



CCM310

Arquitetura de Software e

Programação Orientada a Objetos

Profa. Dra. Gabriela Biondi

Prof. Dr. Isaac Jesus

Prof. Dr. Luciano Rossi

Construtores

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**

Construtores - Definição

- O **construtor** tem o **mesmo nome** da **classe**
- Exemplo:
 - Classe Soma
 - Construtor: **Soma()**
- O construtor não pode retornar um **valor** (nem mesmo *void*)

Construtores no Java

- **Construtor-padrão:** não tem argumentos; pode ser explícito ou é chamado quando nenhum construtor é declarado
- **Construtor parametrizado:** Aceita um ou mais parâmetros

Construtores no Java

Construtor-padrão / sem argumento

```
1 class Student
2 {
3     private String name;
4     private int age;
5     private String branch;
6
7     public Student( ) //construtor
8     {
9         age = 10; branch = "QRT";
10    }
11 }
12
13
14 class TesteStudent
15 {
16     public static void main( String args[])
17     {
18         Student s1 = new Student( );
19     }
20 }
```

Construtor parametrizado

```
1 class Student
2 {
3     private String name;
4     private int age;
5     private String branch;
6
7     public Student( int a, String b ) //construtor
8     {
9         age = a; branch = b;
10    }
11 }
12
13
14 class TesteStudent
15 {
16     public static void main( String args[])
17     {
18         Student s1 = new Student(21, "CSE");
19     }
20 }
```

Sobrecarga de construtores

- Construtores sobrecarregados são construtores que tem o **mesmo nome** com **assinaturas diferentes**



A assinatura de um construtor é constituída pelo seu **nome** e pela lista de seus parâmetros, considerando: **tipo, número e ordem**

```
1 class Student
2 {
3     public String name;      //deveria ser private
4     public int age;          //deveria ser private
5     public String branch;    //deveria ser private
6
7     public Student( ) //construtor
8     {
9         age = 10; branch = "QRT";
10    }
11
12    public Student( int a, String b, String c)
13    {
14        age = a; branch = b; name = c;
15    }
16 }
17
18
19 class TesteStudent
20 {
21     public static void main( String args[])
22     {
23         Student s1 = new Student( );
24         Student s2 = new Student( 15, "CSE", "Fulano" );
25
26         System.out.println("s1" + " " + s1.age + " " + s1.name + " " + s1.branch);
27         System.out.println("s2" + " " + s2.age + " " + s2.name + " " + s2.branch);
28
29     }
30 }
```

```
perico@nuc:~/Dropbox/CC3642 - Orientação a Objetos/Aula 3$ javac student.java
perico@nuc:~/Dropbox/CC3642 - Orientação a Objetos/Aula 3$ java TesteStudent
s1 10 null QRT
s2 15 Fulano CSE
perico@nuc:~/Dropbox/CC3642 - Orientação a Objetos/Aula 3$
```

operador
condicional
(?)

```

4 public class Time2
5{
6    private int hour; // 0 - 23
7    private int minute; // 0 - 59
8    private int second; // 0 - 59
9
10   // construtor sem argumento Time2 : inicializa cada variável de instância
11   // com zero; assegura que objetos Time2 iniciam em um estado consistente
12   public Time2()
13   {
14       this( 0, 0, 0 ); // invoca o construtor Time2 com três argumentos
15   } // fim do construtor sem argumento Time2
16
17   // Construtor Time2: hora fornecida, minuto e segundo padronizados para 0
18   public Time2( int h )
19   {
20       this( h, 0, 0 ); // invoca o construtor Time2 com três argumentos
21   } // fim do construtor de um argumento Time2
22
23   // Construtor Time2: hora e minuto fornecidos, segundo padronizado para 0
24   public Time2( int h, int m )
25   {
26       this( h, m, 0 ); // invoca o construtor Time2 com três argumentos
27   } // fim do construtor de dois argumentos Time2
28
29   // Construtor Time2: hour, minute e second fornecidos
30   public Time2( int h, int m, int s )
31   {
32       setTime( h, m, s ); // invoca setTime para validar a data/hora
33   } // fim do construtor de três argumentos Time2
34
35   // Construtor Time2: outro objeto Time2 fornecido
36   public Time2( Time2 time )
37   {
38       // invoca o construtor de três argumentos Time2
39       this( time.getHour(), time.getMinute(), time.getSecond() );
40   } // fim do construtor Time2 com um argumento de objeto Time2
41
42   // Métodos set
43   // configura um novo valor de data/hora usando UTC; assegura que
44   // os dados permaneçam consistentes configurando valores inválidos como zero
45   public void setTime( int h, int m, int s )
46   {
47       setHour( h ); // configura hour
48       setMinute( m ); // configura minute
49       setSecond( s ); // configura second
50   } // fim do método setTime

```

```

52   // valida e configura a hora
53   public void setHour( int h )
54   {
55       hour = ( ( h >= 0 && h < 24 ) ? h : 0 );
56   } // fim do método setHour
57
58   // valida e configura os minutos
59   public void setMinute( int m )
60   {
61       minute = ( ( m >= 0 && m < 60 ) ? m : 0 );
62   } // fim do método setMinute
63
64   // valida e configura os segundos
65   public void setSecond( int s )
66   {
67       second = ( ( s >= 0 && s < 60 ) ? s : 0 );
68   } // fim do método setSecond

