Parte I: Programación en un lenguaje orientado a objetos

1. **Introducción a los lenguajes de programación**
2. **Datos y expresiones**
3. **Clases**
4. **Estructuras algorítmicas**
5. **Estructuras de Datos**
6. **Tratamiento de errores**
7. **Entrada/salida**
8. ***Herencia y Polimorfismo***

- Jerarquía de clases. Herencia. Clases abstractas. Polimorfismo.

8.1. Jerarquía de clases

Uno de los mecanismos importantes de la programación orientada a objetos (OOP) es poder crear clases a partir de otras, por extensión

- Cuando la nueva clase se parece a la anterior se programan solo las extensiones, sin repetir lo común
  - programación por *extensión*

Esto da lugar a una jerarquía:

- clase madre o ***superclase***
- clases hijas o ***subclases***

Notas:

La programación orientada a objetos se basa principalmente en tres conceptos:

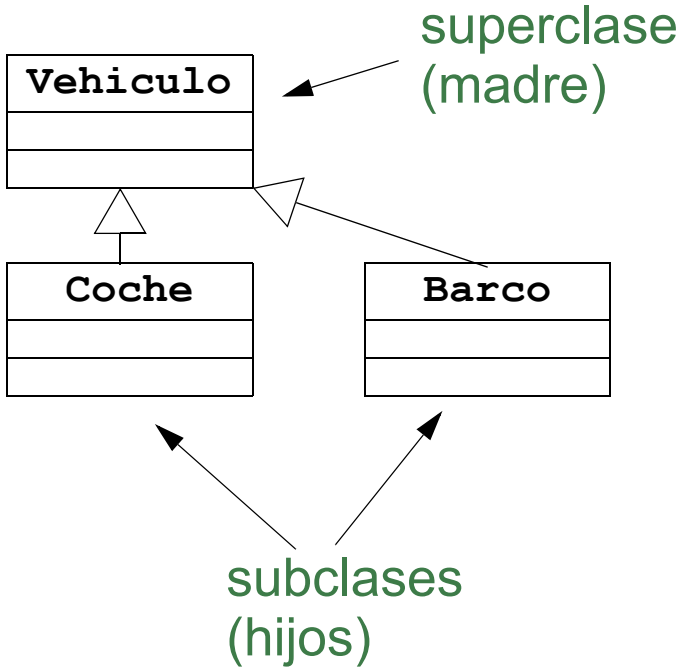
- *Encapsulamiento* de datos y operaciones. Esto se consigue con las clases, en las que se definen los datos (atributos) y operaciones (métodos).
  - A partir de la definición en la clase se pueden crear objetos con datos concretos (también llamados instancias de la clase).
- *Herencia*. Se crea una clase a partir de otra. La clase nueva (hija o subclase) hereda los datos y operaciones de la clase madre (o superclase).
  - Esto da lugar a una jerarquía de clases.
- *Polimorfismo*. Permite tener objetos que pueden ser instancias de cualquier clase de una jerarquía de clases.
  - Son objetos polimórficos, pues pueden tomar muchas formas: la de cualquiera de las clases de la jerarquía.
  - En el ejemplo de la página siguiente, tenemos una jerarquía de clases que parte de la clase `Vehiculo` y define otras dos más: `Coche` y `Barco`
  - Un objeto polimórfico de esta jerarquía podría ser un objeto de cualquiera de estas tres clases: `Vehiculo`, `Coche`, `Barco`. O de otras que creemos en el futuro.

En este capítulo veremos los dos últimos conceptos: herencia y polimorfismo.

8.2. Herencia

Ejemplo de relación de ***herencia***:

- Extendemos la clase `Vehiculo` creando el `Coche` y el `Barco`
- todos los coches son vehículos
- pero no al revés



La ***herencia***: mecanismo para crear nuevas clases a partir de otras existentes,

- heredando y posiblemente añadiendo ***atributos***
- heredando, y posiblemente redefiniendo, y/o añadiendo ***operaciones***

El mecanismo de herencia ***no suprime*** atributos ni operaciones

Herencia y extensión de clases en Python

En Python, para expresar que la clase `Coche` es una extensión de `Vehiculo` se escribe en el encabezamiento de la clase:

```
class Coche(Vehiculo):
    ...
```

Esto pone en marcha todos los mecanismos de la herencia

Notas:

El mecanismo de la herencia se aplica al crear una clase a partir de la otra.

La herencia se aplica a los atributos: se heredan todos los atributos de la superclase.

- Además, la subclase puede *añadir* otros atributos nuevos.

La herencia también se aplica a los métodos: se heredan todos los métodos de la superclase.

- Además, la subclase puede *añadir* otros métodos nuevos.
- Asimismo, la subclase puede modificar (o *redefinir*) métodos heredados de la superclase, para adaptarlos a sus propias necesidades.

Hay que destacar que la herencia nunca permite *suprimir* ni atributos ni métodos.

Herencia de operaciones

Al extender una clase

- se ***heredan*** todas las operaciones de la madre
- se puede ***añadir*** nuevas operaciones

La nueva clase puede elegir para las operaciones *heredadas*:

- ***redefinir*** la operación: se vuelve a escribir
  - la nueva operación redefinida puede usar la de la madre y hacer más cosas: programación *incremental*
  - o puede ser totalmente diferente
- o dejarla como está: heredarla tal como está en la madre

La herencia se puede aplicar múltiples veces

- da lugar a una jerarquía de clases

Notas:

Al heredar una operación tenemos dos opciones:

- No hacer nada. La operación se hereda tal cual.
- Redefinir la operación, volviendo a escribirla.

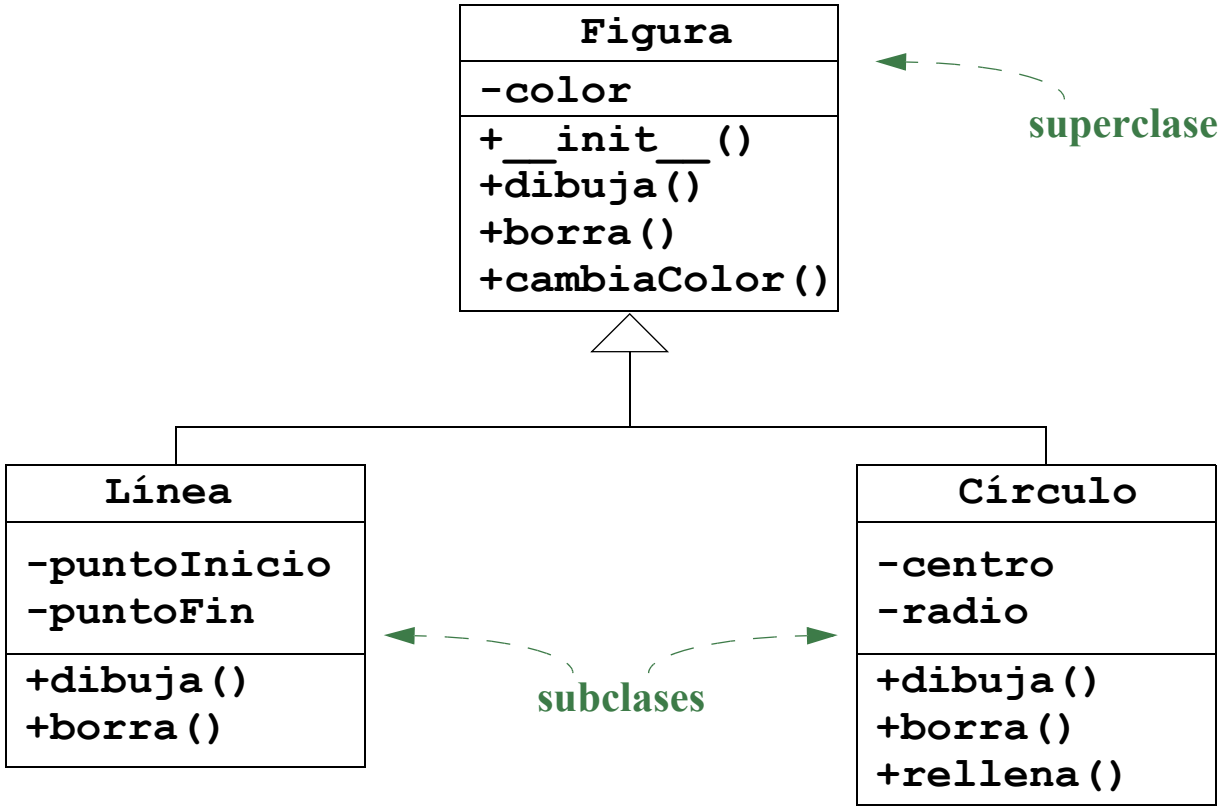
Al *redefinir* una operación heredada de la madre tenemos dos opciones:

