Programación 2

Herencia y Polimorfismo

Herencia

El proceso de clasificación realizado en un diseño orientado a objetos se organiza en niveles.

En el primer nivel los objetos se agrupan en **clases** de acuerdo a sus atributos y comportamientos.

En el segundo nivel del proceso de clasificación las clases se estructuran a través de un mecanismo de **especialización-generalización** llamado **herencia**.

La herencia favorece la **reusabilidad** y la **extensibilidad** del software.

Caso de estudio: Empresa - clientes y empleados

Una empresa mantiene información referida a sus empleados y clientes, en el primer caso con el objetivo fundamental de liquidar sueldos y en el segundo para mantener el saldo en cuenta corriente que se actualiza a partir de ventas y cobros.

El sueldo de un empleado se calcula a partir de su antigüedad en la empresa, su salario básico y la cantidad de hijos. Esto es, al salario básico se le suma $1000 si tiene entre 10 y 15 años de antigüedad y $2000 si tiene más de 15 años de antigüedad. A este valor se le suma un valor dado por cada hijo.

Los días de vacaciones se calculan a partir de la antigüedad. Una semana cuando cumple 1 año, 2 semanas cuando cumple 5 años y 3 semanas cuando su antigüedad es mayor a 10 años.

Caso de estudio: Empresa - clientes y empleados

*Cada empleado tiene un **nombre** y algunos datos postales como **calle, número, teléfono** y **dirección de correo electrónico**, una **fecha de ingreso** a la empresa, un **salario básico** y una **cantidad de hijos**.*

Durante su permanencia en la empresa pueden modificarse los datos postales, el salario o la cantidad de hijos.

Caso de estudio: Empresa - clientes y empleados

*Cada cliente tiene un **nombre** y algunos datos postales como **calle, número, teléfono** y **dirección de correo electrónico**, una **ciudad**, un **saldo adeudado**, un **número de CUIT** y una **categoría de IVA**.*

El saldo aumenta cada vez que se registra una operación de venta y disminuye con los cobros.

Los clientes también pueden tener hijos pero esta información no es relevante para la empresa. Del mismo modo aunque los empleados también viven en una ciudad, todos viven en la misma, de modo que no es necesario representarla como una variable de instancia.

Caso de estudio: Empresa - clientes y empleados

El cliente efectúa compras y pagos de facturas. Ninguno de estos servicios tiene significado para un Empleado, así como no tiene sentido calcular el sueldo de un Cliente.

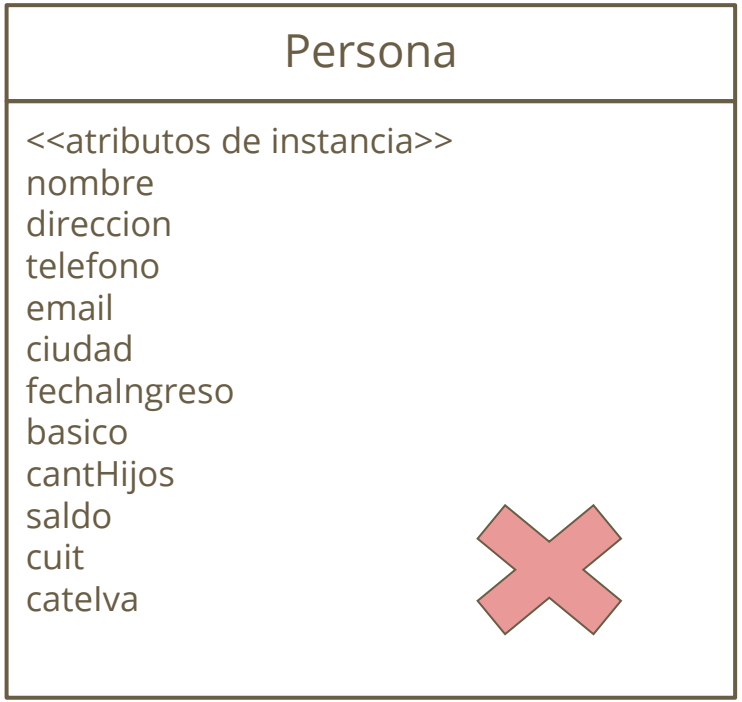
En realidad podría darse el caso que un Empleado de la empresa en ocasiones sea Cliente de la misma.

En ese caso una misma entidad del problema está cumpliendo dos roles, pertenece a dos clases. El modelo es entonces un poco más complejo, por el momento no nos ocuparemos de este tipo de situaciones.

Caso de estudio: Empresa - clientes y empleados

Si agrupamos a empleados y clientes en una única clase Persona, por ejemplo, los atributos podrían ser:

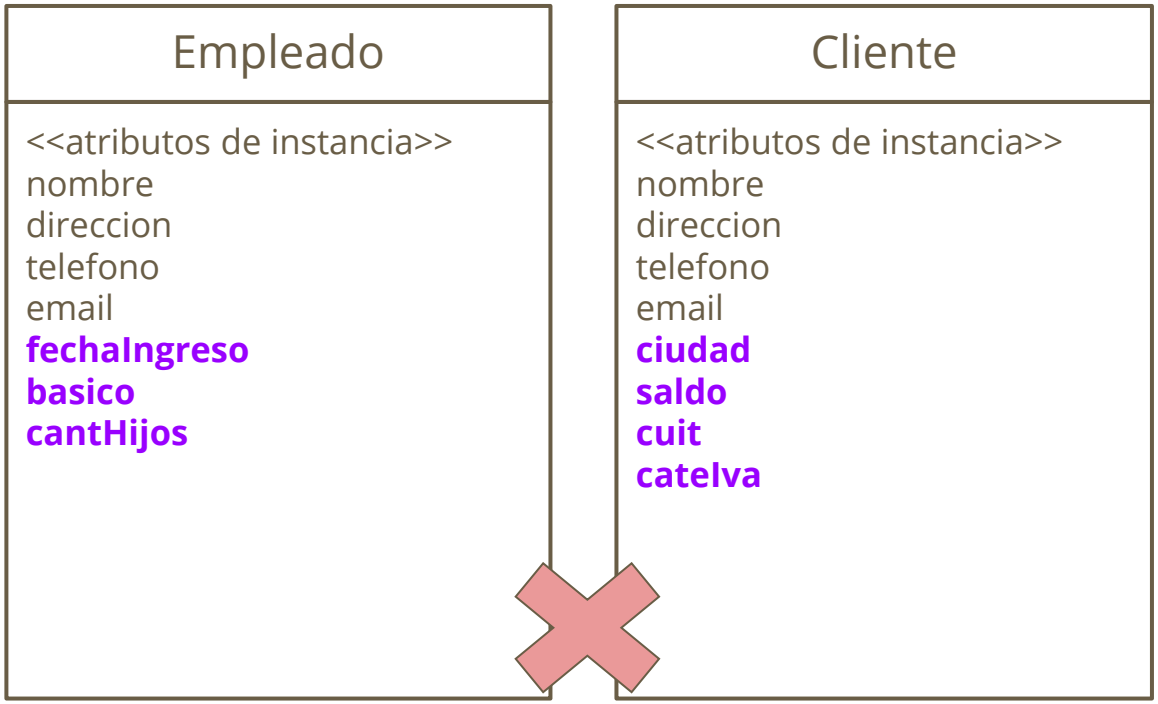
Esta alternativa no es adecuada porque el modelo incluye más atributos y comportamiento que el que realmente caracteriza a un empleado o cliente específico.