```

```

1 // Fig. 8.6: Time2Test.java
2 // Construtores sobrecarregados utilizados para inicializar objetos Time2.
3
4 public class Time2Test
5 {
6     public static void main( String args[] )
7     {
8         Time2 t1 = new Time2(); // 00:00:00
9         Time2 t2 = new Time2( 2 ); // 02:00:00
10        Time2 t3 = new Time2( 21, 34 ); // 21:34:00
11        Time2 t4 = new Time2( 12, 25, 42 ); // 12:25:42
12        Time2 t5 = new Time2( 27, 74, 99 ); // 00:00:00
13        Time2 t6 = new Time2( t4 ); // 12:25:42
14

```

Exemplo - classe Funcionario aprimorada

Crie a classe *Funcionario* com o seguinte:

- atributos: **nome** (string), **sobrenome** (string), **salario mensal** (double), **idade** (int), **sexo** (string), **numero** (int)
- Faça um **construtor padrão** (sem parâmetro nenhum) para inicializar o objeto
- Faça um **construtor com 6 parâmetros**, um para cada atributo
- Teste instanciando vários objetos da classe *Funcionario*

Destruitor

Destruitor

- Não existe o conceito de destrutores em Java
- O Garbage Collector (GC) irá destruir o objeto na hora que ele achar conveniente e o programador não tem controle sobre isso.

Exercício - Classe Retângulo aprimorada

(Deitel 8.14) Crie a classe Retângulo mais sofisticada. A classe deve armazenar as coordenadas cartesianas dos quatro vértices do retângulo. O construtor chama o método set que aceita quatro conjuntos de coordenadas (8 valores no total: 4 xs e 4 ys) e verifica se cada um deles está no primeiro quadrante, limitados a um valor superior igual a 20.0. O método set também verifica se as coordenadas fornecidas especificam um retângulo. A classe deve ter também métodos para calcular a comprimento, a largura e o perímetro do retângulo. O comprimento é a maior das duas dimensões.

Exercício - Classe Carro

- Crie a classe Carro com os seguintes atributos: modelo, cor, ano, preço e km (todos privados)
- Crie métodos get e set para todos os atributos
- Crie um construtor padrão
- Crie um construtor parametrizado para inicializar 3 atributos de carro (podem escolher quais atributos)
- Crie um construtor parametrizado para inicializar todos os atributos de carro
- Teste no main instanciando objetos da classe Carro

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

Breve Histórico

- Na década de 1980 empresas começaram a utilizar a POO para aplicativos
- Assim, em 1990, basicamente cada empresa tinha o seu próprio processo de desenvolvimento
- Contudo, fornecedores de softwares tinham dificuldades para atender tantos padrões diferentes
- Logo, verificou-se a necessidade de um processo padrão de OO
- Em 1996 foram liberadas as primeiras versões da UML
- Em 1997, a OMG assumiu a responsabilidade pela manutenção e revisão da UML

