

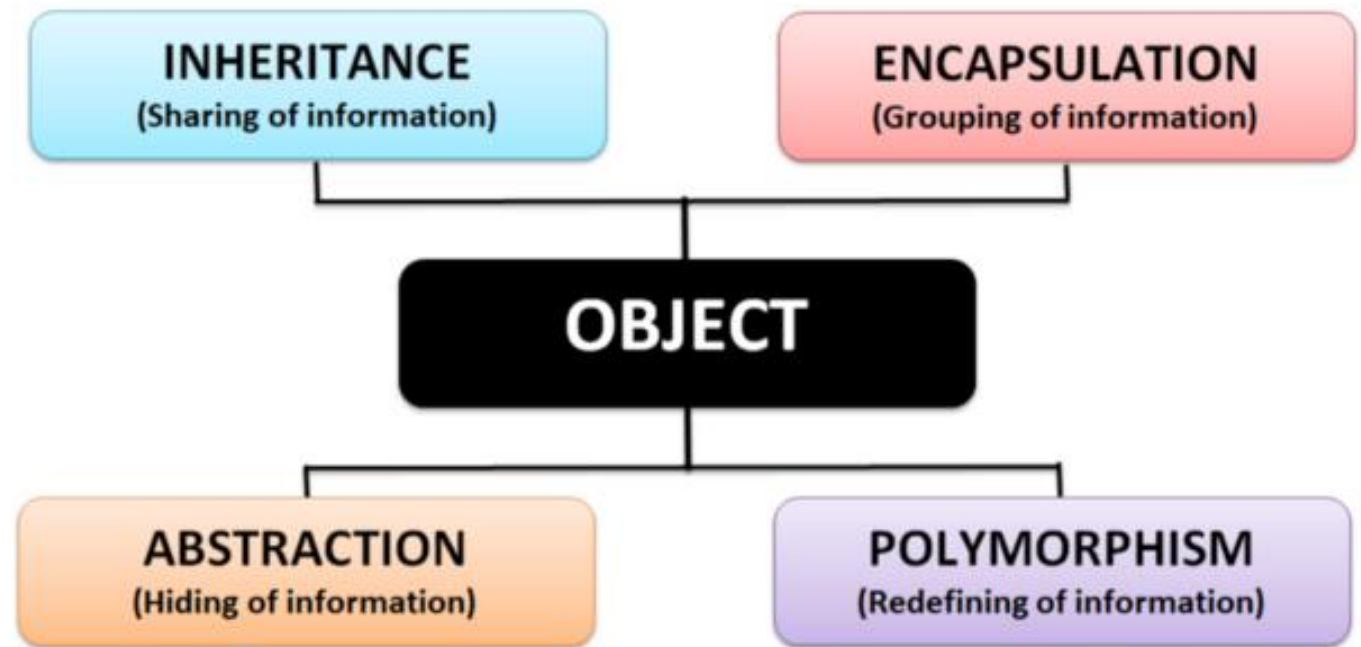


Ciência da **Computação**

Programação Orientada a Objetos
Prof. Luciano Rodrigo Ferretto



