Tema 3. Introducción a la programación orientada a objetos.

Vicente Benjumea García

Programación-II
Departamento de Lenguajes y Ciencias de la Computación.
E.T.S.I. Informática. Univ. de Málaga.

Tema 3. Introducción a la programación orientada a objetos.

- Conceptos fundamentales de la programación orientada a objetos.
- Clases y objetos.
- Métodos especiales.
- Composición.
- Herencia y redefinición del comportamiento.
- Clases abstractas.
- Protocolos.

Conceptos fundamentales de la POO

Programación Orientada a Objetos. Clases y objetos

- Es un paradigma de programación que nos permite diseñar programas definiendo **abstracciones** (clases) que modelan los **datos** que representan el problema que queremos resolver.
- La Programación Orientada a Objetos se caracteriza por la definición de **clases** y la creación de **objetos** que encapsulan datos y algoritmos:
  - **Atributos** (datos): almacenan el estado interno del objeto.
  - **Constructores** (algoritmos): construyen y definen el estado inicial de los objetos.
  - **Métodos** (algoritmos): definen el comportamiento del objeto. Permiten la manipulación e interacción entre objetos.
  - La **Clase** define las características de los **objetos** que pertenecen a ella.

Relaciones entre clases: Herencia

- La relación de **Herencia** representa una *relación* en la cual una Clase (clase derivada, o subclase) **es una** *especialización* o *extensión* de otra Clase (clase base, o superclase).
  - El **principio de sustitución de Liskov** especifica que un objeto de una **clase derivada** (**subclase**) puede *ser considerado* y *utilizado* como un objeto de la **clase base** (**superclase**), sin afectar a la corrección del programa. Proporciona un soporte adecuado para el **polimorfismo**.
  - La **vinculación dinámica** permite que las clases derivadas (subclases) puedan redefinir el comportamiento de los métodos definidos en la clase base (superclase).

- En Python, todas las clases y objetos derivan de la clase **object**, que es la clase base de toda la jerarquía de clases de Python, y define el comportamiento básico y común a todas las clases de Python.

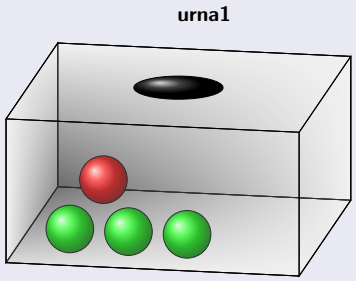
Relaciones entre clases: Composición

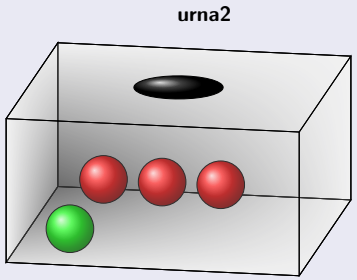
- La relación de **Composición** representa una *relación* en la cual un objeto **tiene** o **está compuesto** por otros objetos.

Abstracción: Urna

Una *urna* es una *caja* que contiene votos positivos y negativos.

- Comportamiento de la *urna* (**métodos**):
  - Introducir votos positivos y negativos.
  - Calcular el resultado de la votación.
- Estado interno de la *urna* (**atributos**):
  - La identificación de la urna.
  - La cuenta del total de votos positivos dentro de la urna.
  - La cuenta del total de votos negativos dentro de la urna.
- Constructor de la *urna* (**constructores**):
  - Un objeto *urna* se debe crear e inicializar con una determinada identificación, y la cuenta votos positivos y negativos con el valor cero.

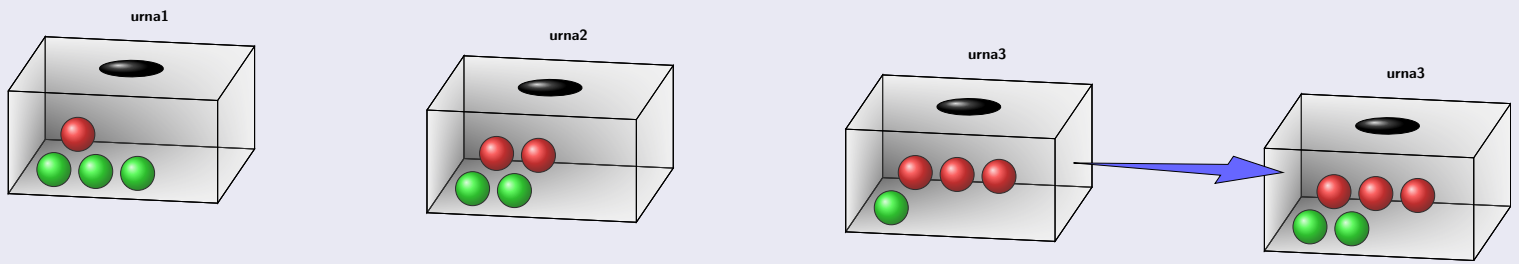




Abstracción: Urna

Dada la abstracción (**clase**) *Urna* especificada en el ejemplo anterior:

- Podemos **crear** múltiples *instancias* (**objetos**) de ella.
  - `urna1 = Urna("vot-1"), urna2 = Urna("vot-2"), urna3 = Urna("vot-3")`
- Cada *instancia* (objeto) de la *Clase Urna* posee su propio estado (**atributos**).
  - La identificación y cuenta de votos positivos y votos negativos que contiene.
- Todas las *instancias* de la **Clase** *Urna* tienen el mismo comportamiento.
- Podemos interactuar con cada objeto independientemente, a través de los **métodos** que definen su comportamiento.
  - Manipular el estado de cada objeto. Por ejemplo, podemos introducir *votos positivos y negativos* en los objetos `urna1`, `urna2` y `urna3`.
  - Consultar el resultado de la votación. Por ejemplo, si consultamos el resultado de la urna `urna3` después de las operaciones anteriores, el resultado es **falso**.



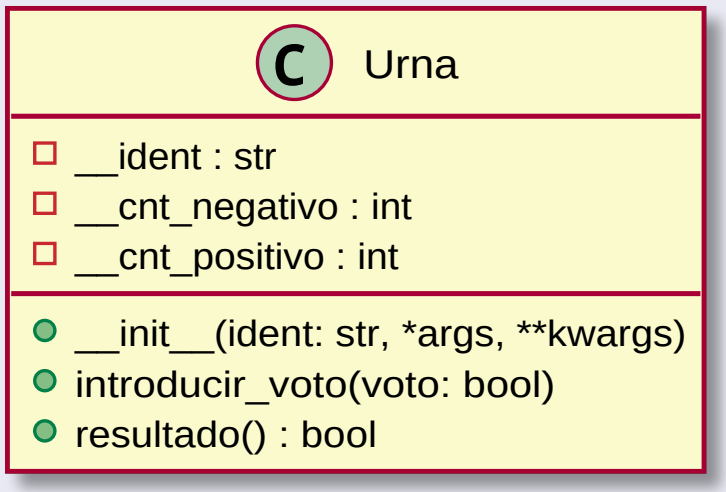
Clases y objetos

Clase

- Una **Clase** representa una *abstracción de datos*, especifica las **características** de unos **objetos**: su estado y su *comportamiento*.

Representación gráfica: diagrama de clases en UML

- Una clase se representa mediante una *caja* con tres compartimentos, conteniendo cada uno de ellos el nombre, atributos y métodos de la clase.



- En los métodos del diagrama de clases no aparece el parámetro `self`.

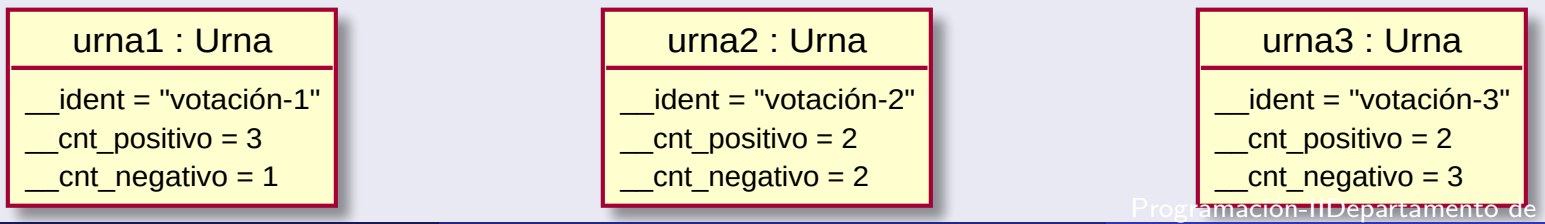
Clases y objetos

Objeto

- Un **objeto** es una *instancia* (*caso particular concreto*) de una determinada **Clase**.
- Las características del *objeto* (estado y comportamiento) están determinadas por la *Clase* a la que pertenece.
  - El constructor permite **crear** e inicializar a cada objeto.
    - Cada *objeto* **almacena** y contiene su propio estado interno (*atributos*), de forma *independiente* de los otros *objetos.*
  - El objeto podrá ser **manipulado** e interactuar con otros objetos a través de los *métodos* definidos por la *Clase* a la que pertenece.
- Los objetos se crean y se destruyen durante la ejecución del programa. Puede haber muchos objetos **distintos** que sean de la misma *Clase* (y también de distintas *Clases*).

Representación gráfica: diagrama de objetos en UML

- Un objeto se representa mediante una *caja* con dos compartimentos, conteniendo el primero el **nombre** del objeto y de la clase a la que pertenece, y el segundo los **valores** concretos de los atributos del objeto.



Niveles de protección y acceso en Python

- Existen tres niveles de protección y acceso, determinados por el **prefijo del nombre**:
  - Acceso **privado**: el acceso a los atributos y la invocación a los métodos sólo se puede realizar desde dentro de la **propia clase**. El nombre del atributo o método debe comenzar con __
  - Acceso **protegido**: el acceso a los atributos y la invocación a los métodos sólo se puede realizar desde dentro de la **propia clase** y también desde dentro de las **subclases**. El nombre del atributo o método debe comenzar con _
  - Acceso **público**: el acceso a los atributos y la invocación a los métodos se puede realizar desde cualquier parte del código. El nombre del atributo o método **no** debe comenzar con _
- Los nombres que comienzan y terminan por __ están **reservados** para uso interno de Python, por ejemplo `__name__`, `__init__`, etc.

Nivel de Acceso			*Desde propia Clase*	*Desde una Subclase*	*Desde resto Código*
▫■	−	`__privado`	SÍ	NO	NO
◇◆	#	`_protegido`	SÍ	SÍ	NO
○●	+	`publico`	SÍ	SÍ	SÍ

Atributos (almacenan la información de cada objeto)

- Los atributos **almacenan los valores del estado interno** de cada objeto.
- Cada objeto tiene su propio estado interno, **independiente** de los otros objetos.
- También se les denomina **variables de instancia**, o **variables del objeto**.
- El acceso a los atributos está **restringido**. Sólo se permite su acceso y manipulación pública a través de los métodos.
  - Las variables del objeto deben tener un **acceso protegido** o **privado** (nombre con prefijo _ o __), no deben tener un acceso público.

- Los atributos del objeto se definen e inicializan en el **constructor** del objeto.
- Los atributos del objeto se destruyen cuando se destruye el objeto automáticamente.
- Desde el constructor y los métodos, se puede acceder y manipular los atributos del objeto a través del parámetro **self**, que referencia al objeto que se está construyendo, o al objeto sobre el que se aplican los métodos.

```
self.__cnt_positivo = 0                # Atributo cuenta de votos positivos
```

- También existen **variables de clase**, que tienen un valor compartido por todos los objetos de la clase.

Clases y objetos

Constructor (construye e inicializa cada objeto)

- Cuando se crea un objeto, se invoca a su **constructor**, para que realice las acciones necesarias para la construcción e inicialización del objeto.
  - Este constructor puede recibir **parámetros**, que proporcionan información adecuada para la inicialización del objeto.
  - El constructor debe realizar las siguientes **acciones**:
    - Debe definir las **variables del objeto**, con sus valores iniciales.
    - Debe invocar al **inicializador de la clase base** (superclase).
    - Puede invocar a otros métodos *privados* para realizar tareas adicionales.

Métodos (manipulan el estado de cada objeto)

- Los métodos definen la manipulación y comportamiento de los objetos de una clase.
- Los métodos son algoritmos definidos por la *Clase*, y se aplican sobre los objetos, manipulando el estado interno del objeto sobre el que se aplican.
- También se les denomina **métodos de instancia**, o **métodos del objeto**.

- También existen **métodos estáticos** y **métodos de clase**, que se aplican sobre clases.

- Un constructor o método **sólo puede tener los efectos** que sean especificados.

