# Introdução a Orientação a Objetos

Prof. Dr. Alex Sandro Roschildt Pinto

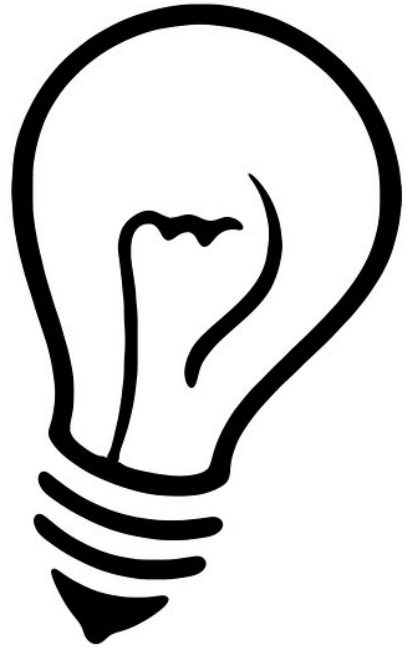
Mateus Manoel Pereira

# OO - Objetos, estado, operações

- Um **objeto** é uma entidade que:
  - encapsula informações de **estado** ou dados
  - possui um conjunto de **operações** associadas que manipulam estes dados.
- Uma **operação** é definida como sendo uma ação que um objeto realiza sobre outro para provocar uma reação.

# OO - Exemplo de Objeto

- **Objeto** Lâmpada
  - **Estado**
    - Status: Apagada;
    - Potência: 60 Watts.
- **Operações**
  - Acende - acende a lâmpada;
  - Apaga - apaga a lâmpada.



# OO - Mensagem

- Em geral, o estado de um objeto é completamente escondido e protegido de outros objetos e a única maneira de examiná-lo é através da invocação de uma operação (isto é, o envio de uma **mensagem**) para este fim.

# OO - Comportamento, identidade

- Objetos apresentam um **comportamento** bem definido e uma identidade que é única.
  - Comportamento define o modo como um objeto age e reage em termos das suas mudanças de estado e envio de mensagens e é completamente definido pelas suas operações.
- **Identidade** é a propriedade de um objeto que o distingue de outros objetos.

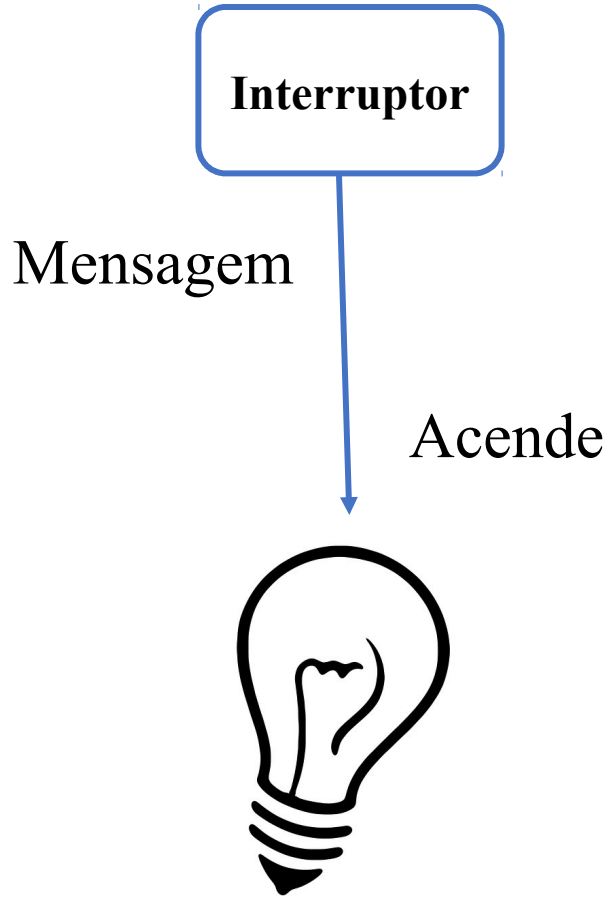
# OO - Comunicação

- Um objeto comunica-se com outro através de mensagens que identificam operações a serem realizados no objeto receptor da mensagem.

- O objeto responde a uma mensagem mudando possivelmente seu estado e/ou retornando um resultado.