- Definir la operación con una implementación totalmente nueva,
- o hacer que la operación nueva invoque a la de la superclase y luego haga más cosas (programación incremental).

Herencia en un diagrama de clases

`Línea` y `Círculo`

- heredan el atributo `color` y añaden otros
- heredan los métodos `dibuja()`, `borra()` y `cambia_color()`



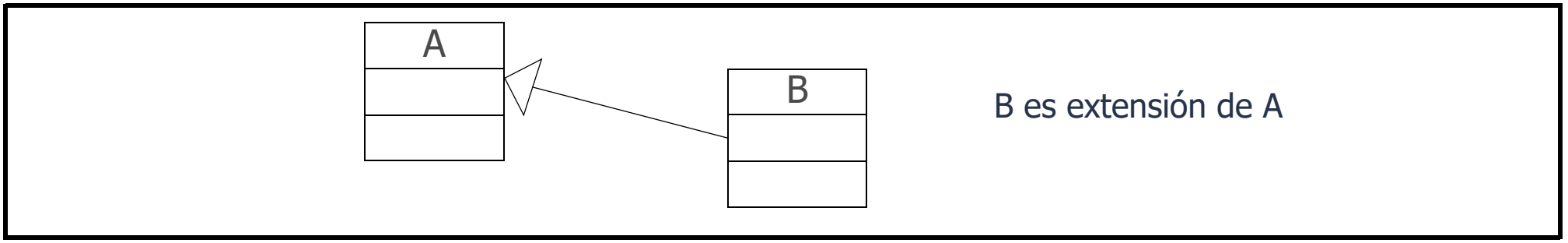
Los métodos de la superclase no se repiten en las subclases, salvo que se hayan redefinido

- por ejemplo, `__init__()` y `cambiaColor()` se heredan sin cambios
- `rellena()` es una operación nueva

Notas:

En los diagramas de clases tenemos una notación específica para la herencia de clases.

La herencia se muestra con una flecha hueca apuntando hacia la superclase. La flecha indica por tanto "*extensión de*".



Los atributos y métodos heredados no se repiten en la subclase.

- Excepto si es un método heredado que se va a *redefinir*. En el caso de las figuras se repiten `dibuja()` y `borra()`.

Los atributos y métodos nuevos se ponen en el diagrama de la subclase.

Herencia y Constructores

El constructor de la superclase se hereda

- pero si tenemos atributos nuevos querremos redefinirlo
- en ese caso también será necesario inicializar los atributos de la superclase
- para ello se llama al constructor de la superclase, `super()`, desde el de la subclase

```
def __init__(self, parámetros...):
  """ Constructor de una subclase """
  # invoca el constructor de la superclase
  super().__init__(parámetros para la superclase)
  # inicializa sus propios atributos
  self.atributo = ...
```

Notas:

Si en una subclase no ponemos constructor, se hereda directamente el de la superclase.

- Al crear un objeto de la subclase se invoca el constructor de la superclase.

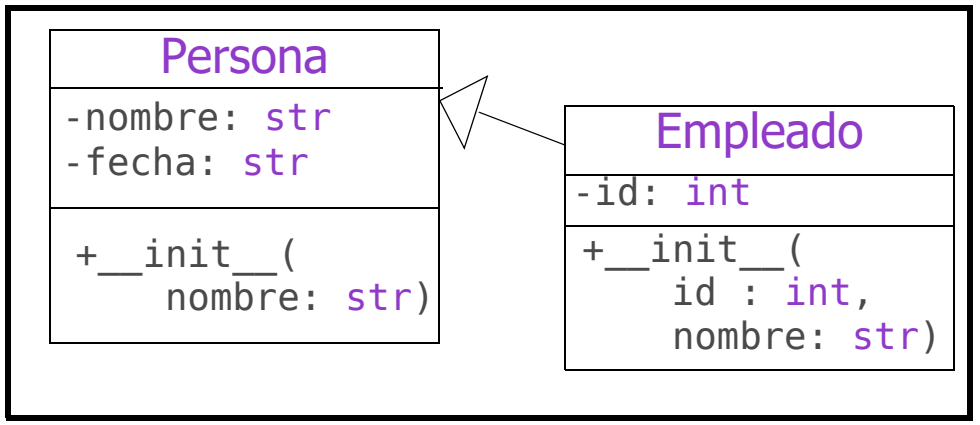
El objetivo del constructor es crear los atributos y darles valor inicial. Por ello, si en la subclase hay atributos nuevos, querremos crearlos y darles valor inicial.

- Y por tanto tendremos que redefinir el constructor.

En el nuevo constructor lo más habitual es aprovechar a invocar el de la superclase, para así crear y dar valor a los atributos definidos allí. Esto es esencial si esos atributos son privados, porque no tenemos acceso.

Ejemplo (sin *docstrings*):

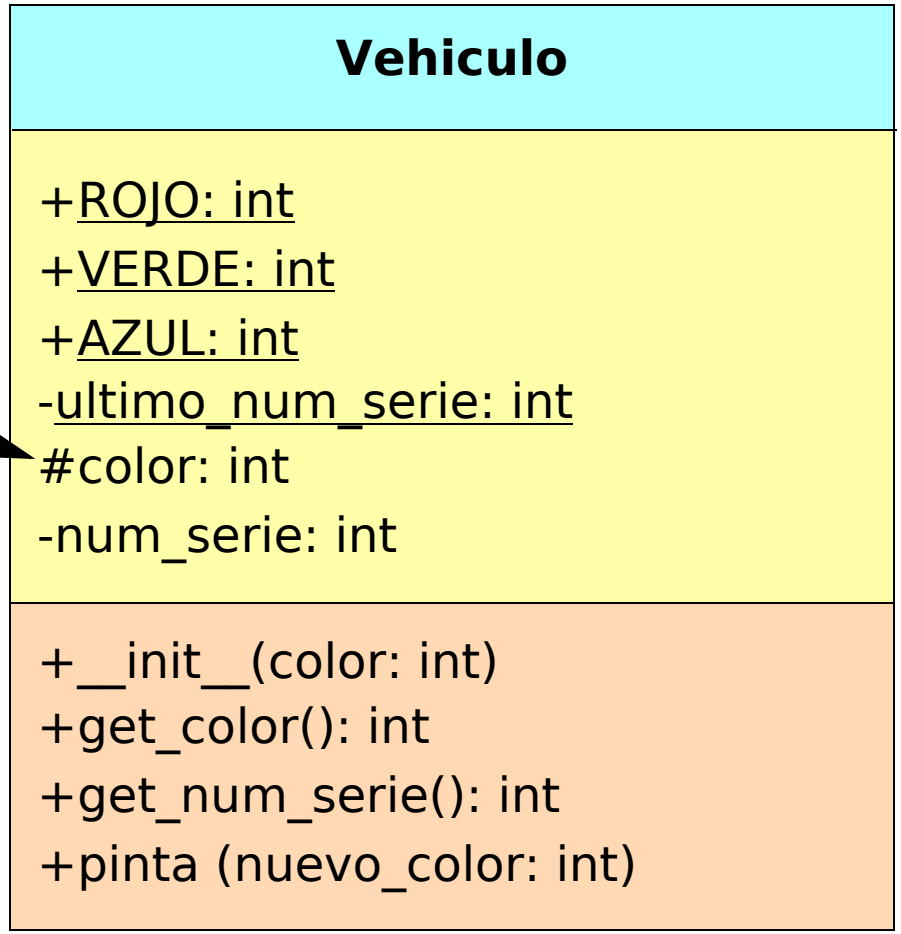
```
class Persona:

    def __init__(self, nombre: str):
        self.__nombre: str = nombre
        self.__fecha: str = "20/5/2022"


class Empleado(Persona):

    def __init__(self, id: int, nombre: str):
        super().__init__(nombre)
        self.__id: int = id
```



