UNIVERSIDADE FEDERAL
UFERSA
RURAL DO SEMI-ÁRIDO

SAE BRASIL
AERODESIGN

PEGAZULS
AeroDesign

Nivelamento em POO

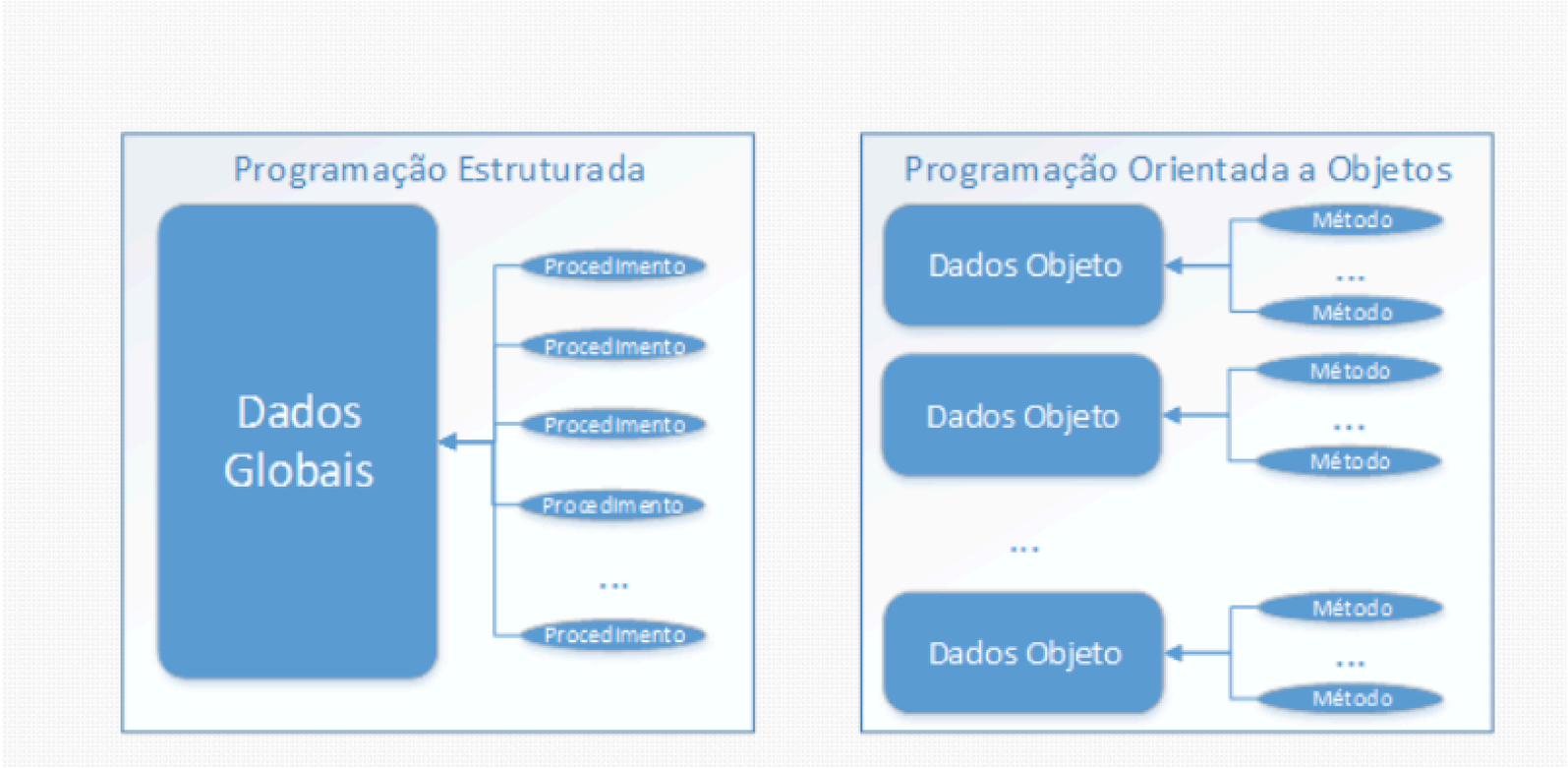
Sumário

Paradigmas de programação

- Um paradigma de programação é uma abordagem, estilo ou filosofia que define como um programador concebe e estrutura o código para resolver um problema. É um conjunto de regras e um modelo que determina a organização e a lógica do software, impactando a forma como os dados e a execução são tratados para criar um programa.

1. Paradigma Imperativo
2. Paradigma Orientado a Objetos
3. Paradigma Orientado a Eventos
4. *Paradigma Funcional*
5. Paradigma Declarativo
6. Paradigma Lógico

