

PREGRADO



UNIDAD 3 | SEMANA 9

# PROPIEDADES, DEPENDENCIA, ASOCIACIÓN, CARDINALIDAD Y NAVEGABILIDAD

1ACC0201 | Programación Orientada a Objetos





# Logro

Al término de la sesión, el alumno comprenderá el uso de propiedades como una forma de encapsulamiento, así como los conceptos de asociación, cardinalidad y navegabilidad entre clases.





## Principio 2

# Encapsulamiento - Propiedades



# Encapsulamiento - Propiedades

## Propiedades de acceso

- ▶ Las propiedades de acceso son una estrategia que permite gestionar los atributos como si fueran públicos, pero incorporando validaciones u otros procesos lógicos en su manipulación.
- ▶ Esta estrategia se basa en el uso de métodos privados para la lectura (getters) y asignación (setters).
- ▶ La definición (o ausencia) de estos métodos determina cómo se gestiona el acceso a los atributos.
- ▶ Por ejemplo, en el código proporcionado, el atributo `__largo` es de lectura y escritura, mientras que el atributo `__ancho` solo puede ser asignado.

```
class Rectangulo:
    #Constructor
    def __init__(self, largo, ancho):
        #Atributos de instancia
        self.__largo = largo
        self.__ancho = ancho

    #Métodos de acceso
    def __get_largo(self):
        return self.__largo

    def __set_largo(self, largo):
        self.__largo = largo

    largo = property(__get_largo, __set_largo)

    def __set_ancho(self, ancho):
        self.__ancho = ancho

    ancho = property(fset = __set_ancho)

    #Métodos de instancia
    def area(self):
        return self.__largo * self.__ancho
```



# Encapsulamiento - Propiedades

## Decorador @property

- ▶ De manera similar a las propiedades de acceso, el decorador @property permite gestionar los atributos como si fueran públicos, pero con la capacidad de incluir validaciones u otros procesos lógicos en su manipulación.
- ▶ Este decorador facilita la definición de los métodos que actuarán como get y set para un atributo. Por defecto, el decorador se aplica al método que funcionará como get.
- ▶ Para definir el método set, se utiliza @seguido del nombre de la propiedad y la palabra setter, como se muestra en el ejemplo.

```
class Rectangulo:
    #Constructor
    def __init__(self, largo, ancho):
        #Atributos de instancia
        self.__largo = largo
        self.__ancho = ancho

    #Métodos de acceso
    @property
    def largo(self):
        return self.__largo

    @largo.setter
    def largo(self, largo):
        self.__largo = largo

    @property
    def ancho(self):
        return self.__ancho

    @ancho.setter
    def ancho(self, ancho):
        self.__ancho = ancho

    #Métodos de instancia
    def area(self):
        return self.__largo * self.__ancho
```



# Encapsulamiento

## Atributos de clase

- ▶ Los atributos de clase son aquellos que se mantienen a nivel de la clase, compartiendo un valor común entre todas las instancias (objetos) de la misma.
- ▶ Estos atributos suelen utilizarse como contadores, acumuladores o para almacenar datos compartidos.
- ▶ En Python, los atributos de clase se definen fuera del constructor (`__init__`) y de cualquier otro método de la clase.
- ▶ El acceso a estos atributos se realiza a través de la clase misma, no de sus instancias.

```
class Persona:
    #Atributos de clase
    registros = 0

    def __init__(self, nombre, edad):
        #Atributos de instancia
        self.nombre = nombre
        self.edad = edad

    #Incremento del atributo de clase
    Persona.registros += 1

persona1 = Persona("Juan Perez", 50)
print(Persona.registros)

persona2 = Persona("Gianluca Lapadula", 30)
print(Persona.registros)
```



# Encapsulamiento

## Atributos de clase

- ▶ Los atributos de clase también pueden ser privados. En este caso, es necesario definir métodos de acceso para interactuar con ellos.
- ▶ Para que un método pueda acceder a un atributo de clase, debe estar decorado con **@classmethod**, como se muestra en el ejemplo.
- ▶ Estos métodos, conocidos como **métodos de clase**, deben recibir una referencia a la clase.
- ▶ Por convención, esta referencia se denomina **cls**, de manera similar a cómo se utiliza **self** en los métodos de instancia.
- ▶ La referencia **cls** apunta a la clase misma, permitiendo el acceso y manipulación de sus atributos y métodos a nivel de clase.

```
class Persona:
    # Atributos de clase
    __registros = 0

    def __init__(self, nombre, edad):
        # Atributos de instancia
        self.nombre = nombre
        self.edad = edad

    # Incremento del atributo de clase
    Persona.__registros += 1

    @classmethod
    def get_registros(cls):
        return cls.__registros

    @classmethod
    def set_registros(cls, registros):
        cls.__registros = registros

# Programa Principal
persona1 = Persona("Juan Perez", 50)
print(Persona.get_registros())
Persona.set_registros(5)

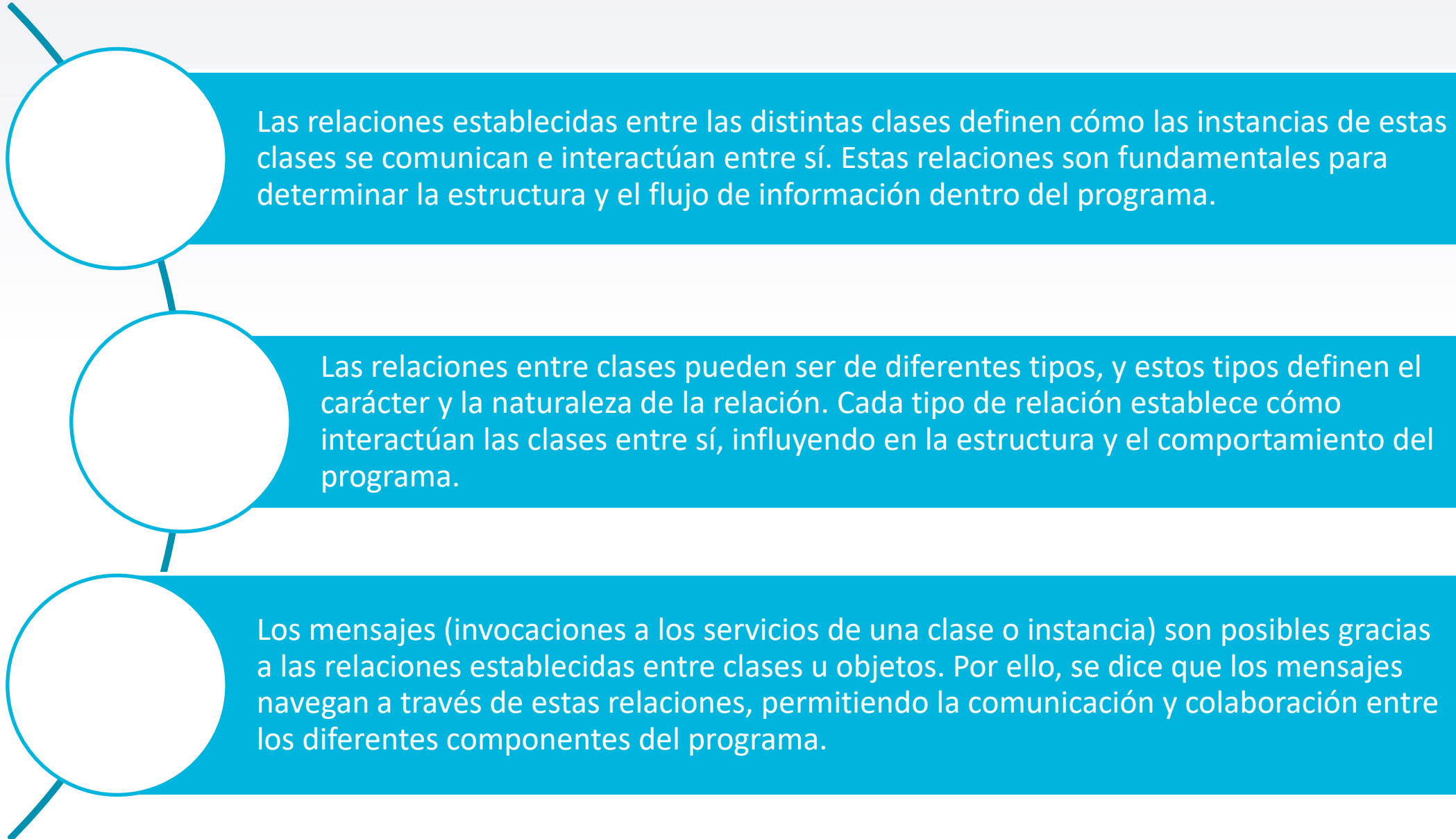
persona2 = Persona("Gianluca Lapadula", 30)
print(Persona.get_registros())
```



