UNIVERSIDAD
DE MURCIA

Tema 4. Clases Abstractas e Interfaces

Tecnología de la Programación

Material original por L. Daniel Hernández (*ldaniel@um.es*)

Editado por:

- Sergio López Bernal (*slopez@um.es*)
- Javier Pastor Galindo (*javierpg@um.es*)

Dpto. Ingeniería de la Información y las Comunicaciones
Universidad de Murcia

13 de octubre de 2025

Índice de contenidos

UNIVERSIDAD
DE MURCIA

Clases abstractas

Clases abstractas

- La **abstracción** es el proceso por el cual extraemos las características esenciales de un objeto, tanto en atributos como en comportamiento.
- Se enfoca en lo que hace un objeto, no en cómo lo hace.
- **Ejemplo**: Podemos hablar del concepto de "Vehículo" (sin especificar si es un coche, moto, etc.), donde sabemos que tiene ciertas características (ruedas, motor) y ciertos comportamientos (arrancar, detenerse), pero los detalles concretos varían.
- Una **clase abstracta** representa objetos que tienen algunas características y funcionalidades conocidas, pero no están completamente definidos.
- **Ejemplo**: Una clase abstracta `Vehículo` puede tener métodos como `acelerar()` o `frenar()`, pero no se especifica cómo funcionan, pues cada tipo de vehículo tiene características que los hace diferentes.

Clases abstractas

- Un método abstracto es aquel que tiene declaración pero no está definido. Una clase abstracta es la que tiene (al menos) un método abstracto.
- Como una **clase abstracta no está completamente definida**, no se puede instanciar (crear objetos). Existirán subclases de una clase abstracta que implementarán los métodos abstractos.
- **Ejemplo**: No podremos tener objetos de la clase abstracta `Vehículo` (pues tiene métodos abstractos `acelerar()` o `frenar()` sin definir), pero sí de subclases como `Coche` o `Moto` que definirán dichos métodos.
- Una subclase puede ser abstracta si la clase padre lo es.
- Por tanto, tendremos un método (abstracto) que se ejecutará de forma diferente en cada subclase (**polimorfismo**):
- Una clase abstracta puede tener constructores:
  - Pero no sirven para construir instancias (por definición de clase abstracta).
  - Las subclases no abstractas deberán sobrescribir el constructor explícito y llamar al constructor del padre 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 clases Ave y Mamífero son subclases de Animal.
- Ave y Mamífero heredan el método caminar():
  - Un ave camina con **dos** patas pero un mamífero con **cuatro**.

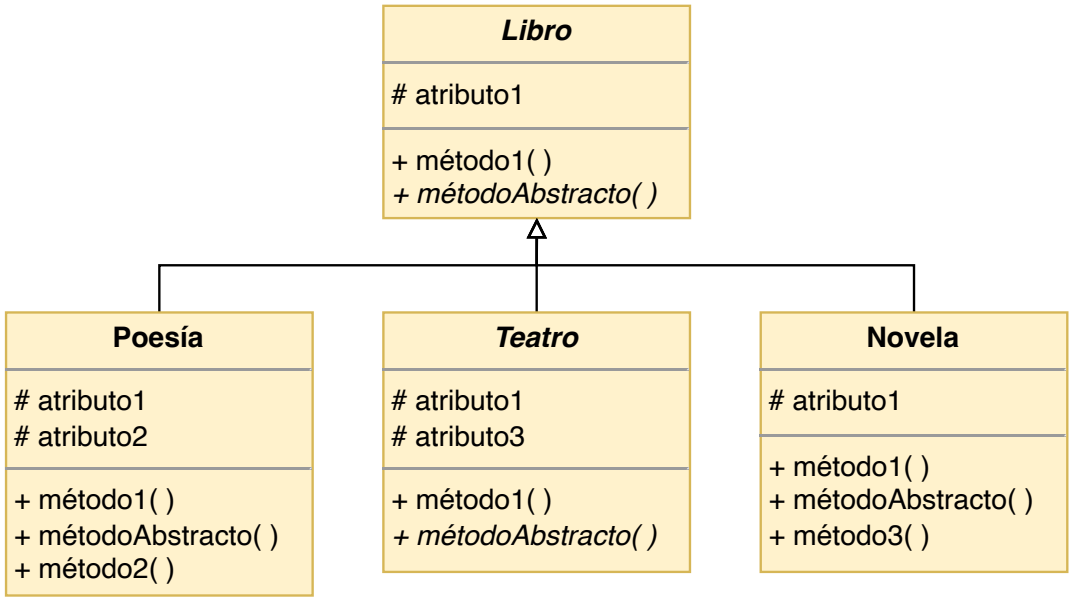
```
from abc import ABC, abstractmethod

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()
gallo = Ave()
```