Clases y objetos

Definición de clases en Python

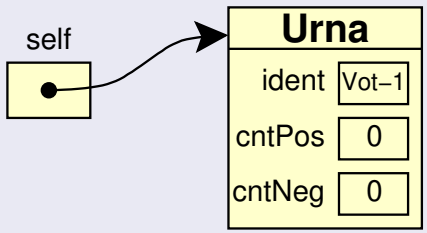
- Un **módulo** de Python puede contener definiciones de **constantes**, **funciones** y **clases**, que están relacionadas lógicamente entre sí.
- Para definir una **clase**, se utiliza la palabra reservada `class`, seguida del nombre de la clase y el símbolo *dos-puntos* (**:**), y a continuación se definen el constructor y los métodos de la clase, además de la documentación correspondiente, con una **indentación** adecuada.
  - El **parámetro especial** `self` debe ser el primer parámetro del constructor y de los métodos de la clase, y referencia al objeto sobre el que se aplica el método.

```
class Urna:
    """Abstracción sobre una urna de votación"""

    def __init__(self, ident: str, *args, **kwargs) -> None:
        """Constructor/inicializador de los objetos de la clase"""
        pass

    def introducir_voto(self, voto: bool) -> None:
        """Añade un voto positivo o negativo"""
        pass

    def resultado(self) -> bool:
        """Devuelve el resultado de la votación"""
        pass
```

Inicializador del objeto en Python: el método `__init__()`

- El constructor de un objeto está predefinido, e invoca automáticamente al método especial `__init__()` para inicializar cada objeto de cada clase.
- El método especial `__init__()` define la inicialización de los objetos de cada clase.
  - Los parámetros del método `__init__()`:
    - El primer parámetro debe ser el **parámetro especial** `self`, que referencia al propio objeto que está siendo inicializado.
    - Los parámetros proporcionan información para la inicialización del objeto.
    - Finalmente, los parámetros `*args, **kwargs` contienen la información que se debe pasar al inicializador de la clase base (superclase).
  - El cuerpo del método `__init__()`:
    - Debe definir e inicializar las **variables del objeto actual**, que está siendo inicializado, que son los atributos del objeto referenciado por el parámetro `self`.
    - Debe invocar al **inicializador de la clase base** (superclase), con `*args`, `**kwargs`.
    - Puede invocar a otros métodos *privados* para realizar tareas adicionales.

```python
class Urna:
    def __init__(self, ident: str, *args, **kwargs) -> None:
        """Inicializa una urna vacía, con la cuenta de votos a cero"""
        self.__ident = ident                # Atributo identificación
        self.__cnt_positivo = 0             # Atributo cuenta de votos positivos
        self.__cnt_negativo = 0             # Atributo cuenta de votos negativos
        super().__init__(*args, **kwargs) # Inicialización de la clase base
```

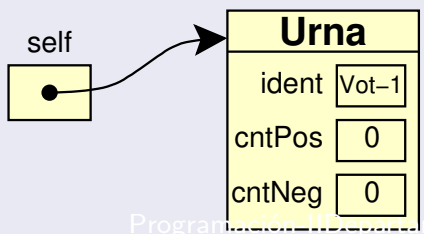


Definición de métodos en Python

- Los parámetros de los métodos:
  - El primer parámetro de un método debe ser el **parámetro especial** `self`, que referencia al propio objeto sobre el que se aplica el método.
  - Los siguientes parámetros proporcionan información adicional para realizar la manipulación del objeto sobre el que se aplica el método.
- El cuerpo de los métodos:
  - El método puede **acceder y manipular** a las **variables del objeto actual**, como atributos del objeto referenciado por el parámetro `self`, y también puede acceder a la información recibida a través de los **parámetros** adicionales.
  - El método puede **invocar a los métodos** disponibles sobre el propio objeto `self`, y a los métodos accesibles sobre los objetos recibidos como parámetros.

```python
def introducir_voto(self, voto: bool) -> None:
    """Añade un voto positivo o negativo"""
    if voto:
        self.__cnt_positivo += 1
    else:
        self.__cnt_negativo += 1
```

```python
def resultado(self) -> bool:
    """Devuelve el resultado de la votación"""
    return self.__cnt_positivo >= self.__cnt_negativo
```



Clases y objetos

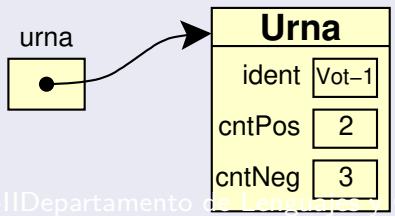
Creación de Objetos

- Para crear un objeto de una determinada clase, se invoca a su **constructor**, con el nombre de la clase, y entre paréntesis los **argumentos** necesarios para su construcción e inicialización.
  - Devuelve una referencia al objeto que ha sido creado e inicializado.

Invocación a Métodos

- Se puede invocar a un método, aplicándolo sobre un determinado objeto.
- La invocación a métodos puede llevar parámetros asociados, produce un resultado, y manipula el estado interno del objeto sobre el que se aplica.
- Los objetos responden a las invocaciones de los métodos dependiendo de su estado interno.
- Para invocar a un determinado método sobre un objeto, ese método debe estar definido por la clase a la que el objeto pertenece.

```
def main() -> None:
    urna = Urna("Vot-1")    # Creación de un objeto Urna
    urna.introducir_voto(True)
    urna.introducir_voto(False)
    print("Resultado:", urna.resultado())
```



Ejemplo: Clase Urna en Python

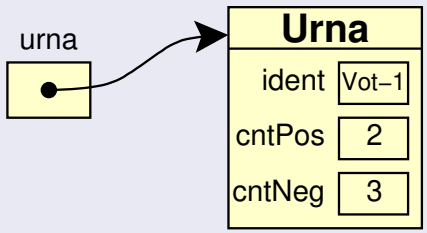
Abstracción Urna (POO)

```python
# Módulo: urna.py

class Urna:
    """Urna de votación. Permite introducir votos positivos y negativos"""

    def __init__(self, ident: str, *args, **kwargs) -> None:
        """Constructor del objeto"""
        self.__ident = ident                    # Atributo identificación de la urna
        self.__cnt_positivo = 0                 # Atributo cuenta de votos positivos
        self.__cnt_negativo = 0                 # Atributo cuenta de votos negativos
        super().__init__(*args, **kwargs)       # Inicializacion de la clase base

    def introducir_voto(self, voto: bool) -> None:
        """Añade un voto positivo o negativo"""
        if voto:
            self.__cnt_positivo += 1  # Incrementa la cuenta de votos positivos
        else:
            self.__cnt_negativo += 1  # Incrementa la cuenta de votos negativos

    def resultado(self) -> bool:
        """Devuelve el resultado de la votación"""
        return (self.__cnt_positivo >= self.__cnt_negativo)
```

Ejemplo: Clase Urna en Python

Abstracción Urna (POO). Continuación de módulo principal

```
import urna

def leer_votos() -> list[str]:
    """Lee de teclado una lista de votos (s,n)"""
    votos: list[str] = list()
    datos = input("Introduce votos (s,n): ")
    while len(datos) > 0:
        votos += [voto.lower() for voto in datos.split() if voto.lower() in ("s", "n")]
        datos = input("Introduce votos (s,n): ")
    return votos

def introducir_votos(urna1: urna.Urna, votos: list[str]) -> None:
    """Introduce en la urna los votos de la lista de votos"""
    for voto in votos:
        if voto in ("s", "n"):
            urna1.introducir_voto(voto == "s")

def main() -> None:
    urna1 = urna.Urna("Vot-1")    # Creación de un objeto Urna
    votos = leer_votos()
    introducir_votos(urna1, votos)
    print("Resultado:", urna1.resultado())

if __name__ == "__main__":
    main()
```

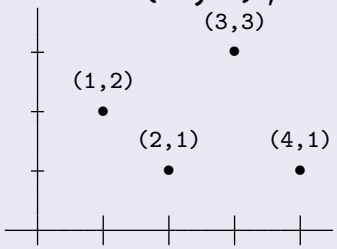


Ejemplo

Abstracción: Punto del plano cartesiano

Un *punto* representa una determinada posición en el plano cartesiano.

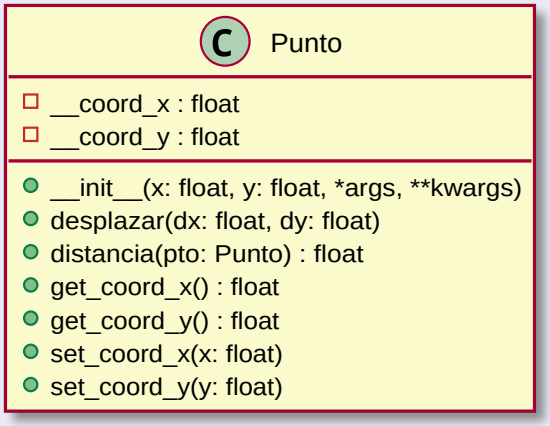
- Comportamiento de un *punto* (**métodos**):
  - Especificar el valor de sus coordenadas X e Y.
  - Consultar el valor de sus coordenadas X e Y.
  - Calcular la distancia que lo separa de otro objeto *punto*.
  - Desplazar según una distancia especificada en ambos ejes.
- Estado interno del *punto* (**atributos**):
  - El valor de la coordenada X (abscisa).
  - El valor de la coordenada Y (ordenada).
- Constructor del *punto* (**constructores**):
  - Un objeto *punto* se debe crear e inicializar con unas determinadas coordenadas X e Y.
- Podemos crear múltiples objetos de la *Clase Punto*:
  - `Punto(1,2)`, `Punto(2,1)`, `Punto(3,3)`, `Punto(4,1)`



Representación Gráfica de Clases y Objetos en UML

Representación gráfica: diagrama de clases en UML

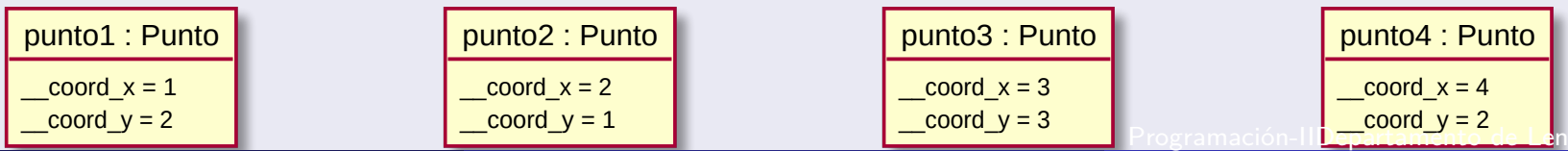
- Una clase se representa mediante una *caja* con tres compartimentos, conteniendo cada uno de ellos el nombre, atributos y métodos de la clase.



- En los métodos del diagrama no aparece el parámetro `self`.

Representación gráfica: diagrama de objetos en UML

- Un objeto se representa mediante una *caja* con dos compartimentos, conteniendo el primero el **nombre** del objeto y de la clase a la que pertenece, y el segundo los **valores** de los atributos del objeto.

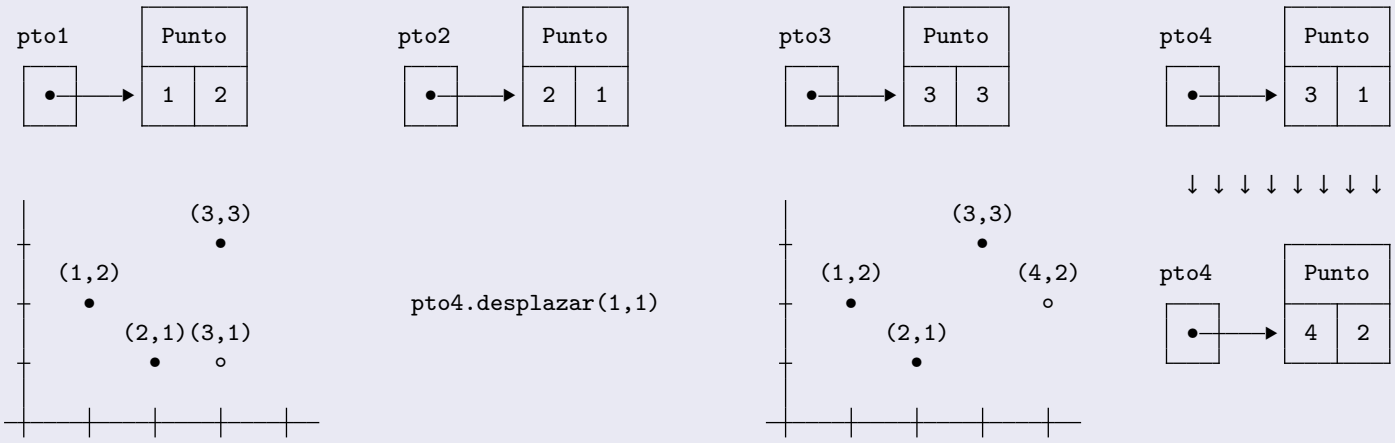


Ejemplo: Clase Punto

```python
import math
class Punto:
    """Punto del plano cartesiano"""
    def __init__(self, x: float = 0, y: float = 0, *args, **kwargs) -> None:
        """Constructor del objeto, a partir de dos coordenadas iniciales"""
        self.__coord_x = x
        self.__coord_y = y
        super().__init__(*args, **kwargs)
    def get_coord_x(self) -> float:
        """Devuelve el valor de la coordenada X (getter)"""
        return self.__coord_x
    def get_coord_y(self) -> float:
        """Devuelve el valor de la coordenada Y (getter)"""
        return self.__coord_y
    def set_coord_x(self, x: float) -> None:
        """Modifica el valor de la coordenada X (setter)"""
        self.__coord_x = x
    def set_coord_y(self, y: float) -> None:
        """Modifica el valor de la coordenada Y (setter)"""
        self.__coord_y = y
    def desplazar(self, dx: float, dy: float) -> None:
        """Desplaza el punto de la posición actual, según un desplazamiento (dx,dy)"""
        self.__coord_x += dx
        self.__coord_y += dy
    def distancia(self, pto: 'Punto') -> float:
        """Devuelve la distancia del punto actual a punto recibido como parámetro (Pitágoras)"""
        return math.sqrt((self.__coord_x - pto.get_coord_x())**2 + (self.__coord_y - pto.get_coord_y())**2)

# La anotación de tipo 'Punto' está entre comillas, ya que la clase Punto todavía no está definida.
# A partir de python version 3.11 se puede utilizar Self (Thonny tiene la version 3.10 de python)
```