Pilares da Orientação a objetos

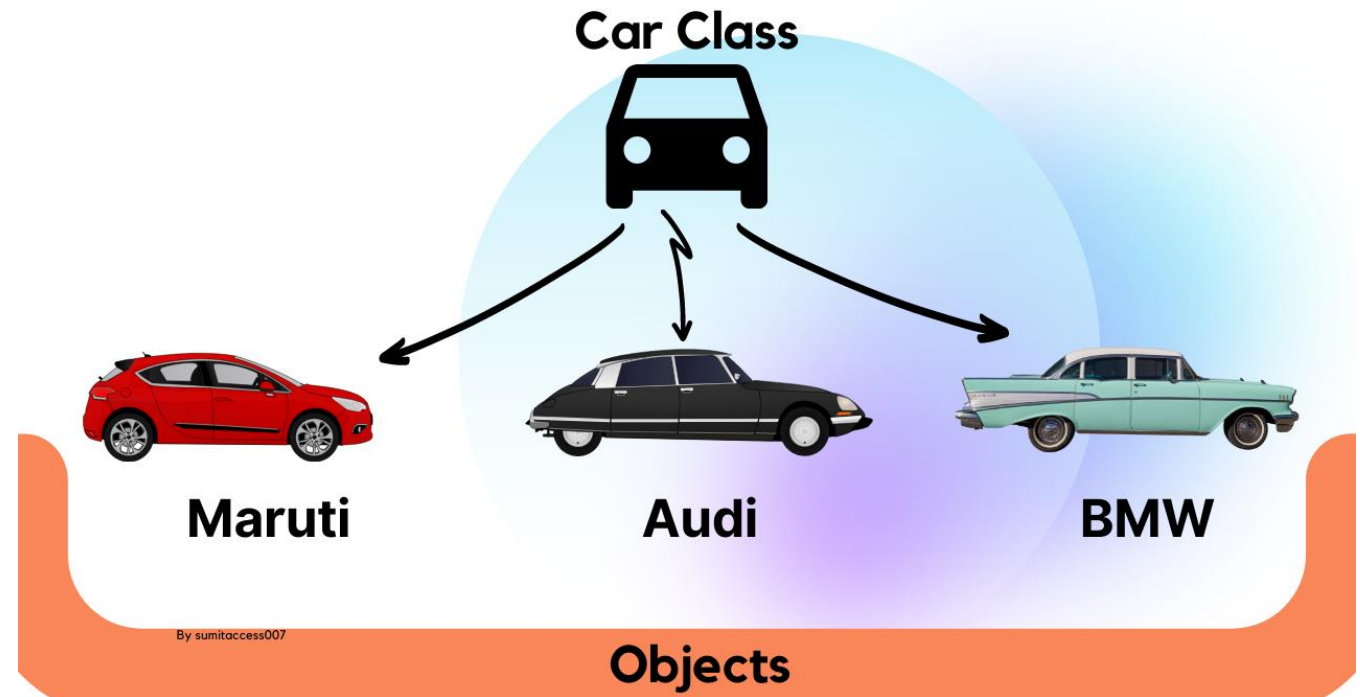
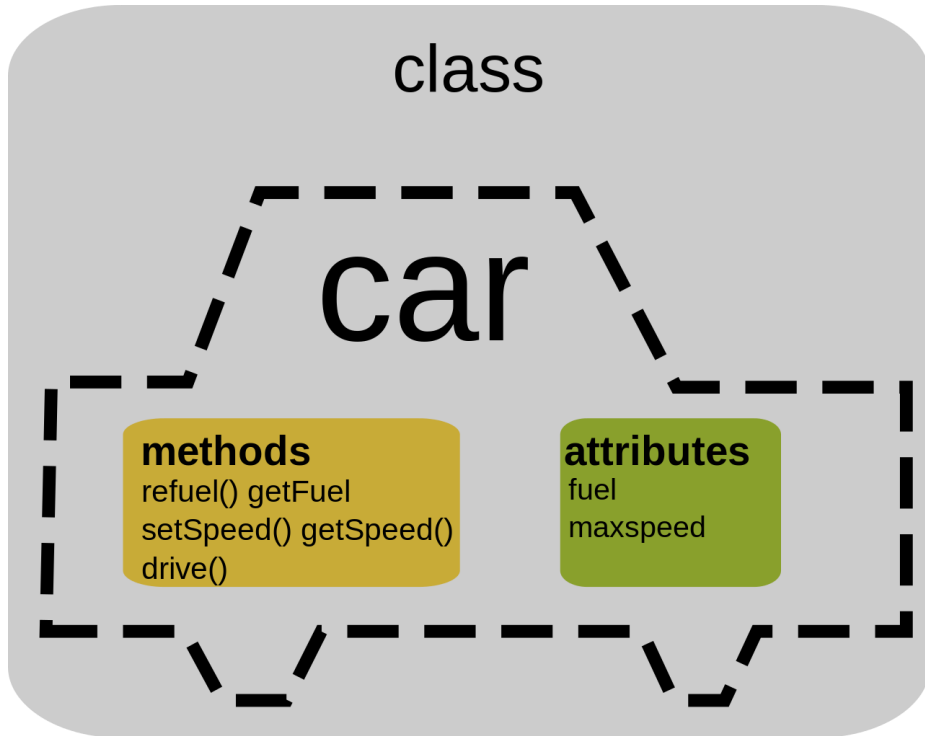