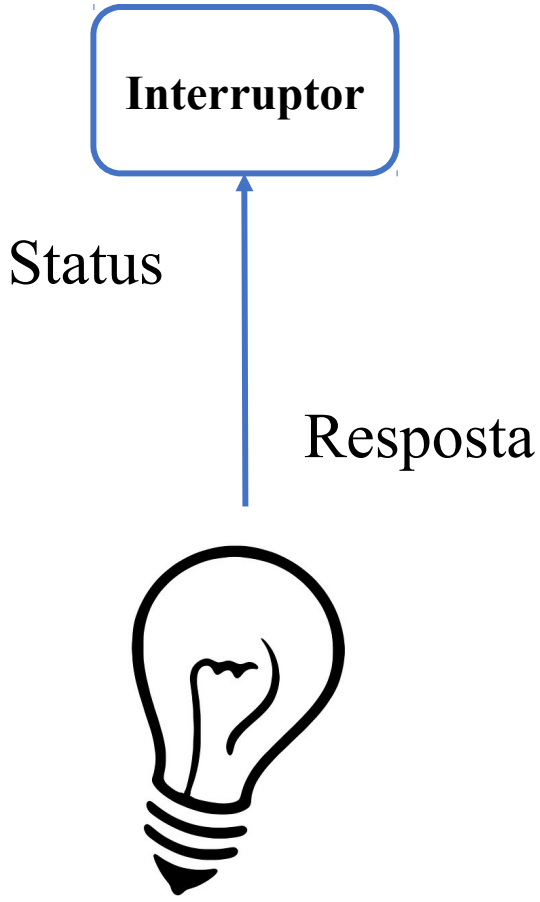
# OO - Exemplo de mensagem

- **Objeto** Lâmpada
  - **Estado**
    - Status: **Apagada**;
    - Potência: 60 Watts.
- **Operações**
  - Acende - acende a lâmpada;
  - Apaga - apaga a lâmpada.



# OO - Exemplo de mensagem

- **Objeto** Lâmpada
  - **Estado**
    - Status: **Acesa**;
    - Potência: 60 Watts.
- **Operações**
  - Acende - acende a lâmpada;
  - Apaga - apaga a lâmpada.



# OO - Variáveis e métodos

- No contexto de programação, um objeto é um conjunto de **variáveis** e **métodos** relacionados, que representam o estado do objeto e modificam este estado, respectivamente.

- Os métodos são implementações das operações.

# OO - Entidades reais/imaginárias

- Objetos em um software podem representar **entidades do mundo real** (lâmpada), ou representar **entidades imaginárias**, tais como um evento de pressionar o mouse no ambiente Windows.

- Tudo que um objeto de software sabe (estado) ou pode fazer (comportamento) é expressado pelas variáveis e métodos dentro daquele objeto.

# OO - Classes

- Uma classe é uma descrição de um molde que especifica as propriedades e o comportamento para um conjunto de objetos similares.

- **Todo objeto é instância de apenas uma classe.**

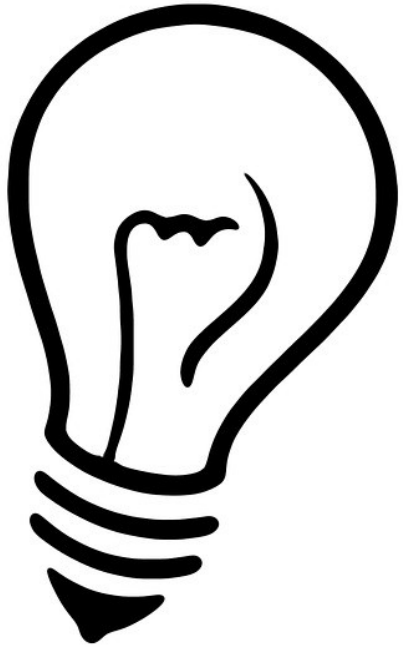
- Toda classe possui um nome e um corpo que define o conjunto de atributos e operações pertencentes às suas instâncias.

# OO - Classes

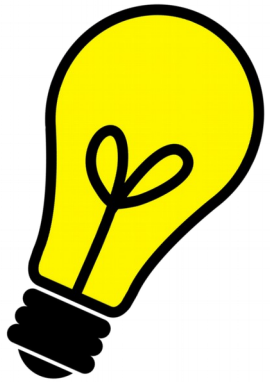
- É importante distinguirmos objetos de suas classes; o termo classe é usado para identificar um grupo de objetos e o termo objeto é usado para identificar uma instância particular de uma classe.
- No modelo de objetos, atributos e operações são parte da definição de uma classe.

