Herança e Classes Abstratas

Programação Orientada a Objetos

Prof. Jônata Tyska Carvalho

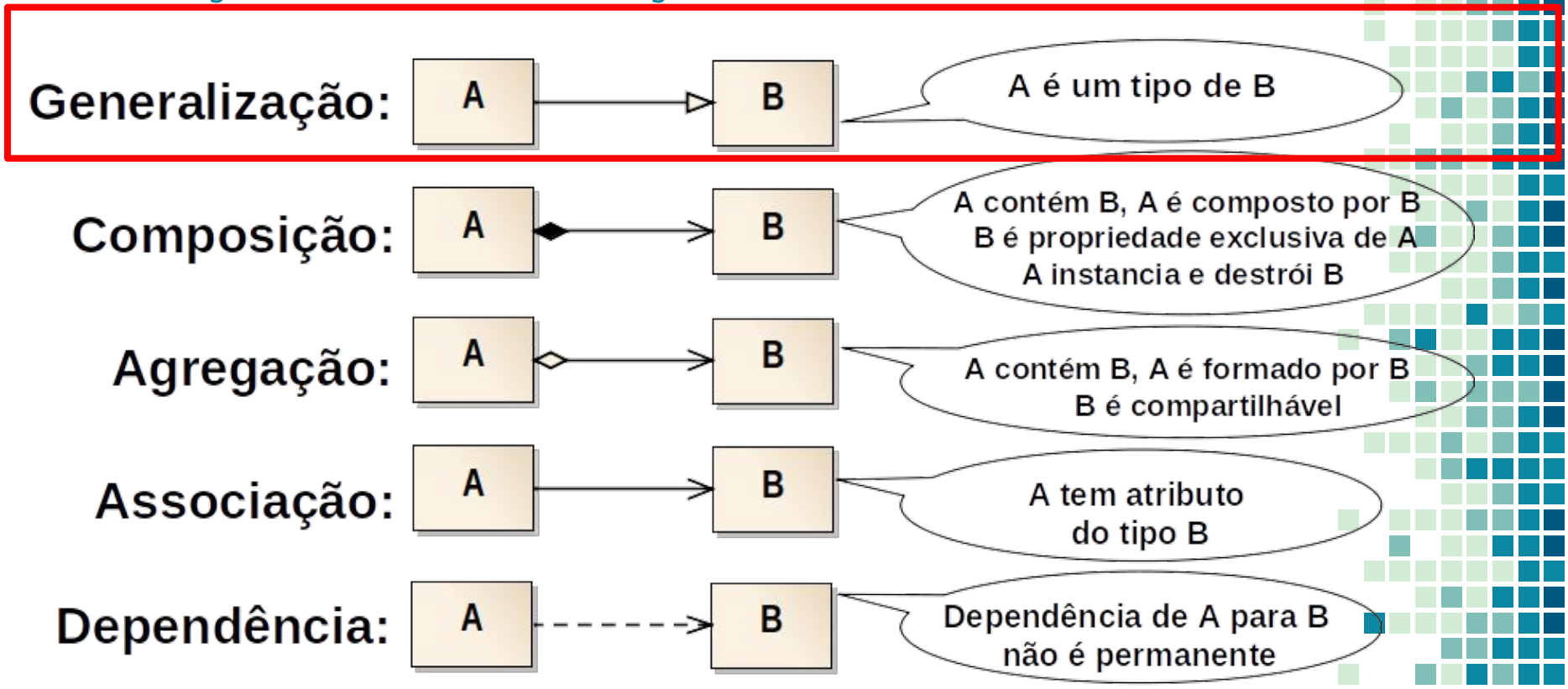
Herança: Agenda

1. Intuição e conceito
2. Benefícios e características
3. Classes Abstratas

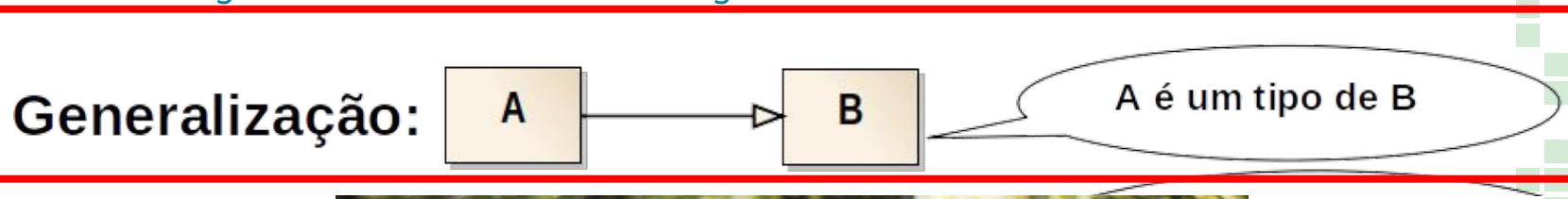
Herança: Agenda

1. **Intuição e conceito**
2. Benefícios e características
3. Classes Abstratas

Herança – Generalização

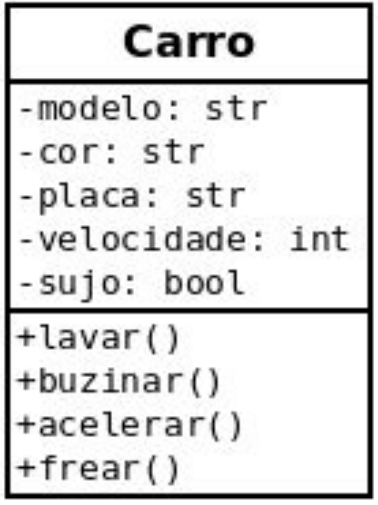


Herança – Generalização



https://unsplash.com/@azganmjeshtri

Exemplo


Carro
-modelo: str
-cor: str
-placa: str
-velocidade: int
-sujo: bool
+lavar()
+buzinar()
+acelerar()
+frear()


Bicicleta
-modelo: str
-cor: str
-velocidade: int
-sujo: bool
-marchas: int
-amortecedor: bool
+lavar()
+pedalar()
+frear()

