Conceitos de Orientação a Objetos em Python

UML

- Antes de criarmos nossa primeira classe em Python, vamos relembrar um pouco de UML que estudamos na aula passada
- Dos vários diagramas disponíveis na UML, vamos utilizar “diagrama de classes” e “diagrama de objetos”
- Possíveis usos:
  - Projeto: desde o rascunho até o projeto detalhado
  - Geração automática de código

UML

DIAGRAMA DE CLASSES

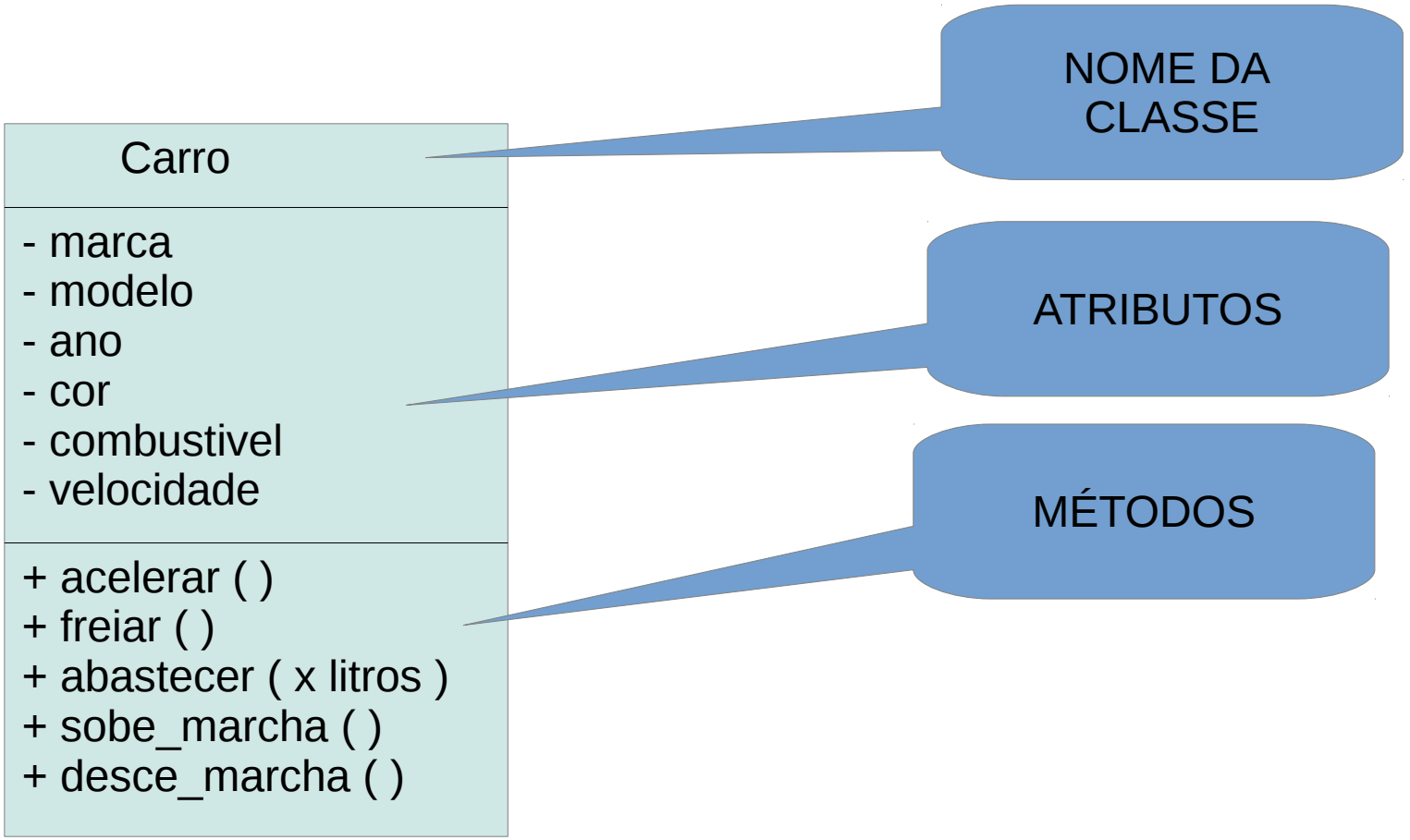
Classe Carro
- marca - modelo - ano - cor - combustivel - velocidade
+ acelerar ( ) + freiar ( ) + abastecer (x litros) + sobe_marcha ( ) + desce_marcha ( )

DIAGRAMA DE OBJETOS

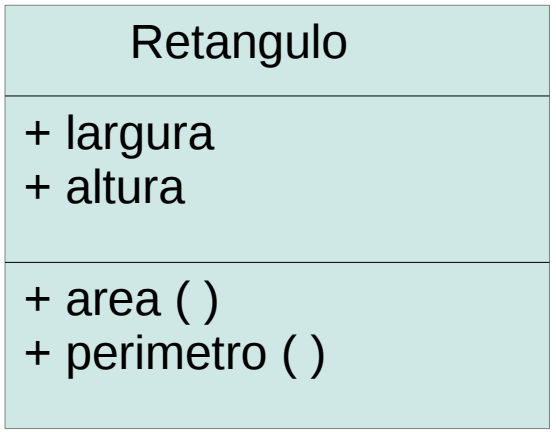
Carro: carro1
- marca: “Ford” - modelo: “Mustang GT Shelby” - ano: 2012 - cor: “vermelho” - combustível: 20 - velocidade: 180

Carro: carro2
- marca: “Chevrolet” - modelo: “Camaro SS” - ano: 2012 - cor: “amarelo” - combustível: 23 -velocidade: 164

UML


Carro
- marca
- modelo
- ano
- cor
- combustivel
- velocidade
+ acelerar ( )
+ freiar ( )
+ abastecer (x litros)
+ sobe_marcha ( )
+ desce_marcha ( )
NOME DA CLASSE
ATRIBUTOS
MÉTODOS

Uma classe bem simples



Declarando classe em Python

- Podemos declarar uma classe “Retângulo” em Python da seguinte maneira:

```
class Retangulo:
    def __init__(self):
        self.largura = 1.0
        self.altura = 2.0

    def area (self):
        return self.largura * self.altura

    def perimetro (self):
        return 2*self.largura + 2*self.altura
```

Dê à classe o nome que você quiser. Geralmente, começa com letra maiúscula, mas não é obrigatório

Esse método __init__ é especial. Chama-se **construtor**!

