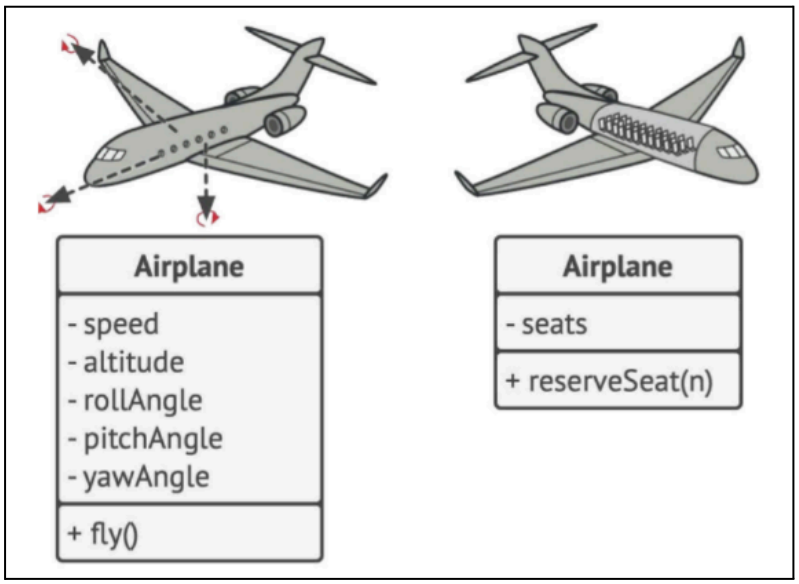
Abstracción



La Abstracción es el modelo de un objeto o fenómeno del mundo real, limitado a un contexto específico, que representa todos los datos relevantes a este contexto con gran precisión, omitiendo el resto.

Encapsulación

Es la capacidad que tiene un objeto de esconder partes de su estado y comportamiento de otros objetos, exponiendo únicamente una interfaz limitada al resto del programa**.**

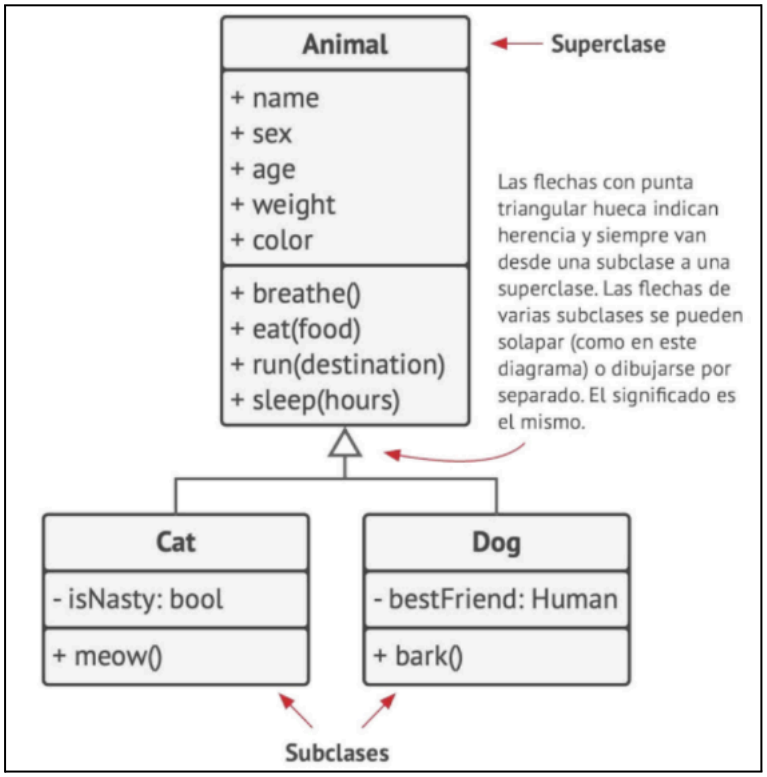
Encapsular algo significa hacerlo privado y, por ello, accesible únicamente desde dentro de los métodos de su propia clase.

La encapsulación oculta lo que hace un objeto de lo que hacen otros objetos y del mundo exterior, por lo que se denomina también ocultación de datos.

Herencia

La herencia es la capacidad de crear nuevas clases sobre otras existentes. La principal ventaja de la herencia es la **reutilización de código**. Si quieres crear una clase ligeramente diferente a una ya existente, no hay necesidad de duplicar el código.