Ejemplo: Clase Punto

```python
import punto

def main() -> None:
    pto1 = punto.Punto(1, 2)
    pto2 = punto.Punto(2, 1)
    pto3 = punto.Punto(3, 3)
    pto4 = punto.Punto(3, 1)
    pto4.desplazar(1, 1)
    d = pto1.distancia(pto3)

if __name__ == "__main__":
    main()
```



Métodos especiales (I)

- La definición de algunos **métodos especiales** permite especificar el comportamiento de estos objetos en varios usos habituales.
  - `def __init__(self, ..., *args, **kwargs) -> None`: define la **inicialización** de las variables del objeto. Se invoca desde el **constructor** del objeto. Debe invocar al inicializador de la clase base (**superclase**).
  - `def __repr__(self) -> str`: devuelve la **representacion textual alternativa** (**string**) del objeto, según un formato adecuado para representar el objeto.
    - Se invoca al utilizar `str(x)`, `repr(x)`, `print(x)`, etc. sobre el objeto.
  - `def __str__(self) -> str`: devuelve la **representacion textual** del objeto.
    - Si no se define `__str__`, entonces invoca automáticamente a `__repr__`.

```python
class Persona:
    def __init__(self, dni: str, nombre: str, fnac: tuple[int,str,int], *args, **kwargs) -> None:
        self.__dni = dni
        self.__nombre = nombre
        self.__fnac = fnac
        super().__init__(*args, **kwargs)

    def __repr__(self) -> str:
        return f"P({self.__dni}, {self.__nombre!r}, {self.__fnac})"
```

Métodos especiales (II)

- La definición de algunos **métodos especiales** permite especificar el comportamiento de estos objetos en varios usos habituales.
  - Las clases pueden redefinir los métodos `__eq__`, `__hash__` y `__lt__` para especificar el comportamiento de las comparaciones.

- La comparación de igualdad (`==`) de objetos devuelve `True` si los dos objetos referenciados tienen valores *"equivalentes"*. En otro caso, devuelve `False`.
  - En la implementación por defecto (proporcionada por la clase `object`), dos objetos son iguales si son el mismo objeto, es decir, si ambas referencias son iguales (comparación de identidad `is`).

- La comparación `(a == b)` se realiza **internamente** de la siguiente forma:
  1. Si la clase de `b` es subclase de la clase de `a`, entonces invoca `b.__eq__(a)`.
  2. Si *no-implementado*, entonces invoca `a.__eq__(b)`.
  3. Si *no-implementado* y no invocado anteriormente, entonces invoca `b.__eq__(a)`.
  4. Si *no-implementado*, entonces invoca `(a is b)`.

Métodos especiales (III)

- La definición de `__eq__`, `__hash__` y `__lt__` debe ser consistente, deben utilizar una **tupla** con los **mismos atributos de comparación** del objeto.
  - El **orden de los atributos** en la tupla es importante para `__lt__`.

- `def __eq__(self, other: object) -> bool:` compara si el **contenido** de ambos objetos es **igual** (*equivalente*). Se invoca en `==`, `!=`, `<`, `<=`, `>`, `>=`, en *set* y *dict*.
  - Si los dos parámetros referencian al mismo objeto (**is**), entonces devuelve `True`.
  - Si la clase del parámetro *other* es una subclase (**isinstance**) de la clase actual, entonces devuelve el resultado de comparar (`==`) las tuplas con los **atributos de comparación** de ambos objetos.
  - En otro caso, devuelve `NotImplemented`.

```
class Persona:
    def _atrib_comp(self) -> tuple:              # redefinible por las subclases
        return (self.__dni, self.__nombre)       # comparar por dni y nombre
    def __eq__(self, other: object) -> bool:
        if self is other:
            ok = True
        elif isinstance(other, Persona):
            ok = (self._atrib_comp() == other._atrib_comp())
        else:
            ok = NotImplemented
        return ok
```

Métodos especiales (IV)

- La definición de `__eq__`, `__hash__` y `__lt__` debe ser consistente, deben utilizar una **tupla** con los **mismos atributos de comparación** del objeto.
  - El **orden de los atributos** en la tupla es importante para `__lt__`.

- `def __hash__(self) -> int:` devuelve un **código numérico** para poder utilizar el objeto en *conjuntos* y como *claves de diccionarios*.
  - Si dos objetos son **iguales** (`a == b`), entonces el **código hash** de ambos objetos debe ser **igual** (`hash(a) == hash(b)`).
  - Es posible que dos objetos distintos generen el mismo código hash (**colisión**).
  - Los **atributos utilizados** del objeto deben ser **inmutables**.
  - Devuelve el resultado de invocar a la función **hash** sobre la tupla con los **atributos de comparación** del objeto.

```
class Persona:

    def _atrib_comp(self) -> tuple:              # redefinible por las subclases
        return (self.__dni, self.__nombre)       # dni y nombre son INMUTABLES

    def __hash__(self) -> int:
        return hash(self._atrib_comp())
```

Métodos especiales (V)

- La definición de `__eq__`, `__hash__` y `__lt__` debe ser consistente, deben utilizar una **tupla** con los **mismos atributos de comparación** del objeto.
  - El **orden de los atributos** en la tupla es importante para `__lt__`.

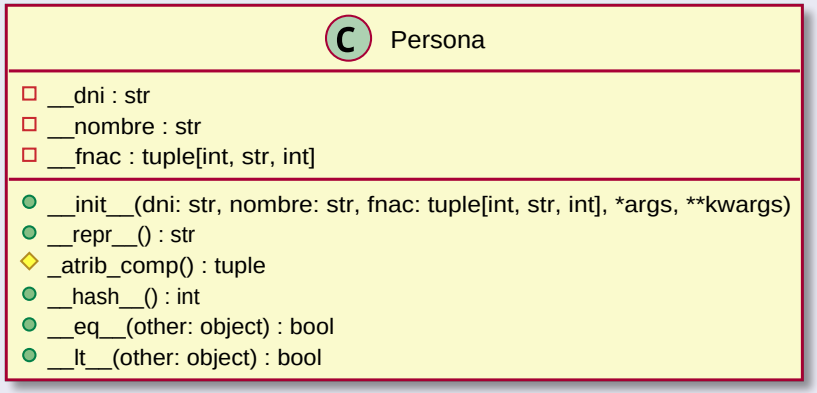
- `def __lt__(self, other: object) -> bool:` compara si el **contenido** del objeto actual **es menor** que el **contenido** del otro objeto. Se invoca en `<`, `<=`, `>`, `>=`, y *sort*.
  - Si la clase del parámetro *other* es una subclase (**isinstance**) de la clase actual, entonces devuelve el resultado de comparar (`<`) las tuplas con los **atributos de comparación** de ambos objetos.
  - En otro caso, devuelve `NotImplemented`.
  - El decorador `@functools.total_ordering` genera el resto de comparadores.

```
import functools
@functools.total_ordering
class Persona:
    def _atrib_comp(self) -> tuple:              # redefinible por las subclases
        return (self.__dni, self.__nombre)       # comparar por dni y nombre
    def __lt__(self, other: object) -> bool:
        if isinstance(other, Persona):
            ok = (self._atrib_comp() < other._atrib_comp())
        else:
            ok = NotImplemented
        return ok
```

Métodos especiales. Ejemplo Persona

Abstracción Persona

- Un objeto de la clase *Persona* representa una persona, que contiene la siguiente información: el DNI de la persona, su nombre y su fecha de nacimiento.
  - El constructor permitirá construir objetos de esta clase, recibiendo la información anteriormente especificada.
  - Los métodos permitirán obtener la representación textual de los objetos, así como compararlos y utilizarlos en conjuntos y como claves de diccionarios.
    - Representación textual: `P(dni, 'nombre', (dia, 'mes', año))`
    - Comparación (y hash) basada en DNI y nombre (en ese orden). En este ejemplo, la comparación de nombres no diferencia mayúsculas de minúsculas.



- En los métodos del diagrama no aparece el parámetro `self`.

Métodos especiales. Ejemplo Persona

```python
import functools

@functools.total_ordering
class Persona:
    """Datos de una persona.
    Representación textual: P(dni, 'nombre', (dia, 'mes', año))
    Comparación (y hash) basada en DNI y nombre (en ese orden).
    Comparación de nombres no diferencia mayúsculas de minúsculas """

    def __init__(self, dni: str, nombre: str, fnac: tuple[int,str,int], *args, **kwargs) -> None:
        self.__dni = dni
        self.__nombre = nombre
        self.__fnac = fnac
        super().__init__(*args, **kwargs)

    # @override
    def __repr__(self) -> str:
        return f"P({self.__dni}, {self.__nombre!r}, {self.__fnac})"

    # def _atrib_comp(self) -> tuple:                    # redefinible por las subclases
    #     """Devuelve la tupla con los atributos de comparación.
    #     Esta comparación de nombres sí diferencia mayúsculas de minúsculas """
    #     return (self.__dni, self.__nombre)             # dni y nombre son INMUTABLES

    def _atrib_comp(self) -> tuple:                     # redefinible por las subclases
        """Devuelve la tupla con los atributos de comparación.
        Esta comparación de nombres no diferencia mayúsculas de minúsculas """
        return (self.__dni, self.__nombre.lower())      # dni y nombre son INMUTABLES
```

▲ ▲ ▲

Métodos especiales. Ejemplo Persona

```
    # @override
    def __hash__(self) -> int:
        return hash(self._atrib_comp())

    # @override
    def __eq__(self, other: object) -> bool:
        if self is other:
            ok = True
        elif isinstance(other, Persona):
            ok = (self._atrib_comp() == other._atrib_comp())
        else:
            ok = NotImplemented
        return ok

    # @override
    def __lt__(self, other: object) -> bool:
        if isinstance(other, Persona):
            ok = (self._atrib_comp() < other._atrib_comp())
        else:
            ok = NotImplemented
        return ok
```

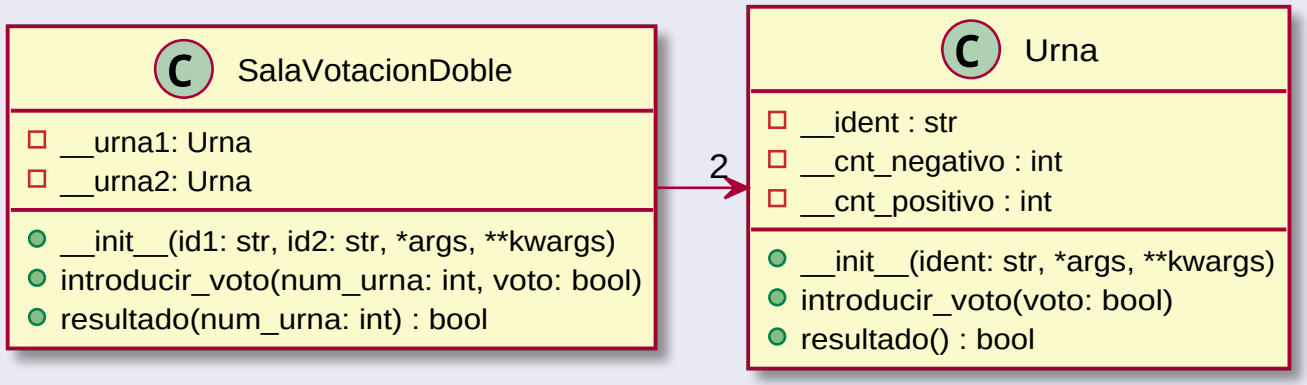
Métodos especiales. Ejemplo Persona

En esta versión, la comparación de nombres **no** diferencia entre mayúsculas y minúsculas.

```
import persona

def main() -> None:
    a1 = persona.Persona("123X", "Pepe Luis", (20, "Julio", 2000))
    a2 = persona.Persona("123X", "pepe luis", (21, "Agosto", 2000))
    a3 = persona.Persona("123Z", "pepe luis", (22, "Abril", 2000))

    print("Persona:", a1)          # Persona: P(123X, 'Pepe Luis', (20, 'Julio', 2000))
    print("Persona:", a2)          # Persona: P(123X, 'pepe luis', (21, 'Agosto', 2000))
    print("Persona:", a3)          # Persona: P(123Z, 'pepe luis', (22, 'Abril', 2000))

    print("Hash:", hash(a1))      # Hash: -8904233186154980011   # conjuntos y diccionarios
    print("Hash:", hash(a2))      # Hash: -8904233186154980011   # conjuntos y diccionarios
    print("Hash:", hash(a3))      # Hash: 2034364357619238399    # conjuntos y diccionarios

    print("Igual:", a1 == a2)     # Igual: True          # comparaciones, conjuntos y diccionarios
    print("Igual:", a1 == a3)     # Igual: False         # comparaciones, conjuntos y diccionarios
    print("Igual:", a2 == a3)     # Igual: False         # comparaciones, conjuntos y diccionarios

    print("Menor:", a1 < a2)      # Menor: False         # comparaciones
    print("Menor:", a1 < a3)      # Menor: True          # comparaciones
    print("Menor:", a2 < a3)      # Menor: True          # comparaciones

if __name__ == "__main__":
    main()
```

Composición

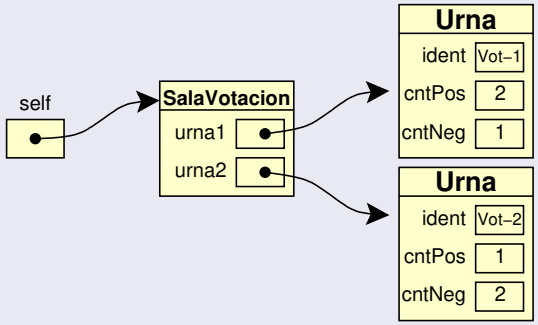
- Permite la definición de *nuevas clases* a partir de otras clases ya definidas.
- Representa una *relación* en la cual un objeto **tiene** o **está compuesto** por otros objetos.
  - Por ejemplo, una `SalaVotacionDoble` **tiene** dos `Urnas`, la *urna1* y la *urna2*.
  - El objeto *contenido* forma parte de los **atributos** del objeto *contenedor*.
  - La composición se puede expresar en UML mediante una línea continua con flecha de **punta abierta** entre la *clase poseedora* y la *clase poseída*.
  - Esta línea continua con flecha de punta abierta se puede etiquetar con el nombre del atributo, su nivel de acceso, y opcionalmente su **multiplicidad** (donde el **asterisco** indica múltiples elementos).