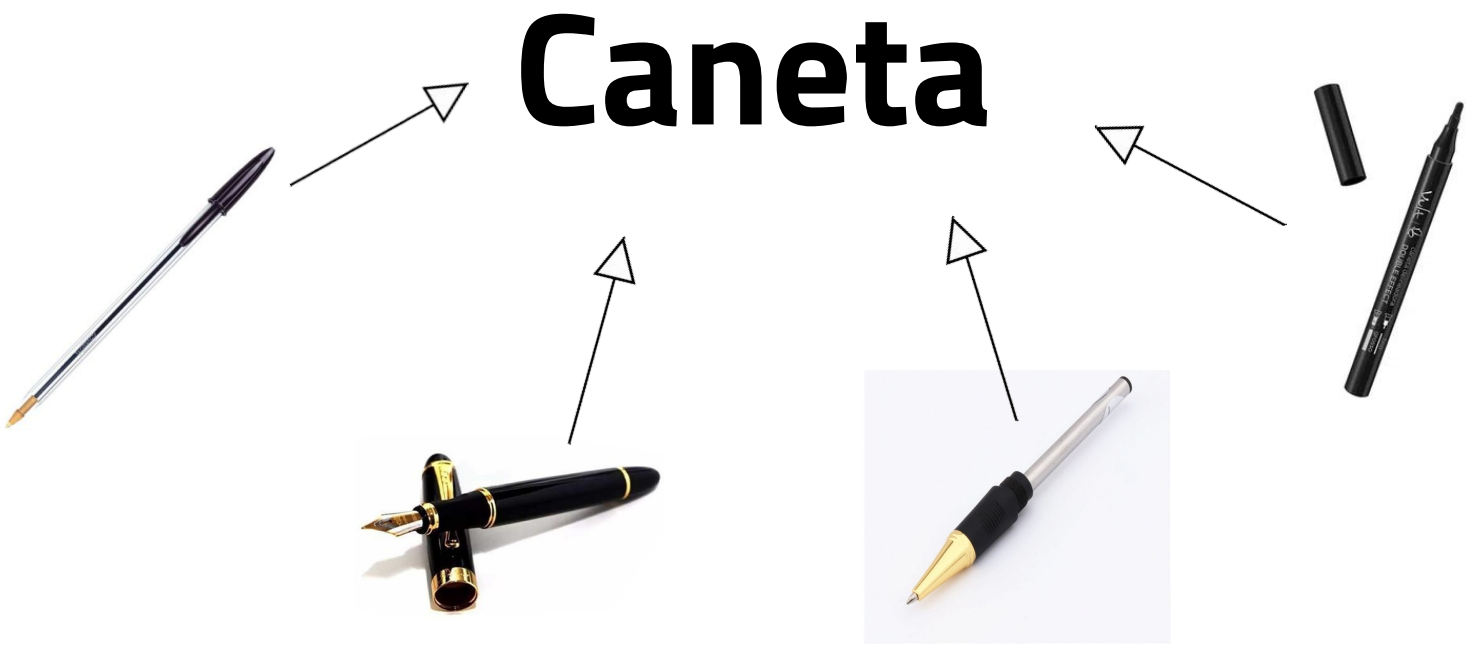
Intuição e conceito – O que é isso?

Intuição e conceito – O que é isso?

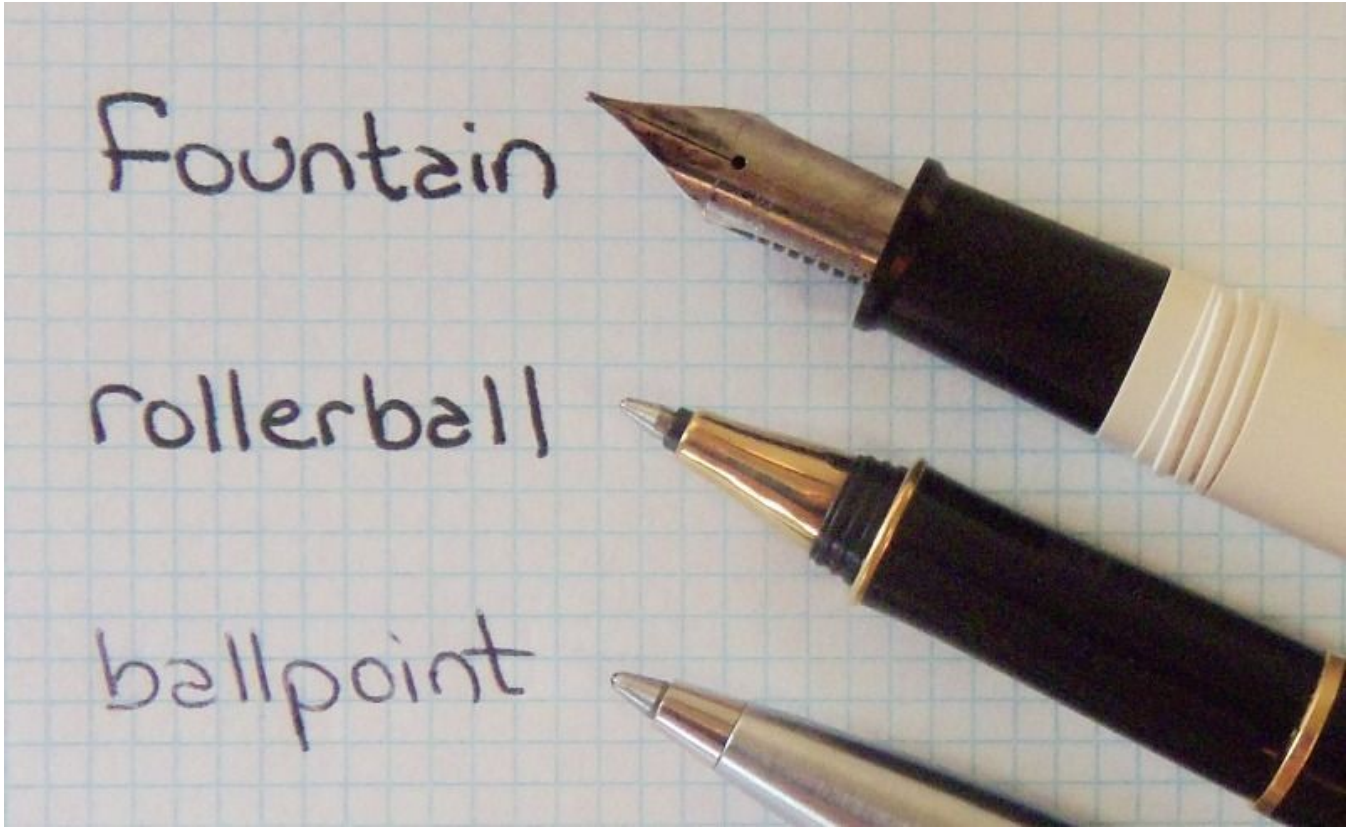
Intuição e conceito – O que é isso?

Intuição e conceito – O que é isso?