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

Clases abstractas en UML

- El nombre de la clase abstracta se representa en *cursiva*.
- Los métodos abstractos se indican también en *cursiva*.
- Una subclase que implementa el método abstracto no lo indica en cursiva (ya no es abstracto en esa clase).
- Una clase abstracta puede incluir tanto métodos abstractos como concretos.
- Ejemplo de **Libro** como clase abstracta. El método abstracto es implementado por las clases **Poesía** y **Novela**. Sin embargo, **Teatro** no ofrece una implementación, por lo que también se declara como abstracta.



UNIVERSIDAD
DE MURCIA

Duck Typing

Duck Typing en Python

- **Duck Typing** en Python:
  - *"Si camina como un pato y suena como un pato, probablemente sea un pato"*
  - Si un objeto tiene los métodos que se esperan (por ejemplo, `volar()`), podrá ejecutarse independientemente de su tipo.

```
class Pato:
    def volar(self):
        print("El pato vuela.")

class Avion:
    def volar(self):
        print("El avión vuela.")

...
objeto.volar() # No importa el tipo, solo que exista el método
```

- En Python, el **Duck Typing** se basa en la presencia de métodos sin forzar a que el objeto tenga un tipo concreto.
  - Los lenguajes de programación con *tipado nominal* (como Java) verificaría si el objeto es de una clase antes de ejecutar el método.
- Si queremos forzar o asegurar que una clase implemente ciertos métodos, se debe usar **interfaces formales**.

UNIVERSIDAD
DE MURCIA

Interfaces

Interfaces

- Una **interfaz** es una abstracción que se centra **solo en la funcionalidad** y define un tipo de objeto.
- Contiene solo **métodos**, no incluye atributos o datos. Es un conjunto de acciones que los objetos que la implementan deben poder realizar.
- Los métodos de una interfaz son necesarios para que los objetos que la implementen cumplan con los **requisitos mínimos de funcionalidad**.
- Los objetos que implementan una interfaz son considerados del tipo definido por la interfaz. Esto se conoce como *subtipado estructural*.
- Las interfaces añaden una **especialización** o una **extensión** a los objetos, permitiéndoles ofrecer funcionalidad adicional.
- Cada objeto puede implementar la interfaz de manera diferente:
  - Los métodos de las interfaces son abstractos (sin implementación por defecto) y polimórficos (pueden comportarse de manera distinta).

Interfaces formales en Python

- En Python, se pueden usar **interfaces formales** para definir un conjunto obligatorio de métodos.
- Python es demasiado versátil para trabajar con registros de interfaces (`https://realpython.com/python-interface/`).
- Estas interfaces formales se definen usando la clase base 'abc.ABC':

```python
from abc import ABC, abstractmethod

# Definimos la interfaz formal "Volador"
class IVolador(ABC): # En Python se suele añadir una I en interfaces

    @abstractmethod
    def volar(self):
        """Método que debe implementarse."""
        pass
```

- Se dice que **una clase implementa una interfaz** cuando se desea que tengan obligatoriamente todos los métodos declarados en dicha interfaz.
  - Si quiero que una clase tenga la funcionalidad de Volador, dicha interfaz debe definir métodos personalizados.
  - No tiene tanto acoplamiento como en herencia, donde las superclases ya definen todo el estado y comportamiento de las subclases.