Abstração

- **Abstração** em Programação Orientada a Objetos 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**.
- 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



Encapsulamento em Programação Orientada a Objetos

Encapsulamento

- **Encapsulamento** se refere à prática de ocultar os detalhes internos de um objeto e expor apenas as operações relevantes para manipular esses detalhes.
- As principais vantagens do encapsulamento são:
 - **Segurança:** Protege contra alterações indevidas nos dados e garante a integridade do código.
 - **Manutenabilidade:** Facilita a compreensão e a modificação do código, isolando as mudanças.
 - **Reutilização:** Permite que classes sejam reutilizadas em diferentes contextos sem afetar o código.
 - **Flexibilidade:** Permite modificar detalhes de implementação sem afetar o código externo.

Encapsulamento – Modificadores de Acesso

- Na maioria das linguagens de programação orientada a objetos, o encapsulamento é alcançado através da definição de **modificadores de acesso**.
- Aqui estão os principais modificadores de acesso em Java:
 - **public**: Atributos e métodos são acessíveis de qualquer lugar.
 - **private**: Atributos e métodos são acessíveis somente dentro da própria classe.
 - **protected**: Atributos e métodos são acessíveis dentro da própria classe, dentro do mesmo pacote, e em suas subclasses em outros pacotes.
 - **default** (sem modificador): Atributos e métodos são acessíveis apenas dentro do mesmo pacote.

Encapsulamento – Modificadores de Acesso em Python

- No Python não existem modificadores de acesso?
 - Não é que não existam modificadores de acesso em Python, mas sim que eles funcionam de maneira diferente do que em outras linguagens como Java, C++, C#, etc.
- Em Python, os modificadores de acesso são implementados como **convenções de nomenclatura**, não como palavras-chave.
 - Como são “convenções” na prática eles não conseguem realmente impedir um acesso dado “private” por exemplo.

Encapsulamento – Modificadores de Acesso em Python

- Aqui estão as convenções de nomenclatura para modificadores de acesso em Python:
 - **public**: Atributos e métodos começam com letras minúsculas.
 - **private**: Atributos e métodos começam com dois underlines __
 - **protected**: Atributos e métodos começam com um underline _