Intuição e conceito – Características comuns

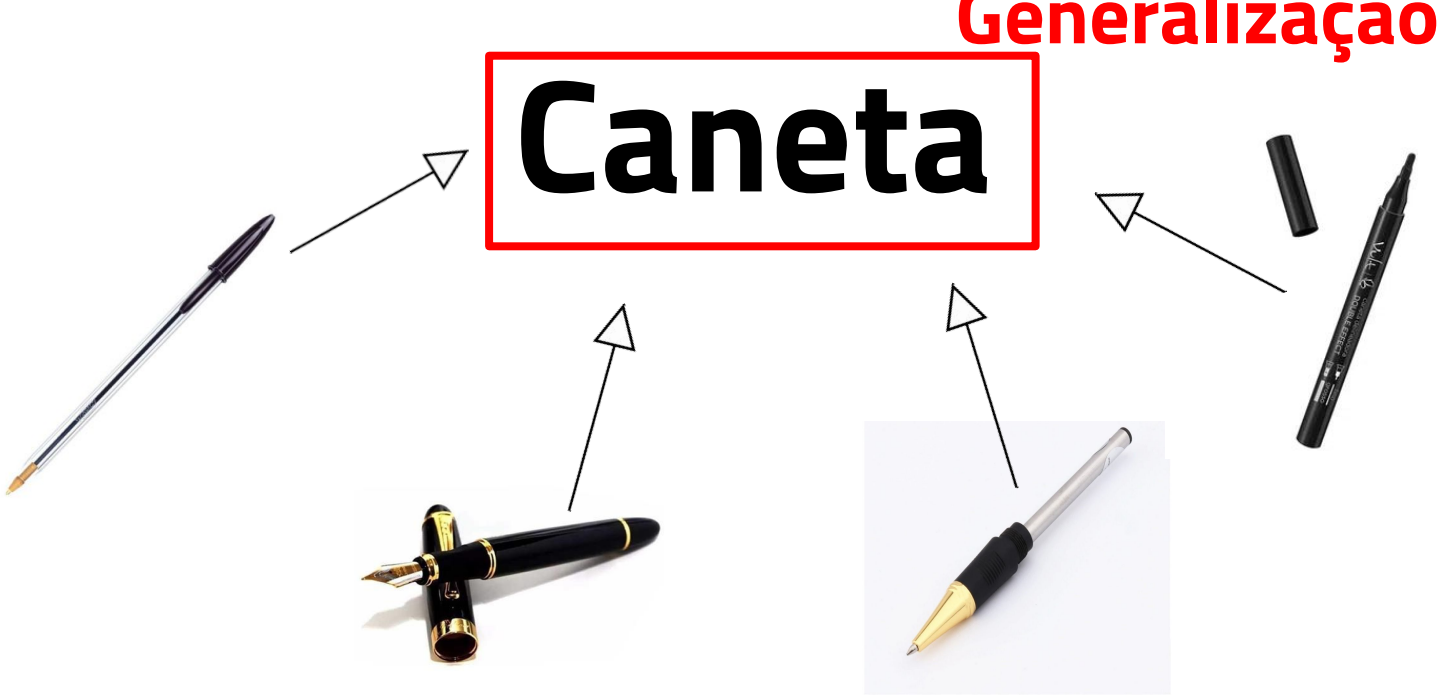


Intuição e conceito - Características Específicas



Intuição e conceito - abstração

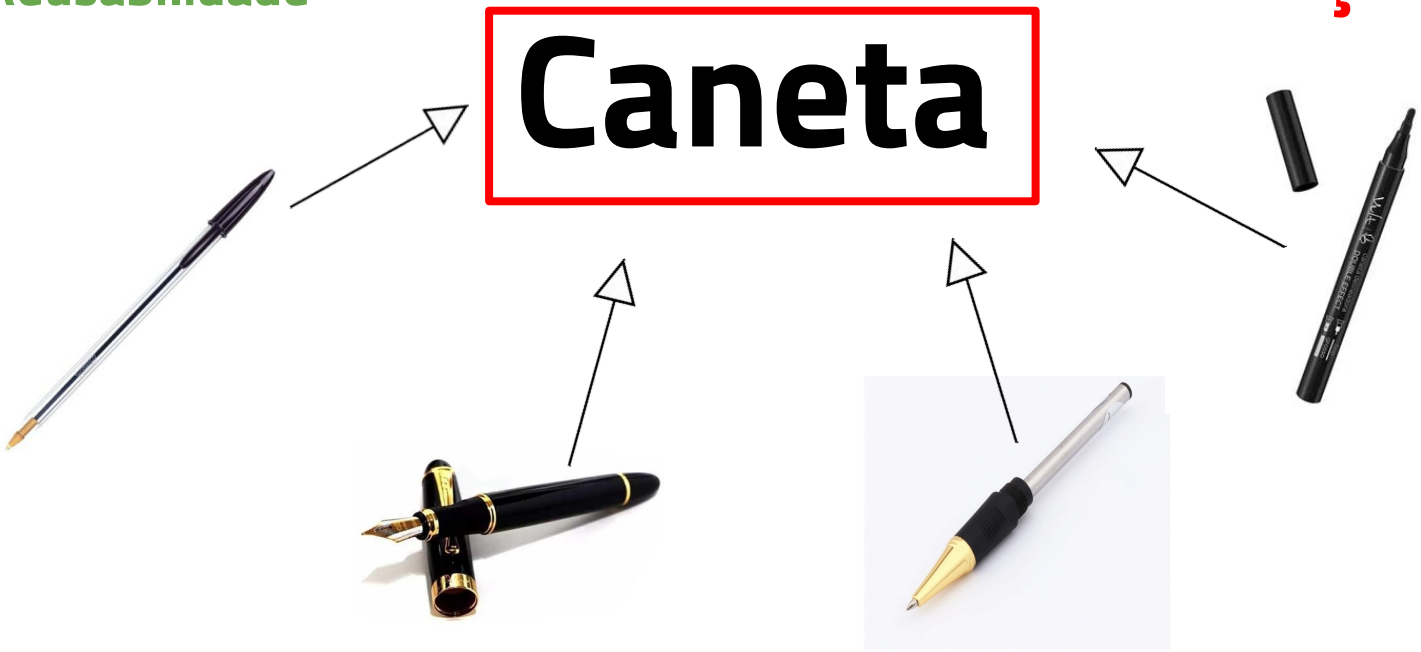
Generalização



Intuição e conceito - abstração

Evitar a replicação
Reusabilidade

Generalização



Intuição e conceito - especializar - refinar

Reusabilidade
Extensibilidade

Caneta

Especializações

Herança: Agenda

1. Intuição e conceito
2. **Benefícios e características**
