Modificadores de Acesso e Construtores

Programação Orientada a Objetos — QXD0007

UNIVERSIDADE FEDERAL DO CEARÁ

CAMPUS QUIXADÁ

Prof. Atílio Gomes Luiz

gomes.atilio@ufc.br

Universidade Federal do Ceará

1º semestre/2022

Leituras para esta aula

- **Capítulo 3** (Introdução a classes, objetos e métodos) do livro Java Como Programar, Décima Edição.

- **Capítulos 4 e 5** da apostila da Caelum – Curso FJ-11.

Pilares da Orientação a Objetos

1. Abstração

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

Abstração é a habilidade de concentrar nos aspectos essenciais de um contexto qualquer, ignorando características menos importantes.

Processo pelo qual se isolam atributos de um objeto, considerando os que certos grupos de objetos tenham em comum.

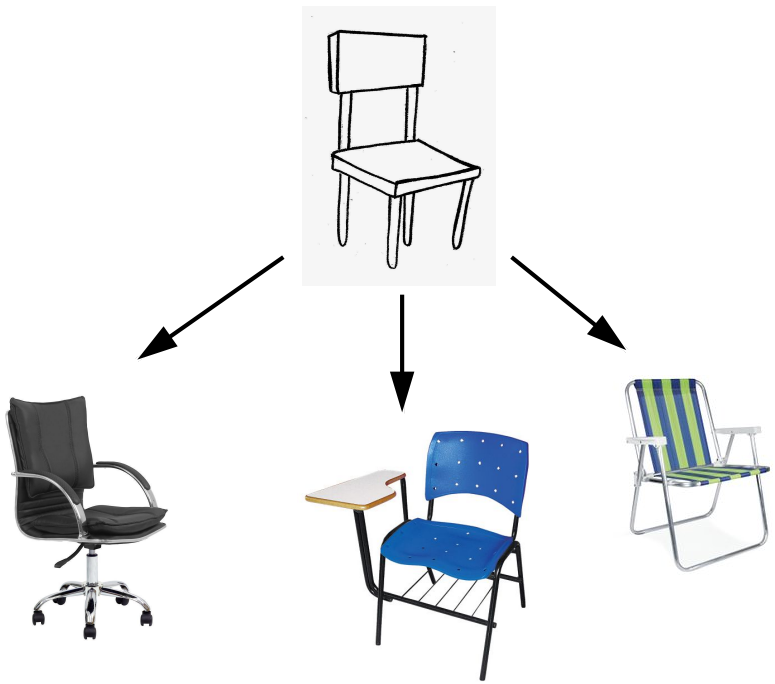
1. Abstração

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

Abstração é a habilidade de concentrar nos aspectos essenciais de um contexto qualquer, ignorando características menos importantes.

Processo pelo qual se isolam atributos de um objeto, considerando os que certos grupos de objetos tenham em comum.

- Por natureza, as abstrações são incompletas e imprecisas.
  - Esta é sua grande vantagem, pois nos permite, a partir de um contexto inicial, modelar necessidades específicas.



2. Reúso

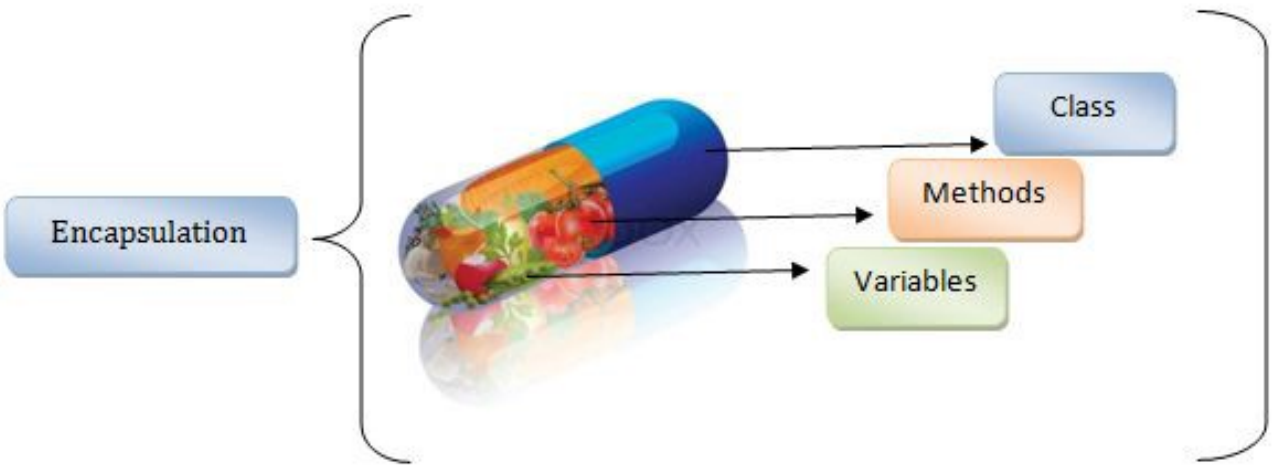
Reúso: Ato ou efeito de reusar; reutilização.

