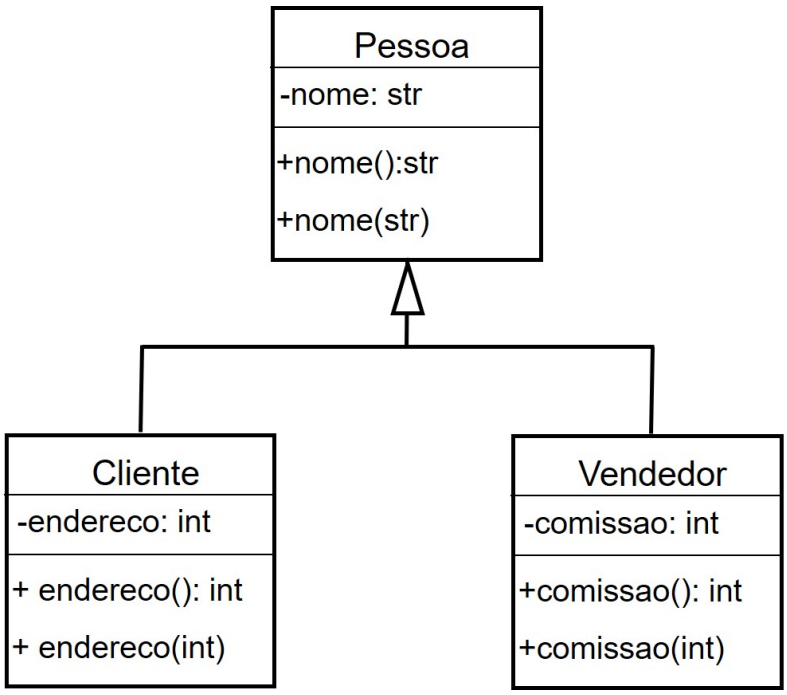
Herança

- Mecanismo que permite a reutilização daquilo que já foi implementado
- Define um relacionamento entre classes, onde verifica-se aquilo que é comum entre determinadas classes
- Uma classe compartilha a estrutura e/ou comportamento de uma ou mais classes
- É um relacionamento de especialização/ generalização (“é um” ou “tipo de”)

Vamos ver um exemplo prático:



</br>

> Nesse exemplo, as classes **Cliente** e **Vendedor** herdam de **Pessoa**. Ou seja, **Pessoa é uma Generalização e** Cliente**,** Vendedor **são especializações. Isso implica em dizer que:** Cliente e **Vendedor** herdam atributos e métodos de **Pessoa**.

Vamos implementar a classe Pessoa

Note que o atributo está com o símbolo: **"-"** no nome, o que indica que ele é privado, portanto deve iniciar com **"__"**.

São criados dois métodos especiais para cada atributo:

- *@property* que permite recuperar o dado do atributo
- *@NOME_DO_ATRIBUTO.setter* que permite alterar o dado do atributo

Como exemplo, incluímos um teste para verificar o tipo de dado que está sendo enviado utilizando `isinstance`

In [9]:

```
class Pessoa:

  def __init__(self, cpf: int, nome: str):
    if isinstance(cpf, int):
      self.__cpf = cpf
    if isinstance(nome, str):
      self.__nome = nome

  @property
  def cpf(self):
    return self.__cpf

  @cpf.setter
  def cpf(self,cpf: int):
    if isinstance(cpf, int):
      self.__cpf = cpf

  @property
  def nome(self):
    return self.__nome

  @nome.setter
  def nome(self,nome: str):
    if isinstance(nome, str):
      self.__nome = nome
```

Agora a classe Cliente

A classe **Cliente** herda de **Pessoa**, por isso o nome da classe vai entre parêntesis.

Construtor

O construtor da classe é o método `def __init__` que recebe como parâmetros `self, endereco: str, cpf: int, nome: str` :

- `self` representa o próprio objeto que está sendo instanciado. É nesse objeto que são criados os atributos
- `endereco` é um atributo da própria classe **Cliente**, por isso é armazenado em um atributo da própria classe: `self.__endereco = endereco`
- `cpf` e `nome` são atributos da classe **Pessoa** e por isso são repassados para a classe **Pessoa**, invocando-se o construtor da *superclasse*: `super().\_\_init\_\_(cpf, nome)`

In [10]:

```
class Cliente(Pessoa):

  def __init__(self, endereco: str, cpf: int, nome: str):
    super().__init__(cpf, nome)
    if isinstance(endereco, str):
      self.__endereco = endereco

  @property
  def endereco(self):
    return self.__endereco

  @endereco.setter
  def endereco(self,endereco: str):
    if isinstance(endereco, str):
      self.__endereco = endereco
```

Por fim a classe Vendedor

A classe **Vendedor** também herda de **Pessoa**, por isso o nome da classe vai entre parêntesis.