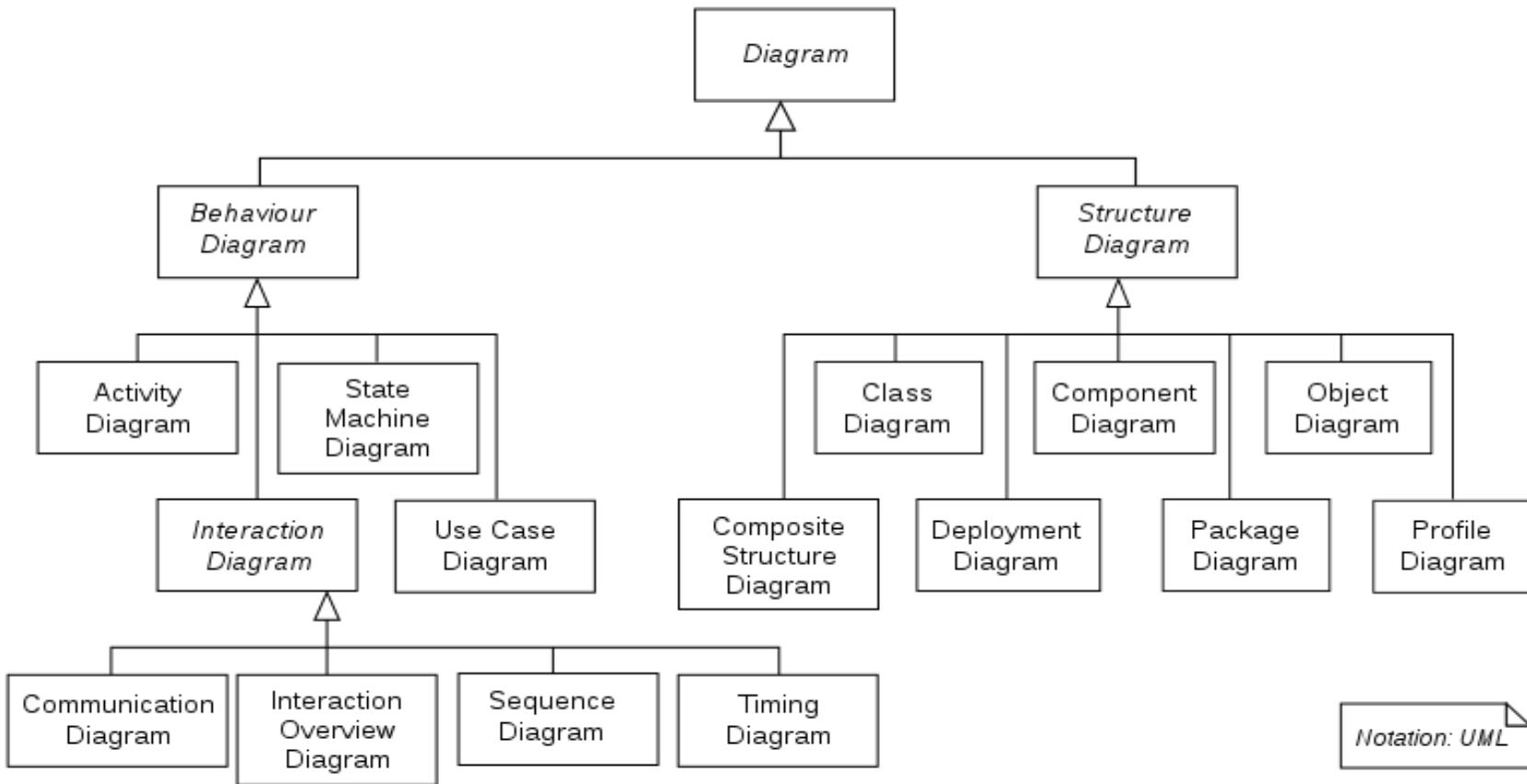
UML

- 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
- Como ela é feita com base nos conceitos de OO, se encaixa perfeitamente com linguagens como Java e C++

UML

- Versão atual: 2.5.1 (*Desde Dez/2017*)
- Conforme a **OMG***, a **UML** possui 15 tipos de diagramas, divididos em duas grandes categorias:
 - Estruturais (7 diagramas) e
 - Comportamentais (8 diagramas)