Programação estruturada x POO



Programação estruturada x POO

- **Problemas da programação estruturada**

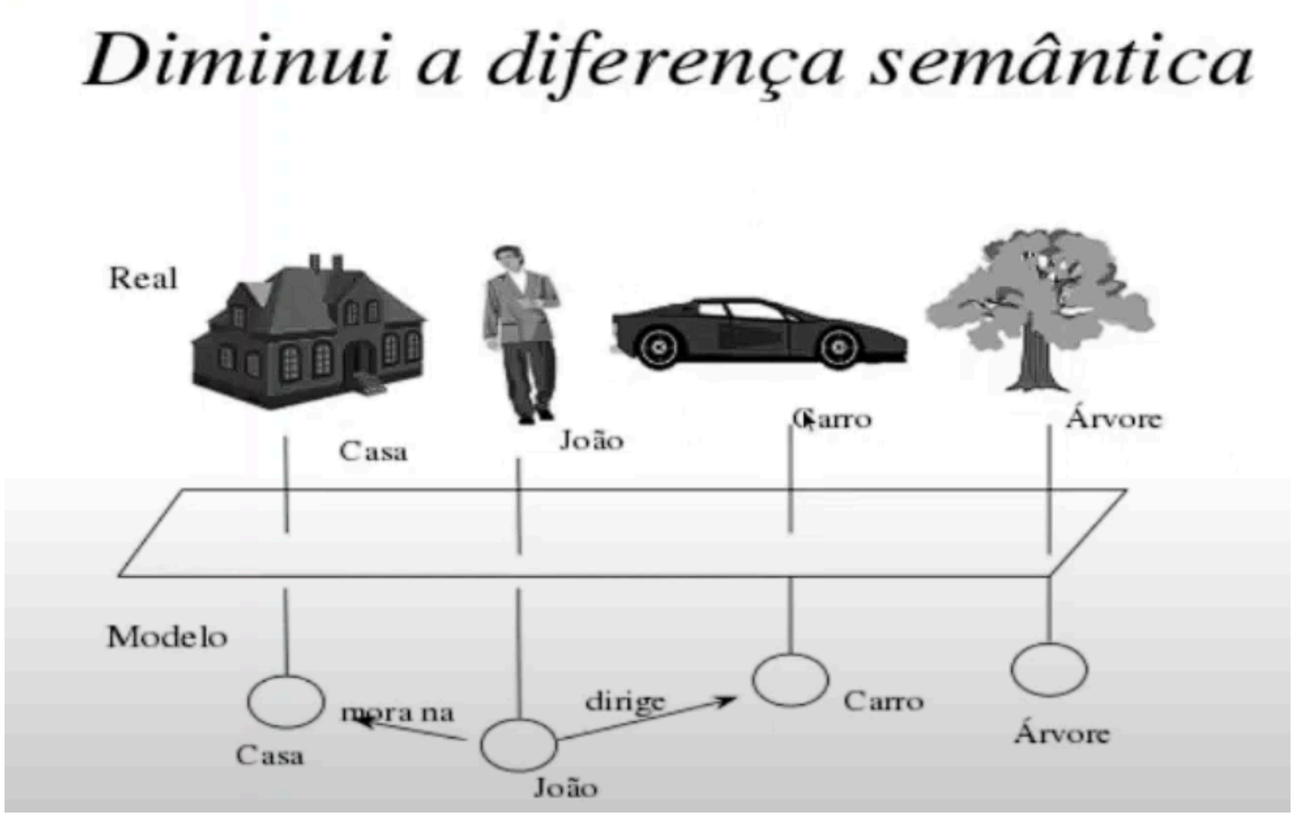
1. **Dados e Comportamentos Separados (Falta de Encapsulamento):** Não há uma conexão forte e obrigatória entre os dados e as operações que podem ser realizadas neles. Como demonstrado, qualquer um pode modificar.
2. **Falta de Clareza e Contrato Explícito:** Não há um "contrato" claro que defina o que uma conta é e o que ela pode fazer.
3. **Lógica de Negócio Espalhada e Duplicada:** Isso torna a manutenção do código um pesadelo.
4. **Dificuldade de Escalabilidade:** E se quisermos adicionar uma "Conta Poupança" com um método render_juros()?

Programação Orientada a Objetos

- A Programação Orientada a Objetos transcende a simples manipulação de linhas de código, introduzindo uma filosofia que se baseia na modelagem do mundo real.
- Em vez de tratar os programas apenas como instruções abstratas, a POO nos permite criar representações tangíveis de entidades do mundo real, conhecidas como “objetos”.
- POO nos permite construir sistemas de software que imitam a lógica e a estrutura das situações do mundo real.

Por que utilizar POO?

- Diminuição do GAP semântico



Por que utilizar POO?

- **Benefícios:**

- **Reutilização de Código:** A reutilização de classes e objetos permite economizar tempo e esforço, resultando em um desenvolvimento mais eficiente.

- **Organização Estruturada:** A POO promove a organização estruturada do código, tornando-o mais compreensível e gerenciável.

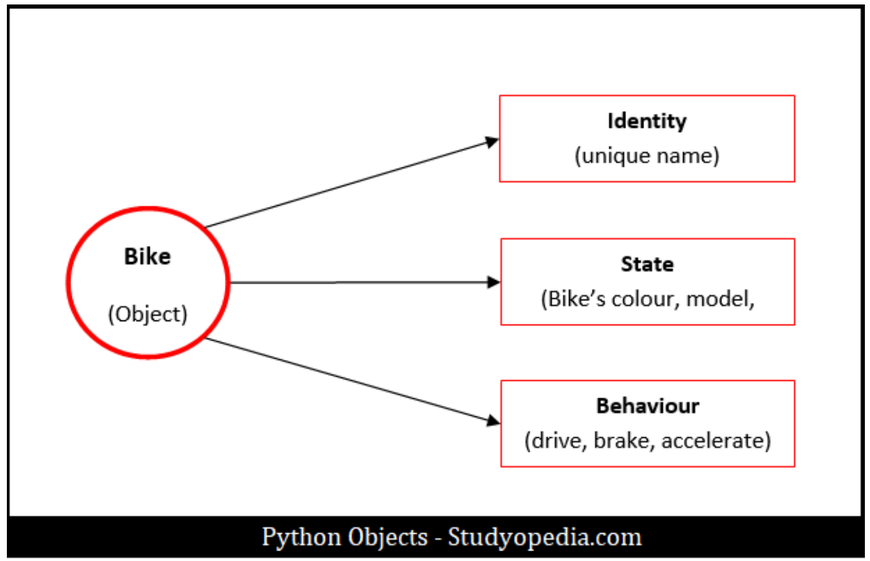
- **Modularidade**: A divisão do código em classes e módulos independentes facilita a manutenção e atualização do software.

- **Flexibilidade e Escalabilidade**: A hierarquia de classes e a capacidade de estender classes permitem a criação de sistemas flexíveis e escaláveis.

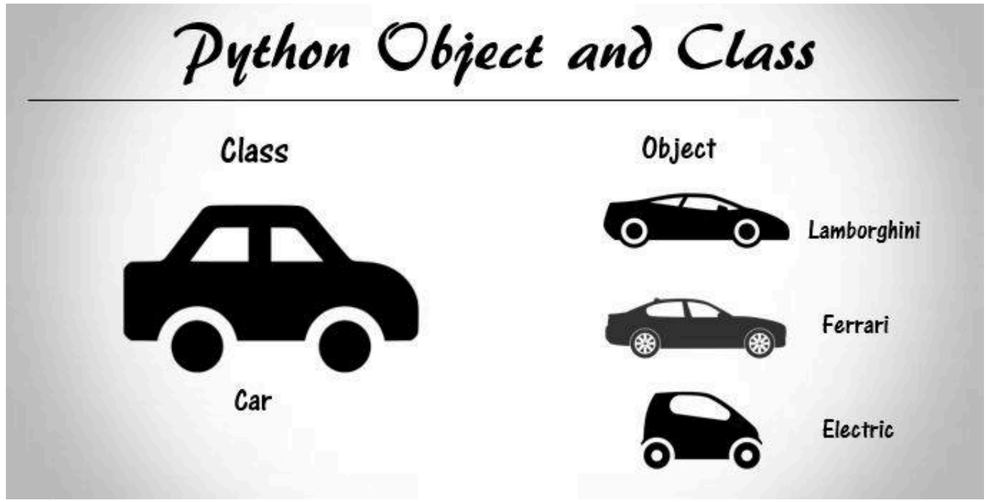
O que é um Objeto?