# Relaciones entre clases



# Relaciones entre clases



The diagram consists of three white circles arranged vertically on the left side of the slide. Each circle is connected to a blue rectangular text box on the right by a thin black line. The top circle is connected to the first text box, the middle circle to the second, and the bottom circle to the third. The text boxes contain information about class relationships.

Las relaciones establecidas entre las distintas clases definen cómo las instancias de estas clases se comunican e interactúan entre sí. Estas relaciones son fundamentales para determinar la estructura y el flujo de información dentro del programa.

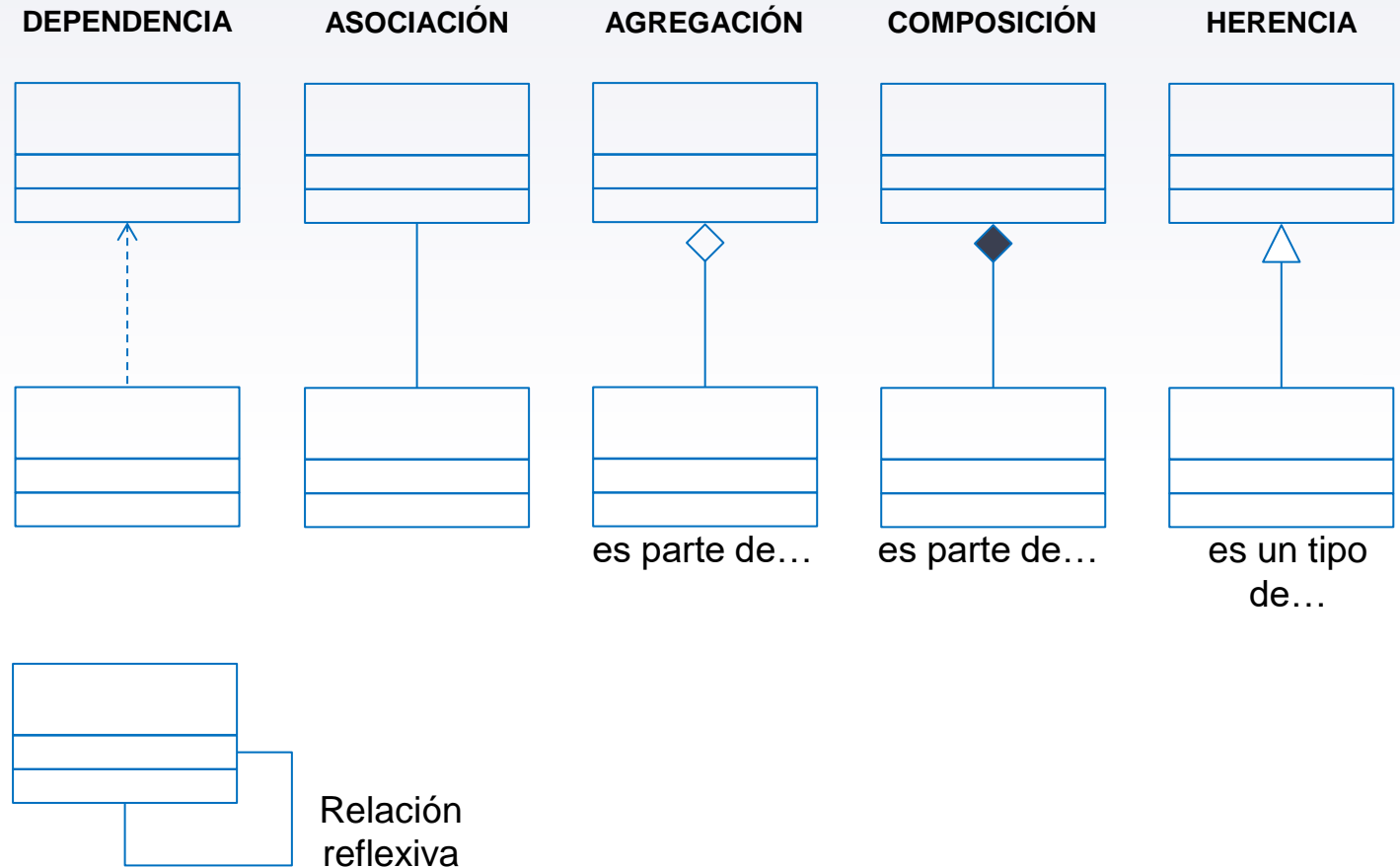
Las relaciones entre clases pueden ser de diferentes tipos, y estos tipos definen el carácter y la naturaleza de la relación. Cada tipo de relación establece cómo interactúan las clases entre sí, influyendo en la estructura y el comportamiento del programa.

Los mensajes (invocaciones a los servicios de una clase o instancia) son posibles gracias a las relaciones establecidas entre clases u objetos. Por ello, se dice que los mensajes navegan a través de estas relaciones, permitiendo la comunicación y colaboración entre los diferentes componentes del programa.



# Relaciones entre clases: Tipos

- ▶ En la POO, las relaciones entre clases representan dependencias que definen cómo interactúan las clases entre sí.
- ▶ Estas dependencias pueden darse entre dos o más clases (el caso más frecuente) o, en situaciones menos comunes, de una clase consigo misma.
- ▶ Este último caso se conoce como relación reflexiva y es particularmente útil en contextos donde los objetos de una clase necesitan establecer conexiones entre sí, como en estructuras jerárquicas.
- ▶ Gráficamente, las relaciones se representan mediante líneas que conectan las clases involucradas.
- ▶ La forma y el estilo de estas líneas varían según el tipo de relación, permitiendo una identificación clara y precisa en diagramas UML.

