INFINITY SCHOOL

VISUAL ART CREATIVE CENTER

AULA 13 - POO III - ASSOCIAÇÃO E ENCAPSULAMENTO

O QUE IREMOS APRENDER

01 RESUMO DA AULA PASSADA

02 CONTEXTUALIZAÇÃO DA AULA DE HOJE

03 ASSOCIAÇÃO

04 TIPOS DE ASSOCIAÇÃO (UNIDIRECIONAL E BIDIRECIONAL)

05 ENCAPSULAMENTO

06 IMPORTÂNCIA DO ENCAPSULAMENTO

07 GETTERS E SETTERS

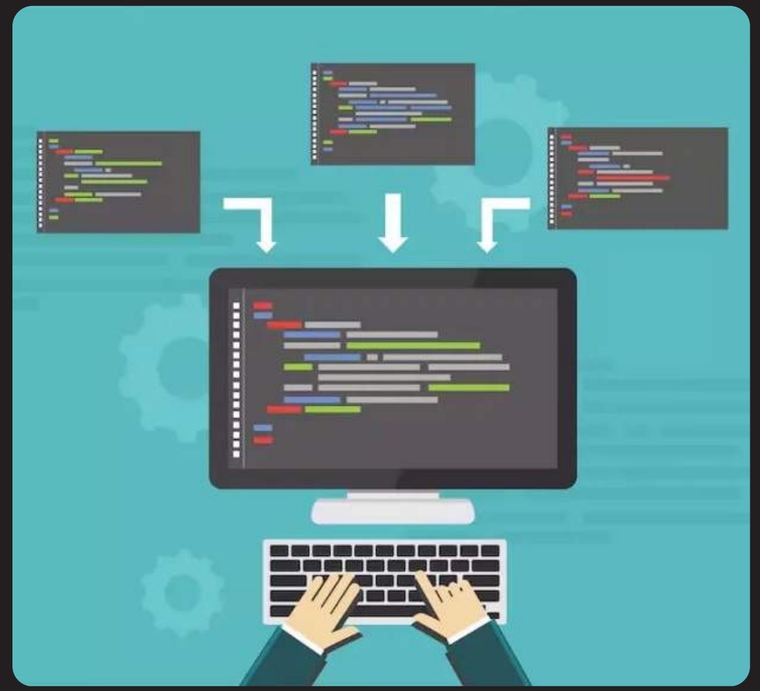
08 PROPERTY

09 MÃOS NO CÓDIGO

Resumo da aula passada

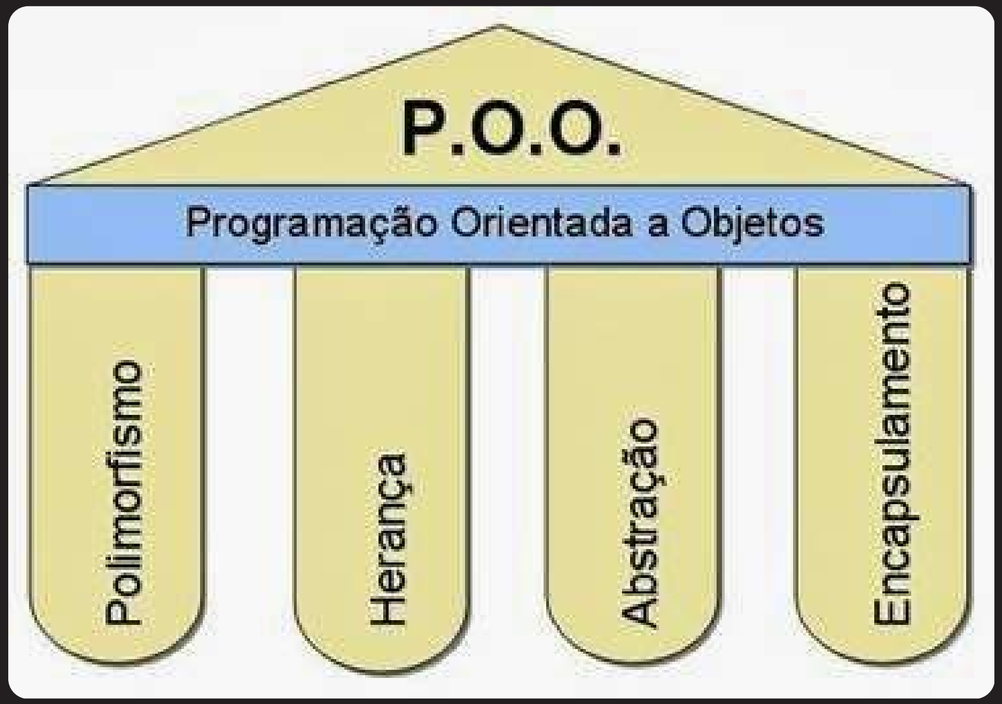
Programação Orientada a Objetos (POO) é um paradigma de programação que se baseia no conceito de "objetos" como a unidade fundamental de estruturação do código. Nesse paradigma, os objetos representam entidades do mundo real e possuem características (atributos) e ações (métodos) associadas a eles

