad en los métodos del Mixin)

Intefaces vs Clases Abstractas vs Mixin - III

	C. Abstracta	Interface	Mixin
Constructor	Sí	**No**	**No**
Atributos de clase	Sí	Sí	Sí
Atributos de instancia	Sí	**No**	Sí
Métodos estáticos	Sí	Sí	Sí
Métodos de clase	Sí	Sí	Sí
Métodos de instancia	Sí. Al menos uno abstracto.	Todos abstractos	Sí. Alguno podría ser abstracto.
Sobreescritura	Sí	Sí. Todos.	**No** se debería.
Herencia	Sí	Sí	**No**

Desarrollo

Tipos Genéricos: Introducción

- Considera el siguiente código para definir un objeto `Entero`

```
class Entero:
    self._valor: int
    def get(self) -> int:
        return self._valor
    def set(self, valor: int):
        self._valor = valor
```

- Si ahora queremos objetos con la misma funcionalidad para `float`, `str`, ... **habrá que repetir el código para cada uno de los tipos de datos**.
- Una alternativa es utilizar el siguiente código a falta de indicar el tipo de dato `<T>`

```
1 class Tipo<T> {  // Clase paramétrica
2   private T t;
3   public  T   get()    { return t;   }
4   public void set(T t) { this.t ← t; }
5 }
```

- Estaría genial poder generalizar esta idea con datos simples y estructuras de datos. P.e. Se podría reutilizar todo el código de cierta clase `Lista` a falta de determinar el tipo de dato; es decir, utilizar `Lista[T]` en vez de una para cada tipo de dato como `ListaEntero`, `ListaDouble`, ...

Uso de Tipos Genéricos en Python

```
>>> from typing import TypeVar, Generic # Definición de tipos
>>> T = TypeVar('T')           # Definición de un tipo genérico
>>> class Tipo(Generic[T]):  # Clase genérica
...     _valor: T
...     def get(self) -> T:
...         return self._valor
...     def set(self, valor: T):
...         self._valor = valor
...
>>> entero: Tipo[int] = Tipo[int]()  # Especifica el genérico como entero
>>> entero.set(5)
>>> entero.get()
5
>>> # El tipado de Python es caprichoso
>>> entero.set(5.6)   # Pero Python funciona por contrato !!!
>>> entero.get()
5.6
```

- Los errores se mostrarán en el IDE siempre que declares el tipo antes del constructor, pero te dejará ejecutar el programa !!!

Desarrollo

Ejercicio

Ejercicio.

Los artefactos dispone de un motor capaz de indicar el número de revoluciones al que va. Existen muchos tipos de motores cada uno con distintos tipos de atributos. ?Cómo modelar entonces la situación?

Ejercicio.

- La clase **SerieTV** tiene el atributo número de episodios.
- La clase **VideoJuego** tiene el atributo horas estimadas de juego.
- Implementa para ambas clases su interface Getter/Setter.
- Se considera ahora que ambas clases pueden ser `prestadas`. ¿qué se debería de hacer para que tengan las siguientes acciones?
  - entregar(): cambia el atributo prestado a true.
  - devolver(): cambia el atributo prestado a false.
  - esEntregado(): devuelve el estado del atributo prestado.
- Se considera ahora que comparables: Si dos videojuegos tienen el mismo número de horas estimadas se consideran que son iguales. Si dos series de tv tienen el mismo número de episodios se consideran que son iguales.

  Ten en cuenta que pueden existir otros tipos de ocio que no pueden ser prestados, como los canales de streaming, pero si pueden ser comparables, por ejemplo, por el precio.

Ejercicio

Ejercicio.

Modela la siguiente situación y dibuja el diagrama UML asociado.
Un taller de carpintería tiene diferentes tipos de herramientas que conforme se van rompiendo se van añadiendo herramientas nuevas. Todas las herramientas tienen un uso que es diferente dependiendo de su objetivo. Se distinguen las herramientas universales y las herramientas específicas. Las universales constan de un conjunto de componentes que se pueden ir cambiando. Un ejemplo son las navajas suizas. Por otro lado las herramientas específicas se catalogan con un tipo basado en las tareas en las que pueda ser usado. Entre las específicas se distinguen la sierra circular que admite cierta potencia eléctrica y la lijadora que debe usarse a cierta velocidad de forma manual y cuya cuchilla se puede cambiar. Todas las herramientas eléctricas se pueden encender o apagar.

Ejercicio.

Modela la siguiente situación y dibuja el diagrama UML asociado.
Un zoológico tiene diferentes tipos de animales. Todos los animales tienen un nombre con un número de identificación. Todo carnívoros tienen una dieta especial diferente. Los hay que cazan y los que no. De entre los que cazan están los que son capaces de nadar, como el tigre, y los que no, como el león. También hay carnívoros cazadores que vuelan como el halcón peregrino. De entre los que no cazan está los carroñeros como los buitres. También hay animales herbívoros, como los loros, capaces de volar, los hipopótamos, capaces de nadar, y las cebras

Ejercicio

Ejercicio.

Modela la siguiente situación y dibuja el diagrama UML asociado.
Cada recurso particular tiene un nombre y un precio; pero no se pueden construir instancias de esta clase. Los recursos forman grupos (el de los coches, el de la carne, etc...) y de cada grupo se puede saber si es comestible o no, y cada uno tiene un IVA diferente. Un producto es capaz de calcular su precio y obtener el precio final de una lista de productos. Son recursos los coches (de los que sabe su velocidad), la carne (de lo que se sabe su origen) y el pan (se conoce sus calorías). El nombre de cada coche, carne o pan coincide con el de la clase a la que pertenece. De todos ellos se puede calcular su precio. De entre todos los objetos se pueden distinguir los comestibles (de los que se sabe cómo se comen y el sonido que hacen al masticarlos) y los que se mueven (acelerando y frenando).

Ejercicio.

Se quiere una clase que almacene una **lista** de figuras geométricas. A dicha lista se le puede añadir o quitar elementos en tiempo de ejecución. Hay figuras abiertas, formadas por una secuencia de segmentos rectos, y las hay cerradas, como el círculo (caracterizado por su radio) o el cuadrado (caracterizado por la longitud del lado). Construida la lista se quiere saber el área de todas las figuras y el número de lados finitos que hay entre todas las figuras. También se quiere conocer el número total de figuras que se han construido durante la ejecución del programa.