- A OO provê conceitos que visam facilitar a aplicação do reúso de software.
  - **Herança:** é possível criar classes a partir de outras classes, reaproveitando código.
  - **Associação:** Uma classe pede ajuda a outra para poder fazer o que ela não consegue fazer. Em vez de repetir, em si, o código que está em outra classe, a associação permite que uma classe forneça uma porção de código a outra.

3. Encapsulamento (ou Ocultação de dados)

Ocultação de dados

- Em muitos casos será desejável que os dados não possam ser acessados ou usados diretamente, mas somente através das operações cuja especialidade será a manipulação destes dados.



3. Encapsulamento

- **Uma analogia:** Considere uma câmera fotográfica automática. Quando um usuário da câmera clica o botão para tirar uma foto, diversos mecanismos entram em ação que fazem com que a velocidade e abertura apropriada do obturador sejam selecionadas, levando em conta o tipo do filme e as condições de iluminação.

3. Encapsulamento

- **Uma analogia:** Considere uma câmera fotográfica automática. Quando um usuário da câmera clica o botão para tirar uma foto, diversos mecanismos entram em ação que fazem com que a velocidade e abertura apropriada do obturador sejam selecionadas, levando em conta o tipo do filme e as condições de iluminação.

- Para o usuário, os detalhes de como os dados como velocidade, abertura, tipo de filme e iluminação são processados são irrelevantes, o que interessa a ele é que a foto seja tirada.

3. Encapsulamento

Regra de ouro da POO: Sempre que existir uma maneira de deixar ao modelo a capacidade e responsabilidade pela modificação de um de seus dados, devemos criar uma operação para fazê-lo.

3. Encapsulamento

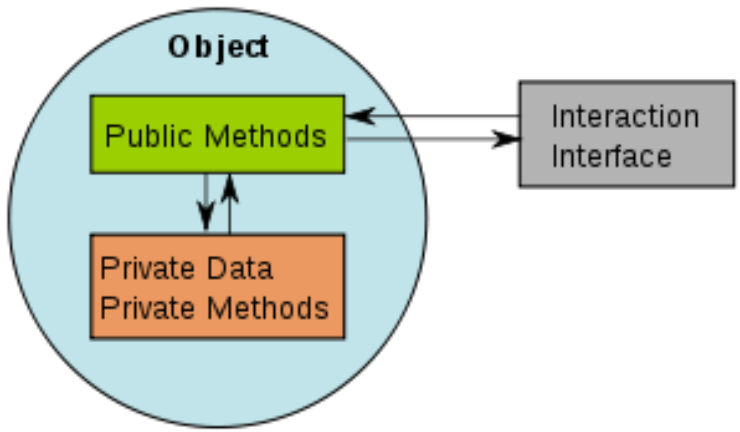
> Regra de ouro da POO: Sempre que existir uma maneira de deixar ao modelo a capacidade e responsabilidade pela modificação de um de seus dados, devemos criar uma operação para fazê-lo.

- Encapsulamento: a capacidade de ocultar dados dentro de modelos, permitindo que somente operações especializadas ou dedicadas manipulem os dados ocultos.
  - Modelos que encapsulam os dados possibilitam a criação de programas com menos erros e mais clareza. **Encapsulamento de dados em modelos deve ser um dos principais objetivos do programador que use linguagens orientadas a objetos.**

Modificadores de Acesso

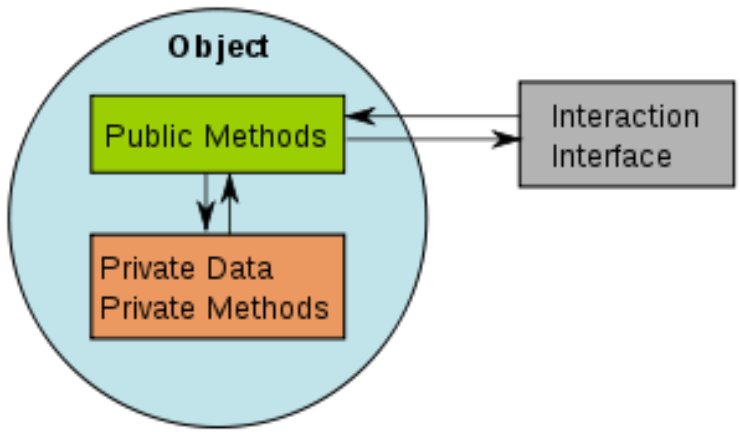
Modificadores de acesso

- Modificadores de acesso são padrões de visibilidade de acessos às classes, atributos e métodos.



Modificadores de acesso

- Modificadores de acesso são padrões de visibilidade de acessos às classes, atributos e métodos.



- Os modificadores são palavras-chaves reservadas pelo Java que são declarados antes dos nomes de classes, atributos e métodos.

Modificadores de acesso

- **public**: o atributo ou método declarado com este modificador poderá ser acessado ou executado a partir de qualquer outra entidade que possa visualizar a classe a que eles pertencem.

Modificadores de acesso

- **public**: o atributo ou método declarado com este modificador poderá ser acessado ou executado a partir de qualquer outra entidade que possa visualizar a classe a que eles pertencem.

- **private**: atributos e métodos declarados com este modificador só podem ser acessados ou executados por métodos da mesma classe, sendo completamente ocultos para o programador usuário que for usar instâncias desta classe ou criar classes herdeiras.

