Vimos que o paradigma da Orientação a Objetos está relacionado com a organização do código, um conceito que começamos a aplicar no código. Juntamos as características — como `numero`, `titular`, `saldo` e `limite` — com as funcionalidades de uma conta.

Organizamos o código, porém, será que devemos continuar com essa abordagem? No mundo procedural, não somos obrigados a adotar OO, não há algo que reforça a necessidade de utilizarmos essa maneira mais organizada de escrever o código. Ou seja, cabe ao desenvolvedor decidir se quer adotar o paradigma OO.

Em um sistema maior, é grande a chance de que as funções fiquem separadas em arquivos e módulos diferentes do projeto. No entanto, pode ser trabalhoso encontrar onde está cada trecho do código e isso, pode resultar em retrabalho e escrever funções já existentes. O paradigma Orientado a Objetos nos incentiva a agrupar funcionalidades relacionadas em um mesmo lugar. Este é um dos principais problemas.

Mas existe ainda outro problema, nós temos a opção de acessar o saldo diretamente no console, sem chamar uma funcionalidade e definir um novo valor como veremos abaixo:

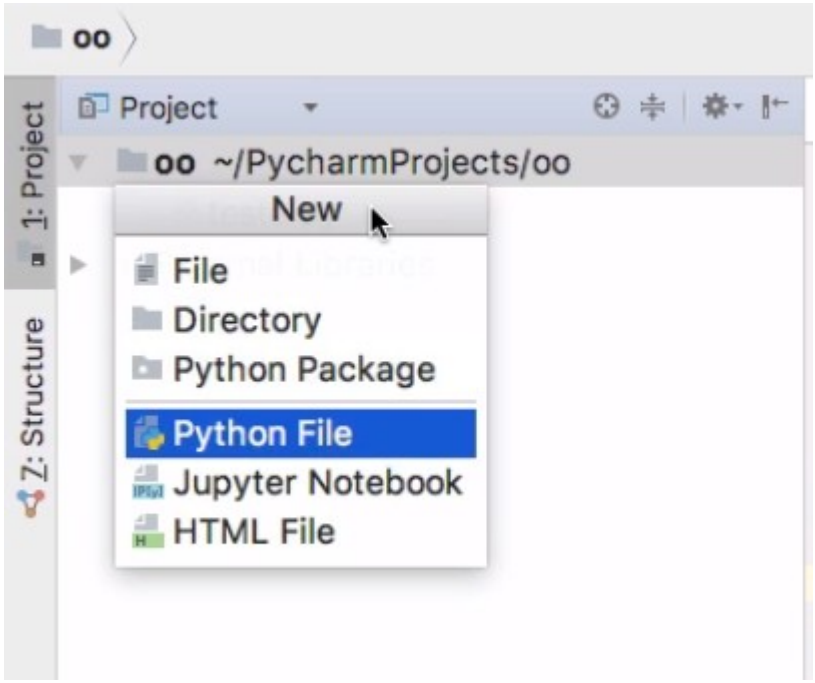
```
>>> conta["saldo"] = -300.0
```

Porém, ninguém deveria ter acesso ao saldo diretamente, sendo primeiramente necessário **depositar** ou **sacar** o dinheiro. Deveríamos manipular os dados somente por meio das funções.

Nós queremos resolver essas duas questões utilizando OO. Para isto, criaremos um objeto, no caso, falamos de uma conta. Trabalharemos com algo do mundo real, uma conta é algo concreto e inclui diversos dados. Porém, antes de termos um objeto, devemos definir as suas características.

A seguir faremos uma analogia: quando preparamos um bolo, geralmente, seguimos uma receita que define os ingredientes e o modo de preparação. A nossa conta é um objeto concreto assim como o bolo, e também precisamos definir antes uma receita. Porém, a "receita" no mundo OO recebe o nome de classe. Ou seja, antes de criarmos um objeto definiremos uma classe.

O próximo passo será gerar um novo arquivo Python, que receberá o nome de `conta.py`.



Dentro do arquivo gerado, definiremos a classe, que será antecedida pela palavra reservada `class`, utilizada na linguagem **Python** para definir a "receita". A classe vai receber o nome `Conta`, que por não ser uma palavra reservada, poderia ser outro qualquer.

Poderíamos ter adotado `ContaCorrente` como nome da classe, por exemplo. Desta forma, seguiríamos o padrão **CamelCase**, no qual cada palavra é iniciada com a letra maiúscula.

Seguiremos esta boa prática e escreveremos `Conta` com a primeira letra em caixa alta, seguida pela pontuação `:`, com isto indicaremos que se trata do início de um bloco de código.

Todo o conteúdo adicionado após `:` fará parte desta classe. Para fazer o código funcionar, incluiremos a palavra-chave `pass` no arquivo `conta.py`.

```
class Conta:


  pass
```

Em seguida, reiniciaremos o console e importaremos do módulo conta.

```
>>> from conta import Conta
```

Para criarmos um primeiro objeto, no console, digitaremos o nome da classe `Conta` utilizando os parênteses (()) para que o código seja executado.

```
>>> Conta()
<conta.Conta object at 0x10715f518>
```

Ele imprimiu a informação de que temos um objeto baseado na classe `Conta`, localizado dentro do módulo `conta`. Observe que a letra maiúscula foi utilizada para diferenciar as duas nomenclaturas. O objeto foi criado em memória e imprime o endereço `0x10715f518`.

Seguiremos com a analogia do bolo... É como se tivéssemos passado a receita para uma fábrica, determinando a tarefa de fabricação do objeto para outro. No entanto, onde essa fábrica vai criar o bolo? Temos um endereço, mas não precisaremos manuseá-lo. A linguagem Python irá abstrair esse dado para nós.

Se quisermos trabalhar com o objeto, teremos que fazer algo a mais. Chamaremos a classe `Conta` novamente, guardando o retorno dentro da referência `conta`.

```
>>> conta = Conta()
```

Esta referência guardará o endereço em memória para localizar o objeto posteriormente. É como se a fábrica nos avisasse em qual armário o bolo foi guardado, porém, o endereço não é o objeto em si. Trata-se apenas de uma referência.

Se executarmos a linha `conta = Conta()` não teremos nenhum retorno, porque o resultado da execução da `Conta` foi atribuída a `conta`. Mas, se executarmos `conta` no console, teremos o seguinte retorno:

```
>>> conta
<conta.Conta object at  0x10715fe10>
```

Observe que o endereço retornado é diferente do que foi impresso na primeira execução. Isto ocorreu porque chamamos a classe e foi criado um segundo objeto.

Geramos dois objetos do tipo `conta`, baseada na mesma classe. Vale lembrar que uma classe pode gerar vários objetos, porém, queremos que ela tenha várias informações. É necessário que `Conta` trabalhe com vários dados para depositar e sacar. Faremos isto adiante.

Seguimos com nossa introdução ao mundo de Orientação a Objetos, vimos três conceitos fundamentais. A classe `Conta` está praticamente vazia, mas ela já existe e funciona como a receita do projeto, que é construído a partir da seguinte execução:

```
>>> conta = Conta()
>>> conta
<conta.Conta object at 0x10715fef0>
```

Chamaremos a classe como se fosse uma função. O objeto é criado em memória por baixo dos panos pelo Python. Ele abstrai este processo, portanto não devemos nos preocupar com isso.

O objeto será criado na hora de chamar a classe `Conta`. Em memória, o objeto é criado e o Python devolve o endereço que será guardado dentro da variável `conta`.
Em Orientação a Objetos, as variáveis são denominadas: referências.
Estes conceitos servem para diversas linguagens, além de Python. Se trabalharmos com PHP ou Java, faremos o mesmo. Teremos uma referência, uma classe, uma construção de objeto que poderemos aplicar em outra linguagem.

Nossa classe continua vazia, a seguir, definiremos o conteúdo dela, definindo quais são suas características. No mundo Orientação a Objetos, essas características são chamadas de atributos.
Os atributos da conta são: `numero`, `titular`, `saldo` e `limite`. Na hora de criar um objeto, o Python pode executar uma função, automaticamente, dentro da classe para definir os atributos.
Sendo uma função automática, receberá um nome especial. Adicionaremos dois caracteres _ antes e depois do nome da função construtora, para criarmos `__init__`. O Python constrói o objeto, cria um lugar na memória e depois chama a função `__init__`. Como demonstração, segue o código:

```
class Conta:

    def __init__(self):

        print("Construindo objeto...")
```

Em seguida, reiniciaremos o console e importaremos novamente:

```
>>> from conta import Conta

>>> conta = Conta()

Construindo objeto ...
```

Automaticamente ele retornou `Construindo objeto...`, pois o Python cria o objeto em memória, encontra um espaço e, depois, a função construtora é chamada.
Criamos uma função, mas a ideia é definir os atributos e as características. Para isso, precisaremos da variável `self`, que está dentro da função `__init__()`.
Em seguida, no arquivo `conta.py`, usaremos interpolação e `format(self)` dentro de `print()`. O Python cria automaticamente `self`.

```
class Conta:

    def __init__(self):

        print("Construindo objeto...{}".format(self))
```

Reiniciaremos o console e testaremos o código.

```
>>> from conta import Conta

>>> conta = Conta()

Construindo objeto ... <conta.Conta object at 0x1020d7f28>
```

Repare que ele já conhece esta saída. Localizamos a conta, o objeto e o endereço. Considerando que é a mesma saída, ele imprimirá o valor da referência.

> `self` é a referência que sabe encontrar o objeto construído em memória.

Agora que temos o endereço, utilizaremos `self` para acessar o objeto e definir seus atributos e características.

```
class Conta:


    def __init__(self):

        print("Construindo objeto...{}".format(self))

        self.numero = 123

        self.titular = "Nico"
```

```
        self.saldo = 55.0
        self.limite = 1000.0
```

Em self.numero, o caractere "ponto" (.) é um comando de ida ao objeto
e numero, titular, saldo e limite são atributos.
Reiniciaremos o console e testaremos novamente, agora, com os atributos. Tudo continua funcionando, sem mudanças porque não utilizamos os atributos.

No entanto, não queremos deixar o valor dos atributos fixos, o ideal é que eles variem de acordo com a conta que está sendo criada.

Em teste.py, nós havíamos definido alguns parâmetros na função cria_conta (),:

```
def cria_conta(numero, titular, saldo, limite):
    conta = {"numero": numero, "titular": titular, "saldo": saldo, "limite": limite}
    return conta
```

De volta a conta.py, o próximo passo será também definir parâmetros para a função __init__(), aproveitando os mesmo dados da função cria_conta():

```
class Conta:


    def __init__(self, numero, titular, saldo, limite):
        print("Construindo objeto...{}".format(self))
        self.numero = 123
        self.titular = "Nico"
        self.saldo = 55.0
        self.limite = 1000.0
```

Além do self que é passado automaticamente, queremos que os dados sejam alterados, por isso, adicionaremos os nomes dos parâmetros.

```
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
```

Desta forma, o limite do objeto (self.limite) será o limite recebido do parâmetro da função __init__(). E qual é a origem dos dados? O valor do self é passado pelo Python, responsável pela criação final do objeto em memória. No entanto, os valores dos parâmetros como numero deverão ser definidos usando a aplicação.
Iremos importar Conta no console, criar a conta e especificar os valores:

```
>>> from conta import Conta
```

```
>>> conta = Conta(123, "Nico", 55.5, 1000.0)
Construindo objeto ... <conta.Conta object at 0x11077bcc0>
```

Observe que passamos os parâmetros da conta e nosso código foi executado. Isso é um bom sinal e, por isso, criaremos novos objetos.

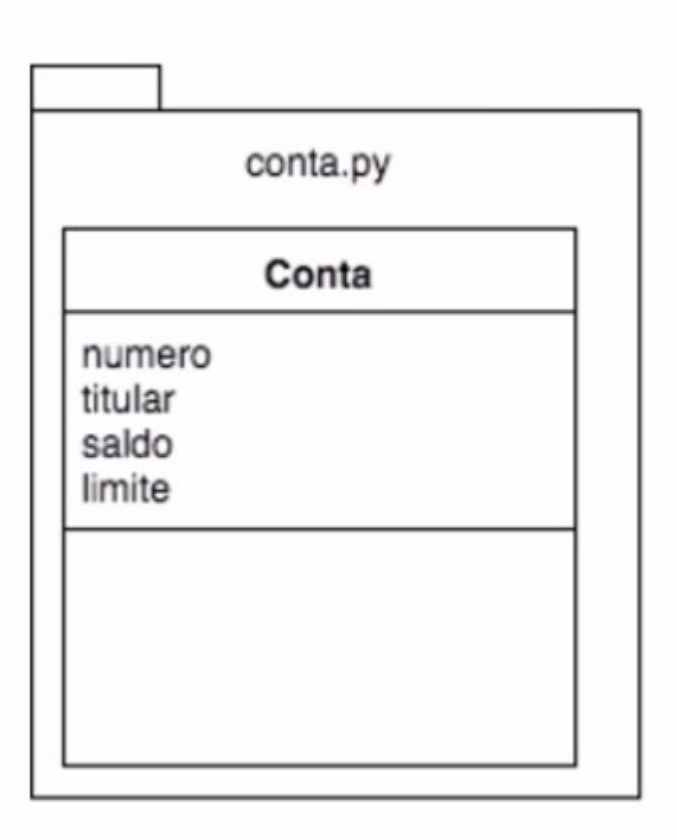
```
>>> conta2 = Conta(321, "Marco", 100.0, 1000.0)
```

Será gerada a `conta2`, sendo possível acessar as duas contas criadas pelo console.

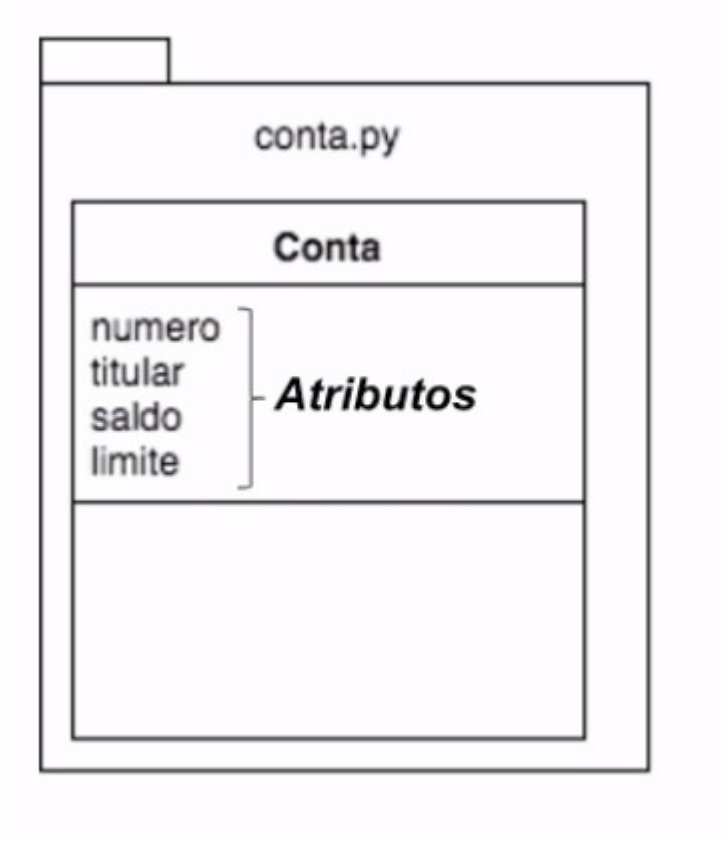
```
>>> conta
<conta.Conta object at 0x11077bcc0>


>>> conta2
<conta.Conta object at 0x1109e1da0>
```

Cada conta possui um número, titular, saldo e limite, agora, falta trabalharmos com esses atributos. Nós faremos isso a seguir.