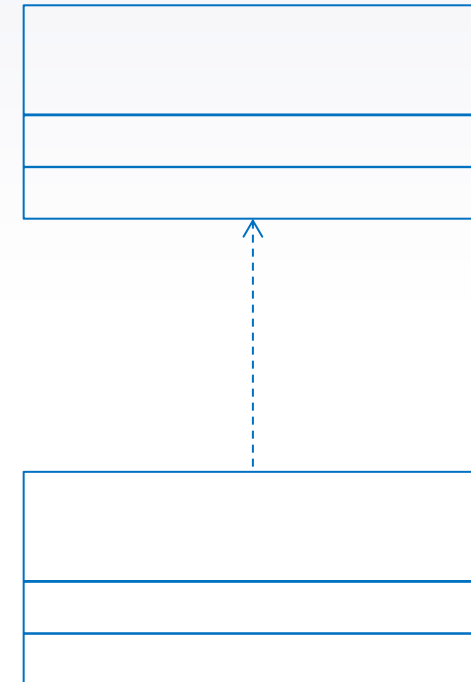




# Relación de Dependencia

- ▶ Una **relación de dependencia** es un tipo de relación donde una clase depende **temporalmente** de otra para realizar una tarea específica.
- ▶ Esta dependencia ocurre cuando una clase **utiliza objetos de otra clase** como parámetros en sus métodos, pero no mantiene una referencia permanente a ellos.
- ▶ La dependencia es una **relación débil y transitoria**, ya que no implica una conexión estructural entre las clases.
- ▶ **Características Clave:**
  - **Temporalidad:** La dependencia existe solo durante la ejecución de un método o una operación específica.
  - **No hay Referencia Permanente:** La clase que depende no almacena una referencia a la clase de la que depende.
  - **Representación en UML:** Se representa con una flecha discontinua que apunta desde la clase dependiente hacia la clase de la que depende.

## DEPENDENCIA

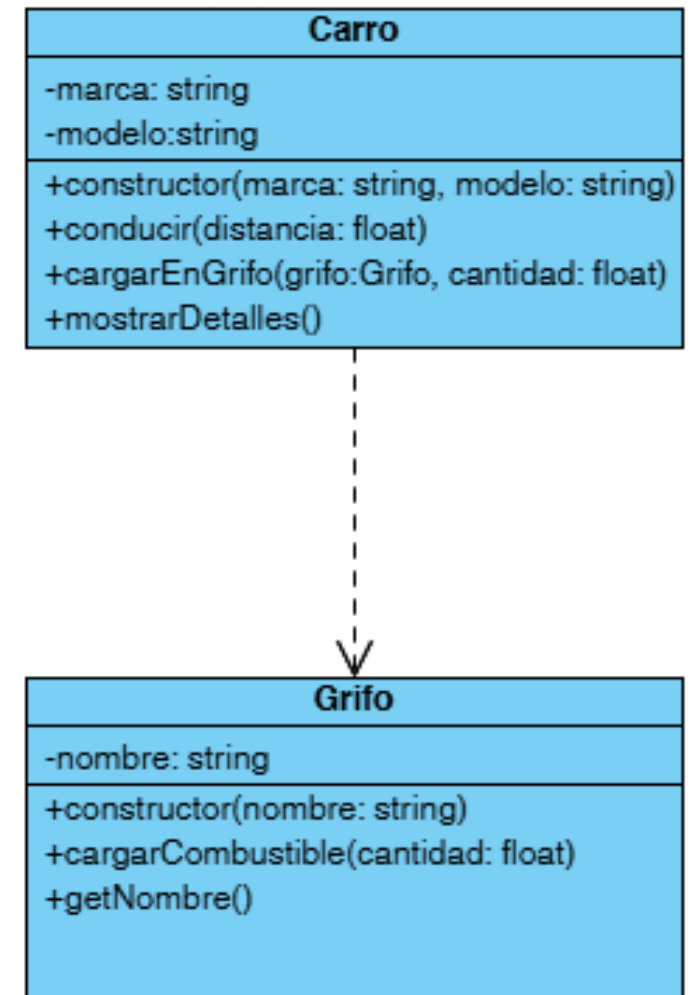




## Ejemplo dependencia

```
class Grifo:
    def __init__(self, nombre):
        self.__nombre = nombre

    def cargarCombustible(self, cantidad):
        print(f"Cargando {cantidad} litros de combustible en "
              f"{self.__nombre}...")
        )
```



# Ejemplo dependencia

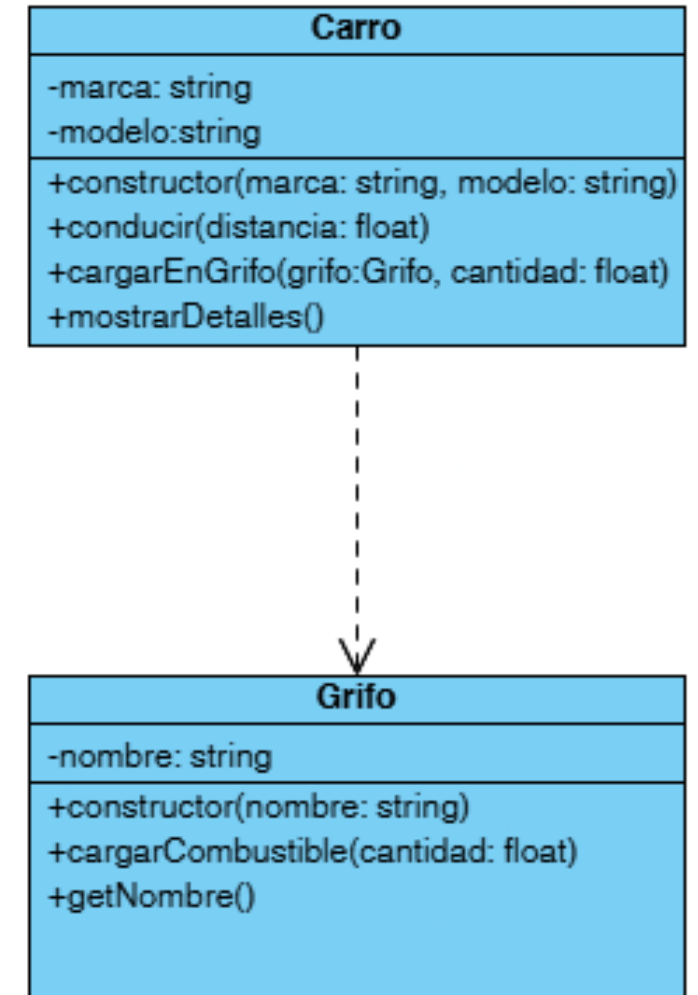


```
class Coche:
    def __init__(self, marca, modelo):
        self.__marca = marca
        self.__modelo = modelo
        self.__combustible = 0

    def conducir(self, distancia):
        if self.__combustible >= distancia * 0.1:
            print(f"Conduciendo {distancia} km en un "
                  f"{self.__marca} {self.__modelo}...")
            self.__combustible -= distancia * 0.1
        else:
            print(f"No hay suficiente combustible para conducir "
                  f"{distancia} km.")

    def cargarEnGrifo(self, grifo, cantidad):
        grifo.cargarCombustible(cantidad)
        self.__combustible += cantidad
        print(
            f"El {self.__marca} {self.__modelo} ahora tiene "
            f"{self.__combustible} litros de combustible."
        )

    def mostrarDetalles(self):
        print(
            f"Coche: {self.__marca} {self.__modelo}, "
            f"Combustible: {self.__combustible} litros"
        )
```

