Encapsulamento

CIÊNCIA DA COMPUTAÇÃO

ATITUS EDUCAÇÃO

- Aula 04 - Organização e Abstração

Prof. Me. Lucas R. C. Pessutto

ATITUS EDUCAÇÃO

Na aula de hoje...

Encapsulamento

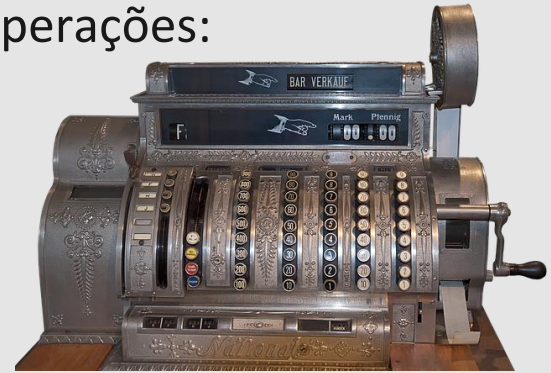
Conceito
Formas de Encapsular Atributos em Python
Slots
Encapsulamento na UML

Problema: Caixa Registradora

* Dispositivo que permite que um vendedor registre uma venda, informando uma sequência de itens que foram comprados

* Seu display exibe o total a ser pago e o número de itens daquela compra

* Uma caixa registradora deve realizar as seguintes operações:
  - → Add o preço de um item e a quantidade
  - → Finalizar uma compra (zerar as variáveis), para que uma nova compra possa ser registrada.



Problema: Caixa Registradora

```
class CaixaRegistradora:
    def __init__(self):
        self.total_venda = 0
        self.numero_itens = 0

    def add_item(self, preco, quantidade):
        self.total_venda += preco * quantidade
        self.numero_itens += quantidade

    def finalizar_venda(self):
        print("Obrigado por comprar conosco!")
        print(f"Foram comprados {self.numero_itens} itens")
        print(f"TOTAL: R${self.total_venda:.2f}")
        print("=" * 50)

        self.total_venda = 0
        self.numero_itens = 0
```

Problema: Caixa Registradora

```
## Programa Principal
mercado = CaixaRegistradora()

mercado.add_item(1.5, 2)
mercado.add_item(15.2, 1)
mercado.add_item(39.4, 6)

mercado.finalizar_venda()
```

1.5 * 2 = 3

15 .2* 1 = 15.2

39.4 * 6 = 236.4


```
Obrigado por comprar conosco!
Foram comprados 9 itens
TOTAL: R$254.60
==============================
```

Problema: Caixa Registradora

```python
## Programa Principal
mercado = CaixaRegistradora()

mercado.add_item(1.5, 2)
mercado.add_item(15.2, 1)
mercado.add_item(39.4, 6)

mercado.total_venda  = -0.5
mercado.numero_itens = -10

mercado.finalizar_venda()
```


```
Obrigado por comprar conosco!
Foram comprados -10 itens
TOTAL: R$-0.50
==============================
```

Interface Pública

* Toda classe possui uma interface pública: uma coleção de métodos através dos quais os objetos daquela classe podem ser manipulados.

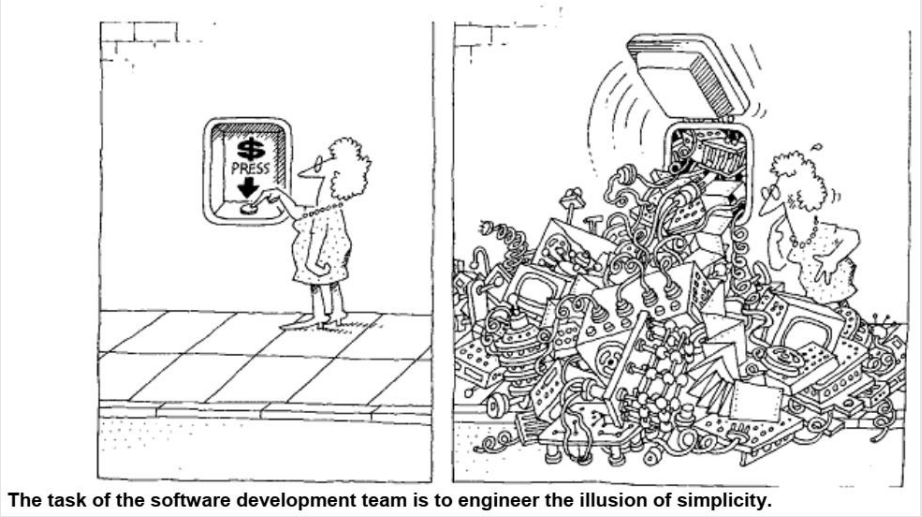
* Ao utilizar uma classe, não é necessário saber como o objeto armazena dados ou como os métodos foram implementados.

* O processo de prover uma interface pública adequada, escondendo os detalhes de implementação é chamado encapsulamento.