O que é um objeto:

- **Um objeto em POO possui as três principais características:**
- **Atributos:** São os dados que caracterizam o objeto (variáveis de escopo).
- **Métodos:** Um método é uma função que pertence a uma classe, sendo responsável por definir o comportamento de seus objetos. (Define aquilo que o objeto pode fazer)
- **Estado:** O estado de um objeto é o conjunto de todos os valores de seus atributos em um determinado momento.



Classe x Objeto



- **Classe:** É uma estrutura fundamental em POO que serve como um molde ou blueprint (um "projeto" ou "molde") para a criação de objetos. Ela define um conjunto de atributos e métodos que os objetos instanciados a partir dela terão.
- **Objeto**: É aquilo que é construído com base na base e tem identidade e estados próprios.

Implementação em python:

.1) Definição da classe

```python
class Aerodesign:
    asa: str = "Monoplano"
    fuselagem: str = "Caixão"
    MTOW: float = 9.18
    equipe: str = "Dragão Branco"
    voando: bool = False

    def decolar(self):
        """
        Muda o estado do avião para 'voando' se ele
        """
        if not self.voando:
            self.voando = True
            print("Decolando...")
        else:
            print("O avião já está voando.")

    def pousar(self):
        """
        Muda o estado do avião para 'em terra' se el
```

2) Instanciação do objeto

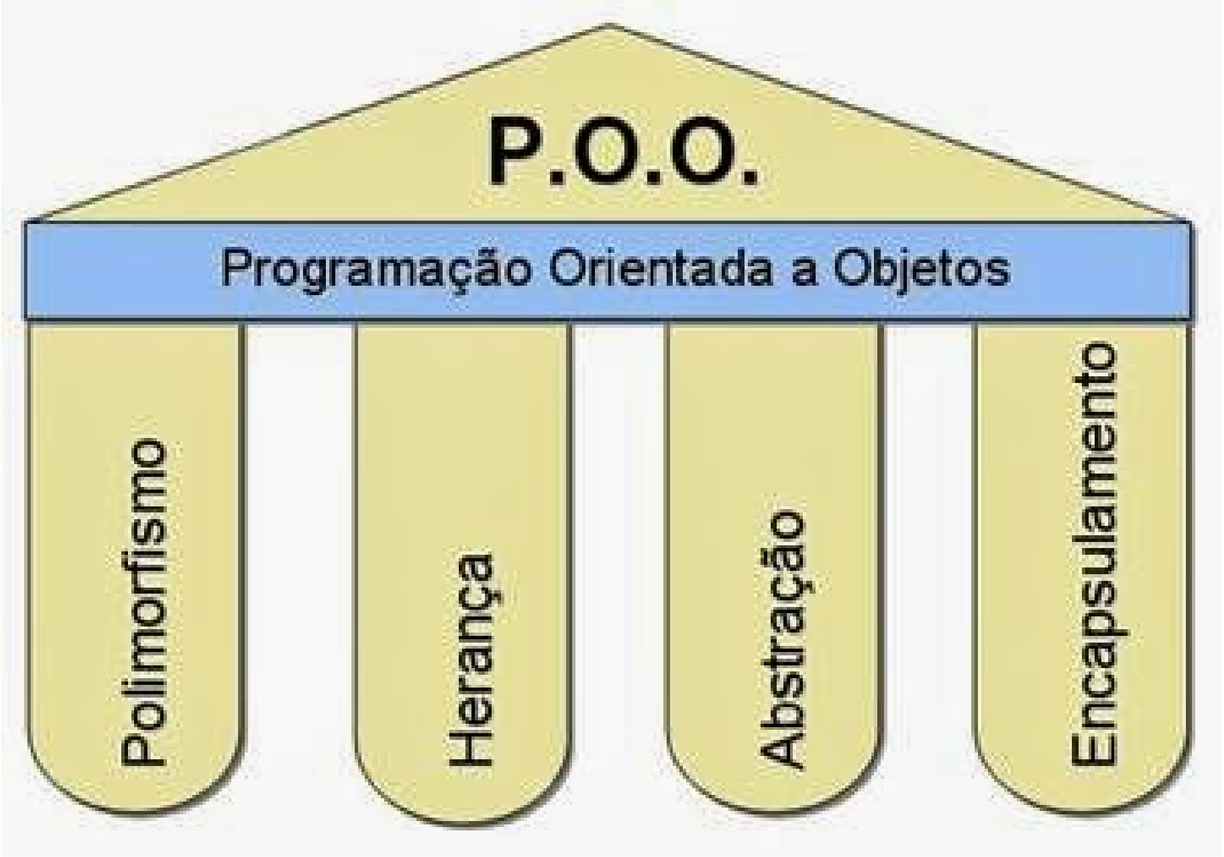
```python
if __name__ == "__main__":
    # Criando uma instância (objeto) da cla
    meu_aviao = Aerodesign()

    # Exibindo a ficha técnica do avião
    meu_aviao.ficha_tecnica()

    # Tentando decolar
    meu_aviao.decolar()

    # Exibindo a ficha técnica novamente pa
    meu_aviao.ficha_tecnica()

    # Tentando pousar
    meu_aviao.pousar()

    # Exibindo a ficha técnica novamente pa
    meu_aviao.ficha_tecnica()
```

Método Construtor

```python
class Aerodesign:

    def __init__(self, asa: str, fuselagem: str, MTOW: float, equipe: str):
        self.asa = asa
        self.fuselagem = fuselagem
        self.MTOW = MTOW
        self.equipe = equipe
        self.voando = False
```