En su lugar, extiendes la clase existente y colocas la funcionalidad adicional dentro de una subclase resultante que hereda los campos y métodos de la superclase.



Las subclases pueden sobrescribir el comportamiento de los métodos que heredan de clases padre. Una subclase puede sustituir completamente el comportamiento por defecto o limitarse a mejorarlo con líneas de código extras.

Polimorfismo

El polimorfismo es la propiedad que permite tener el **mismo nombre de método** en clases diferentes y que **actúe de modo diferente en cada una de ellas.**

Clases y Objetos

¿Qué sabemos sobre la mayoría de los perros domésticos?
Pues todos tienen un nombre y una edad. También sabemos que la mayoría de los perros se sientan y se dan vuelta. Esos dos datos (nombre y edad) y esos dos comportamientos (sentarse y darse vuelta) se incluirán en nuestra clase Dog porque son comunes a la mayoría de los perros.

El método __init__()

Una función que es parte de una clase es un método. El método __init__() es un método especial que Python ejecuta automáticamente cada vez que creamos una nueva instancia basada en la clase Dog. Es un método de clase.

Definimos el método __init__() en este ejemplo para que tenga tres parámetros: self, name y age.

El parámetro self es obligatorio en la definición del método y debe aparecer primero, antes que los demás parámetros que son opcionales.

Debe incluirse el parámetro self en la definición porque cuando Python llame a este método más adelante (para crear una instancia de Dog), la llamada al método pasará automáticamente el argumento self.

Por lo general, podemos suponer que un **nombre en mayúscula** como Dog se refiere a una **clase**, y un **nombre en minúscula** como my_dog se refiere a una única **instancia** creada a partir de una clase.

Accediendo a los atributos y comportamientos

Visibilidad de atributos y métodos en Python

No obstante, como convención se prefiere **un guión bajo** antes del nombre de un miembro para interpretar el mismo como **protected** y **dos guiones bajo** para interpretarlo como **privado**.

Los modificadores de acceso cumplen con el propósito de **Encapsulamiento.**

Su misión es hacer inaccesible los detalles internos de la clase, con el fin de evitar que otras entidades sepan de su existencia y accedan a ellos directamente produciendo comportamiento inesperado.

Esto, con el fin de minimizar el coupling entre entidades.

mi_clase.py > MiClase

```python
class MiClase:

    def __init__(self):
        self._atributo_protejido = 1
        self.__atributo_privado = ''

    def _metodo_protejido(self):
        return self._atributo_protejido

    def __metodo_privado(self):
        return self.__atributo_privado
```

Vamos a un ejemplo con la clase Car:

car.py > Car > _get_short_descriptive

```python
class Car:

    def __init__(self, model: str, year: int):
        """Initialize attributes"""
        self.model = model
        self.year = year
        self._cost = 0

    def get_description(self):
        if self._cost > 0:
            return self._get_full_descriptive()
        else:
            return self._get_short_descriptive()

    def _get_full_descriptive(self):
        full = f"El auto es modelo {self.model} del año {self.year} y la valuación es de {self._cost}"
        return full.title()

    def _get_short_descriptive(self):
        short = f"El auto es modelo {self.model} del año {self.year}, se desconoce su valuación"
        return short.title()
```

app.py > ...

```python
from car import *

my_car = Car('Siena', 1998)
print(my_car.get_description())
```

Esta clase tiene métodos y atributos públicos y privados.

Los atributos se pueden llegar a definir sin necesidad de pasarlos como parámetros (por ejemplo en este caso el costo es 0 en todos los objetos Cars).

Un **atributo público** va a poder ser editado a través de una instancia poniendo **objeto.atributo**

Un **método público** va a poder ser llama llamado a través de una instancia poniendo **objeto.método()**

Si bien python permite acceder a los atributos y métodos que declaramos como **protegidos** y **privados**. Se sabe bien que esto no es conveniente.

Getters y Setters

Tenemos a la clase House:

```
house.py > House > __init__
1   class House:
2
3       def __init__(self, price):
4           self.price = price
```

Tiene un precio público, cualquiera podría ingresar y cambiarle el atributo mediante un punto.

```
9   # Access value
10  obj.price
11  # Modify value
12  obj.price = 40000
```

Si aplicamo getters y setters:

```
house.py > ...
1   class House:
2
3       def __init__(self, price):
4           self._price = price
5
6       # getter method
7       def get_price(self):
8           return self._price
9
10      # setter method
11      def set_price(self, price):
12          self._price = price

14  # Changed from obj.price
15  obj.get_price()
16  # Changed from obj.price = 40000
17  obj.set_price(40000)
```

@Property - decorador

El decorador nos permite agregar funcionalidad a una función sin modificarla.

