

Programação Científica

Prof. Dr. Danilo H. Perico

Programação Orientada a Objetos

Retomando a última aula...

Classes

- Representam itens do mundo real:
 - Exemplos:
 - Pessoas
 - Veículos
 - Robôs
- São Compostas de:
 - **Atributos** (variáveis de instância)
 - **Métodos** (funções-membro)

Objetos

- Todo objeto pertence a uma **classe**
- Representam **instâncias** de entidades no mundo real
- São criados a partir das classes
- São instâncias das classes
- Os objetos associam valores específicos aos atributos

Instanciar (Informática)

- Instanciar é criar um objeto, ou seja, alocar um espaço na memória, para posteriormente poder utilizar os métodos e atributos que o objeto dispõe

Classes e Objetos

Quando definimos uma **classe** de objetos, estamos, na verdade, definindo que **propriedades** e **métodos** o **objeto possui!**

Diferença entre Classe e Objeto

- **Classe:**

- É um modelo
- De maneira mais prática, é como se fosse a **planta de uma casa**

- **Objeto:**

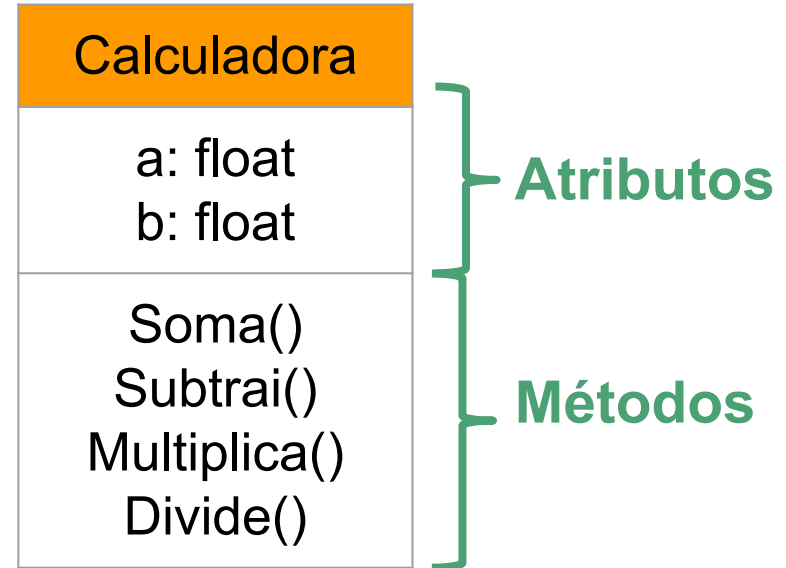
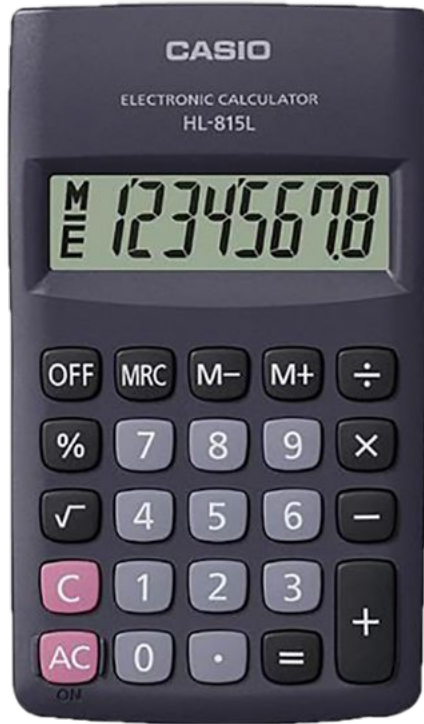
- É criado a partir da classe
- É como se fosse a **própria casa** construída
- Pode-se construir várias casas a partir da mesma planta, assim como podemos instanciar vários objetos de uma só classe

Estrutura de uma Classe

Nome da Classe
- Atributos
- Métodos

- **Atributos** são **variáveis** que armazenam informações do objeto.
- **Métodos** são as operações (**funções**) que o objeto pode realizar.

Exemplo de Classe



Exemplo de Classe e Objetos

Cat
size: float color: string positionX: float positionY: float
moveForward() moveBackward() moveUP() moveDown()

Cat garfield;
Cat tom;
Cat felix ;
Cat scratchy;



Algumas Linguagens Orientadas a Objetos



POO no Python

```
1 | class NomeClasse:  
2 |     # atributos  
3 |  
4 |     # métodos
```

```
1 | class Aluno:  
2 |     nome = ""  
3 |     ra = 0  
4 |  
5 |     def mostraAluno(self):  
6 |         print("Nome: %s" % self.nome)  
7 |         print("R.A.: %d" % self.ra)
```

POO no Python

self serve para identificar os atributos e os métodos da classe

Serve principalmente para delimitar o escopo e tirar ambiguidade com outras possíveis variáveis

```
1 class Aluno:
2     nome = " "
3     ra = 0
4
5     def mostraAluno(self):
6         print("Nome: %s" % self.nome)
7         print("R.A.: %d" % self.ra)
8
9 aluno = Aluno()
10 aluno.nome = "Danilo"
11 aluno.ra = 123456789
12 aluno.mostraAluno()
13
14
```

```
Nome: Danilo
R.A.: 123456789
```

POO no Python

cria o objeto aluno
da classe **Aluno**

```
1 class Aluno:
2     nome = " "
3     ra = 0
4
5     def mostraAluno(self):
6         print("Nome: %s" % self.nome)
7         print("R.A.: %d" % self.ra)
8
9     aluno = Aluno()
10    aluno.nome = "Danilo"
11    aluno.ra = 123456789
12    aluno.mostraAluno()
13
14
```

```
Nome: Danilo
R.A.: 123456789
```

POO no Python

executa o método mostraAluno() do objeto aluno

```
1 class Aluno:
2     nome = " "
3     ra = 0
4
5     def mostraAluno(self):
6         print("Nome: %s" % self.nome)
7         print("R.A.: %d" % self.ra)
8
9 aluno = Aluno()
10 aluno.nome = "Danilo"
11 aluno.ra = 123456789
12 aluno.mostraAluno()
13
14
```

```
Nome: Danilo
R.A.: 123456789
```


Aula de hoje...

Métodos especiais

Construtor

No Python, o construtor da classe é chamado de __init__

```
1 class Aluno:
2
3     def __init__(self, nome, ra):
4         self.nome = nome
5         self.ra = ra
6
7     def mostraAluno(self):
8         print("Nome: %s" % self.nome)
9         print("R.A.: %d" % self.ra)
10
11 aluno = Aluno("Danilo", 123456789)
12 aluno.mostraAluno()
13
14
15
16
```

construtor:
inicializa o valor
dos **atributos**
quando o objeto é
instanciado

Cria / instancia o objeto
aluno já com valores
específicos para os
atributos **nome** e **ra**

```
Nome: Danilo
R.A.: 123456789
```

Construtores - O que são?

- Um **construtor** é um tipo especial de método chamado para criar um objeto
- O construtor prepara o novo objeto para uso, aceitando argumentos para inicializar os atributos
- Toda classe tem pelo menos um construtor, se ele não for declarado explicitamente, o compilador fornece um construtor-padrão

Exemplo - *classe Funcionario*

Crie uma classe para modelar um *Funcionario* com o seguinte:

- adicione os atributos: **nome**, **sobrenome**, **salario**, **idade**, **numero**
- Faça um **construtor** que permita a inicialização do objeto sem que nenhum valor seja enviado e, ao mesmo tempo, que também permita que os 5 parâmetros, um para cada atributo, sejam enviados já no momento em que o objeto é criado
- Teste a classe instanciando vários objetos da classe *Funcionario*

Método `__str__`

- O método `__str__()` é um *instance method* que são funções definidas dentro de uma classe e só podem ser chamados a partir de uma instância dessa classe, assim como `__init__()`
- O método `__str__()` permite modificar a forma como uma instância irá imprimir

Método `__str__`

- Exemplo: Crie uma classe para modelar pessoas - **classe** *Pessoa*
- Utilize o método `__str__` para exibir informações relevantes do objeto Pessoa

Método __str__

```
class Pessoa:
    nome = ""
    cpf = ""
    telefone = ""
    |
    def __init__(self, nome, cpf, telefone):
        self.nome = nome
        self.cpf = cpf
        self.telefone = telefone

    def __str__(self):
        return f"Nome: {self.nome}, CPF: {self.cpf}, Telefone: {self.telefone}"
#=====

pess = Pessoa("Fulano", "111.111.111-11", "(11) 2222-2222")

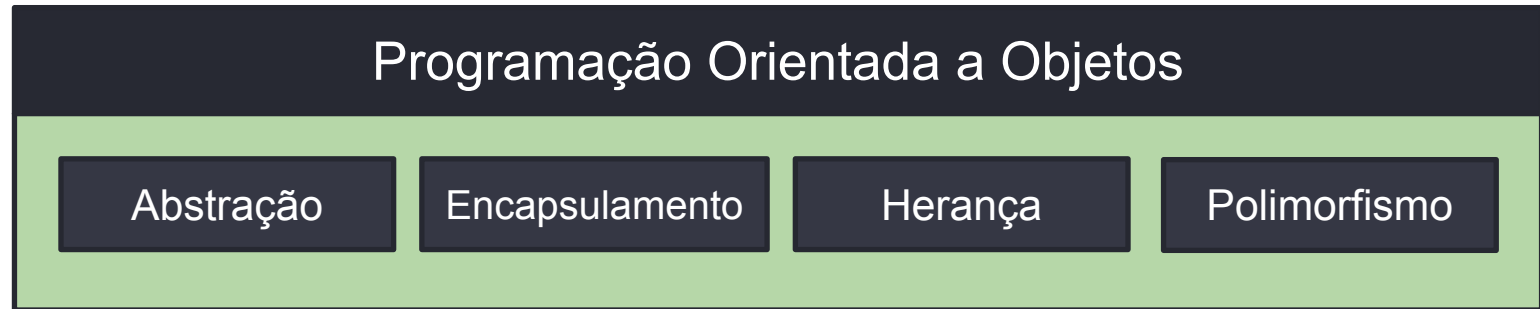
print(pess)
```

Nome: Fulano, CPF: 111.111.111-11, Telefone: (11) 2222-2222

Encapsulamento

Encapsulamento

- Encapsulamento é um dos 4 pilares de Programação Orientada a Objetos tradicional:



Encapsulamento

- Encapsulamento vem de **encapsular**, que em programação orientada a objetos significa separar o programa em partes, o mais isoladas possível
- A ideia é tornar o software mais flexível, fácil de modificar e de criar novas implementações

Encapsulamento

- Uma parte importante do Encapsulamento é a definição dos modificadores de acesso dos atributos ou métodos
 - Os modificadores de acesso mais comuns são: ***public***, ***private*** e ***protected***
- Eles são usados como uma forma eficiente de proteger os dados manipulados pela classe/objeto