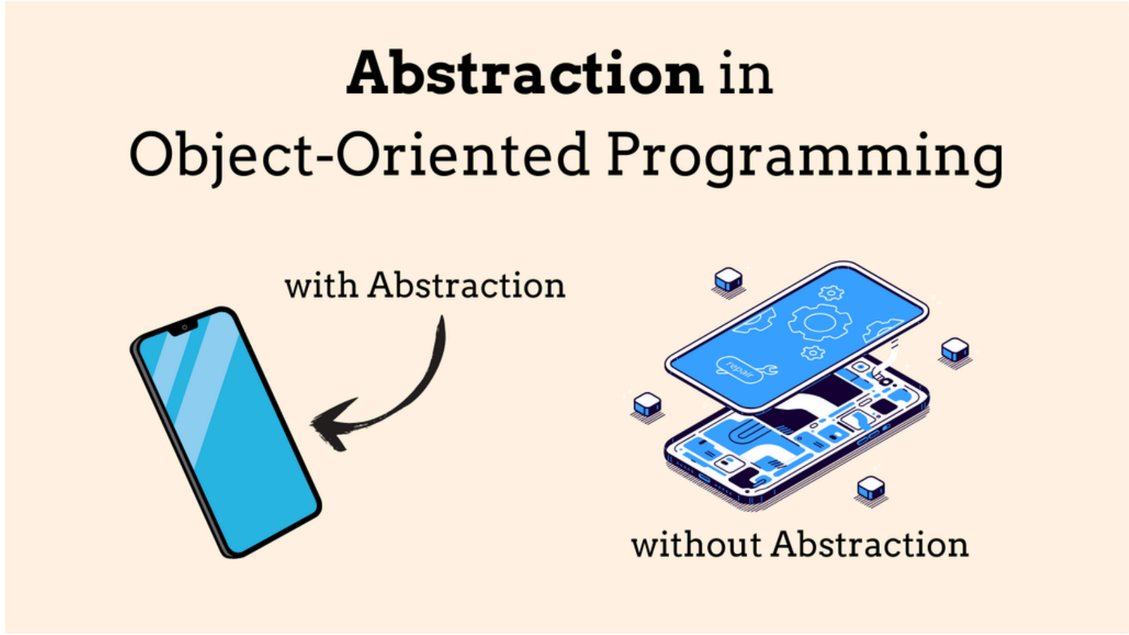
```python
if __name__ == "__main__":
    # Criando uma instância (objeto) da classe Aerodesign
    meu_aviao = Aerodesign(
        asa="Asa alta",
        fuselagem="Monocoque",
        MTOW=25.0,
        equipe="Equipe de Aerodesign XYZ"
    )
```

Pilares da Programação Orientada a Objetos



Abstração: Simplificando a Complexidade

- A abstração envolve a criação de modelos simplificados que capturam os aspectos essenciais de um objeto do mundo real.
- Em vez de se preocupar com todos os detalhes, concentramo-nos apenas nas informações relevantes para o contexto do programa.
- A abstração ajuda a reduzir a complexidade, permitindo que os desenvolvedores se concentrem no que é importante sem serem sobrecarregados por detalhes triviais.

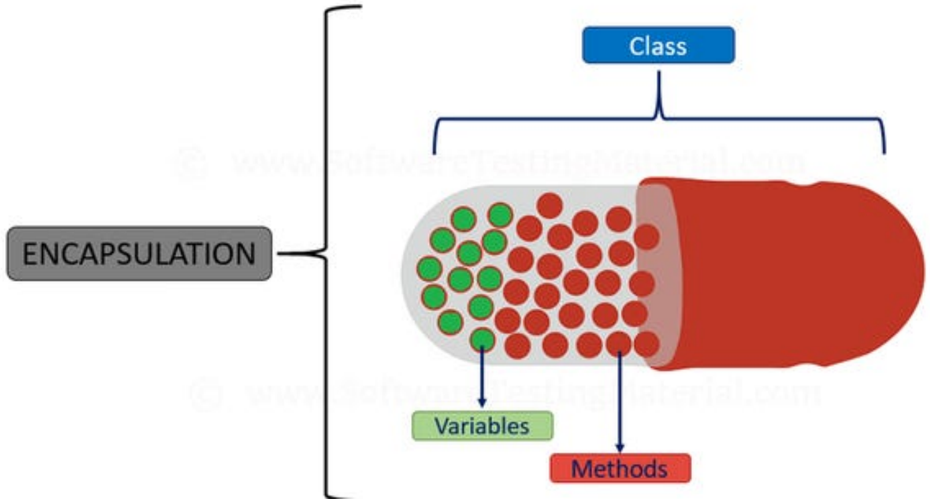


Problemática da abstração:

“"Voce foi contratado para desenvolver o sistema de uma empresa de locação de veiculos, um veiculo é uma máquina bastante complexa cheia de nuancia mecanicas e tudo mais, contudo, o seu sistema deve mostrar somente as informações necessárias para o cliente na hora da locação, como deve ser implementada essa classe?"”

Encapsulamento

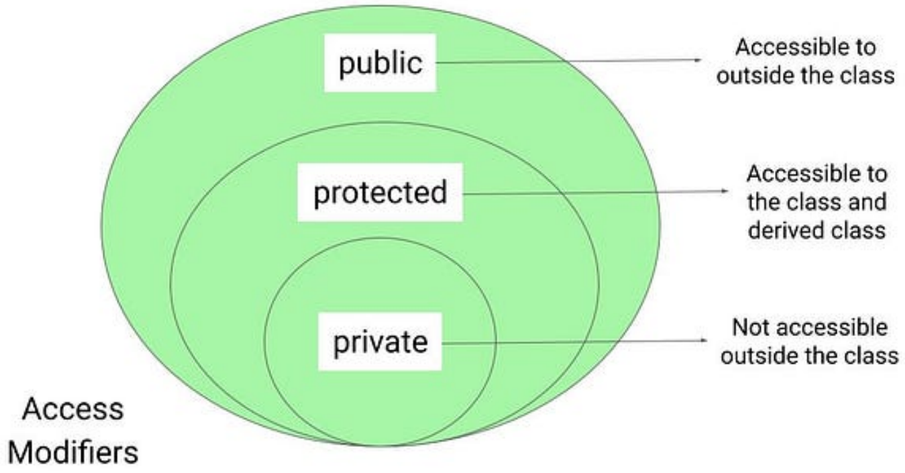
- o encapsulamento oculta a complexidade interna e protege os dados de modificações externas indevidas. Ele cria uma "barreira de proteção" ao redor do estado do objeto.