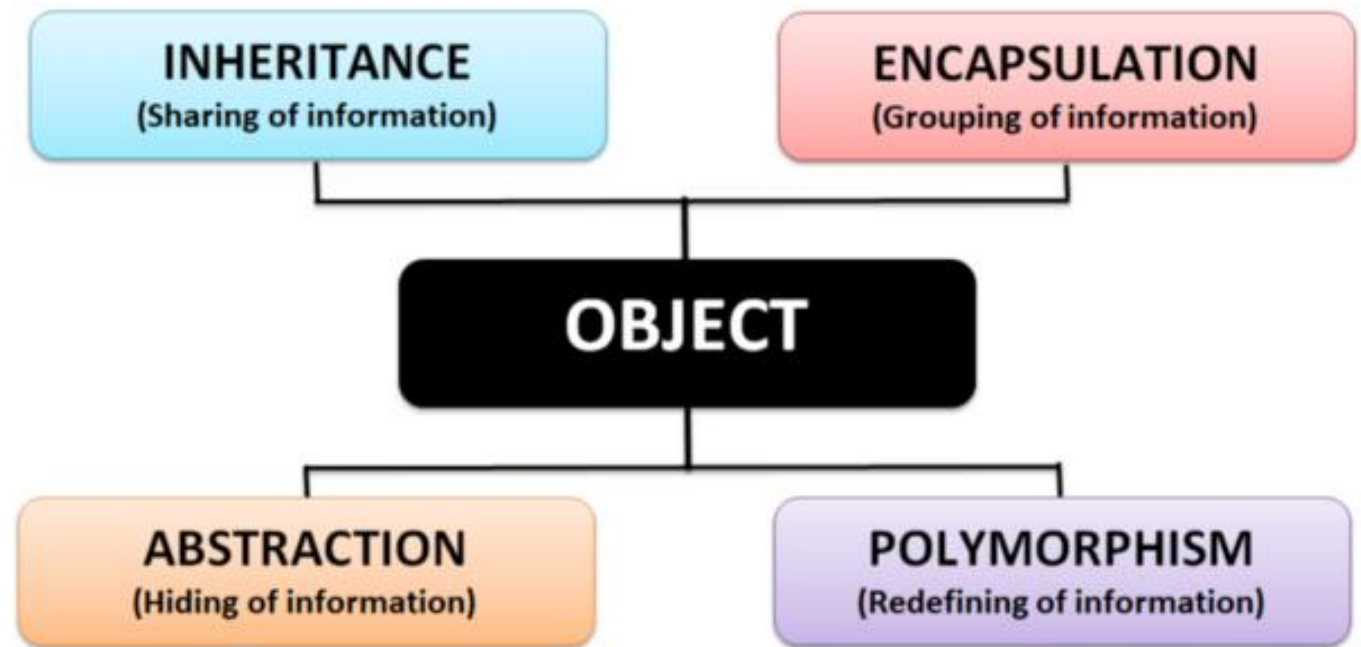
Esta são apenas convenções de nomenclatura, ou seja, não possuem efeito prático

Encapsulamento – Modificadores de Acesso em Python

- Em geral, os atributos de uma classe são privados (private) por padrão. Exceções a essa regra existem para casos específicos onde o padrão não se aplica.
- Para acessar ou modificar os valores dos atributos privados, outros objetos/classes devem utilizar métodos específicos, conhecidos como **getters and setters**

```
1 class Veiculo:
2     def __init__(self, marca, modelo, ano, placa):
3         self.__marca = marca
4         self.__modelo = modelo
5         self.__ano = ano
6         self.__placa = placa
7     def get_marca(self):
8         return self.__marca
9     def set_marca(self, marca):
10        self.__marca = marca
11
12    def get_modelo(self):
13        return self.__modelo
14    def set_model(self, modelo):
15        self.__modelo = modelo
16
17    def get_ano(self):
18        return self.__ano
19    def set_ano(self, ano):
20        self.__ano = ano
21
22    def get_placa(self):
23        return self.__placa
24    def set_placa(self, placa):
25        self.__placa = placa
```

Pilares da Orientação a objetos



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 em Programação Orientada a Objetos

Vamos pensar um pouco ...

- Pense no algoritmo que temos até agora:



```
1 class Veiculo:
2     def __init__(self, marca, modelo, ano, placa):
3         self.marca = marca
4         self.modelo = modelo
5         self.ano = ano
6         self.placa = placa
7
8     def __str__(self):
9         return f'Marca: {self.marca}, Modelo: {self.modelo}, Ano: {self.ano}, Placa: {self.placa}'
```

- Agora queremos trabalhar com um tipo específico, no caso Moto. Então teríamos a classe:

```
1 class Moto:
2     def __init__(self, marca, modelo, ano, placa, cilindradas):
3         self.marca = marca
4         self.modelo = modelo
5         self.ano = ano
6         self.placa = placa
7         self.cilindradas = cilindradas
8
9     def __str__(self):
10        return f'Marca: {self.marca}, Modelo: {self.modelo}, Ano: {self.ano}, Placa: {self.placa}, Cilindradas: {self.cilindradas}'
```


Vamos pensar um pouco ...