Hay 3 métodos para una propiedad:

- getter: acceder al valor del atributo
- setter: establecer un valor al atributo
- deleter: eliminar un atributo de instancia

```
@property
def price(self):
    return self._price
```

@property se utiliza para indicar que vamos a definir una propiedad. Mejora la legibilidad porque podemos ver claramente el propósito del método **def price(self)**.

“def price (self) ” El nombre del método coincide con el atributo que estamos tratando. Recibe solo self.

“return self._price” Retorna el precio acordado al instanciar el objeto.

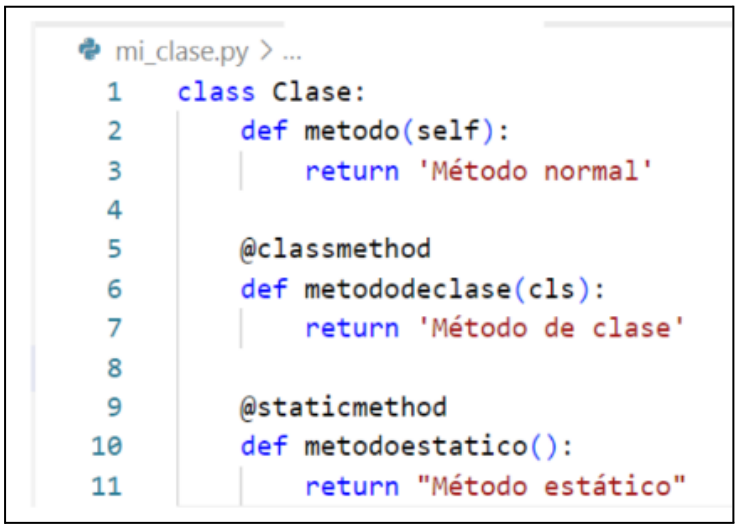
```
@price.setter
def price(self, new_price):
    if new_price > 0 and isinstance(new_price, float):
        self._price = new_price
    else:
        print("Please enter a valid price")
```

se valida que sea positivo y que sea float.

```
@price.deleter
def price(self):
    del self._price
```

Se pueden definir atributos de solo lectura al desarrollar solo el método getter. También puede no estar el deleter.

Métodos en Python: instancia, clase y estáticos



Método de instancia:

- No lleva decorador.
- Recibe como parámetro la palabra "**self**"
- **Accede a la instancia** desde la cual lo llamamos
- Puede recibir otros argumentos.
- **Se requiere instanciar** al objeto para ser llamados.

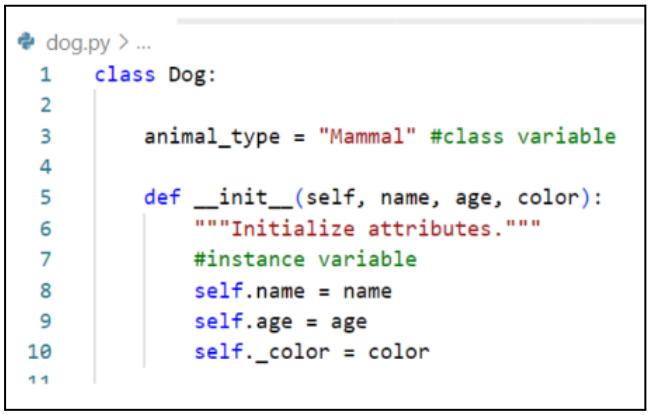
Método de clase:

- Lleva el decorador **@classmethod**
- Recibe como parámetro la palabra "**cls**"
- **Accede a la clase** desde la cual lo llamamos, no a la instancia!
- **No hace falta instanciar** el objeto para llamarlo

Método estático:

- Lleva el decorador **@staticmethod.**
- No reciben parámetro.
- No acceden ni a la clase ni a la instancia.