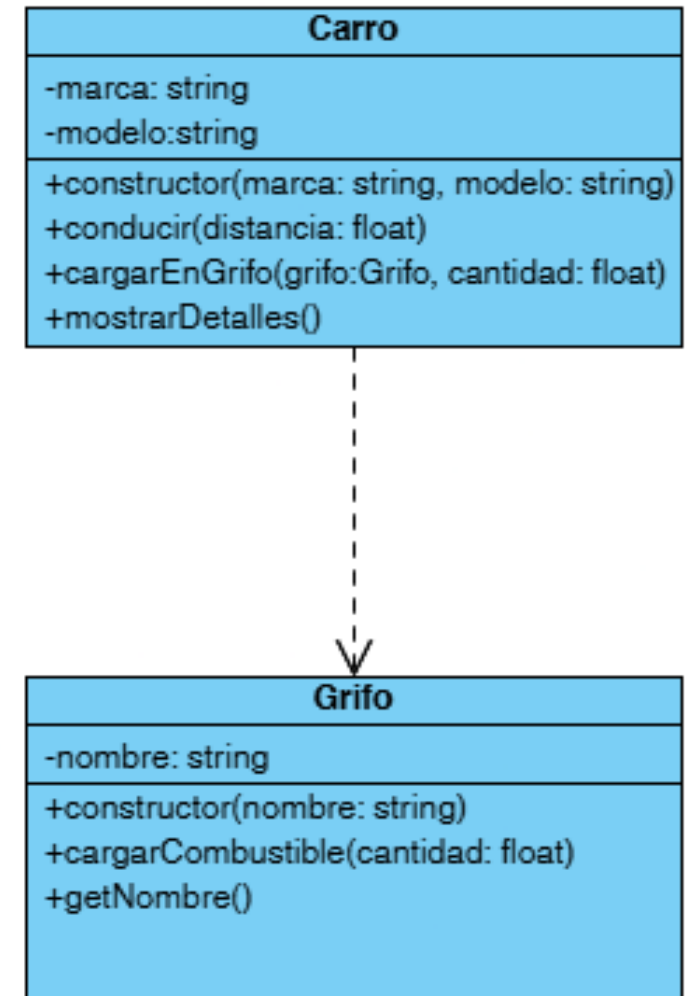




## Ejemplo dependencia

```
# Programa Principal
miGrifo = Grifo("Grifo 24 Horas")
miCoche = Coche("Nissan", "Sentra")

# Uso de la dependencia
miCoche.mostrarDetalles()
miCoche.cargarEnGrifo(miGrifo, 30)
miCoche.conducir(100)
miCoche.conducir(400)
miCoche.mostrarDetalles()
```

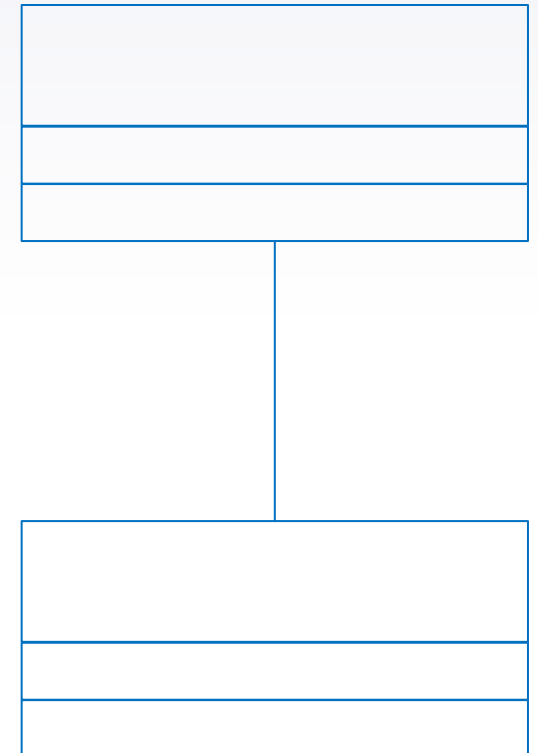




# Relación de Asociación

- ▶ Una relación de asociación es un **vínculo estructural** entre dos o más clases que permite que los objetos de estas clases interactúen entre sí.
- ▶ Esta relación describe cómo los objetos de una clase están **conectados** con los objetos de otra clase, y puede ser bidireccional o unidireccional.
- ▶ La asociación es una relación más fuerte que la dependencia, ya que **implica una conexión permanente** entre las clases.
- ▶ Características Clave:
  - ▶ Conecta Objetos: Establece una conexión entre instancias de dos o más clases.
  - ▶ Puede ser Bidireccional o Unidireccional:
    - Bidireccional: Ambas clases conocen y pueden interactuar entre sí.
    - Unidireccional: Solo una clase conoce y puede interactuar con la otra.
  - ▶ Cardinalidad: Define cuántos objetos de una clase están relacionados con cuántos objetos de la otra clase (por ejemplo, 1 a 1, 1 a muchos, muchos a muchos).
  - ▶ Relación Estructural: Implica una conexión más permanente que la dependencia, ya que los objetos pueden mantener referencias entre sí.
  - ▶ Es una relación donde todos los objetos tienen su propio ciclo de vida y no hay dueño.

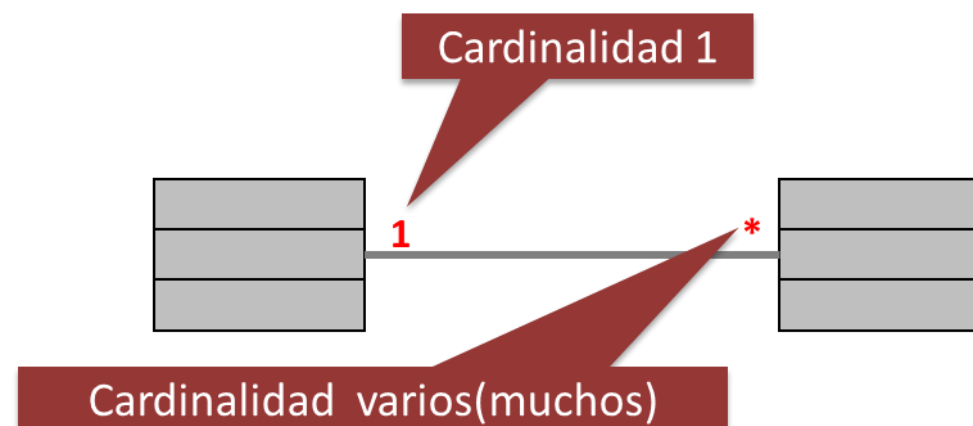
## ASOCIACIÓN





# Relación de Asociación: Cardinalidad

- ▶ La **cardinalidad**, también conocida como **multiplicidad**, es un concepto fundamental en las relaciones de asociación dentro de la POO.
- ▶ Define **cuántos objetos** de una clase están involucrados en una relación con **objetos de otra clase**.
- ▶ En otras palabras, la **cardinalidad** especifica el **número de instancias de una clase** que pueden o deben relacionarse con **un número determinado de instancias de otra clase**.
- ▶ Cada asociación tiene dos multiplicidades, una en cada extremo de la relación. Estas multiplicidades indican:
  - Cuántos objetos de la primera clase pueden relacionarse con un objeto de la segunda clase.
  - Cuántos objetos de la segunda clase pueden relacionarse con un objeto de la primera clase.