Modificadores de acesso

- **public**: o atributo ou método declarado com este modificador poderá ser acessado ou executado a partir de qualquer outra entidade que possa visualizar a classe a que eles pertencem.

- **private**: atributos e métodos declarados com este modificador só podem ser acessados ou executados por métodos da mesma classe, sendo completamente ocultos para o programador usuário que for usar instâncias desta classe ou criar classes herdeiras.

- **protected**: funciona como o modificador private, exceto que classes herdeiras também terão acesso ao atributo ou método marcado com este modificador.

Modificadores de acesso

- **public**: o atributo ou método declarado com este modificador poderá ser acessado ou executado a partir de qualquer outra entidade que possa visualizar a classe a que eles pertencem.

- **private**: atributos e métodos declarados com este modificador só podem ser acessados ou executados por métodos da mesma classe, sendo completamente ocultos para o programador usuário que for usar instâncias desta classe ou criar classes herdeiras.

- **protected**: funciona como o modificador private, exceto que classes herdeiras também terão acesso ao atributo ou método marcado com este modificador.

- Campos e métodos declarados sem modificadores são visíveis a todas as classes pertencentes ao mesmo pacote (package).

Regras gerais de ocultação e de acesso de atributos e métodos

1. Todos os atributos de uma classe devem ser declarados com o modificador private ou com o modificador protected.

Regras gerais de ocultação e de acesso de atributos e métodos

1. Todos os atributos de uma classe devem ser declarados com o modificador private ou com o modificador protected.

2. Métodos de uma classe que devam ser acessíveis devem ser declarados explicitamente com o modificador public.

Regras gerais de ocultação e de acesso de atributos e métodos

1. Todos os atributos de uma classe devem ser declarados com o modificador private ou com o modificador protected.

2. Métodos de uma classe que devam ser acessíveis devem ser declarados explicitamente com o modificador public.

3. Como os atributos terão o modificador private, métodos que permitam a manipulação controlada dos valores dos atributos devem ser escritos nas classes, e estes métodos devem ter o modificador public.

Regras gerais de ocultação e de acesso de atributos e métodos

1. Todos os atributos de uma classe devem ser declarados com o modificador private ou com o modificador protected.

2. Métodos de uma classe que devam ser acessíveis devem ser declarados explicitamente com o modificador public.

3. Como os atributos terão o modificador private, métodos que permitam a manipulação controlada dos valores dos atributos devem ser escritos nas classes, e estes métodos devem ter o modificador public.

4. Se for necessário ou desejável, métodos de uma classe podem ser declarados como private – esses métodos não poderão ser executados por classes escritas por programadores usuários, mas poderão ser executados por outros métodos dentro de uma mesma classe.

Exemplo — Classe Data

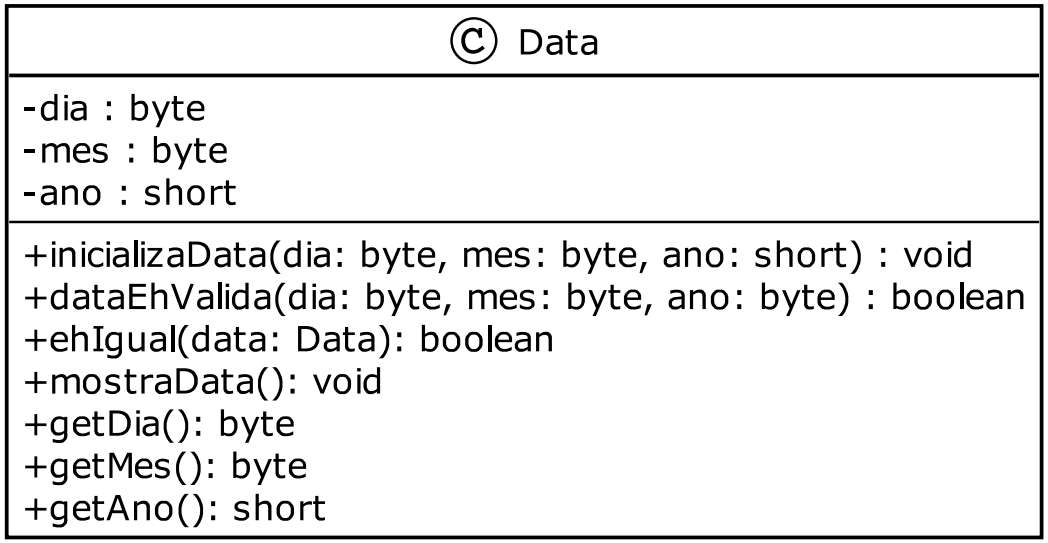


Diagrama UML representando a classe Data

Observe que a classe Data tem três métodos que permitem "ler" os atributos: getData, getMes e getAno

