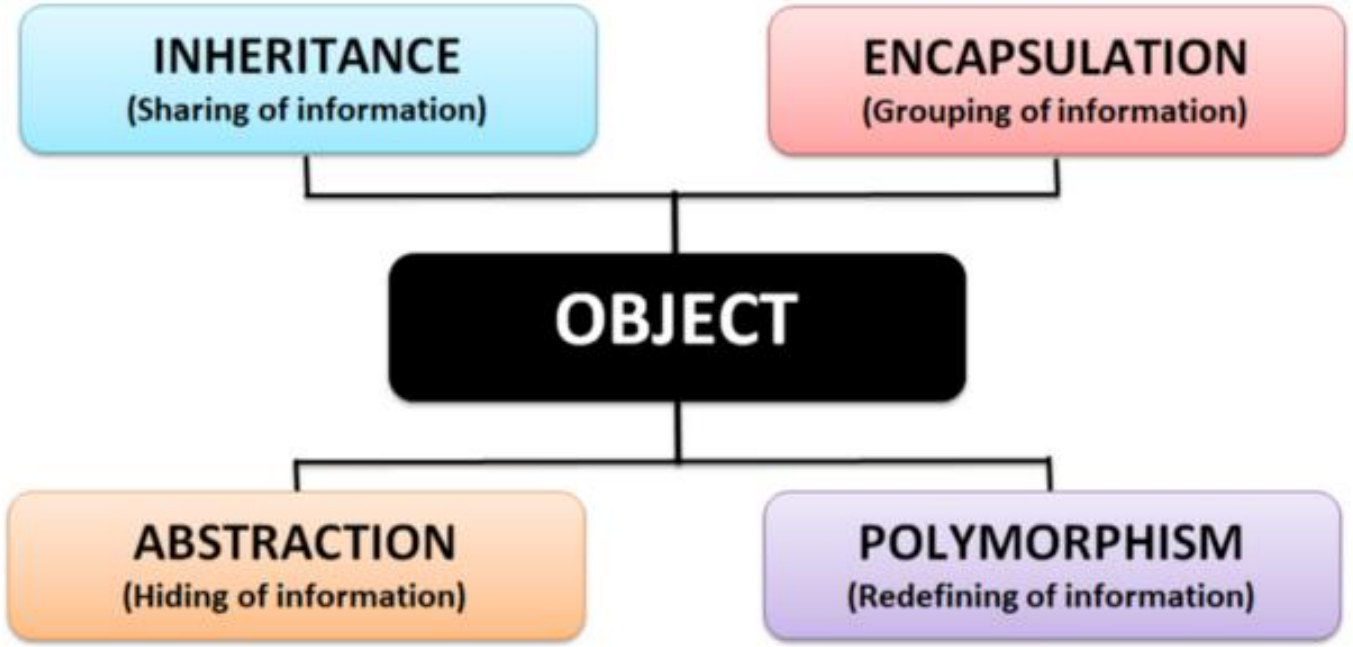
Ciência da Computação

Programação Orientada a Objetos

Prof. Luciano Rodrigo Ferretto

Pilares da Orientação a objetos



Ciência da Computação

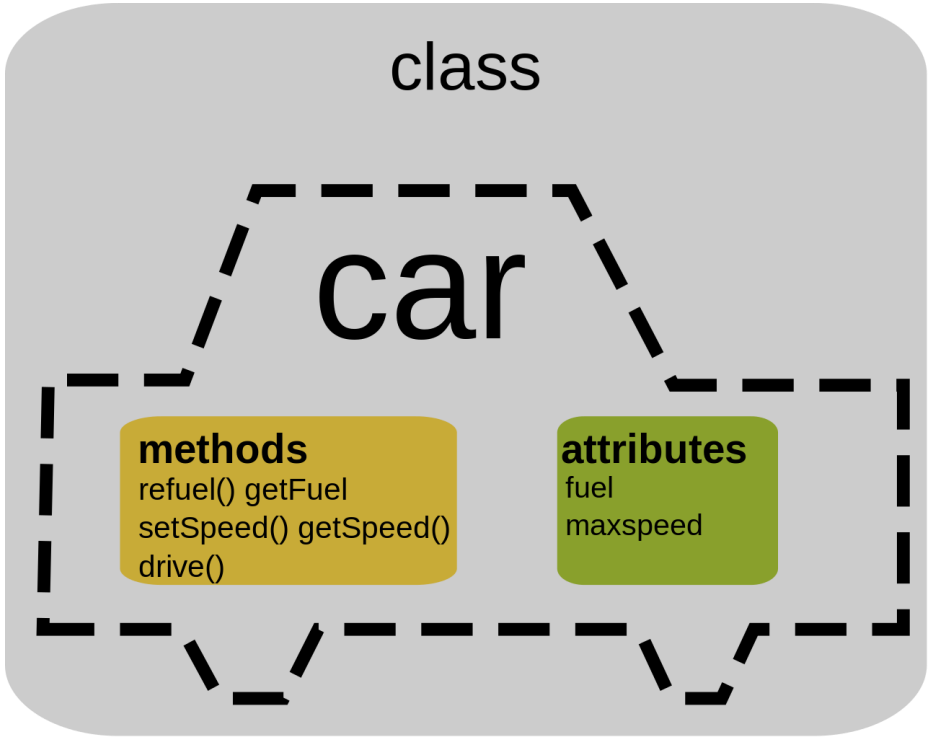
Abstração