Atributos en Python: instancia y clase



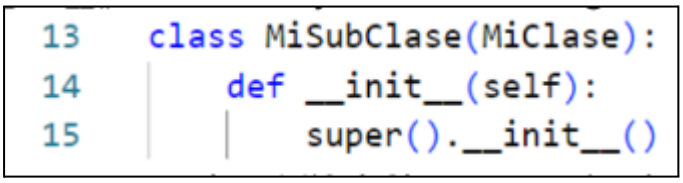
Variables de instancia:

- Llamadas **atributos de instancia**
- Información particular de cada instancia creada con una clase.
- Se pueden crear en el **constructor** o en un **método de instancia**

Variables de clase:

- Llamada **atributos de clase**
- Información de todas las instancias creadas de una clase, general.
- Si se modifica se modifica para todas las instancias

Herencia



En el ejemplo anterior tenemos una clase MiSubClase que extiende la MiClase (hereda todos los miembros). Luego al crear un objeto de MiSubClase, se ejecuta ambos métodos __init__( ).

La función super() es una función especial que le permite llamar a un método de otra clase superior en la jerarquía. El nombre super proviene de una convención de llamar a la clase principal superclase y a la clase secundaria subclase.

Polimorfismo

```python
class Animal:

    def __init__(self) -> None:
        pass

    def hablar(self) -> str:
        pass

class Perro(Animal):

    def __init__(self) -> None:
        pass
    def hablar(self) -> str:
        return "Guau!"

class Gato(Animal):

    def __init__(self) -> None:
        pass

    def hablar(self) -> str:
        return "Miau!"
```

En programación orientada a objetos, el polimorfismo se refiere a la propiedad por la que es posible enviar mensajes sintácticamente iguales a objetos de tipos distintos.

```python
pet = Perro()
print(pet.hablar())
pet = Gato()
print(pet.hablar())
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

PS C:\Users\valon\Desktop\PythonEjercicic
Guau!
Miau!
```

El concepto de polimorfismo podemos aplicarlo para **sobrecargar o sobrescribir métodos**.

Sobreescritura de métodos

```python
class Banco():
    ...

    def get_address(self) -> str:
        return "La dirección de la casa central del banco es " + self.__address

    ...

class Brubank():
    ...

    def get_address(self) -> str:
        return "Brubank no tiene dirección, Brubank es un banco digital"

    ...
```

Sobrecarga de métodos

Consiste en tener **diferentes métodos con el mismo nombre en una misma clase**, y que el intérprete o compilador logre **diferenciarlos por los tipos de datos que se envían como argumentos** para los parámetros **en la llamada al método.**

Herencia Múltiple

Una clase puede heredar comportamientos y características de más de una superclase.

Esto contrasta con la **herencia simple**, donde una clase sólo puede heredar de una superclase.

Python soporta la herencia múltiple pero **no es una buena práctica** en programación y hay pocos casos dónde deba ser implementada.

Clase Abstracta

Unas clases en las que se pueden **definir tanto métodos como propiedades**, pero que **no pueden ser instancias directamente.**

Solamente se pueden usar para construir subclases.

Proporciona atributos y métodos comunes para todas las subclases **evitando así la necesidad de duplicar código.**

Para poder crear clases abstractas en Python es necesario importar la **clase ABC** y el **decorador abstractmethod** del módulo abc (Abstract Base Classes).

```python
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def mover(self) -> None:
        pass

class Gato(Animal):
    def mover(self) -> str:
        return 'Mover gato'

my_cat = Gato()
```

raise NotImplementedError()

En las clases base definidas por el usuario, los métodos abstractos deben generar esta excepción cuando requieren que las clases derivadas anulen el método.

También se puede utilizar en los métodos de una clase abstracta cuando se deriva el funcionamiento de un método a su clase hija.

```python
@abstractmethod
def un_metodo(self):
    raise NotImplementedError
```

```python
from abc import ABC, abstractmethod

class Persona(ABC):

    # Constructor de la clase abstracta necesario para inciializar el campo nombre
    def __init__(self, nombre: str) -> None:
        self._nombre = nombre #campo protegido

    @property #Propiedad concreta, puede sobreescribirse en la clase hija
    def nombre(self) -> str:
        return self._nombre.title()

    @nombre.setter #Propiedad abstracta, debe sobreescribirse en la clase hija
    @abstractmethod
    def nombre(self, nuevo_nombre: str):
        self._nombre = nuevo_nombre

    @abstractmethod #Metodo abstracto, debe sobreescribirse en la clase hija
    def __str__(self) -> str:
        raise NotImplementedError

    def mensaje(self) -> str: #Metodo concreto, puede sobreescribirse en la clase hija
        return "Esto es una persona y su nombre es " + self.nombre
```