Encapsulamento

- **Controle de visibilidade em Python: A Filosofia do "Adulto Consentido"**

- Python não impede tecnicamente o acesso direto aos atributos. Em vez disso, ele usa convenções de nomenclatura para sinalizar a intenção do programador:
- _(um_underline): Protegido
- __(dois_underline): Privado



Problemática do encapsulamento

"Você foi contratado para desenvolver a classe ContaBancaria para um novo banco digital. O atributo mais crítico desta classe é, sem dúvida, o saldo. Por padrão, qualquer parte do seu programa pode acessar e modificar este valor diretamente. Contudo, o sistema deve ser seguro, garantindo que o saldo NUNCA fique negativo e que saques só ocorram se houver fundos. Uma modificação externa e sem validação poderia criar um rombo financeiro catastrófico.
Como você deve implementar essa classe para proteger o saldo, forçando que toda e qualquer alteração passe pelas regras de negócio?
"

Herança

- É o pilar focado em reutilizar e especializar. A herança permite que uma nova classe (chamada de subclasse ou classe filha) seja baseada em uma classe existente (a superclasse ou classe mãe).
- A classe filha herda todos os atributos e métodos da classe mãe. Isso cria uma relação "É UM" (IS-A).

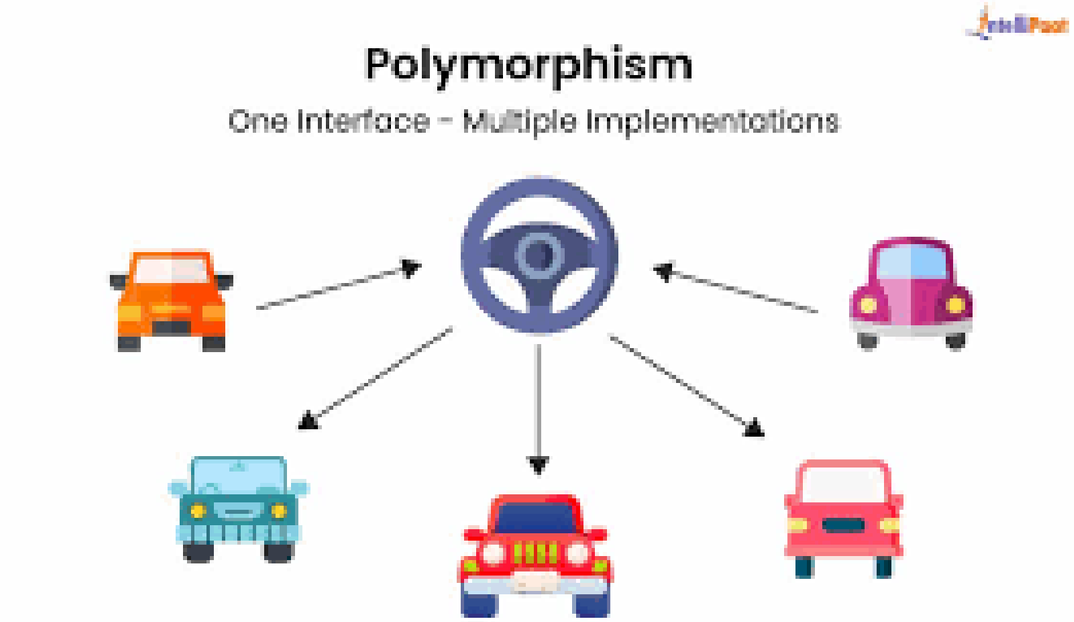
```python
class Animal:
    def comer(self):
        # Código para comer

class Cachorro(Animal):
    def latir(self):
        # Código para latir
```

Problemática da herança

"O banco digital para o qual você trabalha está crescendo e precisa oferecer novos produtos. Além da ContaBancaria comum que você já criou, eles agora querem lançar uma ContaPoupanca, que terá um método para render juros, e uma ContaCorrente, que terá um limite de cheque especial.
Você percebe que todas essas contas são muito parecidas: todas têm titular, saldo, e todas precisam de métodos para sacar e depositar. Criar cada classe do zero resultaria em uma enorme quantidade de código repetido, tornando a manutenção um pesadelo."

Polimorfismo

- *É o pilar focado em flexibilidade e múltiplas formas. A palavra Polimorfismo vem do grego e significa "muitas formas". Em POO, é a capacidade de objetos de diferentes classes responderem à mesma mensagem (chamada de método), cada um de sua maneira específica.*
- *Ele permite que você trate objetos de classes filhas como se fossem objetos da classe mãe, simplificando o código e tornando-o muito mais flexível e extensível.*



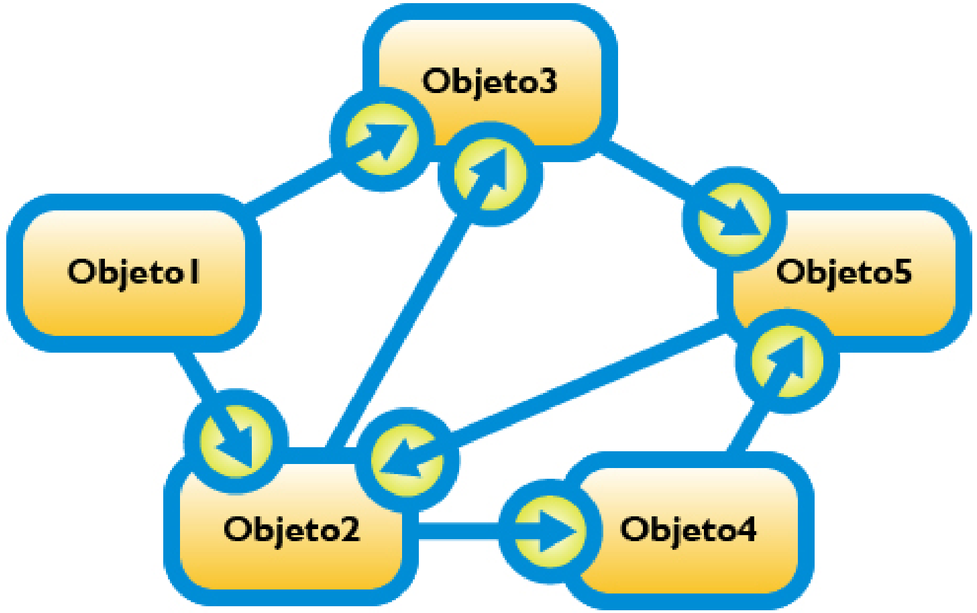
Problemática do Polimorfismo

“O sistema do seu banco está funcionando bem com os diferentes tipos de contas (ContaPoupanca, ContaCorrente). Agora, surge um novo requisito: criar uma função para a "virada de mês".

