

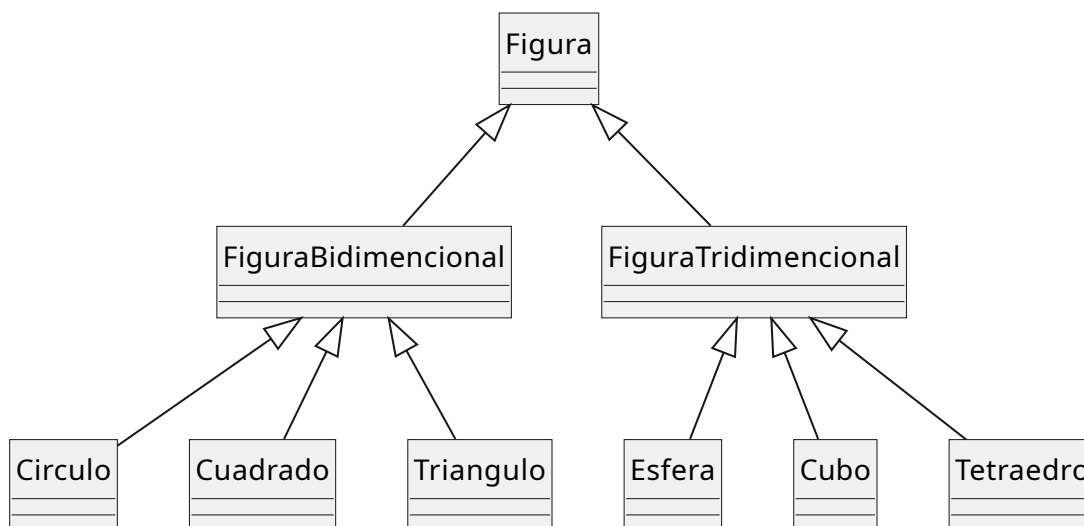
1. Introducción

Una de las características principales de la programación OO es la **herencia**, que permite reutilizar código al crear una nueva clase basándose en los miembros de una clase existente (*herencia simple*) o incluso de varias (*herencia múltiple*). Gracias a la herencia, los programadores ahorran tiempo durante el desarrollo, al reutilizar software probado y depurado de alta calidad. Esto también aumenta la probabilidad que un sistema se implemente con eficiencia.

A la clase previamente existente se la conoce como **superclase** (o *clase base*), y a las nuevas clases se las conoce como **subclases** (o *clases derivadas*). Cada subclase puede convertirse en la superclase de futuras subclases.

Una subclase añade atributos y métodos propios, por lo que resulta más específica que su superclase: describe un conjunto de objetos más especializado. Así, la subclase hereda los comportamientos de la superclase y, a la vez, define características adicionales. Por este motivo a la herencia se la llama a veces **especialización**.

La *superclase directa* es la clase de la cual la subclase hereda en forma explícita. Una *superclase indirecta* es cualquier clase que se encuentre arriba de la superclase directa en la **jerarquía de clases**, que es la jerarquía que define las relaciones de herencia entre las clases. Por ejemplo, en la siguiente jerarquía de clases, **FiguraBidimensional** es una superclase directa de **Circulo**, **Cuadrado** y **Triangulo**, mientras que **Figura** es una superclase indirecta de **Circulo**, **Cuadrado**, **Triangulo**, **Esfera**, **Cubo** y **Tetraedro**.



Cada flecha en la jerarquía resulta de realizar una **generalización** y representa una relación “es un”. Por ejemplo, al seguir las flechas en esta jerarquía de clases, podemos decir que un **Triangulo** es una **FiguraBidimensional**, además de una **Figura**; de igual modo, una **Esfera** es una **FiguraTridimensional** y también una **Figura**.

En **Python**, la jerarquía de clases empieza con la clase **object**, de la cual heredan *todas* las clases, ya sea de forma directa o indirecta.

En **Python**, la herencia se indica colocando entre paréntesis el nombre (o los nombres) de las clases de las cuales se hereda, separados por comas, al definir la clase derivada.

