



Paradigma Orientado a Objetos

Encapsulamento e Visibilidade



Encapsulamento e visibilidade

Na programação Orientada a Objetos, é desejável e, muitas vezes, muito importante, que os atributos dos objetos tenham o devido nível e forma de acesso externo ao objeto.

- Para isso, é necessário definir a visibilidade dos atributos e métodos de um objeto.
- Como 'dono' dos atributos, um objeto é o mais indicado para lidar com seus atributos e métodos e não o cenário externo, como outros objetos.
- O encapsulamento permite maior controle e validação dos dados de um objeto.



Encapsulamento

- ◇ É importante evitar que atributos de uma classe sejam diretamente acessíveis de fora da classe

```
class Conta:  
    def __init__(self):  
        self.saldo = 0
```

- ◇ Para acessar esses atributos, métodos são definidos
 - permitem maior controle dos valores, como validação dos dados

```
c1 = Conta()  
c1.saldo = 100000000
```

Encapsulamento e visibilidade

- ◇ É importante evitar que atributos de uma classe sejam diretamente acessíveis de fora da classe
- ◇ Para acessar esses atributos, métodos são definidos
 - permitem maior controle dos valores, como validação dos dados

```
class Conta:  
    def __init__(self):  
        self.saldo = 0
```


```
c1 = Conta()  
c1.saldo = 100000000
```



Encapsulamento

O atributo saldo, de um objeto da classe Conta poderá ter esse valor acessado externamente. O que pode levar a erros na validação dos dados. Como evitar isso?

```
class Conta:  
    def __init__(self):  
        self.saldo = 0
```



```
c1 = Conta()  
c1.saldo = 100000000
```



Visibilidade

A visibilidade é utilizada para indicar o nível de acesso de um determinado atributo ou método;

- Os três modos distintos são:
 - Público;
 - Privado;
 - Protegido;





Visibilidade

A visibilidade é utilizada para indicar o nível de acesso de um determinado atributo ou método;

- Três modos distintos:

- Público
- Privado
- Protegido

Objetos de quaisquer classes podem ter acesso a atributos, ou métodos, públicos;

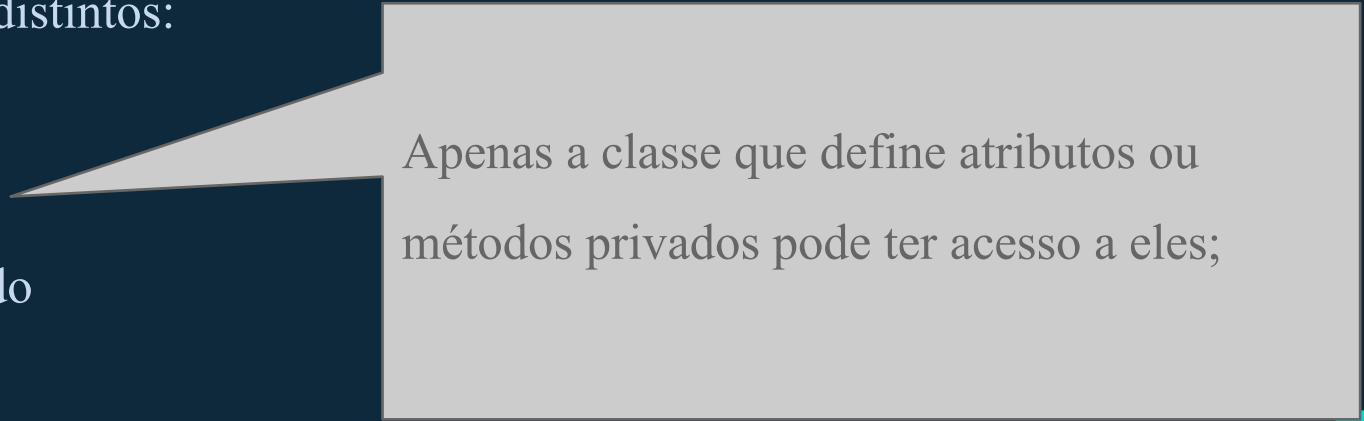




Visibilidade

A visibilidade é utilizada para indicar o nível de acesso de um determinado atributo ou método;

- Três modos distintos:
 - Público
 - Privado
 - Protegido



Apenas a classe que define atributos ou métodos privados pode ter acesso a eles;





Visibilidade

A visibilidade é utilizada para indicar o nível de acesso de um determinado atributo ou método;

- Três modos distintos:
 - Público
 - Privado
 - Protegido

Apenas a classe e suas subclasses podem ter acesso a atributos e métodos protegidos;



