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*

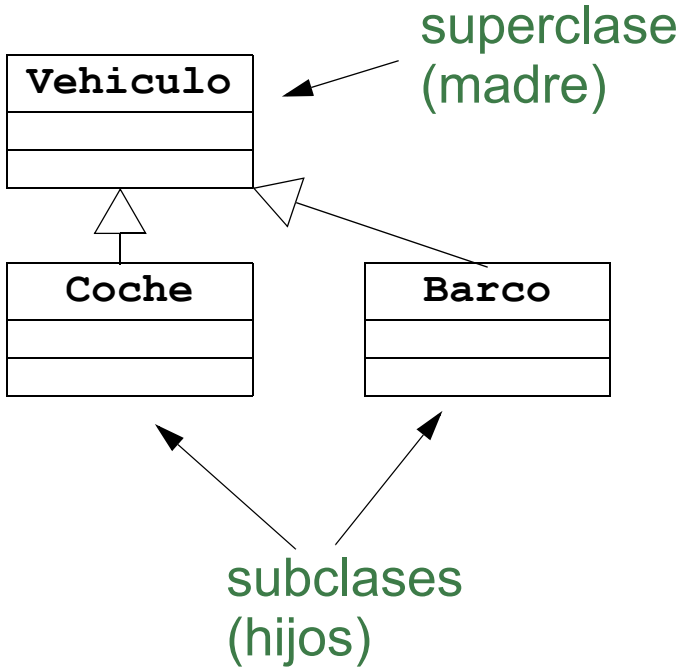
Esto da lugar a una jerarquía:

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

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

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

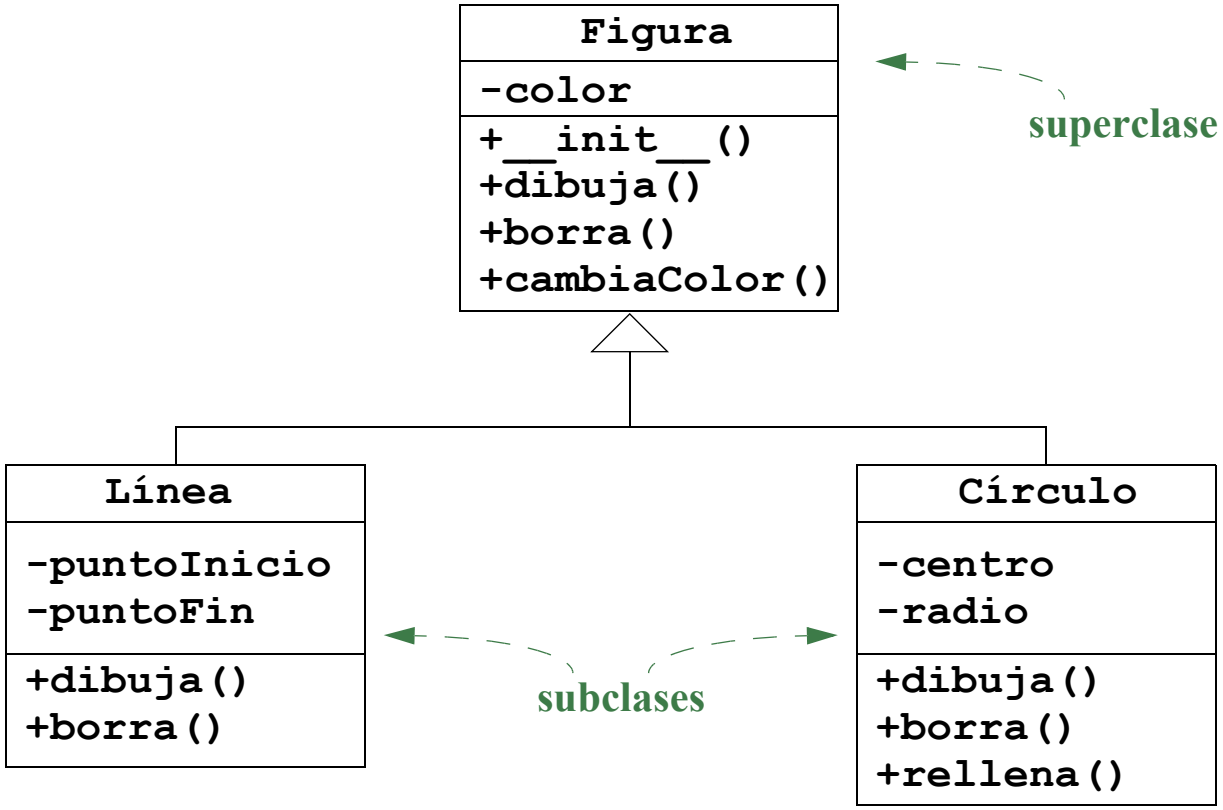
La herencia se puede aplicar múltiples veces