Vamos começar a utilizar os atributos da conta. Anteriormente, criamos a classe `Conta()` dentro de `conta.py`. Usaremos UML (Linguagem de Modelagem Unificada), para explicar a estrutura de `Conta`:



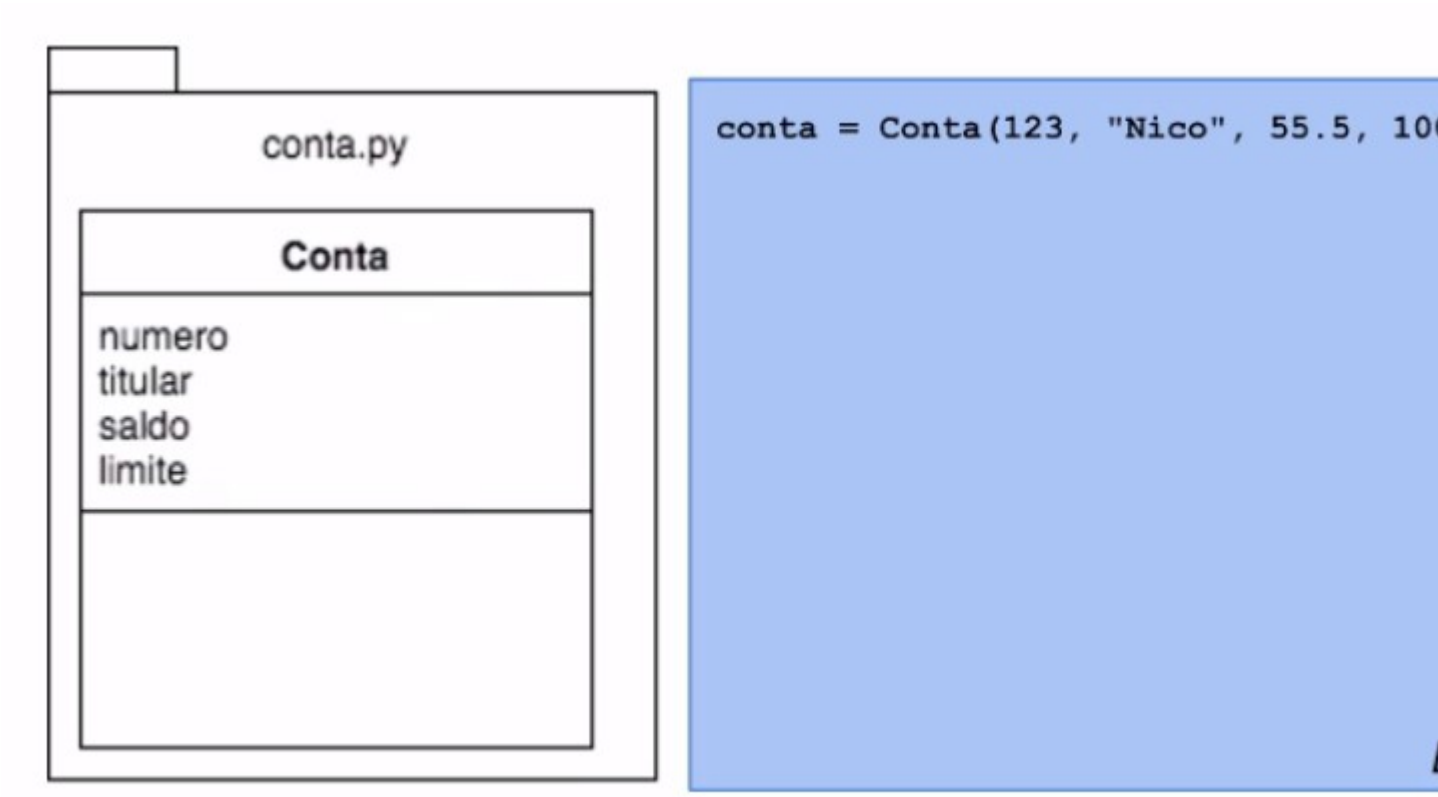
Logo acima temos um diagrama de classes, que mostra os atributos: "numero", "titular", "saldo" e "limite". A linguagem UML é uma anotação visual, usada para descrever o nosso sistema por meio de gráficos e desenhos.



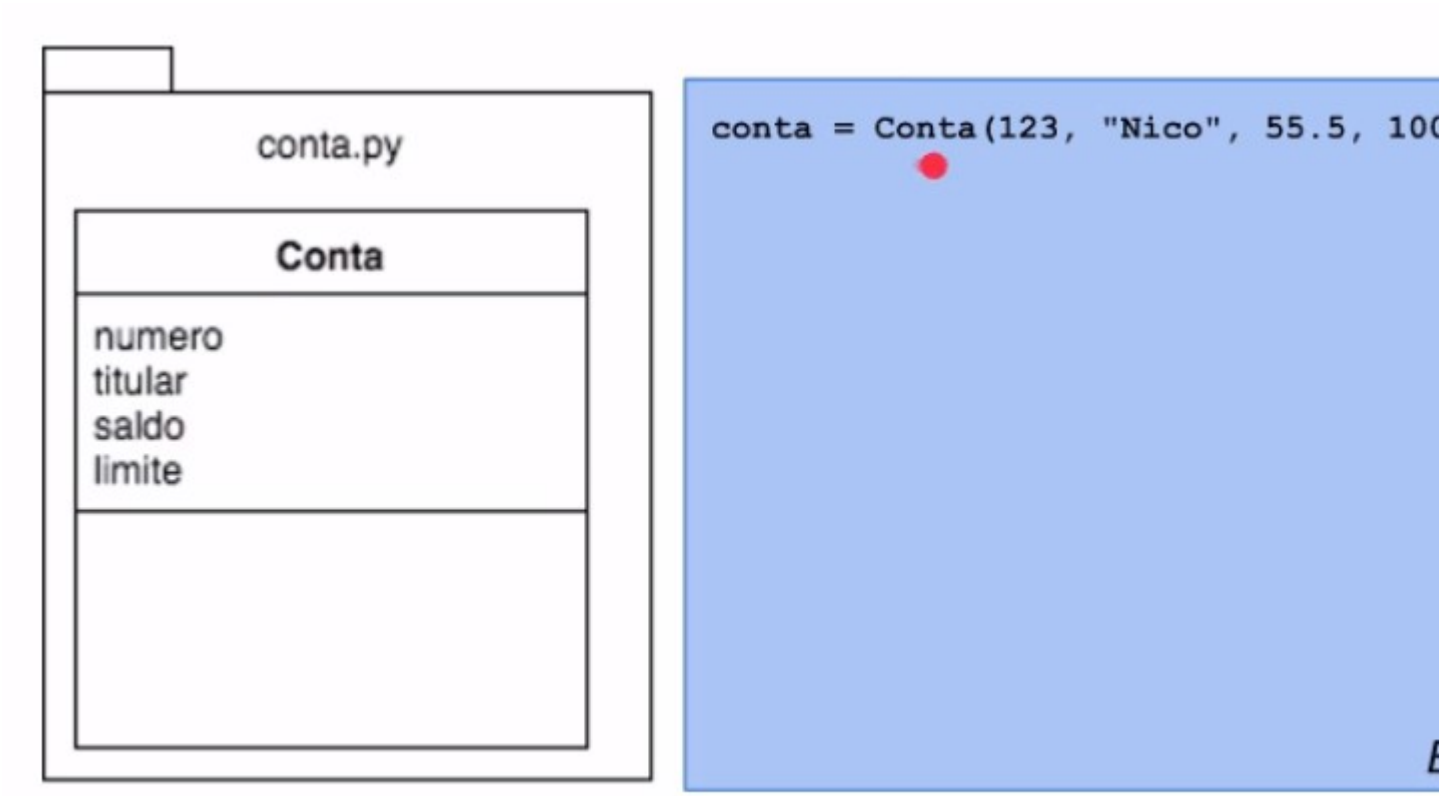
A Conta() está dentro de um módulo — que em algumas linguagens receberá o nome de package ou namespace —, sendo que um módulo poderia ter uma ou mais sintaxes. Por enquanto, usamos a classe para criar uma conta, passando alguns valores como parâmetro para a função construtora.

```
>>> conta = Conta(123, "Nico", 55.5, 1000.0)
```

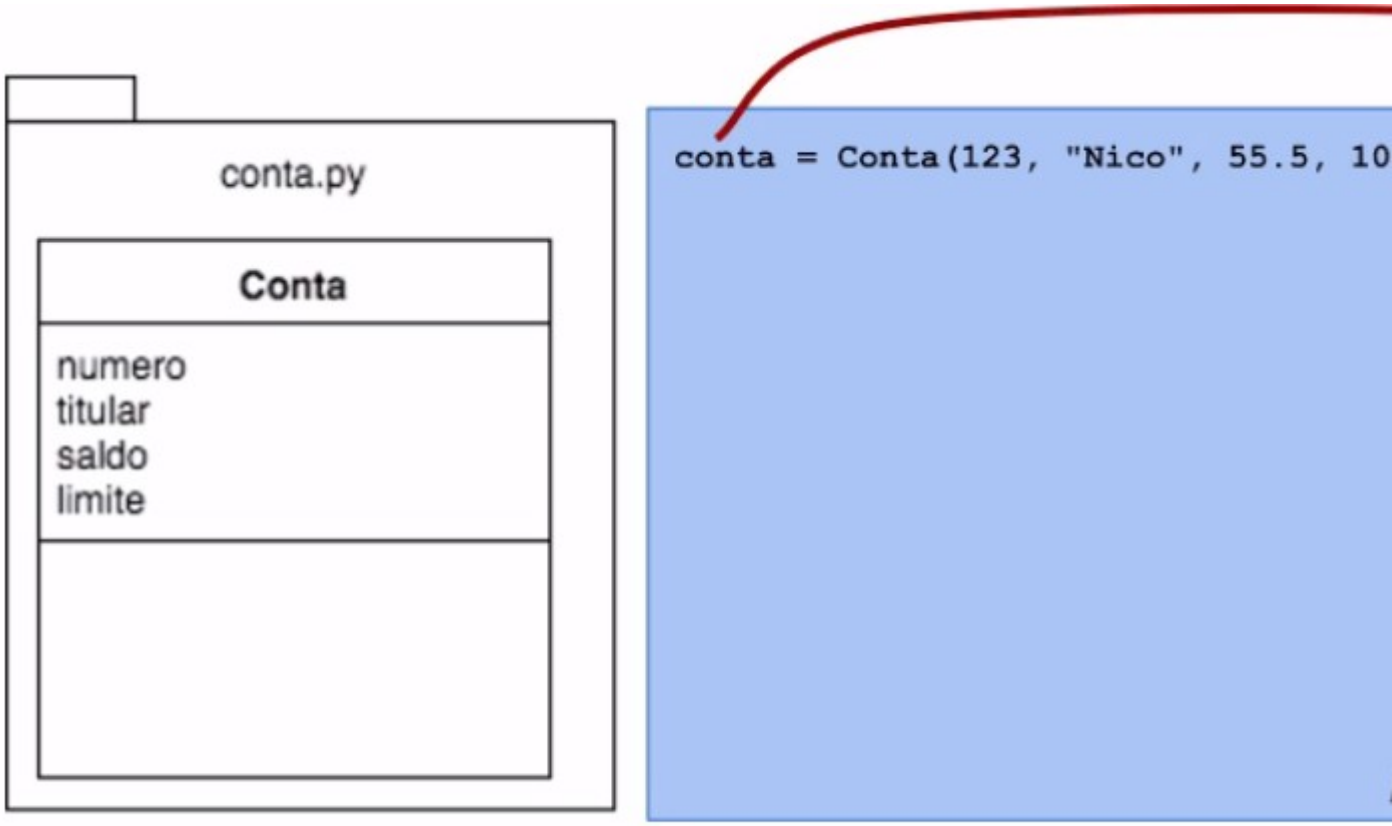
O diagrama ficará da seguinte maneira:



Observe que `__init__` não é exibido no diagrama de classes, porque se trata de uma função implícita, que é chamada automaticamente. Nós chamamos a classe `Conta` seguida de parênteses e, por baixo dos panos, o Python passará os valores para a função construtora. Quando executamos a linha com a referência `conta`, em memória, o Python vai gerar o objeto em que serão guardados os valores.



A referência `conta` sabe onde se encontra o objeto em memória. Mesmo sem sabermos como, o Python aloca isso e encontra espaço; nós não temos controle sobre essa parte do processo.



Falta ainda utilizarmos os atributos. De volta ao PyCharm, no console, vamos importar Conta e criar a primeira conta.

```
>>> from conta import Conta
>>> conta = Conta(123, "Nico", 55.5, 1000.0)
Construindo objeto ... <conta.Conta object at 0x10293ae48>
```

Em seguida, criaremos a segunda conta.

```
>>> conta2 = conta(321, "Marco", 100.0, 1000.0)
Construindo objeto ... <conta.Conta object at 0x10293acf8>
```

Temos dois objetos criados, agora, falta fazer alguns atributos. Nós chegaremos ao objeto por meio da referência conta, responsável por indicar onde se encontra o objeto. Precisamos dizer usando a linguagem Python "vai para esse objeto e acessa aquele atributo". O pedido de "vai" nas linguagens Orientadas a Objeto é indicado com ., e o console compreenderá que queremos fazer algo com esse objeto. No caso, nós queremos mostrar o saldo, logo, executaremos conta.saldo no console, e o objeto será impresso.

```
>>> conta.saldo
55.5
>>> conta2.saldo
100.0
```

Ele imprimiu o objeto guardado dentro do atributo saldo. Então, para imprimirmos a referência, podemos utilizar os outros atributos, como conta.titular, conta.limite.
Agora, você pode praticar o conteúdo apresentado com os exercícios, e mais adiante, a classe Conta receberá novos métodos.

Chegou a hora de criar a sua primeira classe, a classe Conta. Para tal, crie o arquivo conta.py e siga os passos abaixo:

1 - Defina a classe, utilizando a palavra-chave class, e em seguida defina o seu nome.
2 - Defina a função construtora da classe, recebendo uma referência do próprio objeto como argumento.
3 - Receba também como argumento os valores dos atributos da classe, isto é, numero, titular, saldo e limite.
4 - Através da referência do objeto, defina os atributos numero, titular, saldo e limite com os respectivos valores recebidos como argumento.

O código do arquivo conta.py ficará assim:

```
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
```

No Python Console, dentro do próprio PyCharm, teste o código, criando contas e acessando os seus atributos, por exemplo:

```
>>> from conta import Conta
>>> conta = Conta(123, "Nico", 55.5, 1000.0)
Construindo objeto ... <conta.Conta object at 0x7fc4ed132048>
>>> conta2 = Conta(321, "Marcos", 100.0, 1000.0)
Construindo objeto ... <conta.Conta object at 0x7fc4ed1324a8>
>>> conta.titular
'Nico'
>>> conta2.titular
'Marcos'
```

O que aprendemos?

Nesta aula, aprendemos:

•Classes
•Objetos
•Função construtora
•Endereço e referência de objetos
•Atributos de classe
•Acesso aos atributos através do objeto

AULA 3

Nós avançamos bastante no conteúdo e já definimos a função construtora `__init__`. Em algumas linguagens (como Java), esta função recebe o nome de construtor. O Python não se enquadra neste caso, porque uma função construtora não é um equivalente exato do método construtor.

É importante ter definido os atributos e suas principais características, o que mostramos como fazer. Agora falta adicionar o que um objeto `Conta` pode fazer.
De acordo com `teste.py`, definimos que podemos: depositar, sacar e tirar extrato.

```
def cria_conta(numero, titular, saldo, limite):
    conta = {"numero": numero, "titular": titular, "saldo": saldo, "limite": limite}


def saca(conta,valor):
    conta["saldo"] -= valor
```

```
def extrato(conta):
    print("Saldo é {}".format(conta["saldo"]))
```

Estas são as funções relacionadas com uma conta, que nas linguagens Orientadas a Objetas são nomeadas como métodos. Ou seja, os métodos são referentes às ações que um objeto sabe fazer. Onde eles serão colocados? Na classe, desta forma, sempre adicionaremos elementos relacionados à conta na classe `Conta`.
Primeiramente, adicionaremos uma nova função na classe. Neste caso, temos liberdade de usar o termo "função", mas ela não tem uma utilidade específica como `__init__`. Incluiremos a função `extrato`, que receberá a referência `self` do próprio editor porque iremos usá-lo para imprimir o saldo do titular, incluindo sempre a função `format()` dentro do `print()`.

```
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


    def extrato(self):
        print("Saldo {} do titular {}".format(self.saldo, self.titular))
```

O saldo está no objeto, para alcançá-lo, usamos a referência que sabe onde ele está. No caso, usaremos o `self` como referência.
Em seguida, criaremos a conta do titular `Nico` no console.

```
>>> from conta import Conta
>>> conta = Conta(123, "Nico", 55.5, 1000.0)
Construindo objeto ... <conta.Conta object at 0x105c24cf8>
```

Porém, se executarmos apenas `extrato()` no console, receberemos uma mensagem de erro, informando que `extrato` não existe. Isso aconteceu, porque o Python não sabe quais objetos queremos utilizar.
O próximo passo será criar o objeto `conta2`, que receberá novos atributos. Ou seja, temos dois objetos em uma mesma classe.

```
>>> from conta import Conta
>>> conta2 = Conta(321, "Marco", 100.0, 1000.0)
Construindo objeto ... <conta.Conta object at 0x109fdff60>
```

Para chamarmos `extrato()`, especificaremos de qual é o objeto que queremos os dados referentes ao extrato. Esta é uma diferença da abordagem procedural para OO.