3. Classes Abstratas

Ideia básica

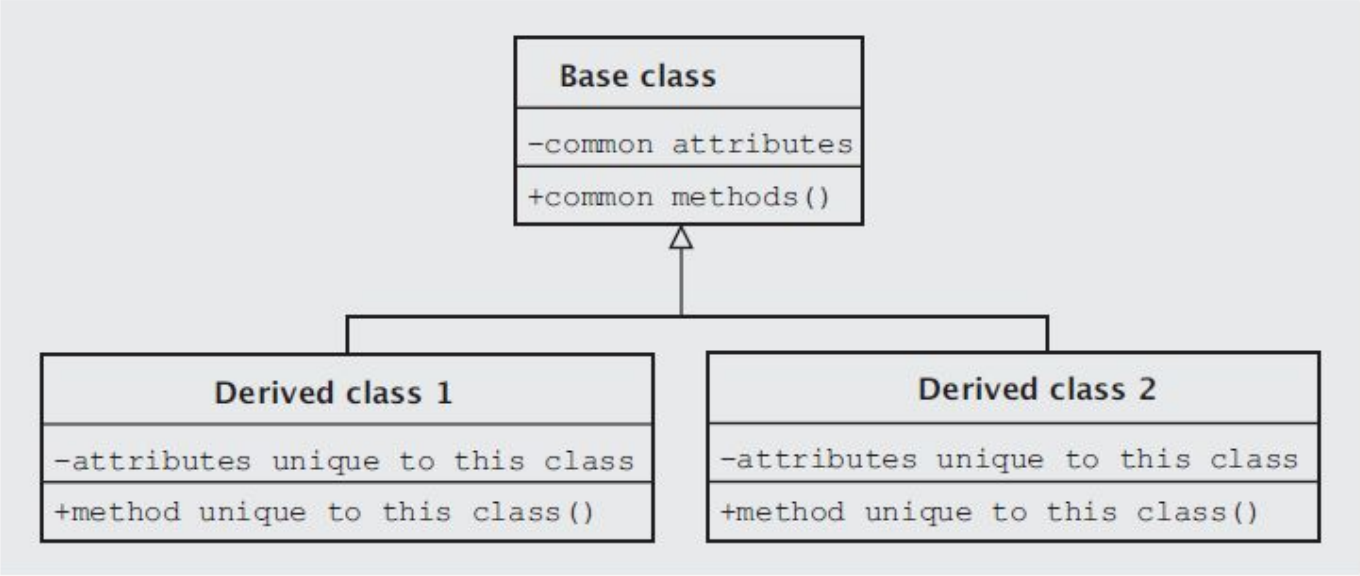


Figure 3.5 *Basic idea of inheritance*

DATHAN, B.; RAMNATH, S. Object-Oriented Analysis, Design and Implementation. Cham: Springer, 2015

Ideia básica - termos

Superclasse
Classe pai
Classe base

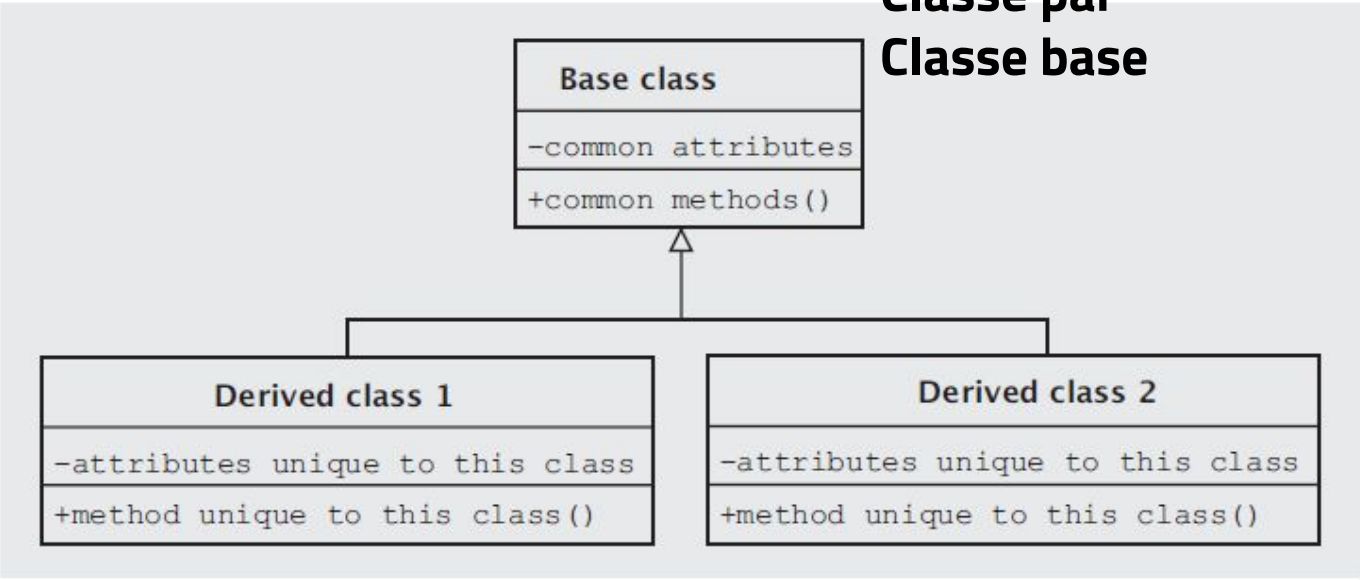


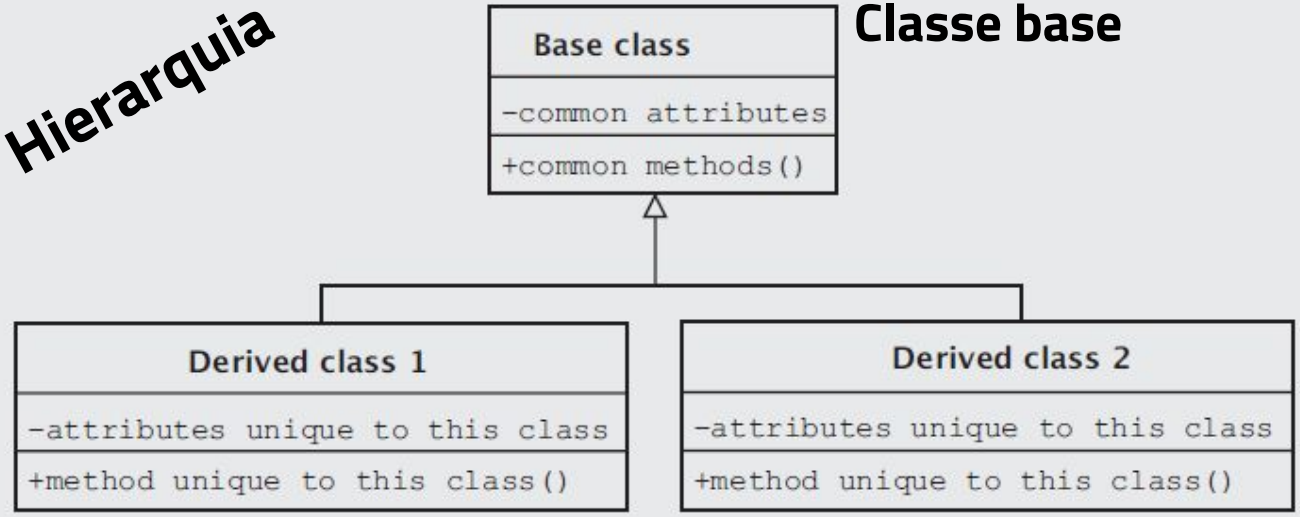
Figure 3.5 *Basic idea of inheritance*

DATHAN, B.; RAMNATH, S. Object-Oriented Analysis, Design and Implementation. Cham: Springer, 2015