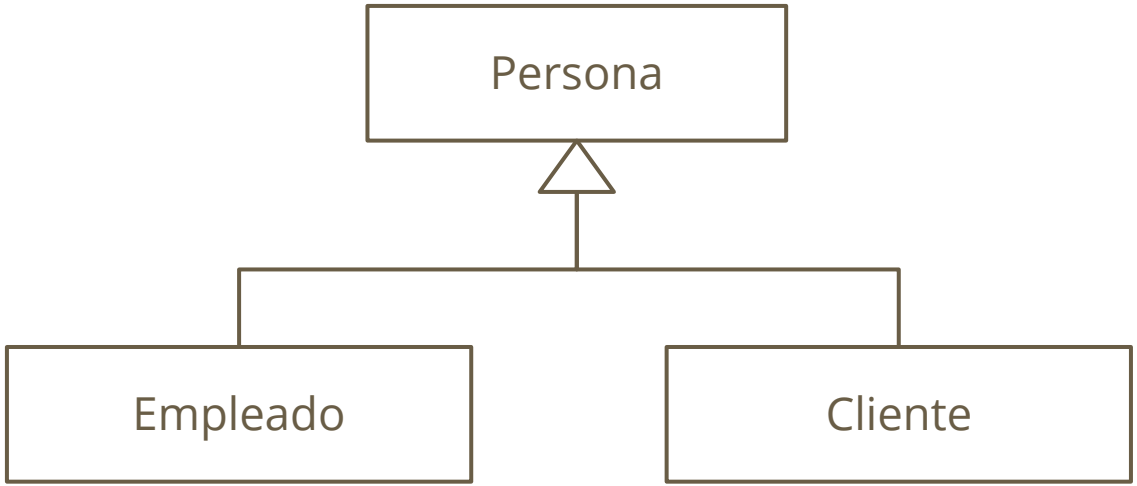
Caso de estudio: Empresa - clientes y empleados

Si definimos un modelo para cada grupo de entidades:

Esta alternativa no es adecuada porque tiene redundancia, parte de los atributos y del comportamiento se repite en las dos clases.



Caso de estudio: Empresa - clientes y empleados



Un modelo más adecuado factoriza los atributos y comportamiento compartidos de Cliente y Empleado en una **clase general** y retiene los atributos y comportamientos específicos en **clases especializadas**.

Caso de estudio: Empresa - clientes y empleados

Las clases Cliente y Empleado se vinculan con la clase Persona a través de un relación de **herencia**.

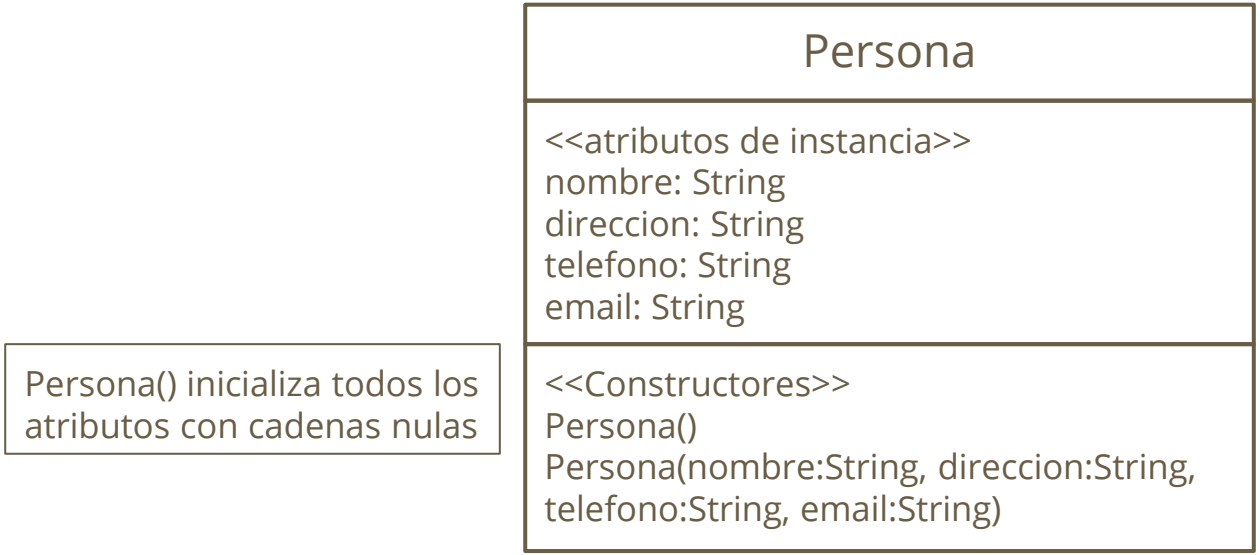
La organización es **jerárquica**, la clase Persona es más general y las clases Cliente y Empleado son más específicas.

La colección de clases se dibuja como un **árbol**, las clases generales ocupan los niveles superiores y las clases más específicas ocupan los niveles inferiores.

En Python la clase más general es Object.