Programa 1: Herencia directa

```
1 class Animal: # herencia implícita de object
2     pass
3
4 class Humanoide(object): # herencia explícita (redundante)
5     pass
6
7 class HombreLobo(Animal, Humanoide):
8     pass
9
10 if __name__ == "__main__":
11     print(object.__bases__)
12     print(Animal.__bases__)
13     print(Humanoide.__bases__)
14     print(HombreLobo.__bases__)
```

El atributo especial `__bases__`, que poseen todos los *objetos clase* devuelve una *tupla* que contiene las clases de las cuales heredan directamente (*herencia directa*). La salida del programa anterior será la siguiente:

```
()
(<class 'object'>,)
(<class 'object'>,)
(<class '__main__.Animal'>, <class '__main__.Humanoide'>)
```

Como se observa, la clase **object** es la única que devuelve una tupla vacía (no hereda de nadie); las demás devuelven tuplas con al menos un elemento (**object**) y luego las otras se ven con el formato `<módulo>.Clase`. Por *default* el script que ejecutemos, es decir el módulo (*archivo.py*) que se ejecuta, recibe el nombre interno “`__main__`” y se puede consultar a través de la variable “`__name__`”.

En **Python**, todos son objetos, incluso los módulos, clases y funciones. Los archivos

.py tienen algunos atributos especiales a los que podemos acceder.

Programa 2: Atributos de los módulos

```
1 # main.py
2 """
3 Módulo principal, muestra variables
4 """
5 if __name__ == "__main__":
6     print(__name__)      # __main__
7     print(__package__)   # None
8     print(__doc__)       # "Modulo principal, muestra variables"
```

2. Reaprovechamiento

Las clases nuevas pueden heredar de las clases disponibles en bibliotecas de clases. Muchas organizaciones desarrollan sus propias bibliotecas de clases y también pueden aprovechar otras de terceros. Es probable que algún día, la mayoría del software nuevo se construya a partir de componentes reutilizables estándar, como ocurre con la industria automotriz o el hardware informático. Esto facilitará el desarrollo de software más potente, abundante y económico.

En el siguiente ejemplo, crearemos una nueva clase **MiString** que hereda todas las funcionalidades de **str**.

Programa 3: Clase **MiString** que hereda de **str**

```
1 class MiString(str):
2     """
3     Clase MiString hereda de str
4     """
5     def cantidad_vocales(self) -> int:
6         """Devuelve la cantidad de vocales en un texto
7
8         Returns:
9             int: cantidad de vocales
10        """
11        vocales = "aeiouáéíóú"
12        cant = 0
13        # str admite recorrer letra a letra con for
14        for letra in self.lower():
15            if letra in vocales:
16                cant += 1
17        return cant
18
```

```
19 if __name__ == '__main__':  
20     mstr = MiString("hola, este es un texto.")  
21  
22     print(mstr.cantidad_vocales()) # método de MiString()  
23     print(mstr.upper()) # método heredado de str()
```

La clase `MiString` extiende la clase `str` agregando el método `cantidad_vocales()`, pero conserva todas las propiedades de un `str` y tiene todos sus métodos disponibles. Puede recorrerse utilizando un `for` (línea 14) y acceder a métodos, como por ejemplo `lower()` y `upper()` (línea 23). La salida de este programa será como se ve a continuación.

```
8  
HOLA, ESTE ES UN TEXTO.
```

2.1. Inicializadores

Los inicializadores no se heredan automáticamente. Si una clase derivada define su propio `__init__`, no se invoca el inicializador de la superclase a menos que lo hagamos explícitamente. En consecuencia, si la superclase declaraba atributos en su `__init__`, la derivada no los recibirá.

Para ello, existe el objeto especial `super` que permite invocar el inicializador de la clase base:

```
1 class Animal:  
2     def __init__(self, nombre : str) -> None:  
3         self.__nombre = nombre  
4  
5     def nombrar(self) -> None:  
6         print(f"Soy un(a) {self.__nombre}")  
7  
8 class Mamifero(Animal):  
9     def __init__(self, nombre : str, patas : int) -> None:  
10        super().__init__(nombre)  
11        self.__patas = patas  
12  
13    def amamantar(self) -> None:  
14        self.nombrar()  
15        print("Y estoy amamantando")  
16  
17 if __name__ == "__main__":  
18     m1 = Mamifero("perro", 4)  
19     m2 = Mamifero("nutria", 4)  
20     m1.amamantar()  
21     m2.amamantar()
```

En la *línea 16* el inicializador de **Mamifero** llama al inicializador de la clase base **Animal** con el parámetro **nombre**. Como se observa, **Mamifero** puede acceder al método **nombrar** enviando el mensaje en la *línea 14* a través de la instancia **self**, pero también podríamos acceder a través del objeto **m1** o **m2**. Sin embargo, el atributo **__nombre** de **Animal** no es accesible (a priori) por **Mamifero**.