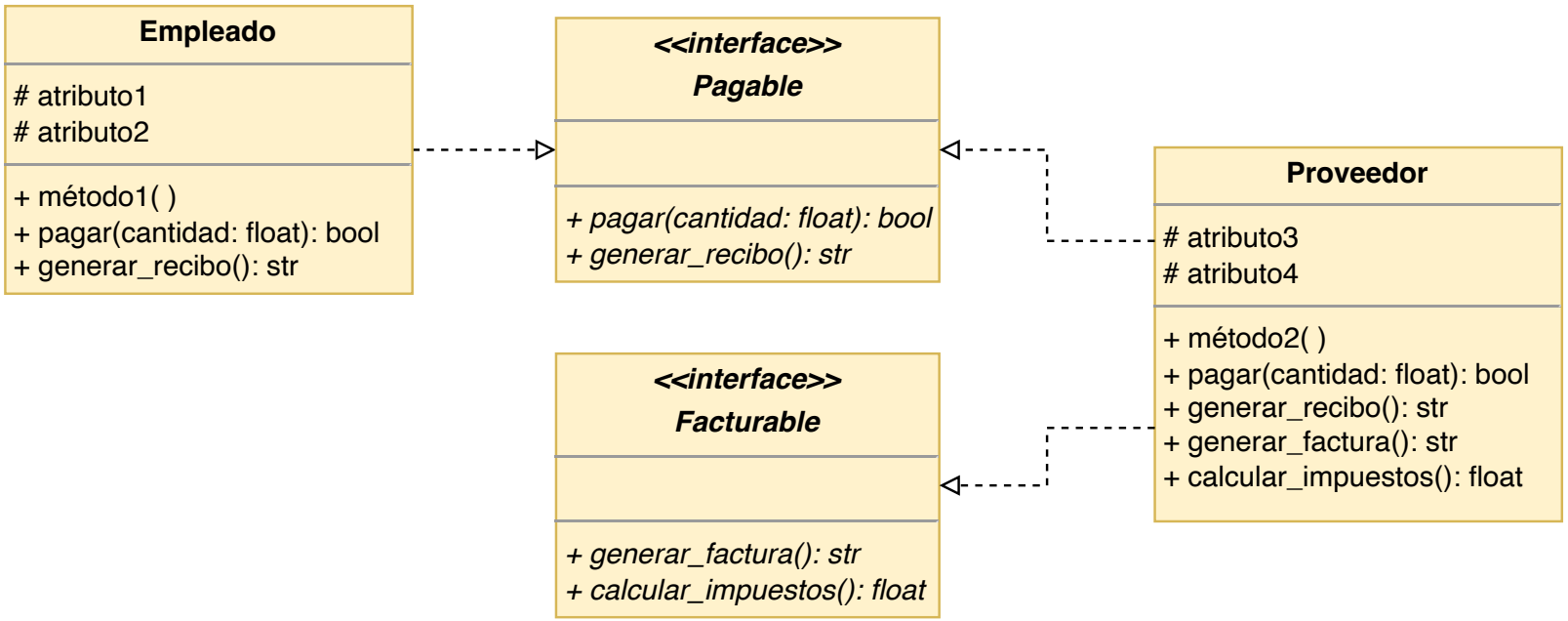
Interfaces formales en Python

- Para que un objeto implemente una interfaz formal, la clase hereda directamente de la interfaz.
  - Al ser métodos abstractos, están obligados a implementarlos.
  - Una clase también es del tipo de la interfaz que implementa.
  - Una clase puede implementar varias interfaces.

```
# Clase Pato que implementa la interfaz Volador
class Pato(IVolador):

    def volar(self):
        print("El pato está volando.")

# Clase Avion que implementa la interfaz IVolador
class Avion(IVolador):

    def volar(self):
        print("El avión está volando.")

pato = Pato()    # tipo Pato y también IVolador
pato.volar()

avion = Avion() # tipo Avion y también IVolador
avion.volar()

print(isinstance(pato, IVolador))    # Salida: True
print(issubclass(Avion, IVolador))   # Salida: True
```