# OO - Classes

- Classe Lâmpada
  - **Estado**
    - Status: String(Acesa/Apagada)
    - Potência: Inteiro
- **Operações**
  - Acende
    - Status = "Acesa"
  - Apaga
    - Status = "Apagada"



# OO - Instanciando Objetos de uma Classe



LampadaCozinha = Lampada()

Status = “Acesa”

Potência = 75

LampadadaSala = Lampada()

Status = “Apagada”

Potência = 100

# OO - Criando classes em Python

- Toda classe Python possui um nome (identificador único).
- Toda classe Python possui atributos. Cada **identificador estado** da classe é representado por um **atributo** dentro da especificação da Classe.

```
class Lampada:
```

# OO - Criando classes em Python

- Toda a operação da classe é representada por um **método** na especificação da classe Python.

```
class Lampada:

    #Métodos
    def apaga(self):
        self.status = "Apagada"
```

# OO - Construtores

- **Construtor é bloco de código que define o estado do objeto quando criado.** Operação que é executada quando o objeto é criado.

- Em Python o construtor é criado como um método chamado __init__.

# OO - Criando classes em Python

```
class Lampada:
        #Construtor
    def __init__(self):
        self.status = "Apagada"
        self.potencia = 60
```

```
#Métodos
def apaga(self):
    self.status = "Apagada"

def acende(self):
    self.status = "Acesa"
```

# OO - Destrutores

- **Destrutor é bloco de código que define o estado do objeto quando finalizado.** Operação que é executada quando o objeto é finalizado.

- Em Python o destrutor não precisa ser declarado, pois a linguagem tem um coletor de lixo (Garbage Collector) que cuida do gerenciamento da memória automaticamente.

# OO - Destrutores

```
class Lampada:

    #Construtor
    def __init__(self):
        self.status = "Apagada"
        self.potencia = 60

    #Métodos
    def apaga(self):
        self.status = "Apagada"

    def acende(self):
        self.status = "Acesa"
```

- Em Python um destrutor pode ser declarado definindo um método chamado __del__.

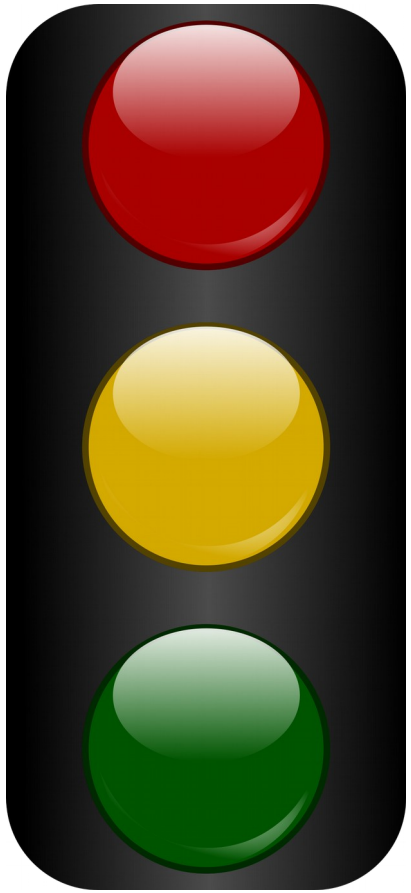
```
#Destrutor
def __del__(self):
    print("Lâmpada destruída")
```

- Para deletar um objeto explicitamente utiliza-se a palavra reservada **del**.
  - lampada = Lampada()
  - del lampada

# OO - Agregação/Composição

- Agregação é o ato de reutilizar o código definido em uma classe criando várias instâncias desta classe que serão componentes de objetos de outras classes.

# OO - Instanciando Objetos de uma Classe



LampadaVermelha = Lampada()

Status = “Acesa”

Potência = 200

LampadaAmarela = Lampada()

Status = “Apagada”

Potência = 200

LampadaVerde = Lampada()

Status = “Apagada”

Potência = 200

# OO - Instanciando Objetos de uma Classe

Classe Semáforo

**Estado**

LampadaVermelha = Lampada();
LampadaAmarela = Lampada();
LampadaVerde = Lampada();

**Operações**

Abre - Abre o semáforo

LampadaVermelha.Status = "Apagada"
LampadaAmarela.Status = "Apagada"
LampadaVerde.Status = "Acesa"

Fecha - Fecha o semáforo

LampadaVermelha.Status = "Acesa"
LampadaAmarela.Status = "Apagada"
LampadaVerde.Status = "Apagada"

# OO - Métodos em Python

- Em Python, um método é declarado com o primeiro argumento representando o objeto que o chama.

```
def modoLuzFraca(meuObjeto):
    meuObjeto.status = "Acesa - Fraca"
```

- Sem esse argumento o método pode ser chamado mesmo sem um instância da classe ter sido criada. Isso não é errado, mas deve-se pensar se realmente é o que problema demanda.