Ejemplo

Clase que representa un vehículo cualquiera

Indica atributo protegido
Lo pueden usar los hijos
No se debe usar desde otras clases
En Python comienza por '_'



Notas:

En los diagramas de clases estamos acostumbrados a poner un símbolo delante de los atributos y métodos para indicar su visibilidad:

Símbolo	Tipo de elemento	Descripción	Prefijo usado en Python
+	público	Es visible desde cualquier parte del programa	ninguno
-	privado	Solo es visible desde la propia clase. Ni siquiera es visible desde las subclases	__
#	protegido	Lo pueden usar las subclases además de la propia clase. Aunque también se puede usar desde otras partes del programa esto no debe hacerse, pues la intención expresada por el programador es mantener este elemento oculto	_

Por otro lado, en los diagramas de clases marcamos los atributos y métodos de *clase* o los *estáticos* mediante el subrayado.

- En el ejemplo superior los atributos `ROJO`, `VERDE`, `AZUL` y `ultimo_num_serie` son de *clase*. Van con subrayado.
- Los atributos `color` y `num_serie`, así como todos los métodos, son de *instancia* (también llamados de *objeto*).

Ejemplo: clase Vehiculo

```
class Vehiculo:
    """
    Clase que representa un vehículo cualquiera

    Class Attributes:
        ROJO: un color
        VERDE: un color
        AZUL: un color
        __ultimo_num_serie: el último número de serie asignado
                            a un vehículo
    Instance Attributes:
        _color: el color del vehículo (ROJO, VERDE o AZUL)
        __num_serie: número de serie del vehículo
    """

    # colores de los que se puede pintar un vehículo
    ROJO: int = 1
    VERDE: int = 2
    AZUL: int = 3

    __ultimo_num_serie: int = 0
```

Ejemplo (cont.)

```
def __init__(self, color: int):
    """
    Construye un vehículo dándole un número de serie único
    Args:
        color: el color del vehículo
    """
    self._color = color
    # obtener un nuevo número de serie
    Vehiculo.__ultimo_num_serie += 1
    self.__num_serie = Vehiculo.__ultimo_num_serie


def get_color(self):
    """
    Retorna el color del vehículo
    Returns:
        color del vehículo
    """
    return self._color
```

Ejemplo (cont.)

```
def get_num_serie(self):
    """
    Retorna el número de serie del vehículo
    Returns:
        número de serie del vehículo
    """
    return self.__num_serie


def pinta(self, nuevo_color: int):
    """
    Pinta el vehículo de un color
    Args:
        nuevo_color: color con el que pintar el vehículo
    """
    self._color = nuevo_color
```

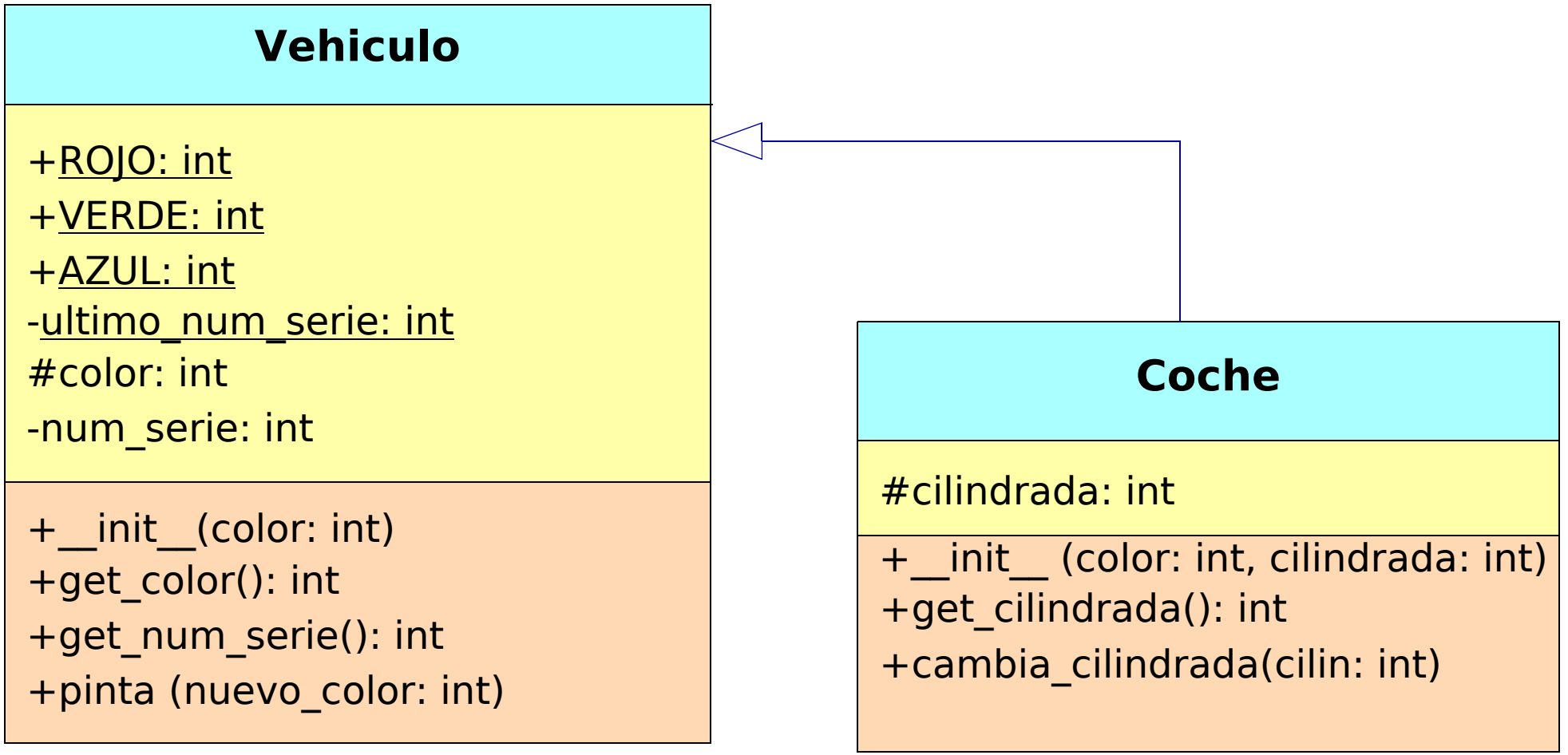
Notas:

En este ejemplo se muestra una clase que no tiene nada de novedoso respecto a lo que hemos visto en capítulos anteriores, excepto el atributo *protegido* llamado `_color`.