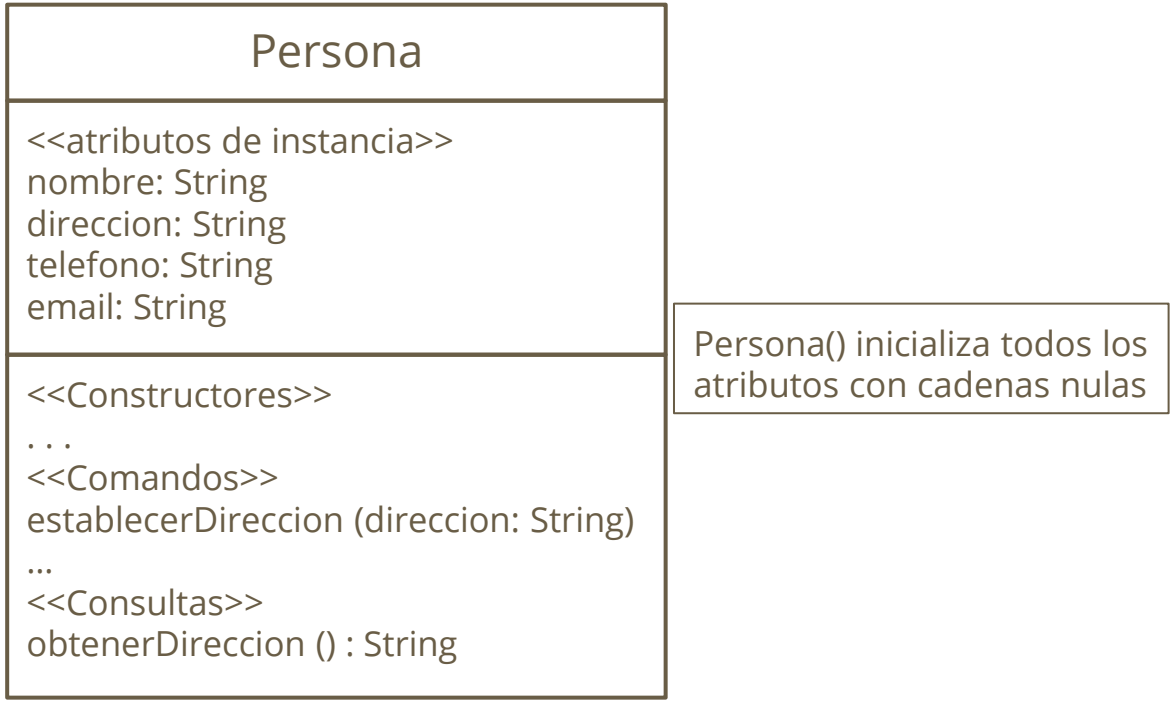
Todas las clases que implementamos heredan implícitamente de Object.

Caso de estudio: Empresa - clientes y empleados



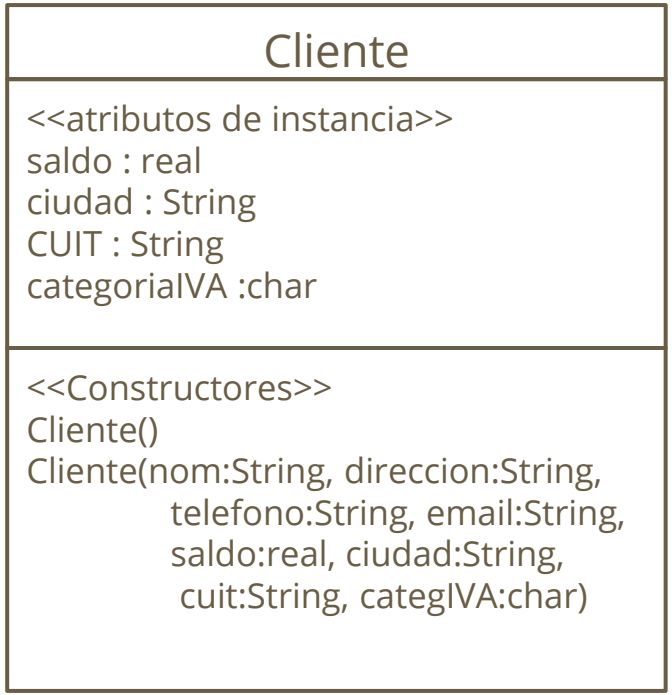
Caso de estudio: Empresa - clientes y empleados

Implementamos todos los métodos que son comunes a los clientes y empleados.



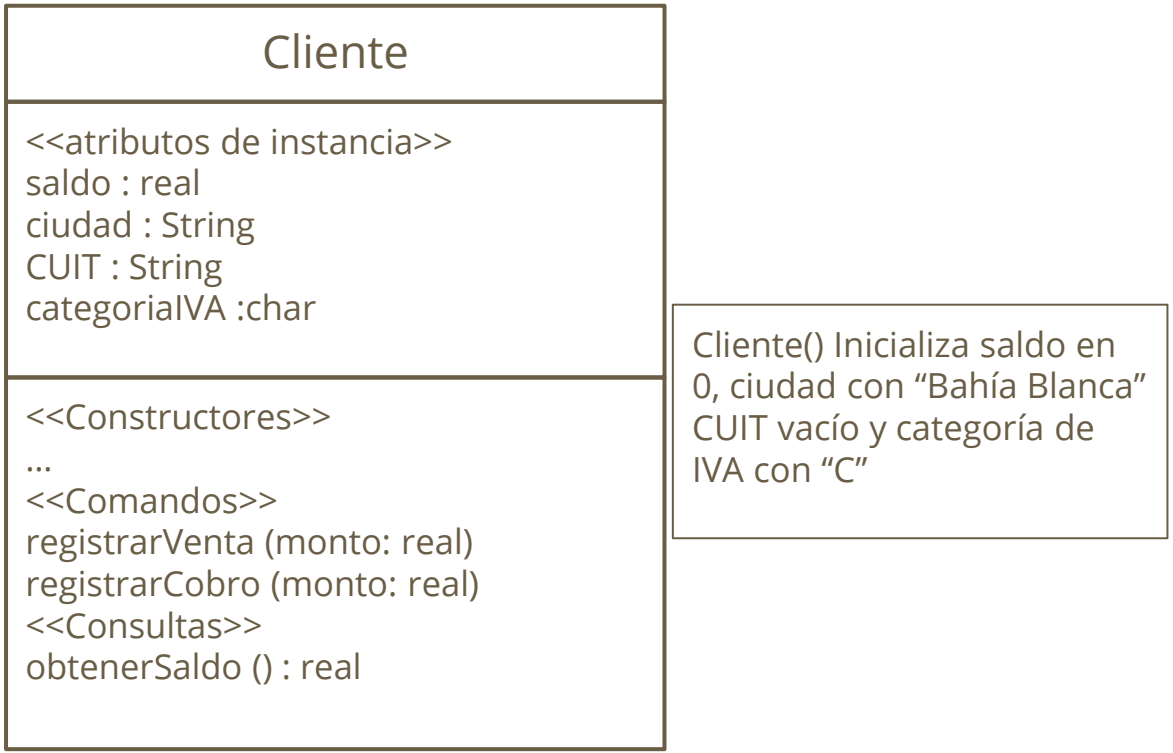
Caso de estudio: Empresa - clientes y empleados

Implementamos los métodos que son específicos de los clientes.