Dentro da função __init__(), usamos self seguido do . para acessar o objeto, como em self.numero = numero. Faremos o mesmo para acessar extrato, que será antecedido pela referência.

```
>>> conta.extrato()
Saldo 55.5 do titular Nico
```

Ao executarmos esta linha, o Python entenderá que a referência conta aponta para o objeto Conta, baseado na classe homônima. O retorno será a mensagem Saldo 55.5 do titular Nico. Como usamos a referência, ele encontrou a referência e imprimiu o valor do saldo e o titular. Podemos fazer o mesmo com o outro objeto.

```
>>> conta2.extrato()
Saldo 100.0 do titular Marco
```

O Python nos retornou o valor da referência. Conseguimos adicionar o primeiro método dentro da classe Conta. No teste.py, onde seguimos a programação procedural, criamos deposita e saca. Iremos adicionar os dois métodos em conta.py, que receberam automaticamente a variável self, lembrando que deposita() deve agregar um valor ao saldo, por isso, além de self, receberemos valor como parâmetro.
Usaremos o valor para modificar o saldo do objeto, utilizando a referência que sabe onde está o objeto, e acessando o saldo deste, no qual incrementamos um valor. O método saca() é bastante semelhante, com a diferença que ele vai subtrair um valor.

```
def extrato(self):
    print("Saldo de {} do titular {}".format(self.saldo, self.titular))


def deposita(self, valor):
    self.saldo += valor


def saca(self, valor):
    self.saldo -= valor
```

A seguir criaremos um novo objeto no console e chamaremos o método extrato():

```
>>> from conta import Conta
>>> conta = Conta(123, "Nico", 55.5, 1000.0)
Construindo objeto ... <conta.Conta object at 0x10db29e80>
>>> conta.extrato()
Saldo de 55.5 do titular Nico
```

Em seguida, será a vez de invocar o método deposita(). No caso, depositaremos 15 reais.

```
>>> conta.deposita(15.0)
```

Se executarmos, não receberemos uma mensagem de erro. Para testar se tudo saiu bem, vamos pedir o extrato novamente:

```
>>> conta.deposita(15.0)
>>> conta.extrato()
Saldo de 70.5 do titular Nico
```

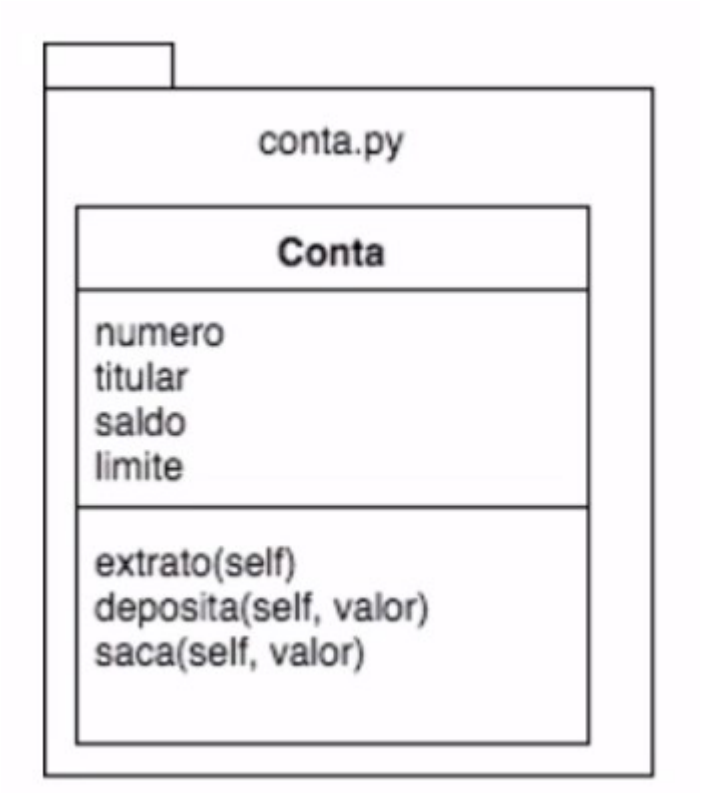
Agora o saldo da conta tem 70.5 reais. Vamos experimentar usar o método saca().

```
>>> conta.saca(10.0)
>>> conta.extrato()
Saldo de 60.5 do titular Nico
```

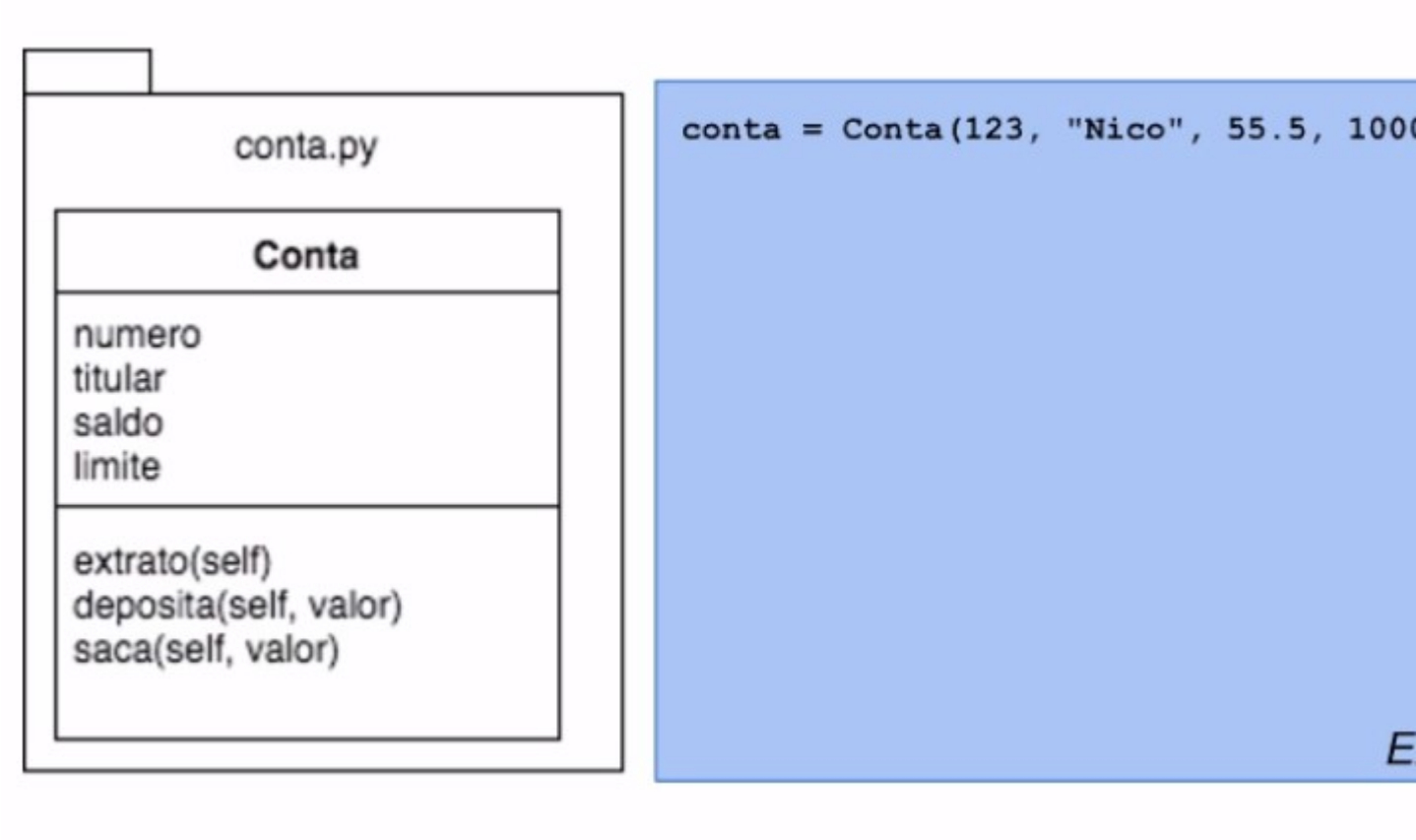
O saldo tem o valor 60.5. Conseguimos sacar e manipular o saldo, além de imprimirmos o valor atualizado do objeto, com o método extrato().
Observe que quem utiliza o objeto se comunica com ele por meio da referência conta, mas não sabemos como funciona o método extrato(), por exemplo, porque a função está encapsulada.
Em uma aplicação do mundo real, os métodos serão mais complexos. A função saca() deveria, por exemplo, verificar o limite. Mas ao executarmos saca(), é irrelevante a complexidade da funcionalidade, porque ela ficou encapsulada dentro do método. O uso de encapsulamentos é uma característica da Orientação a Objetos.
A classe possui todas as informações, como os atributos e funcionalidades da conta que, certamente em um sistema real, serão numerosas.

Vimos por que criamos a classe, mostramos como criar o objeto por meio da função construtora __init__() e definir os atributos dentro dela, além dos métodos e funcionalidades que o objeto deve ter. Aprendemos também como chamar os métodos usando a referência.
Nós já temos uma base de conhecimento para praticar com os exercícios.

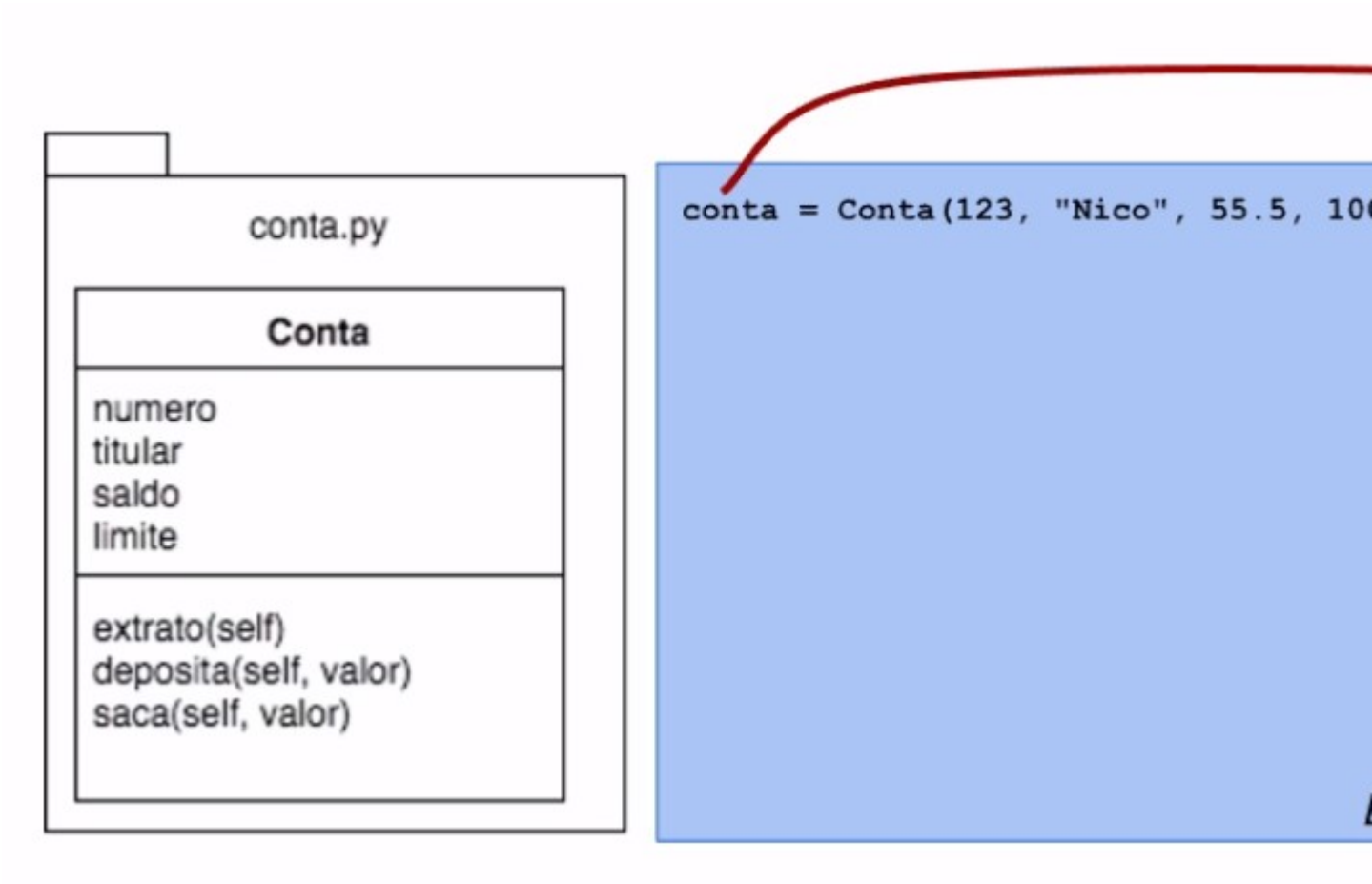
O conteúdo visto neste curso, pode ser aplicado para Java, C#, PHP, Ruby ou qualquer outra linguagem Orientada a Objeto.



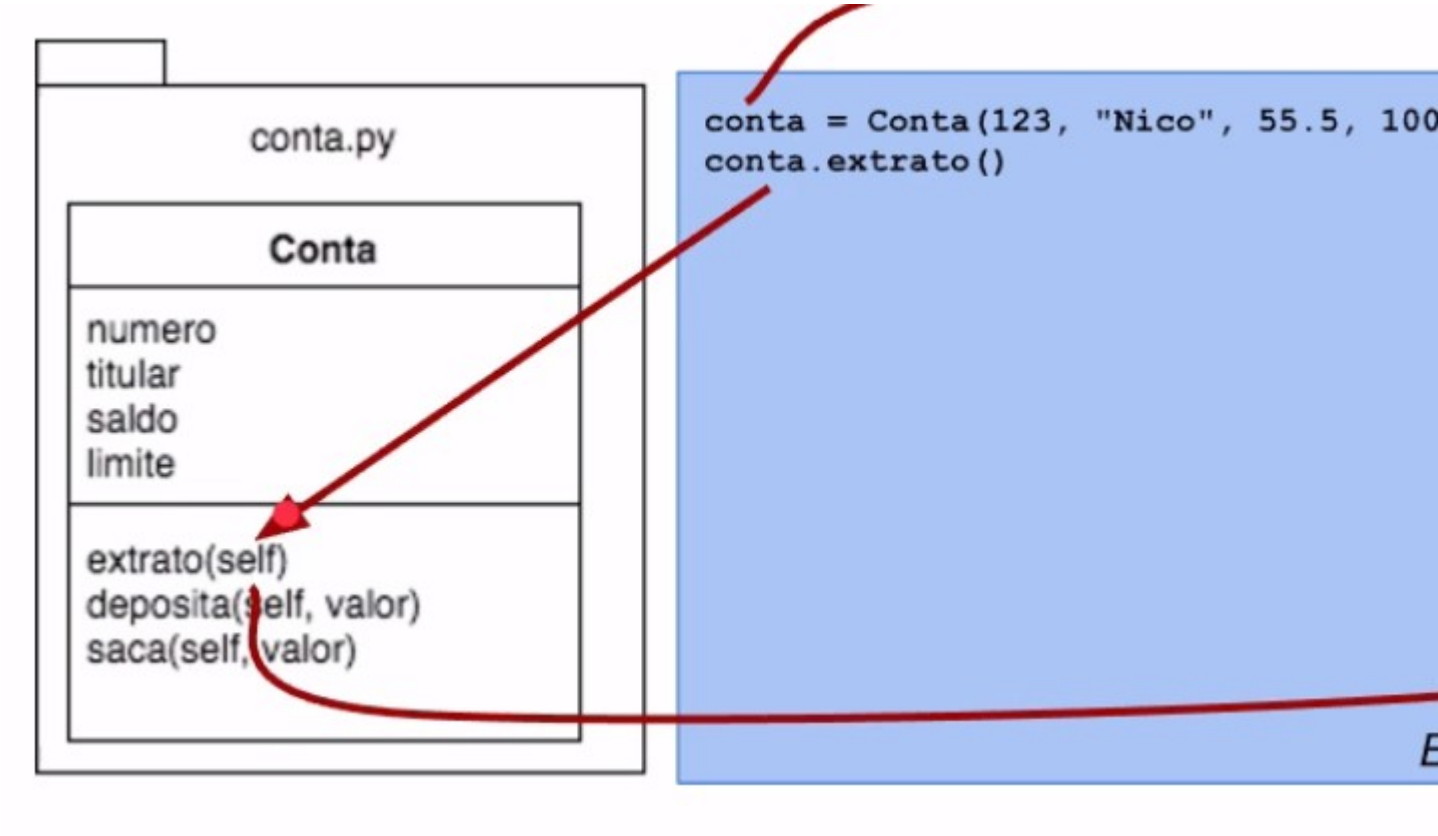
O diagrama da classe Conta contém novos dados: os métodos extrato(), deposita(), saca() foram incluídos juntos com seus parâmetros. Normalmente, não incluímos a função construtora __init__, considerando que ela é chamada de forma implícita.
Depois, criamos a classe Conta, usamos parâmetros, dentro da referência conta — como vemos na segunda parte do diagrama. O resultado é o objeto Conta criado em memória, com os parâmetros passados para a função construtora, como é ilustrado na parte verde do diagrama.