Encapsulamento em Python

- O Python não possui os mecanismos tradicionais de modificador de acesso!
- No entanto, existe uma convenção que é seguida:
 - Um nome prefixado com um sublinhado deve ser tratado como uma parte *não-pública* da API
 - Tais nomes devem ser considerados um detalhe interno de implementação e sujeito a alteração sem aviso prévio pelos métodos da classe

Encapsulamento em Python

- Um nome prefixado com dois sublinhados invocará as regras de desfiguração (*mangling*) de nomes do Python
- Python desfigura esses nomes com o nome da classe:
 - se a classe **Foo** tem um atributo chamado `__a`, será `_Foo__a`
 - Mas isso ainda não torna o campo privado de verdade, pois ainda é possível acessá-lo

Encapsulamento em Python - Exemplo (“_”)

```
class Robot:  
    _name = "Tiago++"  
    _positionX = 0.0  
    _positionY = 0.0  
    _direction = 0.0
```

Atributos
considerados privados

Encapsulamento em Python - Exemplo (“_”)

- Mesmo com a inclusão do _ no início do nome dos atributos, eles ainda são acessíveis:

```
C3_PELE = Robot()
R2D_DUNGA = Robot()
ROBOMARIO = Robot()

print (C3_PELE._positionX)
print (C3_PELE._positionY)

0.0
0.0
```


Encapsulamento em Python - Exemplo (“__”)

```
class Robot:  
    __name = "Tiago++"  
    __positionX = 0.0  
    __positionY = 0.0  
    __direction = 0.0
```

**Atributos
considerados privados
e desconfigurados**

Exemplo (“__” - 2 sublinhados)

```
class Robot:
    __name = "Tiago++"
    __positionX = 0.0
    __positionY = 0.0
    __direction = 0.0

    def __init__(self, nome = "Tiago++"):
        self.name = nome
        print("Construindo o %s :-)" % self.name)

    def __del__(self):
        print("Bye bye!!")

    def moveForward(self):
        print("Anda para frente")

    def moveBackward(self):
        print("Anda para tras")

    def turnLeft(self):
        print("Vira para esquerda")

    def turnRight(self):
        print("Vira para direita")

    def stop(self):
        print("Para")
```

```
C3_PELE = Robot()
R2D_DUNGA = Robot()
ROBOMARIO = Robot()
```

```
print (C3_PELE.__positionX)
print (C3_PELE.__positionY)
```

```
Construindo o Tiago++ :-)
Bye bye!!
Construindo o Tiago++ :-)
Bye bye!!
Construindo o Tiago++ :-)
Bye bye!!
```

```
-----
AttributeError                                Traceback (most recent call last)
<ipython-input-17-fe02229a60c9> in <module>
      3 ROBOMARIO = Robot()
      4
----> 5 print (C3_PELE.__positionX)
      6 print (C3_PELE.__positionY)

AttributeError: 'Robot' object has no attribute '__positionX'
```

Exemplo (“__” - 2 sublinhados)

```
class Robot:
    __name = "Tiago++"
    __positionX = 0.0
    __positionY = 0.0
    __direction = 0.0

    def __init__(self, nome = "Tiago++"):
        self.name = nome
        print("Construindo o %s :-)" % self.name)

    def __del__(self):
        print("Bye bye!!")

    def moveForward(self):
        print("Anda para frente")

    def moveBackward(self):
        print("Anda para tras")

    def turnLeft(self):
        print("Vira para esquerda")

    def turnRight(self):
        print("Vira para direita")

    def stop(self):
        print("Para")
```

```
C3_PELE = Robot()
R2D_DUNGA = Robot()
ROBOMARIO = Robot()

print (C3_PELE.__Robot__positionX)
print (C3_PELE.__Robot__positionY)
```

```
Construindo o Tiago++ :-)
Bye bye!!
Construindo o Tiago++ :-)
Bye bye!!
Construindo o Tiago++ :-)
Bye bye!!
0.0
0.0
```

Exemplo: “__” com métodos de acesso

```
class Robot:
    __name = "Tiago++"
    __positionX = 0.0
    __positionY = 0.0
    __direction = 0.0

    def getPosition(self):
        print ("X = %f" % self.__positionX)
        print ("Y = %f" % self.__positionY)

    def setPosition(self, x, y):
        self.__positionX = x
        self.__positionY = y

c3_pele = Robot()
c3_pele.setPosition(10,20)
c3_pele.getPosition()
```

Atributos considerados
privados e desconfigurados

Métodos de Acesso

X = 10.000000

Y = 20.000000

Exercício 1 - *classe Racional*

Crie uma classe para modelar a aritmética de frações.

A classe deve ser chamada de **Racional**. Utilize variáveis inteiras para representar os atributos da classe: o **numerador** e o **denominador**. Forneça um construtor que permita que um objeto dessa classe seja inicializado com os valores do numerador e do denominador. O construtor deve armazenar a fração em uma forma reduzida, por exemplo, a fração $2/4$ é equivalente a $1/2$ e seria armazenada no objeto como 1 no numerador e 2 no denominador.

Exercício 1 - *classe Racional*

Forneça um construtor sem argumento com valores padrão caso nenhum inicializador seja fornecido. Forneça métodos que realizam cada uma das operações a seguir:

- a) Somar dois números ***Racional***: o resultado da adição deve ser armazenado na forma reduzida.
- b) Subtrair dois números ***Racional***: o resultado da subtração deve ser armazenado na forma reduzida.
- c) Multiplicar dois números ***Racional***: o resultado da multiplicação deve ser armazenado na forma reduzida.

Exercício 1 - *classe Racional*

- d) Dividir dois números ***Racional***: o resultado da divisão deve ser armazenado na forma reduzida.
- e) Retornar uma representação *String* de um número ***Racional*** na forma a/b , em que a é o numerador e b é o denominador.
- f) Retornar uma representação *String* de um número ***Racional*** no formato de ponto flutuante.

Teste a classe instanciando vários objetos ***Racional***

Sobrecarga de Operador

Sobrecarga de Operador

- Não seria bom se pudéssemos pegar 2 objetos **Racional** e utilizar os operadores tradicionais **+**, **-**, ***** e **/** para realizar as operações aritméticas solicitadas?
- Por exemplo: **r3 = r1 + r2** ou **r3 = r1 * r2**
 - Em que **r1**, **r2** e **r3** são objetos **Racional**

Sobrecarga de Operador

- As linguagens Orientadas a Objeto não sabem o que o **operador +** deve fazer com 2 objetos, por exemplo
 - As linguagens de programação sabem o que o **operador +** faz com números e, muitas vezes, com strings
 - Mas não com objetos, que podem ser ou representar qualquer entidade

Sobrecarga de Operador

- Contudo, no Python podemos usar os **magic methods** ou métodos mágicos / especiais!
- Esses métodos permitem a **sobrecarga de operador!**
 - Sobrecarregar o operador significa mudar a função do operador para que ele saiba o que fazer com os objetos daquela classe

Sobrecarga de Operador

- Principais *Magic Methods*:

- `__add__`
- `__sub__`
- `__mul__`
- `__truediv__`
- `__floordiv__`
- `__init__`

Sobrecarga de Operador

```
class Teste:
    def __init__(self, a):
        self.a = a

    # adicionar dois objetos da classe Teste
    def __add__(self, o):
        return self.a + o.a

t1 = Teste(1)
t2 = Teste(2)

print(t1 + t2)
```

3

Sobrecarga de Operador

- Incluindo a sobrecarga de operador na classe ***Racional***

Introdução à Linguagem de Modelagem Unificada - UML

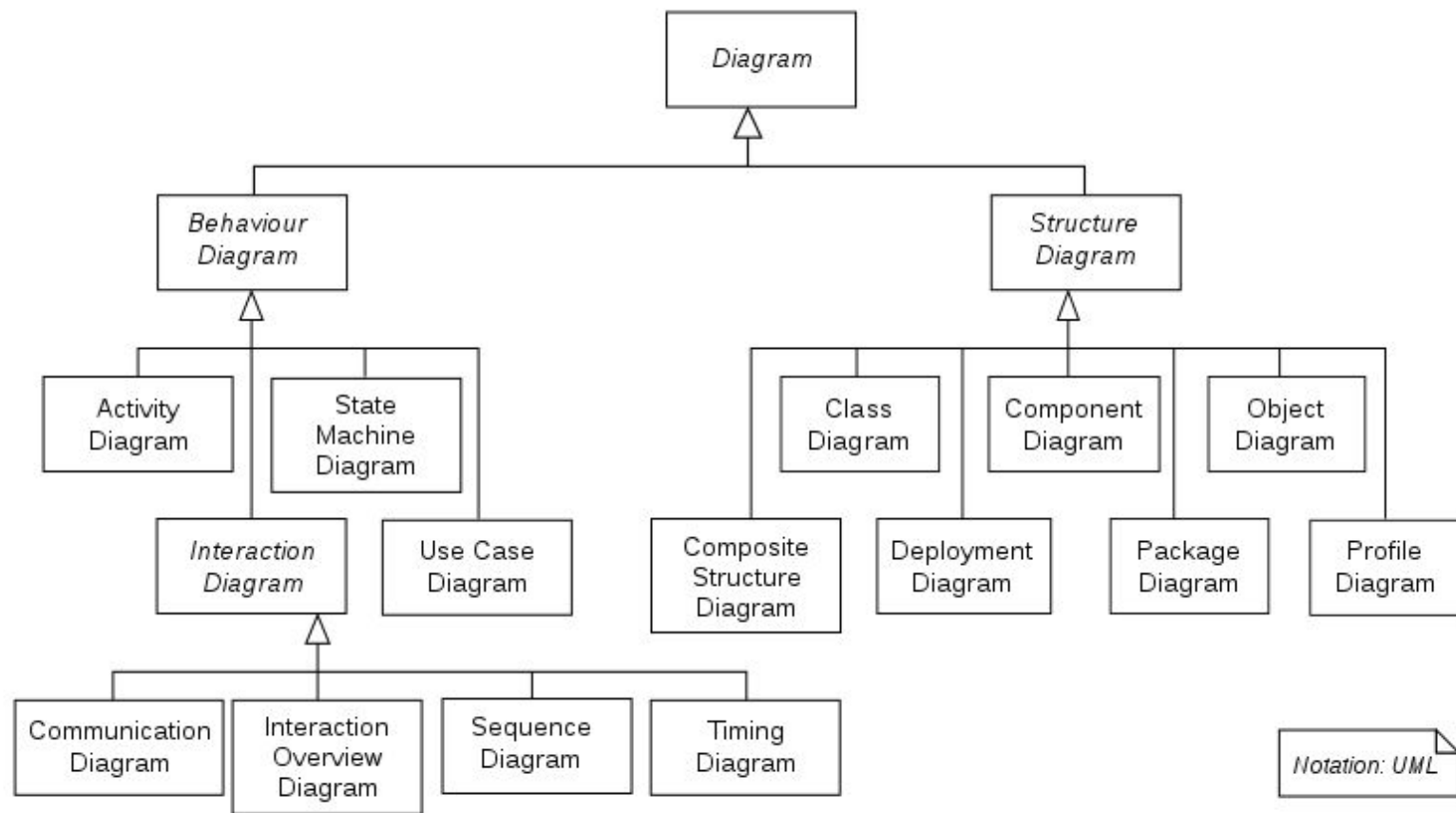


UML

- ***Unified Modeling Language (UML) - Linguagem de Modelagem Unificada*** - é uma linguagem visual utilizada para modelar sistemas computacionais orientados a objeto
- Nos últimos anos, a UML consagrou-se como a linguagem-padrão de modelagem adotada pela indústria de Engenharia de Software, havendo atualmente um amplo mercado para profissionais que a dominem.

