



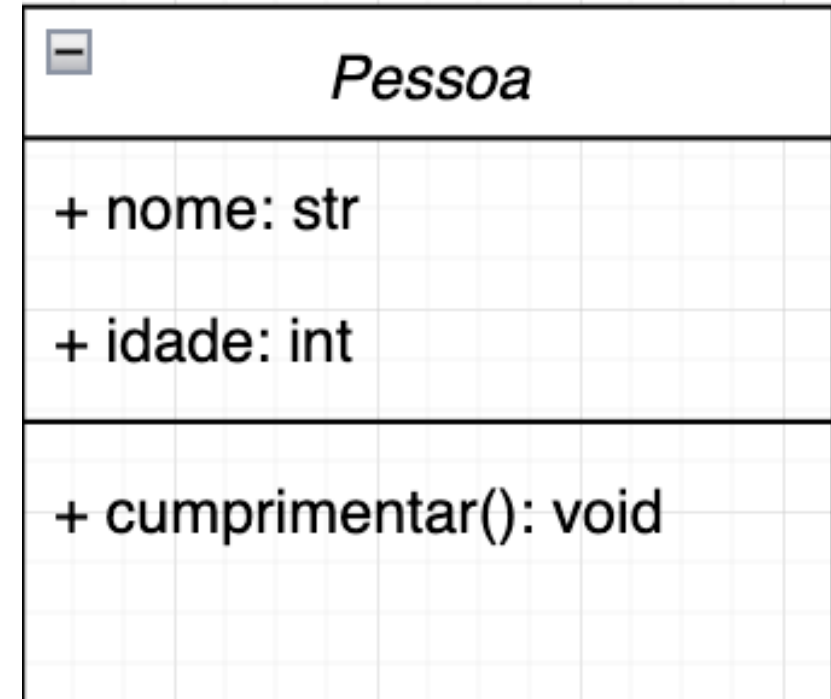
*Orientação a Objetos (OO) é um paradigma de programação que organiza o software em torno de objetos, que são instâncias de classes.*

*Esses objetos encapsulam dados (atributos) e comportamentos (métodos),*

*permitindo a criação de programas que são mais modularizados, reutilizáveis e fáceis de manter.*



*Uma classe é um molde ou uma estrutura que define um tipo específico de objeto. Ela agrupa dados e métodos que operam sobre esses dados. Em Python, uma classe é definida usando a palavra-chave class.*



# Classe

*\_\_init\_\_*: É o método construtor da classe, que é chamado automaticamente quando uma nova instância da classe é criada. Ele inicializa os atributos do objeto.

*self*: É uma referência à instância atual da classe e é usada para acessar variáveis que pertencem à classe.

python

```
class Pessoa:
    def __init__(self, nome, idade):
        self.nome = nome
        self.idade = idade

    def cumprimentar(self):
        print(f"Olá, meu nome é {self.nome} e eu tenho {self.idade} anos.")
```

*Um objeto é uma instância de uma classe. Quando você cria um objeto, você cria uma “cópia” da classe com seus próprios valores de atributos.*

## Classe



A  
T  
R  
I  
B  
U  
T  
O  
S  
  
M  
É  
T  
O  
D  
O  
S

## Objetos



Maria

|

Pedro

|



# Objetos

```
class Carro:
    def __init__(self, marca: str, modelo: str, ano: int):
        self.marca = marca
        self.modelo = modelo
        self.ano = ano

    def ligar_motor(self):
        print(f"O motor do {self.modelo} foi ligado.")

    def desligar_motor(self):
        print(f"O motor do {self.modelo} foi desligado.")
```

```
# Criando instâncias da classe Carro
carro1 = Carro(marca="Toyota", modelo="Corolla", ano=2022)
carro2 = Carro(marca="Honda", modelo="Civic", ano=2021)
```

```
# Usando métodos das instâncias
carro1.ligar_motor() # Saída: O motor do Corolla foi ligado.
carro2.desligar_motor() # Saída: O motor do Civic foi desligado.
```



*Atributos são as variáveis que pertencem a uma classe. Eles podem ser de instância (específicos para um objeto) ou de classe (compartilhados entre todas as instâncias da classe).*

python

```
class Carro:
    rodas = 4 # Atributo de classe

    def __init__(self, marca, modelo):
        self.marca = marca # Atributo de instância
        self.modelo = modelo # Atributo de instância
```



*Métodos são funções definidas dentro de uma classe que descrevem os comportamentos de um objeto.*

```
class ContaBancaria:
    def __init__(self, titular, saldo=0):
        self.titular = titular
        self.saldo = saldo

    def depositar(self, quantia):
        self.saldo += quantia

    def sacar(self, quantia):
        if quantia <= self.saldo:
            self.saldo -= quantia
            return True
        else:
            return False
```





*Herança é um mecanismo que permite que uma nova classe (subclasse) herde atributos e métodos de uma classe existente (superclasse). Isso promove o reuso de código e a criação de uma hierarquia de classes.*



```
class Veiculo:
    def __init__(self, marca: str, modelo: str, ano: int):
        self.marca = marca
        self.modelo = modelo
        self.ano = ano

    def ligar_motor(self):
        print(f"O motor do {self.modelo} foi ligado.")

    def desligar_motor(self):
        print(f"O motor do {self.modelo} foi desligado.")
```

```
class Carro(Veiculo):
    def __init__(self, marca: str, modelo: str, ano: int, num_portas: int):
        super().__init__(marca, modelo, ano) # Chama o construtor da superclasse
        self.num_portas = num_portas

    def abrir_porta_malas(self):
        print(f"O porta-malas do {self.modelo} foi aberto.")
```



```
class Veiculo:
    def __init__(self, marca: str, modelo: str, ano: int):
        self.marca = marca
        self.modelo = modelo
        self.ano = ano

    def ligar_motor(self):
        print(f"O motor do {self.modelo} foi ligado.")

    def desligar_motor(self):
        print(f"O motor do {self.modelo} foi desligado.")
```

```
class Moto(Veiculo):
    def __init__(self, marca: str, modelo: str, ano: int, cilindradas: int):
        super().__init__(marca, modelo, ano) # Chama o construtor da superclasse
        self.cilindradas = cilindradas


    def empinar(self):
        print(f"A {self.modelo} está empinando!")
```