# OO - Métodos em Python

- def modoLuzFraca():
  - Sem o parâmetro **meuObjeto**, não é possível saber qual objeto está chamando o modoLuzFraca(), assim mudar o status do objeto torna-se impossível.

- def quantidadeLampada():
  - Nesse caso o parâmetro **meuObjeto** se vê desnecessário, pois não faz sentido um objeto específico conhecer a quantidade total de Lâmpadas, mas sim a classe em si.

# OO - Métodos em Python

- Um método pode retornar ou não um valor, para retornar utiliza-se a palavra reservada **return.**

- Ex.:

```
def verStatus(self):
    return self.status
```

# OO - Métodos em Python

- Além do parâmetro que representa o objeto chamador, um método pode receber parâmetros (opcionais) na sua chamada.

  - Ex.:

```
def alteraPotencia(self, potencia)
def calculaPotencia(self, amp, volts)
```

# OO - Métodos em Python

- Diferente de outras linguagens, Python não pertime a sobrecarga de métodos e funções.
- Em caso de métodos/funções com mesmo nome, a última definição será utilizada.

```
def alteraPotencia(self, potencia):
    self.potencia = potencia
def alteraPotencia(self, amp, volts):
    self.potencia = amp*volts
```

Nesse caso **alteraPotencia(self, amp, volts)** seria chamado.

# OO - Classificação, generalização, especialização

- Lista de animais
  - Canário
  - Homem
  - Cobra
  - Sapo-dourado
  - Elefante
  - Mosquito
  - Sabiá
  - Lagarto
  - Borboleta
  - Píton

# OO - Classificação

- Classificar é o ato de categorizar os objetos em classes de acordo com suas características.