- En los métodos del diagrama no aparece el parámetro `self`.

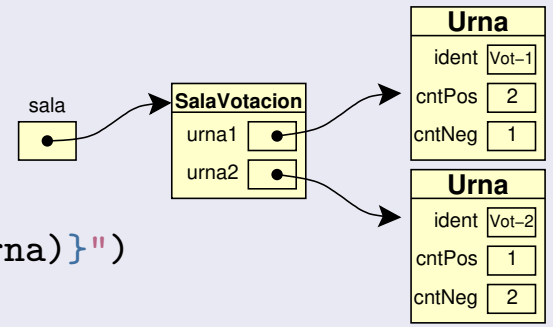
Ejemplo: Clase SalaVotacionDoble

```python
# Módulo: salavotaciondoble.py
import urna

class SalaVotacionDoble:
    """Abstracción sobre una sala de votación con dos urnas"""

    def __init__(self, id1: str, id2: str, *args, **kwargs) -> None:
        self.__urna1 = urna.Urna(id1)
        self.__urna2 = urna.Urna(id2)
        super().__init__(*args, **kwargs)

    def introducir_voto(self, num_urna: int, voto: bool) -> None:
        """Añade un voto positivo o negativo a urna"""
        if (num_urna == 1):
            self.__urna1.introducir_voto(voto)  # Añade un voto a la urna
        elif (num_urna == 2):
            self.__urna2.introducir_voto(voto)  # Añade un voto a la urna
        else:
            raise ValueError(f"Número de urna erróneo {num_urna}") # Notifica del Error

    def resultado(self, num_urna: int) -> bool:
        """Devuelve el resultado de la votación en una urna"""
        if (num_urna == 1):
            res = self.__urna1.resultado()      # resultado de la urna
        elif (num_urna == 2):
            res = self.__urna2.resultado()      # resultado de la urna
        else:
            raise ValueError(f"Número de urna erróneo {num_urna}") # Notifica del Error
        return res
```



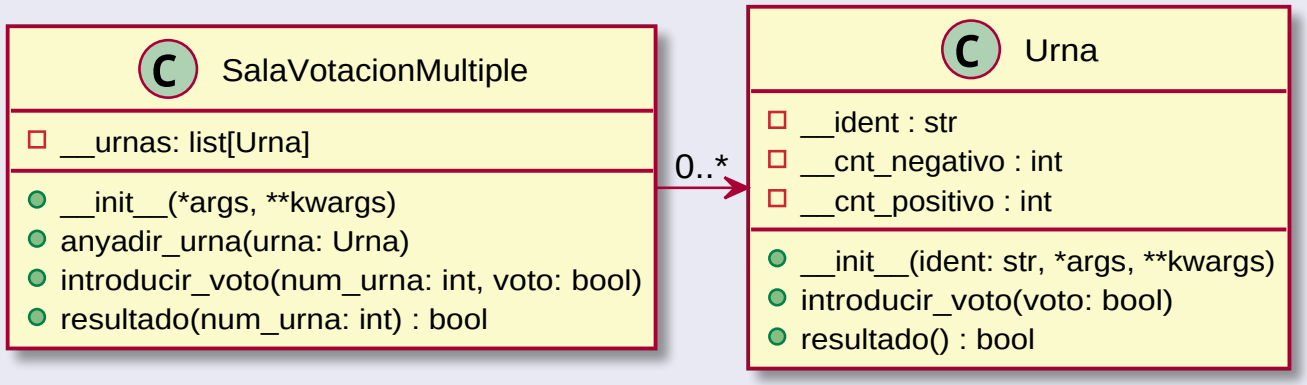
Ejemplo: Prueba de SalaVotacionDoble

```python
import salavotaciondoble as svd

def introducir_votos(sala: svd.SalaVotacionDoble, num_urna: int, votos: list[str]) -> None:
    """Introduce en la urna de la sala los votos de la lista de votos"""
    for voto in votos:
        if voto in ("s", "n"):
            sala.introducir_voto(num_urna, voto == "s")

def main() -> None:
    sala = svd.SalaVotacionDoble("Vot-1", "Vot-2")  # Creación de objeto SalaVotacionDoble
    for num_urna in range(1, 3):
        print(f"Introduce votos para la Urna {num_urna}")
        votos = urna.leer_votos()
        introducir_votos(sala, num_urna, votos)
    for num_urna in range(1, 3):
        print(f"Resultado Urna {num_urna}: {sala.resultado(num_urna)}")

if __name__ == "__main__":
    main()
```



Composición

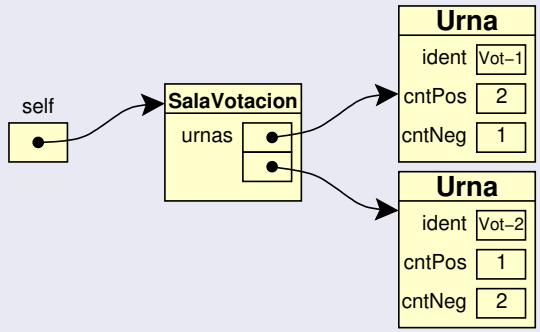
- Permite la definición de *nuevas clases* a partir de otras clases ya definidas.
- Representa una *relación* en la cual un objeto **tiene** o **está compuesto** por otros objetos.
  - Por ejemplo, una `SalaVotacionMultiple` **tiene** múltiples `Urnas`, en una lista.
  - El objeto *contenido* forma parte de los **atributos** del objeto *contenedor*.
  - La composición se puede expresar en UML mediante una línea continua con flecha de **punta abierta** entre la *clase poseedora* y la *clase poseída*.
  - Esta línea continua con flecha de punta abierta se puede etiquetar con el nombre del atributo, su nivel de acceso, y opcionalmente su **multiplicidad** (donde el **asterisco** indica múltiples elementos).



- En los métodos del diagrama no aparece el parámetro `self`.

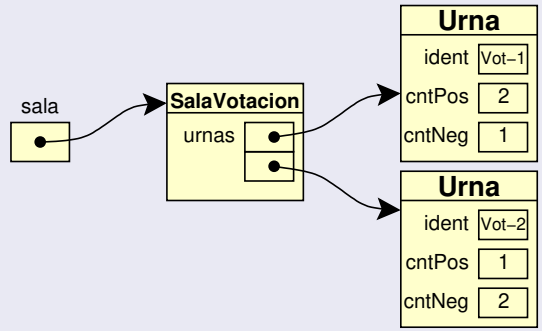
Ejemplo: Clase SalaVotacionMultiple

```python
# Módulo: salavotacionmultiple.py
import urna

class SalaVotacionMultiple:
    """Abstracción sobre una sala de votación con múltiples urnas"""

    def __init__(self, *args, **kwargs) -> None:
        self.__urnas: list[urna.Urna] = list()
        super().__init__(*args, **kwargs)

    def anyadir_urna(self, urn: urna.Urna) -> None:
        self.__urnas.append(urn)

    def introducir_voto(self, num_urna: int, voto: bool) -> None:
        """Añade un voto positivo o negativo a urna"""
        if ((num_urna < 0) or (num_urna >= len(self.__urnas))):
            raise ValueError(f"Número de urna erróneo {num_urna}") # Notifica del Error
        self.__urnas[num_urna].introducir_voto(voto)  # Añade un voto a la urna

    def resultado(self, num_urna: int) -> bool:
        """Devuelve el resultado de la votación en una urna"""
        if ((num_urna < 0) or (num_urna >= len(self.__urnas))):
            raise ValueError(f"Número de urna erróneo {num_urna}") # Notifica del Error
        return self.__urnas[num_urna].resultado()      # resultado de la urna
```



Ejemplo: Prueba de SalaVotacionMultiple

```python
import urna
import  salavotacionmultiple as svm

def introducir_votos(sala: svm.SalaVotacionMultiple, num_urna: int, votos: list[str]) -> None:
    """Introduce en la urna de la sala los votos de la lista de votos"""
    for voto in votos:
        if voto in ("s", "n"):
            sala.introducir_voto(num_urna, voto == "s")

def main() -> None:
    num_urnas = int(input("Introduce el número de urnas: "))
    sala = svm.SalaVotacionMultiple(num_urnas)  # Creación de objeto SalaVotacionMultiple
    for num_urna in range(num_urnas):
        sala.anyadir_urna(urna.Urna(f"Votación-{num_urna}"))

    for num_urna in range(num_urnas):
        print(f"Introduce votos para la Urna {num_urna}")
        votos = urna.leer_votos()
        introducir_votos(sala, num_urna, votos)

    for num_urna in range(num_urnas):
        print(f"Resultado Urna {num_urna}: {sala.resultado(num_urna)}")

if __name__ == "__main__":
    main()
```

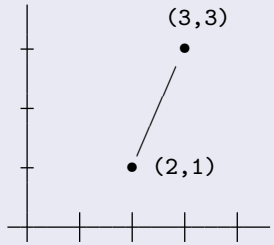


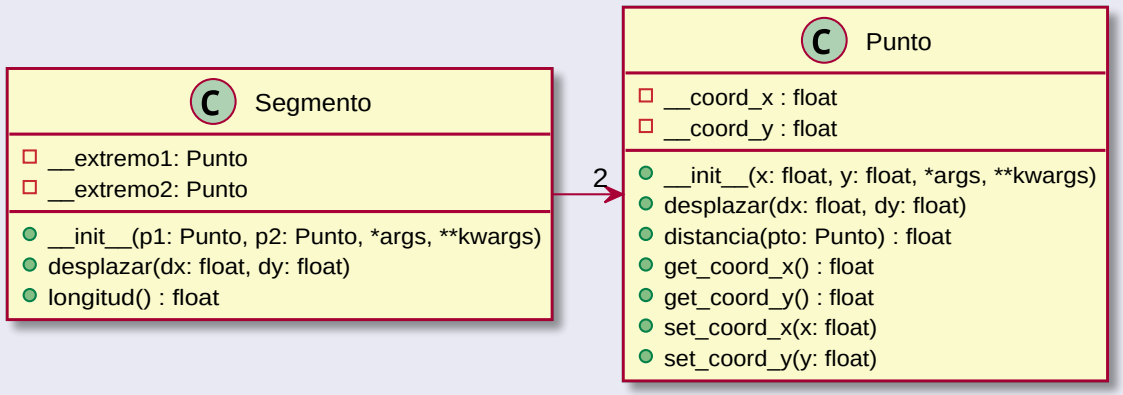
Ejemplo: Clase Segmento

Abstracción: Segmento del plano cartesiano

Un *segmento* es un fragmento de una recta que está comprendido entre dos *puntos* en el plano cartesiano.

- Comportamiento de un *segmento*:
  - Especificar el valor de las coordenadas X e Y de ambos puntos.
  - Calcular la longitud del segmento.
  - Desplazar el segmento según una distancia especificada en ambos ejes.
- Estado interno del *segmento*:
  - Ambos puntos (extremos) que delimitan el segmento.
- Podemos crear múltiples objetos de la *Clase Segmento*.

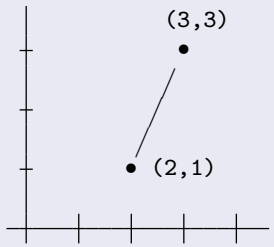


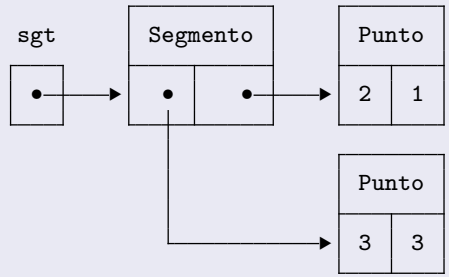


- En los métodos del diagrama no aparece el parámetro `self`.