La salida de este programa sería:

```
Soy un(a) perro
Y estoy amamantando
Soy un(a) nutria
Y estoy amamantando
```

3. Sobrescritura

Podemos reemplazar los métodos heredados de una superclase, haciendo que la derivada tenga un comportamiento distinto al heredado, a esto se lo conoce como sobrescritura de métodos (*method overriding*). En el caso de querer *extender* la funcionalidad del método en lugar de reemplazarlo, esto podría lograrse nuevamente a través de **super**.

Programa 4: Sobrescritura de un método heredado

```
1 class Animal:
2     def __init__(self, nombre : str) -> None:
3         self.__nombre = nombre
4     def nombrar(self) -> None:
5         print(f"Soy un(a) {self.__nombre}")
6
7 class Mamifero(Animal):
8     def __init__(self, nombre : str, patas : int) -> None:
9         super().__init__(nombre)
10        self.__patas = patas
11
12    def amamantar(self) -> None:
13        self.nombrar()
14        print("Y estoy amamantando")
15
16 class Perro(Mamifero):
17     def __init__(self, nombre : str, ladrido : str) -> None:
18         super().__init__("perro", 4)
19         self.__nombre = nombre
20         self.__ladrido = ladrido
21
22    def ladrar(self) -> None:
23        print(self.__ladrido)
```

```
24
25     def amamantar(self) -> None:
26         print(f'Me llamo {self.__nombre}')
27         super().amamantar()
28
29     if __name__ == "__main__":
30         f = Perro("Firulais", "¡Guau! ¡Guau!")
31         f.ladrear()
32         f.amamantar()
```

Dentro del método `amamantar` de la clase `Perro`, en la *línea 28*, se envía el mensaje `amamantar` utilizando `super()` para ejecutar el método perteneciente a la clase base `Mamifero`. Y la salida que produce este programa es la siguiente.

```
¡Guau! ¡Guau!
Me llamo Firulais
Soy un(a) perro
Y estoy amamantando
```

Se darán más aclaraciones acerca de la sobrescritura en apunte 6 sobre **polimorfismo**.

4. Métodos estáticos vs métodos de clase

En el apunte anterior sobre el **comportamiento de los objetos**, no se expuso una diferencia aparente entre declarar un método de clase y uno estático. Pero, al heredar, el atributo `cls` (referencia a la clase) ofrece ventajas que un método estático no tendría.

Programa 5: Vehículo con método estático

```
1  class Vehiculo:
2      __cantidad_vehiculos = 0
3      def __init__(self, tipo: str) -> None:
4          self.tipo = tipo
5          Vehiculo.__cantidad_vehiculos += 1
6      @staticmethod
7      def cantidad_de_vehiculos():
8          return Vehiculo.__cantidad_vehiculos
9
10     if __name__ == "__main__":
11         # Crear instancias de la clase
12         v1 = Vehiculo("Auto")
13         v2 = Vehiculo("Motocicleta")
14         # Uso del método estático
15         numero_de_vehiculos = Vehiculo.cantidad_de_vehiculos()
16         print(f"Se han creado {numero_de_vehiculos} vehículos")
```

Programa 6: Vehículo con método de clase

```
1 class Vehiculo:
2     __cantidad_vehiculos = 0
3     def __init__(self, tipo : str) -> None:
4         self.tipo = tipo
5         Vehiculo.__cantidad_vehiculos += 1
6     @classmethod
7     def cantidad_de_vehiculos(cls):
8         return cls.__cantidad_vehiculos
9
10 if __name__ == "__main__":
11     # Crear instancias de la clase
12     v1 = Vehiculo("Auto")
13     v2 = Vehiculo("Motocicleta")
14     # Uso del método estático
15     numero_de_vehiculos = Vehiculo.cantidad_de_vehiculos()
16     print(f"Se han creado {numero_de_vehiculos} vehículos")
```

En principio el comportamiento de ambos para el programador parecieran el mismo. Veamos que ocurre si se quisiera llevar un stock de *autos* y *motocicletas*, pero a la vez no perder la cantidad de vehículos totales.