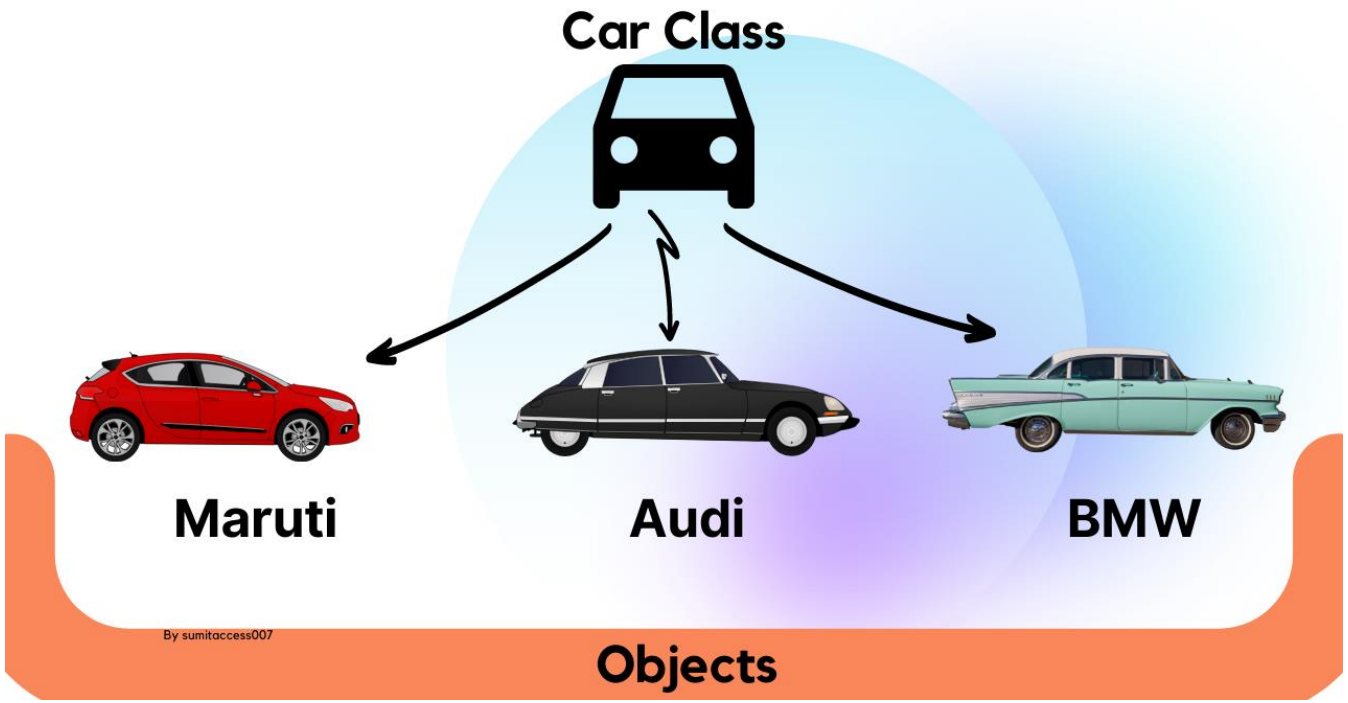
- **Abstração** em Programação Orientada a Objetos é o processo de identificar e definir características essenciais de objetos do mundo real, “traduzindo” (representando) essas características em forma de **classes** e **objetos**.