Ejemplo: Clase Segmento

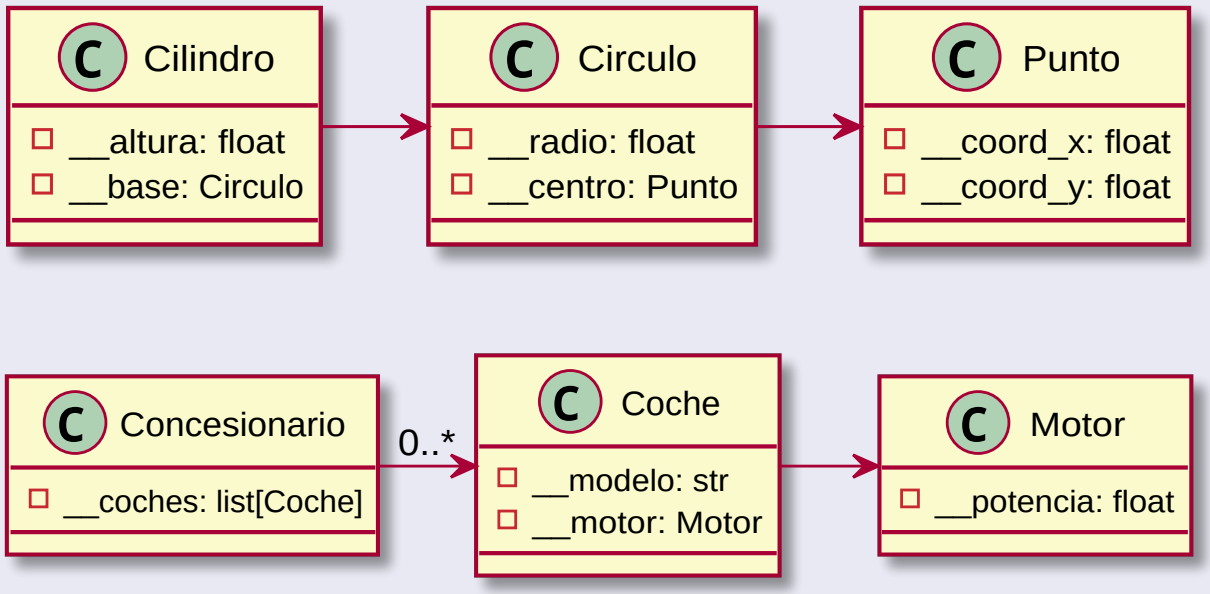
```python
# Módulo: segmento.py
import punto

class Segmento:
    """Segmento del plano cartesiano, delimitado por dos puntos"""

    def __init__(self, p1: punto.Punto, p2: punto.Punto, *args, **kwargs) -> None:
        self.__extremo1 = p1
        self.__extremo2 = p2
        super().__init__(*args, **kwargs)

    def desplazar(self, dx: float, dy: float) -> None:
        self.__extremo1.desplazar(dx, dy)   # invocación a métodos de la clase Punto
        self.__extremo2.desplazar(dx, dy)   # invocación a métodos de la clase Punto

    def longitud(self) -> float:
        return self.__extremo1.distancia(self.__extremo2) # invocación a métodos de Punto

def main() -> None:
    segmento = Segmento(punto.Punto(2, 1), punto.Punto(3, 3))
    print("Longitud:", segmento.longitud())

if __name__ == "__main__":
    main()
```





Composición

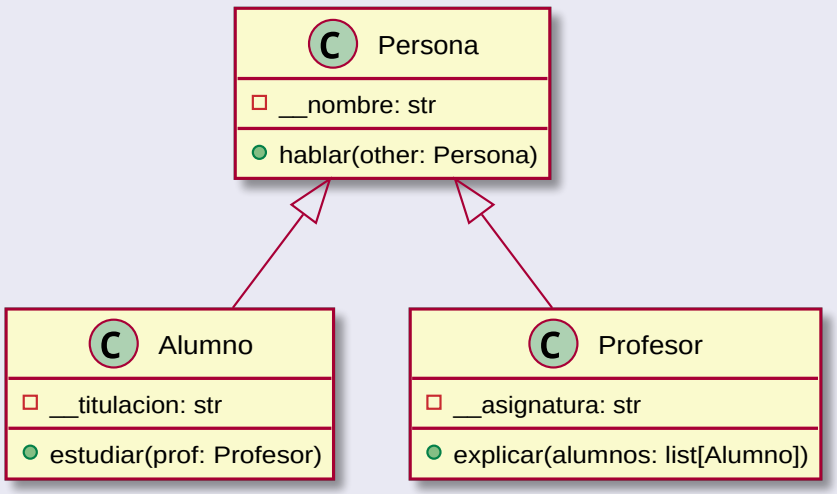
- Otros ejemplos de composición.
  - El objeto *contenido* forma parte de los **atributos** del objeto *contenedor*.
  - La composición se puede expresar en UML mediante una línea continua con flecha de **punta abierta** entre la *clase poseedora* y la *clase poseída*.
  - Esta línea continua con flecha de punta abierta se puede etiquetar con el nombre del atributo, su nivel de acceso, y opcionalmente su **multiplicidad** (donde el **asterisco** indica múltiples elementos).

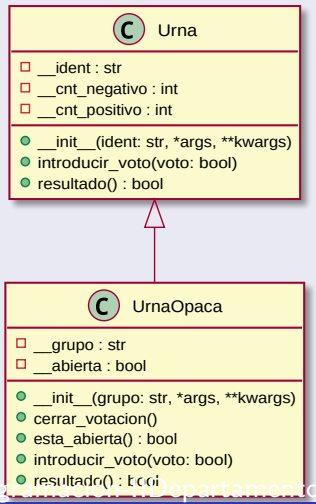


Herencia y redefinición del comportamiento

Herencia

- Representa una *relación* en la cual una Clase **ES UNA** *especialización* y/o *extensión* de otra Clase.
- Permite definir una nueva **subclase** (o clase derivada) como una especialización o extensión de una **superclase** (o clase base) más general.
  - La *subclase* **hereda** tanto los **atributos** como los **métodos** definidos por la *superclase* (reusabilidad del código). No es necesario que los vuelva a definir.
  - La subclase puede **añadir** *nuevos atributos* y *nuevos métodos* (extensibilidad).
  - La subclase puede **redefinir** métodos de la superclase (especialización).
- La herencia se expresa en UML mediante una línea continua desde la *subclase* con un **triángulo hueco** en el extremo de la *superclase*.
- Por ejemplo, un *Alumno* **es una** *Persona*, y un *Profesor* también **es una** *Persona*.

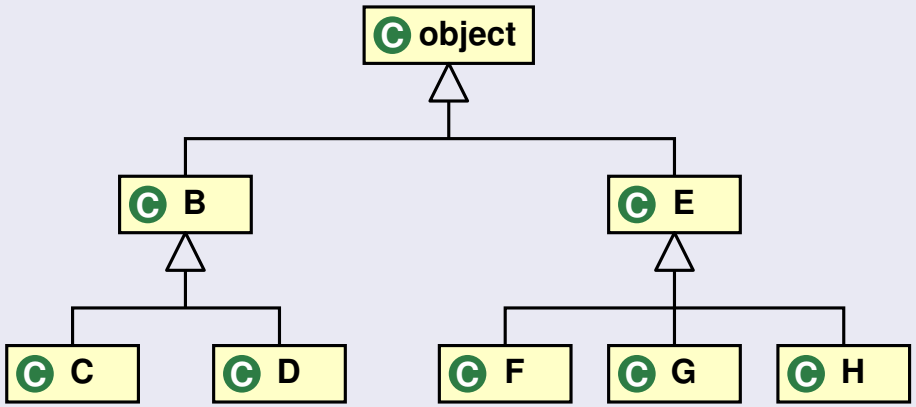




Herencia y redefinición del comportamiento

Herencia

- Permite definir **jerarquías** de clases (ascendientes, y descendientes).
- La relación de herencia es **transitiva**, si **C** hereda de **B** y **B** hereda de **object**, entonces **C** también hereda de **object**. Todos los objetos heredan de **object**.
  - En Python la clase **object** es la base de toda la jerarquía de objetos.
  - Todas las clases son **herederas directas o indirectas** de la clase **object**, que define los comportamientos comunes y básicos de cualquier objeto.
  - Si una clase no deriva de ninguna otra clase, entonces deriva implícitamente de la clase **object**.

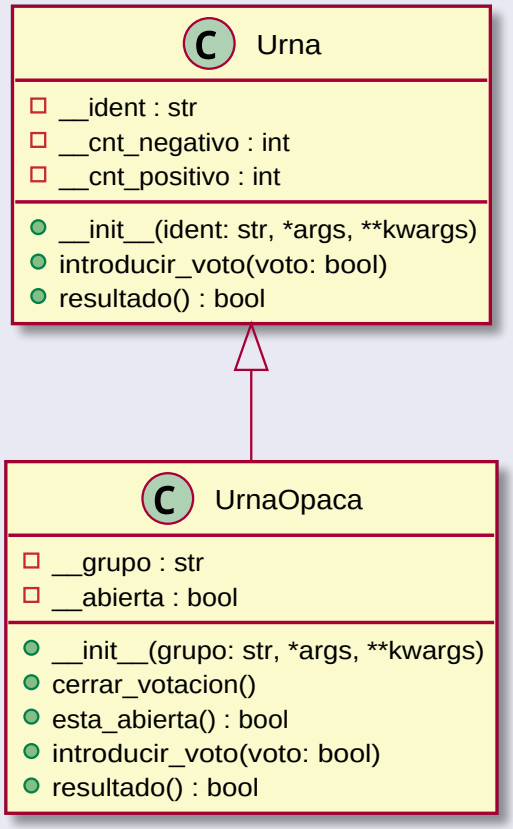


- En la **herencia simple**, una subclase hereda de una única superclase.
- En la **herencia múltiple**, una subclase hereda de varias superclases.

Herencia y redefinición del comportamiento

Ejemplo: Abstracción urna opaca

- Una `UrnaOpaca` **es una** `Urna` en la cual sólo se puede consultar el resultado al final de la votación.
- Por lo tanto, se podrán introducir votos mientras la **votación está abierta**, pero cuando se consulte el resultado, se **cerrará la votación** y no se permitirá introducir nuevos votos.
- La `UrnaOpaca` **hereda** las características de `Urna` (atributos y métodos), y la **extiende añadiendo** dos nuevos atributos, el grupo de la urna, y el estado de la votación (si la votación está abierta o cerrada). Además, permite consultar el estado de la votación, así como cerrar la votación.
- Así mismo, la `UrnaOpaca` también **especializa** la `Urna`, **redefiniendo** su comportamiento para evitar que se introduzcan votos cuando la votación está cerrada.
- Todas las `UrnasOpacas` **son** `Urnas`, pero **no** todas las `Urnas` son `UrnasOpacas`.
- En los métodos del diagrama no aparece el parámetro `self`.



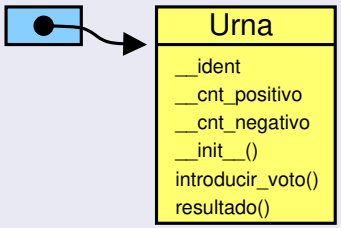
Herencia y redefinición del comportamiento

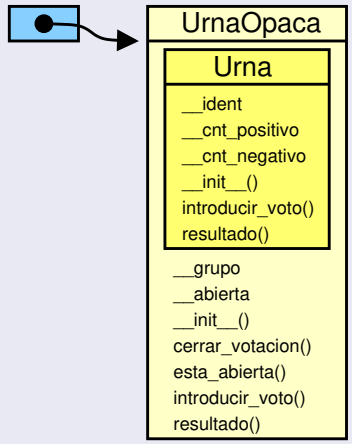
Herencia y extensibilidad

- Para definir una nueva `SubClase`, derivada de una `SuperClase`, se especifica el nombre de la superclase entre paréntesis.

```
class SubClase(SuperClase):
    ...
```

- La *subclase* **hereda** tanto los **atributos** como los **métodos** definidos por la *superclase* (reusabilidad del código). No es necesario que los vuelva a definir.
  - Un objeto de la **subclase** tiene **incluido internamente** a un objeto de la superclase (con sus atributos y métodos), y así sucesivamente en toda la línea jerárquica de herencia (hasta **object**).

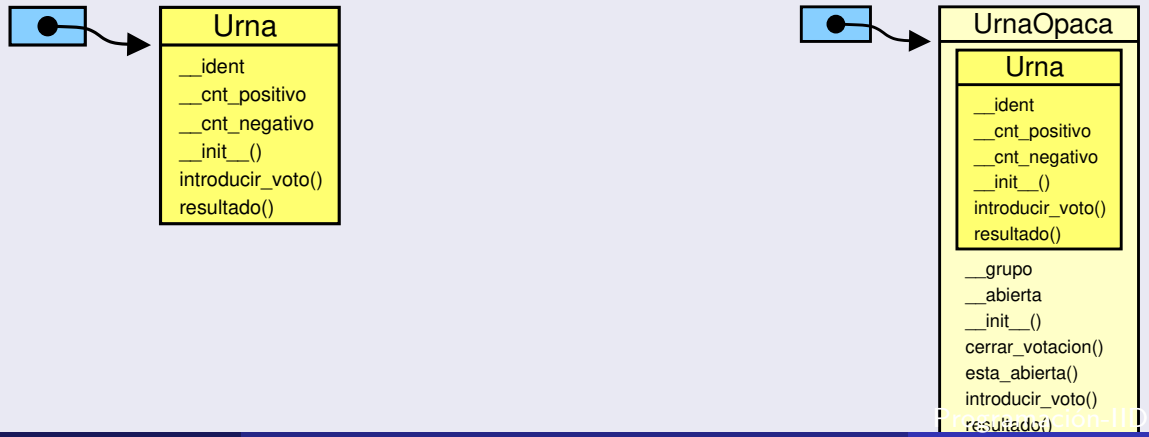




- La subclase puede **añadir** *nuevos atributos* y *nuevos métodos* (extensibilidad).
  - La clase *UrnaOpaca* añade los atributos privados `__grupo` y `__abierta`, y los métodos para *cerrar la votación*, comprobar si la votación *está abierta*.