*<https://www.omg.org/spec/UML/About-UML/>



Um dos mais
utilizados*

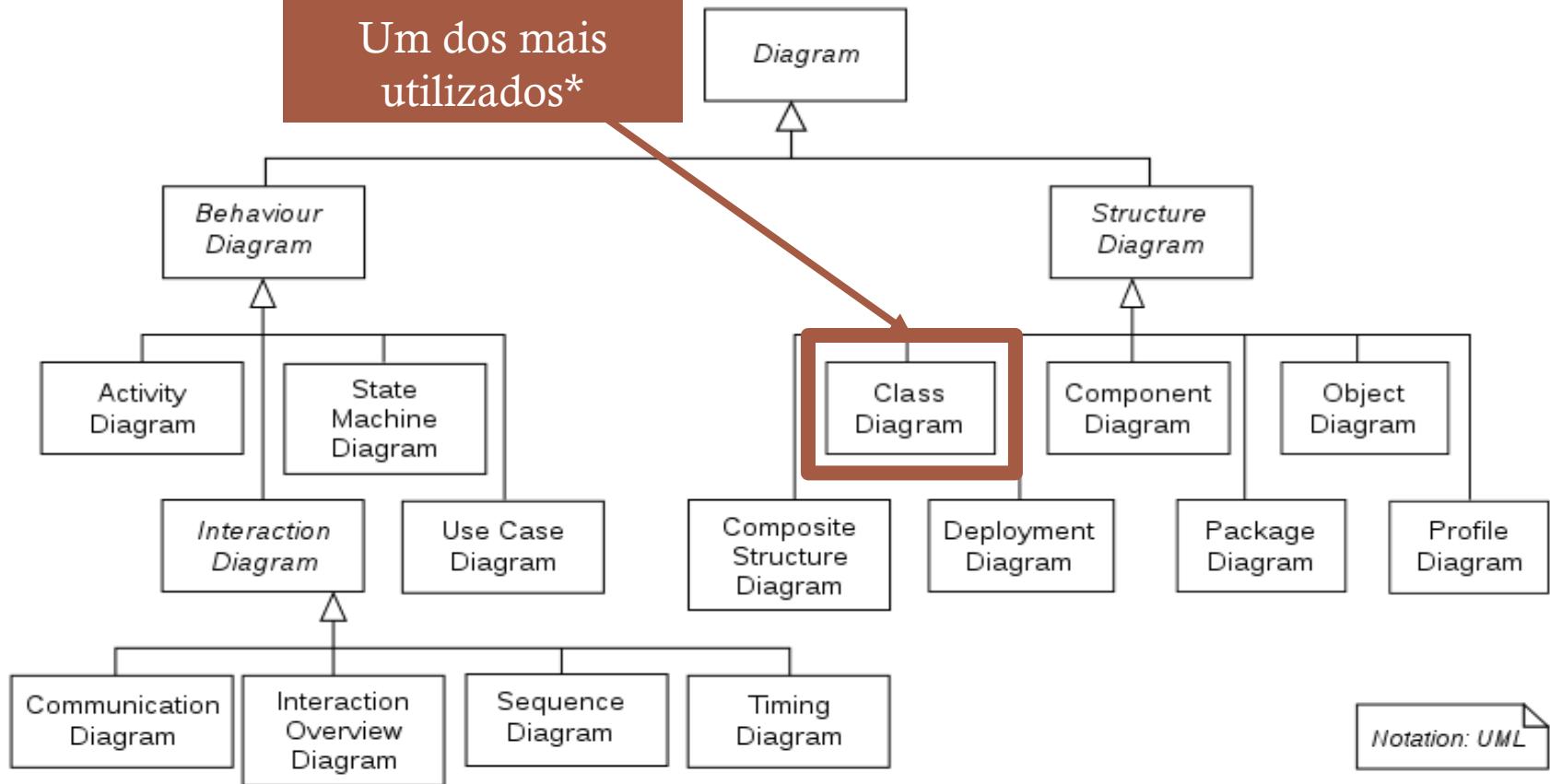


Diagrama de Classes

- 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

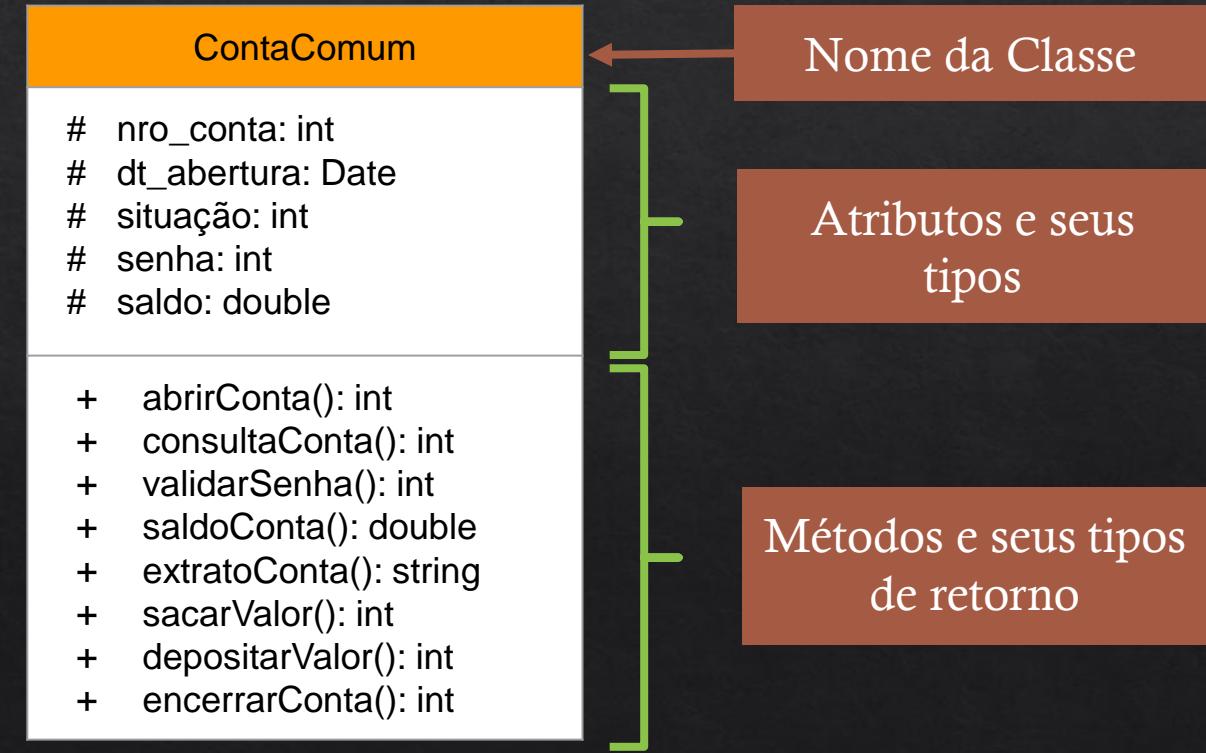


Diagrama de Classes

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: **#** (sustentado)