- da lugar a una jerarquía de clases

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

Herencia y Constructores

El constructor de la superclase se hereda

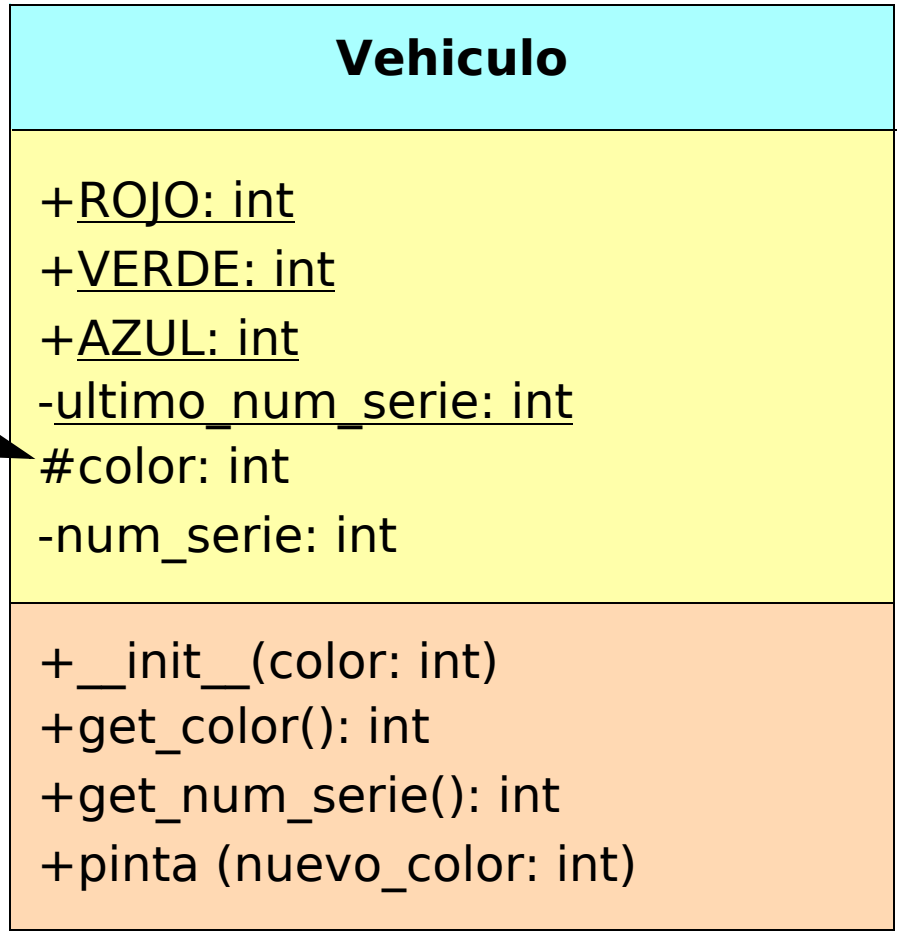
- pero si tenemos atributos nuevos querremos redefinirlo
- en ese caso también será necesario inicializar los atributos de la superclase
- para ello se llama al constructor de la superclase, `super()`, desde el de la subclase

```
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 = ...
```

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 '_'