Abstração, por outro lado, é um conceito relacionado à POO que envolve a identificação das características e comportamentos essenciais de um objeto, enquanto ignora detalhes irrelevantes.



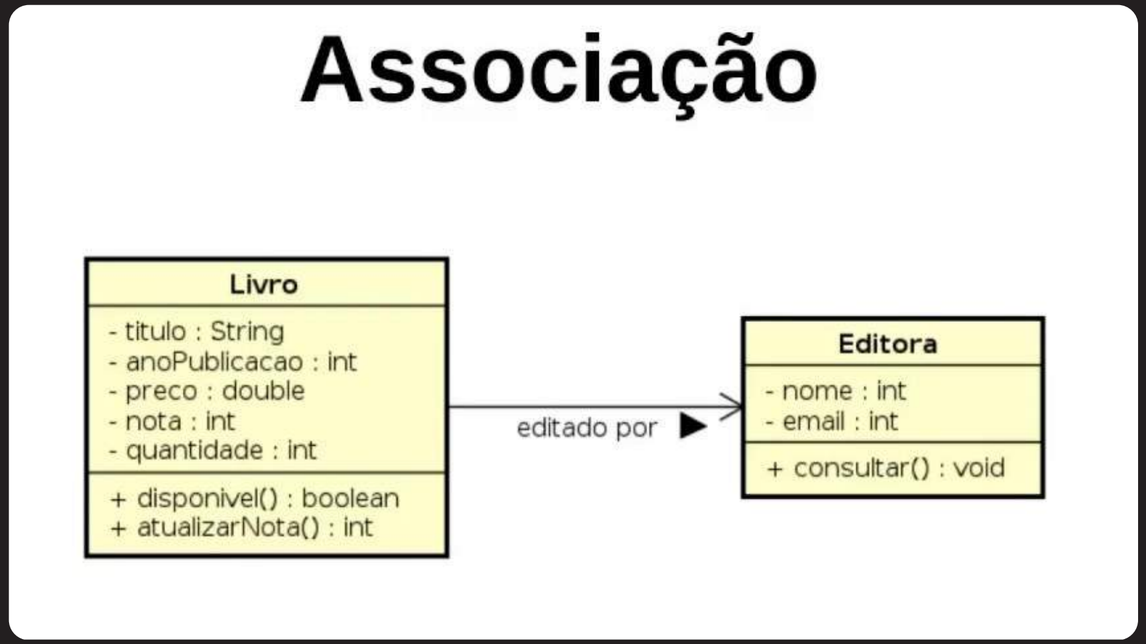
Contextualização da aula de hoje

Associação e o Encapsulamento desempenham um papel crucial na construção de software orientado a objetos de alta qualidade. Eles ajudam a modelar relacionamentos entre objetos, promovem a reutilização de código, facilitam a manutenção, protegem dados e a modularidade do sistema. Esses princípios são essenciais para o desenvolvimento de sistemas eficientes e de fácil manutenção.



Associação

Associação é um relacionamento entre duas ou mais classes, em que uma classe usa objetos de outra classe como parte de suas operações. Esses objetos são normalmente passados para a classe que usa através de parâmetros de método ou variáveis de instância. A associação pode ser de dois tipos: unidirecional ou bidirecional.