- Aves
  - Canário
  - Sabiá

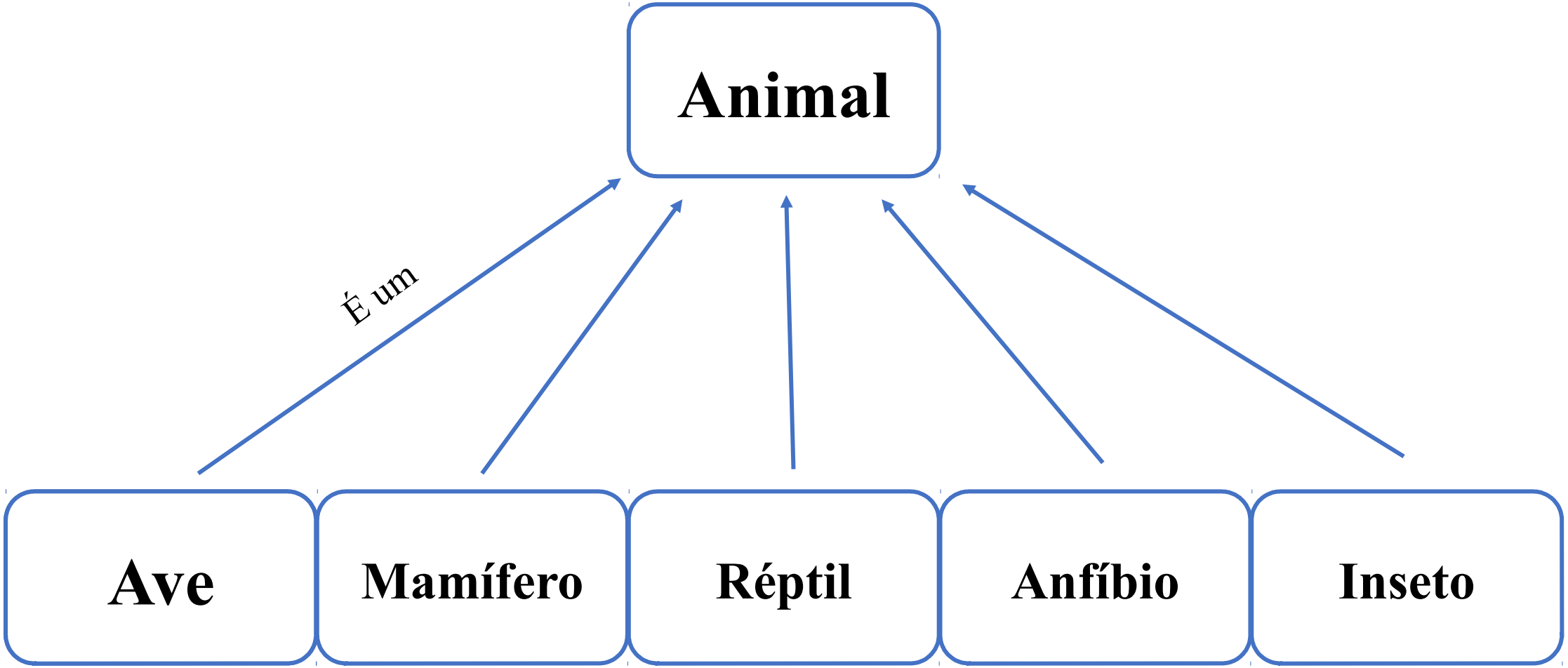
- Anfíbios
  - Sapo-dourado
  - Píton

- Mamíferos
  - Homem
  - Elefante

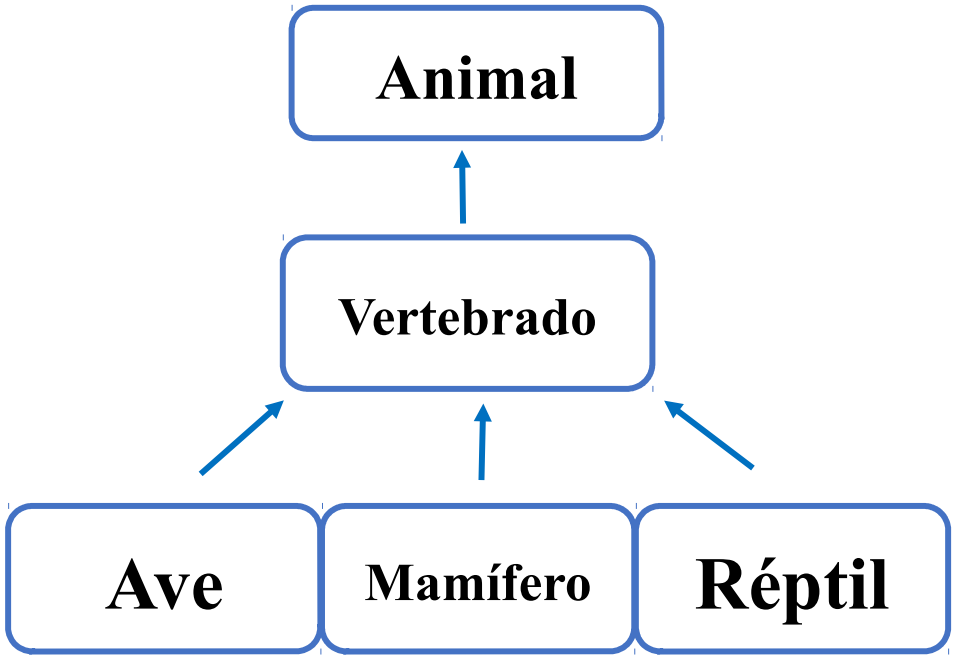
- Insetos
  - Borboleta
  - Mosquito

- Répteis
  - Cobra
  - Lagarto

# OO - Hierarquia de Classes



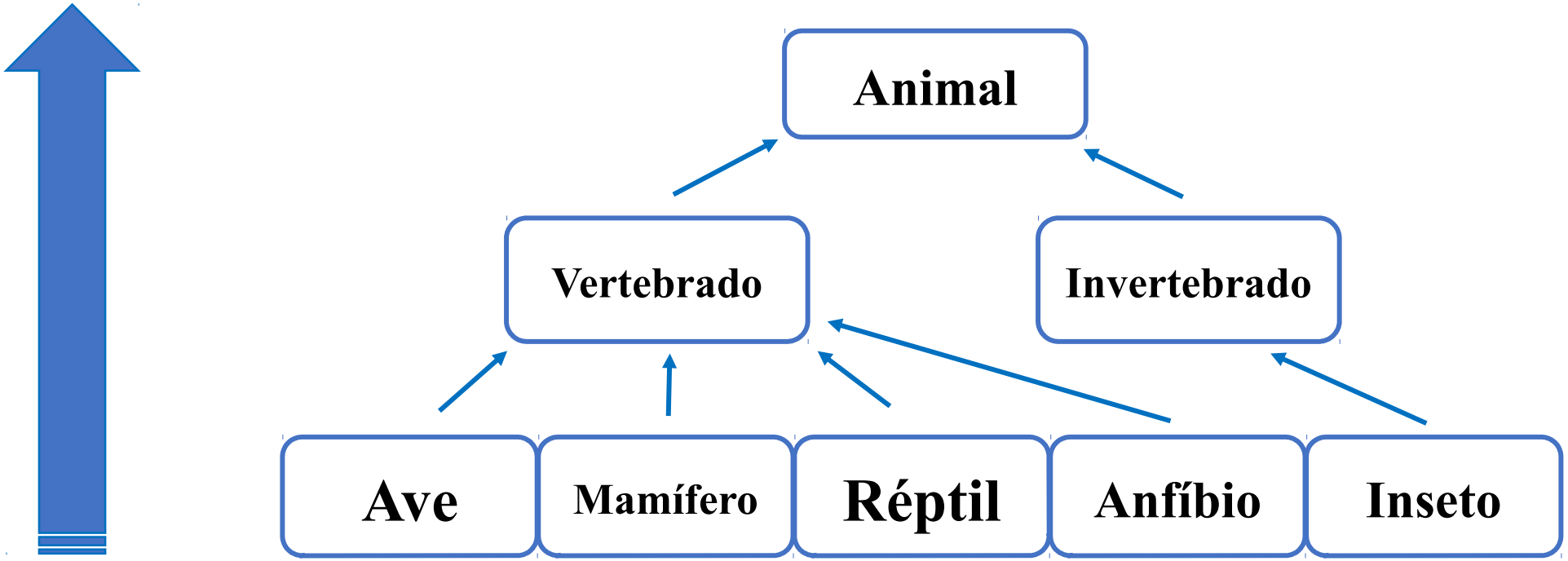