- observar el prefijo `"_"` que hemos puesto a su nombre,
- diferente al `"__"` que hemos puesto al atributo privado `__num_serie`.

A continuación extenderemos la clase `Vehiculo` creando las clases `Coche` y `Barco`.

Ejemplo: extensión a la clase Coche



Notas:

Podemos ver que la clase `Coche` es una extensión del `Vehiculo`:

- Se heredan los atributos de instancia `color` y `num_serie`, así como los atributos de clase `ROJO`, `VERDE`, `AZUL` y `ultimo_num_serie`.
  - Aunque el atributo `num_serie` es privado, aún así se hereda. La clase `Coche` no lo puede usar directamente, pero todos los objetos de la clase `Coche` lo tienen. Desde la clase `Coche` se puede conocer el número de serie mediante su método observador `get_num_serie()`, pero no se puede cambiar.
  - En cambio, el atributo `color` es protegido. Se hereda, y la clase `Coche` lo puede ver y usar.
- Se añade el atributo protegido `cilindrada`.
- Se heredan los métodos `get_color()`, `get_num_serie()` y `pinta()` sin cambios.
- Se redefine el constructor.
- Se añaden los métodos `get_cilindrada()` y `cambia_cilindrada()`.

Extensión a la clase Coche (cont.)

```
class Coche(Vehiculo):
    """
    Clase que representa un coche

    Instance Attributes:
        _cilindrada: cilindrada del coche

    Inherited class Attributes:
        ROJO: un color
        VERDE: un color
        AZUL: un color
        __ultimo_num_serie: el último número de serie asignado
                            a un vehículo

    Inherited instance Attributes:
        _color: el color del que se pinta un vehículo
                (ROJO, VERDE o AZUL)
        __num_serie: número de serie del vehículo
                     (no lo puede usar por empezar por __)
    """
```

Extensión a la clase Coche (cont.)

```
    def __init__(self, color: int, cilindrada: int):
        """
        Crea el coche con el color y la cilindrada indicados
        Args:
            color: el color del coche
            cilindrada: la cilindrada del coche en cc
        """
        super().__init__(color)
        self._cilindrada = cilindrada

    def get_cilindrada(self) -> int:
        """
        Retorna la cilindrada del coche
        Returns:
            La cilindrada del coche en cc
        """
        return self._cilindrada
```

Extensión a la clase Coche (cont.)

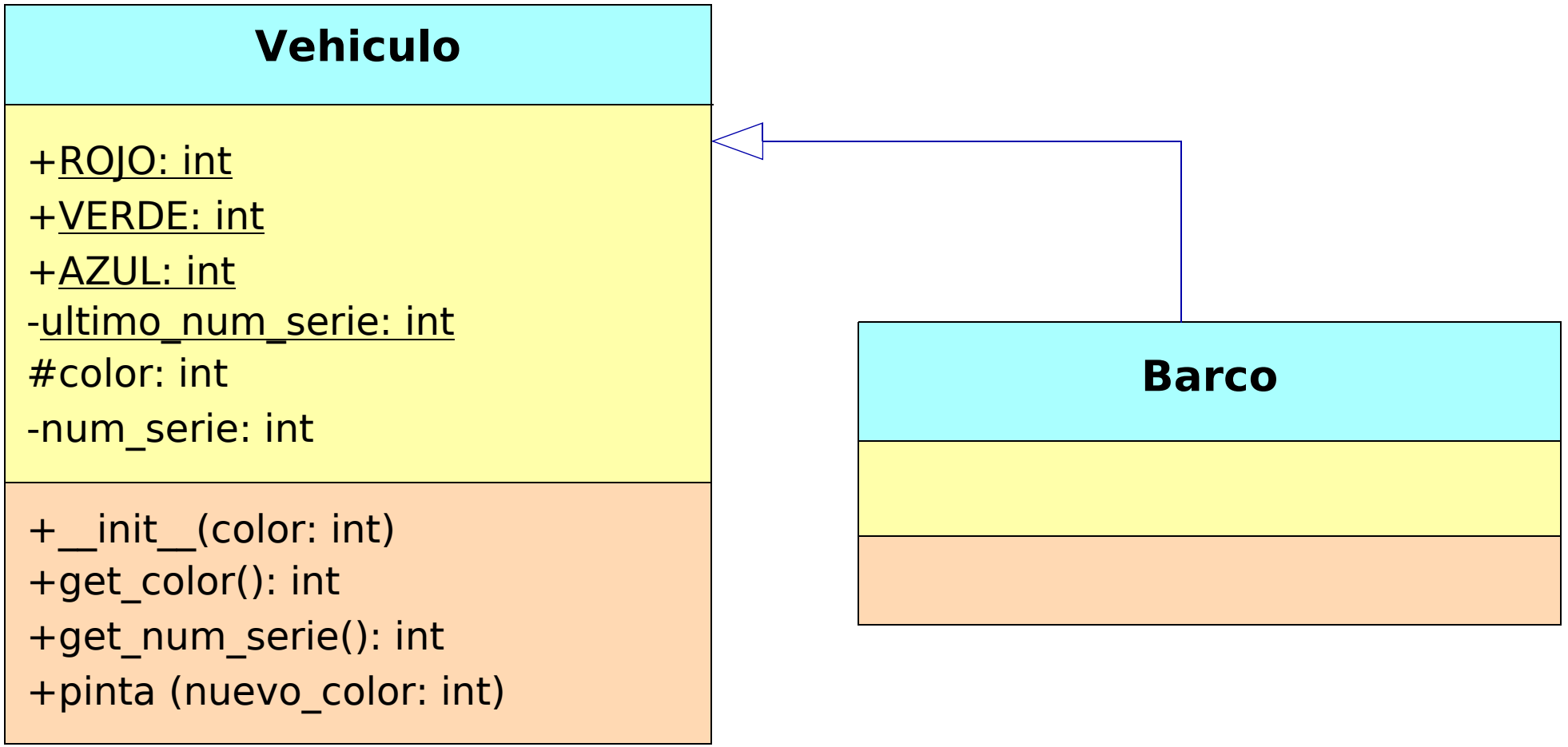
```
    def cambia_cilindrada(self, cilin: int):
        """
        Cambia la cilindrada del coche al valor indicado, en cc
        Args:
            cilindrada: la nueva cilindrada del coche en cc
        """
        self._cilindrada = cilin
```

Notas:

Es interesante observar el carácter incremental del constructor, que invoca al de la superclase para crear e inicializar los atributos heredados (`_color` y `__num_serie`) y luego crea e inicializa el nuevo atributo `_cilindrada`.

Los nuevos métodos son convencionales.

Ejemplo: extensión a la clase Barco



Se puede hacer herencia sin añadir métodos ni atributos

Notas:

Vemos ahora una nueva extensión de la clase `Vehiculo`, llamada `Barco`.

Es un caso curioso pero frecuente. El `Barco` no añade ni atributos ni métodos.

- Entonces, ¿para qué se hace? El motivo es que la clase `Barco` representa un *concepto* diferente al del `Vehiculo` en general. El `Vehiculo` puede ser un `Coche`, un `Barco`, o una nave espacial si la creásemos en el futuro. Pero el `Barco` representa otro concepto, que es el de un `Vehiculo` que navega sobre el agua.

Ejemplo (cont.)