Programa 7: Ventaja del método de clase

```
1 class Vehiculo:
2     _cantidad = 0
3     def __init__(self, tipo : str) -> None:
4         self.tipo = tipo
5         Vehiculo._cantidad += 1
6     @classmethod
7     def cantidad_de_vehiculos(cls):
8         return cls._cantidad
9
10 class Auto(Vehiculo):
11     _cantidad = 0
12     def __init__(self) -> None:
13         super().__init__("Auto")
14         Auto._cantidad += 1
15
16 class Motocicleta(Vehiculo):
17     _cantidad = 0
18     def __init__(self) -> None:
19         super().__init__("Motocicleta")
20         Motocicleta._cantidad += 1
21
22 if __name__ == "__main__":
23     # Crear instancias de la clase
24     v1 = Auto()
25     v2 = Motocicleta()
26     v3 = Auto()
```

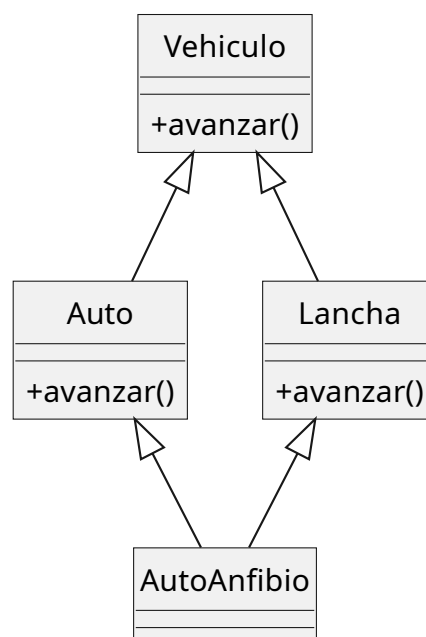
```
27     v4 = Auto()
28
29     # Uso del método de clase
30     cant_vehiculos = Vehiculo.cantidad_de_vehiculos()
31     print(f"Se han creado {cant_vehiculos} vehículos")
32     cant_autos = Auto.cantidad_de_vehiculos()
33     print(f"Se han creado {cant_autos} autos")
34     cant_motos = Motocicleta.cantidad_de_vehiculos()
35     print(f"Se han creado {cant_motos} motos")
```

Debe notarse que para que este ejemplo tenga la funcionalidad esperada, debe utilizar solo un guion bajo (_) para el nombre del atributo de clase. Ya que en este ejemplo es necesario sobrescribir el atributo `_cantidad` de la clase base **Vehiculo**.

Se profundizará sobre este comportamiento en particular en el apunte 5: **Encapsulamiento**.

5. Problema del diamante

En los lenguajes de programación OO, se describe un problema asociado a la herencia múltiple, conocido como el **problema del diamante** (llamado así por la forma que tiene el diagrama de clases en el que surge esta ambigüedad). Este se da cuando una subclase hereda el mismo método de dos superclases diferentes.



Supongamos que la clase **Vehiculo** define un método **avanzar**. Las clases **Auto** y **Lancha** extienden **Vehiculo**: un auto es un vehículo y una lancha también es un vehículo, por lo tanto ambas clases heredan el método **avanzar**. Ahora bien, dado que un vehículo anfibio es un auto y también es una lancha, podría declararse mediante herencia múltiple que **AutoAnfibio** extiende **Auto** y **Lancha** ¿Qué comportamiento debería tener entonces un auto anfibio: el método **avanzar** que hereda de **Auto** o el que hereda de **Lancha**?

En **Python** no existe este problema porque implementa un *orden de resolución de métodos*, que se explica a continuación.

5.1. Orden de resolución de métodos (MRO)

El MRO (*Method Resolution Order*) es el mecanismo que utiliza **Python** para solucionar el problema del diamante anteriormente expuesto. Ante la posibilidad de tener dos métodos heredados con el mismo nombre se ejecuta el de la primer clase que se encuentre en la variable de clase especial `__mro__`. Generalmente, esta será la primera declarada en la herencia (en caso de no sobrescribirla).