Diagrama de Classes

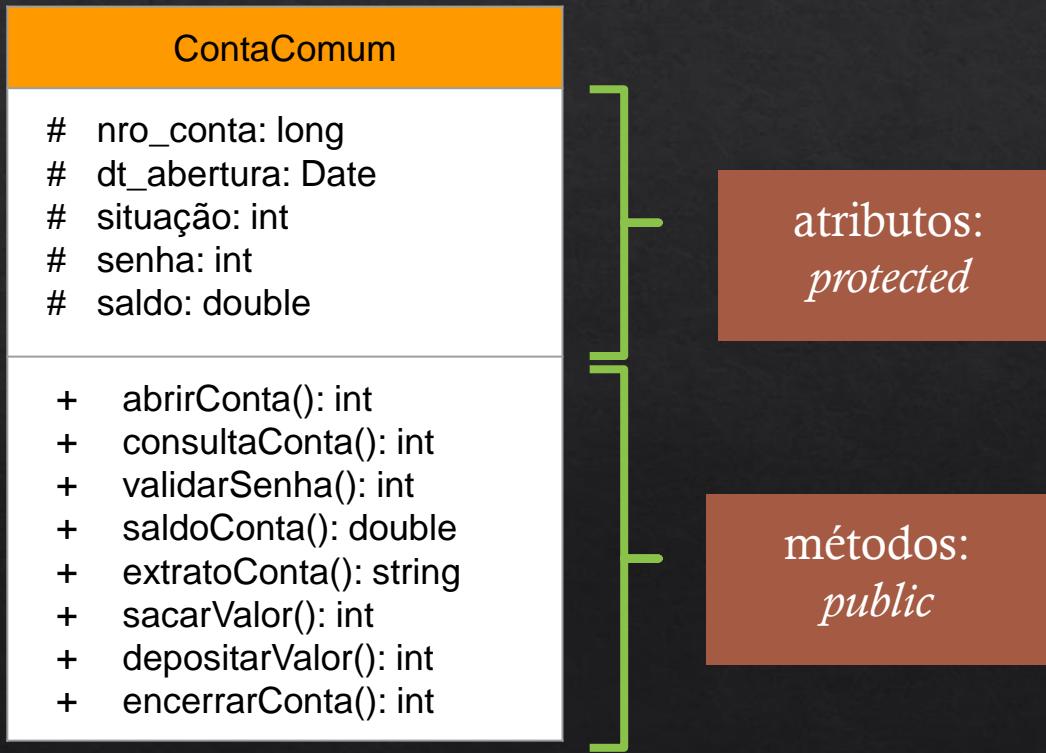


Diagrama de Classes

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 - 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..***

Diagrama de Classes - Agregação

- Informações de um objeto (**objeto-todo**) precisam ser complementadas pelas informações contidas em um ou mais objetos de outra classe (**objetos-parte**)

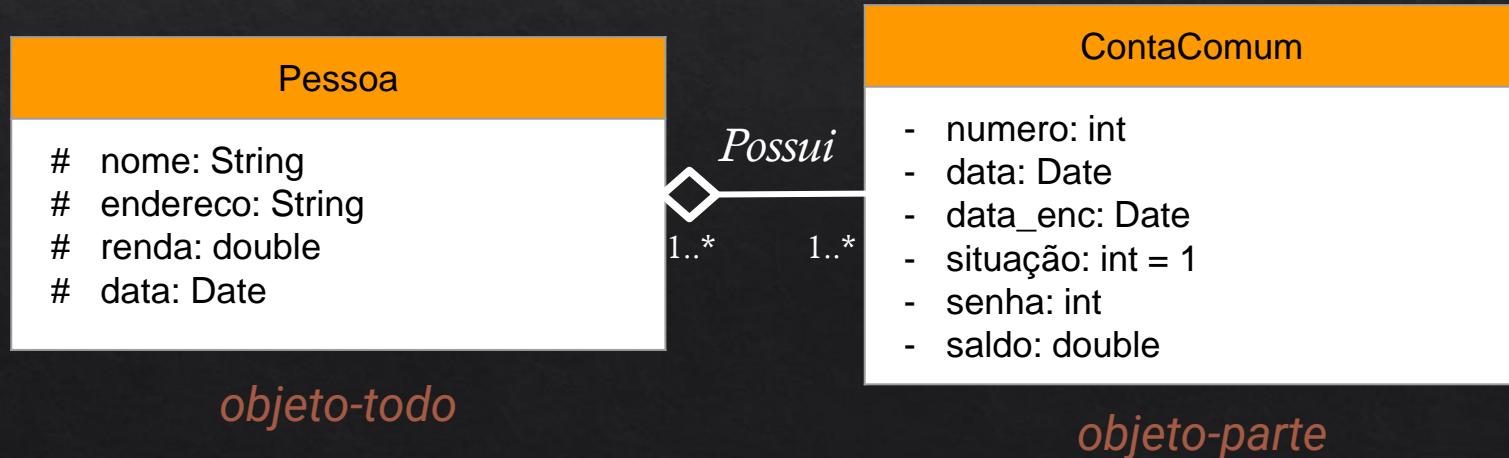
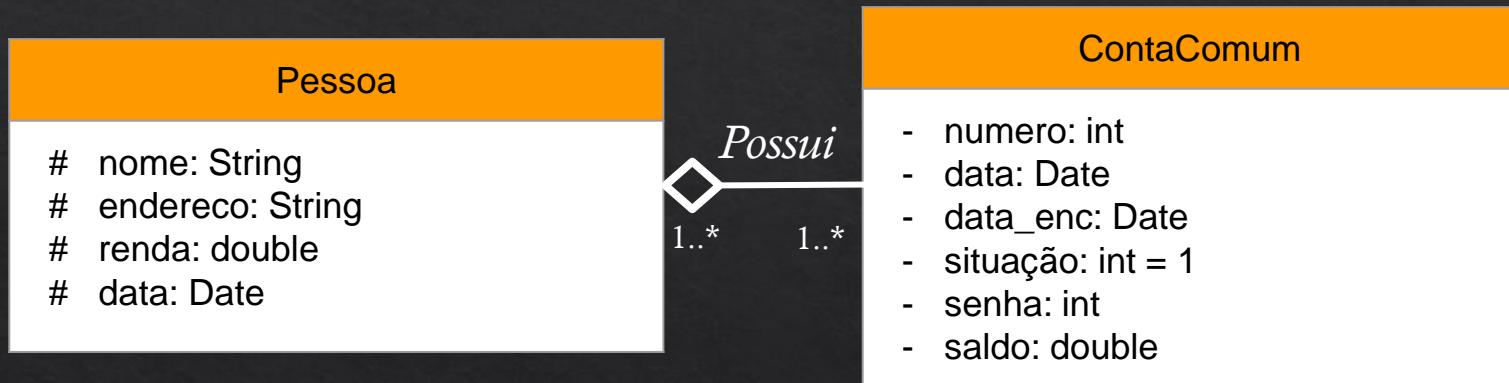