- Ver os arquivos Data.java e DemoData.java

Getters e Setters

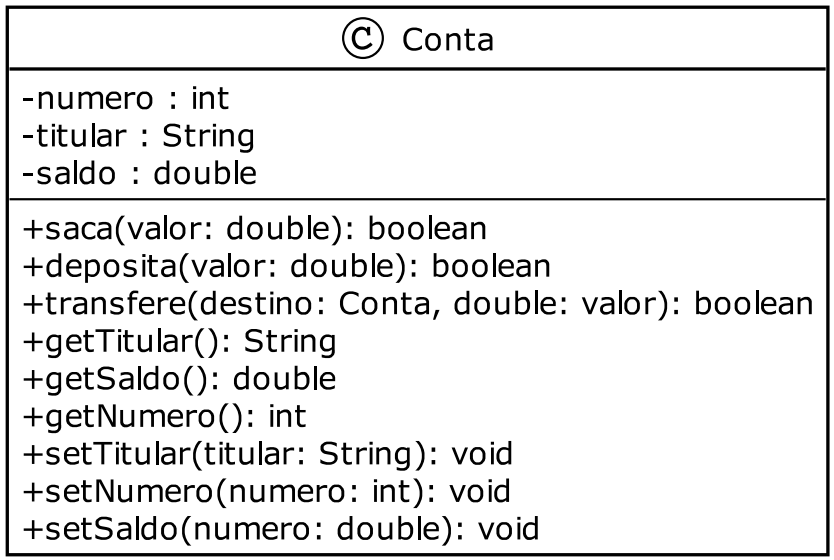
- Para permitir o acesso aos atributos (já que eles são private) de uma maneira controlada, a prática mais comum é criar dois métodos, um que retorna o valor (getter) e outro que muda o valor (setter).

- A convenção para esses métodos é de colocar a palavra `get` ou `set` como as primeiras letras do nome do método.

Criando o tipo Conta

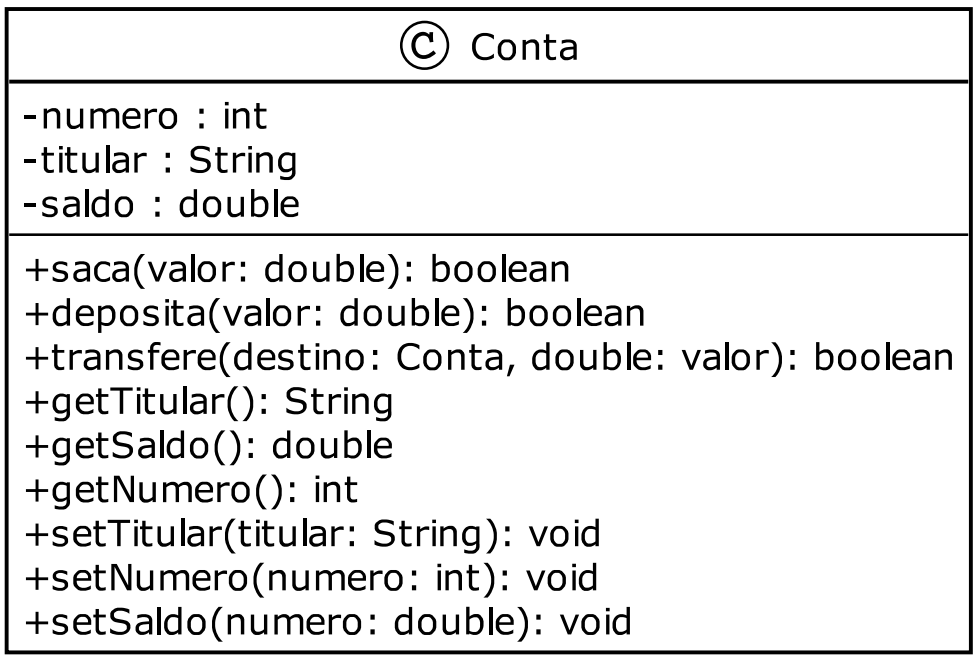
Em um sistema bancário, uma entidade importante é a Conta.

- Atributos que toda conta tem são:
  - número da conta
  - nome do titular da conta
  - saldo
- O que toda conta faz é:
  - saca uma quantidade x
  - deposita uma quantidade x
  - transfere uma quantidade x para uma outra conta y



Classe Conta

- Para sacar, é preciso ter saldo suficiente na conta.
- Os valores de saque, depósito ou transferência não podem ser negativos. Se for negativo, a operação correspondente não deve ser realizada.
- Antes de transferir de uma conta para outra, é preciso checar se há dinheiro suficiente na conta.



Classe Conta

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

```
1  class Conta {
2      private int numero;
3      private String titular;
4      private double saldo;
5
6      /**
7       * Método para sacar uma quantia da conta.
8       * @param valor o valor a ser sacado da conta
9       * @return true se o saque for bem-sucedido, false caso
       contrário
10      */
11     public boolean saca(double valor) {
12         if(valor >= 0 && valor <= this.saldo) {
13             this.saldo -= valor;
14             return true;
15         }
16         else return false;
17     }
```

Classe Conta (Cont.)


```
18      /**
19       * Método usado para depositar uma quantia na conta.
20       * Antes de depositar, verifica se o valor é não-negativo.
21       * @param valor quantia a ser depositada na conta
22       * @return true se o depósito for bem-sucedido, false caso
         contrário
23       */
24      public boolean deposita(double valor) {
25          if(valor >= 0) {
26              this.saldo += valor;
27              return true;
28          }
29          else return false;
30      }
```