Programa 8: MRO con métodos

```
1 class Vehiculo:
2     def __init__(self, tipo: str) -> None:
3         self.__tipo = tipo
4         print(f"Se creo un {self.__tipo}")
5
6     def avanzar(self) -> None:
7         # método no implementado, se implementa en clase derivadas
8         ...
9
10    class Auto(Vehiculo):
11        def __init__(self, tipo: str = "Auto") -> None:
12            super().__init__(tipo)
13        def avanzar(self) -> None:
14            print("Avanza sobre tierra")
15
16    class Lancha(Vehiculo):
17        def __init__(self, tipo: str = "Lancha") -> None:
18            super().__init__(tipo)
19        def avanzar(self) -> None:
20            print("Avanza sobre el agua")
21
22    class AutoAnfibio(Auto, Lancha):
23        def __init__(self, tipo: str = "AutoAnfibio") -> None:
24            super().__init__(tipo)
```

```
25
26 if __name__ == "__main__":
27     aa = AutoAnfibio()
28     aa.avanzar()
29     print(AutoAnfibio.__mro__)
```

En el ejemplo anterior se podrá observar la siguiente salida.

```
Se creo un AutoAnfibio
Avanza sobre tierra
(<class '__main__.AutoAnfibio'>, <class '__main__.Auto'>, <class
 '__main__.Lancha'>, <class '__main__.Vehiculo'>, <class '
object'>)
```

El método que se ejecuta es el de la clase **Auto**, ya que es la primera clase derivada en el `__mro__` que implementa este método.

En caso de necesitar extender las funcionalidades de las clases base podemos realizar la siguiente sobrescritura llamando a los métodos desde el *objeto clase* correspondiente.

```
# ...
class AutoAnfibio(Auto, Lancha):
    def __init__(self, tipo: str = "AutoAnfibio") -> None:
        super().__init__(tipo)

    def avanzar(self) -> None:
        Auto.avanzar(self)
        print("y también...")
        Lancha.avanzar(self)
# ...
```

Esto se conciderará mala práctica, ya que no es escalable, si se agrega otra clase base que también implemente **avanzar**, y se quiera aprovechar esa funcionalidad, o poder tomar decisiones de cual es mejor dependiendo un determinado escenario no es posible hacerlo. Para ello se aplican otras estrategias de diseño que se analizarán en el apunte 6: **Polimorfismo**.

5.2. MRO y super

Esto también influye en qué clase utilizará el objeto **super**, este seguirá el orden en el **MRO**. Observe el siguiente ejemplo.

Programa 9: cadena de `__init__` del MRO

```
1 class A:
2     def __init__(self) -> None:
3         print("Soy A")
4
5 class B(A):
6     def __init__(self) -> None:
7         super().__init__()
8         print("También soy B")
9
10 class C(A):
11     def __init__(self) -> None:
12         super().__init__()
13         print("También soy C")
14
15 class D(B,C):
16     def __init__(self) -> None:
17         super().__init__()
18         print("También soy D")
19
20 if __name__ == "__main__":
21     d = D()
```

Cuando se crea una instancia de D, se ejecuta lo siguiente:

1. Instanciación de D:

- Llama a `D.__init__()`.
- `D.__init__()` llama a `super().__init__()`, que sigue el MRO.

2. MRO para D:

- Llama a `B.__init__()`.
 - `B.__init__()` llama a `super().__init__()`, que sigue el MRO.
- `super()` en B sigue el MRO, que llama a `C.__init__()`.
 - `C.__init__()` llama a `super().__init__()`, que sigue el MRO.
- `super()` en C sigue el MRO, que llama a `A.__init__()`.