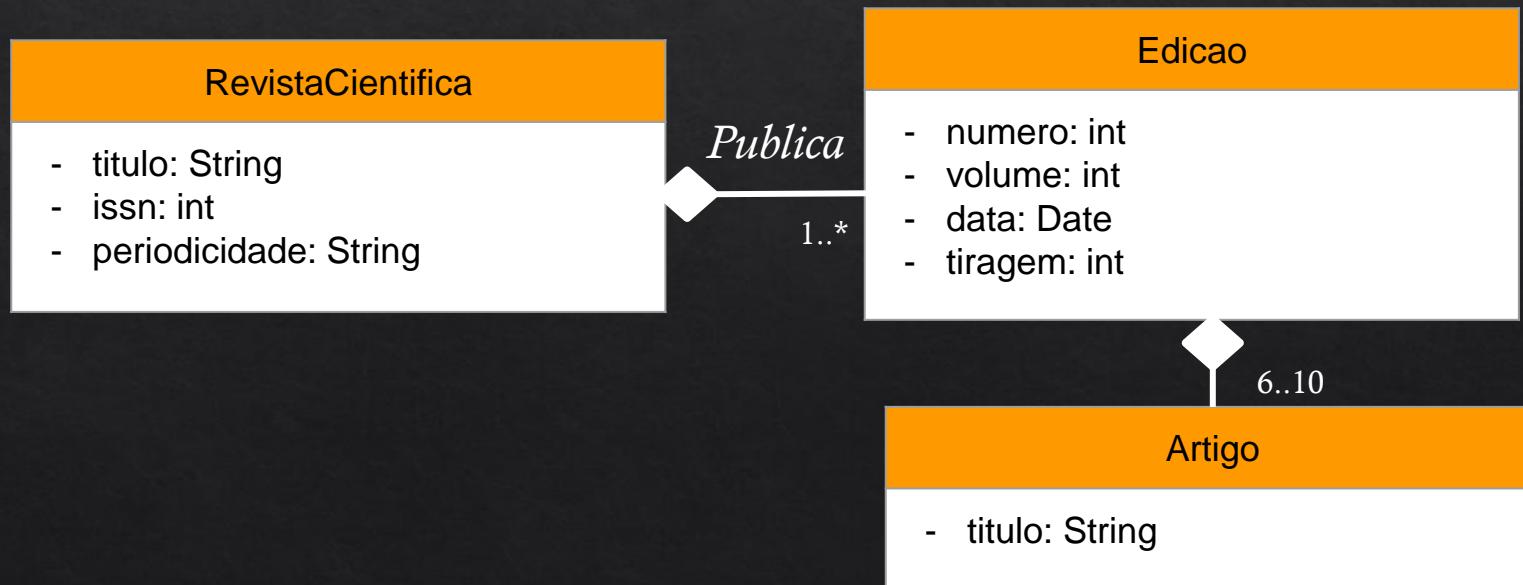
Diagrama de Classes - Agregação



- As informações dos objetos da classe *Pessoa* (*objeto-todo*) precisam ser complementadas por objetos da classe *ContaComum* (*objetos-parte*): **relação tem um**
- Sempre que uma pessoa for consultada, serão apresentadas também todas as contas que ela possui.

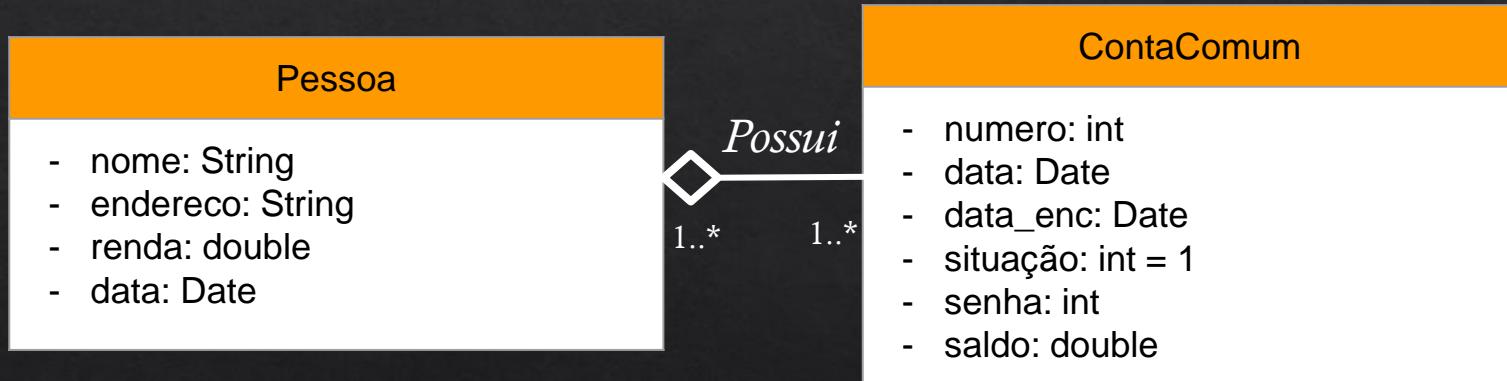
Diagrama de Classes - Composição

- Variação da agregação em que os vínculos são mais fortes:
 - objetos-parte têm de estar associados a **um único** objeto-todo.