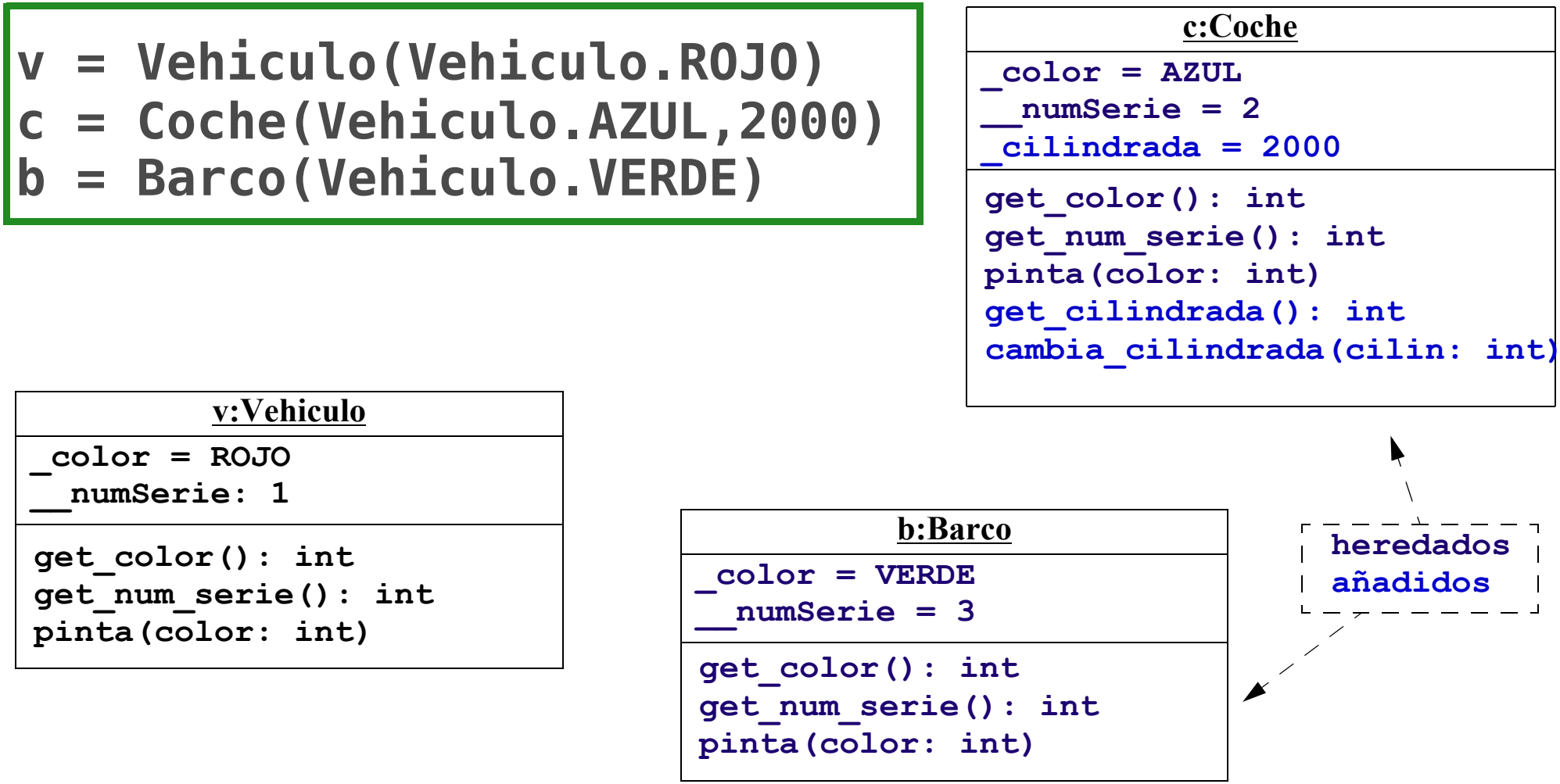
```
class Barco(Vehiculo):
    """
    Clase que representa un barco: un Vehiculo que navega

    Instance Attributes:

    Inherited class Attributes:
        ROJO: un color
        VERDE: un color
        AZUL: un color
        __ultimo_num_serie: el último número de serie
                            asignado a un vehículo

    Inherited instance Attributes:
        _color: el color del que se pinta un vehículo
                (ROJO, VERDE o AZUL)
        __num_serie: número de serie del vehículo
                     (no lo puede usar por empezar por __)
    """
```

Notas:

Observar que la clase `Barco` solo necesita en Python la cabecera y la documentación. El resto no se pone, ya que se hereda todo lo que tiene el `Vehiculo`, incluido su constructor.

Ejemplo: objetos y herencia

```
v = Vehiculo(Vehiculo.ROJO)
c = Coche(Vehiculo.AZUL,2000)
b = Barco(Vehiculo.VERDE)
```



Notas:

En la página anterior se puede ver dentro del rectángulo verde las instrucciones necesarias para crear un objeto de cada una de las clases `Vehiculo`, `Coche` y `Barco`.

Asimismo se muestra un diagrama de objetos, que permite mostrar los objetos creados, con su estructura y sus datos concretos.

El *diagrama de objetos* se parece al de clases, con estas diferencias:

- En el encabezamiento de cada objeto se usa la notación `nombre_objeto:Clase`
- En la segunda parte se ponen los atributos, con sus valores concretos.
- En la tercera parte se ponen los métodos. Esta parte puede omitirse.

Redefiniendo operaciones

Una subclase puede *redefinir* una operación en lugar de heredarla directamente

- basta repetir la operación con la misma cabecera

En muchas ocasiones (no siempre) la operación redefinida invoca la de la superclase

- se usa para ello la palabra `super()`
- se refiere a la superclase directa del objeto actual

Invocar un método de la superclase:

```
super().nombre_método(parámetros...)
```

Ejemplo: nueva operación en la clase Vehiculo

Todas las clases Python heredan de una clase general llamada `object`

En `object` existe la operación `__str__()` que es interesante para convertir un objeto en texto

- retorna un string
- se usa directamente al hacer `print(mi_objeto)`
  - equivale a `print(mi_objeto.__str__())`
- podemos redefinirla en cualquier clase

Notas:

La operación `__str__()` es muy útil para convertir un objeto en string. Si la redefinimos podemos controlar el contenido y formato de ese string.

En el ejemplo de la página 12, donde se creaban la clases `Persona` y `Empleado`, supongamos que creamos una persona y la mostramos en pantalla con la función `print()`:

```
p = Persona("pepe")
print(p)
```

Obtendríamos la siguiente salida, porque la función `__str__()` aplicada a un objeto nos muestra el nombre de la clase y la dirección en memoria, que no suele ser muy útil:

```
<__main__.Persona object at 0x000001FD982B0588>
```

Podemos redefinir la función `__str__()` y mostrar el nombre de la persona, que es mucho más útil. En la clase `Persona` haríamos:

```
def __str__(self) -> str:
    return f"Persona de nombre: {self.nombre}"
```

Al mostrar una `Persona` o `Empleado` veremos su nombre. Por ejemplo, con `print(p)` obtendremos:

```
Persona de nombre: pepe
```

Que es mucho más útil. A continuación mostramos lo mismo con las clases `Vehiculo` y `Coche`.

Ejemplo (cont.)

```
class Vehiculo:

    ...

    def __str__(self) -> str:
        """
        Retorna un string que describe el vehículo
        """
        nombre_color: list[str] = ["ROJO", "VERDE", "AZUL"]
        # restamos 1 al color porque los colores se numeran
        # de 1 a 3 y sus nombres de 0 a 2
        return f"num_serie={self.__num_serie} color=" + \
            f"{nombre_color[self._color-1]}"
```

Ejemplo: redefinir la nueva operación en la clase Coche

```
class Coche:

    ...

    def __str__(self) -> str:
        """
        Retorna un string que describe el coche, incluyendo
        su cilindrada
        """
        return f"Coche: {super().__str__()}, " + \
               f"cil={self._cilindrada}"
```

Ejemplo: redefinir la nueva operación en la clase Barco

```
class Barco:

    ...

    def __str__(self) -> str:
        """
        Retorna un string que describe el barco
        """
        return f"Barco: {super().__str__()}"
```

Notas:

Vemos que en la clase Vehiculo redefinimos la operación `__str__()` para mostrar el número de serie y el color del vehículo.