Tipos de Associação

Na associação unidirecional, uma classe usa objetos de outra classe, mas a outra classe não usa objetos da primeira classe. Na associação bidirecional, as duas classes usam objetos um do outro.

Associação Unidirecional

```python
class Autor:
    def __init__(self,nome):
        self.nome = nome

class Livro :
    def __init__(self, titulo, autor):
        self.titulo = titulo
        self.autor = autor

autor1 = Autor("João Silva")
livro1 = Livro("Aventuras na Floresta", autor1)
```

Neste exemplo, temos duas classes, Autor e Livro. A classe Livro está associada à classe Autor. Um livro é escrito por um autor, então a classe Livro possui um atributo autor, que é uma instância da classe Autor.

Associação Bidirecional

```python
class BiBlioteca:
    def __init__(self,nome):
        self.nome = nome
        self.livros = []

class Livro:
    def __init__(self, titulo, biblioteca):
        self.titulo = titulo
        self.biblioteca = None
        # Inicialmente, o livro não pertence
        # a nenhuma biblioteca

biblioteca1 = BiBlioteca("Biblioteca Central")
livro1 = Livro("Aventuras na Floresta")
livro1.biblioteca = biblioteca1
biblioteca1.livros.append(livro1)
```

Neste exemplo, temos duas classes, Biblioteca e Livro. A classe Biblioteca possui uma lista de livros que ela contém. Cada livro também tem uma referência à biblioteca à qual pertence. Isso permite que a biblioteca saiba quais livros ela contém e que os livros saibam a qual biblioteca pertencem.

Encapsulamento

O encapsulamento é um dos princípios fundamentais da Programação Orientada a Objetos (POO) e se refere à prática de esconder os detalhes internos de um objeto e disponibilizar uma interface controlada para interagir com esse objeto. Isso é alcançado por meio da definição de atributos como privados (ou protegidos) e fornecendo métodos públicos para acessar e modificar esses atributos.

“Encapsulamento é esconder o jogo!”

Importância do Encapsulamento

Proteção de Dados: O encapsulamento permite proteger os dados internos de uma classe, tornando os atributos privados ou protegidos.

Controle de Acesso: Através do encapsulamento, você define métodos públicos (getters e setters) que fornecem uma interface controlada para acessar e modificar os atributos.

Manutenção Simplificada: O encapsulamento torna a manutenção de código mais simples.

Encapsulamento

Um exemplo de encapsulamento na vida real pode ser encontrado em um cofre de segurança. Vamos considerar o cofre como uma analogia para uma classe em programação:

Cofre (Classe): O cofre é a entidade principal que deseja proteger os objetos de valor (atributos).

Senha (Método Público): Para abrir o cofre, é necessário fornecer a senha correta. A senha é a interface pública que permite o acesso controlado ao cofre. Sem a senha, você não pode abrir o cofre.

Encapsulamento

Mecanismo de Bloqueio (Implementação Interna): O mecanismo interno de bloqueio do cofre é complexo e seguro, mantido oculto para o usuário. Os detalhes internos do mecanismo de bloqueio são encapsulados e protegidos.

Objetos de Valor (Dados Privados): Dentro do cofre, você armazena objetos de valor, como dinheiro, jóias ou documentos importantes. Esses objetos são os dados privados que o cofre protege.