Classe Conta (Cont.)


```
31      /**
32       * Método que transfere uma quantia desta conta para outra
33       * Antes de transferir, verifica se a conta possui a
34       * quantidade a ser transferida disponível.
35       * @param destino conta para a qual a quantia deve ser
        transferida
36       * @param valor quantia a ser transferida
37       * @return true se a transferência for bem sucedida, false
         caso contrário
38       */
39      public boolean transfere(Conta destino, double valor) {
40          boolean retirou = this.saca(valor);
41          if(retirou == false) {
42              return false; // não deu para sacar
43          }
44          else {
45              destino.deposita(valor);
46              return true;
47          }
48      }
```

Classe Conta (Cont.)


```
49      public int getNumero() { // getter
50          return numero;
51      }
52
53      public void setNumero(int numero) { // setter
54          this.numero = numero;
55      }
56
57      public String getTitular() { // getter
58          return titular;
59      }
60
61      public void setTitular(String titular) { // setter
62          this.titular = titular;
63      }
64
65      public double getSaldo() { // getter
66          return saldo;
67      }
68
69      public void setSaldo(double saldo) { // setter
70          this.saldo = saldo;
71      }
72  }
```

Problemas com a Classe Conta

- Nesse exemplo de classe, o método setSaldo não deveria ter sido criado, já que o saldo de uma conta só pode ser modificado por meio de saques ou depósitos.

Problemas com a Classe Conta

- Nesse exemplo de classe, o método setSaldo não deveria ter sido criado, já que o saldo de uma conta só pode ser modificado por meio de saques ou depósitos.

- O método setNumero também não deveria ter sido criado. O número da conta deveria ser gerado assim que a instância da conta é criada e nunca mais deveria mudar.

Problemas com a Classe Conta

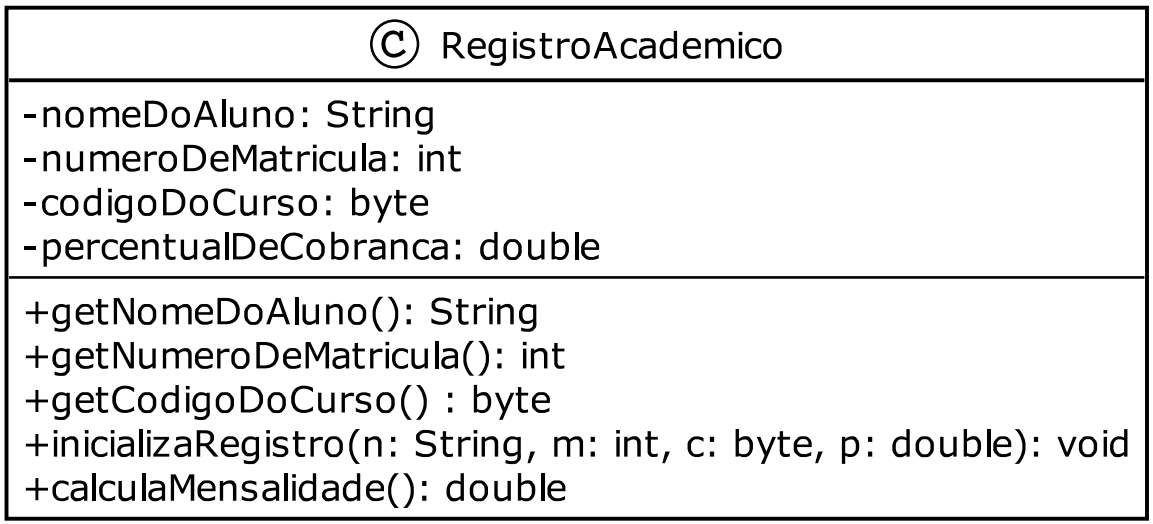
- Nesse exemplo de classe, o método setSaldo não deveria ter sido criado, já que o saldo de uma conta só pode ser modificado por meio de saques ou depósitos.

- O método setNumero também não deveria ter sido criado. O número da conta deveria ser gerado assim que a instância da conta é criada e nunca mais deveria mudar.

Má prática: criar uma classe e, logo em seguida, **criar getters e setters para todos os seus atributos**. Um getter ou setter só deve ser criado se houver a real necessidade.

Construtores

Exemplo de classe problemática



- Mostrar os arquivos RegistroAcademico.java e DemoRegistroAcademico.java

Construtores

Construtores são métodos especiais, que são chamados automaticamente quando instâncias são criadas através da palavra-chave new.

Construtores

Construtores são métodos especiais, que são chamados automaticamente quando instâncias são criadas através da palavra-chave new.

- Por meio da criação de construtores, podemos garantir que o código que eles contêm será executado antes de qualquer outro código em outros métodos, já que uma instância de uma classe só pode ser usada depois de ter sido criada com new, o que causará a execução do construtor.

Construtores

Construtores são métodos especiais, que são chamados automaticamente quando instâncias são criadas através da palavra-chave new.

- Por meio da criação de construtores, podemos garantir que o código que eles contêm será executado antes de qualquer outro código em outros métodos, já que uma instância de uma classe só pode ser usada depois de ter sido criada com new, o que causará a execução do construtor.

- Construtores são particularmente úteis para iniciar atributos de instâncias de classes para garantir que, quando métodos dessas instâncias forem chamados, eles contenham valores específicos e não os default.

Construtores

Diferenças entre construtores e outros métodos

- Construtores devem ter **exatamente** o mesmo nome que a classe a que pertencem.