Encapsulamento

* Encapsulamento é o ato de fornecer uma interface pública para uma classe, escondendo os detalhes de implementação

* O Encapsulamento permite que as mudanças na implementação de um método não afetem os usuários de uma classe



The task of the software development team is to engineer the illusion of simplicity.

Modificadores de Acesso

* Linguagens como Java e C# utilizam modificadores de acesso para restringir o acesso a membros privados de uma classe

```
public class CaixaRegistradora {
    private double totalVendido;
    private int numeroItens;

    public CaixaRegistradoraEncapsulada(){
        this.totalVendido = 0;
        this.numeroItens = 0;
    }

    public void addItem(double preco, int quantidade) {
        this.totalVendido += preco * quantidade;
        this.numeroItens += quantidade;
    }
}
```

Modificadores de Acesso

* No Python, inserimos dois underscores ao atributo para impedir que ele seja acessado de fora da classe

* Isso não vai garantir que ninguém possa acessar de fato o atributo, porque em Python não existem atributos realmente privados

* O atributo com dois underscores na frente pode ser acessado fazendo ‘_NomeDaClasse_NomeDoAtributo_’

Modificadores de Acesso

```python
class CaixaRegistradora:
    def __init__(self):
        self.__total_venda = 0
        self.__numero_itens = 0

    def add_item(self, preco, quantidade):
        self.__total_venda += preco * quantidade
        self.__numero_itens += quantidade

    def finalizar_venda(self):
        print("Obrigado por comprar conosco!")
        print(f"Foram comprados {self.__numero_itens} itens")
        print(f"TOTAL: R${self.__total_venda:.2f}")
        print("=" * 50)

        self.__total_venda = 0
        self.__numero_itens = 0
```

```python
## Programa Principal
mercado = CaixaRegistradora()

mercado.add_item(1.5, 2)
mercado.add_item(15.2, 1)
mercado.add_item(39.4, 6)

mercado.__total_venda = -0.5
mercado.__numero_itens = -10

mercado.finalizar_venda()
```

Não teve efeito!

```
Obrigado por comprar conosco!
Foram comprados 9 itens
TOTAL: R$254.60
==================================================
```

Modificadores de Acesso

```
## Programa Principal
mercado = CaixaRegistradora()

mercado.add_item(1.5, 2)
mercado.add_item(15.2, 1)
mercado.add_item(39.4, 6)

mercado._CaixaRegistradora__total_venda = -0.5
mercado._CaixaRegistradora__numero_itens = -10

mercado.finalizar_venda()
```

```
Obrigado por comprar conosco!
Foram comprados -10 itens
TOTAL: R$-0.50
==============================
```

Modificadores de Acesso

* Como nenhum atributo é de fato privado em Python, muitos programadores preferem substituir o uso de dois underscores por apenas um underscore

* Atributos iniciados com um underscore sinalizam que eles devem ser tratados como atributos privados

* Essa notação não tem significado algum para o interpretador, é apenas uma convenção de código que deve ser respeitada por outros programadores.

Modificadores de Acesso

```
class CaixaRegistradora:
    def __init__(self):
        self._total_venda = 0
        self._numero_itens = 0

    def add_item(self, preco, quantidade):
        self._total_venda += preco * quantidade
        self._numero_itens += quantidade

    def finalizar_venda(self):
        print("Obrigado por comprar conosco!")
        print(f"Foram comprados {self._numero_itens} itens")
        print(f"TOTAL: R${self._total_venda:.2f}")
        print("=" * 50)

        self._total_venda = 0
        self._numero_itens = 0
```

Encapsulamento

* Convenção geral para criação de classes (não é uma regra!):
  - → ATRIBUTOS devem ser privados
  - → MÉTODOS devem ser públicos

* A conversa entre dois objetos deve acontecer sempre por troca de mensagens

Encapsulamento – Modo “Java”

* E se for necessário acessar algum atributo privado?

* Para permitir o acesso aos atributos de uma maneira controlada, a prática mais comum é criar dois métodos
  - → Um que retorna o valor do atributo
  - → E outro que altera o valor do atributo
  - → O padrão para esses métodos é colocar a palavra `get` e `set` antes do nome do atributo

Encapsulamento – Modo “Java”

```
class CaixaRegistradora:
    def __init__(self):
        self._total_venda = 0
        self._numero_itens = 0

    def get_total_venda(self):
        return self._total_venda

    def set_total_venda(self, valor):
        self._total_venda = valor

    def get_numero_itens(self):
        return self._numero_itens

    def set_numero_itens(self, valor):
        self._numero_itens = valor
```

```
    def add_item(self, preco, quantidade):
        self._total_venda += preco * quantidade
        self._numero_itens += quantidade

    def finalizar_venda(self):
        print("Obrigado por comprar conosco!")
        print(f"Foram comprados {self.get_numero_itens()}")
        print(f"TOTAL: R${self.get_total_venda():.2f}")
        print("=" * 50)

        self.set_total_venda(0)
        self.set_numero_itens(0)
```