UML

- A UML não é uma metodologia de desenvolvimento:
 - ela não diz para você como projetar seu sistema
- **Mas ela auxilia a visualizar seu desenho e a comunicação entre os objetos**



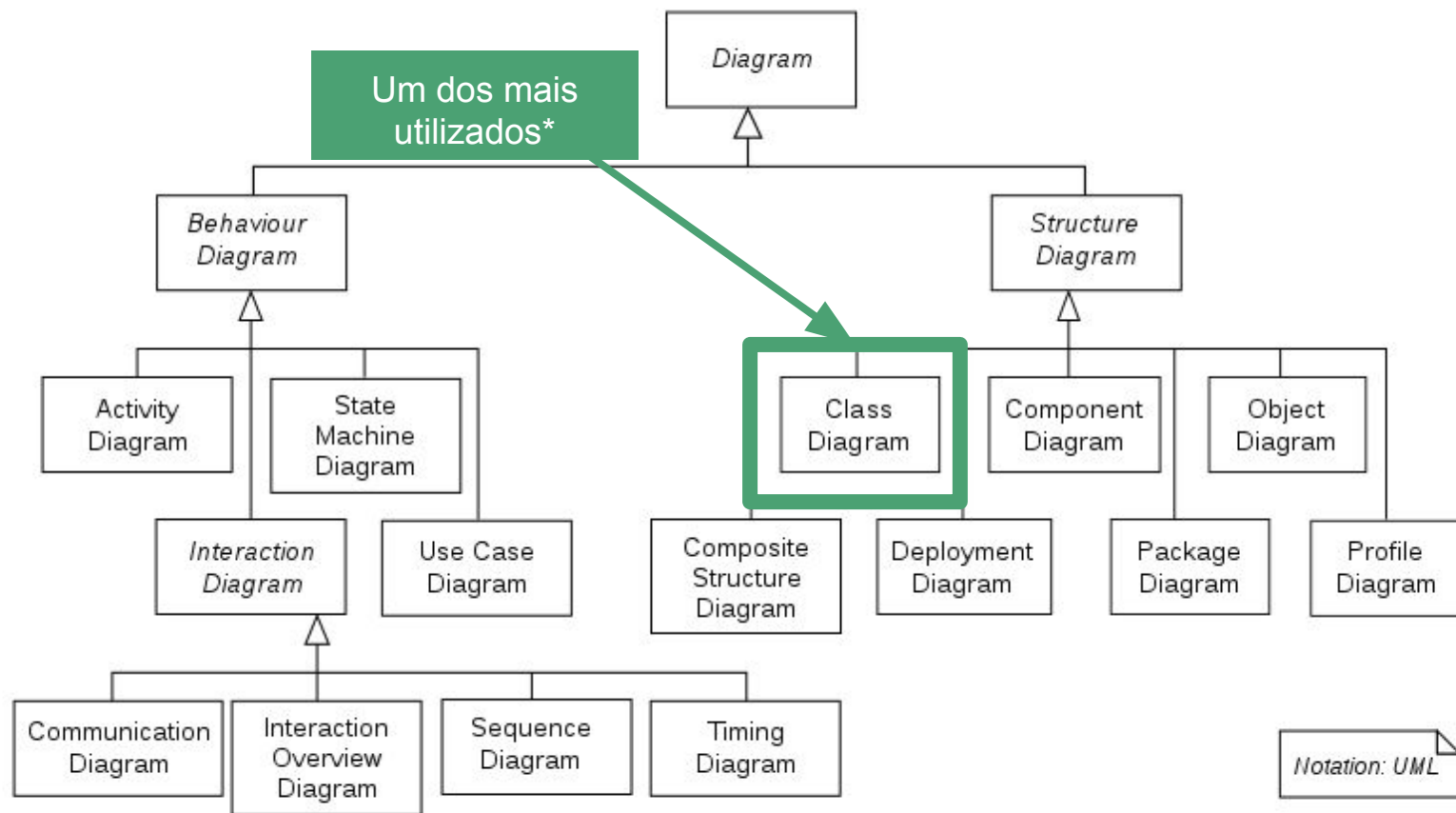


Diagrama de Classes

- Serve de apoio para a maioria dos demais diagramas
- Define a estrutura das classes utilizadas pelo sistema:
 - Define os atributos e métodos que cada classe tem
 - Estabelece como as classes se relacionam e trocam informações entre si

Diagrama de Classes

Dividido em 3 partes

A exibição dos parâmetros dos métodos é opcional

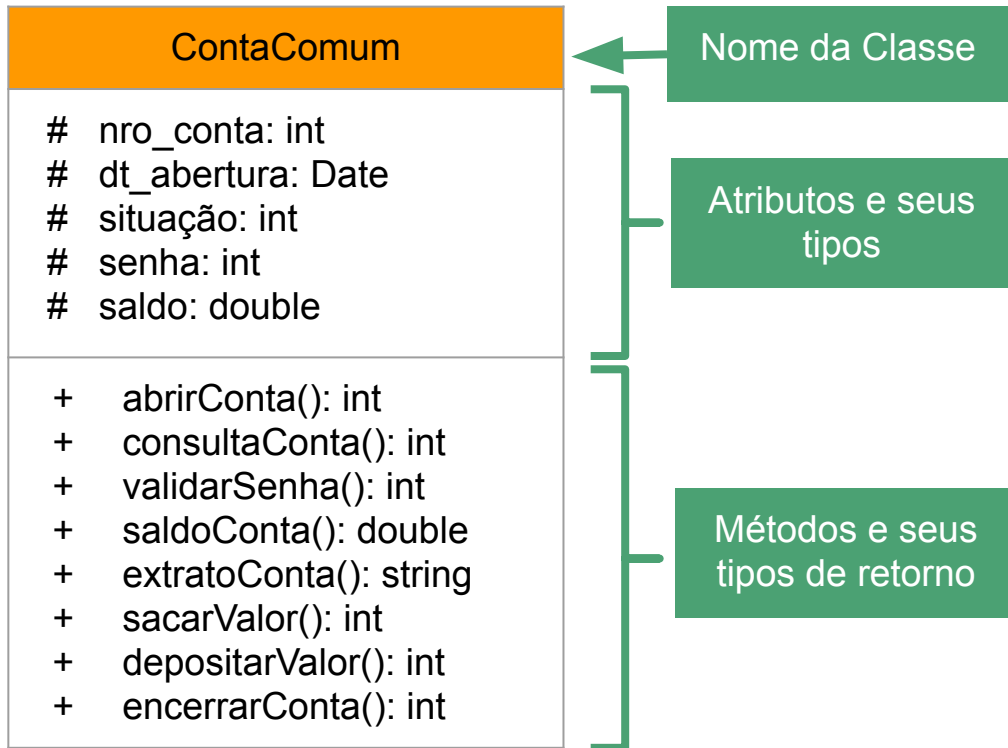


Diagrama de Classes

Métodos com
parâmetros e seus
tipos de retorno

ContaComum

```
# nro_conta: int  
# dt_abertura: Date  
# situação: int  
# senha: int  
# saldo: double
```

```
+ abrirConta(int): int  
+ consultaConta(int): int  
+ validarSenha(int): int  
+ saldoConta(): double  
+ extratoConta(Date): string  
+ sacarValor(double): int  
+ depositarValor(int, double): int  
+ encerrarConta(): int
```

Diagrama de Classes - Visibilidade

- Indica o nível de acessibilidade
- Basicamente 3 modos de visibilidade:
 - Privada: - (menos)
 - Pública: + (mais)
 - Protegida: # (sustenido)

Diagrama de Classes

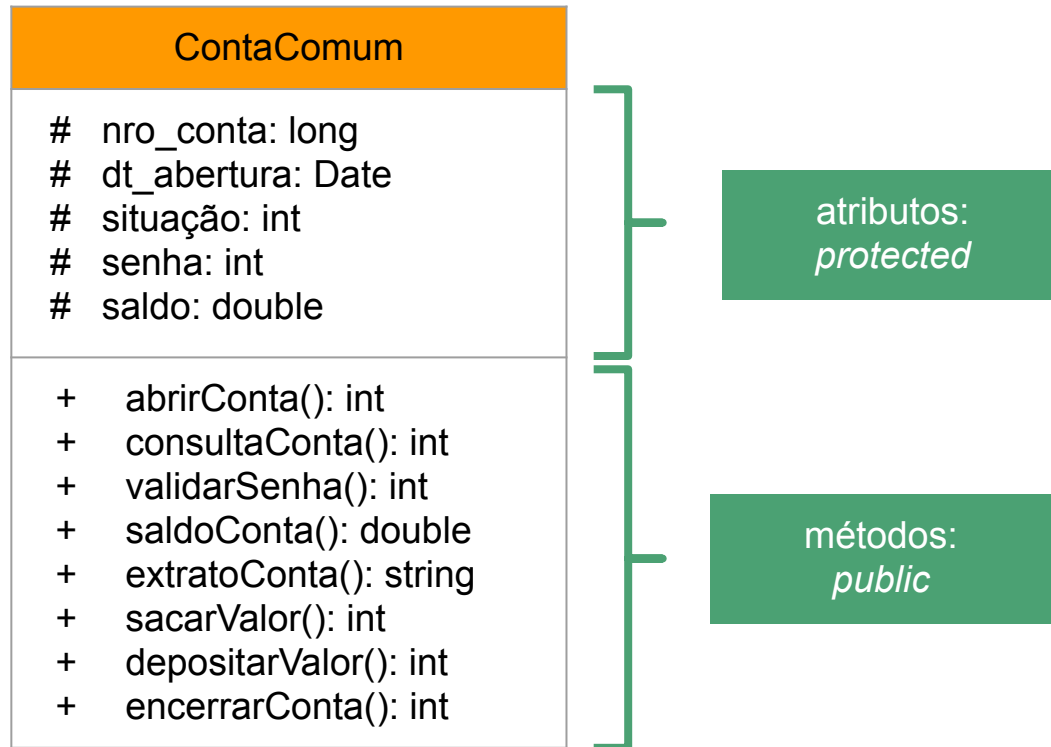


Diagrama de Classes - Associações

- As classes de um projeto costumam ter relacionamentos entre si:
 - **Associações**
 - Uma associação descreve um vínculo que ocorre entre os objetos de uma ou mais classes