Modelado de Aplicaciones: UML - Diagrama de clase

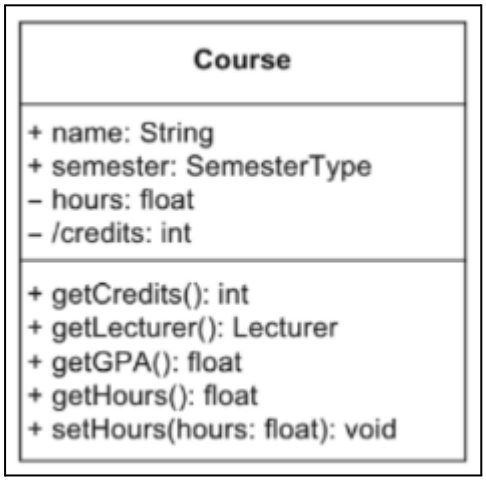
En diseño de sistemas, se modela por una importante razón: gestionar la complejidad. Un modelo es una abstracción de cosas reales.

Diagramas de clases

En el contexto de la programación orientada a objetos, el diagrama de clases visualiza las clases que componen un sistema de software y las relaciones entre estas clases.

Nomenclatura del diagrama de clases

En un diagrama de clases, una clase está representada por un rectángulo que se puede subdividir en varios compartimentos.



El **primer compartimento** debe contener el **nombre** de la clase, que generalmente comienza con una letra mayúscula y se coloca centrado en negrita.

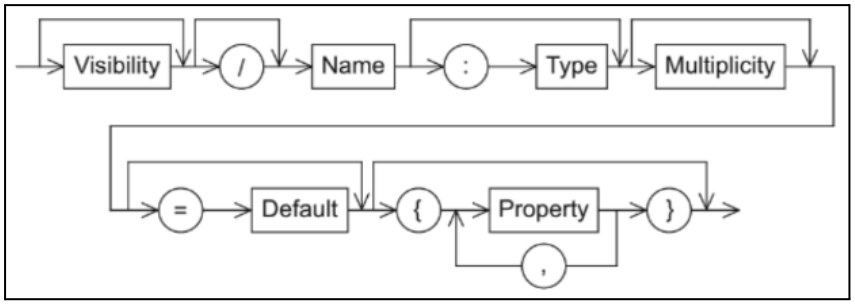
El **segundo compartimento** del rectángulo contiene los **atributos** de la clase.

El **tercer compartimento** contiene los **métodos**.

Atributos

Un atributo tiene al menos un nombre.

Y su sintaxis de definición es:



Una **barra diagonal /** antes del nombre de un atributo indica que el valor de este atributo se deriva de otros atributos. Un ejemplo de atributo derivado es la edad de una persona, que se puede calcular a partir de la fecha de nacimiento.

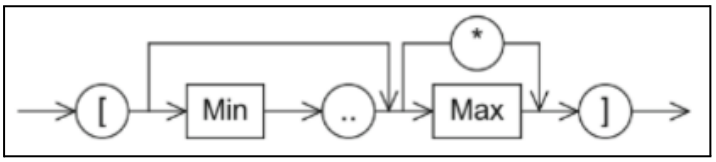
Los posibles **tipos** de atributos incluyen tipos de datos primitivos del lenguaje.

Para definir un **valor predeterminado** para un atributo, especifique = default, donde el valor predeterminado es un valor o expresión definido por el usuario.

Puede especificar propiedades adicionales del atributo entre corchetes. Por ejemplo, la propiedad **{readOnly}** significa que el valor del atributo no se puede cambiar una vez que se ha inicializado.

Multiplicidad

La multiplicidad de un atributo indica cuántos valores puede contener un atributo.

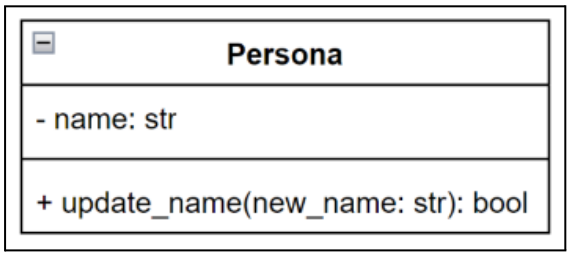


La multiplicidad se muestra como un intervalo encerrado entre corchetes en la forma [mínimo... máximo], donde mínimo y máximo son números naturales que indican los límites superior e inferior del intervalo.

