Programação Orientada a Objetos

Prof. Márcio Miguel Gomes

Sobrecarga de Métodos

- Sobrecarga de métodos é um recurso que permite implementar **funcionalidades diferentes** para métodos com o **mesmo nome**, desde que tenham **lista de parâmetros distintas**

- Em linguagens **fortemente tipadas**, como C++, C# e Java, **não** se pode **repetir** a mesma ordem dos tipos de dados dos parâmetros para métodos com o mesmo nome

- No Python, uma linguagem com **tipagem dinâmica**, a sobrecarga ocorre ao definirmos **valores padrão** para determinados parâmetros

Sobrecarga de Métodos

```
// C++
// Método para cálculo de área
int CalculaArea();

// Sobrecarga com parâmetros distintos
int CalculaArea(int base, int altura);

// Sobrecarga com parâmetros distintos
float CalculaArea(float base, float altura);

-----------------------------------------------------------

# Python
# Sobrecarga com parâmetros padrão
def calcula_area(base=1, altura=1)
```

Construtor

- Construtor é um método especial, público, **executado automaticamente** quando um objeto é **instanciado**
- É utilizado para inicialização do objeto, definição de valores padrão e garantia da consistência do objeto
- Um construtor pode receber nenhum, um ou vários parâmetros
- Em C++, C# e Java, ele possui o **mesmo nome da classe**, e não possui tipo de dado de retorno
- Em Python, o construtor é definido como **__init__(self)**

Construtor

```cpp
// C++
class Pessoa
{
public:
    Pessoa(string nome, char sexo) // Construtor do objeto
    {
        this->nome = nome;
        this->sexo = sexo;
    }
}
```

```python
# Python
class Pessoa:
    def __init__(self, nome, sexo): # Construtor do objeto
        self._nome = nome
        self._sexo = sexo
```

Destrutor

- Destrutor é um método especial, público, **executado automaticamente** quando um objeto é **liberado da memória**
- É utilizado para finalização do objeto e liberação de recursos
- Um destrutor não pode receber nenhum parâmetro
- Em C++, C# e Java, ele é identificado por um **til (~)** seguido do **nome da classe**, e não possui tipo de dado de retorno
- Em Python, o destrutor é definido como **__del__(self)**

Destrutor

```cpp
// C++
class Pessoa
{
public:
    ~Pessoa() // Destrutor do objeto
    {
        cout << "Liberando " << this->nome << endl;
    }
}
```

```python
# Python
class Pessoa:
    def __del__(self): # Destrutor do objeto
        print('Liberando', self._nome)
```

Métodos Setter

- Métodos “**setter**” são utilizados para **definir o valor** dos **atributos privados** de um objeto

- Normalmente, esses métodos possuem como nome o **prefixo “set”** seguido do **nome do atributo privado**

- Esses métodos podem conter **regras** para **permitir** ou **negar escrita** nos dados privados, ou mesmo **transformá-los** por questões de padronização, facilidade de uso ou segurança

Métodos Getter

- Métodos “**getter**” são utilizados para **retornar o valor** dos **atributos privados** de um objeto

- Normalmente, esses métodos possuem como nome o **prefixo “get”** seguido do **nome do atributo privado**

- Esses métodos podem conter **regras** para **permitir ou negar acesso** aos dados privados, ou mesmo **transformá-los** para facilidade de cálculos ou melhoria na apresentação visual

Exemplo

```
class Pessoa():
    def __init__(self):
        self._nome = 'Desconhecido'
        self._sexo = '?'

    def set_nome(self, nome):
        if len(nome) < 3:
            print('Nome deve conter pelo menos 3 letras')
        else:
            self._nome = nome

    def set_sexo(self, sexo):
        sexo = sexo.upper()
        if sexo in 'MF':
            self._sexo = sexo
        else:
            print("Sexo deve ser 'M' ou 'F'")

    def get_nome(self):
        return self._nome

    def get_sexo(self):
        return self._sexo
```

```
    def get_sexo_extenso(self):
        if self._sexo == 'F':
            return 'Feminino'
        elif self._sexo == 'M':
            return 'Masculino'
        else:
            return 'Indefinido'

if __name__ == '__main__':
    p = Pessoa()

    p.set_nome('A') # Nome curto demais
    print('Nome:', p.get_nome())

    p.set_nome('Ana') # Nome correto
    print('Nome:', p.get_nome())

    p.set_sexo('X') # Sexo desconhecido
    print('Sexo:', p.get_sexo_extenso())

    p.set_sexo('F') # Sexo válido
    print('Sexo:', p.get_sexo_extenso())
```

Atividade

- Implementar a classe Pessoa, conforme diagrama de classes abaixo:

Pessoa
- nome: string - sexo: char - cor_olhos: char - pai: Pessoa - mae: Pessoa
+ <construtor>(nome: string, sexo: char, cor_olhos: char, pai: Pessoa, mae: Pessoa) + gera_pessoa(nome: string, sexo: char, pai: Pessoa): Pessoa + set_sexo(sexo: char): void + set_cor_olhos(cor_olhos: char): void + get_nome(): string + get_sexo_str(): string + get_cor_olhos_str(): string + imprime_dados(): void

- O método construtor deve ser sobrecarregado, com e sem os parâmetros pai e mãe
- Os métodos set_sexo() e set_cor_olhos() devem permitir apenas parâmetros válidos: M/F e C/V/A
- Os métodos get_sexo_str() e get_cor_olhos_str() devem retornar os textos correspondentes aos atributos
- O método gera_pessoa() deve ser executado apenas por objetos do sexo feminino, garantir que o parâmetro pai seja masculino e retornar a nova pessoa gerada. Do contrário, mostrar mensagem na tela e retornar um objeto nulo (None)
- A cor dos olhos da nova pessoa segue a ordem de cores dominantes/recessivas dos pais. Castanho é dominante sobre verde e azul, e verde é dominante sobre azul
- O método imprime_dados() deve mostrar na tela o nome, sexo (extenso), cor dos olhos (extenso), nome do pai e nome da mãe da pessoa, se existirem
- No “main”, instanciar de forma dinâmica pessoas pai, mãe e filhos e mostrar seus dados