El método `__init__()` en la herencia

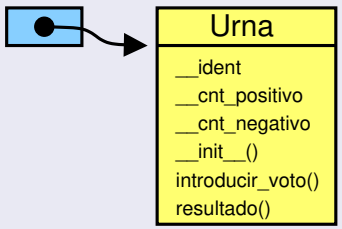
- Cuando se construye un objeto de una determinada clase, se invoca automáticamente al método `__init__(...)` de esa clase, con los argumentos necesarios.
  - El método `__init__(..., *args, **kwargs)` debe **definir e inicializar los atributos** de los objetos de esa clase, utilizando si es necesario los valores de los parámetros.
  - Además, debe **invocar** al método `super().__init__(*args, **kwargs)`, con el resto de argumentos que no ha utilizado, para que se pueda **inicializar** adecuadamente el **objeto de la superclase**, que está *incluido internamente*, y así sucesivamente hasta inicializar todos los objetos incluidos en la línea jerárquica de herencia.

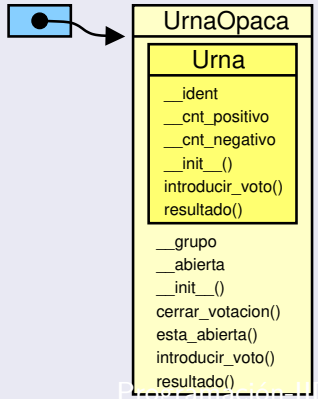


El método __init__() en la herencia

```python
class Urna:
    def __init__(self, ident: str, *args, **kwargs) -> None:
        """Inicializa la urna con la cuenta de votos a cero"""
        self.__ident = ident                        # Atributo identificacion de la urna
        self.__cnt_positivo = 0                     # Atributo cuenta de votos positivos
        self.__cnt_negativo = 0                     # Atributo cuenta de votos negativos
        super().__init__(*args, **kwargs)           # Inicializacion de la clase base
```

```python
class UrnaOpaca(Urna):
    def __init__(self, grupo: str, *args, **kwargs) -> None:
        """Inicializa la urna opaca con la votación abierta"""
        self.__grupo = grupo                        # Atributo grupo de la urna
        self.__abierta = True                       # Atributo estado de la votación
        super().__init__(*args, **kwargs)           # Inicializacion de la clase base (Urna)
```





Alternativa para `__init__()` en la herencia

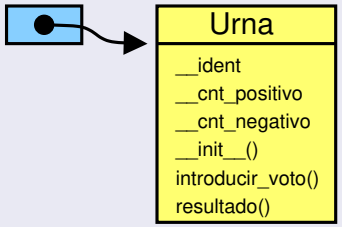
- La siguiente forma **alternativa** de definir los métodos `__init__()` en jerarquías de herencia **no esta recomendada**, ya que **no se adapta** adecuadamente a jerarquías de clases con **herencia múltiple**.

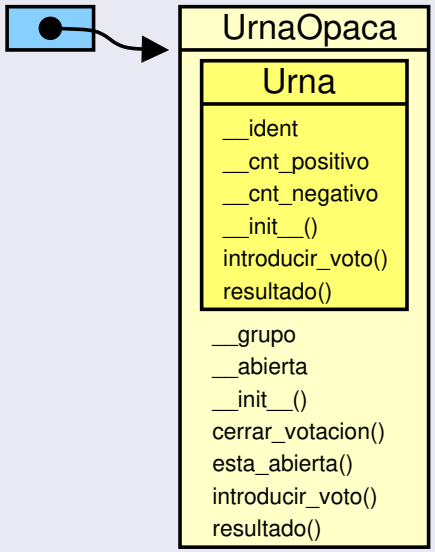
```
class Urna:
    def __init__(self, ident: str) -> None:
        """Inicializa la urna con la cuenta de votos a cero"""
        self.__ident = ident                    # Atributo identificacion de la urna
        self.__cnt_positivo = 0                 # Atributo cuenta de votos positivos
        self.__cnt_negativo = 0                 # Atributo cuenta de votos negativos
        super().__init__()                      # Inicializacion de la clase base
```

```
class UrnaOpaca(Urna):
    def __init__(self, grupo: str, ident: str) -> None:
        """Inicializa la urna opaca con la votación abierta"""
        self.__grupo = grupo                    # Atributo grupo de la urna
        self.__abierta = True                   # Atributo estado de la votación
        super().__init__(ident)                 # Inicializacion de la clase base (Urna)
```

Herencia, acceso a atributos e invocación a métodos

- Los métodos de la *subclase* pueden **acceder** (utilizando **self**) a sus **propios atributos**, y a los **atributos públicos o protegidos** (cuyos nombres no comienzan por __) de la **superclase**, pero no pueden acceder a los atributos privados (cuyos nombres comienzan por __) de la superclase.
- Los métodos de la *subclase* pueden **invocar** (utilizando **self**) a **otros métodos** de la propia clase, y a los **métodos públicos o protegidos** (cuyos nombres no comienzan por __) de la **superclase**, pero no pueden invocar a los métodos privados (cuyos nombres comienzan por __) de la superclase.





Herencia y extensión del comportamiento (extensibilidad)

- La subclase **hereda** los atributos y métodos de la superclase.
- Además, la subclase puede **añadir** *nuevos atributos* y *nuevos métodos* (extensión).
  - La clase *UrnaOpaca* añade el método para *comprobar* si el estado de la votación está abierto.
  - La clase *UrnaOpaca* añade el método para *cerrar* la votación.

```
class UrnaOpaca(Urna):
    def esta_abierta(self) -> bool:
        return self.__abierta

    def cerrar_votacion(self) -> None:
        self.__abierta = False
```

Herencia y redefinición del comportamiento (especialización)

- La subclase **hereda** los atributos y métodos de la superclase. Aquellos métodos que no sean redefinidos, mantienen su comportamiento heredado.
- Además, la subclase puede **redefinir** métodos de la superclase (especialización), para modificar su comportamiento.
- Existen dos alternativas en la redefinición de métodos en la subclase:
  1. El método de la subclase tiene un comportamiento nuevo y diferente, que no depende ni utiliza el comportamiento del método de la superclase.
     - En este caso, este método se programa directamente, sin tener en cuenta el método de la superclase.
  2. El método de la subclase tiene un comportamiento modificado, pero utiliza de alguna forma el comportamiento del método de la superclase.
     - En este caso, este método debe invocar, en algún momento, al método de la superclase (`super().metodo(...)`), además de realizar su propia tarea.
     - Nótese que no se puede invocar, utilizando **self**, al método de la superclase que se está redefiniendo, ya que sería una **invocación recursiva a sí mismo**.
- El decorador `@override`, del módulo `typing` (disponible desde la versión 3.12 de Python) comprueba que en la superclase existe un método con el mismo nombre, por lo que permite detectar errores de nombres en estos casos.

Herencia y redefinición del comportamiento (especialización)

```python
class Urna:
    def introducir_voto(self, voto: bool) -> None:
        """Añade un voto positivo o negativo"""
        if voto:
            self.__cnt_positivo += 1  # Incrementa la cuenta de votos positivos
        else:
            self.__cnt_negativo += 1  # Incrementa la cuenta de votos negativos

    def resultado(self) -> bool:
        """Devuelve el resultado de la votación"""
        return (self.__cnt_positivo >= self.__cnt_negativo)
```

```python
class UrnaOpaca(Urna):
    # @override # a partir de python version 3.12
    def introducir_voto(self, voto: bool) -> None:
        if not self.__abierta:
            raise RuntimeError("Votación cerrada")
        super().introducir_voto(voto)

    # @override # a partir de python version 3.12
    def resultado(self) -> bool:
        self.__abierta = False
        return super().resultado()
```

Herencia y MRO

- El **Orden de Resolución de Métodos** (**MRO**) (*Method Resolution Order*) determina la línea de invocación a los métodos de las superclases.
  - Para cada clase, el *MRO* se puede consultar:

```
print(Urna.__mro__)           # (Urna, object)
print(UrnaOpaca.__mro__)      # (UrnaOpaca, Urna, object)
print([c.__name__ for c in UrnaOpaca.__mro__])
```

- En **herencia simple**, el **MRO** coincide con la **línea jerárquica de herencia**.

Herencia y creación de Objetos

- Para crear un objeto de una determinada clase, se invoca a su **constructor**, con el nombre de la clase, y entre paréntesis los **argumentos** necesarios para su construcción e inicialización. Dos alternativas para pasar los argumentos:
  1. Si los **nombres** de los parámetros son **distintos** en todos los constructores de la jerarquía de herencia de la clase, entonces se pueden proporcionar los **argumentos con nombre**, para todos los constructores, en cualquier orden.
  2. Como conocemos el MRO de la clase del objeto que estamos creando, entonces se pueden proporcionar los **argumentos posicionales**, para todos los constructores, según el **orden del MRO** de la clase del objeto que estamos creando.

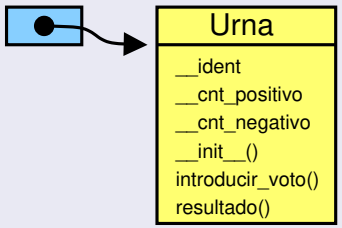
Herencia y creación de Objetos

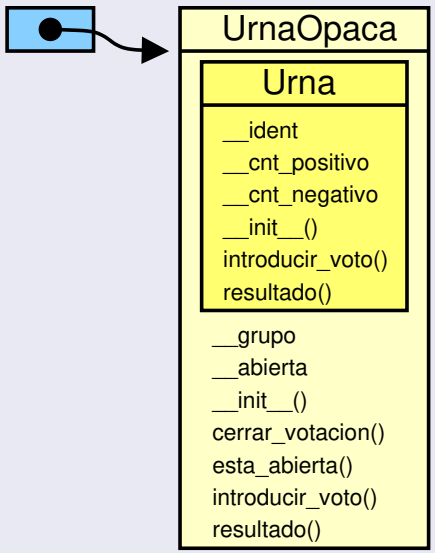
- Si los nombres de los parámetros en los constructores son diferentes, entonces se pueden especificar los argumentos con nombres (sin importar el orden).

```
def main() -> None:
    urna = Urna(ident="Elección-1")                          # Args con nombre
    urna_opaca = UrnaOpaca(ident="Elección-2", grupo="Grupo")  # Args con nombre
```

- Como conocemos el MRO de las clases, podemos proporcionar los argumentos en el **mismo orden** de los parámetros de los constructores, según el MRO de la clase.

```
def main() -> None:
    urna = Urna("Elección-1")                          # MRO(Urna, object)
    urna_opaca = UrnaOpaca("Grupo", "Elección-2") # MRO(UrnaOpaca, Urna, object)
```





Herencia, polimorfismo y redefinición del comportamiento

Polimorfismo

- En Python, las variables no tienen asociado ningún tipo, **todas las variables son polimórficas**, en cualquier momento pueden referenciar a objetos con tipos diferentes.
- Sin embargo, en POO el **polimorfismo** está relacionado con el **principio de sustitución de Liskov**, asociado a la relación de **herencia**:
  - Un objeto de una **subclase** puede ser *referenciado* y *utilizado* como un objeto de la **superclase**, allá donde éste sea necesario, sin afectar a la correción del progr.
    - Un objeto de la subclase **es también un** objeto de la superclase.
  - La dirección de correspondencia opuesta **no** se mantiene:
    - Todos los *Alumnos* **son** *Personas*, pero **no** todas las *Personas* son *Alumnos*.
    - Todos los *Profesores* **son** *Personas*, pero **no** todas las *Personas* son *Profesores*.
  - En *contextos polimórficos*, sólo se debe **invocar** a los métodos definidos por la **clase base** de la jerarquía de los objetos referenciados por una variable o parám.

Vinculación Dinámica

- La **vinculación dinámica** permite que las subclases puedan **redefinir el comportamiento** de los métodos definidos en la superclase.
  - En *contextos polimórficos*, los métodos invocados se seleccionan adecuadamente, durante su ejecución, según la clase a la que pertenece el objeto sobre el que se aplica el método.