Encapsulamento

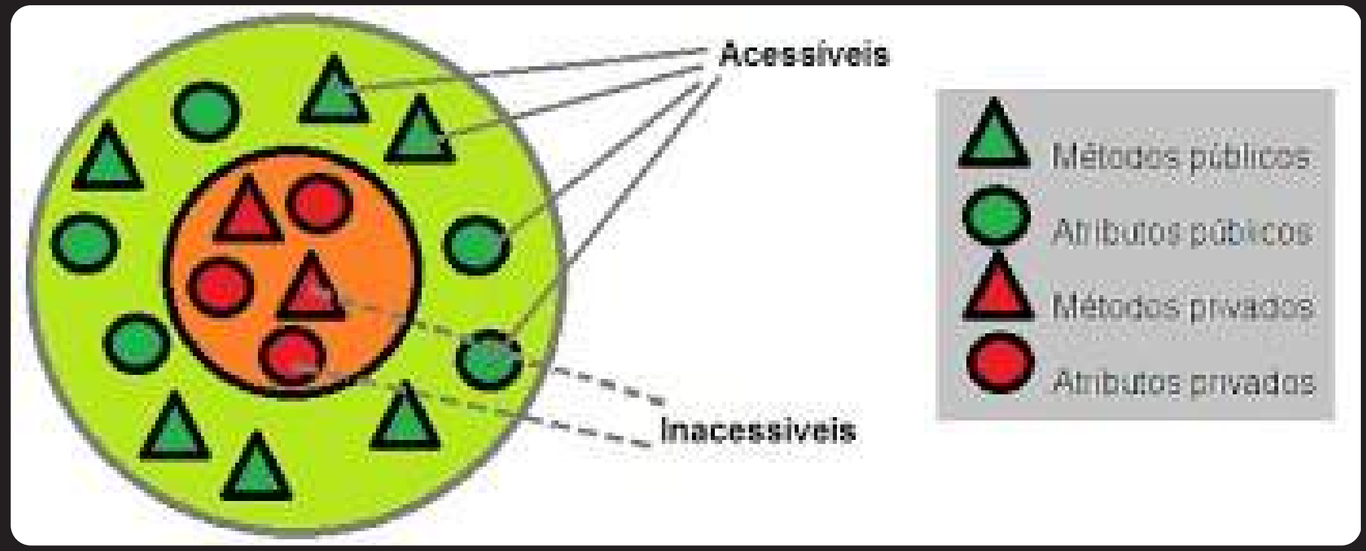
Vamos destrinchar o exemplo trazendo para a programação orientada a objeto (POO):

O mecanismo interno do cofre (mecanismo de bloqueio) é encapsulado e não é visível para quem quer abrir o cofre. Somente o método público (senha) pode interagir com o mecanismo interno.

Os objetos de valor (dados privados) são protegidos e não podem ser acessados diretamente de fora do cofre. Somente o método público (senha) permite acesso controlado aos objetos de valor.

Classe POO

O uso da senha como interface pública (método público) garante que apenas pessoas autorizadas possam acessar os objetos de valor do cofre, protegendo a integridade dos dados.



Getters e Setters

Getters e setters são métodos utilizados em Programação Orientada a Objetos para acessar e modificar atributos privados de uma classe, garantindo o encapsulamento dos dados. Eles são usados para permitir o acesso controlado a esses atributos, o que ajuda a manter a integridade dos dados e implementar regras de validação, se necessário.

Getters e Setters

Getter (Acesso): Um método "getter" é usado para obter o valor de um atributo privado. Ele fornece acesso somente leitura aos atributos, permitindo que outros objetos leiam seus valores.