Diagrama de Classes - Associação Binária

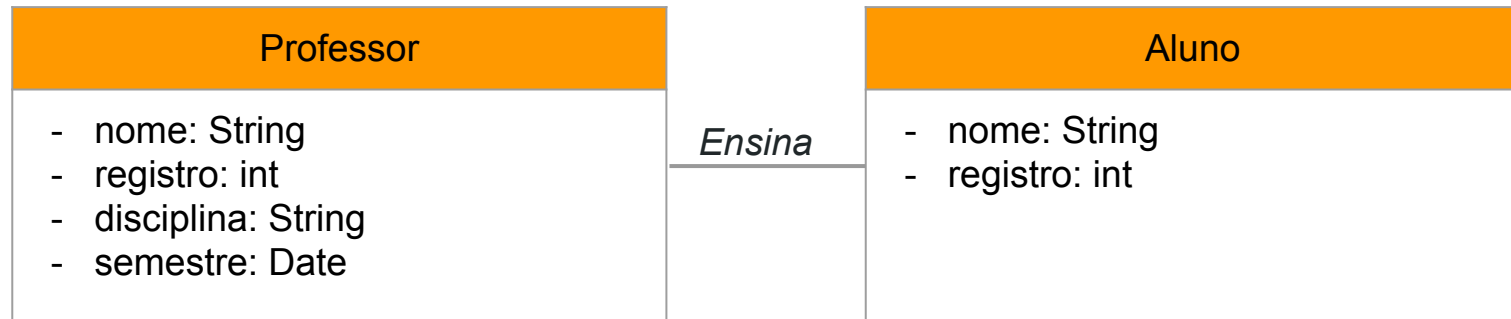


Diagrama de Classes - Associações

- É possível representar o número mínimo e máximo de objetos envolvidos em cada extremidade da associação, por meio da **multiplicidade**

Diagrama de Classes - Associações

Multiplicidade	Significado
0..1	Mínimo 0 e máximo 1: Não precisam necessariamente estar relacionados
1..1	1 e somente 1
0..*	No mínimo nenhum e no máximo muitos
*	Muitos
1..*	No mínimo 1 e no máximo muitos
3..5	No mínimo 3 e no máximo 5

Diagrama de Classes - Associação Binária

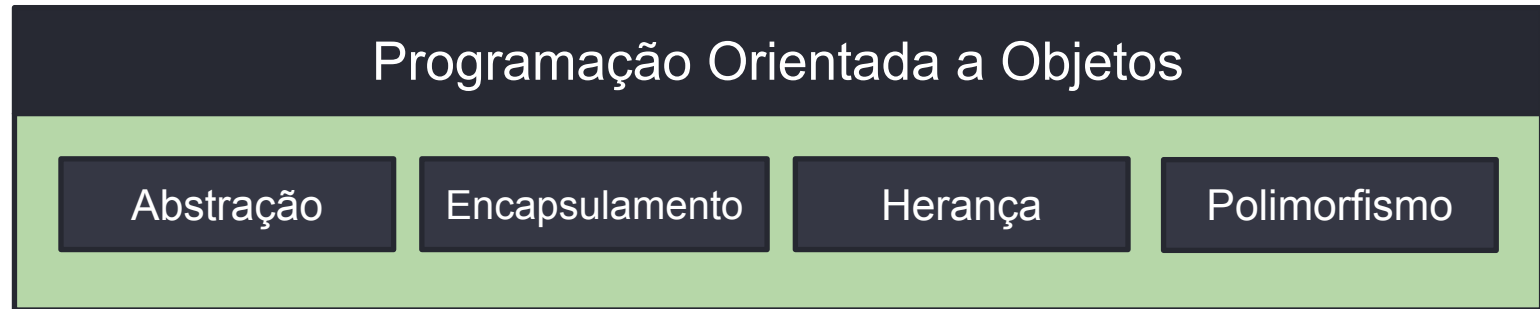


- Um objeto da classe *Aluno* deverá, obrigatoriamente, se relacionar com 1 objeto da classe *Professor*: como a relação omitida, assume-se **1..1**
- Um professor pode não ter alunos ou ter vários: **0..***

Herança

Herança

- Herança é um dos 4 pilares de Programação Orientada a Objetos tradicional:



Reutilização (ou Reuso)

- Reutilização de código é essencial no desenvolvimento de grandes projetos!
- Algumas maneiras:
 - Cópia direta do código (*pior método*)
 - **Herança** (*linguagens Orientadas a Objetos*)

Herança

- É uma forma de reutilização de código!
- O conceito de herança se baseia no princípio de que toda codificação mais genérica pode ser transmitida para classes mais específicas
- **Cria uma nova classe a partir de uma classe existente**
- **Relação é um**

Herança

- Cria uma nova classe como uma **extensão** de uma classe já existente
- Permite utilizar a forma da classe existente, adicionando código, sem destruir a classe existente
- **A nova classe herda os atributos e os métodos da classe existente**

Exemplo - Carros

- Uma **Ferrari** é um **Carro**
- Uma **BMW** é um **Carro**
- Um **Fusca** é um **Carro**

Podemos ter uma classe **Carro** que tem características comuns a todos esses veículos!



A classe carro pode conter os seguintes atributos:

- **rodas**
- **cor**
- **ano de fabricação**
- **km**

Superclasse e Subclasse

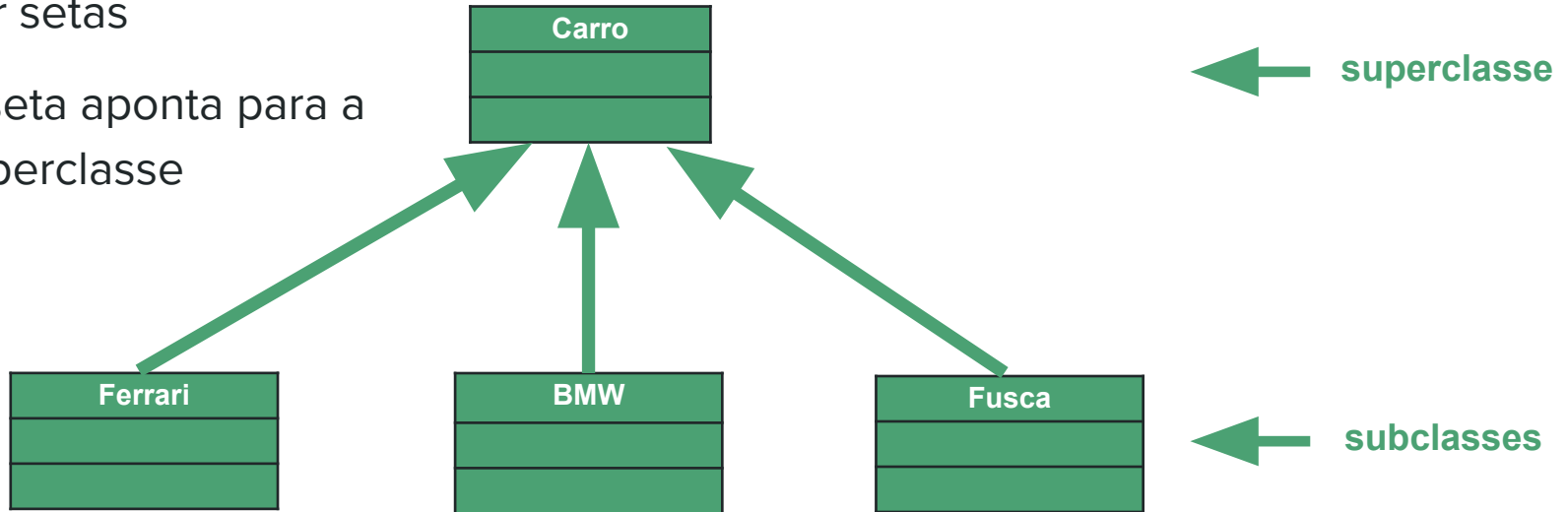
- Considerando os exemplos dados:
 - A classe **Carro** é uma **superclasse** / classe base
 - As classes **Ferrari, BMW, Fusca** são **subclasses** / classes derivadas
- Superclasses tendem a ser mais genéricas
- Subclasses tendem a ser mais específicas

A Herança Permite que as Subclasses

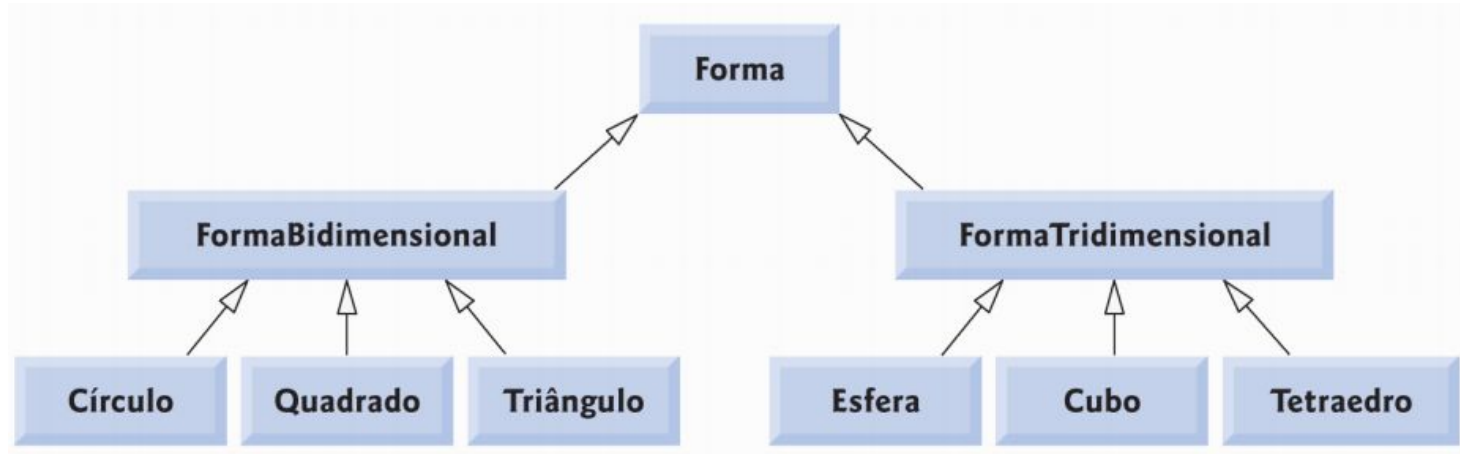
- Herdem os atributos e os métodos da superclasse:
 - atributos e métodos herdados podem ser diretamente utilizados - não é preciso escrevê-los novamente
- Definem novos atributos e métodos
- Modifiquem um método definido na superclasse (sobrescrita - override)

Herança - UML - Diagrama de Classes

- Heranças são indicadas por setas
- A seta aponta para a superclasse



Herança - UML - Diagrama de Classes - Exemplo



Herança

- Quando utilizar a herança? Relação é um
- Realizar a pergunta: “*é um/uma?*”
- Exemplos:
 - **Funcionario** é uma **Pessoa**?
 - **Carro** é um **Veículo**?
 - **Aluno** é uma **Pessoa**?
 - **Gerente** é um **Empregado**?