- Uma **classe** é uma estrutura que define atributos e métodos que representam um tipo de objeto, fornecendo um modelo para criar instâncias desse tipo.
- Um **objeto** é uma instância de uma classe, caracterizado por seus atributos e comportamentos definidos pela classe, e pode interagir com outros objetos por meio de métodos e troca de dados.

Objeto é uma **instância** concreta de uma classe na POO.

Abstração





Ciência da Computação

Encapsulamento

- **Encapsulamento** se refere à prática de ocultar os detalhes internos de um objeto e expor apenas as operações relevantes para manipular esses detalhes, promovendo assim a modularidade, a segurança e a manutenibilidade do código.

- Na maioria das linguagens de programação orientada a objetos, o encapsulamento é alcançado através da definição de **modificadores de acesso.**
  - **public, private**, **protected e default** (sem modificador)
- Em Python, os modificadores de acesso são implementados como **convenções de nomenclatura**, não como palavras-chave.
  - **public**: Atributos e métodos começam com letras minúsculas.
  - **private**: Atributos e métodos começam com dois *underlines* _ _
  - **protected**: Atributos e métodos começam com um *underlines* _

Esta são apenas convenções de nomenclatura, ou seja, não possuem efeito prático

Ciência da Computação

Herança

- **Herança** é o princípio que permite que uma classe (subclasse) herde os atributos e métodos de outra classe (superclasse), facilitando a reutilização de código, a organização hierárquica e a promoção de relações entre objetos.

- Quando você possui duas ou mais classes com atributos e métodos em comum, a herança facilita a criação de uma estrutura hierárquica, onde uma classe "pai" define os elementos comuns que serão herdados pelas classes "filhas".

Herança

Na programação orientada a objetos, a relação de herança entre classes é a relação em que uma classe é do tipo **é uma**, e não do tipo **tem uma**. Esta é uma das confusões recorrentes na construção de programas em orientação a objetos.

Para ilustrar essa diferença, observe a Figura 1 e veja que, neste exemplo, temos um tipo de relação **é uma**, pois o objeto da classe `Cachorro`, assim como o objeto da classe `Gato`, é, por herança, um tipo de objeto da classe `Animal`.

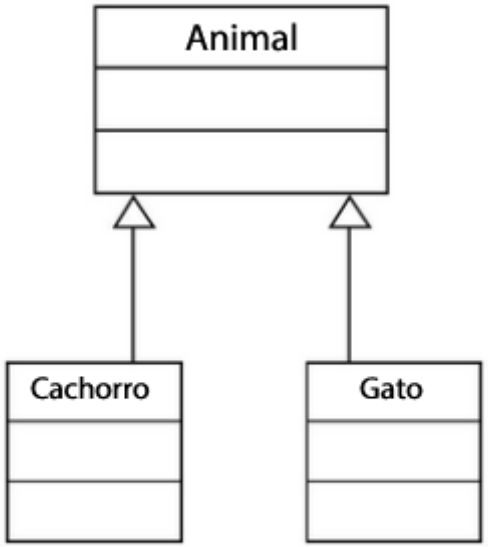


Figura 1. Representação de herança entre classes.

Compare, agora, a relação apresentada na Figura 2, que representa uma relação do tipo **tem uma**. Perceba que, neste caso, uma relação de herança entre as classes `Estado` e `Cidade` não faz sentido, visto que um estado possui cidades, mas uma cidade não é um estado.

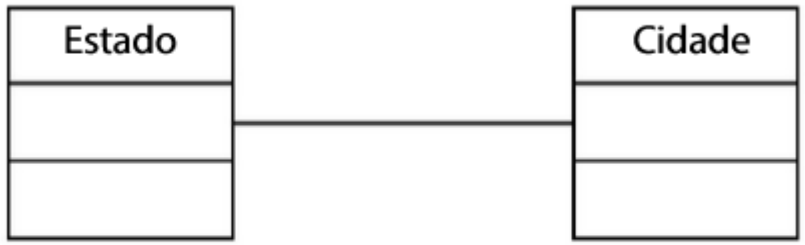


Figura 2. Representação da relação do tipo **tem uma** entre classes.

Fonte: SILVA, F. M.; LEITE, M. C. D.; OLIVEIRA, D. B. **Paradigmas de programação**. Porto Alegre: Sagah, 2019.