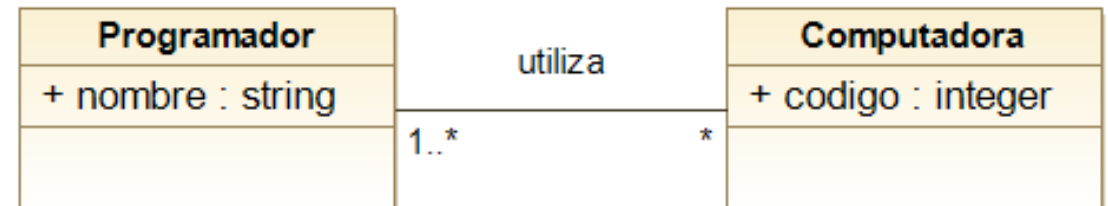




# Relación de Asociación: Cardinalidad

## Interpretación de la Cardinalidad:

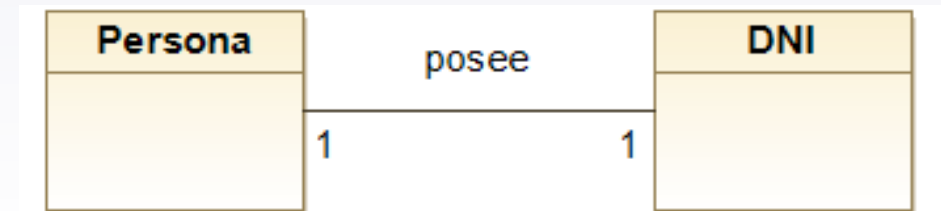
- ▶ Cardinalidad 0..N o \*: Indica que la asociación es opcional. Por ejemplo, un programador puede no utilizar ninguna computadora o utilizar varias.
- ▶ Cardinalidad 1..N o 1..\*: Indica que la asociación es obligatoria. Por ejemplo, una computadora debe ser utilizada por al menos un programador.
- ▶ Entonces el diagrama se puede leer:
  - ▶ Cada programador puede utilizar cero o varias computadoras (cardinalidad 0..\*).
  - ▶ Cada computadora debe ser utilizada por al menos un programador (cardinalidad 1..\*).
- ▶ La cardinalidad permite modelar con precisión las interacciones entre las clases, asegurando que las relaciones sean claras y bien definidas.





# Relación de Asociación: Cardinalidad

- ▶ En la relación Persona – DNI, se debe entender que:
  - Un objeto de tipo Persona se relaciona con solo 1 objeto de tipo DNI.
  - Un objeto de tipo DNI se relaciona con solo 1 objeto de tipo Persona.
- ▶ En la relación Estudiante – Asignatura, se debe entender que:
  - Un objeto de tipo Estudiante se relaciona con muchos objetos de tipo Asignatura.
  - Un objeto de tipo Asignatura se relaciona con muchos objetos de tipo Estudiante.





# Relación de Asociación: Cardinalidad

- ▶ De modo general, la cardinalidad o multiplicidad suele expresarse a través de la siguiente simbología:

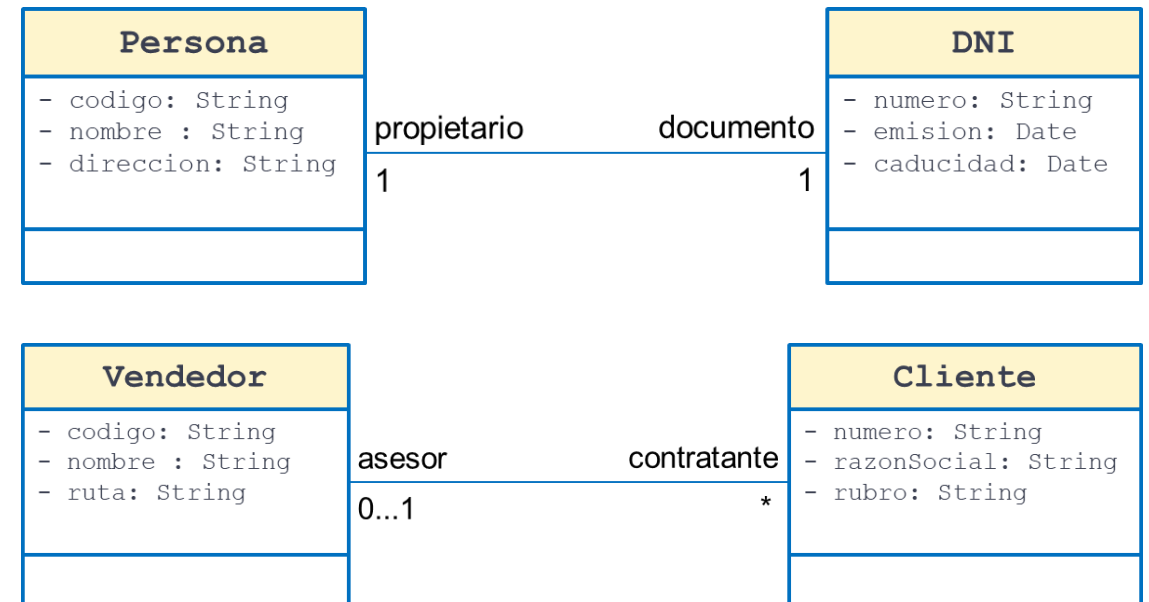
Multiplicidad	Significado
1	1 y sólo 1
0..1	0 ó 1 (Asociación Opcional)
N..M	Un valor entre N y M
*	Varios
0..*	0 ó muchos (ilimitado)
1..*	1 ó muchos
2..4	Rango Específico
2, 4..6, 8	Múltiple, Rangos Disjuntos



# Relación de Asociación: Cardinalidad

## Roles

- ▶ Los roles son una herramienta que permite clarificar la participación de los objetos en una relación. A través de los roles, se puede definir cómo interactúan los objetos entre sí y qué función desempeña cada uno en la relación.
- ▶ Ejemplos de Roles en los diagramas:
  - ▶ Persona y DNI:
    - ▶ Un objeto de tipo Persona es el propietario de un objeto de tipo DNI.
    - ▶ Un objeto de tipo DNI es el documento de un objeto de tipo Persona.
  - ▶ Vendedor y Cliente:
    - ▶ Un objeto de tipo Vendedor es el asesor de muchos objetos de tipo Cliente.
    - ▶ Un objeto de tipo Cliente es el contratante de un objeto de tipo Vendedor.