Construtores

Diferenças entre construtores e outros métodos

- Construtores devem ter **exatamente** o mesmo nome que a classe a que pertencem.

- Construtores não podem retornar nenhum valor, nem mesmo void, portanto devem ser declarados sem tipo de retorno.

Construtores

Diferenças entre construtores e outros métodos

- Construtores devem ter **exatamente** o mesmo nome que a classe a que pertencem.
- Construtores não podem retornar nenhum valor, nem mesmo void, portanto devem ser declarados sem tipo de retorno.
- Não é possível invocar construtores diretamente – construtores só são executados quando a instância é criada com new.

Construtores

Diferenças entre construtores e outros métodos

- Construtores devem ter **exatamente** o mesmo nome que a classe a que pertencem.
- Construtores não podem retornar nenhum valor, nem mesmo void, portanto devem ser declarados sem tipo de retorno.
- Não é possível invocar construtores diretamente – construtores só são executados quando a instância é criada com new.
- Construtores podem receber modificadores como public ou private.

Construtores – Exemplo

- Analisar os arquivos Conta2.java e DemoConta2.java

Construtores – Exemplo

- Analisar os arquivos Conta2.java e DemoConta2.java

- Observe que, na classe Conta2, foram criados dois construtores, um com dois parâmetros e outro com três parâmetros. A isso, damos o nome de **sobrecarga de construtores**.

Construtores – Exemplo

- Analisar os arquivos Conta2.java e DemoConta2.java

- Observe que, na classe Conta2, foram criados dois construtores, um com dois parâmetros e outro com três parâmetros. A isso, damos o nome de **sobrecarga de construtores**.

- **Obs. 1:** Quando não declaramos um construtor para a classe, o Java cria um para ela, chamado **construtor default**. Esse construtor não recebe argumentos e seu corpo é vazio.

- **Obs. 2:** Quando o programador de classes cria um ou mais construtores, o compilador não inclui o construtor default.

Sobrecarga de métodos

Assinaturas de métodos

- Java permite a criação de métodos com nomes iguais, contanto que as suas assinaturas sejam diferentes.

- A assinatura de um método é composta de seu nome mais os tipos de argumentos que são passados para esse método, independentemente dos nomes de variáveis usadas na declaração do método.

Assinaturas de métodos

- Java permite a criação de métodos com nomes iguais, contanto que as suas assinaturas sejam diferentes.

- A assinatura de um método é composta de seu nome mais os tipos de argumentos que são passados para esse método, independentemente dos nomes de variáveis usadas na declaração do método.

- Por exemplo, a assinatura do método
  `public boolean transfere(Conta destino, double valor) {...}`

  é

  `transfere(Conta, double)`

- O tipo de retorno do método **não** é considerado parte da assinatura.

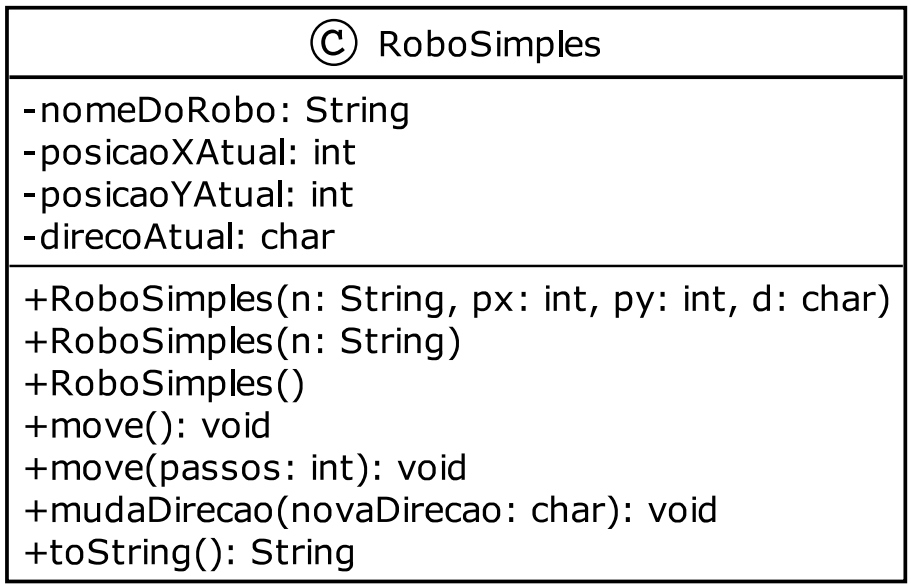
Sobrecarga de métodos

- A possibilidade de criar mais de um método com o mesmo nome e assinaturas diferentes é conhecida como sobrecarga de métodos.

- A decisão sobre qual método será chamado quando existem dois ou mais métodos com o mesmo nome será feita pelo compilador, com base na assinatura dos métodos.