# OO - Subclasse e Superclasse




- **Subclasse**: classe especializada de uma superclasse.
- **Superclasse**: classe generalizada de uma subclasse.

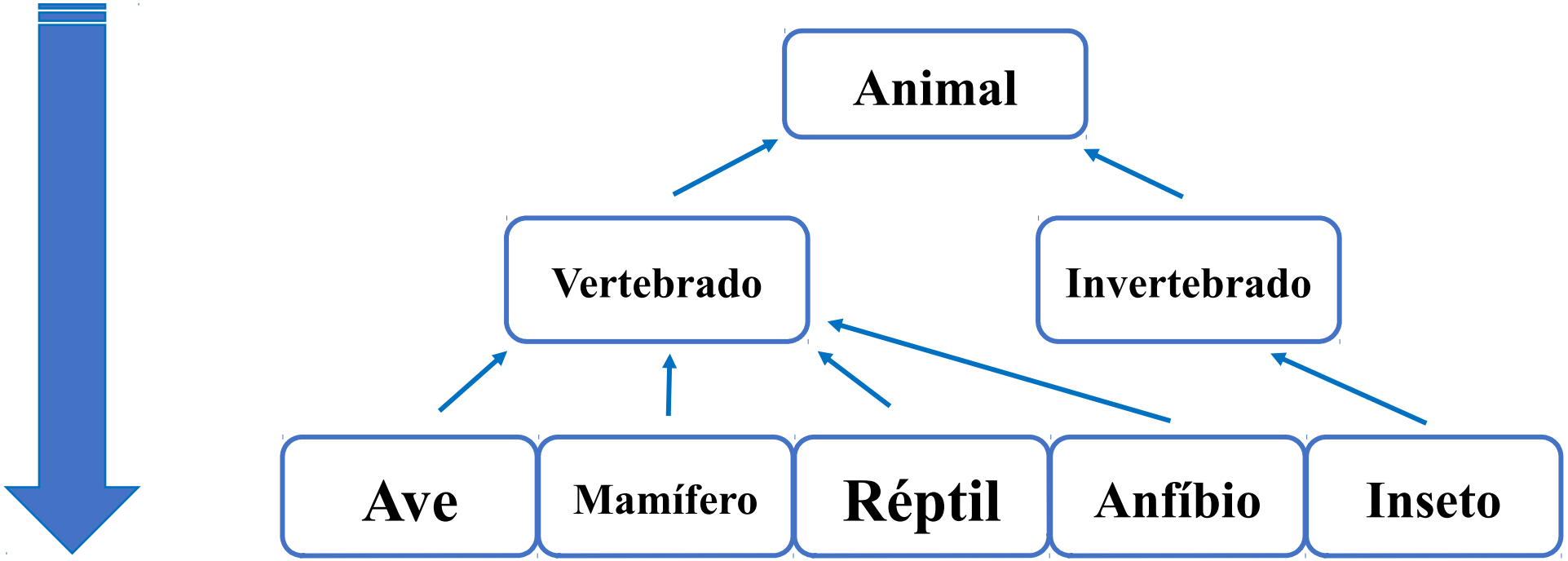
- Vertebrado é **subclasse** de classe Animal.
- Animal é **superclasse** da classe Vertebrado.