## Exemplos de Instâncias

python

 Copiar

```
# Criando instâncias das subclasses
```

```
carro1 = Carro(marca="Toyota", modelo="Corolla", ano=2022, num_portas=4)  
moto1 = Moto(marca="Honda", modelo="CBR", ano=2021, cilindradas=1000)
```

```
# Usando métodos herdados da superclasse Veiculo
```

```
carro1.ligar_motor() # Saída: O motor do Corolla foi ligado.  
moto1.ligar_motor() # Saída: O motor do CBR foi ligado.
```

```
# Usando métodos específicos das subclasses
```

```
carro1.abrir_porta_malas() # Saída: O porta-malas do Corolla foi aberto.  
moto1.empinar()           # Saída: A CBR está empinando!
```



@fpftech.educacional

*Encapsulamento é o conceito de esconder os detalhes internos de um objeto e expor apenas o que é necessário. Em Python, isso é conseguido através de convenções como o uso de prefixos de sublinhado `_` para indicar atributos e métodos privados.*



```
class Conta:
    def __init__(self, titular, saldo=0):
        self.titular = titular
        self.__saldo = saldo # Atributo privado

    def depositar(self, quantia):
        self.__saldo += quantia

    def sacar(self, quantia):
        if quantia <= self.__saldo:
            self.__saldo -= quantia
            return True
        else:
            return False

    def get_saldo(self):
        return self.__saldo
```





```
class Pessoa:
```

```
    def __init__(self, nome, idade):
        self.__nome = nome        # Atributo privado
        self.__idade = idade      # Atributo privado

    def apresentar(self):
        print(f"Nome: {self.__nome}, Idade: {self.__idade}")
```

```
# Getters e Setters (encapsulamento)
```

```
def get_nome(self):
    return self.__nome
```

```
def set_nome(self, nome):
    self.__nome = nome
```

```
def get_idade(self):
    return self.__idade
```

```
def set_idade(self, idade):
    if idade >= 0:
        self.__idade = idade
    else:
        print("Idade inválida.")
```

```
# Classe Aluno herdando de Pessoa
```

```
class Aluno(Pessoa):
```

```
    def __init__(self, nome, idade, matricula):
        super().__init__(nome, idade)    # Chama o construtor da superclass
        self.__matricula = matricula     # Atributo privado
```

```
    def apresentar(self):
        super().apresentar()
        print(f"Matrícula: {self.__matricula}")
```

```
    def get_matricula(self):
        return self.__matricula
```

```
    def set_matricula(self, matricula):
        self.__matricula = matricula
```

# Encapsulamento

# Vantagens de usar atributos privados:

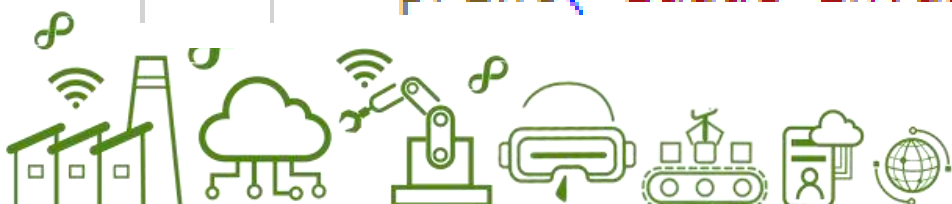
# 1. Proteção dos dados

# Evita que atributos sejam alterados diretamente de forma indevida.

```
aluno.idade = -10 # Se fosse público, isso seria possível e errado.
```

# Com `__idade`, isso só é possível através de um método como `set_idade`,  
# onde podemos validar:

```
def set_idade(self, idade):  
    if idade >= 0:  
        self.__idade = idade  
    else:  
        print("Idade inválida.")
```



- # 2. Facilidade para alterar a lógica interna
- # Você pode mudar a forma como o dado é armazenado sem afetar quem usa sua classe.
- # Exemplo: Hoje o nome está em texto simples. Amanhã, você pode decidir armazenar o nome em partes (nome, sobrenome) e reconstruir no get\_nome.
- # 3. Mais controle e segurança
- # Você controla como e quando os dados podem ser acessados ou modificados.
- # 4. Evita uso incorreto por quem usa a classe
- # Quando um atributo é público, qualquer um pode acessá-lo ou alterá-lo, mesmo que não devesse. Isso pode causar erros difíceis de rastrear.



# Encapsulamento

#  Exemplo comparando:

# Sem encapsulamento:

```
aluno.nome = ""
```

# Pode deixar nome vazio, mesmo que isso seja inválido

# Com encapsulamento:

```
aluno.set_nome("")
```

# Você pode validar dentro do método e impedir



# Por que usar POO?

*Reuso de Código: Com herança e polimorfismo, você pode reutilizar código em várias partes do programa.*

*Modularidade: Cada classe funciona como um módulo que pode ser desenvolvido e testado separadamente.*

*Facilidade de Manutenção: A estrutura clara e organizada facilita a identificação e correção de problemas.*

*Abstração: Permite focar em interfaces simples e abstrair os detalhes de implementação complexos.*



# Obrigado!



    @fpftech.educacional