Ideia básica - termos

Superclasse
Classe pai
Classe base

Hierarquia

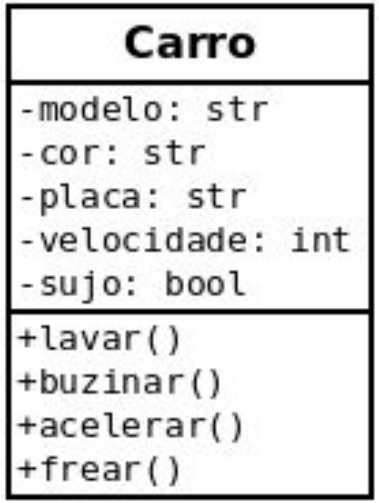


Classe filha
Classe derivada
Subclasse

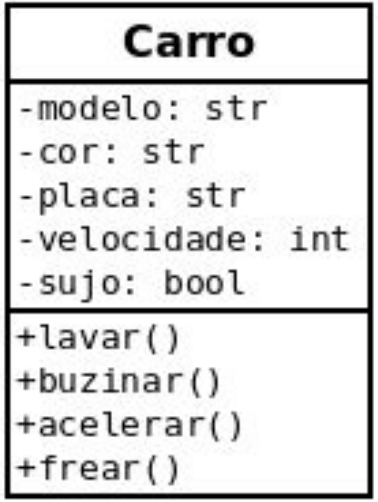
Figure 3.5 *Basic idea of inheritance*

DATHAN, B.; RAMNATH, S. Object-Oriented Analysis, Design and Implementation. Cham: Springer, 2015

Exemplo simples

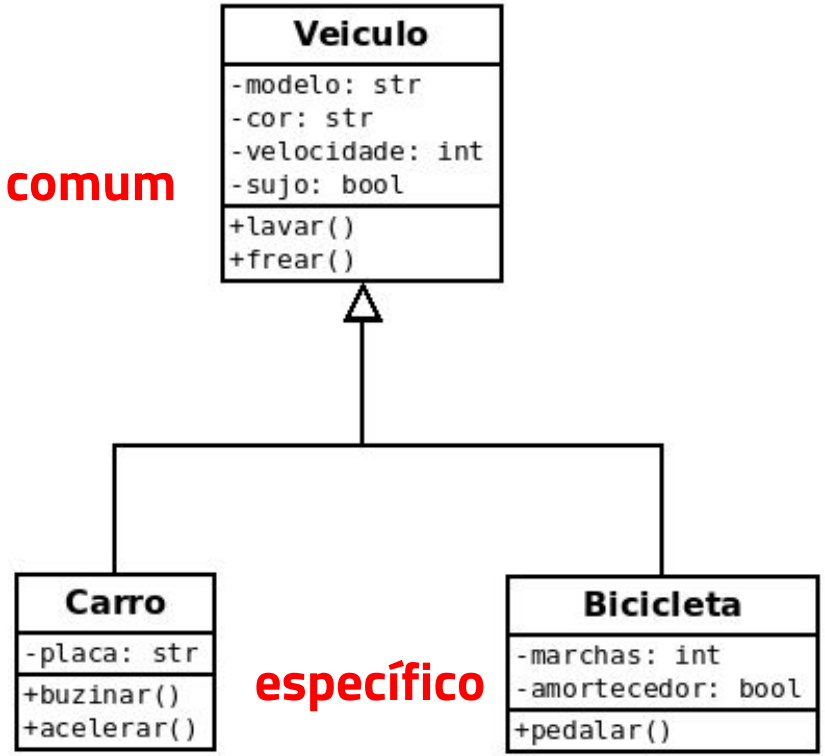


Exemplo simples



Como usar **generalização (herança)** para reduzir a redundância?

Exemplo simples



Benefícios

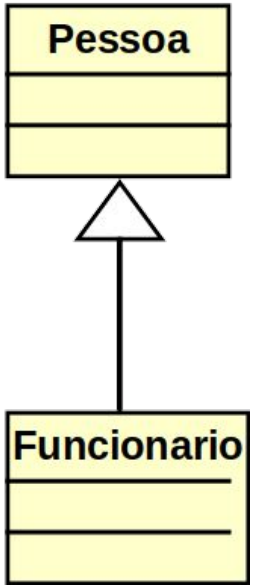
- Evitar replicação de código
- Aumentar reusabilidade
- Facilitar a extensibilidade

Características

Qual é o sentido da dependência?

Qual é a classe mais independente?

Qual é a classe com maior reusabilidade?



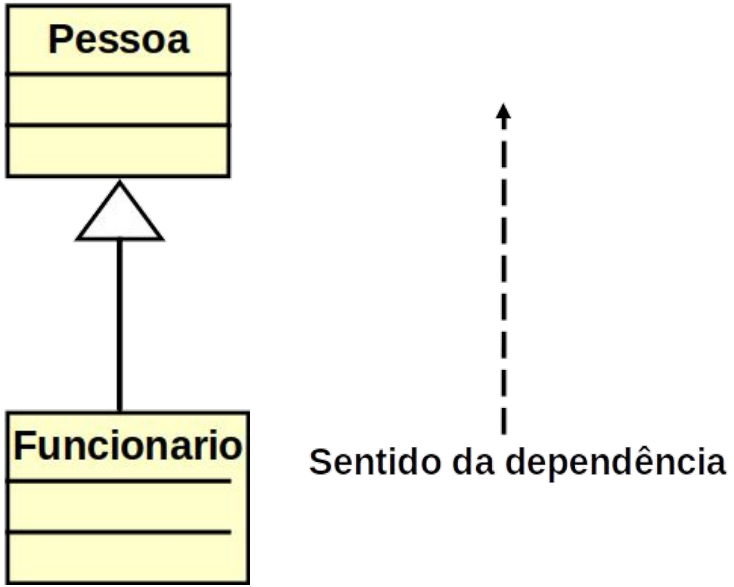
Características

Maior reusabilidade, Mais genérico
Maior abstração

Qual é o sentido da dependência?

Qual é a classe mais independente?

Qual é a classe com maior reusabilidade?