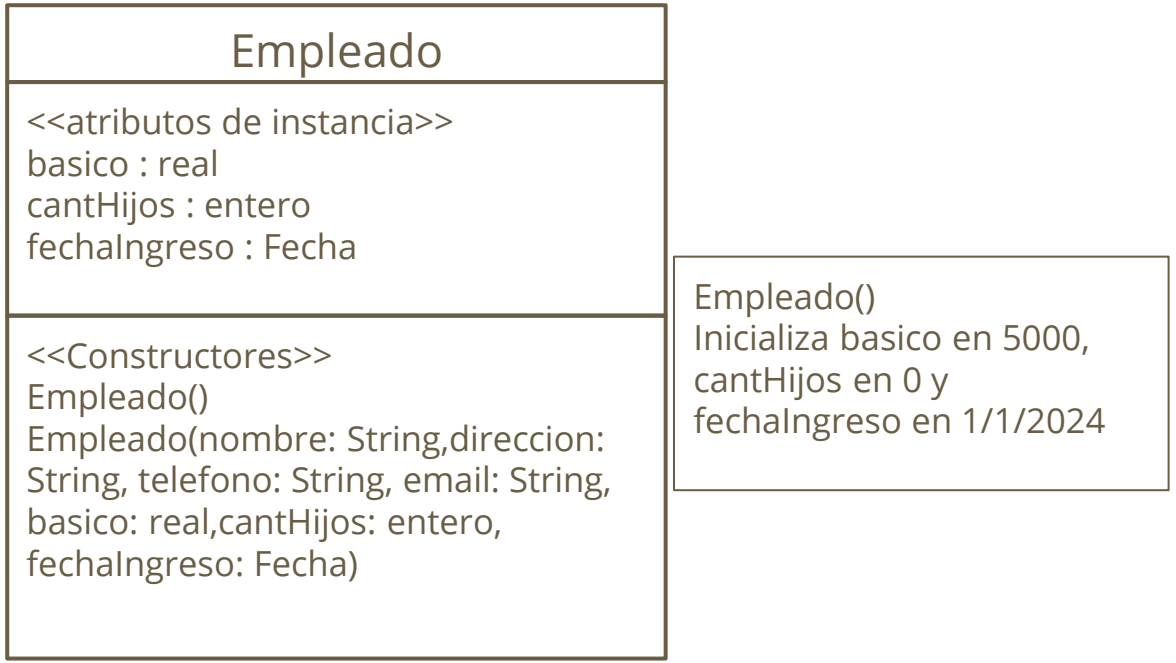
Cliente() Inicializa saldo en 0, ciudad con “Bahía Blanca” CUIT vacío y categoría de IVA con “C”

Caso de estudio: Empresa - clientes y empleados



Caso de estudio: Empresa - clientes y empleados

Implementamos los métodos específicos de los Empleados.



Caso de estudio: Empresa - clientes y empleados

Empleado
<<atributos de instancia>> basico : real cantHijos : entero fechaIngreso : Fecha
<<Constructores>> ... <<Comandos>> establecerBasico (basico: real) aumentarBasico (aumento: real) ... <<Consultas>> masAntiguo(empleado: Empleado): booleano obtenerBasico () : real sueldoNeto (montoPorHijo: real, fecha:Fecha): real diasVacaciones () : entero

masAntiguo(empleado: Empleado): booleano

Retorna verdadero si el empleado que recibe el mensaje es más antiguo que el argumento

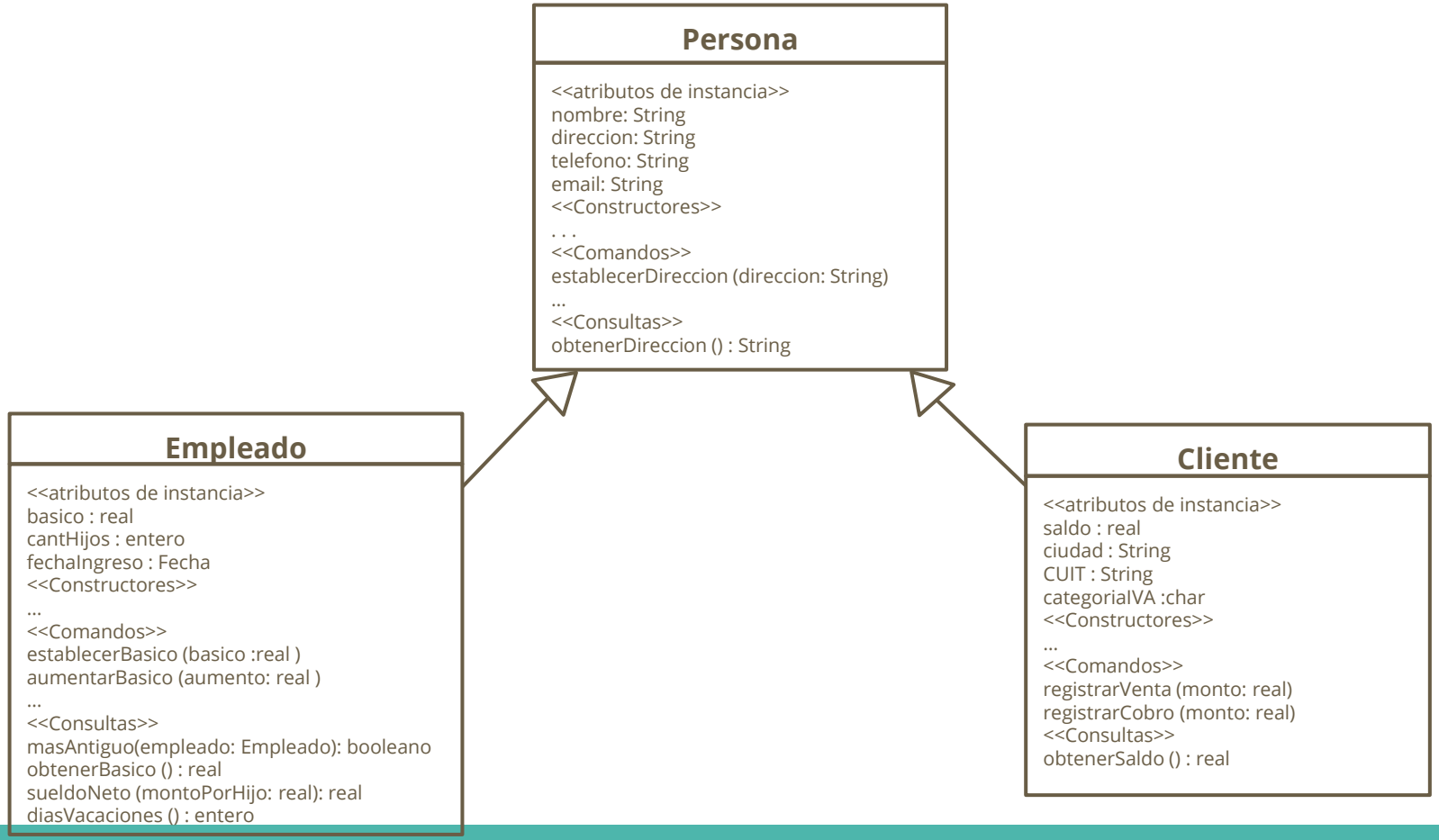
sueldoNeto (montoPorHijo: real, fecha: Fecha): real

Retorna el salario básico más $1000 si tiene entre 10 y 15 años de antigüedad y $2000 si tiene más de 15 años de antigüedad. A este valor se le suma un valor por cada hijo.

diasVacaciones () : entero

Retorna 7 días si la antigüedad es mayor a 1 año y menor a 5, 15 días a partir de 5 años y 21 días a partir de 10 años.