Luego en la clase `Coche` redefinimos de nuevo la operación `__str__()`:

- Lo hacemos poniendo la palabra "Coche", invocando luego a la operación de la superclase, y añadiendo al final lo que es nuevo en el coche: el atributo `cilindrada`.

Finalmente en la clase `Barco` redefinimos de nuevo la operación `__str__()`:

- Lo hacemos poniendo la palabra "Barco" e invocando a la operación de la superclase para mostrar los atributos heredados: el número de serie y el color

Resumen Herencia

Las clases se pueden ***extender***

- la subclase ***hereda*** los atributos y métodos de la superclase
- Se *debe* acceder a los atributos y métodos solo según su visibilidad

Visibilidad	Comienza por	Desde la propia clase	Desde las subclases	Desde otras clases
- privado	__	✓	✗	✗
# protegido	_	✓	✓	✗
+ público		✓	✓	✓

A la subclase se le pueden ***añadir*** nuevas operaciones y atributos

Al extender una clase se pueden ***redefinir*** sus operaciones

- si se desea, se puede invocar desde la nueva operación a la de la superclase: ***programación incremental***

8.3. Clases abstractas

En ocasiones definimos clases de las que no pretendemos crear objetos

- su único objetivo es que sirvan de superclases a las clases "reales"

Ejemplos:

- nunca crearemos objetos de la clase `Figura`
  - lo haremos de sus subclases `Circulo`, `Cuadrado`, ...
- nunca crearemos un `Vehiculo`
  - crearemos un `Coche`, un `Barco`, un `Avion`, ...

A ese tipo de clases las denominaremos ***clases abstractas***

Las clases abstractas pueden tener ***métodos abstractos***, que no tienen implementación, pero deben ser redefinidos en las subclases

Notas:

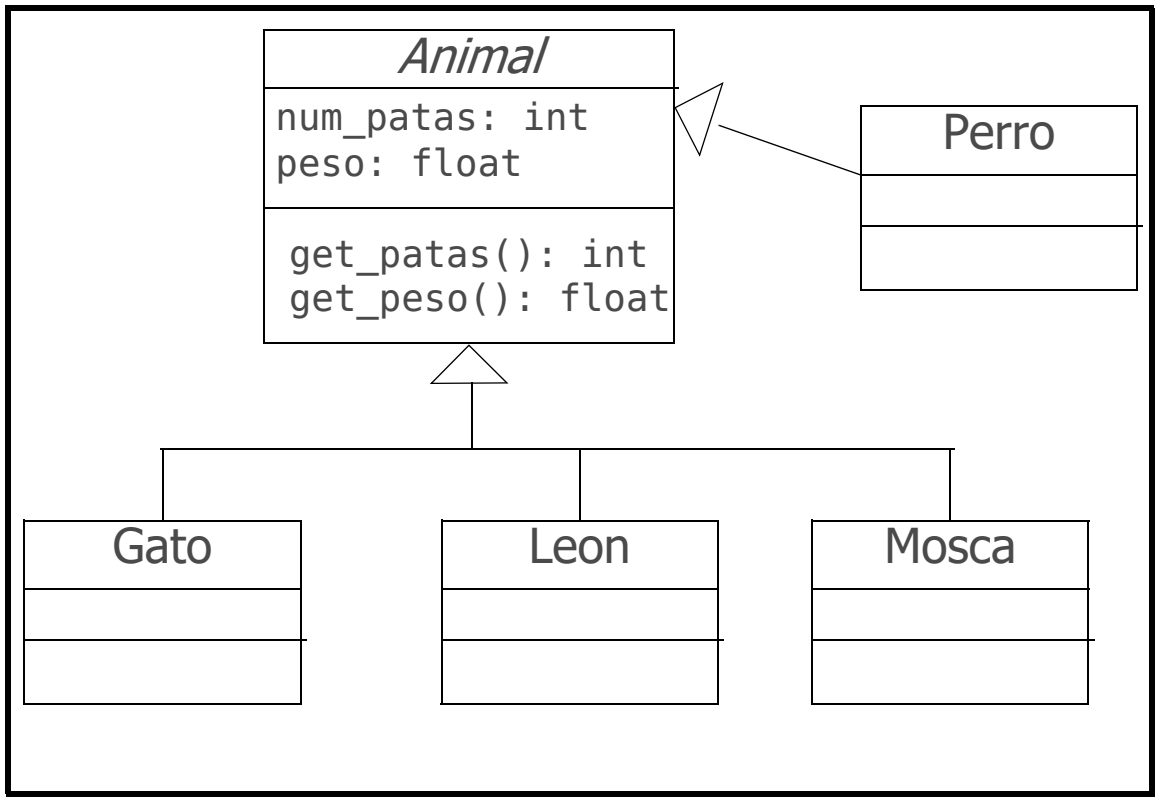
La *clase abstracta* representa conceptos abstractos, que no tienen ninguna realización física.

- Tienen una restricción muy importante: no se puede crear objetos de ellas.
- Pero pueden tener atributos y métodos pensados para ser heredados por sus subclases.
  - Es posible que los métodos no tengan implementación: son los *métodos abstractos*, pensados para ser heredados y con ello garantizar que existen en todas las subclases.

Por ejemplo, podemos pensar en el concepto abstracto "animal". En la realidad no existen animales que solo sean animales sin más. Existen perros, gatos, leones o moscas, que son tipos de animales concretos.

El "animal" podría ser el padre de la jerarquía. Podemos pensar en atributos y métodos comunes a los animales, como los que se muestran en el diagrama. Las especies animales concretas se representarían con subclases que heredarían todos estos atributos y métodos.

En el diagrama de clases, los elementos abstractos van en *cursiva*.



Clases abstractas en Python

Existe un módulo llamado `abc` (*abstract base classes*) que da soporte a las clases abstractas

```
from abc import ABC, abstractmethod
```

Las clases abstractas se crean heredando de `ABC`:

```
class Vehiculo(ABC):
    ...
```

Es erróneo tratar de crear un objeto de una clase abstracta

```
Vehiculo v = Vehiculo(Vehiculo.ROJO) # ¡no hacer!
```

Métodos abstractos

Una clase abstracta puede tener métodos abstractos

- se trata de métodos sin implementación
- que ***es obligatorio redefinir*** en las subclases no abstractas
- ejemplo de método abstracto:

```
class Vehiculo(ABC):
    ...
    @abstractmethod
    def medio_por_el_que_se_mueve(self) -> str:
        """
        Retorna el medio por el que va el vehículo
        """
        pass
```

no tiene instrucciones, u opcionalmente `pass`

Notas:

Hemos visto ver la forma de crear clases y métodos abstractos en Python.

- La clase hereda de la clase ABC.
- El método abstracto lleva la anotación `@abstractmethod` justo encima de su cabecera.
  - Luego, se pone sin instrucciones o simplemente con la instrucción **pass**, que no hace nada.

La existencia de este método abstracto en la clase `Vehiculo` implica:

- Hay garantía de que todas las subclases de `Vehiculo` tendrán este método, por herencia.
  - Esto tiene implicaciones en el polimorfismo, que veremos más abajo.
- Las subclases directas de `Vehiculo` están obligadas a redefinir este método, a no ser que a su vez sean abstractas.

