
BY

IME-USP

# Tec Prog 1

# Orientação a Objetos

Fabio Kon
IME-USP

# Nos primórdios havia o caos...

- Linguagem de máquina
- Linguagem de montagem
- Linguagem de alto nível (FORTRAN)
- Programação estruturada (ALGOL)
  - Muita repetição de código
  - Código mal organizado
  - Código de difícil entendimento
  - Muitos erros (*bugs*)

# Como resolver esses problemas?

- Programação Orientada a Objetos
  - 1967 a 1980
- Simula
  - 1962 a 1970
- Smalltalk
  - 1972 a 1980

- C++ (1990)
- Java (1995)
- Ruby
- Python

# Objeto

Algo que **encapsula**

- dados
  - variáveis
  - atributos
- código
  - funções
  - métodos

# Classe

- Define os elementos de um conjunto de objetos
  - quais os métodos
  - quais os atributos
- Funciona como uma fábrica de objetos

# Instanciação de objetos

- A partir de uma classe, podemos instanciar objetos
- criar instâncias das classes
  - Java/C#: `Carro c = new Carro();`
  - C++: `Carro *c = new Carro();`
  - Python: `c = Carro()`
  - Ruby: `c = Carro.new`
  - Smalltalk: `c ← Carro new`

# Comunicação entre objetos

- Envio de mensagem
- Chamada de método
- Chamada de função

  - como se comporta um sistema de objetos?

# Abstração

- Objetos representam
  - coisas do mundo real
  - do mundo virtual/computacional
- Aumentam o nível de abstração da programação

# Resumindo

- Objetos encapsulam dados e código
- Classes descrevem tipos de objetos e funcionam como fábricas
- Sistemas orientados a objetos são compostos por vários objetos que trocam mensagens entre si
- Um programa OO usa objetos para representar os elementos do seu domínio

# Na prática

- Vamos ver como fazer isso em Python

- Exemplo / Exercício:
  - Musica.py

# Melhoria na impressão

- Ninguém melhor do que o próprio objeto para definir como ele deve ser impresso.
- Um objeto pode definir como ele deve ser impresso usando:
  - def __str__(self) :

    return “*UmStringQueMeDescreve*”
  - para gerar esse string dinamicamente, pode-se usar interpolação de strings em Python. Também conhecido como formatação de Strings:
    - https://www.datacamp.com/community/tutorials/python-string-format

# De uma forma geral:

- **__repr__** é um método que define um string formal que pode ser usado para reconstruir o objeto usando o método eval() de python.
- **__str__** é um método que define um string informal para descrever um objeto para o usuário (humano).
- Exemplo:
  - `import datetime as dt`
  - `agora = dt.datetime.now()`
  - `str(now)`
  - `repr(now)`

# **Principal** característica de um bom Programador OO

- Escolha de bons nomes para
  - variáveis
  - atributos/propriedades
  - métodos e seus argumentos
  - classes

- *Intention-revealing names*
- Classes geralmente são substantivos
- Métodos geralmente são verbos

# Característica importante do projeto de Classes

- Em um sistema orientado a objetos, as classes devem possuir
  - uma única responsabildade.
  - *Single Responsibility Principle*
- Ou seja, uma classe não deve tratar de assuntos diferentes.
- Se uma classe trata de 2 ou 3 assuntos, veja se não dá para quebrá-la em 2 ou 3 classes menores.
  - conta corrente vs. pessoa vs. endereço

# no Musica.py

- Como melhorar o código para torná-lo mais Orientado a Objetos?