Computação

Herança e Sobrescrita de Método

- **Herança** é o princípio que permite que uma classe (subclasse) herde os atributos e métodos ...

- Mas as vezes o filho não faz “igualzito” o pai!!!

- Então, uma subclasse, além de ter seus atributos próprios, também pode ter implementações de métodos diferente da implementação da superclasse, ou seja, sobrescrever o método.

Ciência da Computação

Herança e Sobrescrita de Método

```
class Veiculo:
    # __init_ => é o método construtor
    def __init__(self, marca, modelo, placa, ano):
        self.__marca = marca
        self.__modelo = modelo
        self.__placa = placa
        self.__ano = ano

    def __str__(self):
        return f'''Marca: {self.__marca}
- Modelo: {self.__modelo}
- Placa: {self.__placa}
- Ano: {self.__ano}'''
```

- No nosso exemplo, temos a classe “Moto” que herda os atributos e métodos da superclasse “Veiculo”.
- O método “__str__()” está sendo herdado, ou seja, um objeto do tipo “Moto” terá esse método
- Mas como faço para o “toString” retornar também o atributo “cilindradas”

```
class Moto(Veiculo):
    def __init__(self, marca, modelo, placa, ano, cilindradas):
        super().__init__(marca, modelo, placa,ano)
        self.__cilindradas = cilindradas
```

Herança e Sobrescrita de Método

```
class Moto(Veiculo):
    def __init__(self, marca, modelo, placa, ano, cilindradas):
        super().__init__(marca, modelo, placa,ano)
        self.__cilindradas = cilindradas

    def __str__(self):
        return f'''Marca: {self.get_marca()}
- Modelo: {self.get_modelo()}
- Placa: {self.get_placa()}
- Ano: {self.get_ano()}
- Cilindradas: {self.__cilindradas}'''
```

- Aqui podemos sobrescrever o método __str__()
- A Sobrescrita de métodos é um dos meios pelos quais implementamos o Polimorfismo na Orientação a Objetos.

Ciência da Computação

Polimorfismo

- **Polimorfismo** é o princípio a partir do qual as classes derivadas de uma única classe base são capazes de invocar os métodos que, embora apresentem a mesma assinatura, comportam-se de maneira diferente para cada uma das classes derivadas.

- Com o Polimorfismo, os mesmos atributos e objetos podem ser utilizados em objetos distintos, porém, com implementações lógicas diferentes.

Ciência da Computação

Polimorfismo - Exemplo

- Podemos dizer que uma classe chamada **Vendedor** e outra chamada **Diretor** podem ter como base uma classe chamada **Pessoa**, com um método chamado **CalcularVendas()**.

- Se este método (definido na classe base) se comportar de maneira **diferente** para as chamadas feitas a partir de uma instância de *Vendedor* e para as chamadas feitas a partir de uma instância de *Diretor*, ele será considerado um método polimórfico, ou seja, um método de várias formas.

Polimorfismo - Exemplo

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

    def calcular_vendas(self):
        # Método padrão que pode ser sobrescrito nas subclasses
        return 0
```

```python
class Vendedor(Pessoa):
    def __init__(self, nome):
        super().__init__(nome)
```

```python
class Diretor(Pessoa):
    def __init__(self, nome):
        super().__init__(nome)
```

```python
# Exemplo de uso
vendedor1 = Vendedor("João")
diretor1 = Diretor("Maria")

print(f"Vendas do {vendedor1.nome}: R${vendedor1.calcular_vendas()}")
print(f"Vendas do {diretor1.nome}: R${diretor1.calcular_vendas()}")
```

Qual a saída ???

Ciência da Computação

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

    def calcular_vendas(self):
        # Método padrão que pode ser sobrescrito nas subclasses
        return 0
```

```python
class Vendedor(Pessoa):
    def __init__(self, nome):
        super().__init__(nome)
```

```python
class Diretor(Pessoa):
    def __init__(self, nome):
        super().__init__(nome)
```

```python
# Exemplo de uso
vendedor1 = Vendedor("João")
diretor1 = Diretor("Maria")

print(f"Vendas do {vendedor1.nome}: R${vendedor1.calcular_vendas()}")
print(f"Vendas do {diretor1.nome}: R${diretor1.calcular_vendas()}")
```