- Observe que temos duas classes que possuem atributos e métodos em comum (iguais ou muito semelhantes). Se pensarmos bem, no futuro poderemos ter mais, como por exemplo uma classe Caminhao
- Em POO podemos nos beneficiar do princípio de Herança para eliminar a duplicação de código e organizar nosso sistema.
- Com a Herança, podemos fazer com que a classe Moto herde os atributos e métodos da classe Veiculo, precisando implementar apenas os atributos e métodos específicos dos objetos “Moto”



```
1 class Veiculo:
2     def __init__(self, marca, modelo, ano, placa):
3         self.marca = marca
4         self.modelo = modelo
5         self.ano = ano
6         self.placa = placa
7
8     def __str__(self):
9         return f'Marca: {self.marca}, Modelo: {self.modelo}, Ano: {self.ano}, Placa: {self.placa}'
10
11 class Moto(Veiculo):
12     def __init__(self, marca, modelo, ano, placa, cilindradas):
13         super().__init__(marca, modelo, ano, placa)
14         self.cilindradas = cilindradas
15
16     def __str__(self):
17         return super().__str__() + f'\nCilindradas: {self.__cilindradas}'
18
```

```
1 class Veiculo:
2     def __init__(self, marca, modelo, ano, placa):
3         self.marca = marca
4         self.modelo = modelo
5         self.ano = ano
6         self.placa = placa
7
8     def __str__(self):
9         return f'Marca: {self.marca}, Modelo: {self.modelo}, Ano: {self.ano}, Placa: {self.placa}'
10
11 class Moto(Veiculo):
12     def __init__(self, marca, modelo, ano, placa, cilindradas):
13         super().__init__(marca, modelo, ano, placa)
14         self.cilindradas = cilindradas
15
16     def __str__(self):
17         return super().__str__() + f'\nCilindradas: {self.__cilindradas}'
18
```

Herança



O conceito de herança na programação orientada a objetos é utilizado para se permitir a reutilização de um código. A herança possibilita que classes compartilhem seus atributos e métodos entre si. Segundo Tucker e Noonan (2009, p. 335),

[...] o paradigma orientado a objetos suporta reutilização de código por intermédio da herança. As classes existem em uma linguagem orientada a objetos em uma hierarquia de classes. Uma classe pode ser declarada como uma subclasse de outra classe, que é chamada de classe mãe ou superclasse.

Herança

- Permite a reutilização de código e a organização de classes de forma eficiente.
- 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 - Benefícios