Menor reusabilidade, Mais específico,
Menor abstração

Características

- Subclasse herda atributos e metódos
  - SUBCLASSE NÃO TEM ACESSO AOS PRIVADOS
- Subclasse é um tipo da superclasse - tipagem estática
- É possível **criar** novos atributos e métodos (especializar)
  - E também **redefinir**: sobreposição e sobrecarga
    - Fundamentais para o *polimorfismo*!

Características

- **Sobreposição (*override*)**

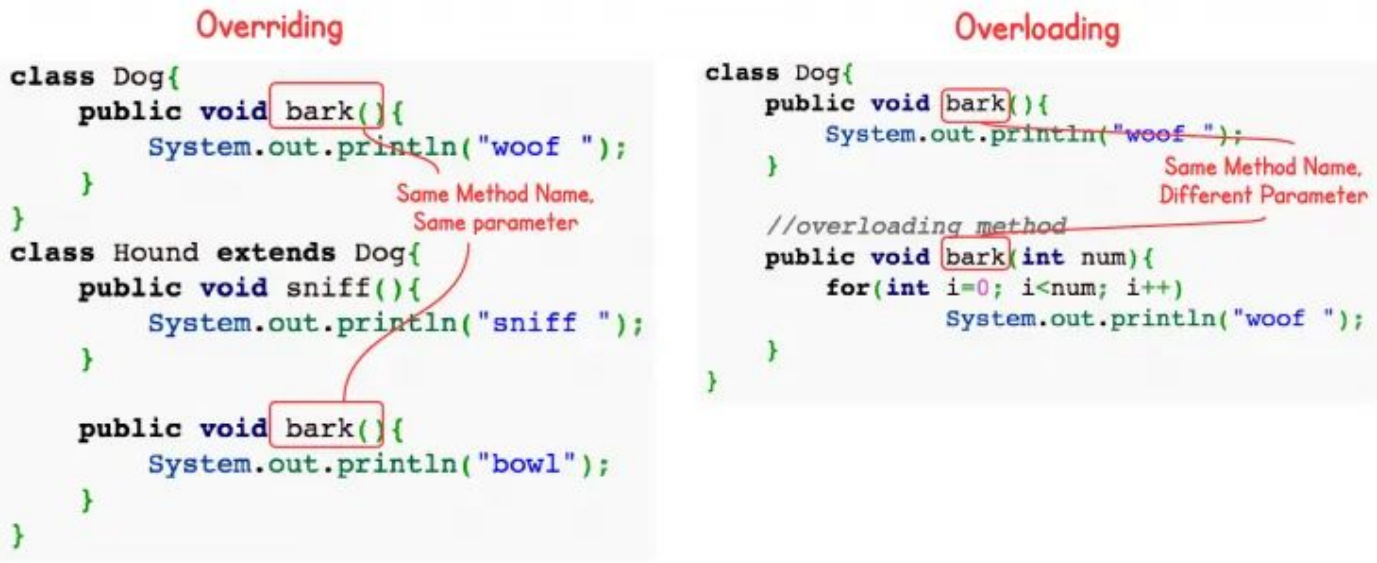
  redefinição de métodos

  nas subclasses

```python
class Veiculo:
    # ...

    def lavar(self):
        total = self.dono.dinheiro
        if total >= 30:
            self.dono.dinheiro = total - 30
            self.__sujo = False

    def frear(self):
        pass

    # ...


class Carro(Veiculo):
    # ...

    def frear(self):
        self.velocidade -= 10
        if self.velocidade < 0:
            self.velocidade = 0


class Bicicleta(Veiculo):
    # ...

    def lavar(self):
        total = self.dono.dinheiro
        if total >= 10:
            self.dono.dinheiro = total - 10
            self.sujo = False

    def frear(self):
        self.velocidade -= 1
        if self.velocidade < 0:
            self.velocidade = 0
```

Características

- **Sobrecarga (*overload*)** - definição de múltiplas assinaturas para um mesmo método (não existe em Python)

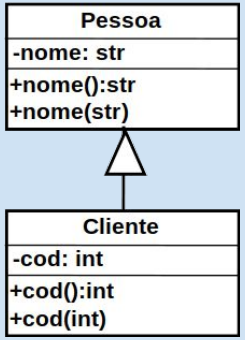


Tipos de herança

- **Simples**

  subclasse herda

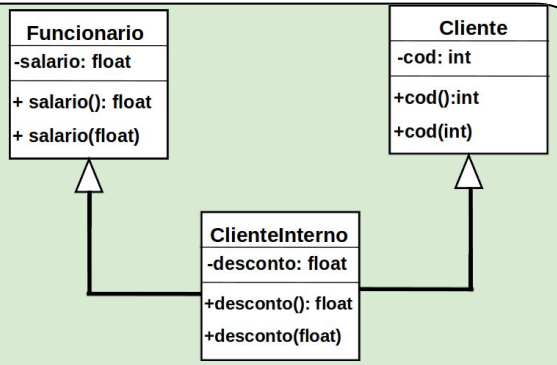
  de uma superclasse



- **Múltipla**

  herda de duas ou mais

  superclasses



Herança: Agenda

1. Intuição e conceito
2. Benefícios e características
3. **Classes Abstratas**

Classes Abstratas

Você já viu um veículo?

https://unsplash.com/@findracadabra

Classes Abstratas

Você já viu uma forma geométrica?