# OO - Generalização




- Generalizar é o ato de capturar características comuns entre as classes e defini-las em novas superclasses generalizadas (ex.: Vertebrado e Invertebrado).

# OO - Especialização




- Especializar é o ato de capturar as diferenças entre as classes e criar novas subclasses distintas com tais diferenças (ex.: Ave, Mamífero, etc.).

# OO - Herança

- **Conceito:** herança é a habilidade que permite a uma subclasse possuir todas as propriedades (atributos e métodos) de uma superclasse sem necessidade de explicitá-los novamente na definição desta subclasse.

- **Vantagens:**
  - Aumenta e facilita a reutilização de código;
  - Facilita a manutenção.

# OO - Herança – Exemplo (sem herança)


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

    #Métodos
    def alteraNome(self, nome):
        pass
    def retornaNome(self):
        pass
```

```
class PessoaFisica:
    #Construtor
    def __init__(self, nome):
        self.nome = nome

    #Métodos
    def alteraNome(self, nome):
        pass
    def retornaNome(self):
        pass
    def alteraCpf(self, cpf):
        pass
    def retornaCpf(self):
        pass
```

```
class PessoaJuridica:
    #Construtor
    def __init__(self, nome):
        self.nome = nome

    #Métodos
    def alteraNome(self, nome):
        pass
    def retornaNome(self):
        pass
    def alteraCnpj(self, cnpj):
        pass
    def retornaCnpj(self):
        pass
```

# OO - Herança – Exemplo (com herança)


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

    #Métodos
    def alteraNome(self, nome):
        pass
    def retornaNome(self):
        pass
```

```
class PessoaFisica(Pessoa):
    #Construtor
    def __init__(self, nome):
        super().__init__(nome)

    #Métodos
    def alteraCpf(self, cpf):
        pass
    def retornaCpf(self):
        pass
```

```
class PessoaJuridica(Pessoa):
    #Construtor
    def __init__(self, nome):
        super().__init__(nome)

    #Métodos
    def alteraCnpj(self, cnpj):
        pass
    def retornaCnpj(self):
        pass
```

# OO - Classes concretas e abstratas

- **Classe abstrata:** Classe a partir da qual nenhum objeto será instanciado. Classes abstratas geralmente tem métodos não implementados, para suas subclasses implementarem.

- **Classe concreta:** Classe a partir da qual os métodos definidos na classe abstrata terão implementação e os objetos serão instanciados.

# OO - Classes concretas e abstratas

```
from abc import ABC, abstractmethod


class Animal(ABC):
    #Construtor
    def __init__(self, nome):
        self.nome = nome

    #Métodos
    @abstractmethod
    def comunicar(self):
        pass
```

```
class Gato(Animal):
    #Construtor
    def __init__(self, nome):
        super().__init__(nome)

    #Métodos
    def comunicar(self):
        print("Meow!")
```

```
class Cachorro(Animal):
    #Construtor
    def __init__(self, nome):
        super().__init__(nome)

    #Métodos
    def comunicar(self):
        Print("Auau!")
```

# OO - Polimorfismo

- **Polimorfismo:** do grego “várias formas”

- Conceito: Habilidade de diferentes objetos executarem o método apropriado em resposta a mesma chamada de método.

- Vamos retomar ao exemplo dos animais...

# OO - Polimorfismo

**Obs.:** A classe Animal neste exempo não é abstrata, entratando isso não impede o polimorfismo de acontecer.

```
class Animal():
    #Construtor
    def __init__(self, nome):
        self.nome = nome

    #Métodos
    def comunicar(self):
        print("O animal disse:")
```

```
class Gato(Animal):
    #Construtor
    def __init__(self, nome):
        super().__init__(nome)

    #Métodos
    def comunicar(self):
        super().comunicar()
        print("Meow!")
```

```
class Cachorro(Animal):
    #Construtor
    def __init__(self, nome):
        super().__init__(nome)

    #Métodos
    def comunicar(self):
        super().comunicar()
        print("Auau!")
```

# OO - Atributos de classe (estático)

- São atributos que armazenam valores que devem ser os mesmos para todas os objetos que forem instanciados a partir desta classe.
- Geralmente utilizado para armazenar constantes ou a quantidade de objetos instanciados.