In [11]:

```
class Vendedor(Pessoa):

  def __init__(self, comissao: int, cpf: int, nome: str):
    super().__init__(cpf, nome)
    if isinstance(comissao, int):
      self.__comissao = comissao

  @property
  def comissao(self):
    return self.__comissao

  @comissao.setter
  def comissao(self,comissao: int):
    if isinstance(comissao, int):
      self.__comissao = comissao
```

Agora vamos brincar um pouco com os conceitos

Primeiro vamos instanciar objetos das três classes:

In [12]:

```
# Instanciando:

uma_pessoa = Pessoa(123, "Jean")
um_cliente = Cliente("Rua Geral, sem nº", 234, "Pedro")
um_vendedor = Vendedor(50, 456, "João")

# Imprimindo os valores dos atributos
print("Pessoa:", uma_pessoa.cpf, uma_pessoa.nome)
print("Cliente:", um_cliente.cpf, um_cliente.nome, um_cliente.endereco)
print("Vendedor:", um_vendedor.cpf, um_vendedor.nome, um_vendedor.comissao)
```

```
Pessoa: 123 Jean
Cliente: 234 Pedro Rua Geral, sem nº
Vendedor: 456 João 50
```

É interessante notar que os objetos `um_cliente` e `um_vendedor` têm os atributos cpf e nome, que herdaram de **Pessoa**

Entendendo o Encapsulamento

Mas, o que acontece se tentarmos acessar um atributo diretamente?

In [13]:

```
print("Pessoa:", uma_pessoa.__cpf)
```

```
---------------------------------------------------------------------------
-
AttributeError                            Traceback (most recent call las
t)
<ipython-input-13-a26a13e38548> in <module>
----> 1 print("Pessoa:", uma_pessoa.__cpf)

AttributeError: 'Pessoa' object has no attribute '__cpf'
```

Ocorre um erro informando que o objeto da classe **Pessoa** não possui o atributo `__cpf` . O atributo `__cpf` está ***encapsulado***, ou seja, está oculto na classe **Pessoa**, para evitar manipulaçoes incorretas.

O atributo `cpf` deve sempre ser acessado por meio dos métodos `@property` e `@setter` .

Vamos fazer um teste para ver o que exatamente acontece quando executamos `uma_pessoa.cpf` ?

Vamos reescrever a classe **Pessoa** para incluir um `print` no método *@property* para ver o que acontece:

In [20]:

```
class Pessoa:

  def __init__(self, cpf: int, nome: str):
    if isinstance(cpf, int):
      self.__cpf = cpf
    if isinstance(nome, str):
      self.__nome = nome

  @property
  def cpf(self):
    print(">> Executando o método property Pessoa.cpf")
    return self.__cpf

  @cpf.setter
  def cpf(self,cpf: int):
    print(">> Executando o método setter Pessoa.cpf")
    if isinstance(cpf, int):
      print(">> Valor válido para o setter Pessoa.cpf")
      self.__cpf = cpf
    else:
        print(">> Valor INVALIDO para o setter Pessoa.cpf:", cpf)

  @property
  def nome(self):
    return self.__nome

  @nome.setter
  def nome(self,nome: str):
    if isinstance(nome, str):
      self.__nome = nome
```

Agora vamos testar:

In [22]:

```
print("Instanciando Pessoa")
outra_pessoa = Pessoa(321, "Outra Pessoa")

print("Tentando alterar o CPF")
outra_pessoa.cpf = "ABC"

print("Pessoa: ", outra_pessoa.cpf)
```

```
Instanciando Pessoa
Tentando alterar o CPF
>> Executando o método setter Pessoa.cpf
>> Valor INVALIDO para o setter Pessoa.cpf: ABC
>> Executando o método property Pessoa.cpf
Pessoa:  321
```

Olha que interessante, `outra_pessoa.cpf = "ABC"` parece uma atribuição direta, mas não é!

Na prática, nesse momento o código do método **setter** é invocado, permitindo realizar o teste do tipo, evitando a atribuição de um valor inválido:

```
    @cpf.setter<br>
       def cpf(self,cpf: int):
```

Entendendo o construtor

Vamos agora entender melhor como funciona o construtor! Esse método especial em Python que é executado quando instanciamos um objeto de uma classe.

Para isso, vamos reescrever novamente só os construtores das classes **Pessoa** e **Cliente** incluindo algumas mensagens para ententer melhor o que está acontecendo:

In [24]:

```
class Pessoa:

  def __init__(self, cpf: int, nome: str):
    print(">> Executando o método construtor da classe Pessoa")
    if isinstance(cpf, int):
      self.__cpf = cpf
    if isinstance(nome, str):
      self.__nome = nome


class Cliente(Pessoa):

  def __init__(self, endereco: str, cpf: int, nome: str):
    print(">> Executando o método construtor da classe Cliente")
    super().__init__(cpf, nome)
    if isinstance(endereco, str):
      self.__endereco = endereco
```

Vamos agora brincar um pouco com esse código:

In [26]:

```
print("Instanciando uma Pessoa:")
pessoa = Pessoa(654, "Antonio")

print("Instanciando um Cliente:")
cliente = Cliente("Endereco", 987, "Ricardo")
```

```
Instanciando uma Pessoa:
>> Executando o método construtor da classe Pessoa
Instanciando um Cliente:
>> Executando o método construtor da classe Cliente
>> Executando o método construtor da classe Pessoa
```

O que aconeceu aqui?

Ao instanciar uma **Pessoa** o método construtor da classe foi executado.

Já ao instanciar um objeto da classe **Cliente**, foi iniciada a execução do método construtor da classe e logo é invocado o método `super().__init__(cpf, nome)` que vai chamar o construtor da classe-pai, a *superclasse* que, no caso é **Pessoa**.

Assim, é possível perceber, que ambos construtores são executados para instanciar um objeto da classe **Ciente**.