https://unsplash.com/@findracadabra

Classes Abstratas

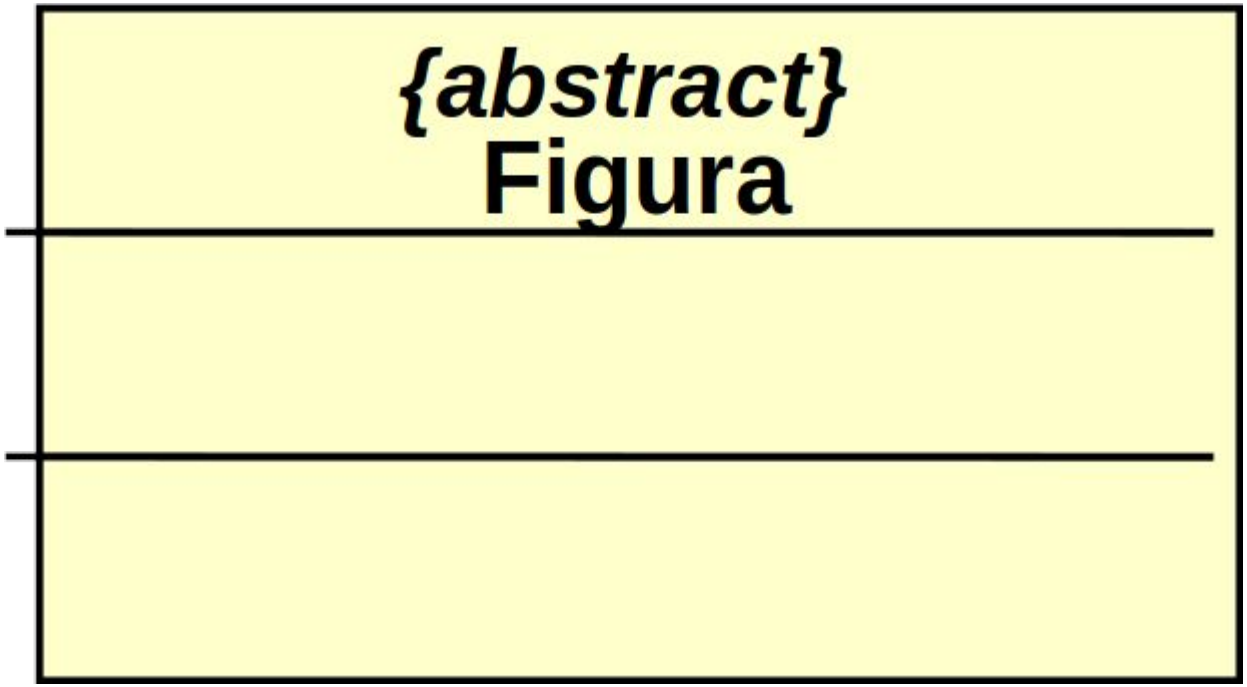
Você já viu um animal?

https://unsplash.com/@findracadabra

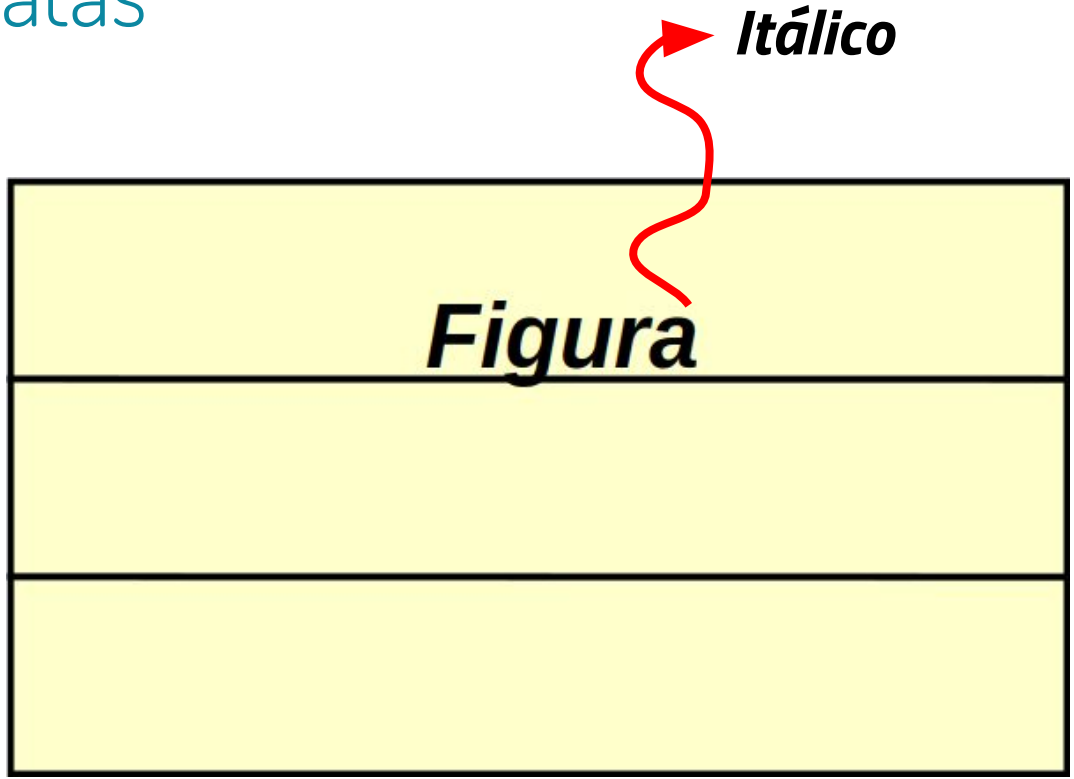
Classes Abstratas

- Abstrato demais (incompleto) para ter objetos
  - Métodos podem ser abstratos
- Permite definir uma base para hierarquia de classes
- Pode ter atributos e métodos definidos
- Python não possui palavra chave - usa ABC e decorators