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
```

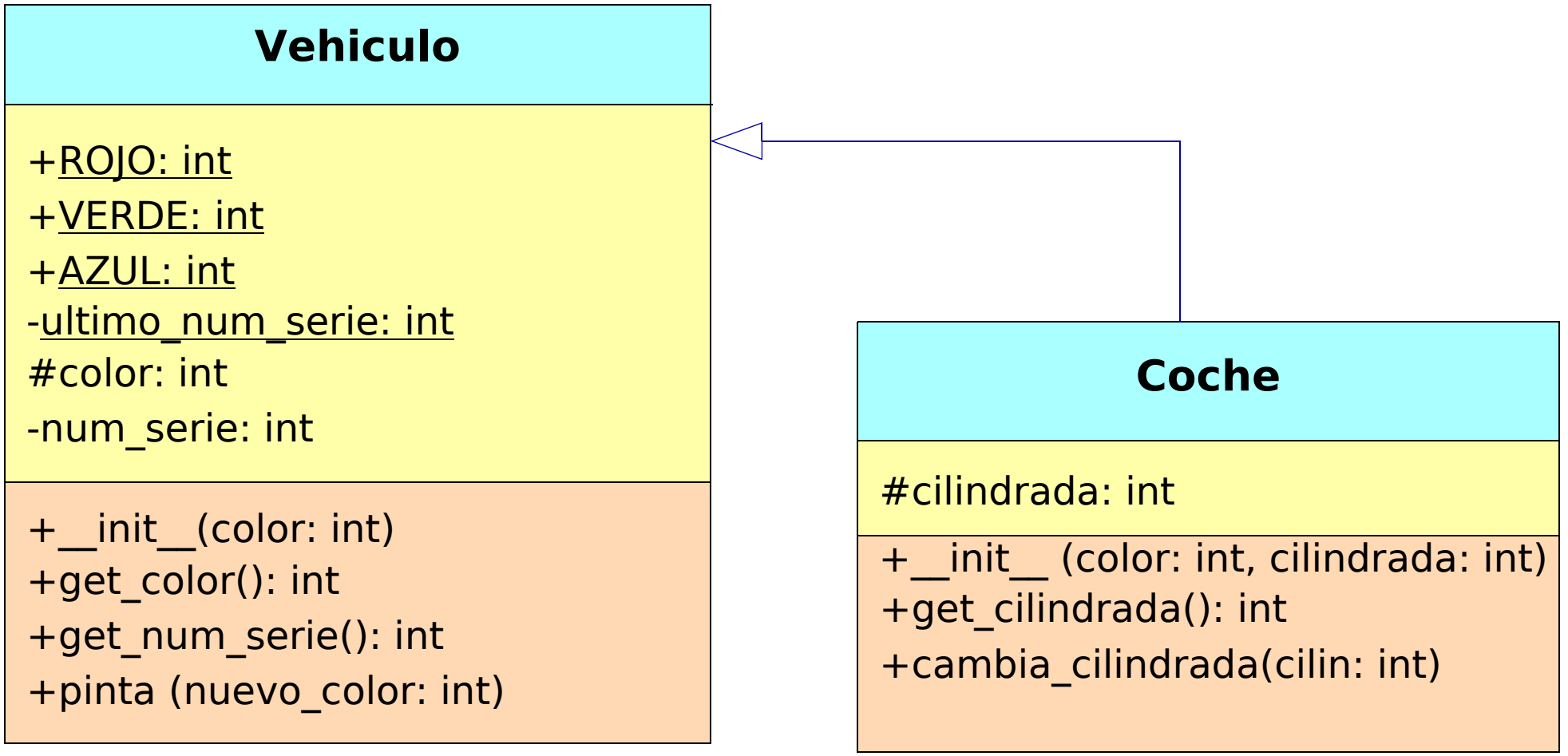
Ejemplo (cont.)

```
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
```

Ejemplo: extensión a la clase Coche



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 __)
    """
```