Essa função deve passar por TODAS as contas do banco, independentemente do tipo, e executar a ação de manutenção específica de cada uma: a ContaPoupanca deve render juros, enquanto a ContaCorrente deve cobrar uma taxa de manutenção.”

Relacionamento entre classes

- *Até agora, focamos em como construir uma classe robusta. Mas em sistemas reais, nenhuma classe trabalha sozinha. Elas precisam colaborar. Os relacionamentos entre classes são os diferentes tipos de "contratos" e "conexões" que elas podem ter, indo do mais forte (Herança) ao mais flexível (Associação).*

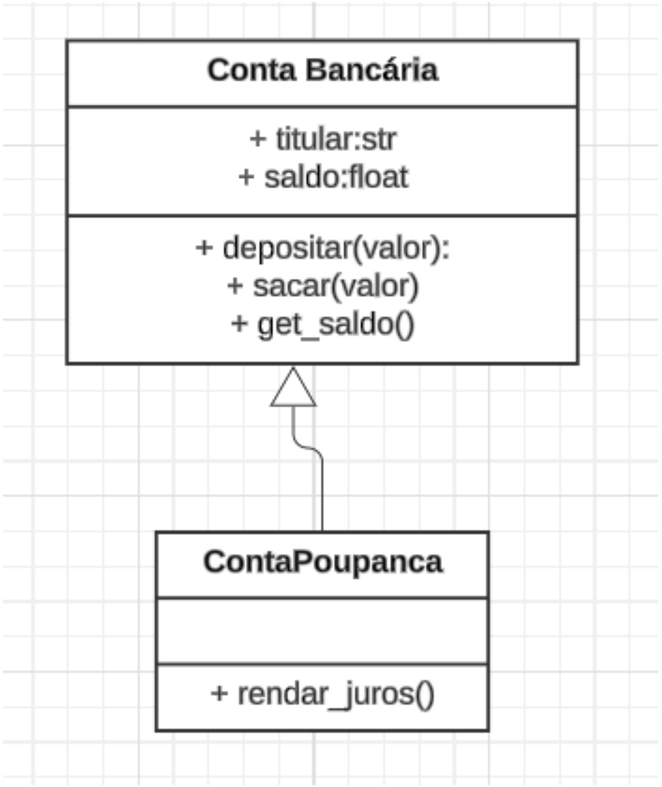


Relacionamento entre classes

- ***Herança (É um)***

- Uma classe (filha) É UMA especialização de outra classe (mãe). Ela herda todos os atributos e métodos, criando um acoplamento muito forte.

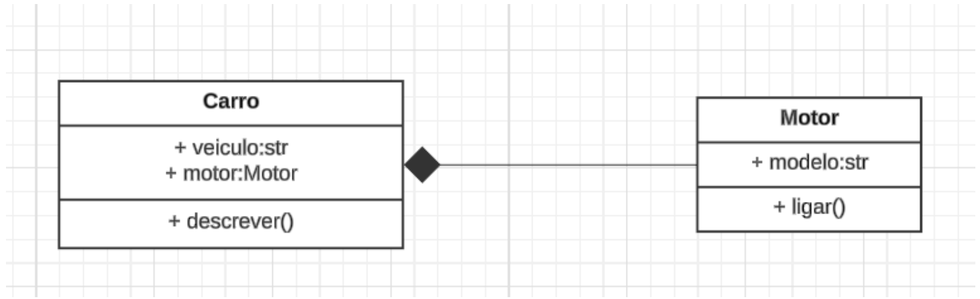
- Já vimos: uma ContaPoupanca É UMA ContaBancaria. Não faz sentido existir uma ContaPoupanca que não seja, fundamentalmente, uma ContaBancaria.



Relacionamento entre classes

- ***Composição (Relação "Tem-Um" / "Parte-De")***

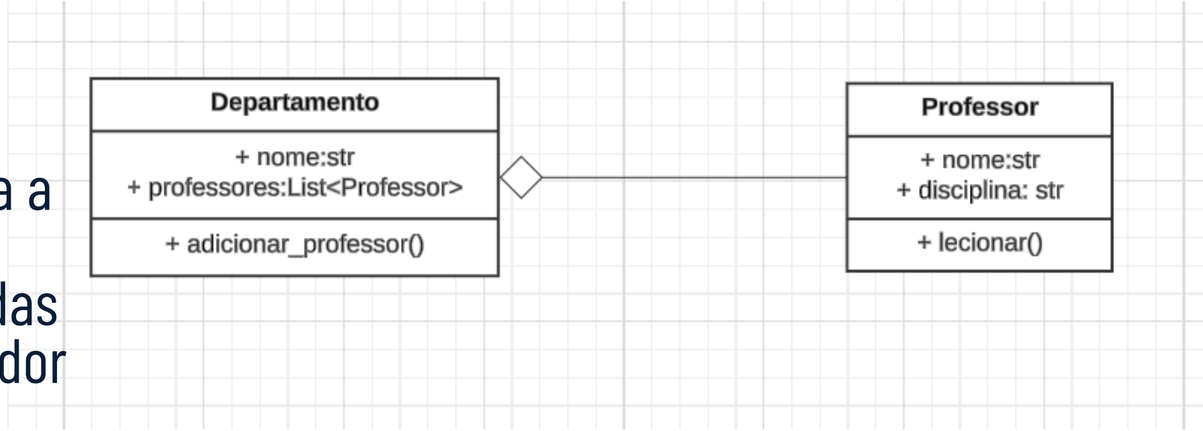
- Um objeto (o "todo") é composto por um ou mais objetos de outras classes (as "partes"). A característica principal é que as partes não existem sem o todo. O ciclo de vida delas está totalmente atrelado. Se o objeto "todo" é destruído, suas "partes" também são.