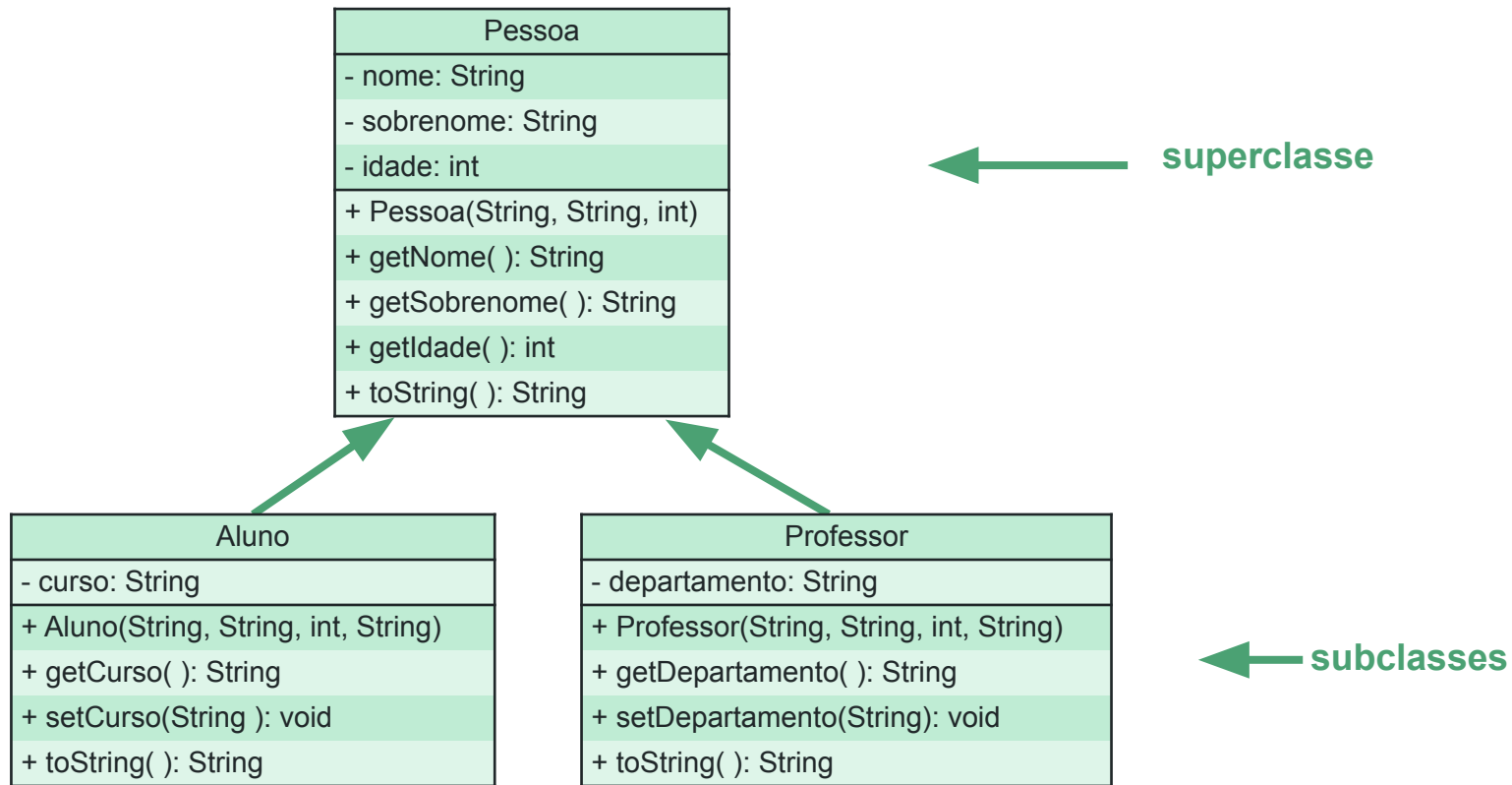
Exemplo

- Duas classes que possuem vários atributos e métodos iguais:

Aluno
- nome: String
- sobrenome: String
- idade: int
- curso: String
+ Aluno(String, String, int, String)
+ getNome(): String
+ getSobrenome(): String
+ getIdade(): int
+ getCurso(): String
+ setCurso(String): void
+ toString(): String

Professor
- nome: String
- sobrenome: String
- idade: int
- departamento: String
+ Professor(String, String, int, String)
+ getNome(): String
+ getSobrenome(): String
+ getIdade(): int
+ getDepartamento(): String
+ setDepartamento(String): void
+ toString(): String

Exemplo de Herança



Herança - Python Sintaxe

- Para criar a Herança no Python:

```
class ClasseDerivada(ClasseBase):  
    <statement-1>  
  
    ...  
  
    <statement-N>
```

- A classe **ClasseDerivada** é uma *subclasse* de **ClasseBase**
- A classe **ClasseBase** é a *superclasse* de **ClasseDerivada**

Declarando uma Classe em Python

```
class Robot:
    def __init__(self):
        self.name = "Tiago++"
        self.positionX = 0.0
        self.positionY = 0.0
        self.direction = 0.0
```

Atributos

```
    def moveForward(self):
        print ("move para frente")
    def moveBackward(self):
        print ("move para trás")
```

Métodos

Declarando uma Classe em Python

Herança

```
class R2d2(Robot):  
    def __init__(self):  
        self.surname = "StarWars"  
  
    def emiteSom(self):  
        print ("bi..bi..pan!")
```

Atributos novos

Métodos novos

Instanciando um objeto da subclasse

```
robo = R2d2()  
robo.name = "R2D2"  
robo.positionX = 15  
robo.positionY = 7  
robo.direction = 1.57  
robo.surname = "droid"  
robo.moveForward()  
robo.emiteSom()
```

move para frente
bi..bi..pan!

Dicas

- Em um grupo de classes relacionadas, coloque os atributos e métodos comuns na superclasse
- Use a herança para criar subclasses sem ter que repetir código
- Herde somente da classe mais parecida com a que você precisa
 - Herdar classes maiores desperdiça memória e processamento

Inicialização

- Construtores não são herdados!
- Se o construtor da superclasse tiver parâmetros, **precisamos invocar o construtor da superclasse** no **construtor da subclasse**
- Devemos inicializar todas as classes utilizadas em uma hierarquia

Herança - Lidando com os construtores (`__init__`)

```
class Robot:
    def __init__(self, name, x, y, direction):
        self.name = name
        self.positionX = x
        self.positionY = y
        self.direction = direction

    def moveForward(self):
        print ("move para frente")
    def moveBackward(self):
        print ("move para trás")
```

Invocando o construtor da
superclasse

```
class R2d2(Robot):
    def __init__(self, name, x, y, direction, surname):
        Robot.__init__(self, name, x, y, direction)
        self.surname = surname

    def emiteSom(self):
        print ("bi..bi..pan!")
```

Herança - Lidando com os construtores

```
robo = R2d2("R2D2", 15, 7, 1.57, "droid")  
print( robo.name, robo.surname )  
robo.moveForward()  
robo.emiteSom()
```

```
R2D2 droid  
move para frente  
bi..bi..pan!
```

Herança - Lidando com os construtores (`__init__`)

```
class Robot:
    def __init__(self, name, x, y, direction):
        self.name = name
        self.positionX = x
        self.positionY = y
        self.direction = direction

    def moveForward(self):
        print ("move para frente")
    def moveBackward(self):
        print ("move para trás")
```

Invocando o construtor da superclasse com `super()` - nesse caso não utilizamos o `self`

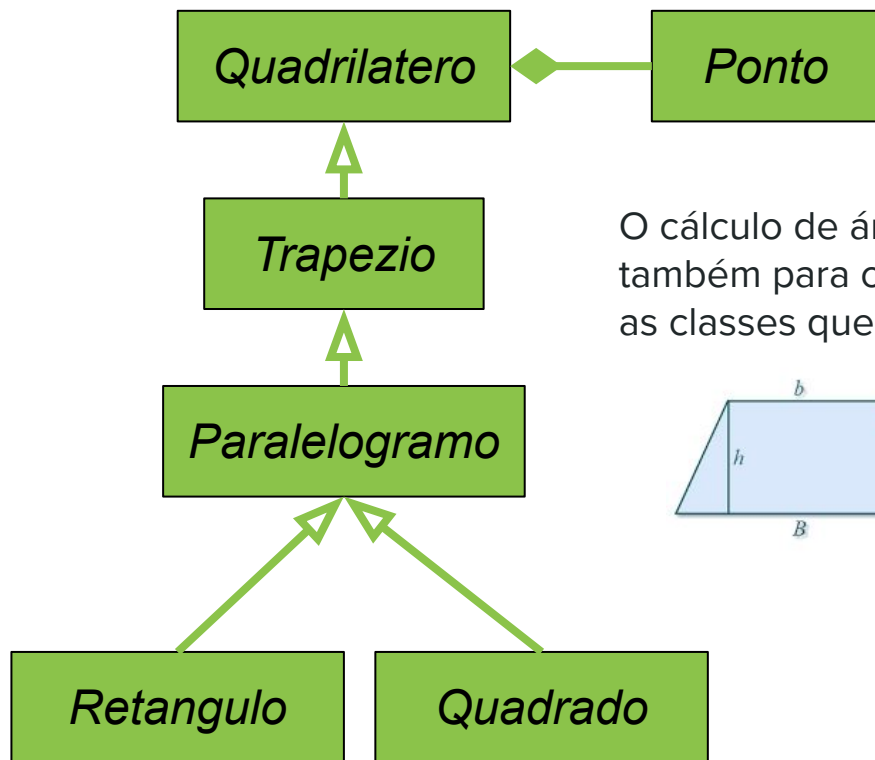
```
class R2d2(Robot):
    def __init__(self, name, x, y, direction, surname):
        super().__init__(name, x, y, direction)
        self.surname = surname

    def emiteSom(self):
        print ("bi..bi..pan!")
```

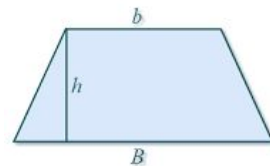
Exercício 2 - Hierarquia de herança - Quadrilátero

(Deitel 9.8) Escreva uma hierarquia de herança para as classes *Quadrilatero*, *Trapezio*, *Paralelogramo*, *Retangulo* e *Quadrado*. Use *Quadrilatero* como superclasse da hierarquia. Crie e use uma classe *Ponto* para representar os pontos (x, y) de cada forma. Faça a hierarquia o mais profunda (isto é, com muitos níveis) possível. Especifique as variáveis de instância e os métodos para cada classe. As variáveis de instância *private* de *Quadrilatero* devem ser *Pontos* para os quatro pontos que delimitam o quadrilátero. Escreva um programa que instancia objetos de suas classes e gera saída da área de cada objeto (exceto *Quadrilatero*). A entrada será feita com a posição de 4 pontos (x, y).