Herencia, polimorfismo y redefinición del comportamiento

```python
def prueba_votacion(ident: str, urna: Urna) -> None:
    # Polimorfismo: urna puede referenciar a cualquier objeto Urna o subclase (UrnaOpaca)
    # Polimorfismo: sólo se pueden invocar a los métodos de Urna
    # Todas las UrnasOpacas son Urnas, pero no todas las Urnas son UrnasOpacas
    try:
        for idx in range(2):
            print(f"{ident}; Iteración-{idx+1}")
            votos = input("Introduce votos (s,n): ")
            for voto in votos.split():
                urna.introducir_voto(voto == "s")                # Vinculación dinámica
            print("Resultado intermedio:", urna.resultado())     # Vinculación dinámica
            # print("Urna abierta?", urna.esta_abierta())        # AttributeError: 'Urna' objec
    except RuntimeError as exc:
        print(f"Error: [{exc!r}]")
    finally:
        print(f"{ident}: Resultado final: {urna.resultado()}") # Vinculación dinámica

def main() -> None:
    try:
        urna = Urna(ident="Elección-1")                                  # Argumentos con nombre
        prueba_votacion("Votación-1", urna)
        urna_opaca = UrnaOpaca(ident="Elección-2", grupo="Grupo")    # Argumentos con nombre
        prueba_votacion("Votación-2", urna_opaca) # Polimorfismo
    finally:
        print("UrnaOpaca abierta?", urna_opaca.esta_abierta()) # se invoca sobre urna_opaca
```

Ejemplo completo: Clase Urna

```python
class Urna:
    """Urna de votación."""

    def __init__(self, ident: str, *args, **kwargs) -> None:
        """Inicializa la urna con la cuenta de votos a cero"""
        self.__ident = ident                    # Atributo identificación de la urna
        self.__cnt_positivo = 0                 # Atributo cuenta de votos positivos
        self.__cnt_negativo = 0                 # Atributo cuenta de votos negativos
        super().__init__(*args, **kwargs)       # Inicialización de la clase base

    def introducir_voto(self, voto: bool) -> None:
        """Añade un voto"""
        if voto:
            self.__cnt_positivo += 1
        else:
            self.__cnt_negativo += 1

    def resultado(self) -> bool:
        """Devuelve el resultado de la votación"""
        return (self.__cnt_positivo >= self.__cnt_negativo)
```

Ejemplo completo: Clase UrnaOpaca

```python
class UrnaOpaca(Urna):
    """Urna de votación, modificada para cerrar la votación cuando se consulta el resultado"""

    def __init__(self, grupo: str, *args, **kwargs) -> None:
        """Inicializa la urna opaca con la votación abierta"""
        self.__grupo = grupo                    # Atributo grupo de la urna
        self.__abierta = True                   # Atributo estado de la votación
        super().__init__(*args, **kwargs)       # Inicializacion de la clase base (Urna)

    def esta_abierta(self) -> bool:
        return self.__abierta

    def cerrar_votacion(self) -> None:
        self.__abierta = False

    # @override # a partir de python version 3.12
    def introducir_voto(self, voto: bool) -> None:
        if not self.__abierta:
            raise RuntimeError("Votacion cerrada")
        super().introducir_voto(voto)

    # @override # a partir de python version 3.12
    def resultado(self) -> bool:
        self.__abierta = False
        return super().resultado()
```

Ejemplo completo: programa principal

```python
def prueba_votacion(ident: str, urna: Urna) -> None:
    # Polimorfismo: urna puede referenciar a cualquier objeto Urna o subclase (UrnaOpaca)
    # Polimorfismo: sólo se pueden invocar a los métodos de Urna
    # Todas las UrnasOpacas son Urnas, pero no todas las Urnas son UrnasOpacas
    try:
        for idx in range(2):
            print(f"{ident}; Iteración-{idx+1}")
            votos = input("Introduce votos (s,n): ")
            for voto in votos.split():
                urna.introducir_voto(voto == "s")                # Vinculación dinámica
            print("Resultado intermedio:", urna.resultado())     # Vinculación dinámica
            # print("Urna abierta?", urna.esta_abierta())       # AttributeError: 'Urna' object has no at
    except RuntimeError as exc:
        print(f"Error: [{exc!r}]")
    finally:
        print(f"{ident}: Resultado final: {urna.resultado()}") # Vinculación dinámica
        print()

def main() -> None:
    try:
        urna = Urna(ident="Elección-1")                          # Argumentos con nombre
        prueba_votacion("Votación-1", urna)
        urna_opaca = UrnaOpaca(ident="Elección-2", grupo="Grupo")  # Argumentos con nombre
        prueba_votacion("Votación-2", urna_opaca) # Polimorfismo
    finally:
        print("UrnaOpaca abierta?", urna_opaca.esta_abierta()) # se invoca sobre urna_opaca

if __name__ == "__main__":
    main()
```

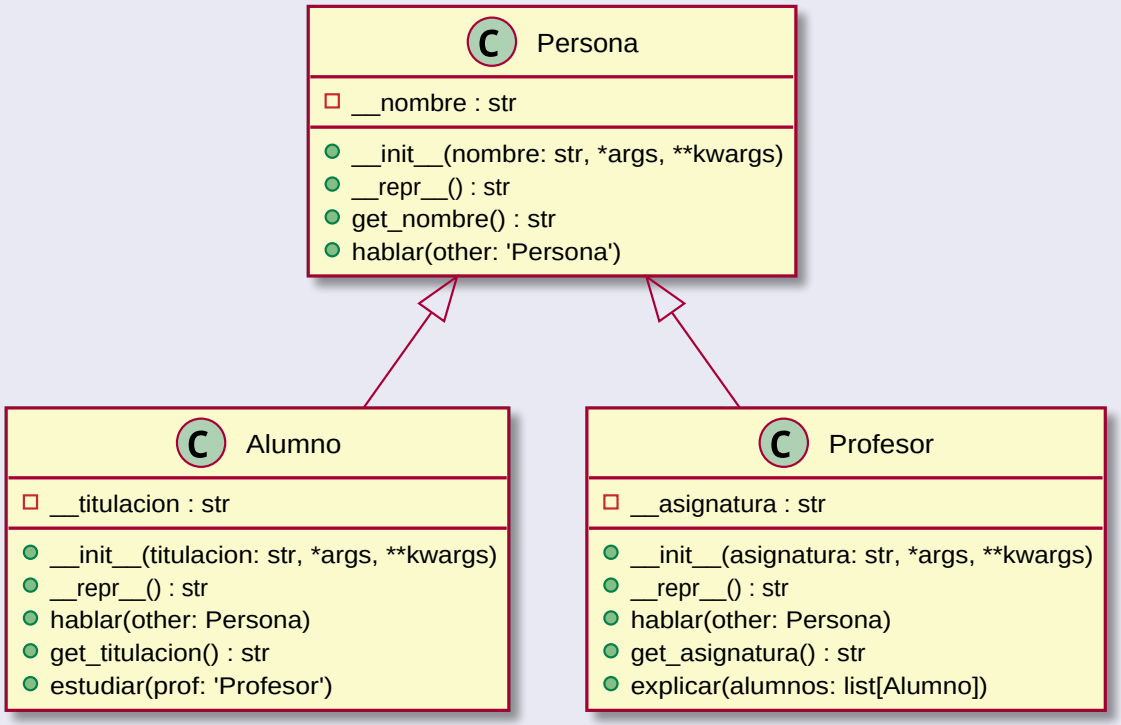
Ejemplo: Salida de la ejecución

```
Votación-1; Iteración-1
Introduce votos (s,n): s s n n n
Resultado intermedio: False

Votación-1; Iteración-2
Introduce votos (s,n): s s s s n n
Resultado intermedio: True

Votación-1: Resultado final: True

Votación-2; Iteración-1
Introduce votos (s,n): s s n n n
Resultado intermedio: False

Votación-2; Iteración-2
Introduce votos (s,n): s s s s n n
Error: [RuntimeError('Votacion cerrada')]

Votación-2: Resultado final: False

UrnaOpaca abierta? False
```

Herencia y redefinición del comportamiento

Ejemplo: Jerarquía de clases: Persona, Alumno y Profesor

- Una *Persona* tiene un nombre, y puede *hablar* con otras personas.
- Un *Alumno* **es una** *Persona* que cursa una *titulación*, y *estudia* con los profesores. Además, cuando *habla*, habla de la titulación que estudia.
- Un *Profesor* **es una** *Persona* que imparte una *asignatura*, y *explica* la materia de su asignatura. Además, cuando *habla*, explica la asignatura que imparte.



- En los métodos del diagrama no aparece el parámetro `self`.

Ejemplo: Jerarquía de clases: Persona, Profesor y Alumno

- Persona:
  - `__init__`: construye una persona, almacenando su nombre.
  - `__repr__`: representación de una persona, *"Persona(Pepe)"*.
  - `get_nombre`: devuelve el nombre de la persona, *"Pepe"*.
  - `hablar`: muestra mensaje de que ambas personas están hablando, *"Persona(Pepe) habla con Persona(María)"*.
- Alumno:
  - `__init__`: construye un alumno, almacenando su titulación y nombre.
  - `__repr__`: representación de un alumno, *"Alumno(Pepe)"*.
  - `get_titulacion`: devuelve la titulación que estudia el alumno, *"CIA"*.
  - `hablar`: muestra mensaje de que ambas personas están hablando. Además, muestra mensaje de que habla de la titulación que estudia.
  - `estudiar`: muestra mensaje de que estudia una titulación, con un profesor, una asignatura. Además, también habla con el profesor.
- Profesor:
  - `__init__`: construye un profesor, almacenando su asignatura y nombre.
  - `__repr__`: representación de un profesor, *"Profesor(María)"*.
  - `get_asignatura`: devuelve la asignatura que imparte el profesor, *"Programación"*.
  - `hablar`: muestra mensaje de que ambas personas están hablando. Además, muestra mensaje de que explica su asignatura.
  - `explicar`: muestra mensaje de que explica una asignatura, habla con todos los alumnos y todos los alumnos estudian con este profesor.

Ejemplo: Jerarquía de clases: Persona, Profesor y Alumno

```python
class Persona:
    """Persona"""
    def __init__(self, nombre: str, *args, **kwargs) -> None:
        self.__nombre = nombre
        super().__init__(*args, **kwargs)

    # @override
    def __repr__(self) -> str:
        return f"Persona({self.__nombre})"

    def get_nombre(self) -> str:
        return self.__nombre

    def hablar(self, other: 'Persona') -> None:
        print(f"{self} habla con {other}")
```

Ejemplo: Jerarquía de clases: Persona, Profesor y Alumno

```python
class Alumno(Persona):
    """Alumno"""
    def __init__(self, titulacion: str, *args, **kwargs) -> None:
        self.__titulacion = titulacion
        super().__init__(*args, **kwargs)

    # @override
    def __repr__(self) -> str:
        return f"Alumno({self.get_nombre()})"

    # @override
    def hablar(self, other: Persona) -> None:
        super().hablar(other)
        print(f"{self} habla de {self.__titulacion}")

    def get_titulacion(self) -> str:
        return self.__titulacion

    def estudiar(self, prof: 'Profesor') -> None:
        print(f"Estudiar({self}, {self.__titulacion}): {prof}, {prof.get_asignatura()}")
        self.hablar(prof)
```

```python
# @override
def __repr__(self) -> str:                                    # ALTERNATIVA
    return f"Alumno({super().__repr__()}, {self.__titulacion})"  # invoca a super().__repr__()
```

Ejemplo: Jerarquía de clases: Persona, Profesor y Alumno

```python
class Profesor(Persona):
    """Profesor"""
    def __init__(self, asignatura: str, *args, **kwargs) -> None:
        self.__asignatura = asignatura
        super().__init__(*args, **kwargs)

    # @override
    def __repr__(self) -> str:
        return f"Profesor({self.get_nombre()})"

    # @override
    def hablar(self, other: Persona) -> None:
        super().hablar(other)
        print(f"{self} explica la asignatura {self.__asignatura}")

    def get_asignatura(self) -> str:
        return self.__asignatura

    def explicar(self, alumnos: list[Alumno]) -> None:
        print(f"Explicar({self}): {self.__asignatura}")
        for alumno in alumnos:
            self.hablar(alumno)
            alumno.estudiar(self)
```

```python
# @override
def __repr__(self) -> str:                                    # ALTERNATIVA
    return f"Profesor({super().__repr__()}, {self.__asignatura})"  # invoca a super().__repr__()
```

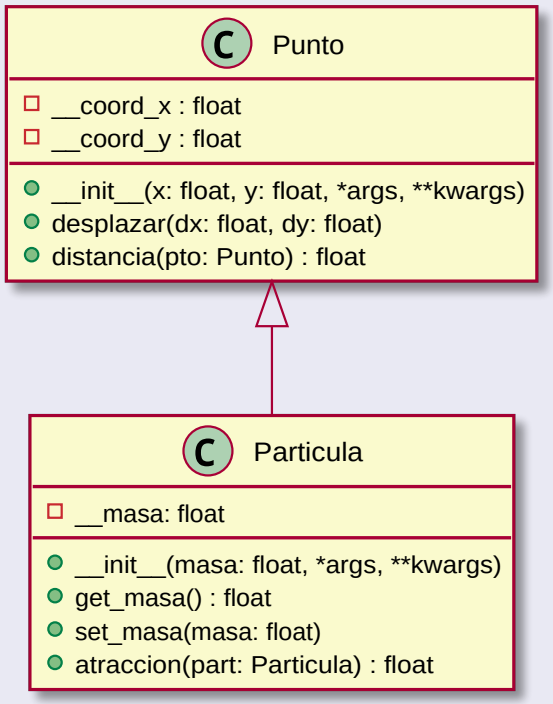
Ejemplo: Jerarquía de clases: Persona, Profesor y Alumno

```
def main() -> None:
    try:
        # persona = Persona("María")                          # El orden de los argumentos MRO(Per, Obj)
        # profesor = Profesor("Programación", "Juan")         # El orden de los argumentos MRO(Prof, Per, Obj)
        # alumno = Alumno("CIA", "Pepe")                      # El orden de los argumentos MRO(Alum, Per, Obj)

        persona = Persona(nombre="María")                                   # Argumentos con nombre
        profesor = Profesor(nombre="Juan", asignatura="Programación")       # Argumentos con nombre
        alumno = Alumno(nombre="Pepe", titulacion="CIA")                    # Argumentos con nombre

        print(persona)                  # Persona(María)
        print(alumno)                   # Alumno(Pepe)
        print(profesor)                 # Profesor(Juan)

        persona.hablar(profesor)        # Persona(María) habla con Profesor(Juan)
        profesor.hablar(alumno)         # Profesor(Juan) habla con Alumno(Pepe) # Profesor(Juan) explica la asignatu
        alumno.hablar(persona)          # Alumno(Pepe) habla con Persona(María) # Alumno(Pepe) habla de CIA

        profesor.explicar([alumno])     # Explicar(Profesor(Juan)): Programación
                                        # Profesor(Juan) habla con Alumno(Pepe) # Profesor(Juan) explica la asignatu
                                        # Estudiar(Alumno(Pepe), CIA): Profesor(Juan), Programación
                                        # Alumno(Pepe) habla con Profesor(Juan) # Alumno(Pepe) habla de CIA

        # profesor.explicar([persona])  # persona no es Alumno ->   AttributeError: 'Persona' object has no attribut
        # alumno.estudiar(persona)      # persona no es Profesor -> AttributeError: 'Persona' object has no attribut

        alumno.estudiar(profesor)       # Estudiar(Alumno(Pepe), CIA): Profesor(Juan), Programación
                                        # Alumno(Pepe) habla con Profesor(Juan) # Alumno(Pepe) habla de CIA
    except ValueError as exc:
        print(f"Error [{exc!r}]")

if __name__ == "__main__":
    main()
```

Ejemplo: Clase Partícula

Ejemplo: Abstracción partícula en el plano cartesiano

- Una `Particula` **es un** `Punto` con **masa**.
  - La `Particula` hereda las características de `Punto` (atributos y métodos), y la extiende añadiendo un nuevo atributo, la masa de la partícula. Además, las partículas tienen la capacidad de *atraerse* entre ellas.