Métodos abstractos: redefinición en el **Coche**

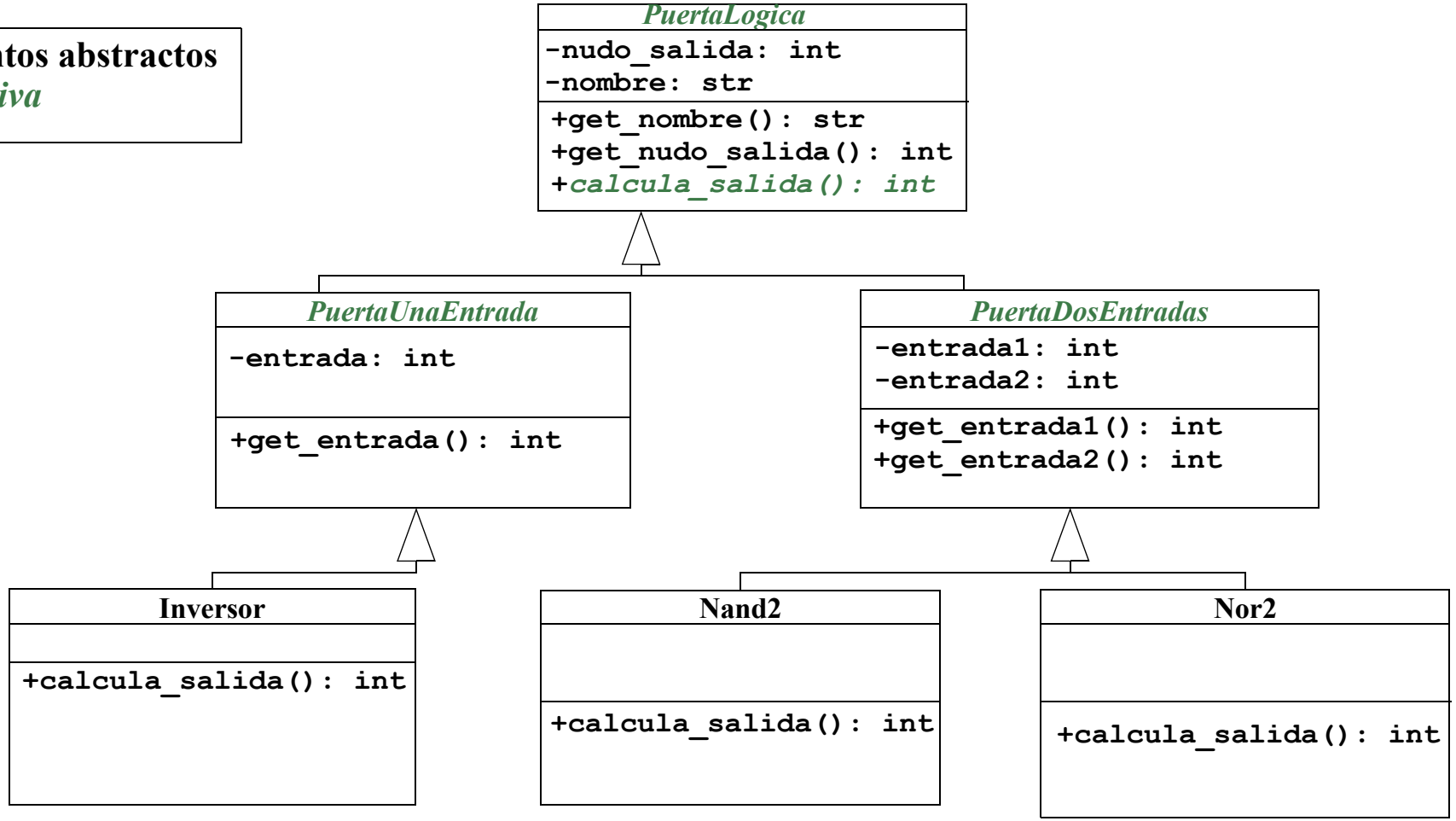
```
class Coche(Vehiculo):
    ...
    def medio_por_el_que_se_mueve(self) -> str:
        """
        Retorna el medio por el que va el coche
        """
        return "Terrestre"
```

Métodos abstractos: redefinición en el **Barco**

```
class Barco(Vehiculo):
    ...
    def medio_por_el_que_se_mueve(self) -> str:
        """
        Retorna el medio por el que va el barco
        """
        return "Acuático"
```

Ejemplo: jerarquía con clases abstractas

Elementos abstractos en *cursiva*



Notas:

En este ejemplo adicional se muestra una jerarquía de clases con varias clases abstractas y otras concretas. Por sencillez no incluimos los constructores.

En la raíz de la jerarquía se muestra la clase abstracta `PuertaLogica`, que representa una puerta lógica de un circuito digital, con entradas y salidas que operan con valores lógicos 0 y 1.

- Su método `calcula_salida()` es abstracto. Nadie sabría cómo calcular la salida de una puerta lógica así, en abstracto.
- Sin embargo, los métodos `get_nombre()` y `get_nudo_salida()` son concretos, pues su implementación es posible: Consistirá en retornar el atributo correspondiente. Esto lo pueden heredar directamente todas las subclases.

En el siguiente nivel de la jerarquía aparecen las clases también abstractas `PuertaUnaEntrada` y `PuertaDosEntradas`, representando puertas lógicas de una o dos entradas, respectivamente. Siguen siendo abstractas pues no representan ninguna puerta real con la que podamos operar.

- `PuertaUnaEntrada` añade el atributo que indica cuál es el identificador del nudo eléctrico de su única entrada. También añade un método para obtener este atributo.
- `PuertaDosEntradas` añade los atributos que indican cuáles son los identificadores de los nudos eléctricos de sus entradas. También añade un par de métodos para obtener estos atributos.

El tercer nivel de la jerarquía contiene las puertas lógicas concretas como inversores y puertas NAND o NOR de dos entradas. Todas redefinen obligatoriamente el método abstracto `calcula_salida()` heredado de la `PuertaLogica`.

8.4. Polimorfismo

En OOP es común proveer una interfaz común con las mismas operaciones, a objetos de clases pertenecientes a la misma jerarquía

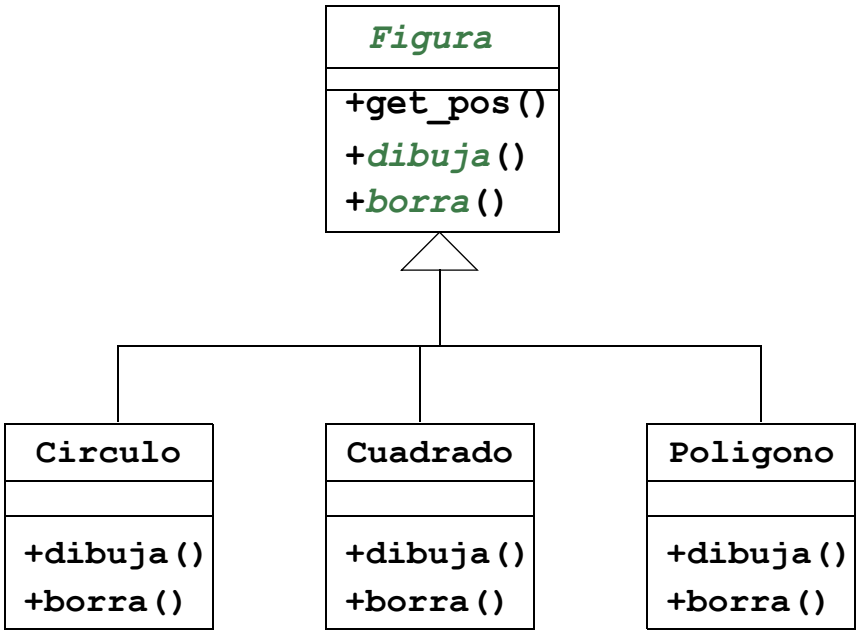
Las ***operaciones polimórficas*** son aquellas que, invocándose de la misma forma, hacen funciones similares con objetos de tipos diferentes

Ejemplo: suponer que existe la clase `Figura` y sus subclases

- `Circulo`
- `Cuadrado`
- `Poligono`

Todas ellas con las operaciones:

- `dibuja()`
- `borra()`
- `get_pos()`



Notas:

El polimorfismo es una propiedad que permite tener objetos que pueden ser instancias de cualquier clase de una jerarquía de clases.