Atributos aqui!

Métodos aqui! Notem como parece definição de funções! Mas não esqueçam o **self**

Instanciando um objeto

- Com os comandos do slide anterior podemos definir uma classe
- A classe serve como um “molde”, uma “fôrma”, para criar (instanciar) objetos
- Em Python, **instanciamos** um objeto da seguinte forma:

```
nomeDoObjeto = NomeDaClasse( )
```

Instanciando objeto em Python

- Por exemplo:

```
ret1 = Retangulo ( )
ret2 = Retangulo ( )
```

- Com isso definimos **dois** retângulos com dimensões 1 x 1. Podemos mudar essas dimensões mais tarde.

Acessando atributos

- Um atributo *atrib* associado a uma instância *obj* pode ser acessado como *obj.atrib*
- No exemplo anterior:

```
>>> print ret1.largura
1.0
>>> x = ret1.largura + ret2.altura
>>> x
2.0
```

Acessando métodos

- Um método *f* associado a uma instância *obj* pode ser acessado como *obj.f()*
    - Lembre-se que o método pode ter argumentos, nesse caso *obj.f(arg1, ...argN)*
- No exemplo anterior:

```
>>> ret1.area( )
1.0
>>> ret2.perimetro( )
4.0
```

Modificando atributos

- Podemos modificar o valor de um atributo como faríamos com uma variável normal. Exemplo:

```
>>> ret1.largura = 2.0
>>> ret1.altura = 3.0
```

Agora, temos

```
>>> ret1.area( )
6.0
```

Instanciando objeto em Python: outra forma

- Antes fazíamos

  ret1 = Retangulo( )

  para definir um retângulo que, por padrão, era inicializado com dimensões 1x1.

- Será que poderíamos fazer algo do tipo:

  ret1 = Retangulo (2, 4)

  para definir logo um retângulo 2 x 4?

- Sim! Muito fácil:

```
class Retangulo:
    def __init__(self, alt, larg):
        self.altura = alt
        self.largura = larg
```

Instanciando objeto em Python: outra forma

- Agora,

```
>>> ret = Retangulo(2.0, 4.0)
>>> ret.area( )
8.0
```

- Mas, agora, se fizermos...

```
>>> ret = Retangulo( )
```

dá erro!

Instanciando objeto em Python: outra forma

- Como aceitar as duas formas? Muito fácil:

```
class Retangulo:
    def __init__(self, alt=1.0, larg=1.0):
        self.altura = alt
        self.largura = larg
```

- Se passarmos parâmetros, eles são utilizados para inicializar o objeto. Se não passarmos, fica valendo 1.0 mesmo.
- Novidade nenhuma aqui. Revisem o conteúdo de **funções** que vocês estudaram em **Programação I**. Caso tenham dúvidas, posso propor exercícios de revisão.

Instanciando objeto em Python: outra forma

- Agora,

```
>>> ret1 = Retangulo( )
>>> ret1.area( )
1.0
```

- E também, se fizermos...

```
>>> ret2 = Retangulo(3.5, 2.0)
>>> ret2.area( )
7.0
```

Exercício

- Considere as classes sugeridas abaixo, faça os diagramas de classses, e em seguida implemente-as em Python. Escreva um pequeno programa que instancie objetos a partir das classes criadas:
  - Aluno
  - ContaCorrente
  - Circulo
  - Carro
  - DNA

Encapsulamento

- O conceito de **encapsulamento** afirma que o estado de um objeto não deve ser acessado diretamente, mas sim por meio de **métodos de acesso**.
- Por que???
- Nem sempre é uma boa ideia permitir que o valor do atributo seja alterado diretamente! Exemplo:

```
>>> ret1.largura = -1
```

- Usuário não precisa acessar diretamente todos os detalhes do objeto. Para dirigir, nós não precisamos saber onde fica cada parafuso do motor!

Métodos acessores

- Como então podemos acessar um atributo, sem acessá-lo diretamente???
- Resposta: através de métodos get e set!
  - Ler o valor de um atributo: método **get**
  - Alterar o valor de um atributo: método **set**
  - get e set são apenas nomes “tradicionais”... você pode chamar como quiser, desde que seja um nome fácil de entender

Métodos acessores: exemplo

- Em vez de alterar diretamente os lados do retângulo, podemos definir métodos get e set:

```
class Retangulo:
        def __init__(self):
            self.largura = 1.0
            self.altura = 1.0

        def get_largura( self ):
            return self.largura

        def set_altura ( self, valor ):
            self.altura = valor
```

Métodos acessores: exemplo

- Em vez de alterar diretamente os lados do retângulo, podemos definir métodos get e set:

```
class Retangulo:
        def __init__(self):
            self.largura = 1.0
            self.altura = 1.0

        def set_altura ( self, valor ):
            if valor > 0:
                self.altura = valor
            else:
                print 'ERRO!'
```

Métodos acessores

- Usando o exemplo anterior, vamos criar um retângulo 3x2 e calcular sua área:

```
ret1 = Retangulo( )
ret1.set_largura( 2.0 )
ret1.set_altura( 3.0 )
ret1.area( )
```