Caso de estudio: Empresa - clientes y empleados



Caso de estudio: Empresa - clientes y empleados

Las clases Cliente y Empleado **especializan** a la clase Persona.

Alternativamente podemos decir que Persona **generaliza** a las clases Cliente y Empleado.

Cliente y Empleado son **subclases** o **clases derivadas** de la **superclase** o **clase base** Persona.

El acceso a los atributos y servicios de una clase desde sus clases derivadas depende del **nivel de encapsulamiento**.

Caso de estudio: Empresa - clientes y empleados

```
class Persona:
    def __init__(self, nombre:str = "", direccion:str = "", telefono:str = "", email:str = ""):
        if not isinstance(nombre, str) or nombre.isspace():
            raise ValueError("El nombre debe ser un string válido.")
        if not isinstance(direccion, str) or direccion.isspace():
            raise ValueError("La dirección debe ser un string válido.")
        if not isinstance(telefono, str) or telefono.isspace():
            raise ValueError("El teléfono debe ser un string válido.")
        if not isinstance(email, str) or email.isspace():
            raise ValueError("El email debe ser un string válido.")
        self._nombre = nombre
        self._direccion = direccion
        self._telefono = telefono
        self._email = email
```

Atributos protegidos para poder accederlos desde las clases derivadas

Caso de estudio: Empresa - clientes y empleados

```
class Empleado(Persona):
    def __init__(self, nombre:str = "", direccion:str = "", telefono:str = "", email:str = "",
basico:float=5000.0, cantHijos:int=0, fechaIngreso:Fecha=Fecha(1,1,2024)):
        super().__init__(nombre, direccion, telefono, email)
        if not isinstance(basico, (int,float)) or basico < 0:
            raise ValueError("El sueldo básico debe ser un número positivo.")
        if not isinstance(cantHijos, int) or cantHijos < 0:
            raise ValueError("La cantidad de hijos debe ser un número entero positivo.")
        if not isinstance(fechaIngreso, Fecha):
            raise ValueError("La fecha de ingreso debe ser un objeto de tipo Fecha.")
        self.__basico = basico
        self.__cantHijos = cantHijos
        self.__fechaIngreso = fechaIngreso
```

Atributos privados, no hay subclases que requieran accederlos

Las clases derivadas de la clase Persona acceden al constructor usando el comando `super()`.

Caso de estudio: Empresa - clientes y empleados

```
class Cliente(Persona):
    def __init__(self, nombre:str = "", direccion:str = "", telefono:str = "", email:str = "",
saldo:float=0.0, ciudad:str="Bahía Blanca", CUIT:str="", categoriaIVA:str="C"):
        super().__init__(nombre, direccion, telefono, email)
        if not isinstance(saldo, (int,float)) or saldo < 0:
            raise ValueError("El descuento debe ser un número positivo.")
        if not isinstance(ciudad, str) or ciudad.isspace():
            raise ValueError("La ciudad debe ser un string válido.")
        if not isinstance(CUIT, str) or CUIT.isspace():
            raise ValueError("El CUIT debe ser un string válido.")
        if not isinstance(categoriaIVA, str) or categoriaIVA.isspace():
            raise ValueError("La categoría de IVA debe ser un string válido.")
        self.__saldo = saldo
        self.__ciudad = ciudad
        self.__CUIT = CUIT
        self.__categoriaIVA = categoriaIVA
```

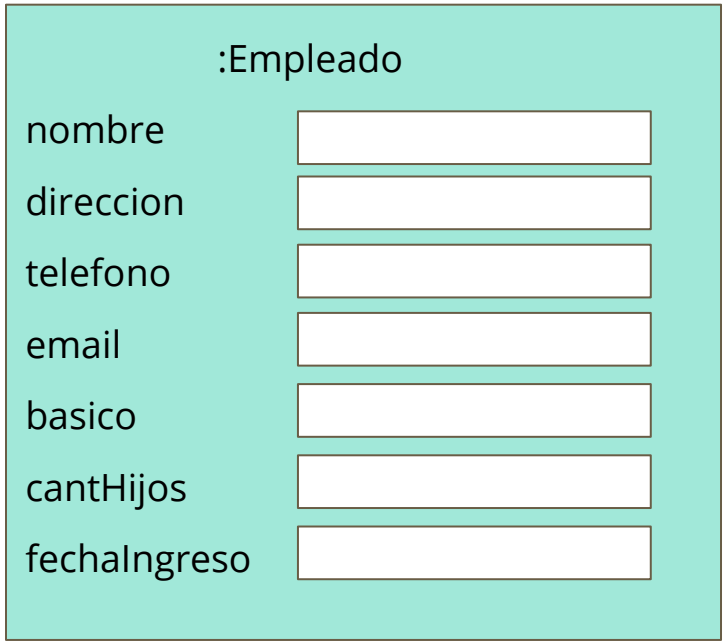
Atributos privados, no hay subclases que requieran accederlos

Las clases derivadas de la clase Persona acceden al constructor usando el comando `super()`.

Caso de estudio: Empresa - clientes y empleados

La clase **`Empleado`** hereda de la clase **`Persona`** todas las variables de clase, de instancia y métodos.

Un objeto de clase Empleado tiene todos los atributos de una Persona **más** los específicos de su clase.