Si no hay un límite superior para el intervalo, éste se expresa con un asterisco ∗.

Comportamiento

Conocido como métodos.



En un diagrama de clases, el nombre del método va seguido de una lista de parámetros entre paréntesis.

La única información obligatoria es el nombre del parámetro. La adición de un tipo y un valor predeterminado son opcionales.

Visibilidad

Para especificar la visibilidad de un miembro de la clase (es decir, cualquier atributo o método), deben colocarse la anotación antes del nombre de los miembros.

public	**+**	Todos los miembros declarados cómo públicos son de libre acceso desde cualquier otra parte de un programa.
private	**-**	Todos los miembros declarados cómo privados no son accesibles por fuera de la clase.
protected	**#**	Todos los miembros declarados cómo protegidos son accesibles por la clase en que fueran declarados y todas las subclases.

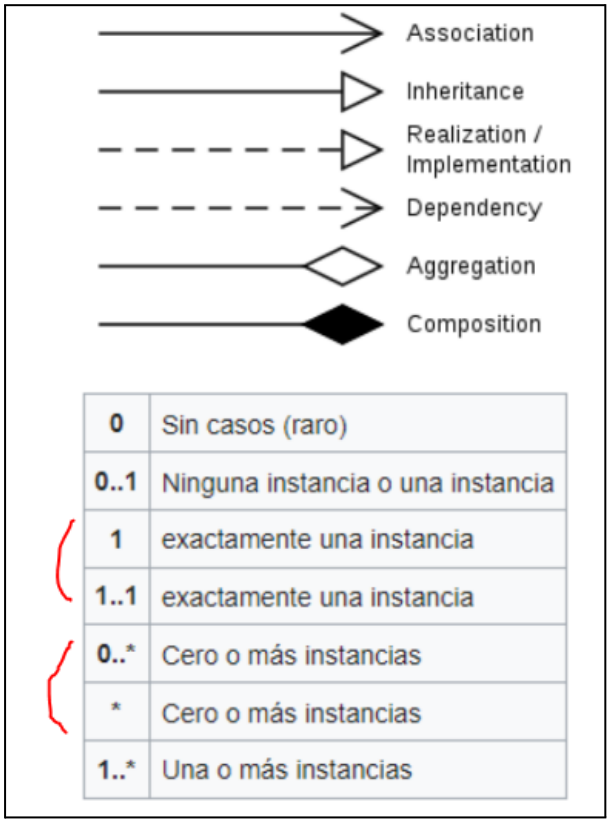
Variables de clase y métodos de clase

Estas variables también se denominan atributos estáticos o atributos de clase.

En el diagrama de clases, los atributos estáticos están subrayados, al igual que los métodos estáticos.

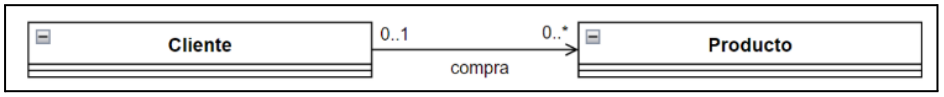
Los métodos estáticos, también llamados métodos de clase, **se pueden utilizar si no se creó ninguna instancia de la clase correspondiente.**

Relaciones

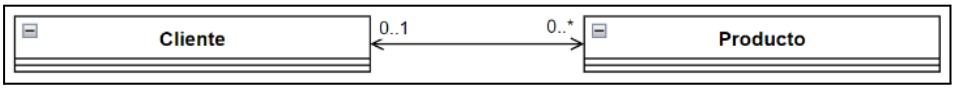


0	Sin casos (raro)
0..1	Ninguna instancia o una instancia
1	exactamente una instancia
1..1	exactamente una instancia
0..*	Cero o más instancias
*	Cero o más instancias
1..*	Una o más instancias

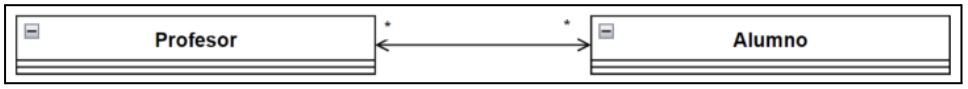
Asociaciones



En este caso el cliente conoce los productos que compró, por ejemplo en un resumen del pedido. Pero el Producto no conoce el cliente que lo compró.