Exercício 2 - Hierarquia de herança - Quadrilátero



O cálculo de área do trapézio serve também para o cálculo de área de todas as classes que herdam de Trapézio

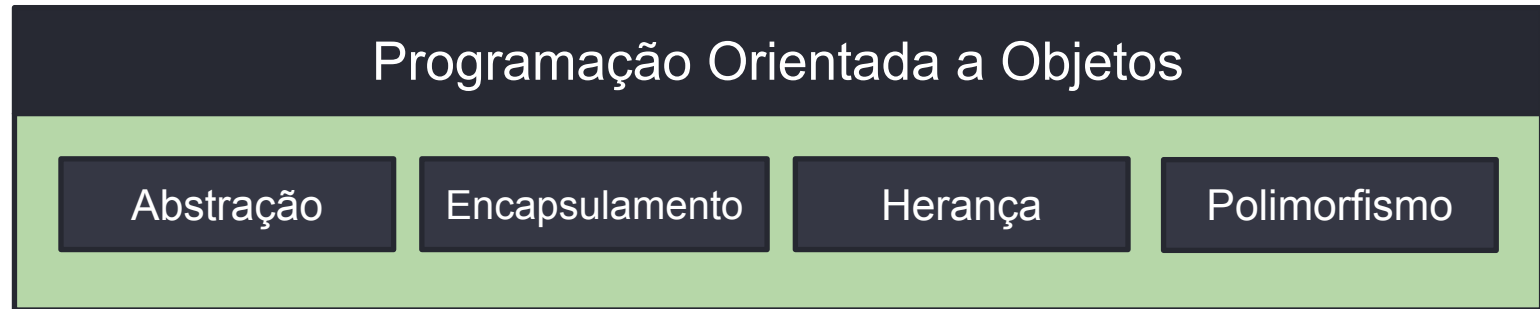


$$A = \frac{(B + b)h}{2}$$

Polimorfismo

Polimorfismo

- Polimorfismo é um dos 4 pilares de Programação Orientada a Objetos tradicional:



Polimorfismo

- **Polimorfismo é a capacidade de um objeto poder ser referenciado de várias formas**
- O polimorfismo está ligado aos conceitos de Herança e Hierarquia de Classes

Método Polimórfico

- **Superclasses** e **subclasses** podem ter métodos com o mesmo nome e parâmetros!
 - Podemos reescrever (*override*) os métodos na subclasse
- Diferentes ações ocorrem, dependendo do objeto instanciado!
- Os **métodos são polimórficos!**

Método Polimórfico

Mesma chamada de método pelo mesmo objeto; porém, funciona de forma diferente pois o objeto mudou

```
class Pessoa:  
    def sayHi(self):  
        print("Olá, sou uma pessoa!")
```

```
class Aluno(Pessoa):  
    def sayHi(self):  
        print("Olá, sou um aluno!")
```

```
objeto = Pessoa()  
objeto.sayHi()  
objeto = Aluno()  
objeto.sayHi()
```

```
Olá, sou uma pessoa!  
Olá, sou um aluno!
```

Método Polimórfico

- O método polimórfico pode ser bastante útil quando uma função/método espera receber um objeto como parâmetro!
- O objeto passado pode mudar em tempo de execução e a função/método pode mudar a sua ação conforme o objeto recebido
- Exemplo em aula

Classes Abstratas

Classes Abstratas

- Nem sempre é desejável que toda a classe possa permitir a criação de objetos!
- Algumas classes servem somente como modelos para outras classes por meio da Herança
 - Exemplo: modelagem dos funcionários de uma empresa
 - *Funcionario* é a superclasse
 - *Gerente*, *Secretario*, *Analista* etc. são subclasses

A ideia nesse projeto é sempre instanciar o funcionário com o seu cargo correto! Não deve existir um funcionário sem cargo!

Classes Abstratas

- Podemos então criar as Classes Abstratas!
- A classe abstrata é sempre uma **superclasse** que não possui instâncias: **não pode ser instanciada**
- Ela define um modelo genérico para determinada funcionalidade e geralmente fornece uma implementação incompleta dessa funcionalidade
- Cada uma das subclasses da classe abstrata completa a funcionalidade da classe abstrata, adicionando um comportamento específico

Classes Abstratas

- Utilizamos o módulo **abc** - *abstract base class*

```
from abc import ABC

class MyABC(ABC):
    pass
```

Classes Abstratas

- Normalmente, com classes abstratas nós utilizamos **métodos abstratos**
 - Métodos abstratos não têm implementação!
 - Definir um método como abstrato é uma maneira de forçar a sua implementação nas subclasses
 - Para definir métodos abstratos utilizamos o decorator ***@abstractmethod***

Classes Abstratas

```
from abc import ABC, abstractmethod
```

```
class Pessoa1(ABC):  
    @abstractmethod  
    def sayHi(self):  
        pass
```

```
p = Pessoa1()
```

TypeError

Traceback (most recent call last)

Cell In [41], line 1

----> 1 p = Pessoa1()

TypeError: Can't instantiate abstract class Pessoa1 with abstract method sayHi

Classes Abstratas

```
class Aluno1(Pessoa1):  
    def sayHi(self):  
        print("Olá! Sou um aluno criado a partir da classe abstrata Pessoa1")
```

```
a = Aluno1()  
a.sayHi()
```

Olá! Sou um aluno criado a partir da classe abstrata Pessoa1

Lista de Objetos

Lista de objetos

```
sala_de_aula = []  
for i in range(100):  
    sala_de_aula.append(Aluno1())
```

```
sala_de_aula[0].sayHi()  
sala_de_aula[1].sayHi()
```

Olá! Sou um aluno criado a partir da classe abstrata Pessoa1

Olá! Sou um aluno criado a partir da classe abstrata Pessoa1

Tratamento de exceções

Tratamento de exceções

- Uma exceção é um problema que acontece em **tempo de execução!**
- É um problema que, normalmente, não pode ser antecipado em um programa
- A exceção precisa ser tratada pelo programador
- Exceções ocorrem quando um método detecta um problema e é incapaz de tratá-lo

Tratamento de exceções - Python

- Exemplo de exceção: Divisão por zero

```
a = 0  
b = 10  
  
print(b/a)
```

```
-----  
ZeroDivisionError                                Traceback (most recent call last)  
Cell In [52], line 4  
      1 a = 0  
      2 b = 10  
----> 4 print(b/a)  
  
ZeroDivisionError: division by zero
```

Tratamento de exceções - Python

- No Python, utilizamos **try** e **except** para capturar e tratar uma exceção

```
try:  
    #bloco com o código a ser executado.  
except <exception>:  
    #bloco para tratar o erro.
```

- **try:**
 - Código que contém um risco de falha
- **except:**
 - Código para tratar uma exceção

Tratamento de exceções - Python

```
a = 0
b = 10

try:
    print(b/a)
except ZeroDivisionError:
    print("Exceção: divisão por zero!")
```

Exceção: divisão por zero!

Tratamento de exceções - Python

Mas podem ocorrer exceções que não foram tratadas pelo programa.

```
try:
    a = int(input("Digite o primeiro número: "))
    b = int(input("Digite o segundo número: "))
    c = a/b
    print(c)
except ZeroDivisionError:
    print("Ocorreu uma divisão por zero")
```

Só captura divisão por zero!

Digite o primeiro número: 10
Digite o segundo número: a

ValueError

Traceback (most recent call last)

<ipython-input-4-6bdcea9493ba> in <module>

```
1 try:
2     a = int(input("Digite o primeiro número: "))
----> 3     b = int(input("Digite o segundo número: "))
4     c = a/b
5     print(c)
```

ValueError: invalid literal for int() with base 10: 'a'

Tratamento de exceções - Python

- Então, podemos agrupar o tratamento de várias exceções:

```
try:
    a = int(input("Digite o primeiro número: "))
    b = int(input("Digite o segundo número: "))
    c = a/b
    print(c)
except (ZeroDivisionError, ValueError):
    print("Ocorreu uma exceção")
```

```
Digite o primeiro número: 10
Digite o segundo número: a
Ocorreu uma exceção
```

Tratamento de exceções - Python

- Também pode-se tratar individualmente as exceções :

```
try:
    a = int(input("Digite o primeiro número: "))
    b = int(input("Digite o segundo número: "))
    c = a/b
    print(c)
except ZeroDivisionError:
    print("Ocorreu uma divisão por zero")
except ValueError:
    print("Foi digitado um valor não numérico")
```

```
Digite o primeiro número: 10
Digite o segundo número: a
Foi digitado um valor não numérico
```

Tratamento de exceções - Python

- Quando não sabemos qual exceção pode ocorrer no programa, podemos capturar a exceção usando somente a palavra **except**:

```
try:
    a = int(input("Digite o primeiro número: "))
    b = int(input("Digite o segundo número: "))
    c = a/b
    print(c)
except ZeroDivisionError:
    print("Ocorreu uma divisão por zero")
except:
    print("Ocorreu um erro")
```

```
Digite o primeiro número: 5
Digite o segundo número: r
Ocorreu um erro
```

Graphical User Interface - GUI

GUI - *Graphical User Interface*

- Interface Gráfica do Usuário
- Por que utilizar *Interface Gráfica do Usuário*?
- Foco no usuário final:
 - Mais amigável
 - Interação mais rápida
 - Mais produtiva

GUI - Frameworks para Python

- Tkinter (Tk interface)
- PyQt
- wxPython
- etc

GUI - Frameworks para Python

- **Tkinter (Tk interface):**
 - **Toolkit padrão do Python para desenvolvimento de GUI**
- PyQt
- wxPython
- etc

GUI - Frameworks para Python

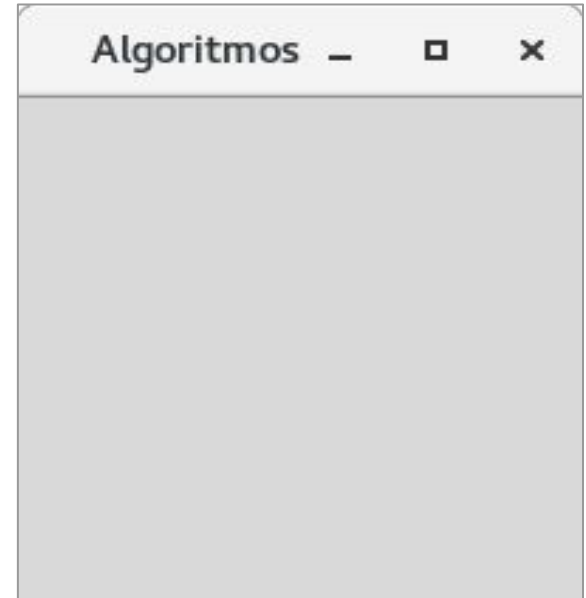
- Vamos usar o **Tkinter**:
 - O **Tkinter** é um pacote interessante que já vem por padrão com o Python.
 - Para usar o Tkinter, só precisamos importá-lo:

```
from tkinter import *
```

- O Tkinter permite a criação de **janelas**, **rótulos**, **botões**, **caixas de texto**, **caixas de mensagem** etc.