Caso de estudio: Empresa - clientes y empleados

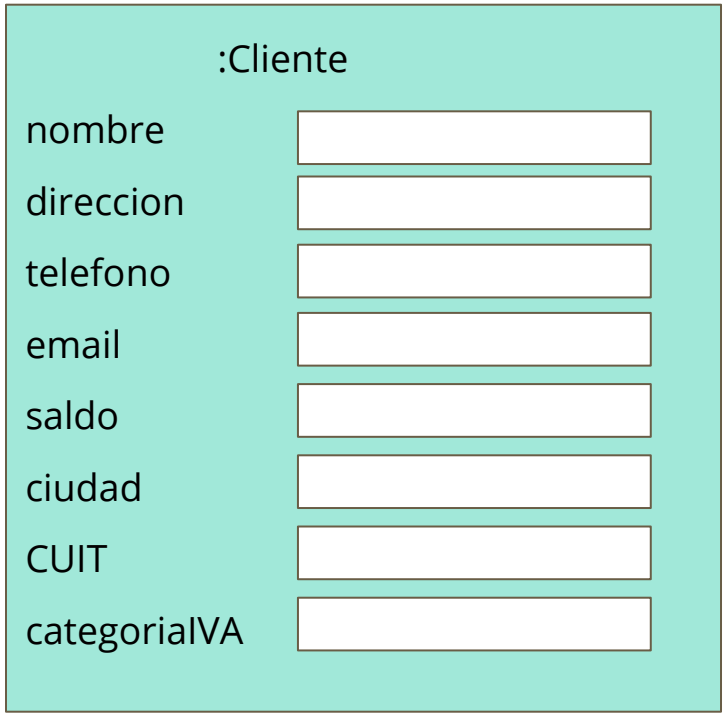
```
class GestionEmpresa:
    @staticmethod
    def gestionar():
        mariaGonzalez = Empleado("María González", "Av. Alem 123", "1234567",
"mail@empresa.com", 30000, 2, Fecha(1,1,2010))
        print(mariaGonzalez.getNombre())
        print(mariaGonzalez.getDireccion())
        print(mariaGonzalez.getBasico())
        print(mariaGonzalez.sueldoNeto(1000, Fecha(2,10,2024)))
```

Un objeto de clase **Empleado** puede recibir todos los mensajes válidos para los objetos de clase **Persona** más los específicos de su clase.

Caso de estudio: Empresa - clientes y empleados

La clase **`Cliente`** hereda de la clase **`Persona`** todas las variables de clase, de instancia y métodos.

Un objeto de clase Cliente tiene todos los atributos de una Persona **<u>más</u>** los específicos de su clase.



Caso de estudio: Empresa - clientes y empleados

```
class GestionEmpresa:
    @staticmethod
    def gestionar():
        cliente = Cliente("Juan Pérez", "Av. Mitre 123", "1234567", "mailCli@mail.com",
ciudad= "Bahía Blanca", CUIT= "20-12345678-9", categoriaIVA="C")
        print(cliente.getNombre())
        print(cliente.getDireccion())
        print(cliente.getSaldo())
        print(cliente.getCiudad())
```

Un objeto de clase **`Cliente`** puede recibir todos los mensajes válidos para los objetos de clase **`Persona`** más los específicos de su clase.

Caso de estudio: Empresa - clientes y empleados

La herencia establece una relación de tipo **es-un** ya que todo objeto de clase Cliente es también un objeto de clase Persona. Un objeto de clase Cliente estará caracterizado por todos los atributos y servicios definidos en la clase, pero además **hereda** los atributos y servicios de la clase Persona.

Análogamente entre la clase Empleado y la clase Persona existe una relación de tipo **es-un**. Una objeto de clase Empleado está caracterizado por todos los atributos definidos en su clase , pero además **hereda** los atributos y servicios de la clase Persona.

Un objeto de clase Persona no puede recibir mensajes que correspondan a servicios provistos por las clases derivadas.

Caso de estudio: Empresa - clientes y empleados

La empresa decide establecer un monto por productividad y un presupuesto para cada ejecutivo y un monto diario para viáticos común para todos los empleados ejecutivos.

Además se decide que todos los ejecutivos tengan 20 días de vacaciones.

Cada ejecutivo recibe un premio anual equivalente al 3% de su productividad.

Caso de estudio: Empresa - clientes y empleados

Ejecutivo

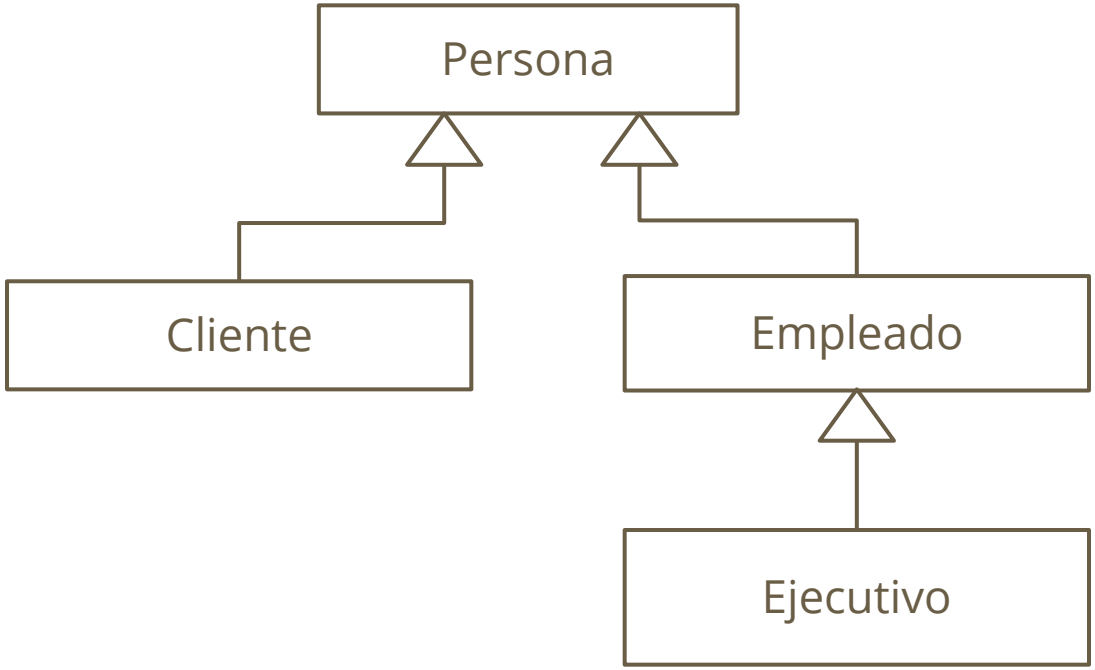
<<atributos de clase>>
viaticos = 1200
<<atributos de instancia>>
productividad: real
presupuesto: real