Aviso

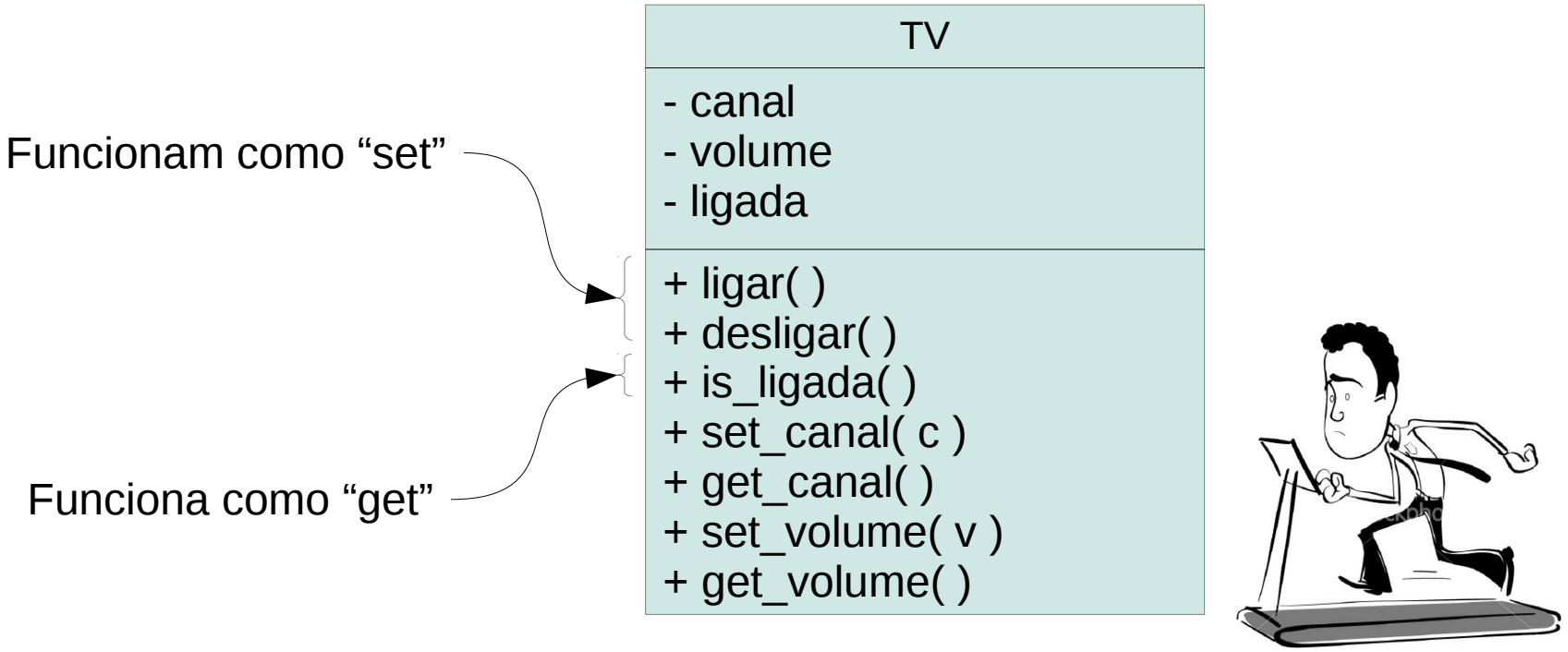
- Há uma forma mais interessante de garantir o encapsulamento em **Python**, sem ficar definindo métodos **get** e **set**.
- Para isso, usa-se o conceito de **properties**.
- Não vamos abordar esse assunto no curso, por fugir do escopo. Quem tiver curiosidade pode me perguntar fora da aula.

Exercício

- Considere as classes sugeridas abaixo, faça os diagramas de classses, e em seguida implemente-as em Python. Faça métodos acessores. Escreva um pequeno programa que instancie objetos a partir das classes criadas:
  - Aluno
  - ContaCorrente
  - Circulo
  - Carro
  - DNA

Exercício

- Implemente em Python a classe abaixo. Crie um programa que instancie um objeto dessa classe e faça pelo menos uma chamada a um de seus métodos.



Membros privados

- Algumas linguagens permitem definir atributos e métodos que só podem ser acessados dentro da própria classe: esses são chamados de atributos ou métodos **privados**
- Tecnicamente, em Python, todos os atributos e métodos são **públicos** (*Guido van Rossum diria: "somos todos adultos"*)
- Porém, há uma forma de simular membros privados em Python: adicionando __ (dois underscores) no início do nome

Membros privados

- Exemplo

```
class Carro:
    def __init__(self):
        self.velocidade = 0.0
        self.combustivel = 0.0
        self.__chassi = “”
```

- Tentem instanciar

```
    c = Carro()
```

  e depois acessar

```
    c.__chassi = “codigo”
```

- Não vão conseguir! chassi é atributo “privado”

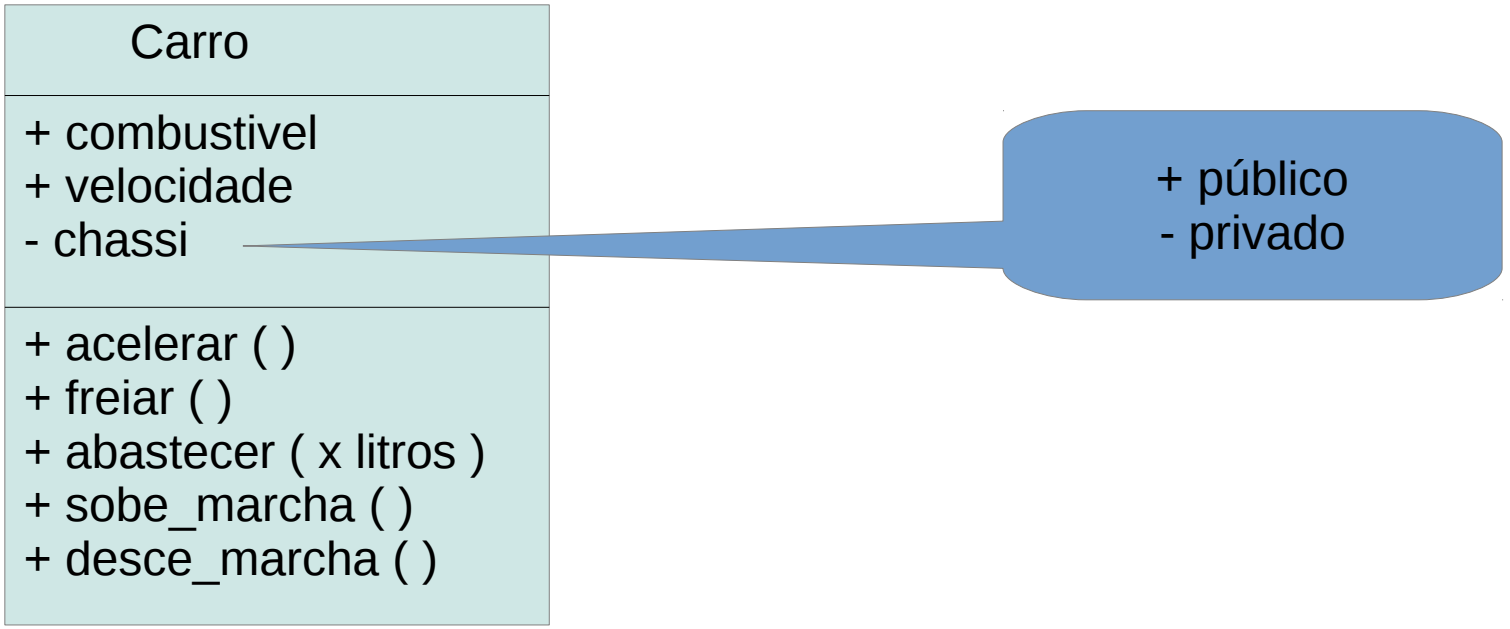
Membros privados

- Como acessar

```
class Carro:
    def __init__(self):
        self.velocidade = 0.0
        self.combustivel = 0.0
        self.__chassi = “”

    def get_chassi(self):
        return self.__chassi
    def set_chassi(self, valor):
        self.__chassi = valor
```