- Ex.: Em uma classe **FabricaRobo** podemos ter o atributo estático **quantidadeRobo**.

# OO - Atributos de classe (estático)

```
class FabricaRobo:
    #Atributos Estático
    quantidadeRobo = 0
    #Construtor
    def __init__(self, numSerie):
        self.numSerie = numSerie
        quantidadeRobo += 1

    #Métodos
    def retornaNumSerie(self):
        return self.numSerie
    def retornaQtdRobo(): #Note que não tem self
        return quantidadeRobo
```

- Do mesmo modo que temos atributos estático podemos ter métodos estático, como o método **retornaQtdRobo** no exemplo ao lado.

# OO - Tratamento de exceções

- Exceção: Condição anormal na execução de um programa.
- Exemplos:
  - Acesso a um índice inválido em um vetor;
  - Tentativa de uso de um objeto não inicializado;
  - Tentar abrir um arquivo inexistente;
  - Tratar um objeto com tipagem errada;
  - Qualquer outra falha não prevista.

- Vantagens:
  - Robustez, segurança e melhor estruturação.

# OO - Tratamento de exceções

- **try:**

  #Algo que pode gerar uma exceção

  - Pessoas[i] = 5
- **except** IndexError **as e**:

  #O que deve ser feito com a exceção gerada

  print ("Posição inválida na lista de Pessoas!")

  print("Exceção gerada: {}".format(**e**))

# OO - Tratamento de exceções

- Python é uma linguagem dinamicamente tipada, isso pode trazer diversos problemas quando uma solução requer um tipo de dado específico.
- Para solucionar isso pode-se usar um tratamento de exceção.

```
def tradutorAnimal(animal):
    try:
        animal.comunicar()
    except AttributeError as e:
        print("O objeto passado não se comunica com humanos!")
```

- Se o objeto passado no parâmetro for do tipo animal (ou de uma de suas subclasses) o programa executará normalmente, entretanto se o objeto passado não implementar o método comunicar uma exceção é gerada.

# Duck Typing

- Geralmente em linguagens dinamicamente tipadas utiliza-se o estilo de codificação conhecido como **Duck Typing**.

**- "Se anda como pato, nada como um pato e faz quack como um pato, então provavelmente é um pato"**

- Isso significa que é preferível simplesmente tentar executar um método e só se preocupar depois caso ele "não seja um pato".

# Duck Typing - Exemplo

- **try:**

  #Tenta executar o método quack( )

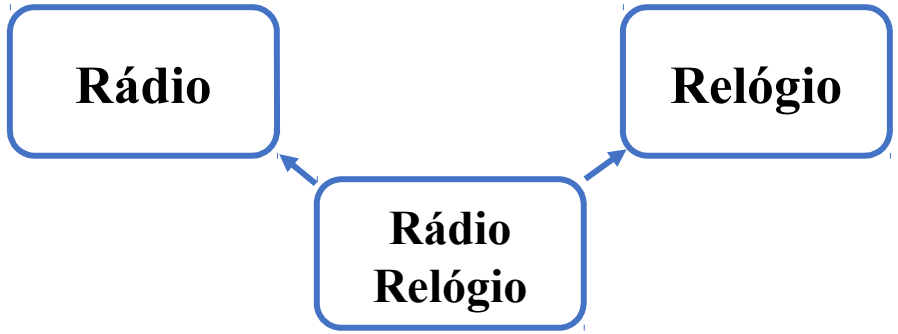
  pato.quack( )
- **except (AttributeError, TypeError):**

  #Caso o objeto pato não tenha o método quack( )

  print ("Objeto inválido!")

# OO - Herança múltipla

- Conceito: Capacidade de uma classe herdar as características de duas ou mais classes.



# OO - Herança múltipla

```
class Radio:
    #Construtor
    def __init__(self, estacao):
        self.estacao = estacao

    #Métodos
    def alteraEstacao(self, estacao):
        pass
```

```
class Relogio:
    #Construtor
    def __init__(self, hh, mm, ss):
        pass

    #Métodos
    def horario(self):
        pass
```

Python começa a procura de atributos e métodos nas superclasses da esquerda para direita, ou seja, no exemplo ao lado se o método alteraEstacao for chamado procura-se sua existência na classe RadioRelogio e depois nas superclasses Radio e Relogio nessa mesma ordem, caso não encontre uma exceção é gerada.

```
class RadioRelogio(Radio, Relogio):
    #Construtor
    def __init__(self, estacao, hh, mm, ss):
        Radio.__init__(estacao)
        Relogio.__init__(hh, mm, ss)
```