Relacionamento entre classes

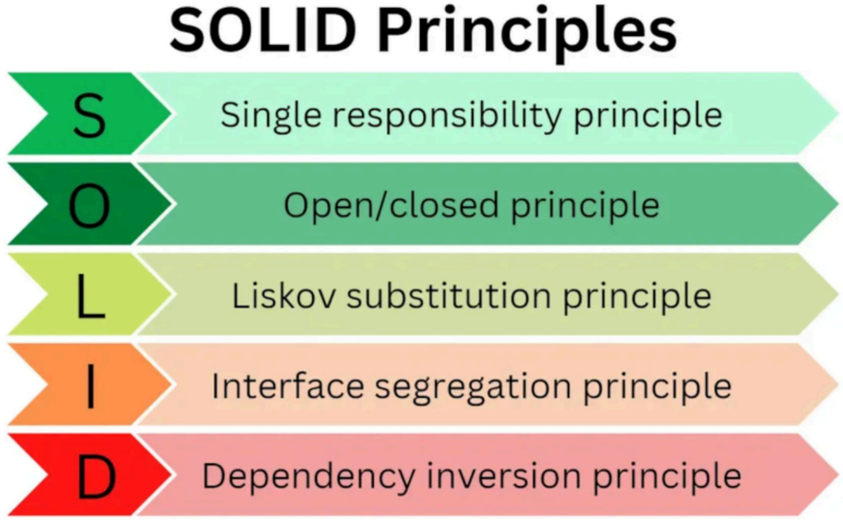
- ***Agregação (Relação "Tem-Um", mas independente)***

- A Agregação é uma forma mais flexível de relacionamento "Tem-Um".
- Um objeto (o "agregador") possui uma referência a um ou mais objetos independentes. A principal diferença da Composição é que o ciclo de vida das partes não depende do todo. Se o objeto agregador for destruído, as partes continuam a existir.



Boas práticas em POO

- O SOLID é uma junção de princípios e boas práticas que visam melhorar a arquitetura e design de um projeto, além de ter como intuito facilitar sua manutenção e compreensão.



SOLID

- **Single Responsibility Principle (Princípio da responsabilidade única)**

  - O princípio da responsabilidade única enfatiza que uma classe deve ter apenas um objetivo, ou seja, ela deve possuir apenas uma função ou funções similares.

SOLID

- **Single Responsibility Principle (Princípio da responsabilidade única)**

> “Você tem uma classe OrdemDeServico que, além de gerenciar os dados da ordem (itens, preço), também é responsável por salvar essa ordem em um banco de dados e enviar um e-mail de notificação. Se no futuro a forma de notificação mudar (de e-mail para SMS) ou o banco de dados for trocado (de MySQL para PostgreSQL), a classe OrdemDeServico precisará ser alterada. Ela tem três motivos para mudar”

SOLID

- **Single Responsibility Principle (Princípio da responsabilidade única)**

Sem SRP:

```python
# --- MODO ERRADO (Múltiplas Responsabilidades) ---
class OrdemDeServico_ERRADO:
    def __init__(self, id, cliente, total):
        self.id = id
        self.cliente = cliente
        self.total = total

    def salvar_no_banco(self):
        print(f"Salvando Ordem {self.id} no banco de dados...")

    def enviar_email_confirmacao(self):
        print(f"Enviando e-mail para {self.cliente}...")
```

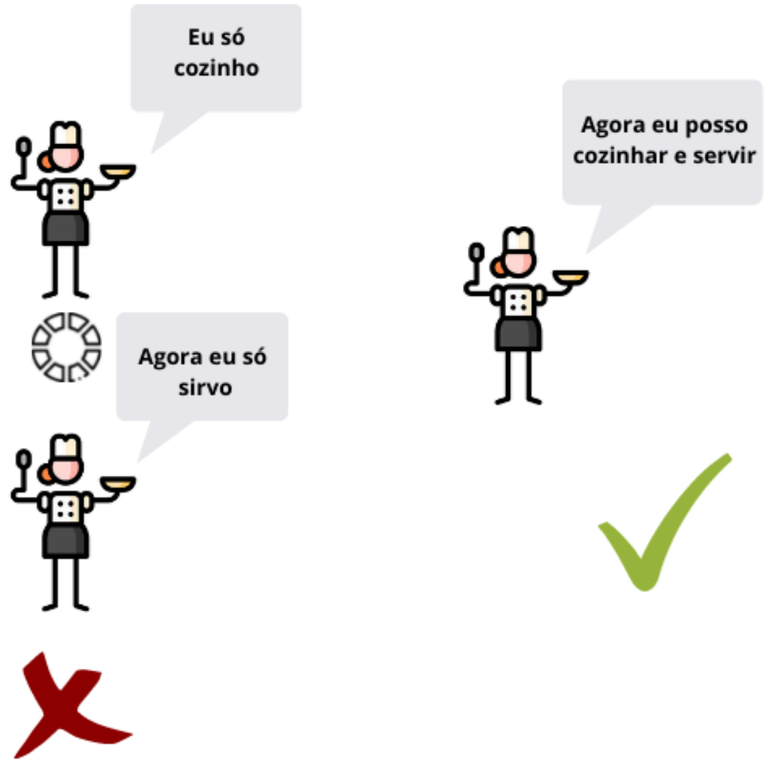
Com SRP:

```python
# --- MODO CORRETO (Responsabilidades Separadas) ---
class OrdemDeServico:
    """Responsabilidade: Apenas gerenciar os dados da ordem."""
    def __init__(self, id, cliente, total):
        self.id = id
        self.cliente = cliente
        self.total = total

class RepositorioDeOrdens:
    """Responsabilidade: Apenas persistir os dados."""
    def salvar(self, ordem):
        print(f"Salvando Ordem {ordem.id} no banco de dados...")

class ServicoDeNotificacao:
    """Responsabilidade: Apenas notificar o cliente."""
    def enviar_email_confirmacao(self, ordem):
        print(f"Enviando e-mail para {ordem.cliente}...")
```

SOLID

- **Open/Closed Principle (Princípio Aberto/Fechado)**

  - As entidades de software (classes, módulos, funções) devem ser abertas para extensão, mas fechadas para modificação. Ou seja, você deve ser capaz de adicionar novas funcionalidades sem alterar o código-fonte existente.