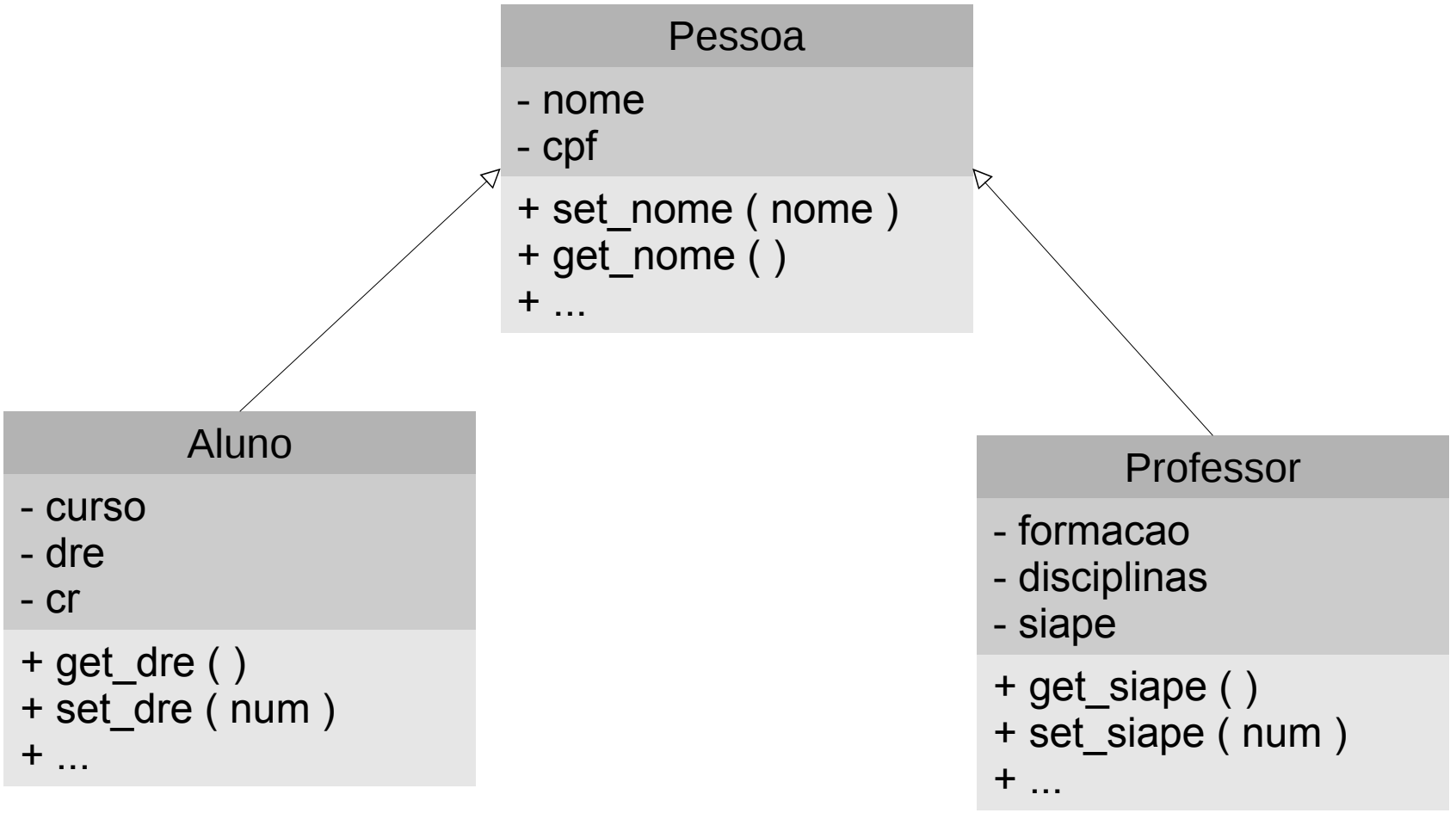
Membros privados na UML



Herança

- Uma classe pode herdar a definição de outra classe
- Ou seja, pode herdar os estados (atributos) e comportamentos (métodos) de uma classe mais “abrangente”
- Nova classe: subclasse, filha, etc.
- Classe original: superclasse, base, etc.

Herança: exemplo



Herança em Python

- As classes do slide anterior, em Python, ficariam assim:

```
class Pessoa:
    def __init__(self):
        self.nome = None
        self.cpf = None
    def set_nome(self, novo_nome):
        self.nome = novo_nome
    …

class Aluno(Pessoa):
    def __init__(self):
        Pessoa.__init__(self)
        self.curso = None
        self.dre = None
        self.cr = None
    def set_curso(self, curso):
        self.curso = curso
    ...
```

Vejam: a nova sintaxe é bem simples!

