04

Encapsulamento

Transcrição

Continuaremos a falar sobre encapsulamento, um dos conceitos fundamentais da programação Orientada a Objeto. Anteriormente, vimos como proteger os atributos da classe `Conta`, deixando-os privados.

```
class Conta:

    def __init__(self, numero, titular, saldo, limite):
        print("Construindo objeto ... {}".format(self))
        self.__numero = numero
        self.__titular = titular
        self.__saldo = saldo
        self.__limite = limite
```

O Python não possui uma palavra-chave para tornar um atributo privado — como Java tem o modificador de visibilidade `private`. Porém, foi convencionada uma nomenclatura especial: os dois *underscore* (__). Quando `_` é utilizado, o atributo é renomeado pelo Python. Por exemplo, `__conta` passou a se chamar automaticamente `_Conta__saldo`. Desta forma, explicitamos para o desenvolvedor que se trata de um atributo privado.

No entanto, o assunto encapsulamento vai além dos atributos. Vamos exemplificar isso a seguir, com a criação de duas contas no console do PyCharm:

```
>>> conta = Conta(123, "Nico", 55.5, 1000.0)
Construindo objeto ... <conta.Conta object at 0x10681d588>
>>> conta2 = Conta(321, "Marco", 100.0, 1000.0)
Construindo objeto ... <conta.Conta object at 0x1065f0940>
```

Temos duas referências, cada uma apontando para um objeto diferente. Agora se quisermos transferir dinheiro da conta do Marco (`conta2`) para o Nico (`conta`), como a quantia de R$ `10.00` que iremos declara a seguir:

```
>>> valor = 10.00

>>> conta2.saca(valor)

>>> conta.deposita(valor)
```

Nós acessamos a conta do Marco para sacar e, depois, a conta do Nico para realizar o depósito. A ação de transferir dinheiro se baseia em tirar o dinheiro de uma conta e depositar em outra. Em seguida, verificaremos o saldo atualizado das duas contas.

```
>>> valor = 10.00
>>> conta2.saca(valor)
>>> conta.deposita(valor)
>>> conta.extrato()
Saldo de 65.5 do titular Nico
```

```
>>> conta2.extrato()
Saldo de 90.0 do titular Marco
```

Foi retirado `10.00` do saldo da `conta2`, enquanto o saldo da `conta` passou a ser `65.5`. A transferência foi bem-sucedida, porém, a ação não ficou clara. Nós programamos algo relacionado a nossa conta que deveria estar localizado dentro da classe `Conta`. A essência do OO é deixar o código **organizado**.

No entanto, implementamos a transferência fora da classe. Se quisermos transferir, é melhor deixar todo o código em um único lugar. Como essa operação está relacionada com a conta, iremos colocá-la na `Conta`. Temos um caso que quebra o encapsulamento, porque o comportamento "transferir" está no lugar equivocado. O próximo passo será movê-lo para a classe `Conta`, onde deveria estar, adicionando para o método `transfere()` logo abaixo de `saca()`.

> Sobre a nomenclatura do método, você tem a liberdade para adotar o nome do método com o verbo no infinitivo, adotando o nome `transferir`, desde que os demais métodos sigam o mesmo padrão.

Dentro do método `transfere()`, vamos passar dois parâmetros: `self` e `valor`, além disso, aproveitaremos o código executado no console para realizar a transferência.

```
def transfere(self, valor):
    conta2.saca(valor)
    conta.deposita(valor)
```

Renomearemos as referências:

- O parâmetro `conta2` estará relacionado com o parâmetro `origem`;
- Enquanto `conta` se relacionará com `destino`.

```
def transfere(self, valor):
    origem.saca(valor)
    destino.deposita(valor)
```

Porém, ainda não criamos as variáveis `origem` e `destino`. Teremos que declará-las dentro do método também.

```
def transfere(self, valor, origem, destino):
    origem.saca(valor)
    destino.deposita(valor)
```

Adiante refatoraremos o código, para aprimorá-lo. Agora, iremos testá-lo. No console, vamos adicionar os dados de duas contas

```
>>> conta = Conta(123, "Nico", 55.5, 1000.0)
Construindo objeto ... <conta.Conta object at 0x10521b128>
>>> conta2 = Conta(321, "Marco", 100.0, 1000.0)
Construindo objeto ... <conta.Conta object at 0x10521b390>
```

Em seguida, executaremos método `transfere()`, utilizando o nome das referências `conta` e `conta2`. Dentro do parênteses, passaremos os valores referentes aos parâmetros `self`, `valor`, `origem` e `destino`. Definiremos que

`conta2` é `origem`, enquanto `conta` será `destino`. O valor do `self` não precisa ser incluído.

```
>>> conta2.transfere(10.0, conta2, conta)
>>> conta2.extrato()
Saldo de 90.0 do titular Marco
```

Para termos um retorno, executamos o método `extrato()`, desta forma, teremos acesso ao saldo de `conta2` atualizado: `90.0`.

```
>>> conta2.transfere(10.0, conta2, conta)
>>> conta2.extrato()
Saldo de 90.0 do titular Marco
>>> conta.extrato()
Saldo de 65.5 do titular Nico
```

A quantia que foi retirada de uma conta foi adicionada em outra. Nós conseguimos criar o código do método que está funcionando bem, mas podemos melhorá-lo ainda. Se observarmos o trecho de código, a referência `conta2` aparece duas vezes na linha executada. Porém, se compreendermos com quem cada parâmetro se relaciona, perceberemos que tanto `self` quanto `origem` são equivalentes `conta2`. Como o Python adiciona `self` automaticamente, removeremos o parâmetro `origem` e usaremos `self` como referência antes de `saca()`. A partir do `self`, além de acessarmos um atributo, poderemos executar um método também.

Ao digitarmos `self.` veremos que o *autocomplete* disponilizará todos os métodos, assim como os atributos. No caso, executaremos o método `saca()`.

```
def transfere(self, valor, destino):
    self.saca(valor)
    destino.deposita(valor)
```

Chamamos um método utilizando o `self`, em seguida, testaremos o código. Agora, `conta2` não será usada como referência equivalente ao parâmetro `origem`. No console, executaremos a seguinte linha:

```
>>> conta2.transfere(10.0, conta)
```

Da `conta2`, vamos transferir `10.0` para `conta` — seria o significado da frase escrita com a sintaxe do Python. Imprimiremos o extrato de `conta2` e veremos se o saldo foi atualizado.

```
>>> conta2.transfere(10.0, conta)
>>> conta2.extrato()
Saldo de 90.0 do titular Marco
```

Conseguimos deixar a nossa intenção de realizar uma transferência por meio do método `transfere()`. Nós encapsulamos o código, que foi adicionado na classe correta.