Atributos privados em Python

A definição de um atributo privado é feita com o uso de dois *underscores* antes do nome do atributo. Equivale a **private** em Java e C++

Assim, um atributo privado não pode ser acessado (teoricamente) diretamente, de fora do objeto

```
class Conta:
```

```
    __slots__ = ['__saldo']
```

```
    def __init__(self):
```

```
        self.__saldo = 0
```

```
c1 = Conta()
```

```
c1.saldo = 100000000 # erro
```





Visibilidade de atributos

```
class A():
```

```
    def __init__(self):
```

```
        self.__priv = "I am private"
```

```
        self._prot = "I am protected"
```

```
        self.pub = "I am public"
```

Dois *underscores* indicam atributo privado. Um *underscore*, protegido. Nenhum, público



Definindo e limitando atributos

O uso de `__slots__` faz a restrição dos atributos do objeto, não permitindo a criação de novos atributos

```
class Conta:
```

```
    __slots__ = ('_Conta__saldo')
```

```
    def __init__(self):
```

```
        self.__saldo = 0
```

```
    def set_saldo(self, saldo):
```

```
        if saldo < 0:
```

```
            raise Exception('Error')
```

```
        self.__saldo = saldo
```

Não é permitida a criação de novos atributos nos objetos da classe `Conta`

```
c1 = Conta()
```

```
c1.new_saldo = 100000000 # Erro
```



Definindo e limitando atributos

O uso de `__slots__` faz a restrição dos atributos do objeto, não permitindo a criação de novos atributos

É uma forma de acessar atributos privados! Python é uma linguagem muito dinâmica, 'pra gente adulta que sabe o que faz'. Mas tem como evitar?

```
class Conta:
```

```
    __slots__ = ('_Conta__saldo')
```

```
    def __init__(self):
```

```
        self.__saldo = 0
```

```
    def set_saldo(self, saldo):
```

```
        if saldo < 0:
```

```
            raise Exception('Error')
```

```
        self.__saldo = saldo
```

```
c1 = Conta()
```

```
c1.new_saldo = 100000000 # Erro
```

```
c1._Conta__saldo = -100000000 # permitido
```





Getters e Setters

- ◇ São métodos específicos para acesso aos atributos de uma classe, principalmente os atributos privados
- ◇ Como padrão na comunidade de programadores, são nomeados com os prefixos 'set_' ou 'get_' para ajustar ou obter os valores dos atributos.
- ◇ Permitem validação e formatação dos valores dos atributos antes de serem acessados ou alterados fora do objeto
- ◇ São métodos, geralmente, públicos