Mas tenham atenção! Tem que chamar explicitamente o construtor da classe base

Herança em Python

- E agora, como fica se eu quiser instanciar objetos dessas classes? Não muda nada!

```
pessoa = Pessoa ( )
alu1 = Aluno( )
alu2 = Aluno( )
prof1 = Professor( )
```

- E para usar os métodos e atributos? Também não muda nada! Mas note que Aluno e Professor herdam os atributos e métodos de Pessoa.

```
alu1.set_nome(“José Silva”)
prof1.set_cpf(123456789)
```

Exercício

- Para o diagrama de classes fornecido anteriormente, diga quais dos seguintes comandos seriam válidos. Justifique suas respostas!

```
pessoa = Pessoa ( )
aluno = Aluno ( )
professor = Professor ( )

pessoa.set_nome(“José Silva”)
aluno.set_cpf(123456789)
nome = professor.get_nome ( )
cr = professor.get_cr ( )
disc = aluno.get_disciplinas ( )
pessoa.set_formacao(“B.Sc. Ciências Biológicas”)
```

Exercício

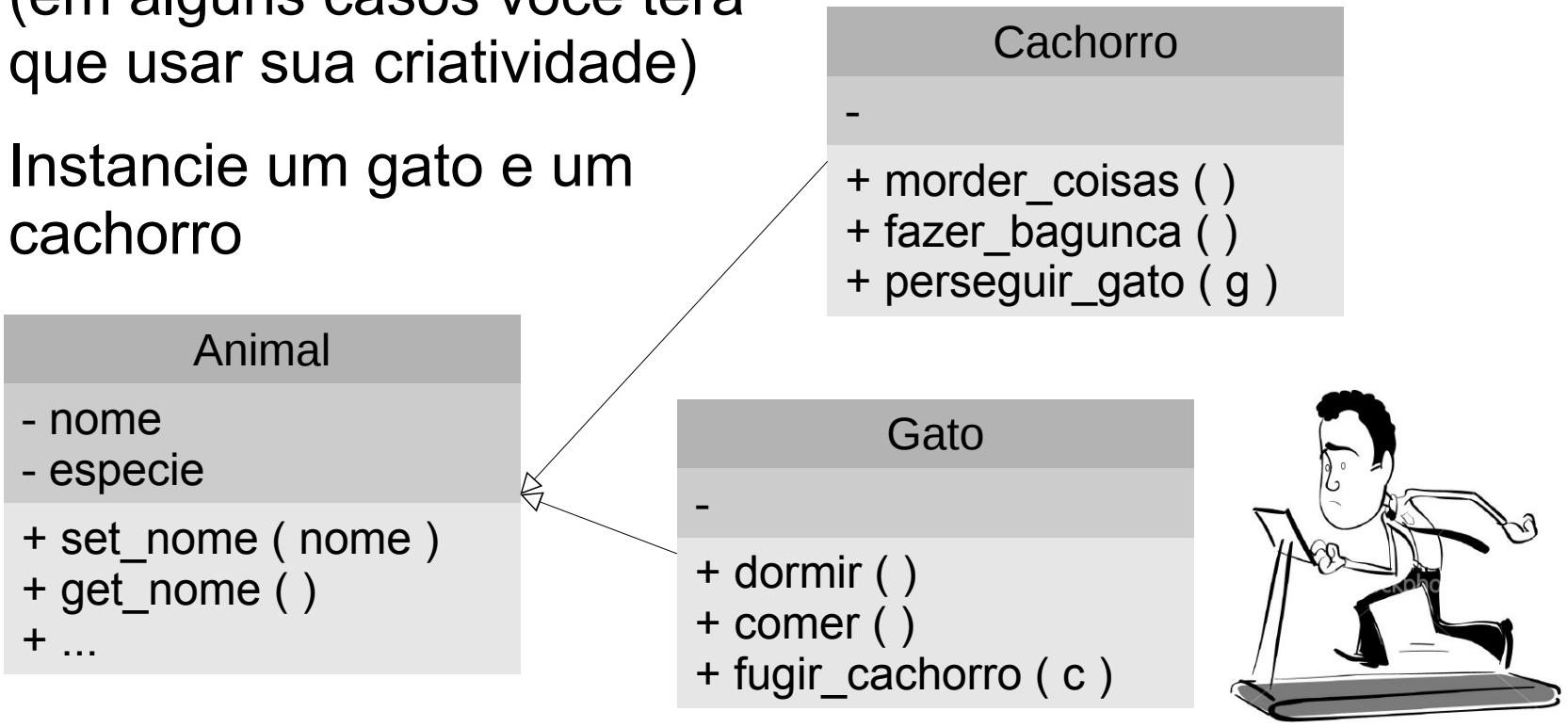
- Para o diagrama de classes fornecido anteriormente, diga quais dos seguintes comandos seriam válidos. Justifique suas respostas!

```
pessoa = Pessoa ( )
aluno = Aluno ( )
professor = Professor ( )

pessoa.set_nome(“José Silva”)
aluno.set_cpf(123456789)
nome = professor.get_nome ( )
cr = professor.get_cr ( )
disc = aluno.get_disciplinas ( )
pessoa.set_formacao(“B.Sc. Ciências Biológicas”)
```

Exercício

- Discuta as seguintes classes
- Implemente-as em Python (em alguns casos você terá que usar sua criatividade)
- Instancie um gato e um cachorro



Outros relacionamentos

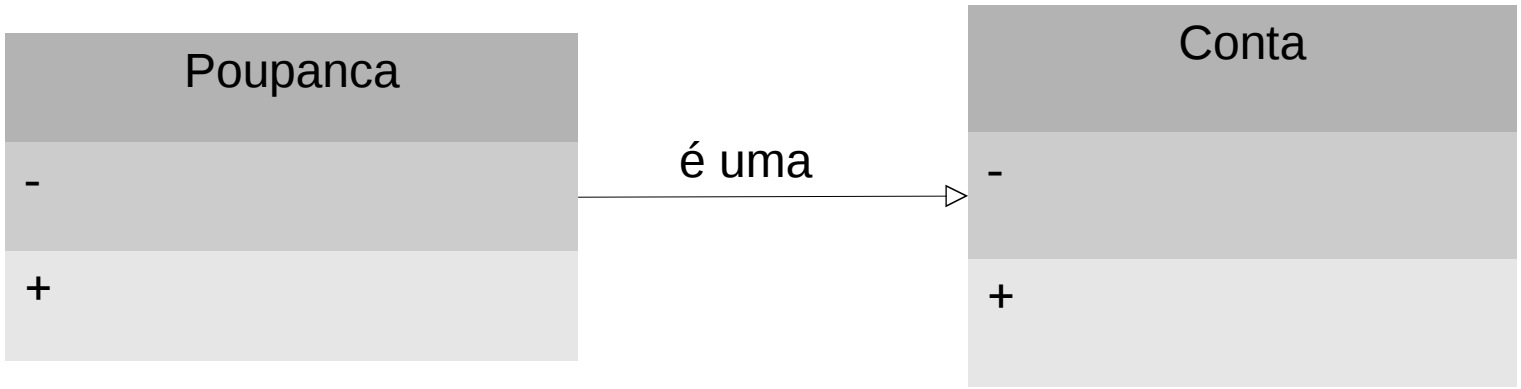
- Além da herança, há outros relacionamentos entre classes que podem ser úteis:
  - Agregação
  - Composição