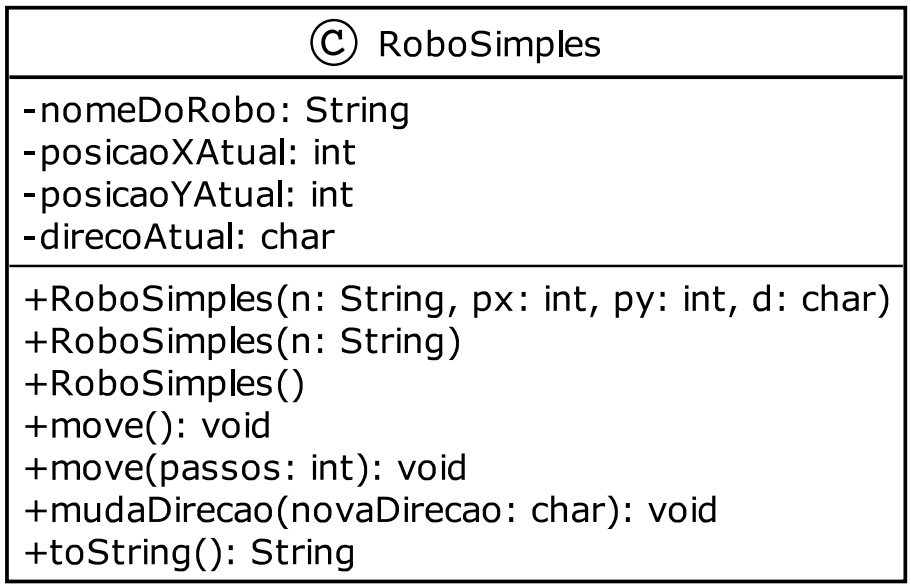
Exemplo

Queremos modelar um robô que tenha uma posição qualquer no espaço de duas dimensões. O robô pode andar em 4 direções (norte, sul, leste e oeste) e pode se movimentar somente para a frente em qualquer uma dessas direções.



Exemplo

Queremos modelar um robô que tenha uma posição qualquer no espaço de duas dimensões. O robô pode andar em 4 direções (norte, sul, leste e oeste) e pode se movimentar somente para a frente em qualquer uma dessas direções.



Exemplo: Analisar os arquivos RoboSimples0.java e DemoRoboSimples0.java

O método `toString()`

- **Toda** classe escrita em Java possui um método chamado `toString`. Esse método não recebe argumentos e retorna uma String.

- O método `toString` sempre é chamado de forma automática quando um objeto é passado para os métodos println, print, printf ou para o operador de concatenação de Strings.

O método `toString()`

- **Toda** classe escrita em Java possui um método chamado `toString`. Esse método não recebe argumentos e retorna uma String.

- O método `toString` sempre é chamado de forma automática quando um objeto é passado para os métodos println, print, printf ou para o operador de concatenação de Strings.

- Por default, a string retornada pelo método toString é composta pelo nome da classe seguido por um arroba (@) e pela representação do código de hash em hexadecimal sem sinal como, por exemplo, `Conta2@76ccd017`

  - Algo semelhante é impresso ao final da execução de DemoConta2.java

O método `toString()`

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

- **Toda** classe escrita em Java possui um método chamado `toString`. Esse método não recebe argumentos e retorna uma String.

- O método `toString` sempre é chamado de forma automática quando um objeto é passado para os métodos println, print, printf ou para o operador de concatenação de Strings.

- Por default, a string retornada pelo método toString é composta pelo nome da classe seguido por um arroba (@) e pela representação do código de hash em hexadecimal sem sinal como, por exemplo, `Conta2@76ccd017`

  - Algo semelhante é impresso ao final da execução de DemoConta2.java

- Porém, podemos sobrepor esse método, como foi feito na classe RoboSimples0.java

A palavra-chave `this`

Um problema com a classe RoboSimples0

- Parte do código é repetida entre os construtores da classe RoboSimples0 e entre os métodos sobrecarregados.
  - Isso fere o pilar do reúso. Existe redundância de código.

```
1      public void move() {
2          if(direcaoAtual == 'N') posicaoYAtual++;
3          if(direcaoAtual == 'S') posicaoYAtual--;
4          if(direcaoAtual == 'L') posicaoXAtual++;
5          if(direcaoAtual == 'O') posicaoXAtual--;
6      }
```

```
1      public void move(int passos) {
2          if(direcaoAtual == 'N') posicaoYAtual += passos;
3          if(direcaoAtual == 'S') posicaoYAtual -= passos;
4          if(direcaoAtual == 'L') posicaoXAtual += passos;
5          if(direcaoAtual == 'O') posicaoXAtual -= passos;
6      }
```

Um problema com a classe RoboSimples0

- Como o código é repetido, podemos esperar que a sua manutenção seja trabalhosa.

Um problema com a classe RoboSimples0

- Como o código é repetido, podemos esperar que a sua manutenção seja trabalhosa.

- Ideia: A redundância entre métodos pode ser solucionada se pudermos executar uns métodos a partir de outros.
  - Por exemplo, o método `move()`, que move o robô sem receber argumentos é funcionalmente igual ao método, `move(int)`, que recebe argumentos, se passarmos o valor 1 como argumento para este último.