Qual a saída ???

```
Vendas do João: R$0
Vendas do Maria: R$0
```

Neste exemplo, o método implementado na superclasse **‘Pessoa’** será executado, uma vez que é herdado pelas subclasses **‘Vendedor’** e **‘Diretor’**. Dessa forma, ele pode ser invocado por objetos instanciados dessas subclasses

Ciência da Computação

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

    def calcular_vendas(self):
        # Método padrão que pode ser sobrescrito nas subclasses
        return 0
```

```python
class Vendedor(Pessoa):
    def calcular_vendas(self):
        # Implementação específica para o vendedor
        valor_unitario = 10
        produtos_vendidos = 1500
        return valor_unitario * produtos_vendidos
```

```python
class Diretor(Pessoa):
    def __init__(self, nome):
        super().__init__(nome)
```

```python
# Exemplo de uso
vendedor1 = Vendedor("João")
diretor1 = Diretor("Maria")

print(f"Vendas do {vendedor1.nome}: R${vendedor1.calcular_vendas()}")
print(f"Vendas do {diretor1.nome}: R${diretor1.calcular_vendas()}")
```

E agora,
qual a saída ???

Ciência da
Computação

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

    def calcular_vendas(self):
        # Método padrão que pode ser sobrescrito nas subclasses
        return 0
```

```python
class Vendedor(Pessoa):
    def calcular_vendas(self):
        # Implementação específica para o vendedor
        valor_unitario = 10
        produtos_vendidos = 1500
        return valor_unitario * produtos_vendidos
```

```python
class Diretor(Pessoa):
    def __init__(self, nome):
        super().__init__(nome)
```

```python
# Exemplo de uso
vendedor1 = Vendedor("João")
diretor1 = Diretor("Maria")

print(f"Vendas do {vendedor1.nome}: R${vendedor1.calcular_vendas()}")
print(f"Vendas do {diretor1.nome}: R${diretor1.calcular_vendas()}")
```

E agora,
qual a saída ???

```
Vendas do João: R$15000
Vendas do Maria: R$0
```

Aqui ocorre uma distinção importante, uma vez que o método **‘calcular_vendas()’** foi sobrescrito na classe ‘**Vendedor**’. Isso significa que, ao chamar esse método a partir de uma instância desta classe, o código que será executado é o que foi especificamente definido na sobrescrita, refletindo o comportamento personalizado destinado aos vendedores.

Polimorfismo - Exemplo

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

    def calcular_vendas(self):
        # Método padrão que pode ser sobrescrito nas subclasses
        return 0
```

```python
class Vendedor(Pessoa):
    def calcular_vendas(self):
        # Implementação específica para o vendedor
        valor_unitario = 10
        produtos_vendidos = 1500
        return valor_unitario * produtos_vendidos
```

```python
class Diretor(Pessoa):
    def calcular_vendas(self):
        # Implementação específica para o diretor
        valor_unitario = 10
        produtos_vendidos = 1500
        comissao = (valor_unitario * produtos_vendidos) * 10 / 100
        return (valor_unitario * produtos_vendidos) + comissao
```

```python
# Exemplo de uso
vendedor1 = Vendedor("João")
diretor1 = Diretor("Maria")

print(f"Vendas do {vendedor1.nome}: R${vendedor1.calcular_vendas()}")  # Saída: Vendas do João: R$15000
print(f"Vendas do {diretor1.nome}: R${diretor1.calcular_vendas()}")    # Saída: Vendas do Maria: R$16500
```

Abstração e Encapsulamento

- **Abstração** é o processo de identificar e definir características essenciais de objetos do mundo real, “traduzindo”, ou seja, representando essas características em forma de **classes** e **objetos**.

- **Encapsulamento** refere-se à ideia de que os detalhes internos de um objeto devem ser ocultos para outros objetos e acessados apenas por meio de interfaces bem definidas.
  - Isso significa que os dados dentro de um objeto devem ser protegidos do acesso direto e modificações externas, sendo acessados apenas por métodos específicos.

Herança e Polimorfismo

- **Herança** é o princípio que permite que uma classe (subclasse) herde os atributos e métodos de outra classe (superclasse), facilitando a reutilização de código, a organização hierárquica e a promoção de relações entre objetos.
  - Lembre-se: "**é uma**" e não "tem uma"

- **Polimorfismo** é o princípio a partir do qual as classes derivadas de uma única classe base são capazes de invocar os métodos que, embora apresentem a mesma assinatura, comportam-se de maneira diferente para cada uma das classes derivadas.

Ciência da Computação

Ciência da
Computação