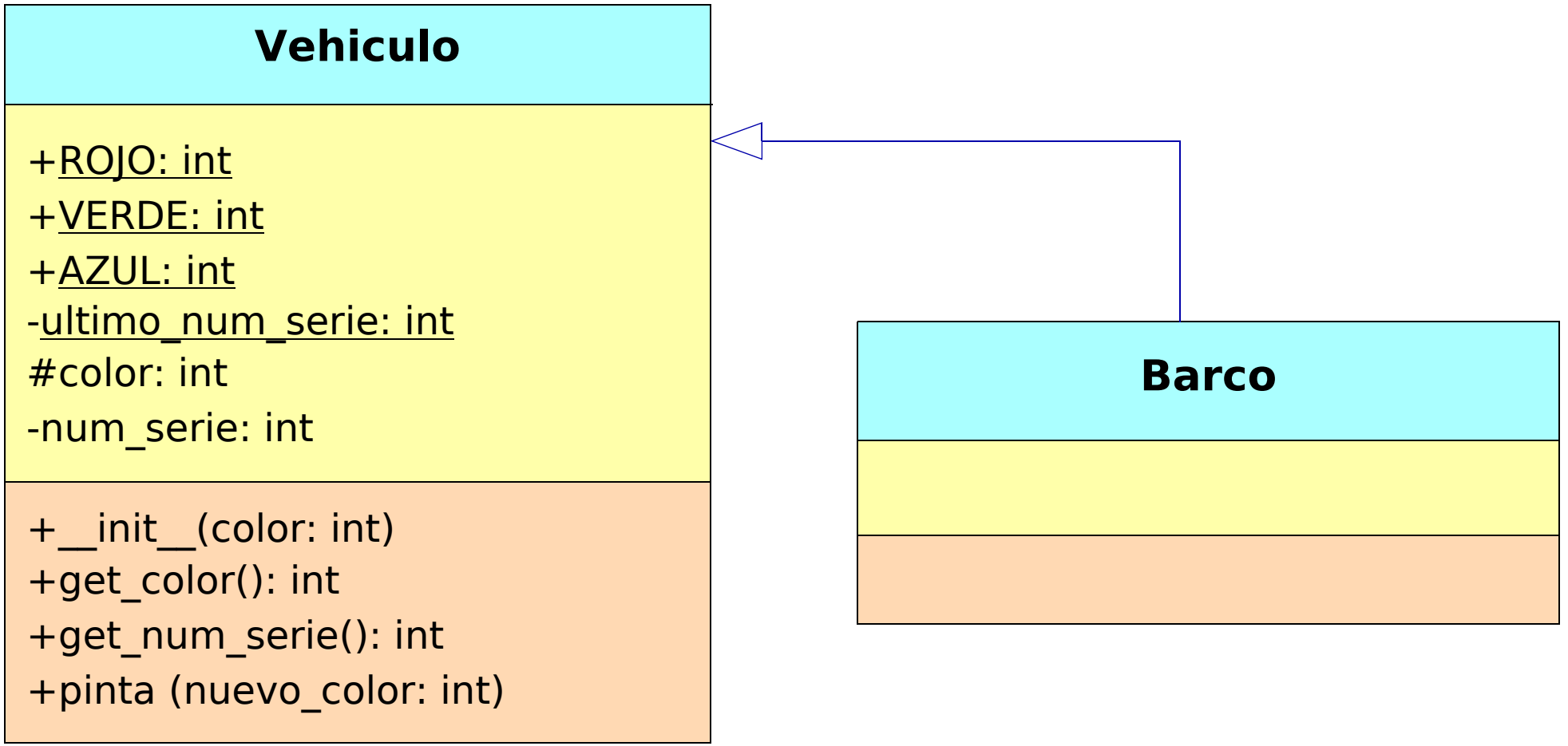
Extensión a la clase Coche (cont.)

```
    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
```