<<Constructores>>
Ejecutivo(nombre: String,
direccion: String,
telefono: String,
email: String,
basico: real,
cantHijos:entero,
fechaIngreso: Fecha,
presupuesto: real, productividad: real)
<<Comandos>>
establecerProductividad (productividad:real)
aumentarPresupuesto (aumento:real)
...
<<Consultas>>
obtenerPres () : real
diasVacaciones () : entero
premioAnual(): real

diasVacaciones(): entero
20 días de vacaciones

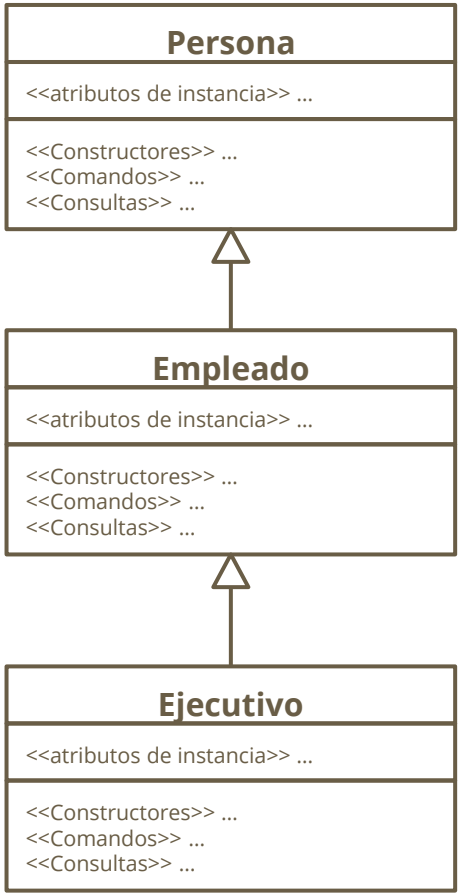
premioAnual(): real
3% de la productividad

Caso de estudio: Empresa - clientes y empleados



La clase Ejecutivo **especializa** a la clase Empleado. Todo objeto de clase Ejecutivo es también instancia de las clases Empleado y Persona.

Caso de estudio: Empresa - clientes y empleados



Caso de estudio: Empresa - clientes y empleados

```
class Persona:
    def __init__(self, nombre:str = "", direccion:str = "", telefono:str = "", email:str = ""):
        if not isinstance(nombre, str) or nombre.isspace():
            raise ValueError("El nombre debe ser un string válido.")
        if not isinstance(direccion, str) or direccion.isspace():
            raise ValueError("La dirección debe ser un string válido.")
        if not isinstance(telefono, str) or telefono.isspace():
            raise ValueError("El teléfono debe ser un string válido.")
        if not isinstance(email, str) or email.isspace():
            raise ValueError("El email debe ser un string válido.")
        self._nombre = nombre
        self._direccion = direccion
        self._telefono = telefono
        self._email = email
```

Atributos protegidos para poder accederlos desde las clases derivadas

Caso de estudio: Empresa - clientes y empleados

```
class Empleado(Persona):
    def __init__(self, nombre:str = "", direccion:str = "", telefono:str = "", email:str = "",
basico:float=5000.0, cantHijos:int=0, fechaIngreso:Fecha=Fecha(1,1,2024)):
        super().__init__(nombre, direccion, telefono, email)
        if not isinstance(basico, (int,float)) or basico < 0:
            raise ValueError("El sueldo básico debe ser un número positivo.")
        if not isinstance(cantHijos, int) or cantHijos < 0:
            raise ValueError("La cantidad de hijos debe ser un número entero positivo.")
        if not isinstance(fechaIngreso, Fecha):
            raise ValueError("La fecha de ingreso debe ser un objeto de tipo Fecha.")
        self._basico = basico
        self._cantHijos = cantHijos
        self._fechaIngreso = fechaIngreso
```

Atributos protegidos para poder accederlos desde las clases derivadas

Ahora que una subclase requiere acceder a los atributos y métodos de Empleado, deberemos modificar la privacidad de los atributos a “protegidos”

Caso de estudio: Empresa - clientes y empleados

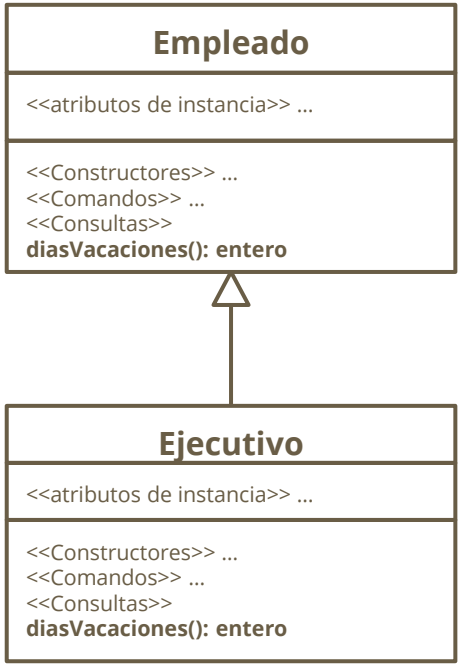
```
class Ejecutivo(Empleado):
    viaticos = 1200
    def __init__(self, nombre:str = "", direccion:str = "", telefono:str = "", email:str = "",
                 basico:float=0.0, cantHijos:int=0, fechaIngreso:Fecha=Fecha(1,1,2024),
                 productividad:float=0.0, presupuesto:float=0.0):
        super().__init__(nombre, direccion, telefono, email, basico, cantHijos, fechaIngreso)
        if not isinstance(productividad, (int, float)) or productividad<0:
            raise ValueError("La productividad debe ser un número positivo.")
        if not isinstance(presupuesto, (int, float)) or presupuesto<0:
            raise ValueError("El presupuesto debe ser un número positivo.")
        self.__productividad = productividad
        self.__presupuesto = presupuesto
```

Atributos privados, no hay subclases que requieran accederlos

Caso de estudio: Empresa - clientes y empleados

En python una clase derivada puede redefinir un método de una de sus clases ancestro, si especifica el **mismo nombre**.

Observación: en otros lenguajes como Java o C++ es obligatorio que el método en la subclase tenga la misma firma (es decir: el mismo nombre, los mismos parámetros y el mismo tipo de retorno) que en la clase base.



Caso de estudio: Empresa - clientes y empleados

```
class Empleado(Persona):
          . . .
    def diasVacaciones(self)->int:
        """Devuelve la cantidad de días de vacaciones de un empleado.
        Corresponden 7 días si la antigüedad es mayor a 1 año y menor a 5, 15 días a partir de
5 años y 21 días a partir de 10 años."""
        antiguedad = int(self._fechaIngreso.diferenciaHoy())
        if 0 < antiguedad < 5:
            vacaciones = 7
        elif antiguedad < 10:
            vacaciones = 15
        elif antiguedad >= 10:
            vacaciones = 21
        else:
            vacaciones = 0

        return vacaciones
```

Caso de estudio: Empresa - clientes y empleados

```
class Ejecutivo(Empleado):
    viaticos = 1200
    vacaciones = 20

        . . .

    def diasVacaciones(self) -> int:
        """Retorna los dias de vacaciones de un ejecutivo."""
        return Ejecutivo.vacaciones
```