- Son objetos *polimórficos*, pues pueden tomar muchas formas: la de cualquiera de las clases de la jerarquía.

En el ejemplo de arriba con la jerarquía definida por la clase `Figura` y sus subclases (`Circulo`, `Cuadrado` y `Poligono`), podemos tener una referencia polimórfica, a un objeto de cualquiera de las clases de la jerarquía.

El mecanismo de la herencia nos garantiza que todos los objetos de estas clases tienen al menos las operaciones `get_pos()`, `dibuja()` y `borra()`.

- Esta es su interfaz común.

Polimorfismo (cont.)

Supongamos que disponemos de la variable `fig1`, que representa una figura cualquiera de la jerarquía

- `fig1` puede ser un objeto de las clases `Circulo`, `Cuadrado` o `Poligono`

Podemos invocar a la operación `fig1.dibuja()` con seguridad

- por la herencia, está garantizado que esta operación existe

Pero ¿cuál de las operaciones `dibuja()` se invocará?

- Si `fig1` es un `Circulo`, se invocará al `dibuja()` de su clase
- Similarmente, si `fig1` es un `Cuadrado` o un `Poligono`

La elección de la operación adecuada es automática

- se dice que la operación `dibuja()` se ha invocado polimórficamente: tiene muchas formas

Notas:

Supongamos que existe la variable `fig2` que es un objeto de las clases `Circulo`, `Cuadrado` o `Poligono`.

Supongamos ahora estas instrucciones Python:

```
fig1: Figura = fig2
fig1.dibuja()          # método polimórfico
```

¿Cuál de los métodos `dibuja()` definidos en la jerarquía de clases se utilizará?

- No lo sabemos. Se seleccionará el método apropiado a la figura concreta que se quiere dibujar.
- Por ello se dice que esta llamada al método `dibuja()` es *polimórfica*.

Polimorfismo en Python

El polimorfismo se basa en dos propiedades:

1. *Referencias polimórficas*: una referencia a una superclase puede apuntar a un objeto de cualquiera de sus subclases

```
v_1: Vehiculo = Coche(Vehiculo.ROJO, 2000)
v_2: Vehiculo = Barco(Vehiculo.AZUL)
```

2. *Selección automática de la operación*: la operación se selecciona en base a la clase del objeto, no a la de la referencia

```
print(v_1)
```
← **usa el método `__str__()` de la clase `Coche`, puesto que `v_1` es un coche**

```
print(v_2)
```
← **usa el método `__str__()` de la clase `Barco`, puesto que `v_2` es un barco**

Notas:

Las variables `v_1` y `v_2` que se muestran arriba están anotadas con el tipo `Vehiculo` del ejemplo de la página 15.

- En OOP esto implica que estas variables pueden referirse a un objeto de la clase `Vehiculo` o cualquiera de sus subclases. Es un objeto *polimórfico*, porque puede adoptar muchas formas.
- Similarmente, si creásemos la variable `v_3: Coche`, esto implicaría que esta variable podría ser un objeto de la clase `Coche` o de cualquiera de sus subclases presentes o futuras, pero no un `Barco`, pues el `Barco` no es una subclase del `Coche`.
  - Es lógico: los coches y barcos son vehículos. Pero los barcos no son coches. Tampoco un coche es un barco.
  - Ahora bien, aunque un coche es un vehículo, es una extensión de él y tiene más cosas que las de un simple vehículo. Por ejemplo, tiene cilindrada.

Cuando se hace una operación `print()` sobre una de estas variables, por ejemplo `print(v_1)` se usará implícitamente el método `__str__()` de conversión a texto.

- Esta operación es *polimórfica*: se seleccionará en función de la clase concreta del objeto.

Ejemplo con polimorfismo

Nos gustaría hacer una operación externa `reemplaza_fig` que opere correctamente con cualquier clase de figura:

```
def reemplaza_fig(figura1, figura2):
  borra la figura1
  dibuja la figura2 en su lugar
```

Pseudocódigo del ejemplo, sin polimorfismo

```
pos = figura1.get_pos()
si figura1 es un Circulo entonces
    borra el círculo
si no, si figura1 es un Cuadrado entonces
    borra el cuadrado
si no, si figura1 es un Poligono entonces
    borra el polígono
...
fin si
si figura2 es un Circulo entonces
    dibuja el círculo en pos
si no, si figura1 es un Cuadrado entonces
    dibuja el cuadrado en pos
si no, si figura1 es un Poligono entonces
    ...
```

Esto es *largo*, pero además *no* serviría para figuras creadas en el *futuro*

Notas:

Para ilustrar el concepto del polimorfismo planteamos un ejemplo con las figuras definidas en la página 48.

En el ejemplo pretendemos escribir una operación que sea capaz de reemplazar en un dibujo una figura por otra.

- Pretendemos que esta operación sirva para cualquier combinación de figuras de cualquiera de las clases de la jerarquía.

Si no tuviésemos el mecanismo del polimorfismo tendríamos que programar la operación como una sucesión de instrucciones condicionales, para hacer las operaciones adecuadas en función de los tipos de las figuras.

El pseudocódigo anterior es muy largo, sobre todo si hay muchas figuras.

Lo peor es que si en el futuro creamos nuevas figuras, ya no serviría. Habría que modificarlo para incluir las nuevas figuras.

Con el polimorfismo evitamos ambos inconvenientes.

Ejemplo con polimorfismo

Gracias a las dos propiedades en que se basa el polimorfismo, la función `reemplaza_fig` sería:

```
def reemplaza_fig(fig1: Figura, fig2: Figura):
    """ Reemplaza fig1 por fig2"""
    pos = fig1.get_pos()
    fig1.borra()
    fig2.dibuja(pos)
```

Y podría invocarse de la forma siguiente:

```
cir = Circulo(...)
pol = Poligono(...)

reemplaza_fig(cir, pos)
```

Ejemplo con polimorfismo (cont.)

- Gracias a las referencias polimórficas, los parámetros `fig1` y `fig2` pueden referirse a cualquier subclase de `Figura`
- Gracias a la selección automática de la operación, en `reemplaza_fig` se llama a las operaciones `borra()` y `dibuja()` apropiadas

El código de esta operación es breve y, sobre todo, funciona con cualquier subclase presente y futura de la clase `Figura`

Ejemplo con lista polimórfica

lista polimórfica

```
def borra_lista(lista : list[Figura]):
    """
    Borra una lista de figuras
    """
    for fig in lista:
        fig.borra()
```

llamada polimórfica a `borra()`

Notas:

La propiedad de las referencias polimórficas nos permite hacer listas o tuplas heterogéneas, que mezclan objetos de las diferentes clases de la jerarquía.

En el ejemplo se muestra una lista de objetos de la clase `Figura`, o cualquiera de sus subclases presentes o futuras.

Se crea una operación para borrar una lista de figuras. Simplemente iteramos sobre todas ellas y las vamos borrando con una llamada polimórfica al método `borra()`.

Resumen

El polimorfismo nos permite abstraer operaciones

- podemos invocarlas sin preocuparnos de las diferencias existentes para objetos diferentes
- el sistema elige la operación apropiada al objeto

El polimorfismo se asocia a las jerarquías de clases:

- una superclase y todas las subclases derivadas de ella

El polimorfismo se basa en dos propiedades:

- *Referencias polimórficas*: una referencia a una superclase puede apuntar a un objeto de cualquiera de sus subclases
- *Operaciones polimórficas*. La operación se selecciona automáticamente en base a la clase del objeto concreto