A referência conta é devolvida na execução, em que é retornado o endereço dentro de conta.



Quando falamos anteriormente sobre atributos, mostramos que as referências são utilizadas para acessar o objeto e imprimir um valor. Vimos que o método extrato() pode ser utilizado para impressão de valores, como em conta.extrato(). Desta vez, a referência foi usada para a chamada do método, assim o objeto será passado automaticamente.



Neste caso, a variável `self` e `conta` serão equivalentes. Ou seja, `self` também sabe onde se encontra o objeto, por isso, dentro do método `extrato()` podemos implementar a maneira de como os dados serão impressos. Quando a função for chamada no console, uma mensagem com o valor do saldo será impressa na saída.

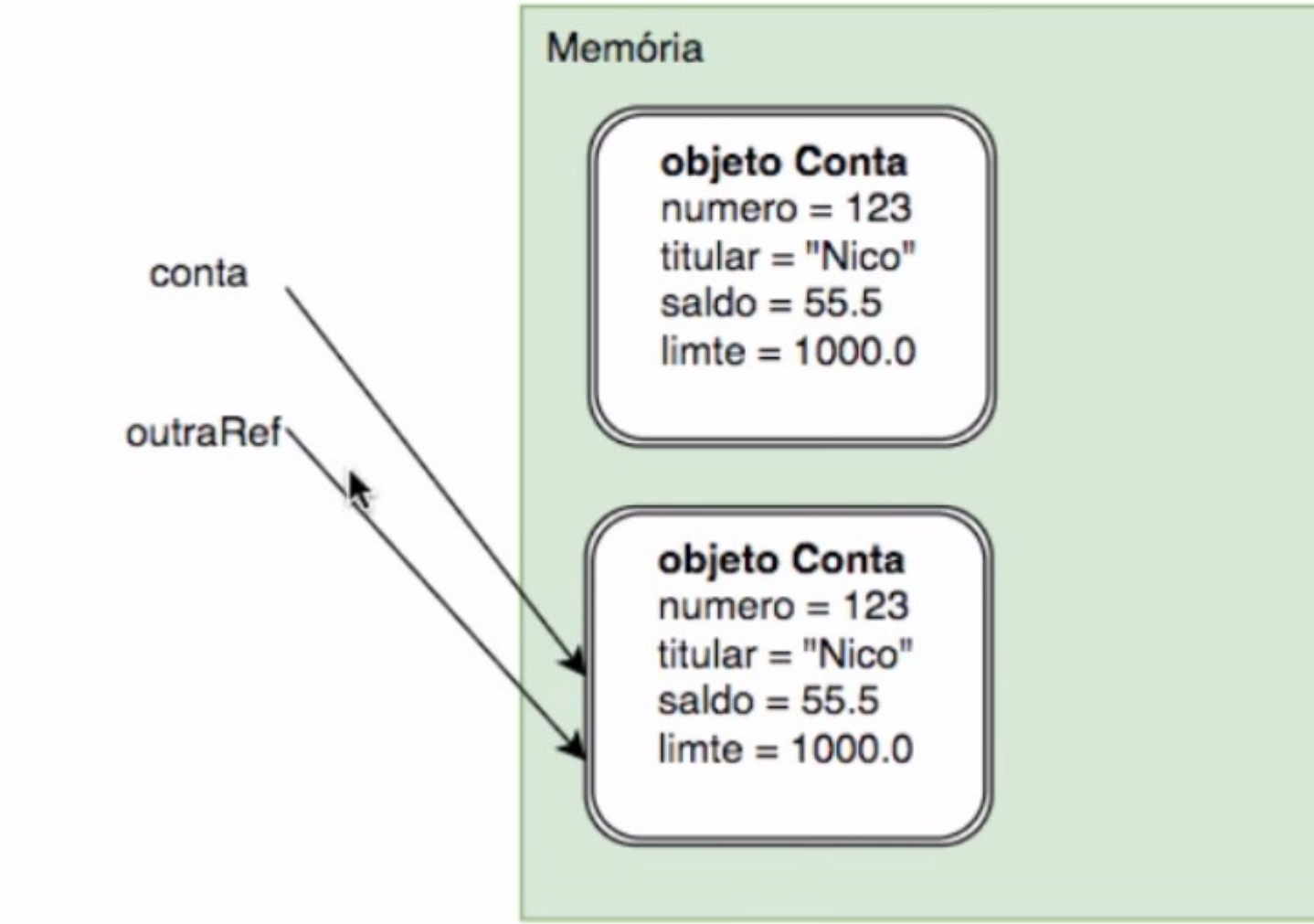
Simularemos no diagrama a criação do segundo objeto. Novamente, o endereço será guardado dentro da referência do objeto. Agora, a variável conta2 vai apontar para Conta com os dados do titular Marco, sendo possível invocar o método extrato(), que retornará os valores dos atributos relacionados a este objeto.

Novamente, teremos a variável self, mas dessa vez, ela apontará para o segundo objeto. Isto significa que, dependendo da referência utilizada, o self apontará para objetos distintos.

Na parte amarela do diagrama, em que vemos o retorno do método extrato(), veremos duas mensagens:

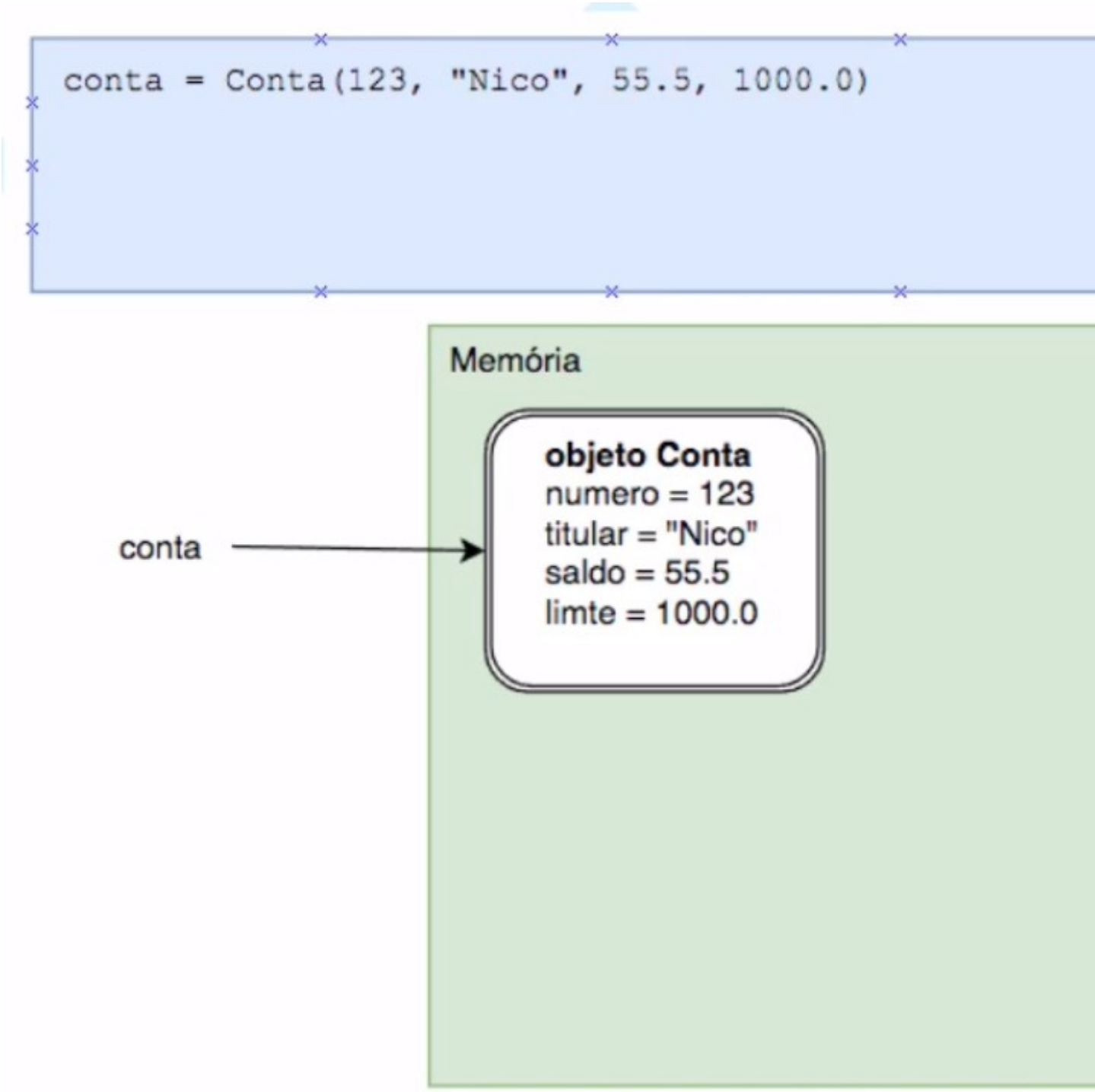
```
Saldo de 55.5 do titular Nico

Saldo de 100.0 do titular Marco
```

```
conta = Conta(123, "Nico", 55.5, 1000.0)
conta = Conta(123, "Nico", 55.5, 1000.0)
outraRef = conta
```



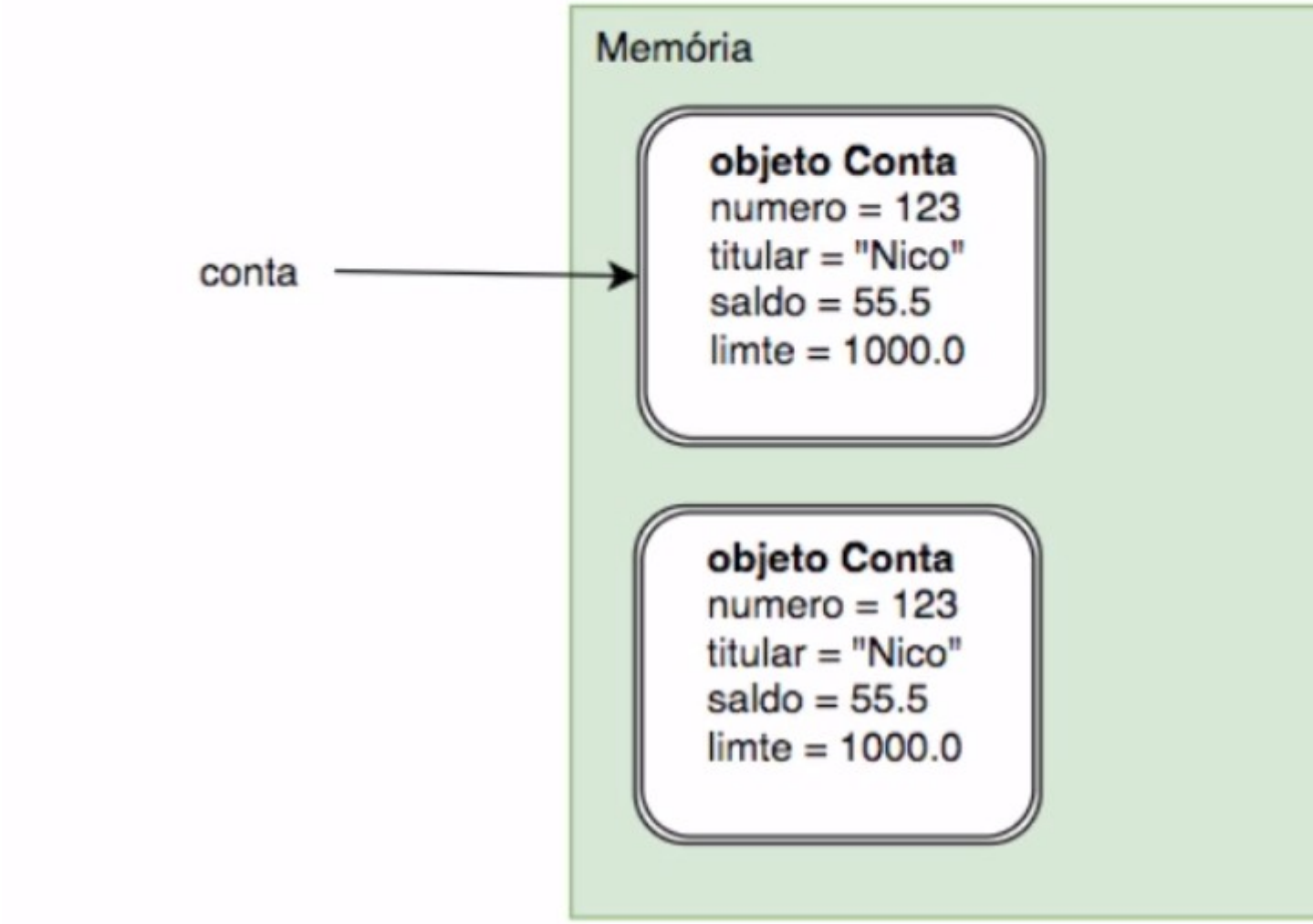
É útil criarmos esses desenhos incluindo as referências utilizadas no código, nós faremos mais isso adiante. Mais adiante, nos aprofundaremos no assunto encapsulamento.