Interfaces en UML

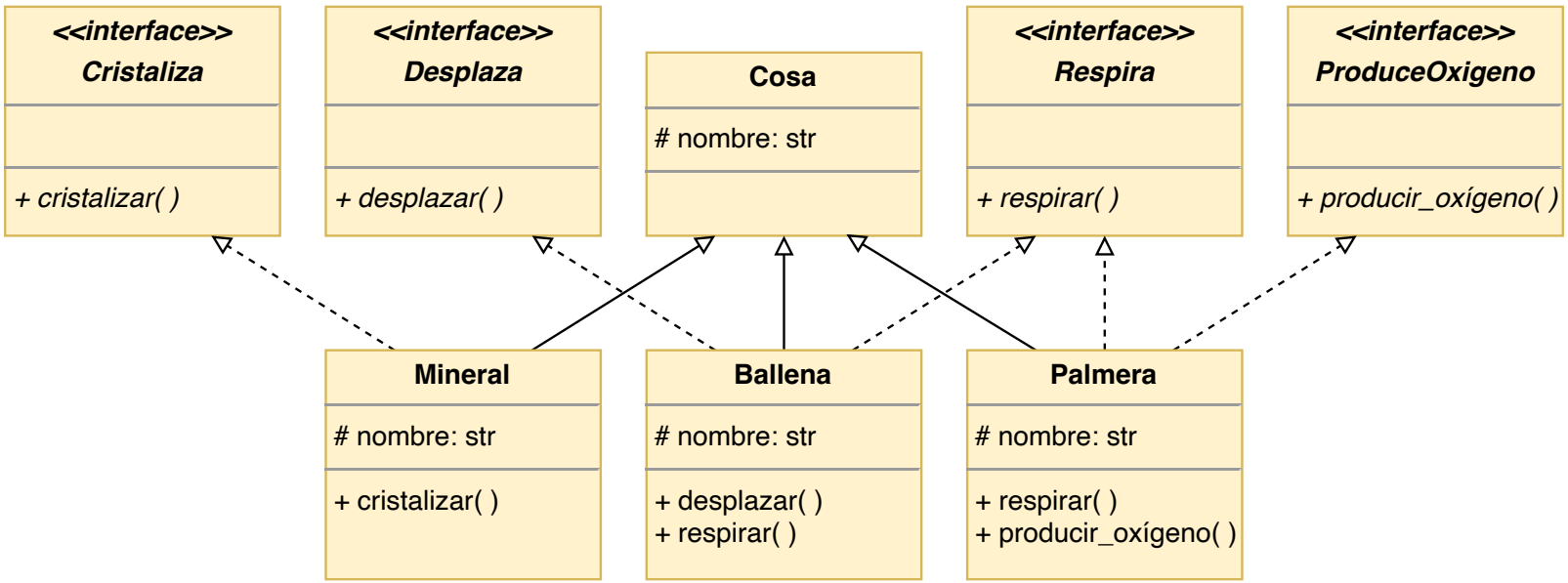
- Una interfaz se representa con el estereotipo <<**interface**>> encima del nombre (sin incluir una I delante del nombre).
- El nombre de la interfaz se indica en *cursiva*.
- Los métodos definidos en una interfaz son siempre **abstractos** y se muestran en *cursiva*.
- La relación entre una clase y una interfaz se representa con una flecha de herencia discontinua (**realización**).
- Una clase puede implementar varias interfaces.



Interfaces en UML

Ejemplo con herencia e interfaces

- Una clase puede tener una o más superclases (aunque en la asignatura no trabajaremos con herencia múltiple).
- Una clase puede implementar una o más interfaces.