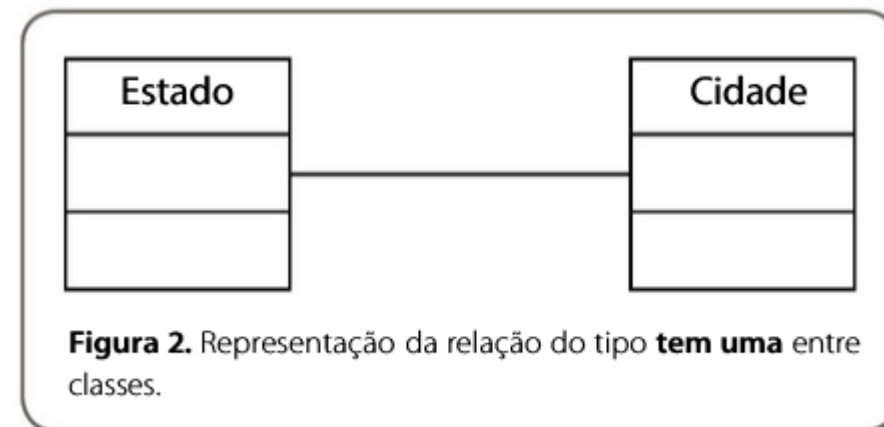
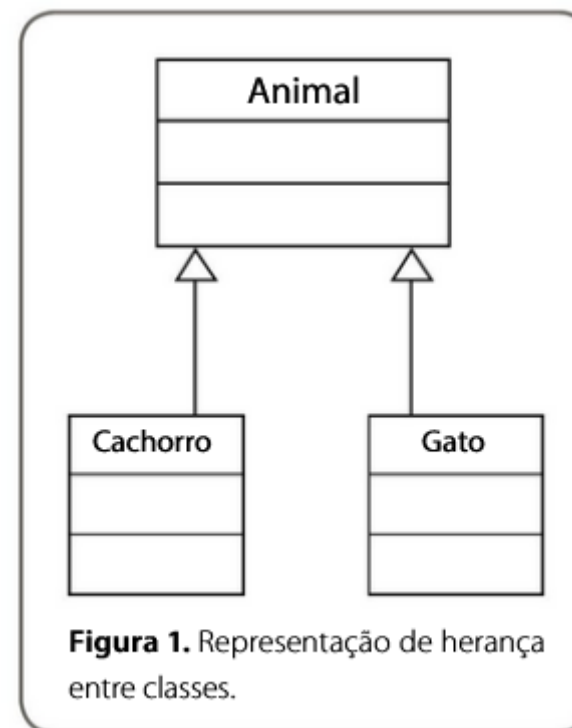
- **Reutilização de código:** Evita duplicação de código ao herdar características e comportamentos de classes já existentes.
- **Organização do código:** Promove a organização do código ao agrupar características e comportamentos comuns em uma única classe pai. Isso torna o código mais legível e facilita a compreensão da estrutura do programa.
- **Manutenibilidade:** Facilita a manutenção do código ao centralizar características e comportamentos em uma única classe.
- **Extensibilidade:** Permite criar novas classes com base em classes existentes, expandindo a funcionalidade do programa.

Herança

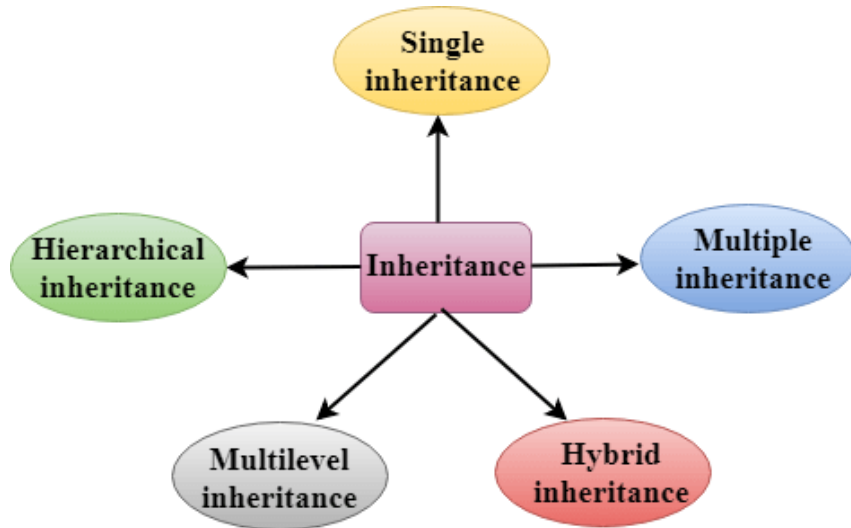
Na programação orientada a objetos, a relação de herança entre classes é a relação em que uma classe é do tipo **é uma**, e não do tipo **tem uma**. Esta é uma das confusões recorrentes na construção de programas em orientação a objetos.

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`.