Vamos falar mais sobre os conceitos fundamentais como "referência" e "objeto". Criamos um novo diagrama:



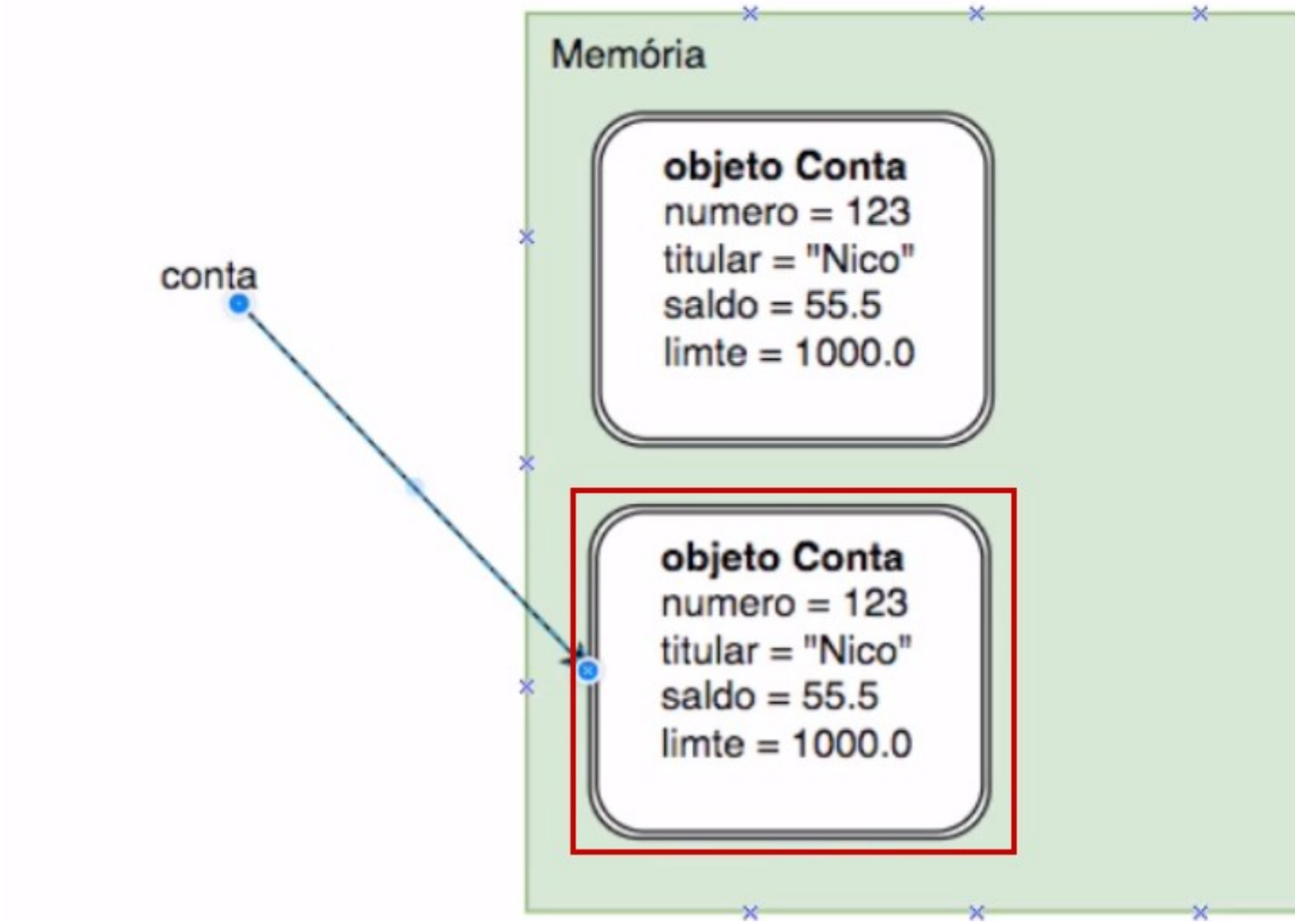
No desenho, mostramos a criação da conta e o objeto em memória. Ao chamarmos a função construtora `__init__`, por baixo dos panos, será gerado o objeto. Em seguida, duplicaremos as linhas inclusas nas duas partes do objeto, desta forma representaremos que temos duas contas e dois objetos, no qual estarão presentes os mesmos dados.

```
conta = Conta(123, "Nico", 55.5, 1000.0)
conta = Conta(123, "Nico", 55.5, 1000.0)
```



Neste caso, teremos dois objetos que representam a conta `123`. Na realidade, só podemos ter uma conta com esse número, mas em um sistema, se chamamos duas vezes o mesmo construtor, teremos dois objetos. Além disso, observem que, com o objeto criado, atribuímos o endereço à mesma referência. Desta forma, a referência `conta` relacionada com um objeto apontará especificamente para o segundo.

```
conta = Conta(123, "Nico", 55.5, 1000.0)
conta = Conta(123, "Nico", 55.5, 1000.0)
```



A referência consegue encontrar o objeto criado mais recentemente. Porém, como faremos para alcançar o primeiro objeto? Ficamos sem referência para ele e, de fato, não temos como alcançá-lo. O primeiro objeto permanecerá ocupando espaço, mas sem ser acessado.

Fazendo uma analogia, seria como se anotássemos um endereço específico em um papel e ao jogarmos essa anotação fora, o local será esquecido e nunca mais localizado.

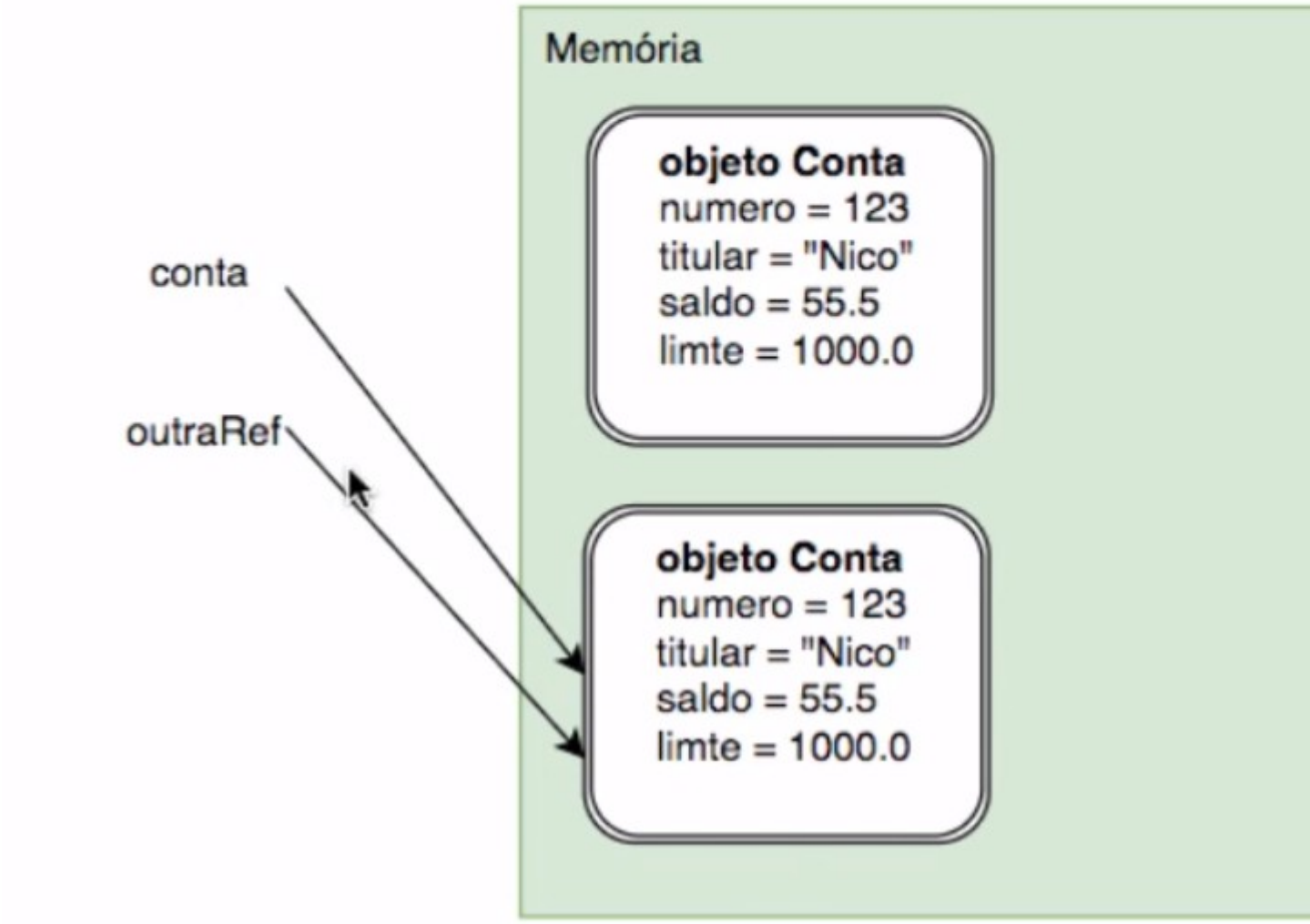
Quando criamos um programa, são gerados diversos objetos que em algum momento serão abandonados. Dentro da máquina virtual, na execução do Python, existe um processo que procura esses objetos esquecidos. Os itens inutilizados serão apagados e o espaço livre em memória será reutilizado. No caso, o responsável por jogar fora esses objetos em desuso é o coletor de lixo (garbage collector, em inglês) do Python.

Em seguida, criaremos uma terceira variável, que receberá o nome de outraRef. Será para ela que atribuiremos o valor da referência conta.

```
>>> conta = Conta(123, "Nico", 55.5, 1000.0)
>>> conta = Conta(123, "Nico", 55.5, 1000.0)


>>> outraRef = conta
```

O valor da referência conta fica com a referência outraRef. Usando novamente a analogia do endereço anotado em um papel, é como se tivéssemos feito uma fotocópia do papel. Em linguagem UML, o diagrama ficaria assim:

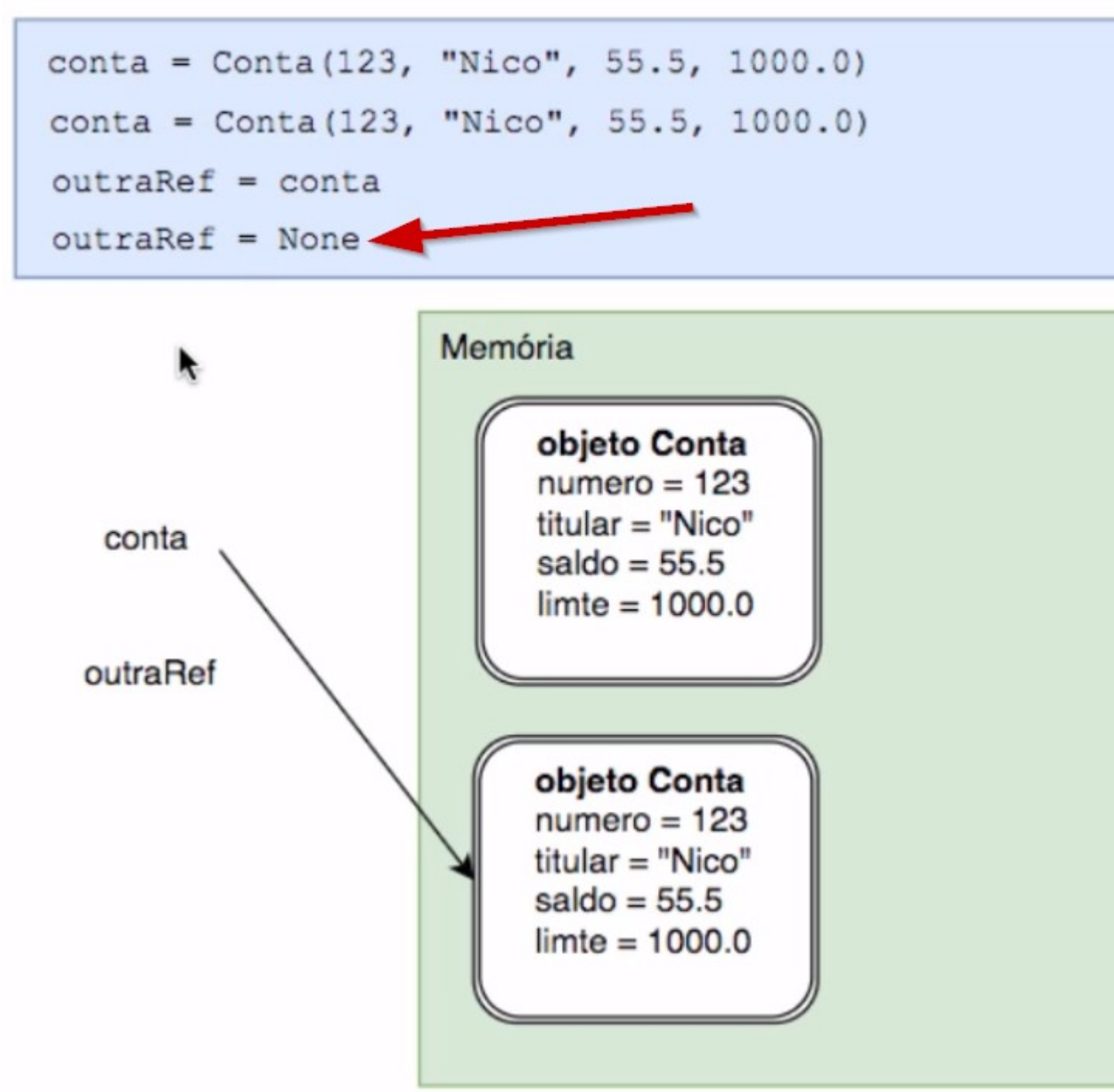
```
conta = Conta(123, "Nico", 55.5, 1000.0)
conta = Conta(123, "Nico", 55.5, 1000.0)

outraRef = conta
```



Observem que temos uma nova referência, no entanto, não criamos um novo objeto. Nós podemos ter diversas referências apontando para um mesmo objeto. Neste caso, podemos usar tanto `outroRef` ou `conta` para acessar um atributo.
O que acontece se quisermos desfazer uma referência, por exemplo, desreferenciar `outraRef`? Para isto, podemos usar a palavra especial `None`:

```
>>> outraRef = None
```

Nosso diagrama ficará assim:



Com o uso do `None`, indicamos que a variável já não aponta para um objeto. A palavra `None` é equivalente a palavra-chave `null` nas linguagens C# ou Java. Nós também removemos a seta que apontava a referência `outraRef` para o objeto `Conta`, porque já não é possível acessá-lo usando a referência `outraRef`.

Revisando: Vimos que os objetos abandonados são removidos pelo coletor de lixo do Python e que podemos ter mais de uma referência apontando para o mesmo objeto. Inclusive, podemos desfazer a referência para um objeto, como fizemos com `outraRef`.
Estamos criando uma base sólida de conceitos, mas falta vermos muita coisa.

Com a classe criada, falta definir o seu comportamento, criando métodos, para sacar, depositar e imprimir o extrato da conta.

Para tal, siga os passos abaixo:

1 - Crie o método `extrato`, que recebe como argumento uma referência do próprio objeto. Esse método imprimirá o saldo da conta.
2 - Crie o método `deposita`, que recebe como argumento uma referência do próprio objeto e o `valor`. Esse método adicionará o valor ao saldo da conta.
3 - Crie o método `saca`, que recebe como argumento uma referência do próprio objeto e o `valor`. Esse método subtrairá o valor do saldo da conta.

O código da classe `Conta` ficará assim:

```
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

    def extrato(self):
        print("Saldo de {} do titular {}".format(self.saldo, self.titular))

    def deposita(self, valor):
        self.saldo += valor

    def saca(self, valor):
        self.saldo -= valor
```

No Python Console, dentro do próprio PyCharm, teste o código, crie uma conta, deposite um valor, visualize o extrato com o saldo incrementado, saque um valor e visualize o extrato com o saldo decrementado, por exemplo:

```
>>> from conta import Conta
>>> conta = Conta(123, "Nico", 55.5, 1000.0)
Construindo objeto ... <conta.Conta object at 0x7fa29f59a518>
```

```
>>> conta.deposita(300.0)
>>> conta.extrato()
Saldo de 355.5 do titular Nico
>>> conta.saca(100.0)
>>> conta.extrato()
Saldo de 255.5 do titular Nico
```

O que aprendemos?

Nesta aula, aprendemos:

- Métodos, que definem o comportamento de uma classe
- Criação de métodos
- Como chamar métodos através do objeto
- Acesso aos atributos através do **self**
- *Garbage Collector*
- O tipo **None**

AULA 4

Nós criamos atributos e métodos na classe Conta, por isso, ela já funciona. Mas ainda está incompleta.

Falamos anteriormente sobre encapsulamento, no console do PyCharm, importaremos da classe Conta.

```
>>> from conta import Conta


>>> conta = Conta(123, "Nico", 55.5, 1000.0)
Construindo objeto ... <conta.Conta object at 0x102b89128>
```

Chamamos uma Conta, chamando a função construtora __init__ por baixo dos panos. Com isso, conseguimos acessar os métodos (como sacar() ou deposita()) para acessar o atributo do objeto. Se quisermos imprimir o saldo, utilizaremos o método extrato() ou saldo(). No caso, invocaremos o último:

```
>>> conta.saldo
55.5
```

Acessaremos o objeto usando a referência. Com ela, conseguiremos também alterar o saldo da conta:

```
>>> conta.saldo = 60.0
```

Agora, se pedirmos para imprimir o extrato, o valor será atualizado:

```
>>> conta.saldo = 60.0


>>> conta.extrato()
Saldo de 60.0 do titular Nico
```

Porém, isto não deveria acontecer. O valor do saldo da conta deveria ser alterado a partir do método `deposita()`, localizalizado em `conta.py`:

```
def deposita(self, valor):
    self.saldo += valor


def saca(self, valor):
    self.saldo -= valor
```

Se quiséssemos saber o nome de alguém, seria uma falta de educação pegar diretamente o documento de identificação da pessoa, sem pedir autorização. Da mesma forma, seria mais apropriado usarmos um método para identificar o saldo, em vez de acessá-lo diretamente.