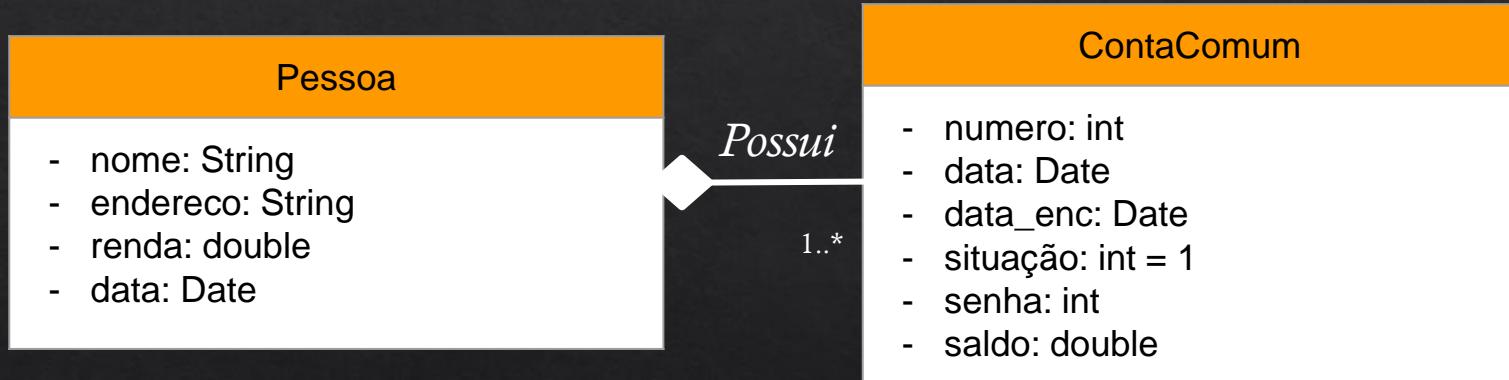
Exemplo 1 - Agregação

- Implemente a seguinte agregação:



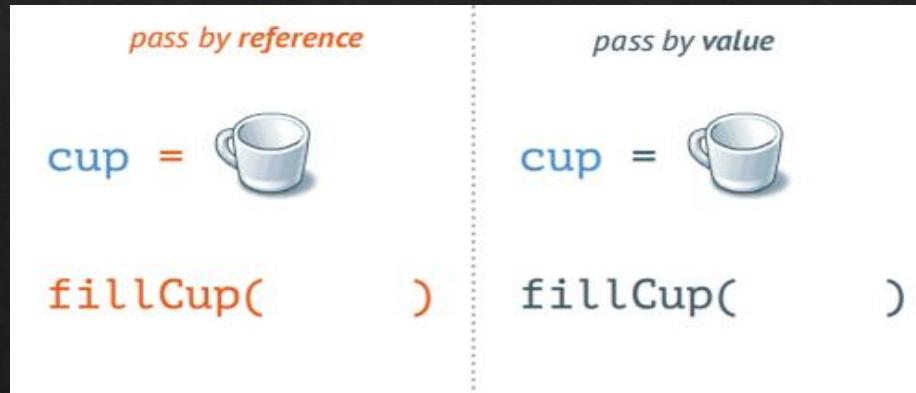
Exemplo 2 - Composição

- Implemente a seguinte composição:



Java - Passagem de Argumentos

- Java não permite que o programador escolha entre passar por valor ou referência
- Variáveis dos tipos primitivos sempre são passadas por **valor**
- Os **objetos** são passados por referência, sendo que as próprias referências são passadas por valor



Exercício 1 - Classe Data aprimorada

(Deitel 8.16) Crie uma classe **Data** com as seguintes capacidades:

A. Gerar saída em múltiplos formatos:

- i. MM/DD/YYYY
- ii. Março 02, 2019
- iii. DDD YYYY

B. Utilizar construtores sobrecarregados para criar objetos **Data** inicializados com datas nos formatos da parte (A). No primeiro caso, o construtor deve receber 3 valores inteiros. No segundo caso deve receber uma String e dois valores inteiros. No terceiro caso deve receber dois valores inteiros, o primeiro sendo o número de dias no ano. [Dica: para converter a representação de string do mês em valor numérico, compare as strings utilizando o método **equals**. Por exemplo, se *s1* e *s2* forem strings, a chamada de método *s1.equals(s2)* retornará true se as strings forem idênticas.]

Exercício 2 - Classe Racional

(Deitel 8.17) Crie uma classe chamada *Racional* para realizar aritmética com frações. Utilize variáveis do tipo inteiro para representar as variáveis de instância *private* da classe: o *numerador* e o *denominador*. Forneça um construtor que permita que um objeto dessa classe seja inicializado quando ele for declarado. 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. Forneça um construtor sem argumento com valores padrão caso nenhum inicializador seja fornecido. Forneça métodos *public* 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.