3. Salida:

```
Soy A
También soy C
También soy B
También soy D
```

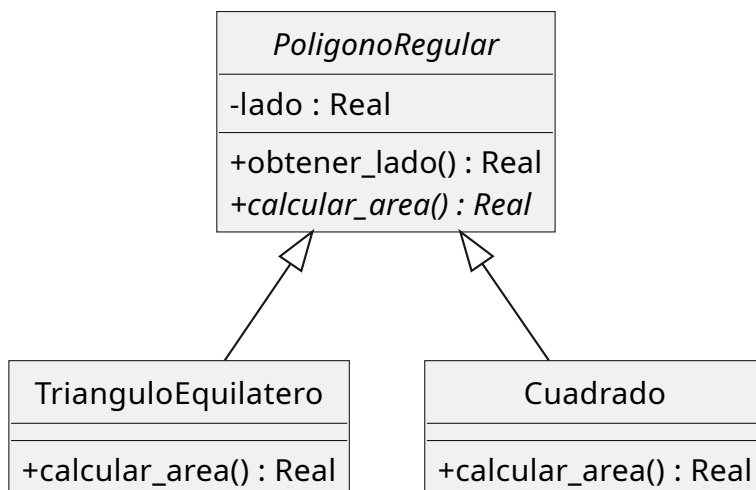
Si bien, **Python** ofrece estos mecanismos para solventar este problema que presenta la herencia múltiple, en general, su uso es desaconsejado, para evitar este tipo de problemas. Por ejemplo, no es lo mismo primero heredar de una *clase A* y luego de una *clase B* que al revés y puede acarrear errores de funcionamiento (*bugs*) difíciles de depurar.

6. Clases Abstractas e Interfaces

En **Python** como en otros lenguajes (C++, por ejemplo), no existe la figura de *Interface* formalmente, y en este caso tampoco existen las *clases abstractas* naturalmente. Las **clases abstractas** son aquellas que no implementan algún método y, por tal, no podría instanciarse un objeto de esa clase, ya que *no está completa*. Esto se ve en los diagramas de clase **UML** con el nombre de la clase *en cursiva/itálicas*.

6.1. Clases abstractas

Las clases que heredan de clases abstractas, deben estar obligados a implementar los métodos que son abstractos dentro de las mismas, si no lo hacen, entonces también se consideran abstractas. Hasta no ser implementadas, no puede instanciarse un objeto de esa clase. En **Python** se creó un mecanismo para lograr este comportamiento, y es heredar de *Abstract Base Class* (**ABC**) del módulo **abc**. Las clases que extiendan **ABC** tendrán el comportamiento esperado.



En el ejemplo que se observa, la clase *PoligonoRegular* es abstracta, porque tiene el método abstracto *calcular_area*. Las clases concretas derivadas *Cuadrado* y *TrianguloEquilatero* implementan ese método según corresponde.

Programa 10: Clase abstracta

```
1 import math
2 from abc import ABC, abstractmethod
3
4 class PoligonoRegular(ABC):
5     def __init__(self, lado: float) -> None:
6         self.__lado = lado
7
8     def obtener_lado(self) -> float:
9         return self.__lado
10
11     @abstractmethod
12     def calcular_area(self) -> float:
13         ...
14
15 class Cuadrado(PoligonoRegular):
16     def calcular_area(self) -> float:
17         return self.obtener_lado()**2
18
19 class TrianguloEquilatero(PoligonoRegular):
20     def calcular_area(self) -> float:
21         return self.obtener_lado()**2 * math.sqrt(3) / 4
22
23 if __name__ == "__main__":
24     c = Cuadrado(5)
25     t = TrianguloEquilatero(5)
26
27     print(f"Area del cuadrado {c.calcular_area()}")
28     print(f"Area del triangulo eq {t.calcular_area()}")
```

En este caso, el inicializador por omisión ya hace el llamado a `super().__init__()` con la cantidad de parámetro necesarios al construir la clase derivada, por ello podemos obviarlo. Solo se hace necesario si queremos instanciar nuevos atributos en la clase derivada o agregar algún tipo de funcionalidad. Si se quisiera instanciar un objeto del tipo `PoligonoRegular`, **Python** nos devolverá el siguiente error.

```
>>> pr = PoligonoRegular(5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class PoligonoRegular with
abstract method calcular_area
```