De una lista de productos vendidos es posible obtener una lista de los clientes que compran más de x cantidad de productos mensuales.

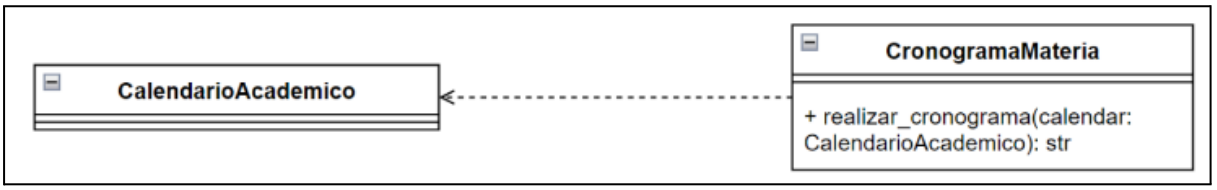


En este caso un profesor accede a todos sus alumnos, y un alumno puede acceder a todos sus profesores.

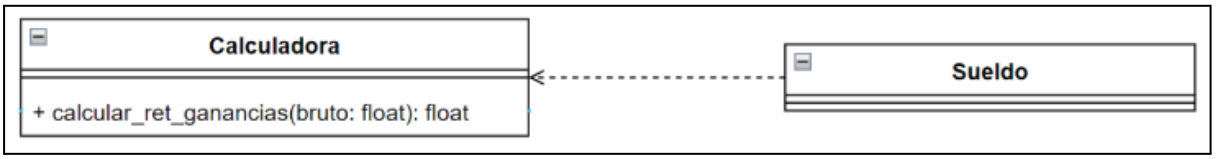
Dependencia

Especifica algún tipo de dependencia entre dos clases, donde un cambio en la clase de la cual se depende puede afectar a la clase dependiente, pero no necesariamente a la inversa.

Se reflejan en los métodos de una clase que utilizan el objeto de otra clase como parámetro.



La clase CronogramaMateria tiene una dependencia con el CalendarioAcademico ya que es pasado cómo parámetro a la llamada del método realizar cronograma.



La clase Sueldo depende de la clase Calculadora, ya que para realizar comportamiento cómo por ejemplo calcular el sueldo a abonar, llama al método calcular_ret_ganancias.

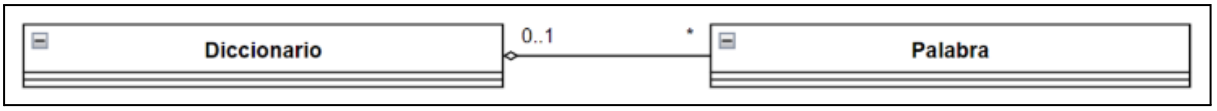
Agregaciones

La agregación es un tipo de asociación que indica que una clase es parte de otra clase (composición débil).

Los objetos miembros son parte del objeto general, pero el objeto miembro puede existir independientemente del objeto general.

La destrucción del compuesto no conlleva la destrucción de los componentes.

Se representa en UML mediante un diamante de color blanco colocado en el extremo en el que está la clase que representa el “todo”.

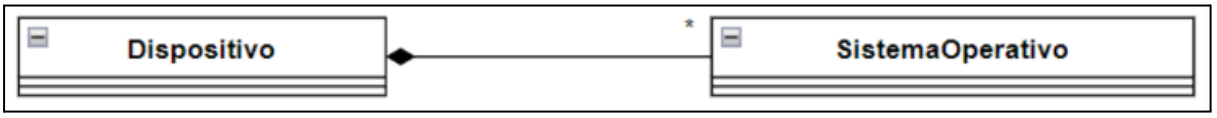


Por ejemplo, las palabras son parte de un diccionario, pero existen por fuera de él.

Composiciones

Los componentes constituyen una parte del objeto compuesto. La supresión del objeto compuesto conlleva la supresión de los componentes.

El símbolo de composición es un diamante de color negro colocado en el extremo en el que está la clase que representa el “todo”.