Exercício 2 - Classe Racional

- c) Multiplicar dois números *Racional*: o resultado da multiplicação deve ser armazenado na forma reduzida.
- 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 , onde a é o numerador e b é o denominador.
- f) Retornar uma representação *String* de um número *Racional* no formato de ponto flutuante. (Considere a possibilidade de fornecer capacidades de formatação que permitam que o usuário da classe)

Escreva um programa para testar sua classe.

Exercício 3 - Classe HeartRates

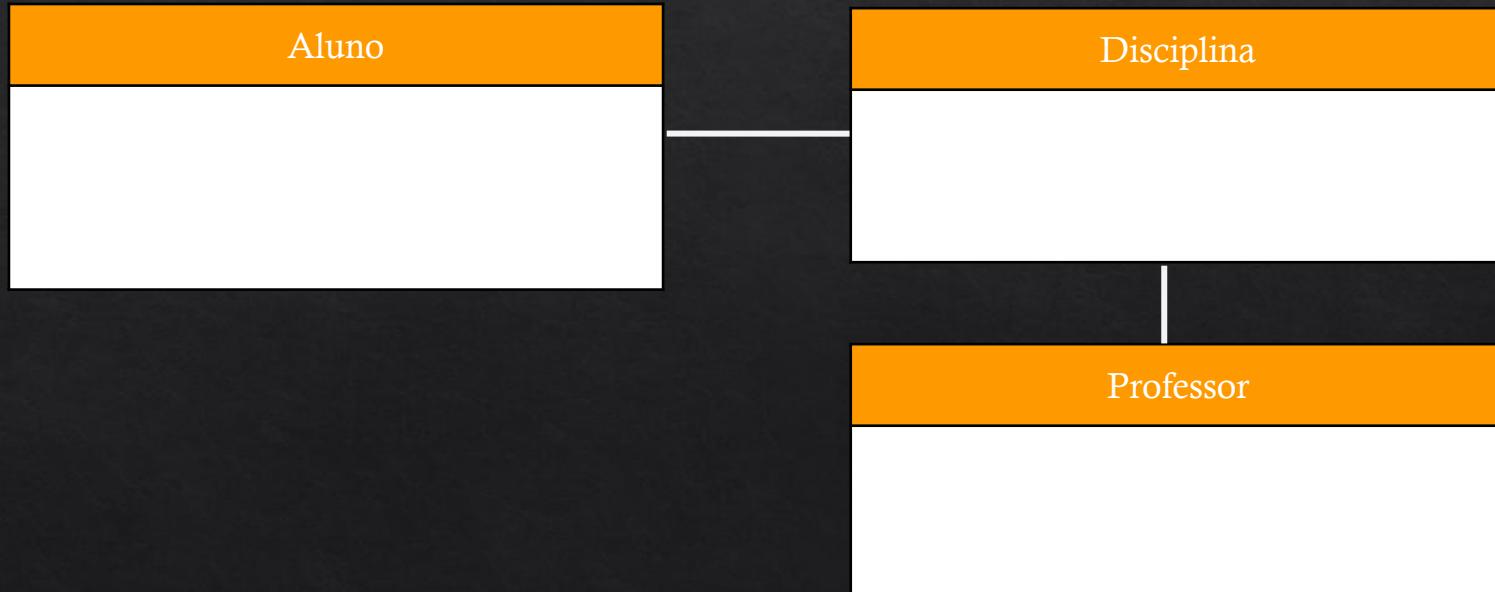
3.16 (Calculadora da frequência cardíaca) Durante o exercício, você pode usar um monitor de freqüência cardíaca para verificar se sua frequência cardíaca permanece dentro de um intervalo seguro sugerido por seus treinadores e médicos. A fórmula para calcular sua frequência cardíaca máxima em batimentos por minuto é 220 menos a sua idade em anos. A sua frequência cardíaca alvo é um intervalo que representa 50 a 85% da sua frequência cardíaca máxima. Crie uma classe chamada *HeartRates*. Os atributos da classe devem incluir o nome, sobrenome e data de nascimento da pessoa (que consistem em atributos separados para o mês, dia e ano de nascimento). Sua classe deve ter um construtor que receba esses dados como parâmetros. Para cada atributo, forneça os métodos set e get. A classe também deve incluir um método que calcula e retorna a idade da pessoa (em anos), um método que calcula e retorna a frequência cardíaca máxima da pessoa e um método que calcula e retorna a freqüência cardíaca alvo da pessoa.

Exercício 3 - Classe HeartRates

Escreva um aplicativo Java que solicite as informações da pessoa, instancia um objeto da classe HeartRates e imprima as informações desse objeto - incluindo o nome, sobrenome e data de nascimento da pessoa - e calcule e imprima a idade da pessoa em (anos), freqüência cardíaca máxima e faixa de freqüência cardíaca alvo.

Exercício 4 - UML

Construa a relação entre as classes / objetos a seguir. Escolha fazer com Composição ou Agregação. Implemente e faça o teste no main. Deixe claro nos comentários do código qual foi a sua escolha (**composição ou agregação**) e o motivo da escolha.



Obrigada pela sua
participação, nos vemos na
próxima aula! :)