Encapsulamento – Modo “Python”

* Python oferece uma outra forma de realizar encapsulamento: que são **propriedades** (properties)

* Propriedades combinam a facilidade de acesso público aos atributos de uma classe com as regras do encapsulamento

* **Método getter**: escrever um método com o nome idêntico ao atributo que está sendo encapsulado, decorado com `@property`
* **Método setter**: escrever um método com o nome idêntico ao atributo que está sendo encapsulado, decorado com `@<atributo>.setter`

* Decorar significa colocar essa linha diretamente acima da declaração do método

```python
class CaixaRegistradora:
    def __init__(self):
        self._total_venda = 0
        self._numero_itens = 0

    @property
    def total_venda(self):
        return self._total_venda

    @total_venda.setter
    def total_venda(self, valor):
        print("Proibido alterar o total venda!")

    @property
    def numero_itens(self):
        return self._numero_itens

    @numero_itens.setter
    def numero_itens(self, valor):
        print("Proibido alterar o numero itens!")
```

```python
## Programa Principal
mercado = CaixaRegistradora()

mercado.add_item(1.5, 5)

print(mercado.total_venda)
print(mercado.numero_itens)
mercado.total_venda = -0.5
mercado.numero_itens = -10
```

```
7.5
5
Proibido alterar o total venda!
Proibido alterar o numero itens!
```

Nesse caso não haveria necessidade de implementar os métodos setter, já que os atributos da classe só podem ser alterados através dos métodos da classe.

Slots

```
mercado = CaixaRegistradora()
mercado.nome = "Meu Mercados"
mercado.abacaxi = "Temos abacaxi"
print(mercado.abacaxi)
```

* Podemos impedir que os usuários de uma classe criem novos atributos em tempo de execução

* Existe uma variável embutida em cada classe chamada `__slots__`, que guarda uma lista com os atributos definidos para uma classe

Slots

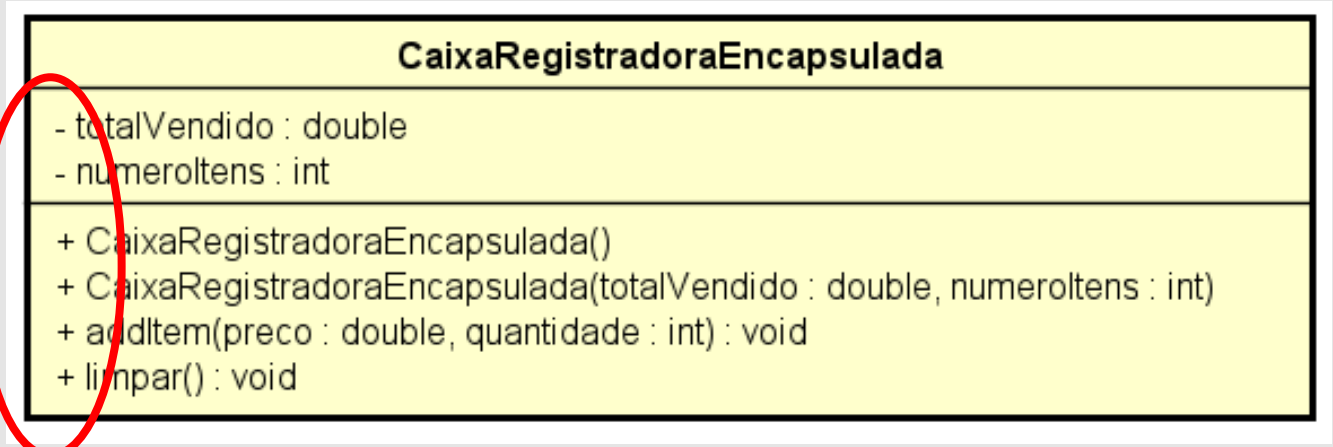
```
class CaixaRegistradora:

    __slots__ = ['_total_venda', '_numero_itens']

    def __init__(self):
        self._total_venda = 0
        self._numero_itens = 0


mercado = CaixaRegistradora()
mercado.nome = "Meu Mercado"
mercado.abacaxi = "Temos abacaxi"
print(mercado.abacaxi)
```

```
AttributeError: 'CaixaRegistradora'
object has no attribute 'nome'
```

Modificadores de Acesso na UML

Modificador	Representação
public	+
private	-
protected	#
package	~

Modificadores de Acesso na UML



Exercício - Contas Bancárias OO

* Refaça o exercício da primeira aula de maneira orientada a objetos!
* Primeiro Passo: Planejamento das Classes Cliente e Conta
* Planeje as classes de modo que elas sigam os princípios do encapsulamento vistos em aula.
* Implemente os métodos getters e setters somente para os atributos que podem ser lidos/modificados.