Um problema com a classe RoboSimples0

Reescrita da função `move()`

```
1       public void move() {
2           move(1);
3       }
```

```
1       public void move(int passos) {
2           if(direcaoAtual == 'N') posicaoYAtual += passos;
3           if(direcaoAtual == 'S') posicaoYAtual -= passos;
4           if(direcaoAtual == 'E') posicaoXAtual += passos;
5           if(direcaoAtual == 'O') posicaoXAtual -= passos;
6       }
```

Um problema com a classe RoboSimples0

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

O caso dos contrutores

```
1      RoboSimples0(String n, int px, int py, char d) {
2          nomeDoRobo = n;
3          posicaoXAtual = px;
4          posicaoYAtual = py;
5          direcaoAtual = d;
6      }
```

```
1      RoboSimples0(String n) {
2          nomeDoRobo = n;
3          posicaoXAtual = 0;
4          posicaoYAtual = 0;
5          direcaoAtual = 'N';
6      }
```

```
1      RoboSimples0() {
2          nomeDoRobo = "";
3          posicaoXAtual = 0;
4          posicaoYAtual = 0;
5          direcaoAtual = 'N';
6      }
```

Um problema com a classe RoboSimples0

O caso dos contrutores

- Java cria, internamente para cada instância, uma "auto-referência". Essa referência é representada pela palavra-chave this.

- Para chamar um construtor dentro de outro, basta usar a palavra-chave this substituindo o nome do construtor.

Nova classe RoboSimples

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

```
1      RoboSimples(String n, int px, int py, char d) {
2          nomeDoRobo = n;
3          posicaoXAtual = px;
4          posicaoYAtual = py;
5          direcaoAtual = d;
6      }
```

```
1      RoboSimples(String n) {
2          // chama o construtor completo passando a
3          // posição e direção como constantes
4          this(n,0,0,'N');
5      }
```

```
1      RoboSimples() {
2          // chama o construtor completo passando
3          // o nome como uma string vazia e a
4          // posição e direção como constantes
5          this("",0,0,'N');
6      }
```

Regras para chamada de construtores com o `this`

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

(1) Construtores não são chamados pelos seus nomes, e sim por this.

Regras para chamada de construtores com o `this`

(1) Construtores não são chamados pelos seus nomes, e sim por this.

(2) Somente construtores podem chamar construtores como subrotinas.

Regras para chamada de construtores com o `this`

(1) Construtores não são chamados pelos seus nomes, e sim por this.

(2) Somente construtores podem chamar construtores como subrotinas.

(3) Se um construtor for chamado dentro de outro, a chamada deve ser a primeira linha de código dentro do corpo do construtor.

Regras para chamada de construtores com o `this`

(1) Construtores não são chamados pelos seus nomes, e sim por this.

(2) Somente construtores podem chamar construtores como subrotinas.

(3) Se um construtor for chamado dentro de outro, a chamada deve ser a primeira linha de código dentro do corpo do construtor.

(4) Construtores podem chamar outros métodos.

- Exemplo: métodos inicializadores.

Regras para chamada de construtores com o `this`

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

(1) Construtores não são chamados pelos seus nomes, e sim por this.

(2) Somente construtores podem chamar construtores como subrotinas.

(3) Se um construtor for chamado dentro de outro, a chamada deve ser a primeira linha de código dentro do corpo do construtor.

(4) Construtores podem chamar outros métodos.

- Exemplo: métodos inicializadores.

(5) Métodos não podem chamar construtores, nem mesmo com this.

Regras para chamada de construtores com o `this`

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

(1) Construtores não são chamados pelos seus nomes, e sim por this.

(2) Somente construtores podem chamar construtores como subrotinas.

(3) Se um construtor for chamado dentro de outro, a chamada deve ser a primeira linha de código dentro do corpo do construtor.

(4) Construtores podem chamar outros métodos.

- Exemplo: métodos inicializadores.

(5) Métodos não podem chamar construtores, nem mesmo com this.

(6) Construtores não podem ser chamados recursivamente: um construtor só pode chamar diretamente outro construtor, e não a si próprio.

Destrutores

Destrutores

- Algumas linguagens orientadas a objetos (como o C++) possuem métodos que são chamados automaticamente quando um objeto é destruído. Esses métodos são chamados destrutores.
  - Em C++, objetos alocados dinamicamente pelo programador com a palavra chave new devem ser liberados explicitamente usando a palavra chave delete.
  - Em C++, quem cuida da liberação da memória alocada dinâmicamente é o próprio programador.

Destrutores

- Algumas linguagens orientadas a objetos (como o C++) possuem métodos que são chamados automaticamente quando um objeto é destruído. Esses métodos são chamados destrutores.
  - Em C++, objetos alocados dinamicamente pelo programador com a palavra chave new devem ser liberados explicitamente usando a palavra chave delete.
  - Em C++, quem cuida da liberação da memória alocada dinâmicamente é o próprio programador.

```
1 class Array { // Uma classe em C++
2     Array(int n) { // Construtor
3         int *vec = new int[n];
4     }
5
6     ~Array() { // Destrutor
7         delete[] vec;
8     }
9 }
```

Destrutores em Java

- Java usa o garbage collector (coletor de lixo), um programa executado na JVM que remove objetos que não estão mais sendo usados por uma aplicação Java.

Destrutores em Java

- Java usa o garbage collector (coletor de lixo), um programa executado na JVM que remove objetos que não estão mais sendo usados por uma aplicação Java.

- Java não tem o conceito que C++ tem de um destrutor. Em Java, a ideia é simplesmente esquecer dos objetos, no lugar de destruí-los, permitindo que o coletor de lixo recupere a memória conforme necessário

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

FIM