Diferenças entre herança, composição e agregação

- Se a classe A se relaciona com a classe B da seguinte forma:
  - **Herança**: significa que A *é um* B
    Por exemplo: Poupanca é uma Conta
  - **Agregação**: significa que A *tem um* B
    Num sentido mais fraco que na composição.
    Por exemplo, Conta tem um Cliente
  - **Composição**: significa que A *tem um* B
    Num sentido mais forte que na agregação. Por exemplo, Conta tem um Historico

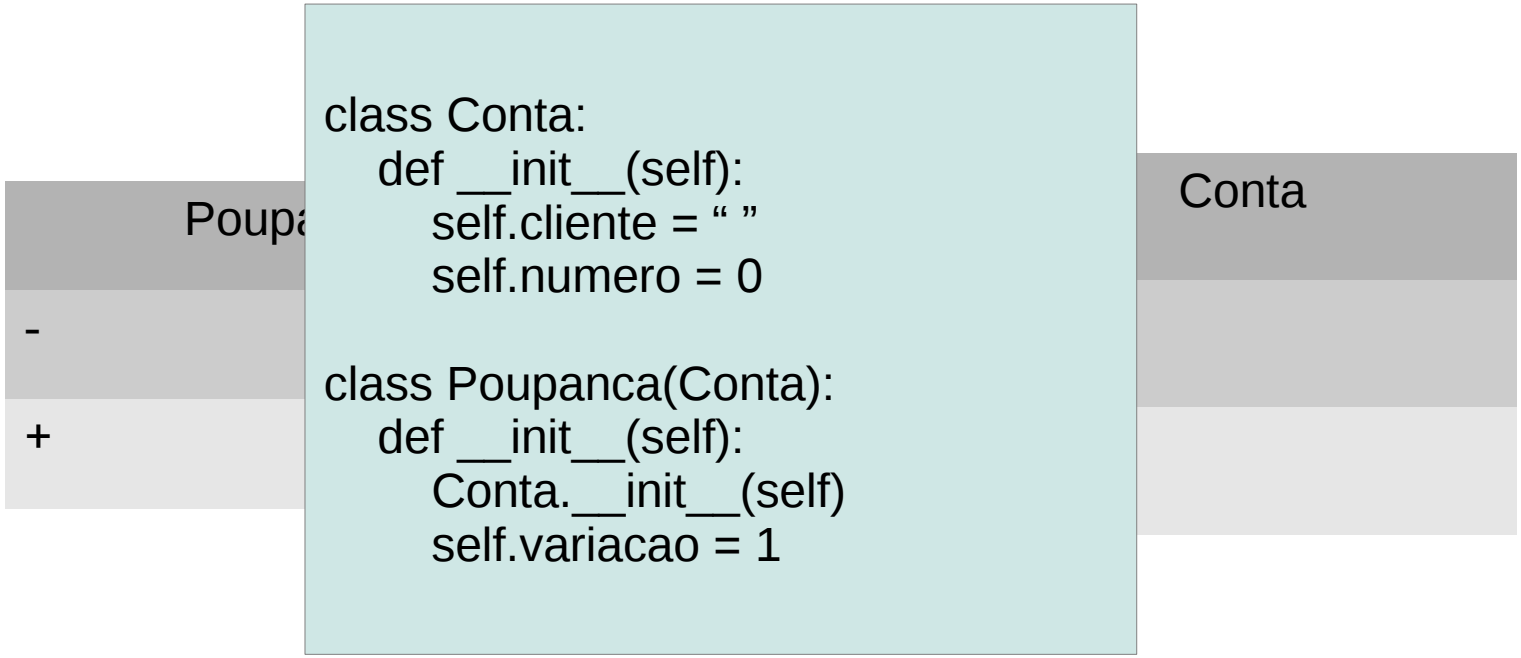
Herança na UML

- Representamos o relacionamento de **herança** através de uma seta com um triângulo vazado.



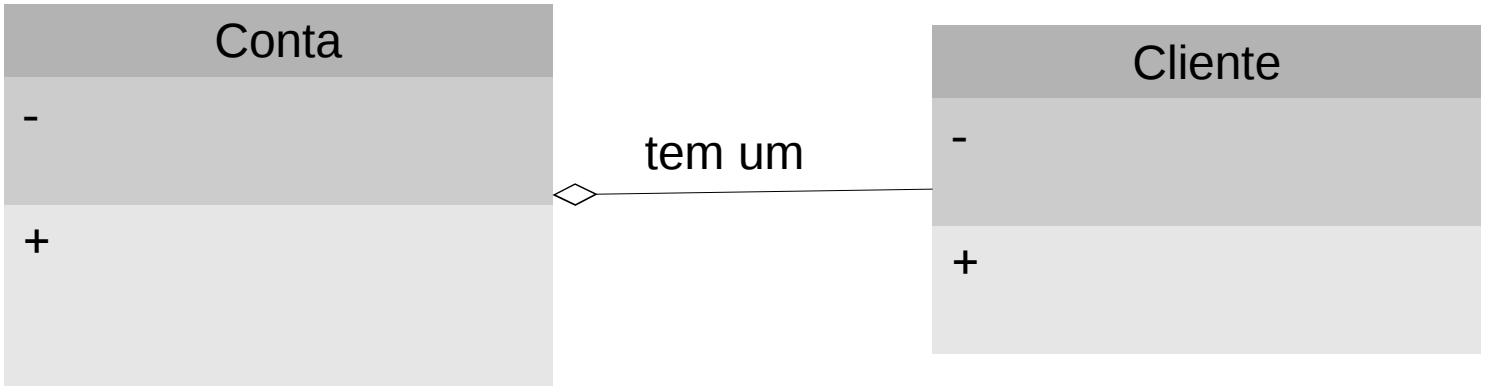
Herança na UML

- Representamos o relacionamento de **herança** através de uma seta com um triângulo vazado.