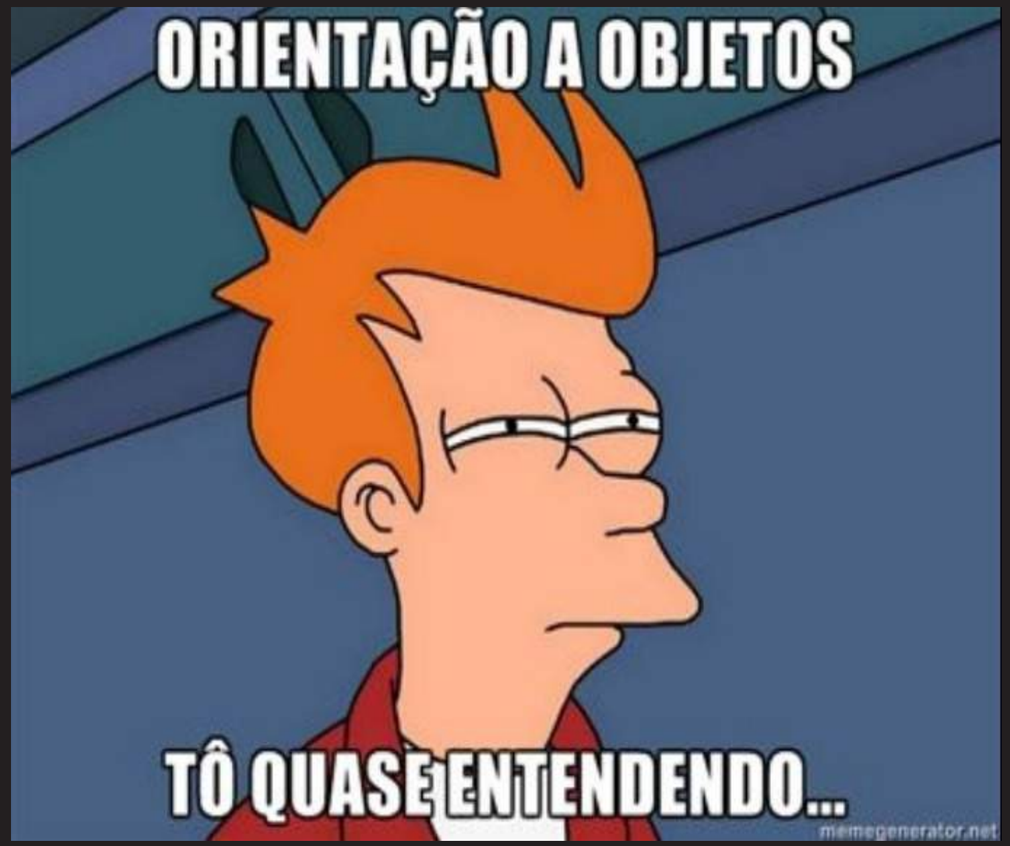
Setter (Modificação): Um método "setter" é usado para definir ou modificar o valor de um atributo privado. Ele fornece acesso somente de escrita aos atributos, permitindo que outros objetos alterem seus valores de acordo com regras específicas.

Getters e Setters

Neste exemplo, a classe Pessoa possui atributos privados __nome e __idade, e os métodos get_nome e set_nome são usados para acessar e modificar o atributo __nome. O encapsulamento é mantido, e o código permite que você acesse e modifique esses atributos por meio dos métodos públicos get_nome e set_nome.

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

    def get_nome(self):
        return self.__nome

    def set_nome(self, novo_nome):
        if novo_nome:
            self.__nome = novo_nome

    def get_idade(self):
        return self.__nome

    def set_idade(self, nova_idade):
        if nova_idade > 0:
            self.__idade = nova_idade

# Criando uma instância da classe Pessoa
pessoa = Pessoa("Alice", 30)

# Usando um getter para acessar o atributo 'nome'
nome = pessoa.get_nome()
print("Nome", nome) # Saída: "Alice"

# Usando um setter para modificar o atributo 'nome'
pessoa.set_nome("Bob")

# Acessando o atributo 'nome' após a modificação
nome = pessoa.get_nome()
print("Nome", nome) # Saída: "Bob"
```

Encapsulamento



Exemplificando: Suponha uma classe que representa uma conta bancária. Nela, colocamos apenas os atributos nome e saldo. Também usaremos um método responsável por depositar um valor nessa conta bancária. O cálculo será feito da seguinte forma: o novo saldo será o somatório entre o valor atual mais o depósito acrescido de 10%.

Encapsulamento

Se os atributos puderem ser acessados diretamente em qualquer trecho do código, haverá o risco de o saldo ser alterado sem passar pelo método de depositar. Para evitar isso, podemos usar os métodos get e set para evitar o acesso direto.

Logo, para proteger as variáveis nome e, principalmente o saldo, utilizamos os métodos get saldo e set saldo. Antes disso, no entanto, é preciso alterar o nível de acesso das variáveis de pública para privada.

Property

O decorador @property em Python é usado para criar um método que permite acessar um atributo de uma classe como se fosse um atributo público, tornando-o de leitura apenas (não permite modificação direta). Isso é útil para implementar um getter, permitindo que você defina uma interface de acesso controlada aos atributos privados de uma classe.

Property

A classe Pessoa possui um atributo privado _nome e um método decorado com @property chamado nome. Esse método permite acessar o atributo _nome como se fosse um atributo público, usando a notação pessoa.nome, sem a necessidade de chamar uma função como um getter.