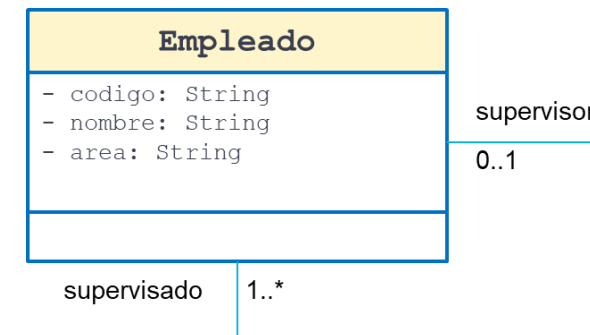
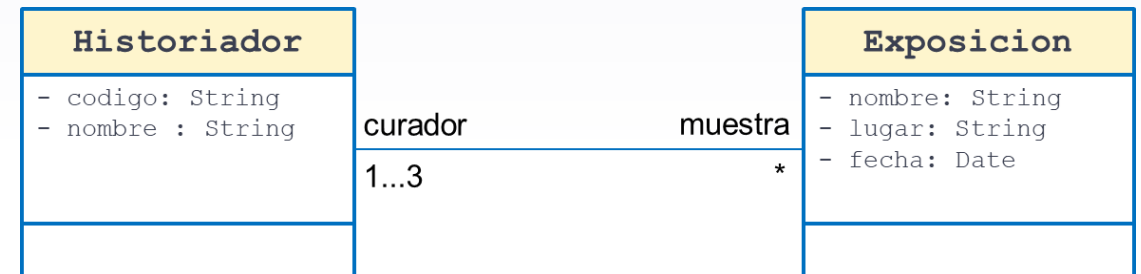




# Relación de Asociación: Cardinalidad

## Roles

- ▶ Ejemplos de Roles en los diagramas:
  - ▶ Historiador y Exposición:
    - ▶ Un objeto de tipo Historiador es el curador de muchos objetos de tipo Exposición.
    - ▶ Un objeto de tipo Exposición es la muestra de uno a tres objetos de tipo Historiador.
  - ▶ Empleado (Supervisor y Supervisado):
    - ▶ Un objeto de tipo Empleado puede ser el supervisor de muchos objetos de tipo Empleado.
    - ▶ Un objeto de tipo Empleado puede ser supervisado por ninguno o un objeto de tipo Empleado.

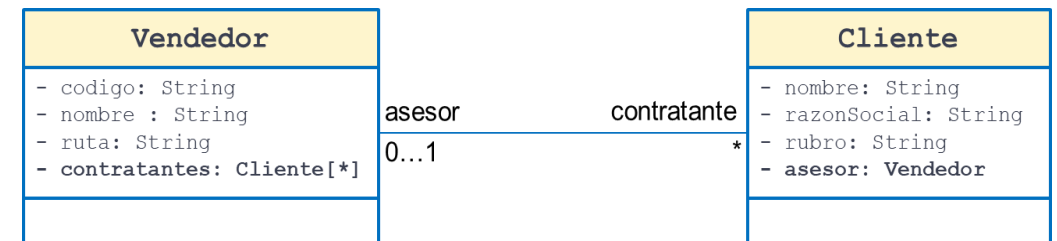
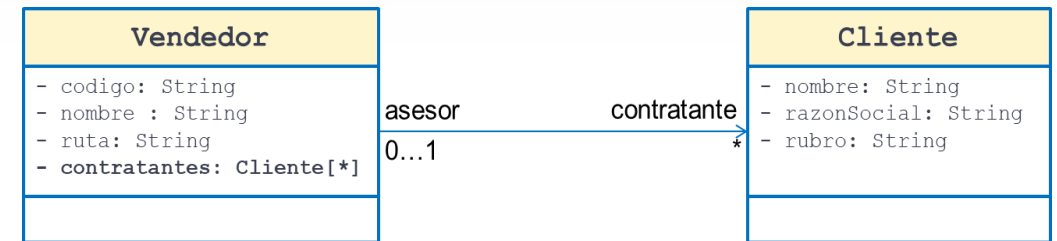
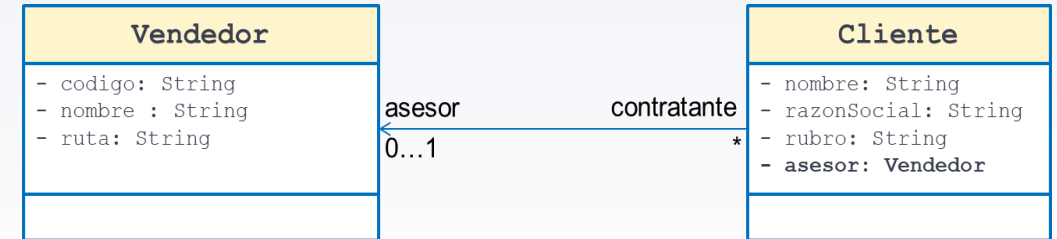




# Relación de Asociación: Navegabilidad

## Navegabilidad

- ▶ La navegabilidad es un concepto clave en las relaciones entre clases, ya que **determina qué clase tiene acceso a la información de la otra**. Esta característica define la dirección en la que se puede acceder a los objetos relacionados y cómo se gestionan las referencias entre ellos.
- ▶ Tipos de Navegabilidad:
  - ▶ **Navegabilidad Unidireccional:**
    - ▶ Indica que una clase tiene acceso a la información de otra, pero no viceversa.
    - ▶ Es la forma más común de navegabilidad y se representa gráficamente con una flecha de una sola punta que apunta hacia la clase accesible.
  - ▶ **Navegabilidad Bidireccional:**
    - ▶ Indica que ambas clases tienen acceso a la información de la otra.
    - ▶ Se representa gráficamente con una línea simple o una flecha de dos puntas.