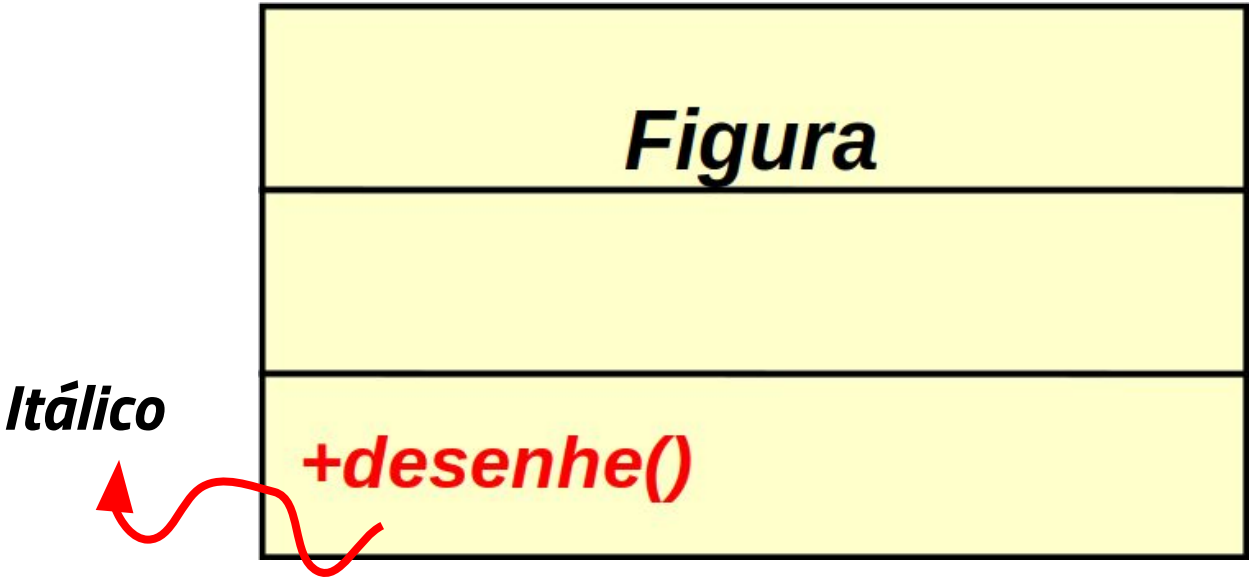
Classes Abstratas



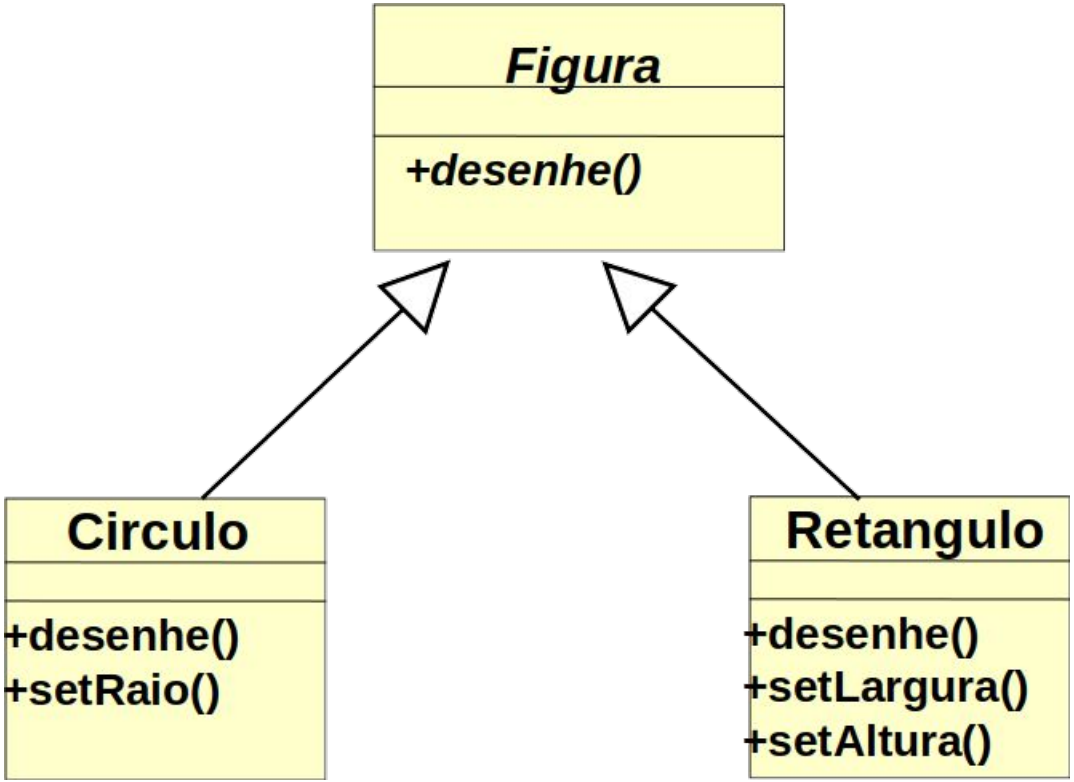
Classes Abstratas



Classes Abstratas



Classes Abstratas



Classes Abstratas - Python

```
from abc import ABC, abstractmethod
```

```
class Figura(ABC):
    @abstractmethod
    def __init__(self):
        pass

    @abstractmethod
    def desenhe(self):
        pass
```

Classes Abstratas - Python

```
from abc import ABC, abstractmethod

class Figura(ABC):
    @abstractmethod
    def __init__(self):
        pass

    @abstractmethod
    def desenhe(self):
        pass
```

Herança de ABC indica que a Classe é abstrata

Classes Abstratas - Python

```
from abc import ABC, abstractmethod

class Figura(ABC):
    @abstractmethod
    def __init__(self):
        pass

    @abstractmethod
    def desenhe(self):
        pass
```

@abstractmethod
método abstrato
impede de
instanciar a classe

Classes Abstratas - Python

```
class Circulo(Figura):
    def __init__(self, raio: int):
        super().__init__()
        self.__raio = raio

    @property
    def raio(self):
        return self.__raio

    @raio.setter
    def raio(self, raio):
        self.__raio = raio

    def desenhe(self):
        return "circulo de raio {0:d}".format(self.__raio)
```

Implementação obrigatória dos métodos que eram abstratos na classe-pai

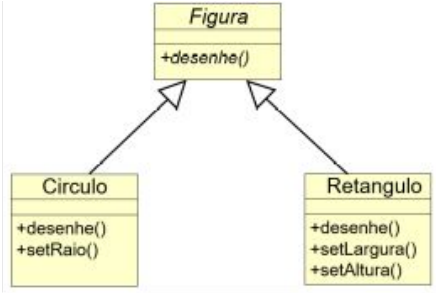
Classes Abstratas - Python

```
class Retangulo(Figura):
    def __init__(self, lado1=0, lado2=0):
        super().__init__()
        self.__lado1 = lado1
        self.__lado2 = lado2

    @property
    def lado1(self):
        return self.__lado1

    @lado1.setter
    def lado1(self, lado1):
        self.__lado1 = lado1

    ...

    def desenhe(self):
        return "retangulo com lados {0:d} e {1:d}".\
            format(self.__lado1, self.__lado2)
```

Implementação obrigatória dos métodos que eram abstratos na classe-pai

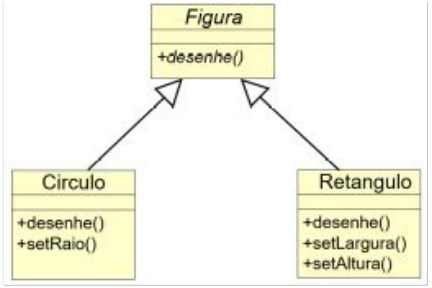
Classes Abstratas - Python

```
figuras = []

retangulo = Retangulo(1, 2)

circulo = Circulo(2)

figuras.append(retangulo)
figuras.append(circulo)

for figura in figuras:
    print(figura.desenhe())
```



Classes Abstratas - Python

```
figuras = []

retangulo = Retangulo(1, 2)

circulo = Circulo(2)

figuras.append(retangulo)
figuras.append(circulo)

for figura in figuras:
    print(figura.desenhe())
```



Polimorfismo: garantido pela herança de Figura

Classes Abstratas - Java

```
abstract class Shape {
    String name;
    abstract public double area();
    public String name() {
        return name;
    }

  //other stuff
}
class Square extends Shape {
    double length;
    public Square(double length) {
        this.length = length;
    }
    public double area() {
        return length*length;
    }
}
```

Output

```
The area of square s is 156.25
```

Referências

DATHAN, B.; RAMNATH, S. **Object-Oriented Analysis, Design and Implementation**. Cham: Springer, 2015

SEIDL, M. et al. **UML@ Classroom: An Introduction to Object-Oriented Modeling**. Cham: Springer, 2015.

Parte do material baseado no material do professor Jean Hauck