```python
class Pessoa:
    def __init__(self, nome):
        self.__nome = nome # Atributo privado com um sublinhado

    @property
    def nome(self):
        return self.__nome # Getter para acessar o atributo privado

# Criando uma instância da classe Pessoa
pessoa = Pessoa("Alice")

# Usando o getter para acessar o atributo nome
print(pessoa.nome) # Saída: "Alice"
```

Property

É importante observar que, com o decorador @property, o atributo _nome pode ser lido, mas não pode ser modificado diretamente. Para permitir a modificação controlada, você pode definir um método com o decorador @nome.setter.

```python
@nome.setter
    def nome(self, novo_nome):
        if novo_nome:
            self._nome = novo_nome
```

ATIVIDADE PRÁTICA 1

Desenvolva um aplicativo de gerenciamento de tarefas em python. Crie duas classes, Tarefa e Projeto, com uma associação unidirecional. Permita que as tarefas sejam associadas a projetos e que você possa listar as tarefas de um projeto em particular.

ATIVIDADE PRÁTICA 2

Desenvolva uma aplicação de loja online em. Crie as classes Cliente e Pedido com uma associação bidirecional. Os clientes podem fazer pedidos, e os pedidos devem estar associados aos clientes que os fizeram. Implemente a capacidade de listar todos os pedidos de um cliente específico.

ATIVIDADE PRÁTICA 3

Crie uma classe Aluno em Python com atributos privados, como nome, idade e matrícula. Implemente métodos públicos para acessar e modificar esses atributos. Em seguida, crie uma instância da classe e demonstre como usar os métodos de acesso.

ATIVIDADE PRÁTICA 4

Desenvolva uma classe Produto em python que contenha atributos privados, como nome, preço e quantidade em estoque. Forneça métodos públicos para acessar e modificar esses atributos e garantir que o preço e a quantidade não sejam definidos como valores negativos.

ATIVIDADE PRÁTICA 5

Desenvolva uma classe ContaBancaria em Python com atributos privados, como saldo e número da conta. Forneça métodos públicos para depositar dinheiro, sacar dinheiro e verificar o saldo. Garanta que o saldo não seja definido como negativo e que as transações sejam registradas.

DESAFIO PRÁTICO

sistema de biblioteca

Imagine um sistema de biblioteca em Python que gerencia livros e usuários. As classes envolvidas são Livro, Usuario, Biblioteca e Emprestimo.

A classe Livro deve ter atributos privados, como título e autor, e métodos públicos para obter esses atributos.

A classe Usuario deve ter atributos privados, como nome e ID, e métodos públicos para obter e modificar esses atributos.

DESAFIO PRÁTICO

A classe Biblioteca deve conter uma lista de livros disponíveis e métodos para adicionar e remover livros.

A classe Empréstimo deve representar um empréstimo de um livro por um usuário e deve estar associada a um Livro e a um Usuário.
O exercício é criar essas classes, estabelecer a associação entre elas (um usuário pode pegar emprestado um livro da biblioteca), aplicar encapsulamento para proteger os atributos privados e implementar métodos para:

DESAFIO PRÁTICO

Adicionar e remover livros da biblioteca.

Registrar um empréstimo de livro por um usuário, verificando se o livro está disponível.

Exibir informações sobre os empréstimo, como qual livro foi emprestado para qual usuário.

Material Complementar

- Exploração: Não tenha medo de explorar e testar diferentes códigos. A experimentação é uma grande aliada da aprendizagem.

- Perguntas: Faça perguntas, seja curioso! Entender o "porquê" das coisas ajuda a consolidar o conhecimento.

- Revisão: Revise o que aprendeu, tente explicar para si mesmo ou para outras pessoas. Ensinar é uma ótima forma de aprender.

- Prática: A prática leva à perfeição. Quanto mais exercícios fizer, mais fácil será lembrar e entender os conceitos.

INFINITY SCHOOL

VISUAL ART CREATIVE CENTER

AULA 13 - POO III - ASSOCIAÇÃO E ENCAPSULAMENTO