- Novo exercício para casa:
  - Introduzir classe Playlist
  - Nesta nova classe, permitir busca por qualquer tipo de atributo:
    - def busca_por (self, atributo, valor)
    - usando getattr(musica, atributo)

# Herança

- Subclasses em Simula-67
  (a primeira linguagem OO da história)
- **Relação “é-um”**
- Bom exemplos de Herança
  - Pessoas numa Universidade
  - Animal
  - Arquivos em um Sistema de Arquivos
- Exemplos de mau-uso de Herança
  - Carro e suas partes

# Herança terminologia

- Superclasse `Pessoa`
- Subclasses `Aluno`, `Professor`, `Funcionário`

- O ato de descer da superclasse para as subclasses é chamado de **Especialização**

- O ato de subir das subclasses para a superclasse é chamado de **Generalização**

# Quando usar Herança?

1 Para organizar as abstrações

2 Para acrescentar comportamento novo

- Novos métodos
- Novos atributos

3 Para alterar comportamento

- Novas implementações de métodos mais especializados

# Exemplo de Herança

```
class Poligono:
    def __init__(self, numero_de_lados):
        self.n = numero_de_lados
        self.lados = [0 for i in range(numero_de_lados)]

    def le_lados(self):
        self.lados = [float(input("Digite tamanho do lado " + str(i+1) + ": "))
                      for i in range(self.n)]

    def mostra_lados(self):
        for i in range(self.n):
            print("Lado", i+1, "is", self.lados[i])



class Triangulo(Poligono):
    def __init__(self):
        Poligono.__init__(self,3)

    def area(self):
        a, b, c = self.lados
        # calcula o semi-perímetro
        s = (a + b + c) / 2
        area = (s*(s-a)*(s-b)*(s-c)) ** 0.5
        print('A área do triângulo é %0.2f' %area)
```

```
class Retangulo(Poligono):
    def __init__(self):
        Poligono.__init__(self, 4)

    def le_lados(self):
        lado1 = float(input("Digite tamanho do lado 1: "))
        lado2 = float(input("Digite tamanho do lado 2: "))
        self.lados[0] = self.lados[2] = lado1
        self.lados[1] = self.lados[3] = lado2

    def area(self):
        return self.lados[0] * self.lados[1]

    def diagonal(self):
        return (self.lados[0]**2 + self.lados[1]**2)** 0.5

class TrianguloRetangulo(Triangulo):
    def éTrianguloRetangulo(self):
        return(self.lados[0]**2 == self.lados[1]**2 + self.lados[2]**2
               or self.lados[1]**2 == self.lados[0]**2 + self.lados[2]**2
               or self.lados[2]**2 == self.lados[0]**2 + self.lados[1]**2)
```

# Polimorfismo

- Um objeto que adquire várias formas
- É o que torna a OO tão poderosa
- Exemplos:
  - Pássaros
  - Visualizador de Arquivos
  - Figuras Geométricas
  - Cálculo de pagamento

```
class Empregado:
    def __init__(self, nome, CPF, RG):
        self.nome = nome
        self.CPF = CPF
        self.RG = RG


class EmpregadoHorista(Empregado):
    def __init__(self, nome, CPF, RG, horasTrabalhadas, pagamentoPorHora):
        Empregado.__init__(self, nome, CPF, RG)
        self.horasTrabalhadas = horasTrabalhadas
        self.pagamentoPorHora = pagamentoPorHora

    def pagamento(self):
        return self.horasTrabalhadas * self.pagamentoPorHora

class EmpregadoCLT(Empregado):
    def __init__(self, nome, CPF, RG, salario):
        Empregado.__init__(self, nome, CPF, RG)
        self.salario = salario

    def pagamento(self):
        return 13.3 * self.salario


class PrestadorDeServico(Empregado):
    def __init__(self, nome, CPF, RG, pagamentoAvulso):
        Empregado.__init__(self, nome, CPF, RG)
        self.pagamentoAvulso = pagamentoAvulso

    def pagamento(self):
        return self.pagamentoAvulso
```

# Polimorfismo

- Graças ao polimorfismo, um código bastante enxuto pode ser muito poderoso.
- Sem OO, precisamos de vários comandos **if** ou **switch** para ter o mesmo efeito
  - Imagine como seria o código abaixo sem OO:

```
for e in empregados:
    custoTotal += e.pagamento()
```

# Conceitos Fundamentais de OO

- Objetos
- Classes
- Abstração
- variáveis de instância vs. variáveis de classe
- Troca de mensagens/chamada de método
- Nomes que revelam intenção
- Princípio da Responsabilidade Única
- Herança
- Polimorfismo

# Introdução à Programação Orientada a Objetos

IME-USP


BY