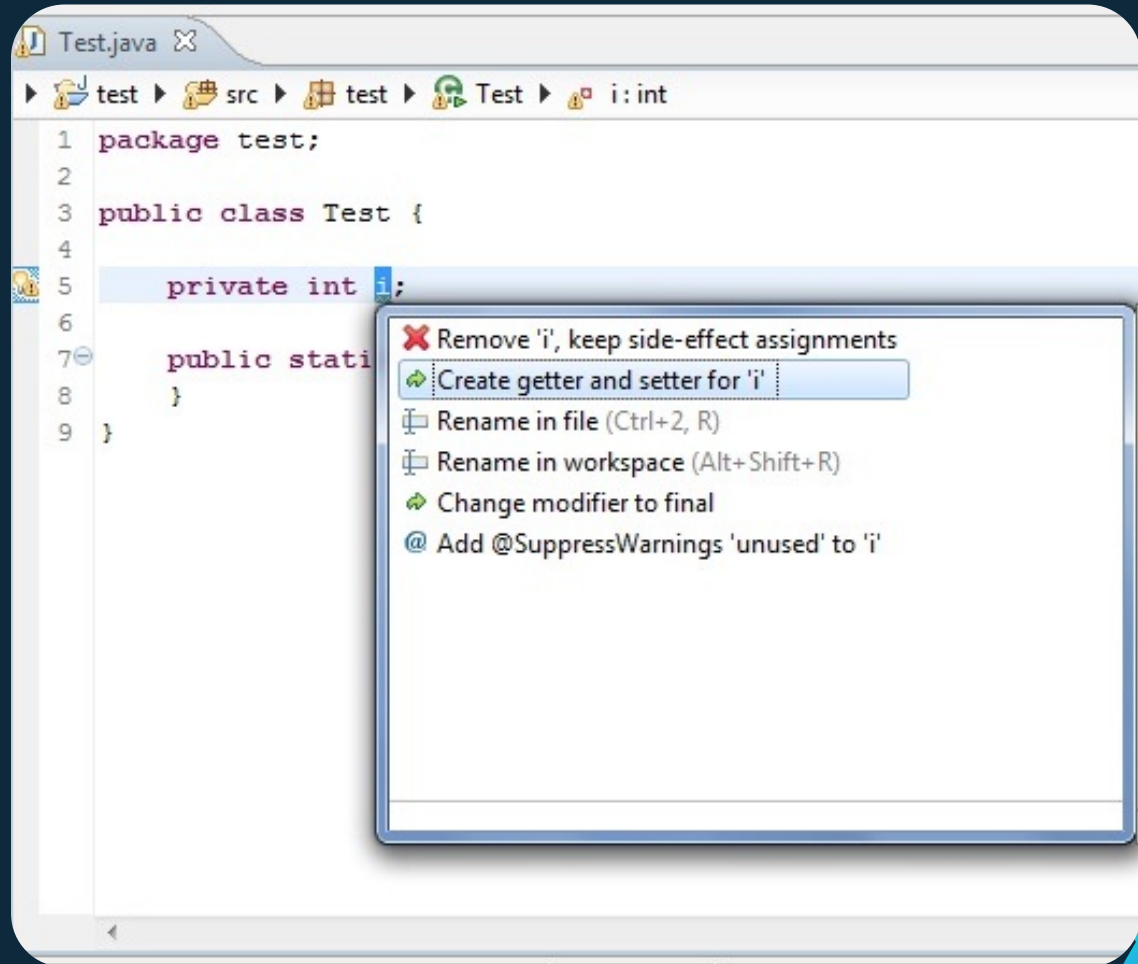
Troca de Mensagens

◊ Na Orientação a Objetos, os objetos interagem pela troca de mensagens, e, nesse contexto, os métodos *getters* e *setters* desempenham papel importante e frequente

Cada objeto sabe os atributos que têm e, portanto, têm métodos para altera-los adequadamente



Algumas IDEs
agilizam a
criação dos
métodos de
acesso aos
atributos. Para
Java, por
exemplo, no
Eclipse:





Padrões de Projeto de Software

- ◇ São soluções gerais para problemas que ocorrem com frequência na programação.
- ◇ Um desses padrões é chamado 'Decorator'
 - Esse padrão adiciona comportamento a um método ou objeto em tempo de execução





Decorators `@property` e `@attr.setter`

- ◇ `@property` decora os métodos *getters*
- ◇ `@attribute_name.setter`, os métodos *setters*
- ◇ Não se utiliza os prefixos `'get_'` e `'set_'`
- ◇ Os métodos têm o nome do atributo a ser manipulado
 - Polimorfismo



Decorators @property e @attr.setter

Decorator *@property*. Transforma o método num *getter* do atributo saldo

@saldo.setter torna o método num *setter*

```
class Conta:
    __slots__ = ('_Conta__saldo')
    def __init__(self):
        self.__saldo = 0
    @property
    def saldo(self):
        return R$ {0:~.2f}'.format(self.__saldo)
    @saldo.setter
    def saldo(self, novo_saldo):
        if novo_saldo > 0:
            self.__saldo = novo_saldo
        else:
            raise Exception()
```

```
>> c1 = Conta()
>> c1.saldo = 100 # chama setter
>> c1.saldo # chama getter
R$ 100.00
>> c1.saldo = -100 # Exception
```





Decorator `@classmethod`

- ◇ Define métodos de classe
- ◇ `@classmethod` recebe uma referência à classe (geralmente chamado de *cls*) como primeiro parâmetro implícito (semelhante ao *self*, referência ao objeto)



Decorators @classmethod

`__contas` é definida como uma variável privada, de classe.

Decorator `@classmethod`.
Transforma o método acessível a nível de classe e não de objeto.

A cada novo objeto instanciado, `__contas` é incrementada. O método `contas_instanciadas` acessa a variável de classe e informa a quantidade de objetos instanciados

Pode-se acessar o método de classe tanto pela classe, quanto por um objeto instanciado

```
class Conta:
    __contas = 0
    def __init__(self):
        Conta.__contas += 1
    ...
    @classmethod
    def contas_instanciadas(cls):
        return '{} contas ativas'.format(cls.__contas)

>> c1 = Conta()
>> print(Conta.contas_instanciadas())
1 contas ativas
>> c2 = Conta()
>> print(c2.contas_instanciadas())
2 contas ativas
```



Referência Bibliográfica

Sebesta, R. W. (2011). *Conceitos de Linguagens de Programação*. 9 ed. Bookman.

Capítulo 12

Perkovic, L. *Introdução à Computação Usando Python - Um Foco no Desenvolvimento de Aplicações*. Editora LTC, 1. ed., 2016.

Capítulos 8 e 9.