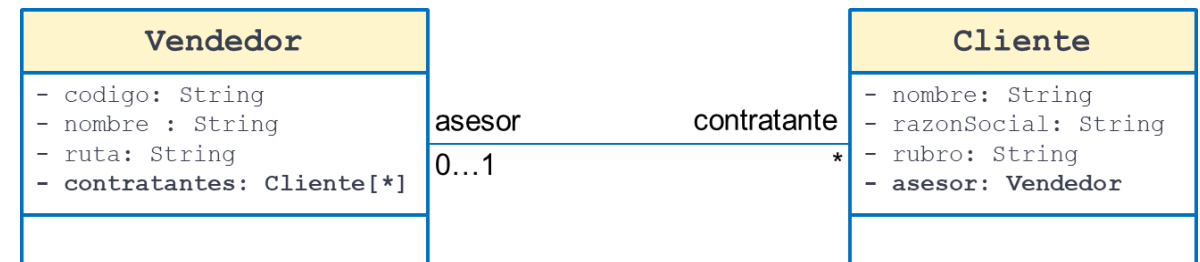
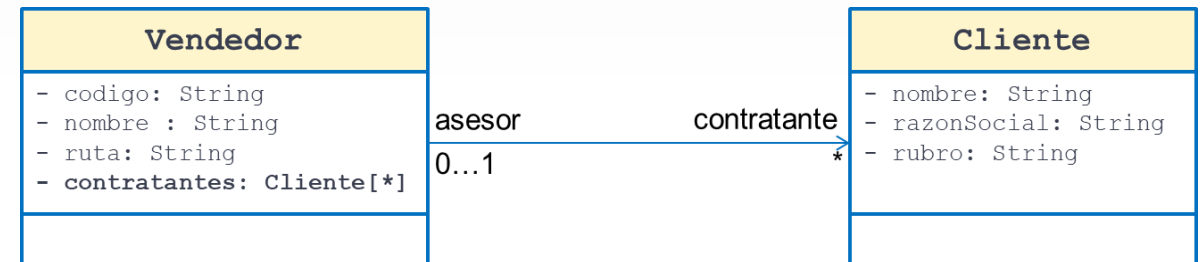
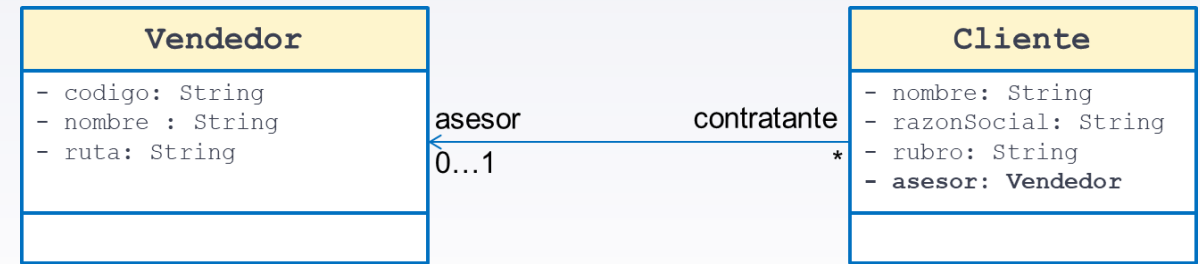




# Relación de Asociación: Navegabilidad

## Navegabilidad

- ▶ Acceso a la Información:
  - ▶ El acceso a la información de la otra clase se logra a través de una referencia a los objetos que las vinculan.
  - ▶ Si la relación especifica roles, estos se utilizan para nombrar los atributos de referencia. En caso contrario, se suele utilizar el nombre de las clases.

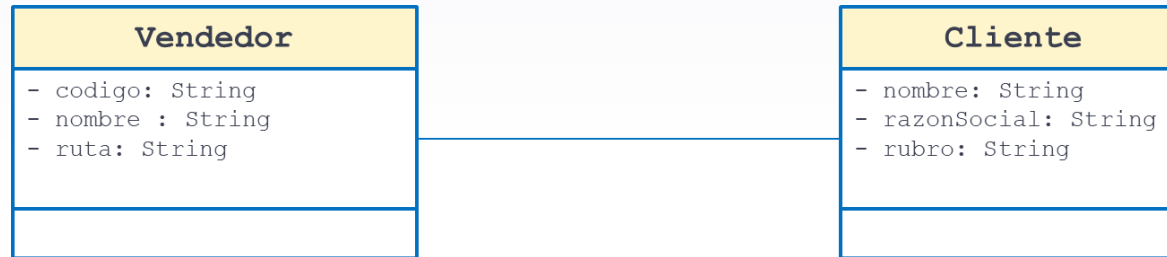




# Relación de Asociación

## Implementación

- ▶ Lo primero es implementar las clases tal como han sido diseñadas y acorde al lenguaje de programación.
- ▶ En el ejemplo mostrado, ambas clases implementan atributos privados, por lo que se asume la existencia de métodos de acceso.



```
class Vendedor:

    def __init__(self, codigo, nombre, ruta):
        #Atributos de instancia
        self.__codigo = codigo
        self.__nombre = nombre
        self.__ruta = ruta

    #Métodos de acceso
    ...

    #Métodos
    def __str__(self) -> str:
        return f"codigo:{self.__codigo}, nombre:{self.__nombre}, ruta:{self.__ruta}"
```

```
class Cliente:

    def __init__(self, nombre:str, razon_social:str, rubro:str):
        #Atributos de instancia
        self.__nombre = nombre
        self.__razon_social = razon_social
        self.__rubro = rubro

    #Métodos de acceso
    ...

    #Métodos
    def __str__(self) -> str:
        return f"nombre:{self.__nombre}, razon_social:{self.__razon_social}, rubro:{self.__rubro}"
```