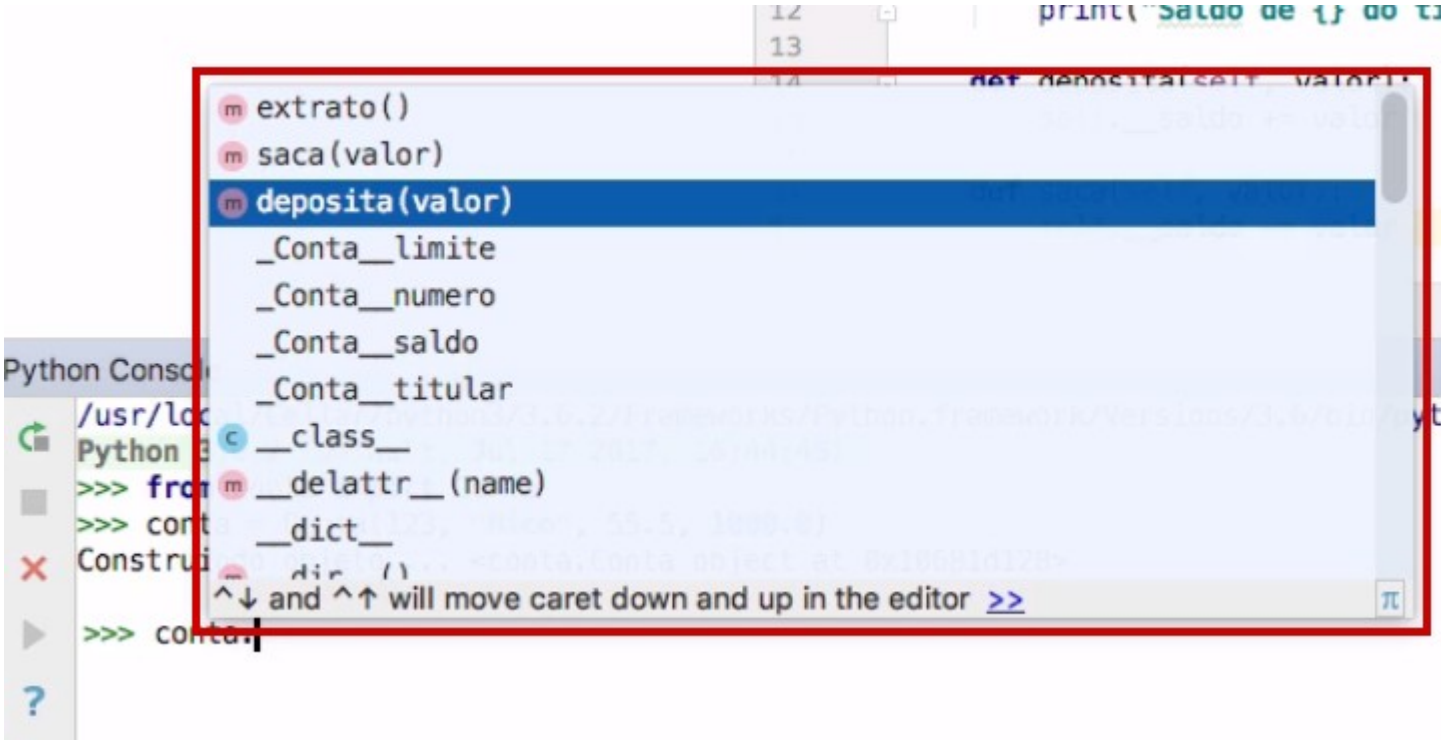
Não podemos acessar o atributo `saldo` do objeto diretamente. Teremos que usar os métodos responsáveis por encapsular o acesso ao objeto.
Então, para melhorarmos a classe `Conta`, devemos restringir o acesso a `saldo`, tornando-o privado, adicionando dois caracteres underscore (`__`).

```
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
```

Em algumas linguagens como Java, a palavra private define o atributo como privado e é chamado como modificador de visibilidade. Porém, em Python, foi convencionado o uso `__`. Com isso, nós renomeamos os atributos seguindo uma nomenclatura especial, por exemplo, `numero` passou a se chamar `__numero`.
Para testar as alterações, reiniciaremos o console e criaremos um novo objeto.

```
>>> from conta import Conta
>>> conta = Conta(123, "Nico", 55.5, 1000.0)
Construindo objeto ... <conta.Conta object at 0x10f6f5630>
```

Tudo continua funcionando corretamente. Em seguida, tentaremos acessar o atributo referente ao saldo. Se você observar, quando digitarmos a referência conta no console do Pycharm, o autocomplete já nos oferecerá opções diferentes.



Primeiramente, serão listados três métodos (extrato, saca e deposita), depois, vemos _Conta_limite, _Conta_numero, _Conta_saldo, _Conta_titular. Vamos tentar acessar os atributos __limite e __saldo.

```
>>> conta._Conta__limite
1000.0
>>> conta._Conta__saldo
55.5
```

Nós continuamos a ter acesso aos atributos, ainda que eles tenham mudado de nome — o Python adicionou a classe antecedido por _. Ao escrevermos conta._Conta__limite, o Python informará ao desenvolvedor que o atributo __saldo não deve ser acessado.
O Python avisa que o atributo foi criado para ser usado dentro da classe, por meio dos métodos. Porém, continuaremos a ter acesso aos valores. Mas se o desenvolvedor decidir acessar o atributo igualmente, ele será alertado de que está fazendo algo inapropriado, ou seja, está "brincando com fogo".
A ação de tornar privado o acesso aos atributos, no mundo Orientado a Objetos, chamamos de encapsulamento. Com isso, definimos que o acesso deve ocorrer apenas por meio dos métodos. A seguir, falaremos mais sobre encapsulamento, mas antes faça os exercícios.

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.

Nós escrevemos a funcionalidade de transferir dinheiro de uma conta para outra, fora da classe. Depois, por uma questão de organização, decidimos colocar a refatoração dentro da classe Conta, por ser um trecho relacionado a conta. No entanto, existem casos em que percebemos, na elaboração do código, que determinadas partes se encaixam em algumas classes específicas.

Lembrem-se que um código bom costuma ser melhorado ao longo da sua criação e a refatoração faz parte do dia a dia do programador. Assim como existem códigos que primeiro criamos em um lugar e incluímos em outra classe depois, o contrário existe também.

Por exemplo, se trabalhássemos com o método eh_inadimplente(), cuja responsabilidade é identificar se alguém é inadimplente, dando como retorno true (verdadeiro) e false (falso). Passaremos como parâmetro cliente, porque quem é inadiplente é a pessoa e não a conta.

```
def eh_inadimplente(self, cliente):
```

Se neste método, não for utilizado os dados da conta, seria correto extrair a funcionalidade e movê-la para outro lugar, por exemplo, para a classe `cliente` que poderia ser criada. Sempre deveríamos verificar se o método está no local mais apropriado.
Idealmente, uma classe deve ter apenas uma responsabilidade. Se adicionássemos o método `eh_inadimplente`, `Conta` passaria a ter duas funções. E, provavelmente, começaríamos a trabalhar com primeiro e segundo nome do titular, ou talvez, precisaríamos especificar o número da agência. Ou seja, os dados seriam mais detalhados.
Neste caso, se desenvolvêssemos o método referente aos clientes inadimplentes, faltaria coesão na nossa classe, por ter mais responsabilidades do que deveria.
Em seguida, removeremos o método `eh_inadimplente()` que foi criada apenas como exemplo.

Atualmente, conseguimos mudar o valor dos atributos da nossa classe. Por exemplo, conseguimos mudar o saldo da conta simplesmente atribuindo um novo valor a ele:

```
>>> from conta import Conta
>>> conta = Conta(123, "Nico", 55.5, 1000.0)
Construindo objeto ... <conta.Conta object at 0x7fa29f59a518>
>>> conta.saldo = 10000
>>> conta.extrato()
Saldo de 10000 do titular Nico
```

Mas o saldo da conta só deve ser modificado através dos métodos `deposita` e `saca`. Então, para avisar ao desenvolvedor que ele não deve alterar o valor dos atributos acessando-os diretamente, torne-os privados, adicionando dois underscores à frente dos atributos, por exemplo:

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

Não esqueça de também modificar os atributos nos métodos.

Transferindo um valor de uma conta para outra
Para transferir um valor de uma conta para outra, crie o método `transfere`, que recebe como argumento uma referência do próprio objeto, o `valor` a ser transferido, e a conta de `destino`. Esse método sacará o valor da conta atual e o depositará na conta de destino.

O código da classe `Conta` ficará assim:

```
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

    def extrato(self):
        print("Saldo de {} do titular {}".format(self.__saldo, self.__titular))

    def deposita(self, valor):
        self.__saldo += valor

    def saca(self, valor):
        self.__saldo -= valor

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

No Python Console, dentro do próprio PyCharm, teste o código, crie duas contas e transfira um valor de uma conta para outra, visualizando os seus extratos em seguida, por exemplo:

```
>>> from conta import Conta
>>> conta = Conta(123, "Nico", 55.5, 1000.0)
Construindo objeto ... <conta.Conta object at 0x7f82af89d048>
>>> conta2 = Conta(321, "Marcos", 100.0, 1000.0)
Construindo objeto ... <conta.Conta object at 0x7f82af89d400>
>>> conta.transfere(10.0, conta2)
>>> conta.extrato()
Saldo de 45.5 do titular Nico
>>> conta2.extrato()
Saldo de 110.0 do titular Marcos
```

Falamos nessa aula sobre a coesão que é ligado ao principio de responsabilidade única. Aprendemos que uma classe deve ter apenas uma responsabilidade (ou deve ter apenas uma razão para existir). Em outras palavras, ela não deve assumir responsabilidades que não são delas.

Além desse princípio de responsabilidade única existem outras que foram definidos através do Robert C. Martin no início dos anos 2000 e são conhecidos pelo acrônimo SOLID:
•S - Single responsibility principle
•O - Open/closed principle
•L - Liskov substitution principle
•I - Interface segregation principle
•D - Dependency inversion principle

Na Alura temos cursos específicos sobre o SOLID, mas fique tranquilo, na medida que você avança no mundo OO esses princípios ficam mais claros e fáceis de se entender.

O que aprendemos?

Nesta aula, aprendemos:

•Atributos privados
•Encapsulamento de código
•Encapsulamento da manipulação dos atributos nos métodos
•Coesão do código

AULA 5

Nós avançamos nos conteúdos apresentados sobre a linguagem e vimos conceitos fundamentais como encapsulamento, coesão e referências. Mostramos como é a criação de objetos, o funcionamento das classes, métodos e atributos.

No entanto, quando falamos de atributos privados — que não são verdadeiramente privados —, ao adotarmos a nomenclatura especial adicionando __, geramos um pequeno problema.
Vamos recriar no console, a conta do Nico:

```
>>> from conta import Conta
>>> conta = Conta(123, "Nico", 55.5, 1000.0)
Construindo objeto ... <conta.Conta object at 0x10f6f5630>
```

Explicamos que não deveríamos poder modificar o valor do saldo diretamente. Apesar de podermos usar `_Conta__saldo`, o desenvolvedor é avisado de que não deveria fazer isso, porque o atributo é privado. Porém, como imprimiremos o `extrato()`? Quando executamos `conta.extrato()`, o retorno será uma string formatada. No entanto, como nosso objetivo é acessar apenas o saldo, a solução será escrever um método que imprima unicamente o saldo, abaixo de `transfere()`.

```
def pega_saldo(self):
    return self.__saldo
```

Criamos um método com uma responsabilidade, que retorna o saldo. Poderíamos criar um método semelhante para retornar `__titular`.

```
def devolve_titular(self):
```

```
    return self.__titular
```

Ou para identificarmos __limite.

```
def retorna_limite(self):
    return self.__limite
```

Com o uso do return, sempre nos será retornado o valor de um atributo. Porém, o design do código está cheirando mal. Vamos testar se o código funciona.

```
>>> from conta import Conta
>>> conta = Conta(123, "Nico", 55.5, 1000.0)
Construindo objeto ... <conta.Conta object at 0x10adfd2b0>
>>> conta.pega_saldo()
55.5
>>> conta.devolve_titular()
'Nico'
>>> conta.retorna_limite()
1000.0
```

Escrevemos métodos específicos que nos devolvem os dados solicitados. É comum utilizarmos funcionalidades como estas para gerar relatórios, que nos mostre os dados principais da conta. Por serem recorrentes, existe uma nomenclatura padrão para esses métodos: getters (que nos dão um dado). Ou seja, a forma mais apropriada de nomear os métodos seria usando o nome get.

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


def get_saldo(self):
    return self.__saldo


def get_titular(self):
    return self.__titular
```

O uso do getters é um dos primeiros conceitos aprendidos pelos desenvolvedores Java. Além desses métodos usados apenas para retornar, existem aqueles que modificam. No caso, falamos dos setters. Nós já temos métodos para acessar saldo, mas ainda temos que criar as formas de trabalhar com limite. O objetivo é podermos aumentar o limite por meio de set_limite().

```
conta.set_limite(10000.0)
```

Este é o método com que definiremos um novo limite. A seguir, vamos definir o método set_limite(), para o qual, além do self, passaremos limite como parâmetro:

```
def set_limite(self, limite):
    self.__limite = limite
```

Lembrem-se que com set nunca retornaremos um valor, nós iremos modificar um atributo. Agora, colocaremos um novo limite no atributo __limite e testaremos no console.

```
>>> from conta import Conta
>>> conta = Conta(123, "Nico", 55.5, 1000.0)
Construindo objeto ... <conta.Conta object at 0x10d92c160>
>>> conta.get_limite()
1000.0
>>> conta.get_saldo()
55.5
>>> conta.get_titular()
'Nico'
>>> conta.set_limite(1000.0)
```

Como explicamos, o set_limite() apenas altera, então, get_limite nos informará se o saldo foi atualizado.

```
>>> conta.get_titular()
'Nico'
>>> conta.set_limite(1000.0)
>>> conta.get_limite()
1000.0
```

> Importante: Usem os getters e setters com parcimônia. Eles devem ser criados apenas quando forem necessários.

O método set_limite() é útil, porque o limite pode ser alterado dentro do contexto de negócio. Mas, por exemplo, o número da conta de um cliente não deve mudar. Neste caso, é inapropriado implementarmos set_numero. Se o cliente encerrar uma conta e, depois, quiser abrir uma outra, ele receberá um novo número. Mas trata-se de um número fixo.
Temos que ficar atentos para evitar criar funcionalidades inutilmente. Mais adiante, conheceremos uma alternativa para esses métodos getters e setters. Aproveite para praticar com os exercícios.

Falamos sobre o uso dos getters e setters, no mundo de Orientação a Objetos. Com eles podemos obter ou alterar um valor específico do nosso objeto.

Usamos como exemplo os métodos get_saldo(), get_titular(), get_limite(), set_limite(), mas o único que utilizaremos no nosso código é set_limite.
Atualmente, o arquivo conta.py está assim:

```
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

    def extrato(self):
        print("Saldo de {} de titular{}".format.saldo, self._titular))

    def deposita(self, valor):
        self.__saldo += valor

    def saca(self, valor):
        self.__saldo -= valor

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

    def get_saldo(self):
        return self._saldo

    def get_titular(self):
        return self._titular

    def get_limite(self):
        return self._limite

    def set_limite(self, limite):
        self._limite = limite
```

Nós evitamos a criação de métodos como set_numero, porque uma conta não deve mudar de número. Observe que não criamos o método set_saldo, considerando que o total do saldo também muda. Evitamos fazer isso, porque temos métodos de mais alto nível e mais expressivos, como transfere(), para realizar esse tipo de alteração:

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

A seguir, mostraremos uma sintaxe alternativa para sintaxe dos getters, para isto, criaremos a classe Cliente, no arquivo cliente.py. Lembrando que não vamos inventar funcionalidades desnecessárias ou que terão utilidade apenas no futuro.