Caso de estudio: Empresa - clientes y empleados

```
emp1 = Empleado("Pedro Pérez", "Av. Alem 123", "1234567", "mail@asd.com", 30000, 2, Fecha(1,1,2021))
emp2 = Ejecutivo("María González", "Mitre 2", "1234567", "mail@def.com", 30000, 2, Fecha(1,1,2020), 1000.65, 100000)
print(f"{emp1.getNombre()} es más antiguo que {emp2.getNombre()}?: {emp1.masAntiguo(emp2)}")
print(f"{emp2.getNombre()} es más antiguo que {emp1.getNombre()}?: {emp2.masAntiguo(emp1)}")
print(f"{emp1.getNombre()} sueldo neto en octubre/24: {emp1.sueldoNeto(1000, Fecha(1,10,2024))}")
print(f"{emp2.getNombre()} sueldo neto en octubre/24: {emp2.sueldoNeto(1000, Fecha(1,10,2024))}")
print(f"{emp1.getNombre()} días de vacaciones: {emp1.diasVacaciones()}")
print(f"{emp2.getNombre()} días de vacaciones: {emp2.diasVacaciones()}")
print("emp1 es de tipo Empleado?:", isinstance(emp1, Empleado))
print("emp1 es de tipo Ejecutivo?:", isinstance(emp1, Ejecutivo))
print("emp2 es de tipo Empleado?:", isinstance(emp2, Empleado))
print("emp2 es de tipo Ejecutivo?:", isinstance(emp2, Ejecutivo))
print("emp1 es de tipo Persona?:", isinstance(emp1, Persona))
print("emp2 es de tipo Persona?:", isinstance(emp2, Persona))
```

Caso de estudio: Empresa - clientes y empleados

```
emp1 = Empleado("Pedro Pérez", "Av. Alem 123", "1234567", "mail@asd.com", 30000, 2, Fecha(1,1,2021))
emp2 = Ejecutivo("María González", "Mitre 2", "1234567", "mail@def.com", 30000, 2, Fecha(1,1,2020), 1000.65, 100000)
print(f"{emp1.getNombre()} es más antiguo que {emp2.getNombre()}?: {emp1.masAntiguo(emp2)}")
print(f"{emp2.getNombre()} es más antiguo que {emp1.getNombre()}?: {emp2.masAntiguo(emp1)}")
print(f"{emp1.getNombre()} sueldo neto en octubre/24: {emp1.sueldoNeto(1000, Fecha(1,10,2024))}")
print(f"{emp2.getNombre()} sueldo neto en octubre/24: {emp2.sueldoNeto(1000, Fecha(1,10,2024))}")
print(f"{emp1.getNombre()} días de vacaciones: {emp1.diasVacaciones()}")
print(f"{emp2.getNombre()} días de vacaciones: {emp2.diasVacaciones()}")
print("emp1 es de tipo Empleado?:", isinstance(emp1, Empleado))
print("emp1 es de tipo Ejecutivo?:", isinstance(emp1, Ejecutivo))
print("emp2 es de tipo Empleado?:", isinstance(emp2, Empleado))
print("emp2 es de tipo Ejecutivo?:", isinstance(emp2, Ejecutivo))
print("emp1 es de tipo Persona?:", isinstance(emp1, Persona))
print("emp2 es de tipo Persona?:", isinstance(emp2, Persona))
```

```
Salida:
Pedro Pérez es más antiguo que María González?: False
María González es más antiguo que Pedro Pérez?: True
Pedro Pérez sueldo neto en octubre/24: 32000
María González sueldo neto en octubre/24: 32000
Pedro Pérez días de vacaciones: 7
María González días de vacaciones: 20
emp1 es de tipo Empleado?: True
emp1 es de tipo Ejecutivo?: False
emp2 es de tipo Empleado?: True
emp2 es de tipo Ejecutivo?: True
emp1 es de tipo Persona?: True
emp2 es de tipo Persona?: True
```

Caso de estudio: Empresa - clientes y empleados

Las variables `emp1` y `emp2` son **polimórficas**.
La ligadura entre el mensaje y el método es **dinámica**.

Las variables emp1 (de tipo Empleado) y emp2 (de tipo Ejecutivo) son consideradas polimórficas porque en tiempo de ejecución pueden referirse a diferentes tipos de objetos.

Polimorfismo en programación orientada a objetos permite que una misma variable (en este caso, de tipo Empleado) pueda contener referencias a objetos de diferentes clases (como Empleado y Ejecutivo), y cuando se invoca un método como diasVacaciones(), se ejecuta la versión correcta de acuerdo con el tipo real del objeto.

La ligadura dinámica significa que la decisión sobre qué método ejecutar se hace en tiempo de ejecución y no en tiempo de compilación.

Caso de estudio: Empresa - clientes y empleados

Polimorfismo: cuando llamamos al método `diasVacaciones()` en las variables `emp1` y `emp2`, aunque ambas son variables del tipo `Empleado`, en el caso de `emp2` se está refiriendo a un objeto de la subclase `Ejecutivo`, que ha redefinido ese método. Entonces, `diasVacaciones()` en `emp2` invocará la versión redefinida en `Ejecutivo`, mientras que en `emp1` se invocará la versión definida en `Empleado`. Esto es el comportamiento polimórfico: el mismo método invocado en diferentes objetos resulta en diferentes implementaciones según la clase real del objeto.

Ligadura dinámica: cuando el programa se ejecuta y llega a `emp2.diasVacaciones()`, Python determina en ese momento que `emp2` es en realidad un objeto de la clase `Ejecutivo`, por lo tanto llama al método `diasVacaciones()` de `Ejecutivo` y no al de `Empleado`, que es lo que llamaría si fuera simplemente un `Empleado`.

Esa "ligadura" (la conexión entre el mensaje o llamada al método y el método específico que se ejecuta) se realiza dinámicamente, en tiempo de ejecución, en función del tipo real del objeto.

Herencia

Para resolver - Sistema de peajes

Una ciudad desea implementar un sistema de peajes en una autopista. Por el momento desea cobrar peaje a los camiones, colectivos y autos, pero en un futuro quizá evalúe la posibilidad de cobrarle también a las motos.

El valor del peaje lo determina de acuerdo a las características del vehículo. En los camiones se determina de acuerdo a la cantidad de ejes que posee el camión y a la tara del mismo (100$ por eje + 5% de la tara). En los colectivos se determina de acuerdo a la cantidad de pasajeros que puede transportar ($60 por pasajero). En el caso de los autos el peaje se determina por el tipo de auto (sedan: 4 puertas, hatchback: 5 puertas) ($50 por puerta).

Cuando el sistema cobra el peaje del vehículo se emite un ticket con la fecha, hora, patente y valor del peaje. Por el momento a las motos no se les cobra pero se registra su paso por el peaje emitiéndoles el ticket.

Se pide:

1. Realizar el diagrama de clases en UML
2. Implementar las clases en python
3. Crear una clase tester que verifique los servicios de las clases generando distintos tipos de vehículos con valores aleatorios para los atributos