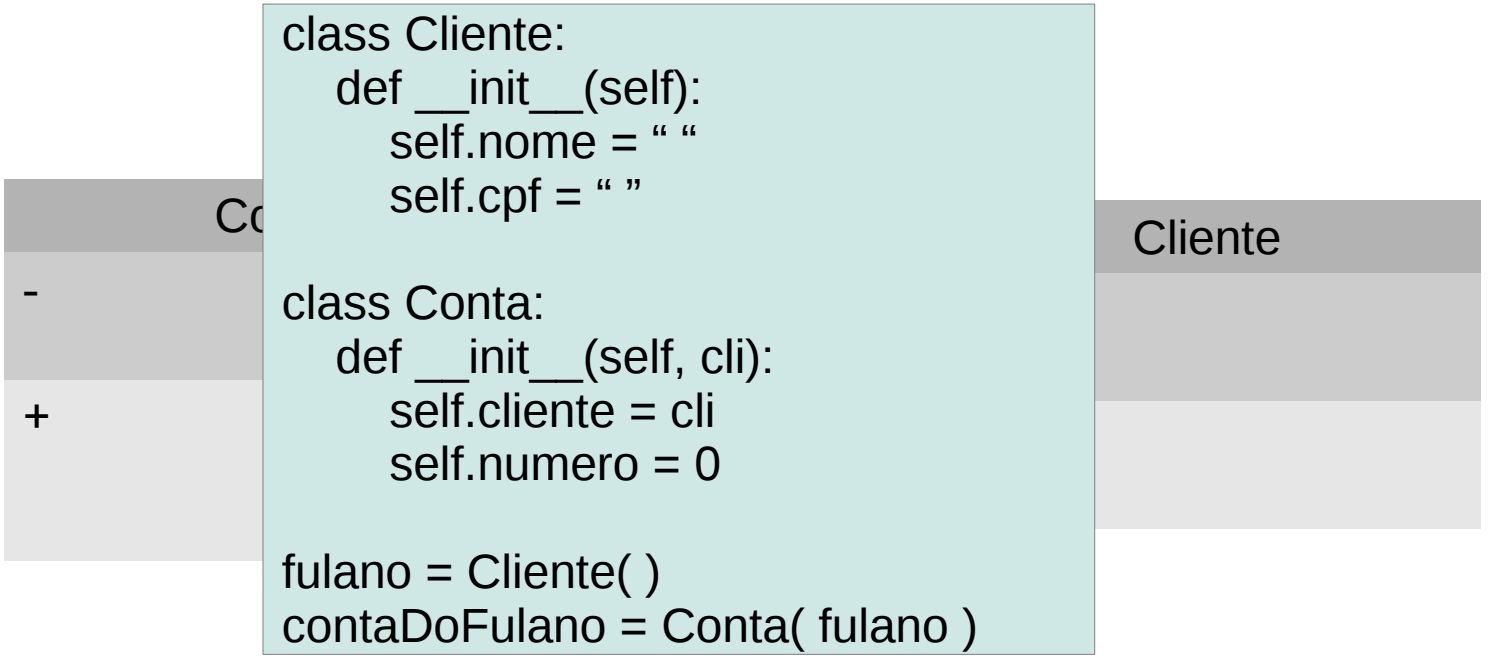
Agregação na UML

- Representamos o relacionamento de **agregação** através de uma seta com um losango vazado.



Agregação na UML

- Representamos o relacionamento de **agregação** através de uma seta com um losango vazado.



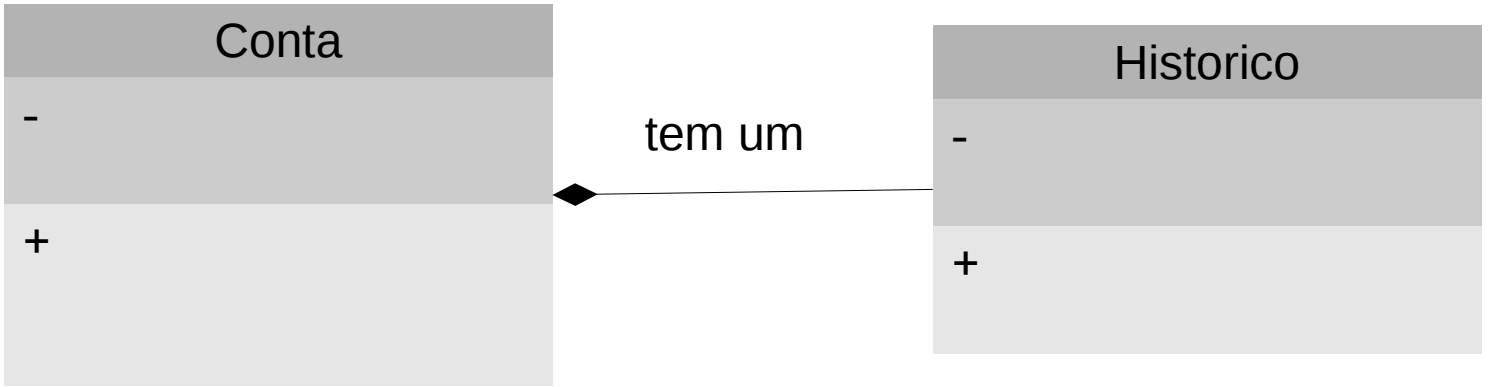

```
class Cliente:
    def __init__(self):
        self.nome = “ “
        self.cpf = “ ”

class Conta:
    def __init__(self, cli):
        self.cliente = cli
        self.numero = 0

fulano = Cliente( )
contaDoFulano = Conta( fulano )
```