> Existe uma expressão em inglês conhecida na Engenharia de software que é: "You Ain't Gonna Need It" (YAGNI, abreviada). Trata-se de uma orientação para que programadores evitem criar funcionalidades para o código fonte de um programa até que estas sejam necessárias.

```
class Cliente:


    def __init__(self, nome):
        self.nome = nome
```

Incluímos no início da classe __init__(). Por enquanto, sabemos que os parâmetros necessários serão self e nome. Para o atributo nome, atribuímos o parâmetro nome. No console, criaremos um novo cliente

```
>>> from cliente import Cliente
>>> cliente = Cliente("Marco")
>>> cliente
<cliente.Cliente object at 0x10b114f28>
```

O construtor da classe Cliente recebe o nome do cliente Marco.
Quando criamos o método __init__() não usamos a sintaxe __, adotada pelo Python. Desta forma, o desenvolvedor consegue facilmente acessar o atributo apenas usando a referência. O atributo nome, conseguimos alterar.

```
>>> cliente.nome = "Nico"
>>> cliente.nome
'Nico'
```

Agora já podemos pensar em criar uma classe que será aproveitada por outras relacionadas a Cliente. Provavelmente, precisaremos do método get para validar o dado do atributo nome, talvez, para garantir que o nome do titular comece com a letra maiúscula. Por exemplo, no caso de atribuirmos o nome nico, com a primeira letra minúscula.
Quando acessarmos o atributo nome de cliente, queremos que seja executado o método title(). Desta forma, o resultado continuará sendo Nico, porque o atributo recebeu o tratamento do get_nome().
Vamos alterar o método que passará a se chamar nome(), no entanto, isso ainda não será o suficiente. No console, precisaremos dos parênteses para que o método seja executado. Mas nosso objetivo é que a execução ocorra, mesmo sem os parênteses.
Na linguagem Python, os métodos que dão acesso são nomeados como properties. Desta forma, indicaremos para o Python nossa intenção de ter acesso ao objeto.
A declaração de uma property é feita com o uso do caractere @.

```
@property
```

Com isto, indicamos que este método representa uma propriedade — um termo já recorrente em outras linguagens, como Delphi e C#. Com @property, indicamos que estamos trabalhando com uma propriedade. Faremos isso com o método nome().

```
class Cliente:
```

```
    def __init__(self, nome):
        self.nome = nome

    @property
    def nome(self):
        return self.nome.title()
```

Agora, quando digitarmos nos console `cliente.nome`, sem a adição dos parênteses, e conseguiremos que o método seja executado como antes.
Para explicitarmos que `nome()` está sendo executado por baixo dos panos, imprimiremos a mensagem `chamando @property nome()`, adicionando um `print()` ao método. Também tornaremos privado o atributo `nome` que será antecedido por `__`.

```
class Cliente:

    def __init__(self, nome):
        self.__nome = nome

    @property
    def nome(self):
        print("chamando @property nome()")
        return self .__nome.title()
```

Se esquecermos de adicionar `__` ao atributo `nome` e torná-lo privado, receberemos uma mensagem de erro quando tentarmos acessá-lo no console. Após as alterações no código, vamos fazer testes no console.
Começaremos criando a referência `cliente` para a conta do `nico` (com a letra minúscula).

```
>>> cliente = Cliente("nico")

>>> cliente.nome
chamando @property nome()
'Nico'
```

A maneira como escrevemos no console, parece que estamos acessando diretamente o atributo, porém o método `nome()` foi chamado. Começamos a classe de forma bastante simples, apenas com a função inicializadora, depois, sentimos a necessidade de criar o getter. No caso, optamos em incluir `@property` para continuarmos com a mesma sintaxe do atributo, mas com o método sendo executado internamente.
Da mesma forma como fazemos isso para um getter, faremos para um setter. Novamente, criaremos um método de `nome()`, logo abaixo da propriedade do getter:

```
def nome(self, nome):
```

```
    print("chamando setter nome()")
    self.__nome = nome
```

Criamos um setter sem a adicionação do `set` antes do nome do método. Mas para que ele funcione, teremos que adicionar também uma configuração: `@nome.setter`.

```
@nome.setter
def nome(self, nome):
    print("chamando setter nome()")
    self.__nome = nome
```

Especificamos qual atributo receberá o setter.

Testaremos no console para garantirmos que a nossa sintaxe simplificada está funcionando.

```
>>> from cliente import Cliente
>>> cliente = Cliente("nico")
>>> cliente.nome
```

Se tentarmos mudar o atributo `nome` para `marco`, o método será executado mesmo sem o uso dos parênteses.

```
>>> from cliente import Cliente
>>> cliente = Cliente("nico")
>>> cliente.nome = "marco"
chamando setter nome()
>>> cliente.nome
chamando @property nome()
'Marco'
```

A seguir, acessaremos o arquivo `conta.py` e vamos trabalhar com o método `get_limite()`, adicionando `@property`. Agora não precisaremos mais da palavra `get`.

```
def get_titular(self):
    return self._titular


@property
def limite(self)
    return self.__limite


@limite.setter
def limite(self, limite):
    self.__limite = limite
```

Observem que retiramos o `get` do `get__limite` e `set` do `set__limite`. Temos a opção de fazer a mesma alteração com outros métodos, mas faremos isso mais adiante nos exercícios. A seguir, tentaremos acessar os dados de uma conta.

```
>>> from conta import Conta
>>> conta = Conta(123, "Nico", 55.5, 1000.0)
Construindo objeto ... <conta.Conta   o bject at 0x1019df3c8>
>>> conta.limite
1000.0
```

A execução de conta.limite é semelhante a de getter. Podemos utilizar o setter também:

```
>>> conta.limite = 2000.0
>>> conta.limite
2000.0
```

Na linha em que escrevemos conta.limite = 2000.0 pareceu que estávamos atribuindo, mas estávamos na realidade executando setter.
Temos ferramentas suficientes para trabalhar nos exercícios, falamos sobre as properties, que colaboram para manter a nossa sintaxe amigável e chamam por baixo dos panos os métodos get e set.

Na aula anterior, foi visto que quando há dois underscores à frente de um atributo, não deve-se acessá-lo diretamente. Para ler um atributo, cria-se um getter para ele, e para modificar um atributo, cria-se um setter para ele.
Então, crie os getters para os atributos saldo, titular e limite. Por exemplo, o getter do atributo saldo ficará assim:

```
def get_saldo(self):
    return self.__saldo
```

Crie também o setter para o limite, por exemplo:

```
def set_limite(self, limite):
    self.__limite = limite
```

O código da classe Conta ficará assim:

```
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


    def extrato(self):
        print("Saldo de {} do titular {}".format(self.__saldo, self.__titular))
```

```
    def deposita(self, valor):
        self.__saldo += valor

    def saca(self, valor):
        self.__saldo -= valor

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

    def get_saldo(self):
        return self.__saldo

    def get_titular(self):
        return self.__titular

    def get_limite(self):
        return self.__limite

    def set_limite(self, limite):
        self.__limite = limite
```

Agora, quando quisermos ler os atributos `saldo`, `titular` e `limite`, basta chamar os métodos `get_saldo`, `get_titular` e `get_limite`, respectivamente. E se quisermos alterar o valor do atributo `limite`, que é o único atributo da nossa classe que geralmente é modificado, basta chamar o método `set_limite`, passando como parâmetro o novo limite da conta.

Agora que vimos as propriedades, crie-os no lugar dos getters da classe `Conta`. Por exemplo, no lugar do `get_saldo`, teremos:

```
@property
def saldo(self):
    return self.__saldo
```

Da mesma forma, crie uma propriedade para o setter do atributo `limite`:

```
@limite.setter
def limite(self, limite):
    self.__limite = limite
```

O código da classe `Conta` ficará assim:

```
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

    def extrato(self):
        print("Saldo de {} do titular {}".format(self.__saldo, self.__titular))

    def deposita(self, valor):
        self.__saldo += valor

    def saca(self, valor):
        self.__saldo -= valor

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

    @property
    def saldo(self):
        return self.__saldo

    @property
    def titular(self):
        return self.__titular

    @property
    def limite(self):
        return self.__limite

    @limite.setter
```

```
    def limite(self, limite):
        self.__limite = limite
```

No Python Console, dentro do próprio PyCharm, teste o código, crie uma conta e acesse o valor de algum atributo, utilizando somente o seu nome, por exemplo:

```
>>> from conta import Conta
>>> conta = Conta(123, "Nico", 55.5, 1000.0)
Construindo objeto ... <conta.Conta object at 0x7f82af89d048>
>>> conta.limite
1000.0
```

Do mesmo jeito, altere o valor do atributo `limite`, deve funcionar como se você estivesse acessando-o diretamente:

```
>>> from conta import Conta
>>> conta = Conta(123, "Nico", 55.5, 1000.0)
Construindo objeto ... <conta.Conta object at 0x7f82af89d048>
>>> conta.limite = 2000.0
>>> conta.limite
2000.0
```

O que aprendemos?
Nesta aula, aprendemos:

•Métodos de leitura dos atributos, os getters
•Métodos de modificação dos atributos, os setters
•Propriedades

AULA 6

Falamos sobre propriedades, como escrever getters e setters de maneira mais elegante. Na resolução dos exercícios, transformamos os dois getters em propriedades adicionando um `@property` em cada, evitando o uso do `get` e `set` .

```
@property
def saldo(self):
    return self.__saldo


@property
def titular(self):
    return self.__titular
```

```
@property
def limite(self):
    return self.__limite

@limite.setter
def limite(self, limite):
    self.__limite = limite
```

No console, executaremos um exemplo de como criamos uma nova `conta` e conseguimos executar o método `saldo()`:

```
>>> from conta import Conta
>>> conta = Conta(123, "Nico", 55.5, 1000.0)
Construindo objeto ... <conta.Conta object at 0x110839668>
>>> conta.saldo
55.5
```

A seguir, continuaremos falando sobre a classe `Conta`, mas focando no método `saca()`:

```
def saca(self, valor):
    self.__saldo -= valor

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

Se analisarmos o `saca()`, veremos que ele tem alguns problemas. A conta do `Nico` tem um saldo de `55.5` e um limite de `10000.0`. Qual o valor máximo de saque que podemos fazer? Teoricamente, só poderíamos sacar `1055.5`. Mas é possível fazer uma malandragem e sacar mais:

```
>>> conta. saca(1200.0)
>>> conta.saldo
-1144.5
```

O saldo negativo ultrapassou o limite de `1000.0`, ou seja, não existe uma verificação. É o que faremos a seguir.

Nós queremos verificar se existe dinheiro suficiente na conta para que seja realizado o saque, ou seja, a soma do saldo com o limite, deve ser maior do que o valor que sacaremos. Para isto, usaremos `if/else` para fazer isso no método.

```
def saca(self, valor):
    if(valor <= (self._saldo + self.__limite)):
        self.__saldo -= valor
    else:
        print("O valor {} passou o limite".format(valor))
```

No console, vamos testar o código criado, chamando o método saca().

```
>>> from conta import Conta
>>> conta = Conta(123, "Nico", 55.5, 1000.0)
Construindo objeto ... <conta.Conta object at 0x10c662cc0>


>>> conta.saca(1200.0)
O valor 1200.0 passou o limite
```

Agora quando tentamos sacar 1200.0, o Python nos informa que ultrapassamos o limite. Se verificarmos o saldo, veremos que ele ainda é o mesmo, porque o saque não foi realizado.
Nós executamos o método saca() e apenas realizamos a verificação se o valor que desejamos sacar é <= ao valor do saldo. Isto deveria ficar um pouco mais expressivo no nosso código. Nós criamos uma regra simples, mas o sistema ainda ficará mais complexo, por isso, nós queremos deixar o código mais expressivo.
O próximo passo será adicionar o novo método pode_sacar(). Em seguida, moveremos a expressão que está localizada atualmente no if, iremos movê-la para o pode_sacar(). E o if de saca() passará a ser o responsável por chamar pode_sacar().

```
def pode_sacar(self):
    pass


def saca(self, valor):
    if(self.pode_sacar(valor)):
        self.__saldo -= valor
    else:
        print("O valor {} passou o limite".format(valor))
```

Com o nosso código mais expressivo, ele se torna mais fácil de entender. Continuaremos trabalhando no método pode_sacar(), passando como segundo parâmetro a variável valor_a_sacar. Já o retorno da função será a condição que antes estava no if.

```
def pode_sacar(self, valor_a_sacar):
    return valor_a_sacar <= (self.__saldo + self.__limite)
```

Para tornar o método mais expressivo, vamos colocar a condição dentro de outra variável:

```
def pode_sacar(self, valor_a_sacar):
    valor_disponivel_a_sacar = self.__saldo + self.__limite
    return valor_a_sacar <= valor_disponivel_a_sacar


def saca(self, valor):
    if(self.pode_sacar(valor)):
        self.__saldo -= valor
    else:
```

```
        print("O valor {} passou o limite".format(valor))
```

Testaremos se está tudo funcionando.

```
>>> from conta import Conta
>>> conta = Conta(123, "Nico", 55.5, 1000.0)
Construindo objeto ... <conta.Conta object at 0x110091358>


>>> conta.saca(1200.0)
O valor 1200.0 passou o limite
>>> conta.saldo
55.5
```

Se tentarmos sacar 1200.0, o valor não passará pela condição e ele retornará uma mensagem avisando isso. Para confirmarmos, pedimos o saldo e vimos que continua 55.5. Podemos testar fazer um saque de um valor que esteja dentro da condição.

```
>>> conta.saca(100.0)
>>> conta.saldo
-44.5
```

Tudo continua funcionando da mesma forma, mas agora tivemos um retorno negativo. Em seguida, vamos testar pode_sacar(), que tem o retorno true ou false.

```
>>> conta.pode_sacar(100.0)
True
```

O método pode_sacar() facilita a compreensão do if. Porém, ele não deve ser usado desta forma. O pode_sacar() deve ter um aviso explícito de que o mesmo só poderá estar disponível dentro da classe. Precisamos alertar o desenvolvedor de que o método é privado e isso feito adicionando __.

```
def __pode_sacar(self, valor_a_sacar):
    valor_disponivel_a_sacar = self.__saldo + self.__limite
    return valor_a_sacar <= valor_disponivel_a_sacar


def saca(self, valor):
    if(self.__pode_sacar(valor)):
        self.__saldo -= valor
    else:
        print("O valor {} passou o limite".format(valor))
```

O método __pode_sacar() foi criado para ser executado apenas dentro da classe, por isso, o caractere underscore foi adicionado dentro do if também.
Se tentarmos executar pode_sacar() diretamente no console, receberemos uma mensagem de erro.

```
>>> conta = Conta(123, "Nico", 55.5, 1000.0)
Construindo objeto ... <conta.Conta object at 0x10159c898>
>>> conta.pode_sacar(100.0)
```

```
Traceback (most recent call last):
  File "<input>", line 1, in <module>
AttributeError: 'Conta' object has no attribute 'pode_sacar'
```

O método pode_sacar já não existe mais. Ele mudou de nome, passando a se chamar _Conta__pode_sacar(). Temos o nome da classe Conta, assim como os atributos privados. O Python permite que o método seja invocado, mas recomenda ao desenvolvedor que evite o __pode_sacar(), que é privado.
Assim como existem métodos privados, temos atributos que seguem a nomenclatura especial.

Vamos conhecer mais um recurso oferecido pelas classes. Anteriormente, falamos sobre os métodos privados — com o uso do underscore (_), sinalizamos para o desenvolvedor quais são eles.

A seguir, imagine que estamos criando um sistema para o Banco do Brasil, no qual todas as contas baseadas nesta classe são referentes ao banco. Geralmente, cada instituição financeira tem um código associado. O código referente ao Banco do Brasil é 001. Então, dentro do método __init__ de Conta, adicionaremos mais um atributo:

```
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
        self.__codigo_banco = "001"
```

Criamos o atributo privado __codigo__banco, com o valor fixo. Quando instanciarmos um objeto, automaticamente, será inserido o código do Banco do Brasil. Para acessarmos um atributo, criaremos um método codigo_banco(), abaixo de @limite.setter. Acima do novo método, incluiremos @property, para que ele possa ser executado sem o parênteses.

```
@property
def codigo_banco(self):
    return self.__codigo_banco
```

No console, vamos digitar:

```
>>> from conta import Conta
>>> conta = Conta(123, "Nico", 55.5, 1000.0)
Construindo objeto ... <conta.Conta object at 0x10d670208>


>>>  conta.codigo_banco
'001'
```

Conseguimos executar codigo_banco sem utilizarmos (). Com isso, obtivemos o retorno 001.
Criaremos outro objeto, agora, referente ao cliente Marco.

```
>>> conta = Conta(321, "Marco" 55.5, 1000.0)
Construindo objeto ... <conta.Conta object at 0x10d43fd30>
>>> conta.codigo_banco
'001'
```

Novamente, teremos o mesmo retorno. Em seguida, vamos reiniciar o console e importar a Conta.

```
>>> from conta import Conta
```

Neste momento, o objeto ainda não foi criado baseado na classe Conta. Se quisermos saber qual é o código do banco, precisamos criar o objeto primeiro. Porém, faria sentido já termos acesso ao código do banco, porque é algo comum entre as contas — uma informação que deveria estar disponível, mesmo antes da criação da conta. O código do banco não depende do objeto.
Então, nosso próximo objetivo é acessar codigo_banco, sem ter o objeto criado. No momento, se tentarmos fazer isso, teremos o seguinte resultado.