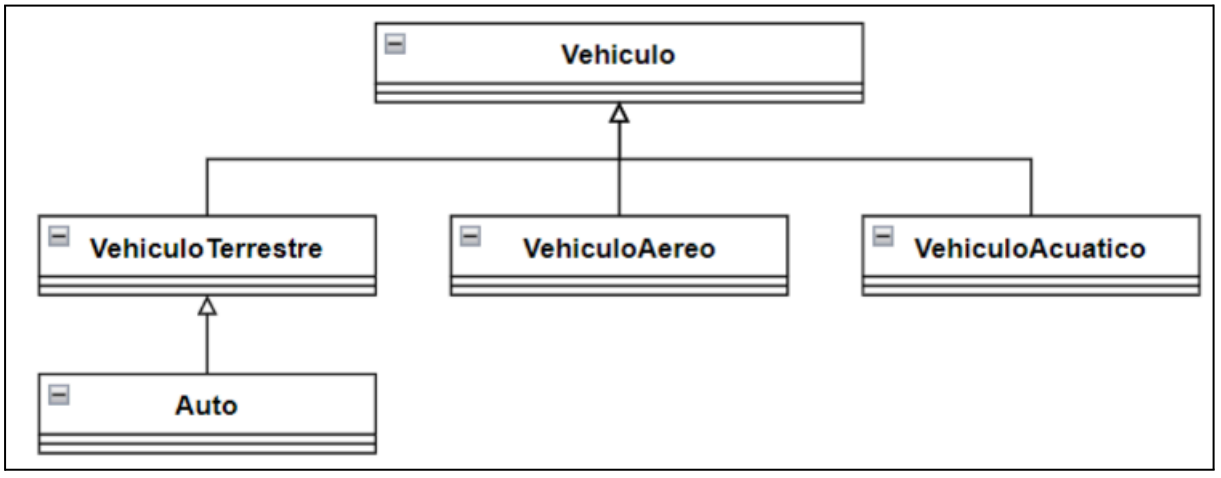


Podemos decir que un celular está compuesto de su SistemaOperativo y que si el Dispositivo desaparece, también desaparece el SistemaOperativoDedicado.

Herencia

Las características (atributos y operaciones) y asociaciones que se especifican para una clase general (superclase) se pasan a sus subclases.

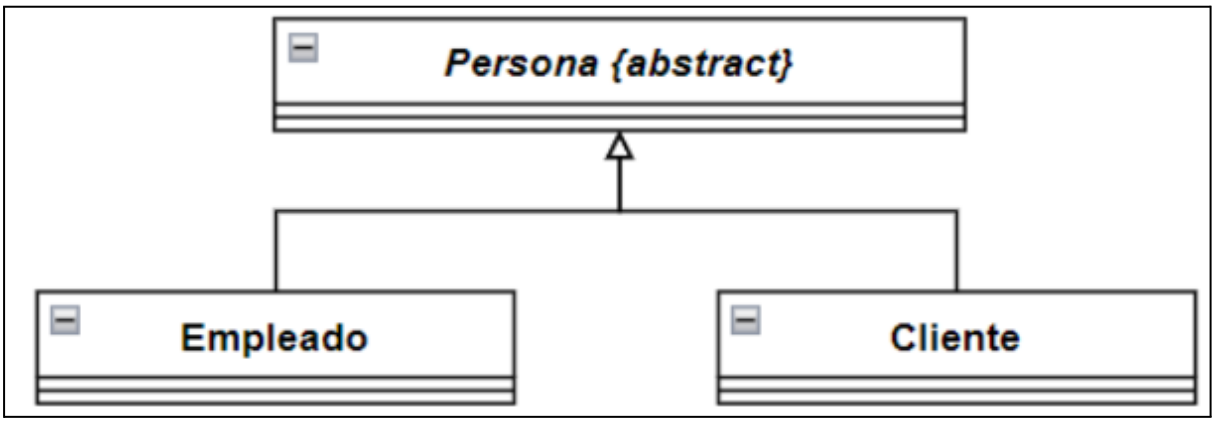
La subclase "posee"todos los atributos de instancia y atributos de clase y todas las operaciones de instancia y operaciones de clase de la superclase **siempre que no hayan sido marcadas con visibilidad privada.**



Clases abstractas

No se pueden crear instancias por sí mismas. Estas son clases para las cuales no hay objetos; sólo se pueden crear instancias de sus subclases.

Se utilizan exclusivamente para resaltar características comunes de sus subclases.



Supongamos que estamos diseñando un sistema para una empresa, entonces las clases Empleado y Cliente heredan de la clase abstracta Persona. No tiene sentido para el contexto del problema crear un objeto de la clase Persona, ya que se acotó el contexto a las personas que o son clientes o son empleados.