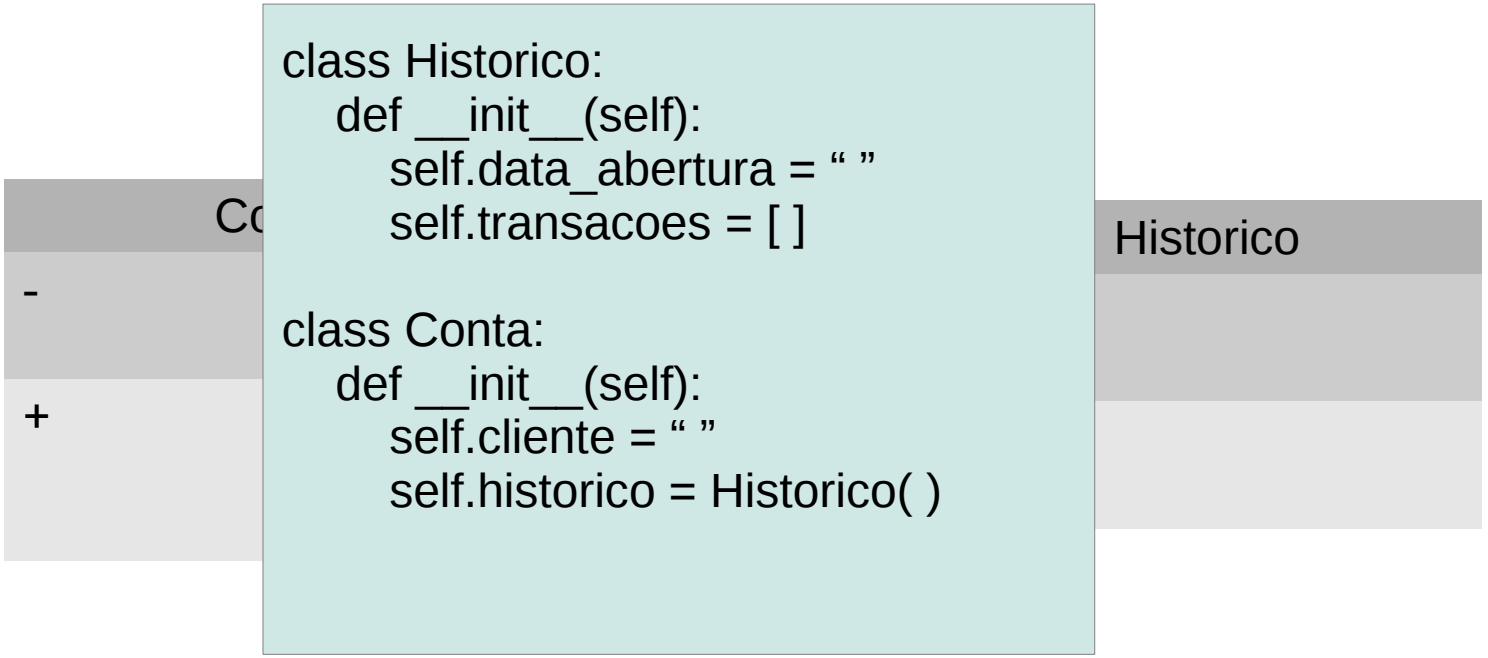
Composição na UML

- Representamos o relacionamento de **composição** através de uma seta com um losango preenchido.



Composição na UML

- Representamos o relacionamento de **composição** através de uma seta com um losango preenchido.



Polimorfismo

- Do grego, *πολύς, polys*, “muitos” e *μορφή, morphē*, “forma”. Ou seja, “muitas formas”
- “O polimorfismo é caracterizado quando duas ou mais classes distintas tem métodos de mesmo nome, de forma que uma função possa utilizar um objeto de qualquer uma das classes polimórficas, sem necessidade de tratar de forma diferenciada conforme a classe do objeto.” (Fonte: Wikipedia)
- Ou seja, podemos tratar instâncias de diferentes classes usando os mesmos “comandos”

Polimorfismo: exemplo

```
class Animal:
    def __init__(self, n):
        self.nome = n
    def falar(self):
        print 'Som genérico'

class Gato( Animal ):
    def falar(self):
        print 'Som de gato miando!'

class Cachorro ( Animal ):
    def falar(self):
        print 'Som de cão latindo!'
```

```
>>> rex = Cachorro( )
>>> tom = Gato( )

>>> rex.falar( )
Som de cão latindo!

>>> tom.falar( )
Som de gato miando!
```

Tratamento de erros e exceções

- Qualquer programa não-trivial certamente vai apresentar erros em algum momento:
  - o usuário pode não entender bem como usar o sistema corretamente
  - pode haver algum problema com o computador do usuário
- Um programa bem escrito precisa lidar com esses erros em tempo de execução

Tratamento de erros e exceções

- Sempre que um programa encontra dificuldades não previstas, ocorre uma **exceção** (exception)
- Se essa situação não é tratada, o programa termina com uma mensagem de **rastreamento** (traceback)

Exceções em Python: exemplo

- Tente fazer uma divisão por zero e ocorre uma exceção:

```
>>> 1/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
```

- Façam outros testes. Errem de propósito e vejam o que acontece. Observem as mensagens.

Exceções em Python

- Como resolver isso na prática? Para encurtar a história, tudo o que precisamos fazer é o seguinte:

```
try:
    # código que pode apresentar exceção
except NomeDaExcecao:
    # o que fazer se ocorrer a exceção
```

Exceções em Python: exemplo

```
try:
    num = input(“Digite um valor: ”)
    den = input(“Digite outro valor: ”)
    resultado = num/den
    print resultado
except ZeroDivisionError:
    print “Nao pode dividir por zero!”
```

Exceções em Python: exemplo

- O exemplo anterior resolve se o usuário tentar dividir por zero. Mas e se ele digitar um texto em vez de um número?

```
try:
    # código que pode apresentar exceção
except NomeDaExcecao:
    # o que fazer se ocorrer a exceção
except OutraExcecao:
    # se ocorrer outra exceção
```

Exceções em Python: exemplo

- Mas e se acontecer uma outra exceção qualquer que eu nem previ inicialmente?

```
try:
    # código que pode apresentar exceção
except NomeDaExcecao:
    # o que fazer se ocorrer a exceção
except OutraExcecao:
    # se ocorrer outra exceção
except:
    # qualquer exceção não contemplada nos
    # casos anteriores
```

Algumas classes de exceção

Classe	Descrição
Exception	Classe base para todas as exceções
AttributeError	Falha no acesso ou atribuição a atributo de classe
IOError	Falha no acesso a arquivo inexistente ou outros de E/S
IndexError	Índice inexistente de seqüência
KeyError	Chave inexistente de dicionário
NameError	Variável inexistente
SyntaxError	Erro de sintaxe (código errado)
TypeError	Operador embutido aplicado a objeto de tipo errado
ValueError	Operador embutido aplicado a objeto de tipo certo mas valor inapropriado
ZeroDivisionError	Divisão ou módulo por zero

Disparando as próprias exceções

- Para sinalizar a ocorrência de uma situação excepcional, usa-se o comando raise.
- Exemplo:

```
raise Classe
raise Classe, mensagem
raise Classe(mensagem)
```

Definindo as próprias classes de exceção

- Basta criar uma classe que herde a classe Exception
- Não precisa definir nenhum atributo e nenhum método.
- Exemplo:

```
class MinhaPropriaExcecao(Exception):
    pass
```