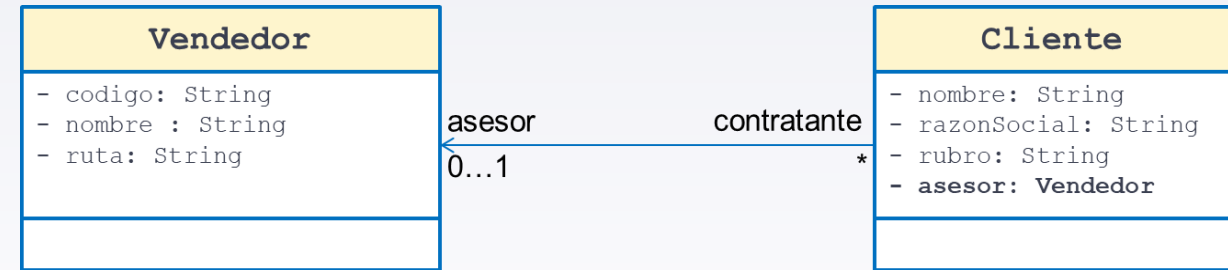




# Relación de Asociación

## Implementación

- ▶ Luego, dependiendo de la cardinalidad, los roles y, especialmente, la navegabilidad de la relación se deben implementar los atributos que guardarán las referencias de una clase en la otra.
- ▶ Por ejemplo, para una **relación unidireccional de Cliente a Vendedor**, en la cual el objeto de tipo Cliente guarda una referencia del objeto de tipo Vendedor con el cual se relaciona:
- ▶ Puesto que la cardinalidad es 0...1, se entiende que la referencia al asesor de tipo Vendedor es opcional y su definición no se requiere desde la construcción del objeto de tipo Cliente.
- ▶ La gestión del objeto de tipo Vendedor asociado como asesor (atributo privado) puede implicar la implementación de los métodos de acceso (get y set).



```
class Cliente:

    def __init__(self, nombre:str, razon_social:str, rubro:str):
        #Atributos de instancia
        self.__nombre = nombre
        self.__razon_social = razon_social
        self.__rubro = rubro
        self.__asesor = None

    #Métodos de acceso
    ...
    def get_asesor(self):
        return self.__asesor

    def set_asesor(self, asesor):
        if isinstance(asesor, Vendedor):
            self.__asesor = asesor
        else:
            raise ValueError("El elemento debe ser de tipo Vendedor")

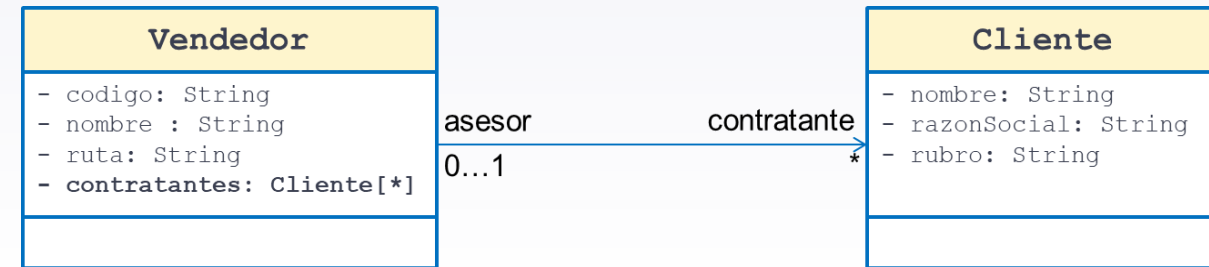
    #Métodos
```

# Relación de Asociación



## Implementación

- ▶ Para una relación **unidireccional de Vendedor a Cliente**, en la cual el objeto de tipo Vendedor guarda referencia de los objetos de tipo Cliente con los cuales se relaciona, se requiere gestionar una colección:
- ▶ Puesto que la cardinalidad es \*, se entiende que las referencias a los contratantes de tipo Cliente es opcional y su definición no se requiere desde la construcción del objeto de tipo Vendedor.

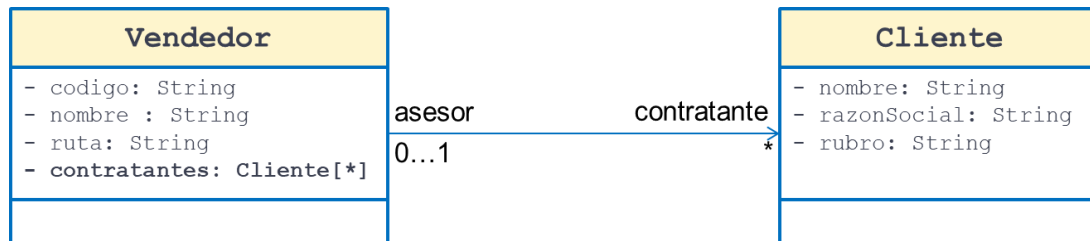


# Relación de Asociación



## Implementación

- ▶ La gestión de la colección de objetos de tipo Cliente asociados como contratantes (atributo privado) puede implicar la implementación de, al menos, métodos tales como:
  - ▶ Agregar (add) el objeto recibido como parámetro
  - ▶ Obtener (get) un objeto de la colección por su índice
  - ▶ Buscar (look) un objeto en la colección por su código identificador
  - ▶ Remover (del) un objeto de la colección por su índice o referencia
  - ▶ Listar (lst) los objetos de la colección



```
class Vendedor:

    def __init__(self, codigo, nombre, ruta):
        #Atributos de instancia
        self.__codigo = codigo
        self.__nombre = nombre
        self.__ruta = ruta
        self.__contratantes = []

    #Métodos de acceso
    ...
    def add_contratante(self, contratante):
        if isinstance(contratante, Cliente):
            self.__contratantes.append(contratante)
        else:
            raise ValueError("El elemento debe ser de tipo Cliente")

    def get_contratante(self, indice):
        if len(self.__contratantes) == 0:
            print("No hay contratantes registrados")
        else:
            if not (0 <= indice < len(self.__contratantes)):
                print("La posición indicada no es válida")
            else:
                return self.__contratantes[indice]

        return None

    def lst_contratantes(self):
        if len(self.__contratantes) == 0:
            print("No hay contratantes registrados")
        else:
            for contratante in self.__contratantes:
                print(contratante)

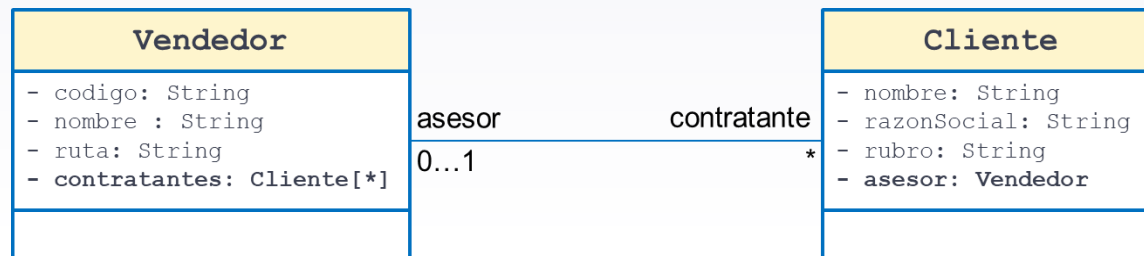
    #Métodos
```

# Relación de Asociación



## Implementación

- Para una relación bidireccional entre Vendedor y Cliente, se requiere implementar los métodos de gestión de referencias en ambas clases:



```
class Cliente:

    def __init__(self, nombre:str, razon_social:str, rubro:str):
        #Atributos de instancia
        self.__nombre = nombre
        self.__razon_social = razon_social
        self.__rubro = rubro
        self.__asesor = None

        #Métodos de acceso
        ...

    def get_asesor(self):
        return self.__asesor

    def set_asesor(self, asesor):
        if isinstance(asesor, Vendedor):
            self.__asesor = asesor
        else:
            raise ValueError("El elemento debe ser de tipo Vendedor")

        #Métodos
```

```
class Vendedor:

    def __init__(self, codigo, nombre, ruta):
        #Atributos de instancia
        self.__codigo = codigo
        self.__nombre = nombre
        self.__ruta = ruta
        self.__contratantes = []

        #Métodos de acceso
        ...

    def add_contratante(self, contratante):
        if isinstance(contratante, Cliente):
            self.__contratantes.append(contratante)
        else:
            raise ValueError("El elemento debe ser de tipo Cliente")

    def get_contratante(self, indice):
        if len(self.__contratantes) == 0:
            print("No hay contratantes registrados")
        else:
            if not (0 <= indice < len(self.__contratantes)):
                print("La posición indicada no es válida")
            else:
                return self.__contratantes[indice]

        return None

    def lst_contratantes(self):
        if len(self.__contratantes) == 0:
            print("No hay contratantes registrados")
        else:
            for contratante in self.__contratantes:
                print(contratante)

        #Métodos
```

# PREGRADO

Ingeniería de Sistemas de Información

Escuela de Ingeniería de Sistemas y Computación | Facultad de Ingeniería



Universidad Peruana  
de Ciencias Aplicadas

Prolongación Primavera 2390,  
Monterrico, Santiago de Surco

Lima 33 - Perú  
T 511 313 3333

<https://www.upc.edu.pe>

*exígete, innova*

UPC