Nota

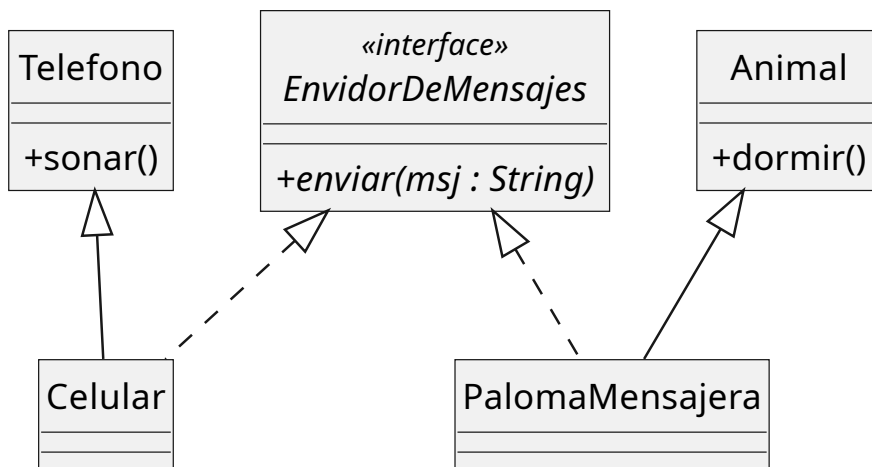
En **Python** existen una palabra clave (`pass`) y una elipsis (`...`) para indicar que el cuerpo de una sentencia o función no tienen implementación. Pero hay una sutil diferencia. Cuando se utiliza `pass`, el programador deliberadamente deja claro que no hay implementación o no hay un cuerpo para la sentencia, método o función declarada. En cambio, al utilizar `...` (elipsis) se está indicando que habrá una implementación; que hace falta completar código allí. Es la opción de preferencia a utilizar para los métodos abstractos que deben ser implementados por las subclases.

6.2. Interfaces

Si bien la herencia múltiple está desaconsejada, el no utilizarla podría resultar limitante para ciertos diseños, ya que es una pérdida importante de potencialidad y funcionalidad. Además que da poca flexibilidad, podríamos tener clases que tengan algoritmos importantes para dos clases de índoles completamente distintas.

Imagine que se quiere implementar una clase `Celular` y otra `PalomaMensajera`, y lo único que tienen en común es que ambas clases pueden enviar mensajes (cada una a su modo). Pero a su vez `PalomaMensajera` es un `Animal` y `Celular` es un `Telefono`. En estos casos, la solución es utilizar una clase completamente abstracta que tampoco implemente ningún atributo, solo declare funcionalidad. Si cumple con esos requisitos, entonces diremos que la clase en realidad es una *interfaz* y en los diagramas de **UML** figuran con su

nombre en cursiva/itálicas y con el arquetipo «*interface*».



Como los métodos declarados en las interfaces (por definición) son abstractos, en las **realizaciones**, no se heredan implementaciones de métodos, sino la responsabilidad de implementarlos. De esta manera, en el ejemplo, la **PalomaMensajera** podrá **dormir** (método heredado de **Animal**) y el **Celular** podrá **sonar** (heredado de **Telefono**), pero a la vez, ambos podrán **enviar** mensajes. En código se vería de la siguiente manera.

Programa 11: Realización con ABC

```
1  from abc import ABC, abstractmethod
2
3  class EnviadorDeMensajes(ABC):
4      @abstractmethod
5      def enviar(self, msj: str) -> None:
6          ...
7
8  class Animal:
9      def dormir(self) -> None:
10         print("ZZzzz...")
11
12  class PalomaMensajera(Animal, EnviadorDeMensajes):
13      def enviar(self, msj: str) -> None:
14         print("Llega volando: " + msj)
15
16  class Telefono:
17      def sonar(self) -> None:
18         print("Ring! Ring!")
19
20  class Celular(Telefono, EnviadorDeMensajes):
21      def enviar(self, msj: str) -> None:
22         print("Ring! Ring! : " + msj)
23
```

```
24 if __name__ == "__main__":
25     print("El celular:")
26     cel = Celular()
27     cel.sonar()
28     cel.enviar("Hola")
29     print("La paloma:")
30     pal = PalomaMensajera()
31     pal.enviar("Hola")
32     pal.dormir()
```

Note que a diferencia de las clases comunes, que usualmente **son nombradas con sustantivos**, las interfaces **suelen ser gerundios** (terminados en *-ando* y *-endo*) u otro nombre que indique claramente la acción o acciones que las clases implementando la interfaz estarán obligadas a realizar.

Las interfaces deben interpretarse como contratos o compromisos cuyas clases que *la realicen* deben cumplir implementando los métodos allí declarados.

7. Referencias

Para más información puede consultar los siguientes enlaces:

- Documentación oficial: <https://docs.python.org/3/tutorial/classes.html>
- UBA – Algoritmos I: <https://algoritmoslrw.ddns.net/>
- Interprete Visual de Python: <http://www.pythontutor.com/visualize.html>