GUI - Janela

```
from tkinter import *  
  
# cria a janela  
janela = Tk()  
  
# titulo para a janela  
janela.title("Algoritmos")  
  
# configura o tamanho da janela  
janela.geometry('400x400')  
  
# chama a função mainloop:  
# loop infinito para manter a janela aberta  
janela.mainloop()
```



GUI - Rótulo (*Label*)

```
from tkinter import *

# cria a janela
janela = Tk()

# titulo para a janela
janela.title("Algoritmos")

# configura o tamanho da janela
janela.geometry('400x400')

# cria o rótulo na janela desejada, com o texto desejado e configura a fonte
rotulo = Label(janela, text="Primeira aplicação gráfica no Python!", font=("Arial Bold", 14))

# configura onde o texto vai aparecer na janela:
# x = 200 e y = 100
# a referência é o centro (CENTER) do rótulo
rotulo.place(x=200, y=100, anchor=CENTER)

# chama a função mainloop:
# loop infinito para manter a janela aberta
janela.mainloop()
```



GUI - Posicionamento dos Elementos (*place*)

- O *place* permite que os elementos sejam explicitamente posicionados de forma absoluta ou relativa
- Sintaxe:
 - Posicionando o elemento w, de forma relativa (centralizando):

```
w.place(relx=0.5, rely=0.5, anchor=CENTER)
```

- Posicionando o elemento w de forma absoluta:

```
w.place(x = 50, y = 100, anchor=CENTER)
```

GUI - Tkinter *place*

- *anchor* refere-se ao elemento que está sendo posicionado:
 - Pode assumir os valores (referências cardeais):
 - NW (default), N, NE, E, SE, S, SW, W, CENTER

```
from tkinter import *  
  
window = Tk()  
  
window.title("Algoritmos")  
  
window.geometry('350x200')  
  
btn = Button(window, text="Clique!")  
btn.place(relx = 0.5, rely = 0.5, anchor=CENTER)  
  
btn2 = Button(window, text="Clique2!")  
btn2.place(x = 100, y = 50, anchor=CENTER)  
  
window.mainloop()
```



GUI - Botão (*Button*)

```
from tkinter import *

# cria a janela
janela = Tk()

# titulo para a janela
janela.title("Algoritmos")

# configura o tamanho da janela
janela.geometry('400x400')

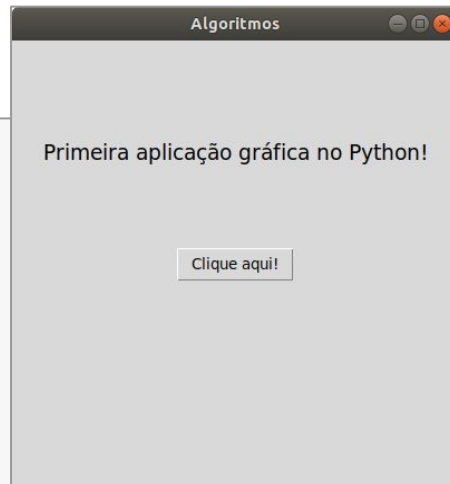
# cria o rótulo na janela desejada, com o texto desejado e configura a fonte
rotulo = Label(janela, text="Primeira aplicação gráfica no Python!", font=("Arial Bold", 14))

# configura onde o texto vai aparecer na janela:
# x = 200 e y = 100
# a referência é o centro (CENTER) do rótulo
rotulo.place(x=200, y=100, anchor=CENTER)

# cria o botão na janela desejada, com o texto desejado
botao = Button(janela, text="Clique aqui!")

# configura onde o botão vai aparecer na janela
botao.place(x=200, y=200, anchor=CENTER)

# chama a função mainloop:
# loop infinito para manter a janela aberta
janela.mainloop()
```



GUI - Botão (*Button*)

- O botão criado no slide anterior **não tem utilidade!**
- Normalmente, o **clique de um botão é associado** a chamada de uma **função!**

GUI - Botão (*Button*)

```
# configura onde o texto vai aparecer na janela:
# x = 200 e y = 100
# a referência é o centro (CENTER) do rótulo
rotulo.place(x=200, y=100, anchor=CENTER)

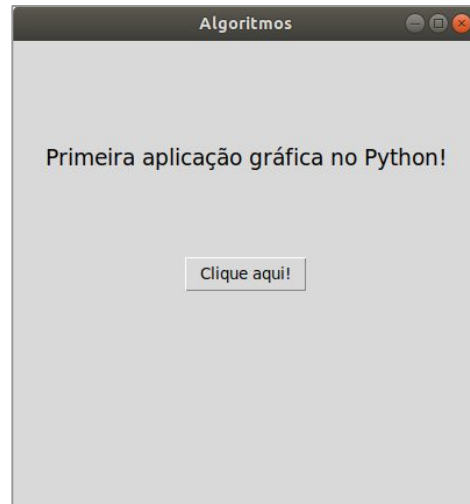
# definição da função clique()
def clique():
    rotulo['text'] = "Novo texto!"

# cria o botão na janela desejada, com o texto desejado e estabelece
# qual a função que será chamada no clique
botao = Button(janela, text="Clique aqui!", command=clique)

# configura onde o botão vai aparecer na janela
botao.place(x=200, y=200, anchor=CENTER)

# chama a função mainloop:
# loop infinito para manter a janela aberta
janela.mainloop()
```

O texto foi alterado depois do clique!



GUI - Caixas de Texto (*Entry*)

```
# configura onde o texto vai aparecer na janela:
# x = 200 e y = 100
# a referência é o centro (CENTER) do rótulo
rotulo.place(x=200, y=100, anchor=CENTER)

# cria o elemento de entrada de texto, configura o tamanho
entrada = Entry(janela, width=14, font=("Arial Bold", 14))
entrada.place(x=200, y=50, anchor=CENTER)

# definição da função clique()
def clique():
    resposta = entrada.get()
    rotulo['text'] = resposta

# cria o botão na janela desejada, com o texto desejado e estabelece
# qual a função que será chamada no clique
botao = Button(janela, text="Clique aqui!", command=clique)

# configura onde o botão vai aparecer na janela
botao.place(x=200, y=200, anchor=CENTER)

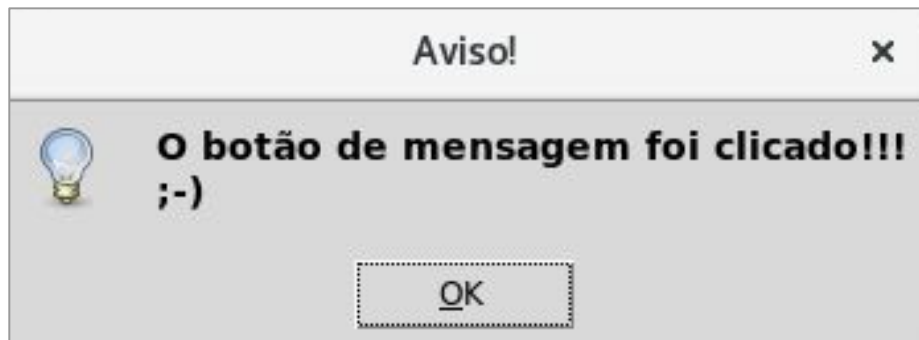
# chama a função mainloop:
# loop infinito para manter a janela aberta
janela.mainloop()
```



GUI - Caixas de Mensagem (*MessageBox*)

```
from tkinter import *  
from tkinter import messagebox
```

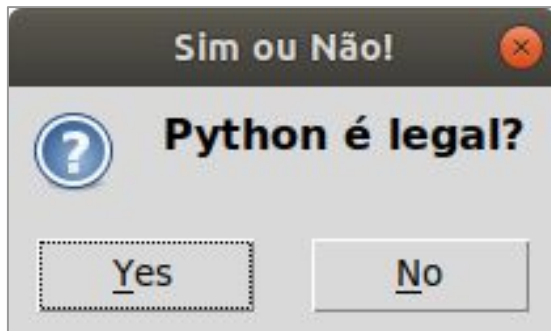
```
def show():  
    res = messagebox.showinfo('Aviso', 'O botão de mensagem foi clicado!!! ;-)')  
    print(res)  
  
botao2 = Button(janela, text='Mensagem', command=show)
```



GUI - Caixas de Mensagem (*MessageBox*)

- Caixas de diálogo com perguntas:

```
def show():  
    res = messagebox.askyesno('Sim ou Não!', 'Python é legal?')  
    print(res)
```



- **res** terá valor **True** ou **False**, dependendo do botão que for clicado.

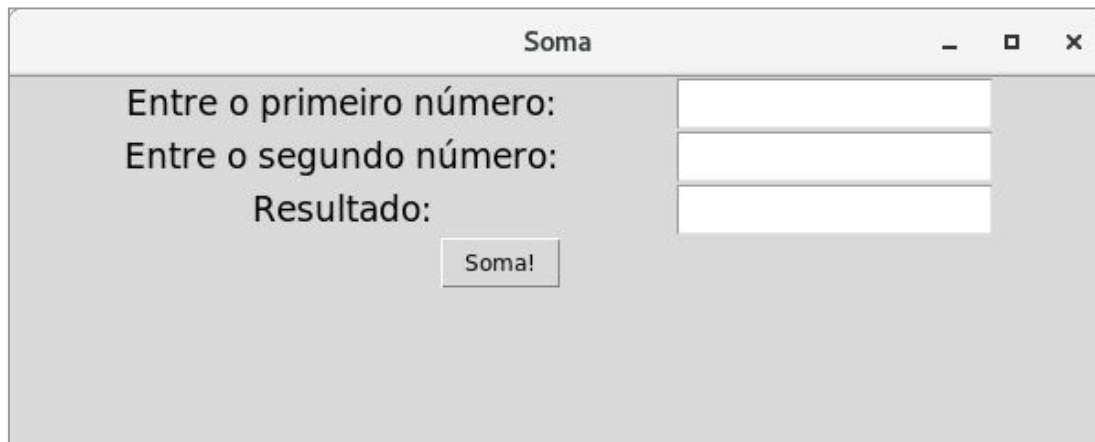
GUI - Caixas de Mensagem (*MessageBox*)

- Mais caixas de diálogo com perguntas:

```
res = messagebox.askquestion('Aviso!', 'O botão de mensagem foi clicado!!! ;-)')  
res = messagebox.askyesnocancel('Aviso!', 'O botão de mensagem foi clicado!!! ;-)')  
res = messagebox.askokcancel('Aviso!', 'O botão de mensagem foi clicado!!! ;-)')  
res = messagebox.askretrycancel('Aviso!', 'O botão de mensagem foi clicado!!! ;-)')
```

Exemplo

Interface gráfica para **somar dois números**:



The image shows a simple graphical user interface window titled "Soma". It has a standard window title bar with minimize, maximize, and close buttons. The main area of the window is light gray. On the left side, there are three labels: "Entre o primeiro número:", "Entre o segundo número:", and "Resultado:". To the right of these labels are three white rectangular input fields. Below the "Resultado:" label, there is a small button labeled "Soma!".