Ejemplo: extensión a la clase Barco



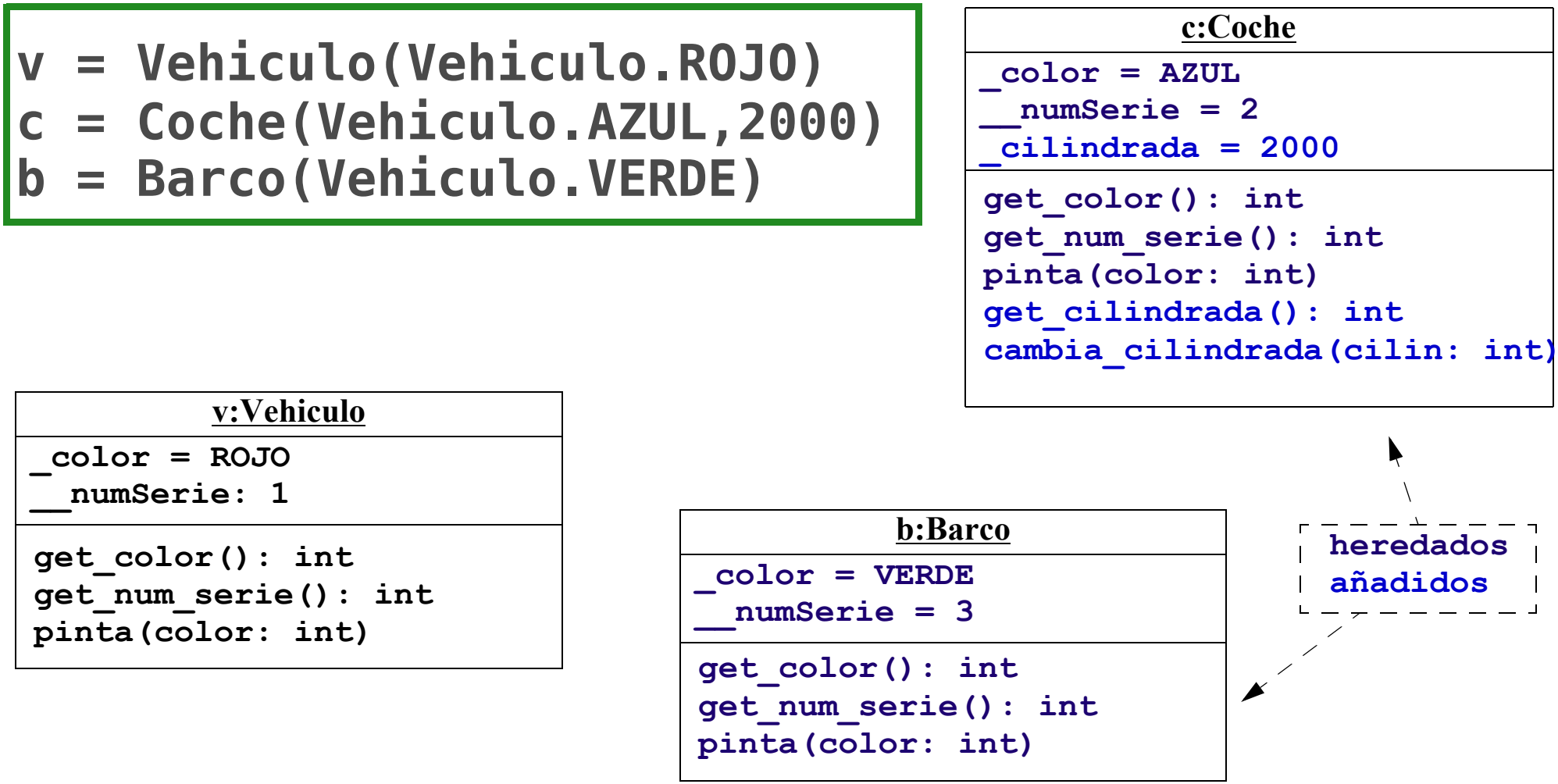
Se puede hacer herencia sin añadir métodos ni atributos

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 __)
    """
```

Ejemplo: objetos y herencia

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



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

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__()}"
```