```
>>> conta.codigo_banco
<property object at 0x10db42e58>
```

Os métodos que estamos trabalhando fazem parte da classe e o objeto é representado pelo self. Nós queremos chamar o método codigo_banco(), sem a inclusão do objeto, por isso, já podemos remover o self:

```
@property
def codigo_banco():
    return self.__codigo_banco
```

Esse métodos que conseguimos chamar sem uma referência recebem o nome de estáticos, porque eles fazem parte da classe. Todas as linguagens orientadas a objeto trabalham com métodos estáticos, mas para que eles sejam utilizados, iremos configurar os métodos. Fica inapropriado usar property, porque ele sempre precisa do self. A configuração correta será @staticmethod.

```
@staticmethod
def codigo_banco():
    return "001"
```

Em seguida, vamos apagar o atributo __codigo_banco dentro do __init__.

```
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
```

Agora se testarmos no console, nosso código funcionará corretamente:

```
>>> from conta import Conta
>>> Conta.codigo_banco()
'001'
```

Observe que nenhum objeto foi criado, mas conseguimos chamar o método estático. Nós especificamos o nome da classe, e depois de acessá-la, chamamos o método.

O próximo passo será criar o passo que devolve todos os códigos dos bancos. Usaremos uma lista com o código de três bancos:

```
{'BB': '001', 'Caixa': '104', 'Bradesco':'237'}
```

Usaremos esse dicionário dentro do `codigos_bancos()`.

```
@staticmethod
def codigos_bancos():
    return {'BB': '001', 'Caixa': '104', 'Bradesco':'237'}
```

Esses códigos foram adicionados apenas como exemplo, mas vocês não precisam conhecê-los. No `return` do método, incluiremos chaves e valores.

```
>>> from conta import Conta
>>> codigos = Conta.codigos_bancos()
>>> codigos
{'BB': '001', 'Caixa': '104', 'Bradesco':'237'}
```

Colocamos a chamada para o método dentro de uma variável. Outra maneira de acessarmos um código específico é por meio de colchetes ([]):

```
>>> codigos['BB']
'001'
>>> codigos['Caixa']
'104'
```

Nosso foco está nos métodos estáticos que são da classe, e mesmo sem o objeto, conseguimos executar o método. Em algumas situações isso pode ser útil. Porém, precisamos ser cautelosos com o uso dos métodos estáticos. A ideia do mundo OO é criar objetos. Se usarmos apenas a classe `Conta`, sem ter um objeto, deixaremos de trabalhar com Orientação a Objeto.
Quando todos os objetos compartilham algo em comum, faz sentido usar esses métodos — como no exemplo em que compartilhamos todos os códigos do banco. Mas se utilizarmos apenas métodos estáticos, não utilizaremos mais objetos e nos aproximaremos do mundo procedural.

Vimos alguns conceitos que podem ser praticados com os exercícios sobre métodos estáticos. Continuamos a seguir.

Avançamos bastante no conteúdo do curso, mas vale ressaltar que o paradigma OO não é uma exclusividade da linguagem Python. Orientação a Objetos é um dos paradigmas mais utilizados entre as linguagens de programação.

Existem linguagens que continuam sendo procedurais, como linguagem C, assim como outros paradigmas funcionais. Inclusive, em alguns casos, os dois começam a misturar.

É possível afirmar que o paradigma OO domina o mercado de desenvolvimento.

Isto significa que se você tem uma boa base de OO vista no curso de Python, também já terá aprendido sobre Java, PHP, C++, além de outras linguagens que seguem o mesmo paradigma.

Podemos perceber isso, comparando o arquivo `conta.py` e o `Conta.java`. A diferença entre os dois são os detalhes da sintaxe, mas o paradigma é o mesmo. Por exemplo, os dois terão uma classe `Conta`, que em Java está assim:

```
class Conta {

   //atributos
   private int numero;
   private String titular;
   private double saldo;
   private double limite;

   //construtor
   Conta(int numero, String titular, double saldo, double limite) {
      this.numero = numero;
      this.titular = titular;
      this.saldo = saldo;
      this.limite = limite;
   }
}
```

Enquanto a classe no Python está:

```
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
```

A diferença na sintaxe é que o primeiro usa chaves (`{}`), enquanto o segundo usa dois pontos (`:`). No Python, nós começamos com a função `__init__`, que é a função construtora. No entanto, no Java, não basta apenas adicionarmos o construtor, temos que definir os atributos antes.
Mas nas duas sintaxes temos uma função ocupando o papel de construtor. No Python, ela vai receber o nome de `__init__`, no Java, ela terá o mesmo nome da classe.

> Quem tem conhecimentos sobre Java, sabe que o seu construtor também se chama <init>.

No entanto, a principal diferença é a parte superior no arquivo Conta.java, em que precisamos definir os atributos explicitamente e especificamos que eles são privados. No conta.python, usamos a convenção __ para fazer o mesmo. Tanto no Python, quanto no Java, existem formas de acessar um atributo, mesmo que definindo como privado.
Se seguirmos comparando os dois arquivos, veremos que ambos têm o método extrato(), mas o Java não receberá self.

```
//metodos
void extrato() {
    System.out.println("Saldo de " + this.saldo)
}
```

Mas também existe o self no mundo Java: this, que é disponibilizado implicitamente, mesmo não estando declarado. Veremos que o uso do . também é correspondente, assim como os métodos privados.
Vemos em conta.py, que definimos as propriedades para acessar o saldo. No mundo Java, escrevemos um método para cada ação, como foi feito no getSaldo() de Conta.java — que precisa do getTitular(), getNumero() e getLimite().

```
public double getLimite() {
    return limite;
}

public void setLimite(double limite) {
    this.limite = limite;
}

public double getNumero() {
    return numero;
}

public void getTitular() {
    return titular;
}

public double getSaldo() {
    return saldo;
}
```

E da mesma forma como existem métodos estáticos no mundo Python, existe no mundo Java:

```
public static String codigo() {
    return "001"
}
```

Observe que usamos a palavra-chave static. Mesmo conhecendo apenas Python, conseguimos entender a lógica do código Java. Como criar um objeto baseado em uma conta Java?

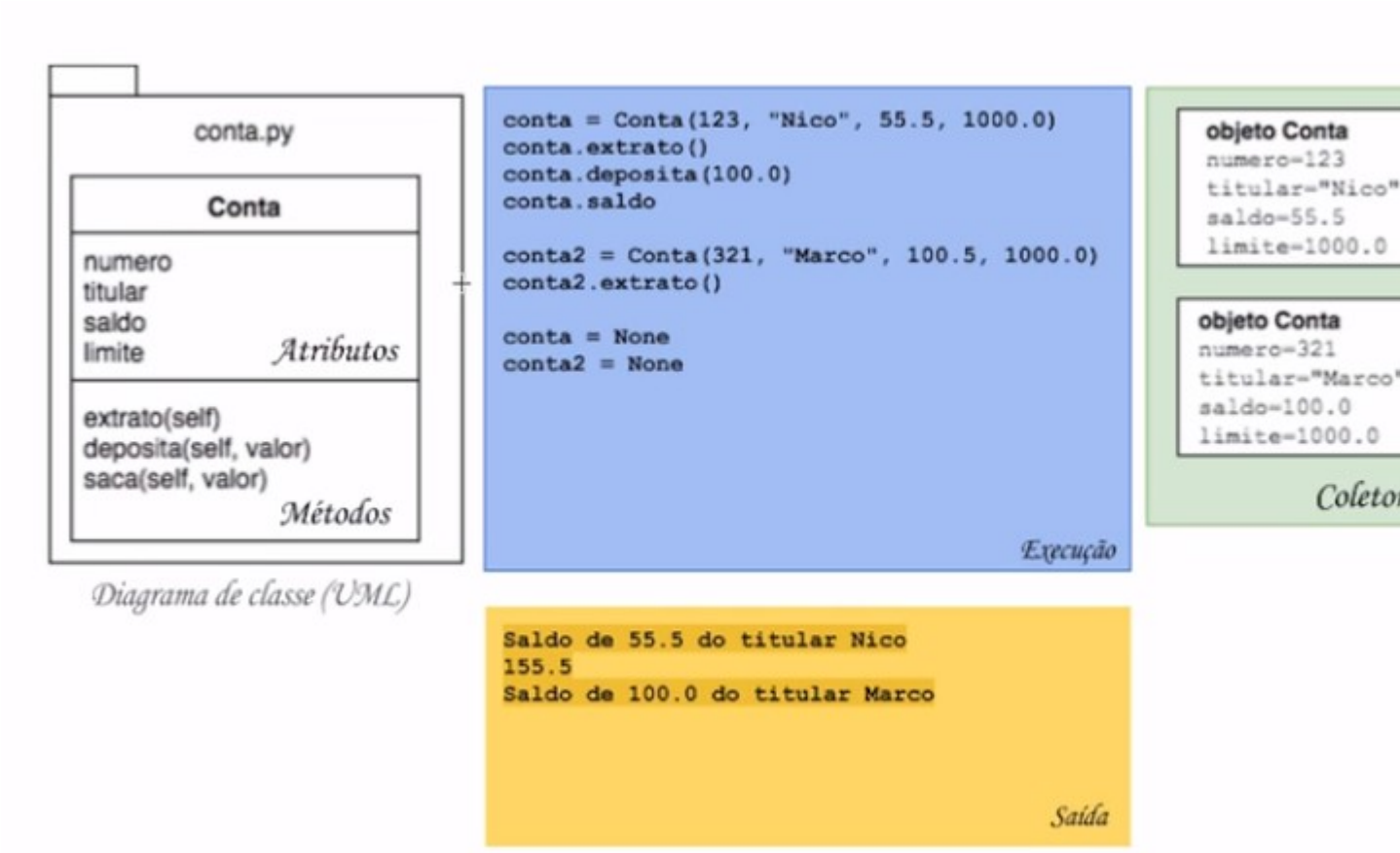
```
Conta contaDoNico = new Conta(123, "Nico", 55.5, 1000.0);
contaDoNico.deposita(100.0);
```

Nós também usamos o construtor Conta(), passamos os parâmetros, mas adotamos a palavra new. O endereço fica guardado na referência contaDoNico, porém, declaramos também o tipo. Mais acima, definimos cada tipo dos atributos: int, String, double. Em Java, ou utilizamos a tipagem estática, ou definimos o tipo da variável.
Fizemos também a chamada usando a referência contaDoNico para passar a função com o valor. Perceba que quando aprendemos sobre o paradigma Orientado a Objetos, podemos aplicar o conceito em diversas linguagens, porque ele é seguido da mesma forma, mudando apenas as sintaxes específicas de cada linguagem.

A seguir, faremos uma revisão. Mesmo que ache estranho fazer tantas revisões, é importante que você tenha uma boa base sobre Orientações a Objeto. Falaremos em outros cursos sobre outros tópicos, como herança, composição, relacionamentos e conceitos mais avançados.

Mas com a base que desenvolvemos, aprender coisas novas será mais fácil. Vamos relembrar os pontos principais vistos no curso.

Nossa motivação inicial era unir os dados e o comportamento, ou seja, juntarmos os atributos e os métodos. Tudo foi agrupado dentro de uma classe.



A classe é a menor unidade de organização no mundo OO. Colocamos os elementos dentro, com os referentes de atributos e os diferentes métodos. Colocamos dentro da classe, tudo o que está relacionado a ela. No caso, os atributos relacionados com a classe `Conta` são:

- numero
- titular
- saldo
- limite

Os métodos relacionados são:

- extrato(self)
- deposita(self, valor)
- saca(self, valor)

Isto significa que evitaremos colocar na `Conta` o código relacionado com impostos da nota fiscal, porque as duas coisas não estão relacionadas. Seria mais apropriado criar a classe `notaFiscal` e adicionar o código relacionado.
Surge a pergunta: onde os programadores procuram as funcionalidades no seu projeto? A resposta é: na classe relacionada. Desta forma, mantemos o código organizado e separamos as responsabilidades.

Vimos como escrever uma classe no mundo Python, adicionamos um construtor e, além disso, mostramos como ele é executado. Usamos o nome da classe, depois, passamos os parâmetros como vemos no conteúdo do quadro azul do diagrama:

```
conta = Conta(123, "Nico", 55.5, 1000.0)
```

```
conta.extrato()
conta.deposita(100.0)
conta.saldo
```

E o resultado será um objeto:

```
objeto Conta


numero = 123
titular = "Nico"
saldo = 55.5
limite = 1000.0
```

A responsabilidade de criar o objeto fica por conta do Python. Sabemos que um espaço é alocado para representar os atributos e o resultado da execução devolve a referência.

Essa referência sabe onde está guardado o objeto. Tendo esse endereço, podemos interagir com a classe, trabalhando com o objeto. Ou seja, quando escrevemos conta.extrato() vai executar o elemento do extrato. Com ele, acessaremos o objeto por meio do self.
Falamos também que o self é uma referência que sempre assume o valor da referência que fez a chamada. Por exemplo, se a referência é conta, o self será um equivalente na linha em que for utilizado: deposita(self, valor).
No arquivo conta.py, implementamos vários métodos, como extrato() e deposita(), mostramos ainda como tornar um método privado. Ao adicionarmos os dois uderscores (__) como em __pode_sacar(), o desenvolvedor é alertado que só deve utilizá-lo dentro da classe Conta.
Criamos atributos privados usando a mesma nomenclatura, como __numero e __titular.
Os métodos podem crescer e ficar ainda maiores e mais complexos, mas para quem faz as chamadas do deposita(), a quantidade de linhas de um método é irrelevante. Isto ocorre, porque o código está encapsulado:

```
def deposita(self, valor):
    self._saldo += valor
```

Em uma conta da vida real, provavelmente, o método seria mais complexo e seria necessário adicionar verificações antes da realização do depósito.

Quem usa a classe Conta e chama deposita(), usa de alto nível, sem a preocupação com os detalhes da implementação.
No curso, falamos sobre propriedades (properties). Quando digitávamos no console conta.saldo, parecia que acessávamos simplesmente um atributo, porque não usamos parênteses dos métodos. Mas por baixos dos panos, o método anotado com @property. Também temos properties que podem alterar os setters.
Uma classe pode ter diversos objetos. Usando a analogia da receita, ela pode ter diversos elementos. Basta repetir a linha que constrói o objeto, passando os novos valores.

Se temos um novo objeto de memória, teremos uma nova referência que guardará o valor do endereço do objeto.

Mostramos que é possível zerar uma referência, com o uso do None.

```
conta = None
conta2 = None
```

Nós podemos falar que um referência não pode apontar para um objeto e se ela não aponta, guardará o valor None. Isto significa que o objeto criado ficou abandonado, porque a conta estava apontando para o objeto, porém ela foi zerada. Por isso, o objeto ficou perdido. Para casos como esse, o Python tem coletor de lixo, responsável por procurar os objetos que foram criados há muito tempo mas não são mais utilizados no projeto.
Criamos uma classe com vários objetos que podem reaproveitar os métodos com as funcionalidades encapsuladas.

Trabalhamos com um diagrama de classes bem simples, que utiliza a linguagem de notação UML. Vimos que em alguns casos, os métodos não estão relacionados com o objeto. Mostramos um exemplo em que gostaríamos de usar um método antes de ter o objeto, que recebem o nome de static method (método estático).
No entanto, eles devem ser usados com parcimônia. O objetivo da Orientação a objetos é a criação de objetos. Se trabalhamos apenas com métodos estáticos isso não acontece. Mas é um recurso oferecido pelo Python e que pode fazer sentido.

Esclarecemos conceitos de método, atributo, como acessar os métodos e como funciona o uso do self e os atributos privados. Falamos sobre encapsulamento e coesão. Vimos conceitos fundamentais do Paradigma Orientado a Objeto.
No próximo curso, iremos além. Serão apresentadas as peças que faltam do mundo OO. Falaremos sobre associações entre classes e mostraremos como funciona a herança no Python. Abordaremos também conceitos como agregação e composição. Entenderemos quando é mais apropriado utilizar cada tipo de relacionamento.
Outro ponto forte será o tratamento de erro, veremos como nossa aplicação deve se comportar quando acontece algo inesperado, como devem se comportar as ações do código. Mostraremos como descobrir onde está o problema e depurar a aplicação. Tudo isso será apresentado passo a passo.
Temos um prato cheio para o próximo curso, além de tópicos que podemos incrementar no mundo Orientado a Objetos, inclusive, indo além dele.

Você está convidado a continuar na nossa jornada nesta viagem pela Orientação a Objeto. Agradeço que tenha chegado até aqui e te espero no próximo curso.

Nesta aula, aprendemos:

•Métodos privados
•Métodos da classe, os métodos estáticos