UNIVERSIDAD
DE MURCIA

Clases abstractas vs Interfaces

Clases abstractas vs Interfaces

- Una clase abstracta se caracteriza por:
  - **Constructores**: Tiene al menos uno (el implícito)
  - **Atributos de clase**: Se admiten (con cualquier modificador de visualización)
  - **Atributos de instancia**: Se admiten (con cualquier modificador de visualización).
  - **Métodos estáticos, de clase o instancia**:
    - Pueden tener signatura y cuerpo
    - **Al menos uno** tendrá el modificador @abstractmethod que tendrá signatura pero no cuerpo.
    - Cualquier modificador de visualización.
  - **Sobreescritura de métodos:** está permitida (y obligatoria en al menos un caso).
  - Admiten herencia.

    En esto casos se habla de subclase y de superclase.

Clases abstractas vs Interfaces

- Una interface se caracteriza por:
  - **Constructores**: NO tiene.
  - **Atributos de clase**: Se admiten.
  - **Atributos de instancia**: NO tiene. Es decir, no existen atributos de objeto y son solo de clase. Algunos lenguajes relajan esto.
  - **Métodos estáticos, de clase o instancia**:
    - Contendrán solo la signatura, sin cuerpo.
    - Tendrán el modificador @abstractmethod.
    - Todos los métodos serán públicos.
  - **Sobreescritura de métodos: Todos** se deben sobreescribir.
  - Admiten herencia.

    En esto casos se habla de subinterface y de superinterface.

Clases abstractas vs Interfaces

	Clase Abstracta	**Interface**
Constructor	Sí	**No**
Atributos de clase	Sí	Sí
Atributos de instancia	Sí	**No**
Métodos	Al menos un abstracto	**Todos abstractos**
Sobreescritura	Sí	Sí. **Todos**.
Herencia	Sí	Sí

UNIVERSIDAD
DE MURCIA

Polimorfismo

Polimorfismo de un objeto

Ahora, un mismo **objeto** puede ser tratado como cualquiera de sus tipos, según lo necesite el contexto:

- **Tipo original**: Tipo de la clase concreta con la que se creó el objeto.
- **Tipos heredados**: El objeto también es del tipo de todas las clases de las que hereda, directa o indirectamente.
- **Interfaces**: Si implementa interfaces, el objeto es de esos tipos también.

```
class Volador(ABC):
    @abstractmethod
    def volar(self):
        pass

class Animal:
    def respirar(self):
        print("El animal respira.")

class Pato(Animal, Volador):
    def volar(self):
        print("El pato vuela.")

pato = Pato()

print(isinstance(pato, Pato))     # True (tipo original)
print(isinstance(pato, Animal))   # True (superclase)
print(isinstance(pato, Volador)) # True (interfaz)
```

UNIVERSIDAD
DE MURCIA

Ejercicios

Ejercicio

Ejercicio.

Los artefactos disponen 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.

1. Implementa los métodos Getter/Setter en la clase SerieTV con el atributo número de episodios y la clase VideoJuego con el atributo horas estimadas de juego.
2. Se considera ahora que ambas clases pueden ser **productos prestados**. ¿qué se debería 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.
3. Se considera ahora que son **comparables**: Si dos videojuegos tienen el mismo número de horas estimadas se considera que son iguales. Si dos series de TV tienen el mismo número de episodios se considera que son iguales.
   Ten en cuenta que pueden existir otros tipos de ocio que no pueden ser prestados, como los servicios de streaming, pero sí pueden ser comparables, por ejemplo, por el precio.

Ejercicio

Ejercicio.

Cada recurso particular tiene un nombre y un precio; pero no se pueden construir instancias de esta clase.

Los recursos forman los grupos de los coches, el de la carne, o el del pan. 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).

Cada grupo tiene un IVA diferente, siendo capaz de calcular su precio y obtener el precio final de una lista de productos.

Hay grupos que son comestibles (de los que se sabe cómo se comen y el sonido que hacen al masticarlos) y movibles (acelerando y frenando).