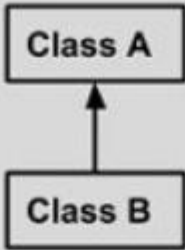
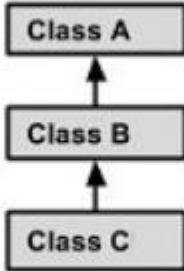
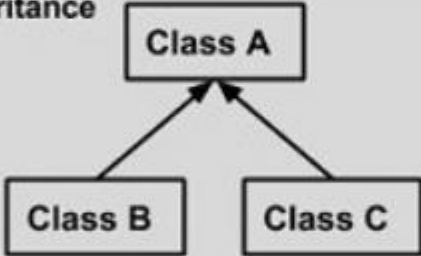
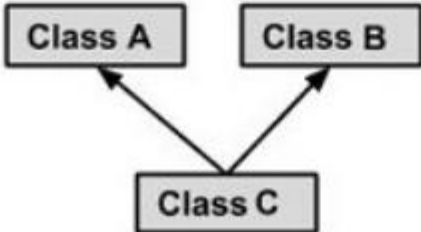
Compare, agora, a relação apresentada na Figura 2, que representa uma relação do tipo **tem uma**. Perceba que, neste caso, uma relação de herança entre as classes `Estado` e `Cidade` não faz sentido, visto que um estado possui cidades, mas uma cidade não é um estado.



Herança



Types of Inheritance:

Single Inheritance  <pre> graph BT B[Class B] --> A[Class A] </pre>	<pre> public class A { } public class B extends A { } </pre>
Multi Level Inheritance  <pre> graph BT C[Class C] --> B[Class B] B --> A[Class A] </pre>	<pre> public class A { } public class B extends A { } public class C extends B { } </pre>
Hierarchical Inheritance  <pre> graph BT B[Class B] --> A[Class A] C[Class C] --> A </pre>	<pre> public class A { } public class B extends A { } public class C extends A { } </pre>
Multiple Inheritance  <pre> graph BT C[Class C] --> A[Class A] C --> B[Class B] </pre>	<pre> public class A { } public class B { } public class C extends A,B { } // Java does not support mutiple Inheritance </pre>

Herança em Python

- Em Python, a herança é implementada usando a sintaxe
 - `class Filha(Pai):`
- onde **Pai** é a classe da qual você deseja herdar.
- A classe **filha** pode adicionar novos atributos e métodos ou substituir os existentes conforme necessário.
- Para acessar os métodos e atributos da classe pai dentro da classe filha, você pode usar a função **super()**.

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

    def comer(self):
        print(f"{self.nome} está comendo.")

    def dormir(self):
        print(f"{self.nome} está dormindo.")

class Cachorro(Animal):
    def __init__(self, nome, idade, especie, raça, porte):
        super().__init__(nome, idade, especie)
        self.raça = raça
        self.porte = porte

    def latir(self):
        print(f"{self.nome} está latindo!")

class Gato(Animal):
    def __init__(self, nome, idade, especie, raça, pelagem):
        super().__init__(nome, idade, especie)
        self.raça = raça
        self.pelagem = pelagem

    def miar(self):
        print(f"{self.nome} está miando!")
```

Herança em Python

- Ao instanciar um objeto a partir de classes *filhas*, podemos invocar e acessar atributos declarados tanto nas *subclasses*, como nas *super classes (pai)*

```
# Criando objetos das classes Cachorro e Gato
cachorro = Cachorro("Rex", 3, "Cachorro", "Labrador", "Médio")
gato = Gato("Mia", 2, "Gato", "Siamês", "Longa")

# Acessando atributos e métodos herdados da classe Animal
print(f"Nome do Cachorro: {cachorro.nome}")
print(f"Idade do Cachorro: {cachorro.idade}")
print(f"Espécie do Cachorro: {cachorro.especie}")
cachorro.comer()
cachorro.dormir()

print(f"Nome do Gato: {gato.nome}")
print(f"Idade do Gato: {gato.idade}")
print(f"Espécie do Gato: {gato.especie}")
gato.comer()
gato.dormir()

# Acessando atributos e métodos específicos das classes Cachorro e Gato
print(f"Raça do Cachorro: {cachorro.raça}")
print(f"Porte do Cachorro: {cachorro.porte}")
cachorro.latir()

print(f"Raça do Gato: {gato.raça}")
print(f"Pelagem do Gato: {gato.pelagem}")
gato.miar()
```

When **Developers** are debugging
a complex code
New bugs popping up be like