SOLID

- **Open/Closed Principle (Princípio Aberto/Fechado)**

"Você tem uma função gerar_relatorio que recebe uma lista de pagamentos e exporta os dados. Inicialmente, ela só exporta para PDF. Agora, o cliente pede para exportar também para CSV. A solução ingênua seria adicionar um if/else na função. E se depois pedirem para exportar para JSON? A função se tornaria uma cadeia de if/elif/else que precisa ser modificada a cada novo formato."

SOLID

- **Open/Closed Principle (Princípio Aberto/Fechado)**

Sem OCP:

```python
# --- MODO ERRADO (Modificando a função existente) ---
def gerar_relatorio_ERRADO(pagamentos, formato):
    if formato == "pdf":
        print("Exportando pagamentos para PDF...")
    elif formato == "csv": # <-- Tivemos que MODIFICAR a função
        print("Exportando pagamentos para CSV...")
```

Com OCP:

```python
# --- MODO CORRETO (Estendendo com novas classes) ---
# 1. Criamos uma abstração (interface)
class Exportador:
    def exportar(self, pagamentos):
        pass

# 2. Criamos implementações concretas (EXTENSÕES)
class ExportadorPDF(Exportador):
    def exportar(self, pagamentos):
        print("Exportando pagamentos para PDF...")

class ExportadorCSV(Exportador):
    def exportar(self, pagamentos):
        print("Exportando pagamentos para CSV...")

class ExportadorJSON(Exportador): # <-- Nova extensão, sem tocar no código antigo!
    def exportar(self, pagamentos):
        print("Exportando pagamentos para JSON...")

# 3. A função principal agora está FECHADA para modificação.
def gerar_relatorio(pagamentos, exportador: Exportador):
    # Ela não se importa com o formato, apenas chama o método 'exportar'.
    exportador.exportar(pagamentos)

# Uso:
pagamentos_de_hoje = [100, 250, 80]
gerar_relatorio(pagamentos_de_hoje, ExportadorPDF())
gerar_relatorio(pagamentos_de_hoje, ExportadorCSV())
gerar_relatorio(pagamentos_de_hoje, ExportadorJSON())
```

SOLID

- **Open/Closed Principle (Princípio Aberto/Fechado)**

  “Um sistema precisa enviar notificações para os usuários. Inicialmente, ele só envia por "email". Agora, é preciso adicionar "SMS" e "Push Notification" como canais de notificação. A classe Notificador tem a lógica de envio de email diretamente em seu método.

  Adicionar novos canais implicaria em reescrever o método enviar com mais condicionais, misturando lógicas que não têm relação entre si.

  Como criar um sistema onde novos canais de notificação possam ser integrados sem alterar a classe Notificador principal?
  ”

SOLID

Sem OCP:

```python
# --- MODO ERRADO ---
class Notificador_ERRADO:
    def enviar(self, mensagem, canal):
        if canal == "email":
            print(f"Enviando via Email: {mensagem}")
        # MODIFICAR aqui para "sms".
```

Com OCP:

```python
# 1. A abstração
class CanalDeNotificacao:
    def enviar(self, mensagem):
        pass

# 2. As extensões
class CanalEmail(CanalDeNotificacao):
    def enviar(self, mensagem):
        print(f"Enviando via Email: {mensagem}")

class CanalSMS(CanalDeNotificacao):
    def enviar(self, mensagem):
        print(f"Enviando via SMS: {mensagem}")

class CanalPush(CanalDeNotificacao): # <-- Nova extensão
    def enviar(self, mensagem):
        print(f"Enviando via Push Notification: {mensagem}")

# 3. O código cliente
class Notificador:
    def __init__(self, canais: list[CanalDeNotificacao]):
        self.canais = canais

    def notificar_todos(self, mensagem):
        for canal in self.canais:
            canal.enviar(mensagem)

# Uso:
canais_do_usuario = [CanalEmail(), CanalPush()]
notificador_app = Notificador(canais_do_usuario)
notificador_app.notificar_todos("Sua fatura fechou!")
```

SOLID

- **Open/Closed Principle (Princípio Aberto/Fechado)**

- **Identifique o Comportamento que Varia:** Olhe para o seu código e encontre a parte que precisa ser constantemente modificada. Esse é o "comportamento extensível".

- **Crie uma Abstração (A "Interface" ou o "Contrato"):** Defina uma "interface" (Uma classe com um método sem implementação) que represente esse comportamento de forma genérica. Este é o seu "contrato" ou a sua "tomada de parede".

- **Crie Implementações Concretas (As "Extensões"):** Para cada bloco if ou elif do código original, crie uma nova classe concreta ("um plug") que herde da sua abstração (passo 2) e implemente o método de forma específica.

- **Inverta a Dependência no Código Cliente:** Volte à sua função ou classe original (o "código cliente") e remova completamente o bloco if/elif/else. Modifique-a para que ela dependa da abstração (a "tomada"), e não mais das implementações concretas (os "plugs").

SOLID

- **Demais Principios:**

- **Liskov Substitution Principle (Princípio da Substituição de Liskov):** As classes derivadas (filhas) devem ser substituíveis por suas classes base (mãe) sem quebrar o funcionamento do programa. Se um código funciona com um objeto do tipo Pai, ele deve funcionar com um objeto do tipo Filho da mesma forma

- **Interface Segregation Principle (Princípio da Segregação de Interfaces):** É melhor ter várias interfaces pequenas e específicas do que uma única interface grande e genérica.

- **Dependency Inversion Principle (Princípio da Inversão de Dependência):** Módulos de alto nível não devem depender de módulos de baixo nível. Ambos devem depender de abstrações (interfaces).

APOIADORES DO NOSSO PROJETO

Ansys

BRASSNOX
REFERÊNCIA EM AÇO INOX

BRINDES MAXWELL®

JPATRÍCIO METAIS

KI SOMBRA
SOLDA, COBERTURA, PORTÕES EM GERAL

MATLAB®

TEK BOND
SAINT-GOBAIN

T-MOTOR
THE SAFEST PROPULSION SYSTEM

SOLIDWORKS

WOODPECKER
BALSA WOOD

PROEC
PRÓ-REITORIA DE EXTENSÃO E CULTURA

PROAE
PRÓ-REITORIA DE ASSUNTOS ESTUDANTIS

CE
CENTRO DE ENGENHARIAS

CCEN
CENTRO DE CIÊNCIAS EXATAS E NATURAIS