- En los métodos del diagrama no aparece el parámetro `self`.

Ejemplo: Clase Particula

```python
from typing import Final
import punto

_CONST_GRAVIT: Final = 6.67408e-11

class Particula(punto.Punto):
    """Una parícula es un Punto con masa"""

    def __init__(self, masa: float, *args, **kwargs) -> None:
        self.__masa = masa
        super().__init__(*args, **kwargs)

    def get_masa(self) -> float:
        return self.__masa

    def set_masa(self, masa: float) -> None:
        self.__masa = masa

    def atraccion(self, part: 'Particula') -> float:
        # Nótese la invocación al método distancia heredado de Punto
        return _CONST_GRAVIT * self.__masa * part.get_masa() / (self.distancia(part)**2)

def main() -> None:
    p1 = Particula(masa=50, x=10, y=10)      # Argumentos con nombre MRO(Par, Pun, Obj)
    p2 = Particula(masa=80, x=20, y=20)      # Argumentos con nombre MRO(Par, Pun, Obj)
    print("Atracción:", p1.atraccion(p2))    # Atracción: 1.334816e-09

if __name__ == "__main__":
    main()
```

Clases abstractas (I)

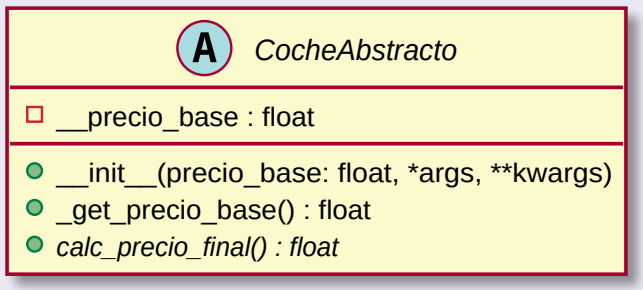
Clases abstractas

- A veces, una clase puede **definir una abstracción de datos incompleta**, porque no le sea posible proporcionar una *definición completa* de todos los métodos de la clase, pero sin embargo, sí puede proporcionar una especificación **abstracta** del comportamiento esperado de los objetos de la clase.
  - Una clase es abstracta si alguno de los **métodos** que define es también **abstracto**, es decir no tiene un cuerpo que defina su comportamiento.
  - Una clase también es abstracta si **hereda** de una clase abstracta, y no implementa todos los **métodos abstractos** de la superclase.
- **No es posible crear objetos (instancias) de clases abstractas**, ya que están incompletas. Se deben definir subclases que no sean abstractas para poder crear objetos.
  - Las **subclases** deben definir adecuadamente los **métodos abstractos** de la superclase, según la abstracción que modelen.
  - Si una subclase no define algún *método abstracto* de la superclase abstracta, entonces esta **subclase** también será **abstracta**.
- Las clases abstractas se utilizan para formar **jerarquías** de clases, especificando los métodos abstractos y comunes para todas las clases derivadas.
  - Garantizan que todos los objetos de las clases derivadas definen los métodos especificados.

Clases abstractas (II)

- En Python, las **clases abstractas** deben heredar de la clase **ABC** (*Abstract Base Class*) del módulo **abc** y pueden tener métodos con el cuerpo vacío (sentencia **pass**), que deben ser *decorados* con **@abstractmethod**.

```python
from abc import ABC, abstractmethod

class CocheAbstracto(ABC):
    """Coche abstracto de un concesionario"""
    def __init__(self, precio_base: float, *args, **kwargs) -> None:
        self.__precio_base = precio_base
        super().__init__(*args, **kwargs)

    def _get_precio_base(self) -> float:
        return self.__precio_base

    @abstractmethod   # método abstracto
    def calc_precio_final(self) -> float:
        """Calcula el precio final del coche"""
        pass
```

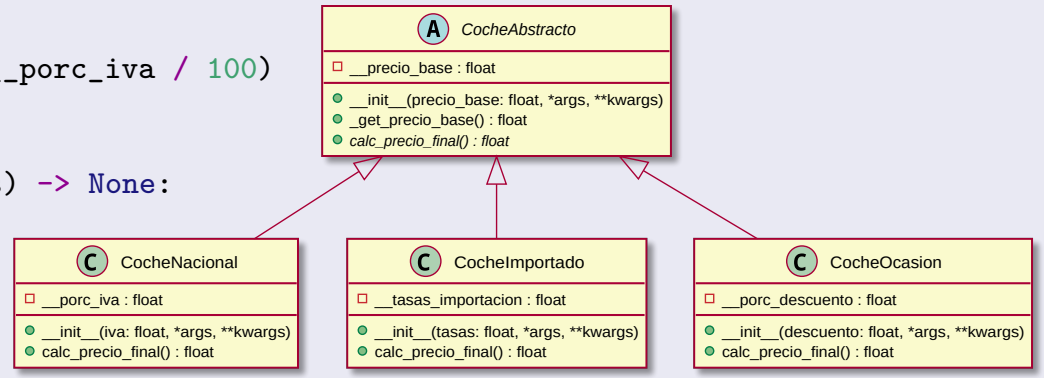


- Las clases abstractas, ademas de los métodos abstractos, también pueden tener métodos concretos (implementados) adicionales, y también pueden tener atributos definidos en el constructor (inicializador).
- En las **clases abstractas puras**, todos los métodos son abstractos, y no tienen ningún atributo, por lo que no se define el constructor (inicializador).

Clases abstractas (III)

- Podemos definir subclases de la *clase base abstracta* que proporcionen una implementación concreta de los métodos abstractos especificados por la superclase.

```
class CocheNacional(CocheAbstracto):
    def __init__(self, iva: float, *args, **kwargs) -> None:
        self.__porc_iva = iva
        super().__init__(*args, **kwargs)

    # @override    # método concreto
    def calc_precio_final(self) -> float:
        return self._get_precio_base() * (1 + self.__porc_iva / 100)

class CocheImportado(CocheAbstracto):
    def __init__(self, tasas: float, *args, **kwargs) -> None:
        self.__tasas_importacion = tasas
        super().__init__(*args, **kwargs)

    # @override    # método concreto
    def calc_precio_final(self) -> float:
        return self._get_precio_base() + self.__tasas_importacion

class CocheOcasion(CocheAbstracto):
    def __init__(self, descuento: float, *args, **kwargs) -> None:
        self.__porc_descuento = descuento
        super().__init__(*args, **kwargs)

    # @override    # método concreto
    def calc_precio_final(self) -> float:
        return self._get_precio_base() * (1 - self.__porc_descuento / 100)
```



Clases abstractas (IV)

- Las **clases abstractas** garantizan que todos los objetos de las clases derivadas definen los métodos especificados.
- Podemos utilizar los objetos de las subclases de la *clase abstracta* en aquellos sitios donde la *clase abstracta* sea necesaria.

```
def mostrar_coches(lista_coches: list[CocheAbstracto]) -> None:
    for coche in lista_coches:
        print(coche.calc_precio_final())

def main() -> None:
    lista_coches: list[CocheAbstracto] = [CocheNacional(16.0, 25000.0),
                                          CocheImportado(5000.0, 25000.0),
                                          CocheOcasion(10.0, 25000.0)]
    mostrar_coches(lista_coches)

if __name__ == "__main__":
    main()
```

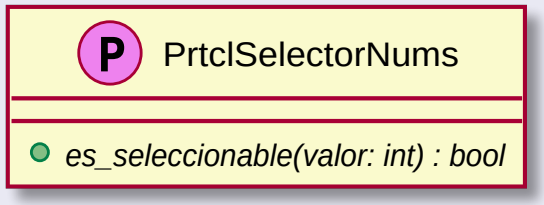
Protocolos

- Un **protocolo especifica los métodos** que deben definir los objetos para poder ser **utilizados** en un determinado contexto.
  - Cuando se especifica un protocolo en la utilización de los objetos, su clase no es importante, lo importante es que definan los métodos requeridos en ese determinado contexto.

- **No es posible crear objetos (instancias) de Protocolos**.
  - Donde se especifique un protocolo, se debe utilizar un objeto de una clase que implemente dicho protocolo. Una clase **implementa un protocolo** si define los métodos especificados por el protocolo.

- Los protocolos son parecidos a las **clases base abstractas puras**, en el sentido que ambos especifican una serie de métodos.
  - Las **ABCs** están orientadas hacia la creación y definición de **jerarquías de relaciones de herencias de clases**. Especifican los métodos comunes que las subclases de la jerarquía de clases deben definir.
  - Los **protocolos** están orientados hacia la **utilización** de los objetos en determinados contextos. Especifican los metodos necesarios para que un objeto pueda ser utilizado en un contexto, independientemente de como haya sido definido, y de sus relaciones de herencia.

Protocolos. Ejemplo (I)



Definición de Protocolo

```
from typing import Protocol

class PrtclSelectorNums(Protocol):
    def es_seleccionable(self, valor: int) -> bool:
        """Devuelve True si el valor recibido debe ser seleccionado, según algún criterio"""
        pass
```

Utilización de Protocolo

```
def seleccionar(lista: list[int], selector: PrtclSelectorNums) -> list[int]:
    """Selecciona los números de la lista que cumplen el criterio de selección,
    especificado por el selector"""
    seleccion: list[int] = list()
    for elemento in lista:
        if selector.es_seleccionable(elemento):
            seleccion.append(elemento)
    return seleccion

    # return [e for e in lista if selector.es_seleccionable(e)]
```

Protocolos. Ejemplo (II)

- Es posible definir **clases independientes** que implementen un determinado protocolo.

```
class SelectorNumsPares:                                    # No utiliza herencia ni ABC
    """Selector de números pares que implementa el protocolo PrtclSelectorNums"""
    def es_seleccionable(self, valor: int) -> bool:
        """Devuelve True si el valor recibido es un número par"""
        return (valor % 2 == 0)
```

```
class SelectorNumsIntervalo:                                # No utiliza herencia ni ABC
    """Selector de números en intervalo que implementa el protocolo PrtclSelectorNums"""
    def __init__(self, inicial: int, final: int, *args, **kwargs) -> None:
        self.__inicial = inicial
        self.__final = final
        super().__init__(*args, **kwargs)

    def es_seleccionable(self, valor: int) -> bool:
        """Devuelve True si el valor recibido está dentro del intervalo"""
        return (self.__inicial <= valor < self.__final)
```

```
def main() -> None:
    res = seleccionar([1, 2, 3, 4, 5, 6], SelectorNumsPares())
    res = seleccionar([1, 2, 3, 4, 5, 6], SelectorNumsIntervalo(2, 5))
```

Protocolos. Ejemplo (III)

- También es posible definir una **jerarquía de clases** que implementen un protocolo.

```
    class SelectorNumsAbstracto(ABC):
        """Selector de números abstracto que implementa el protocolo PrtclSelectorNums"""
        @abstractmethod
        def es_seleccionable(self, valor: int) -> bool:
            """Devuelve True si el valor recibido es seleccionable"""
            pass

    class SelectorNumsPares(SelectorNumsAbstracto):
        # @override
        def es_seleccionable(self, valor: int) -> bool:
            """Devuelve True si el valor recibido es un número par"""
            return (valor % 2 == 0)

    class SelectorNumsIntervalo(SelectorNumsAbstracto):
        def __init__(self, inicial: int, final: int, *args, **kwargs) -> None:
            self.__inicial = inicial
            self.__final = final
            super().__init__(*args, **kwargs)

        # @override
        def es_seleccionable(self, valor: int) -> bool:
            """Devuelve True si el valor recibido está dentro del intervalo"""
            return (self.__inicial <= valor < self.__final)

def main() -> None:
    res = seleccionar([1, 2, 3, 4, 5, 6], SelectorNumsPares())
    res = seleccionar([1, 2, 3, 4, 5, 6], SelectorNumsIntervalo(2, 5))
```