Para atribuir um texto a uma caixa de texto, utilize:

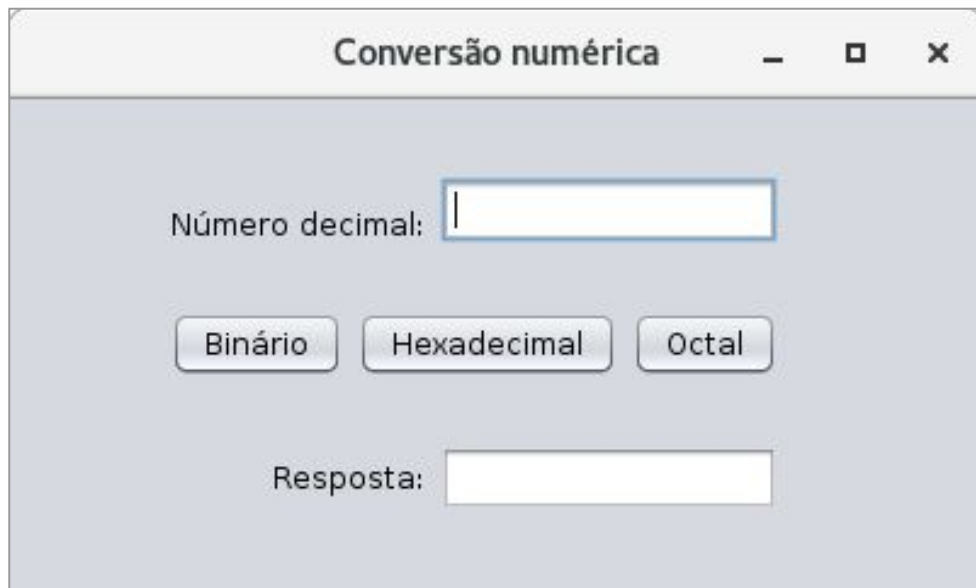
`<nome_da_caixa>.insert(0,<valor>)`



Posição dentro da caixa de texto... se estiver vazia é zero para começar da primeira posição

Exercício 1

Faça uma calculadora para transformar números **decimais** em: **binários**, **hexadecimais** ou **octais**. Cada base numérica deve ter um botão para realizar a conversão.



The image shows a graphical user interface for a numerical conversion tool. The window has a title bar labeled 'Conversão numérica' with standard minimize, maximize, and close buttons. The main area contains a label 'Número decimal:' followed by a text input field. Below this, there are three buttons labeled 'Binário', 'Hexadecimal', and 'Octal'. At the bottom, there is a label 'Resposta:' followed by another text input field.

Exercício 2

Crie um programa que lê uma letra do alfabeto por uma caixa de texto. Se o usuário digitar **a, e, i, o** ou **u**, seu programa deverá exibir uma mensagem indicando que a letra inserida é uma **vogal** (utilize **caixas de mensagem - messagebox**). Se o usuário digitar **y**, seu programa deve exibir uma mensagem indicando que às vezes **y** é uma **vogal** (depende da língua, no inglês, por exemplo), e às vezes **y** é uma **consoante**. Caso contrário, seu programa deve exibir uma mensagem indicando que o letra é uma **consoante**

Entrega

Projeto

Biologia Computacional

Biologia Computacional

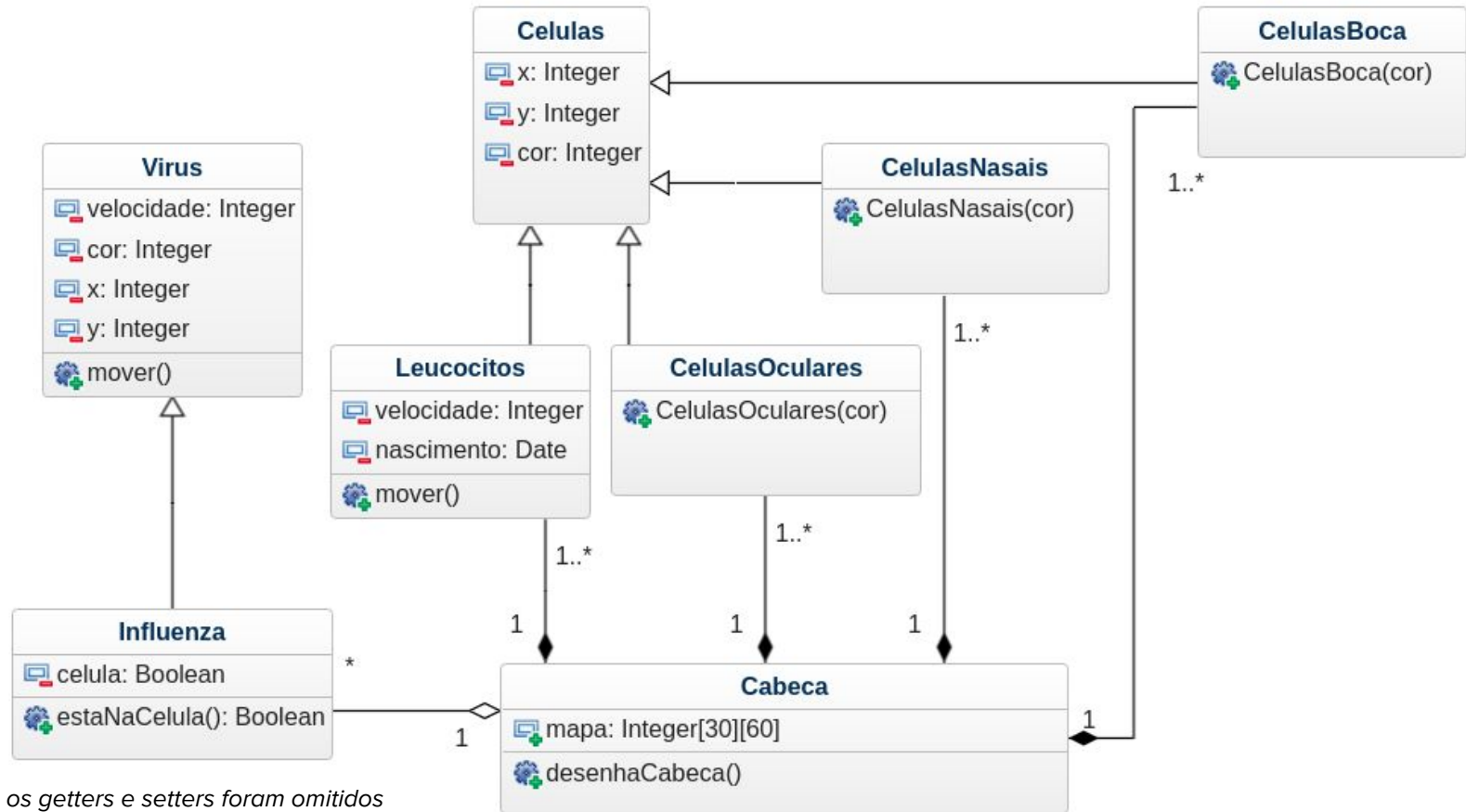
- Biologia Computacional é qualquer técnica computacional desenvolvida e aplicada para estudos da Biologia, Química celular animal ou vegetal, sequenciamento de DNA, simulação de sistemas biológicos e elementos físicos vivos, entre outros.



Projeto

- **Desenvolver um simulador da interação entre um vírus que ataca as vias respiratórias superiores e os leucócitos em uma pessoa**
- O vírus da gripe infecta as células nasais, as células dos globos oculares e as células da cavidade bucal
- Os vírus da gripe devem ser combatidos por leucócitos
- O projeto deve ser feito com base no Diagrama de Classes, dado no próximo slide
- A implementação deve ser feita em **Python**

Diagrama de Classes



Definições da Cabeça

- A classe cabeça é composta por uma **matriz**, com tamanho 30 linhas x 60 colunas: implementar como um **mundo de grades**

Definições da Classe Cabeça

- O atributo mapa, de Cabeça, é cíclico (os objetos móveis devem aparecer na borda oposta quando chegarem em um dos limites de tamanho)
- A Cabeça é composta por dois olhos, um nariz e uma boca. Cada um destes elementos é formado por um conjunto de células de cada respectivo tipo.

Influenza

- Os vírus da gripe (da classe Influenza) deslocam-se pela Cabeça de forma aleatória, sendo que são 4 os possíveis movimentos:
 - **para cima, para direita, para baixo e para esquerda**
- Toda vez que um vírus da gripe (um objeto da classe Influenza) entrar em contato com o nariz, com os olhos ou com a boca, ele é clonado. A cópia deve surgir em um lugar aleatório da cabeça
- *Obs: Somente uma cópia deve ser realizada a cada vez que um objeto vírus entrar em contato com um dos órgãos*

Leucócitos

- Os leucócitos deslocam-se pela Cabeça também de forma aleatória, sendo que são 8 os possíveis movimentos:
 - **para cima, para direita, para baixo, para esquerda e diagonais**
- Toda vez que um vírus da gripe (um objeto da classe Influenza) entrar em contato com um leucócito, o vírus da gripe deve ser deletado. Quando este contato acontece, o leucócito, além de matar o vírus da gripe, deve ser clonado. O clone deve aparecer em uma posição aleatória

Leucócitos

- Todos os objetos leucócitos têm vida útil. Cada leucócito vive por **7 segundos**. Depois de 7 segundos ele deve desaparecer.
- A cabeça nunca fica com menos de 10 leucócitos

Velocidades de movimento

- Influenza: 1 passo por iteração
- Leucócitos: 1 passo por iteração

Interface com o usuário

- A interface com o usuário deve ser feita por meio do console
- A cabeça deve ser exibida com todos os objetos móveis e não móveis
- Além da cabeça, o número atual de leucócitos e de vírus devem ser apresentados

Para construir o mundo colorido no console, procure por **ANSI ESCAPE**

Início

- O sistema deve iniciar com 10 leucócitos e 5 vírus Influenza em posições aleatórias

Exemplo de Funcionamento

Tempo de simulação: 3

■ Influenza: 15 ■ Leucocitos: 12