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

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`

Métodos abstractos: redefinición en el **Coche**

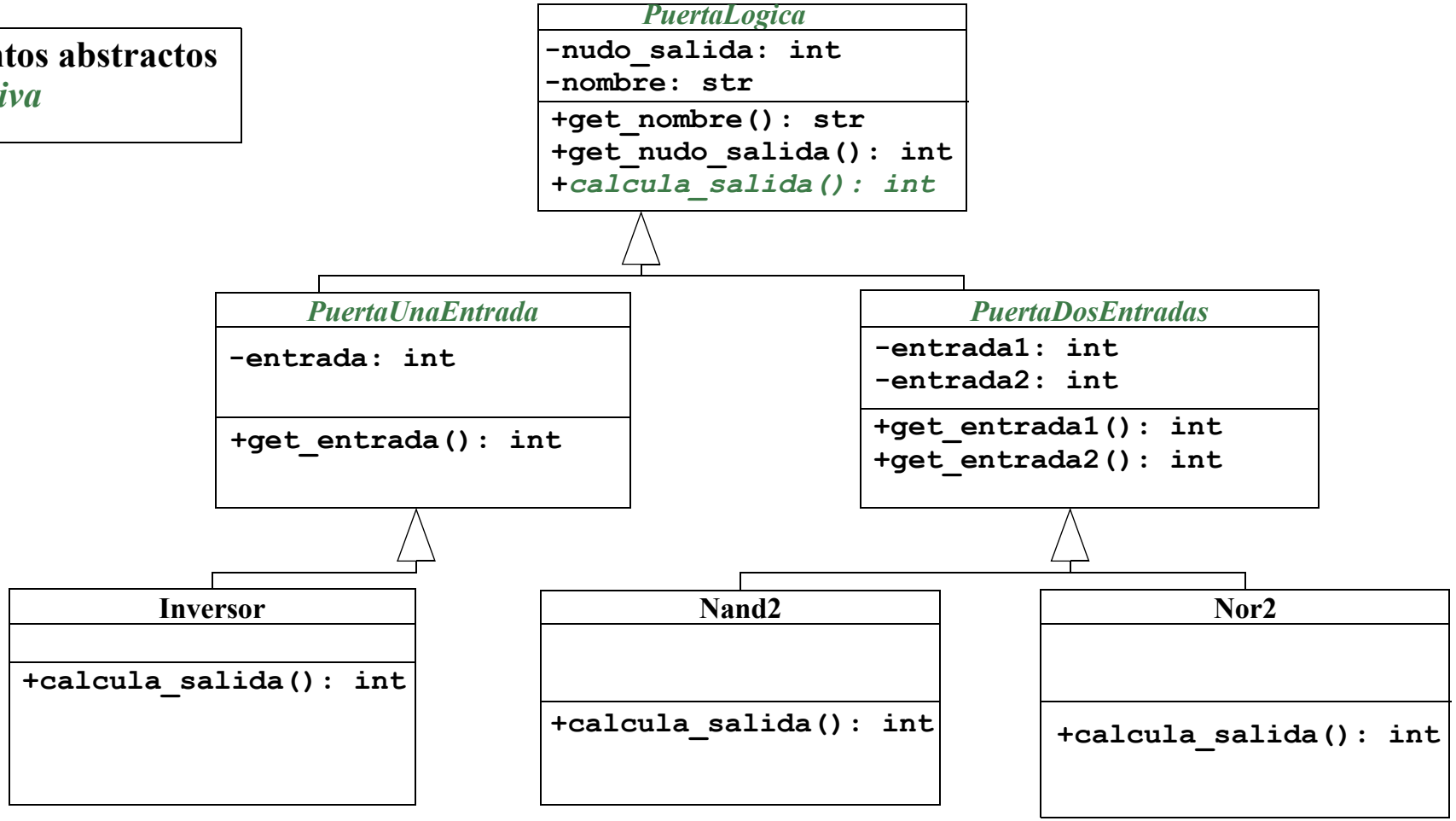
```
class Coche(Vehiculo):
    ...
    def medio_por_el_que_se_mueve(self) -> str:
        """
        Retorna el medio por el que va el coche
        """
        return "Terrestre"
```

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*



8.4. Polimorfismo

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

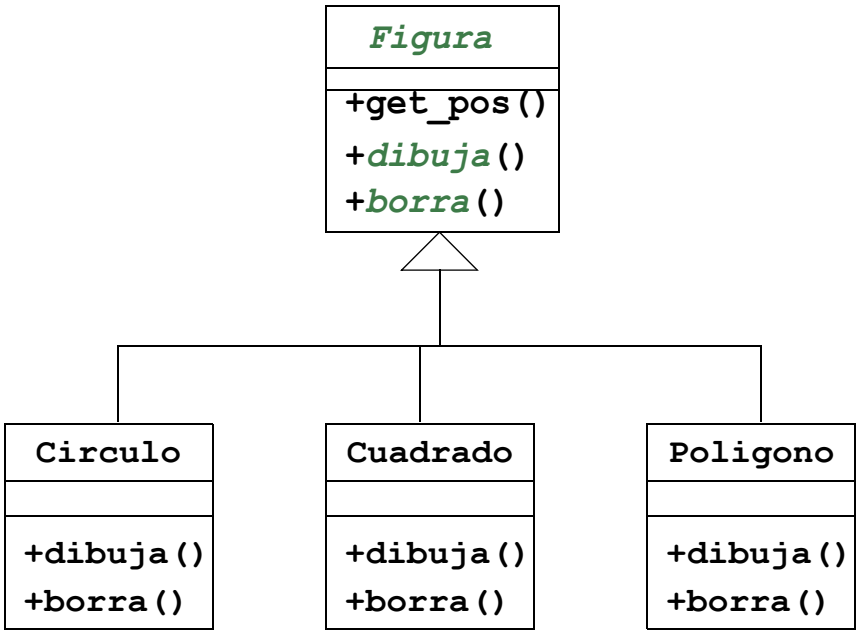
Las ***operaciones polimórficas*** son aquellas que, invocándose de la misma forma, hacen funciones similares con objetos de tipos diferentes

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

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

Todas ellas con las operaciones:

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



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

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**

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*

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()`

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