Centro Tecnológico
Departamento de Informática

Prof. Vítor E. Silva Souza
http://www.inf.ufes.br/~vitorsouza

[Desenvolvimento OO com Java]

Modificadores de acesso e atributos de classe

UNIVERSIDADE FEDERAL
DO ESPÍRITO SANTO

Conteúdo do curso

- O que é Java;
- Variáveis primitivas e controle de fluxo;
- Orientação a objetos básica;
- Um pouco de vetores;
- Modificadores de acesso e atributos de classe;
- Herança, reescrita e polimorfismo;
- Classes abstratas e interfaces;
- Exceções e controle de erros;
- Organizando suas classes;
- Utilitários da API Java.

Estes slides foram baseados na apostila do curso FJ-11: Java e Orientação a Objetos da Caelum e na apostila Programação Orientada a Objetos em Java do prof. Flávio Miguel Varejão.

Questão de responsabilidade (de novo!)

- Imagine um sistema...
  - *Dezenas de formulários;*
  - *CPFs são validados – função `validar(cpf)` deve ser chamada em cada formulário;*
  - *Todos os desenvolvedores são responsáveis!*
- Sem problemas! Define-se a classe Pessoa com o atributo cpf e o método `validar()`!
  - *E aí o desenvolvedor foi lá e...*

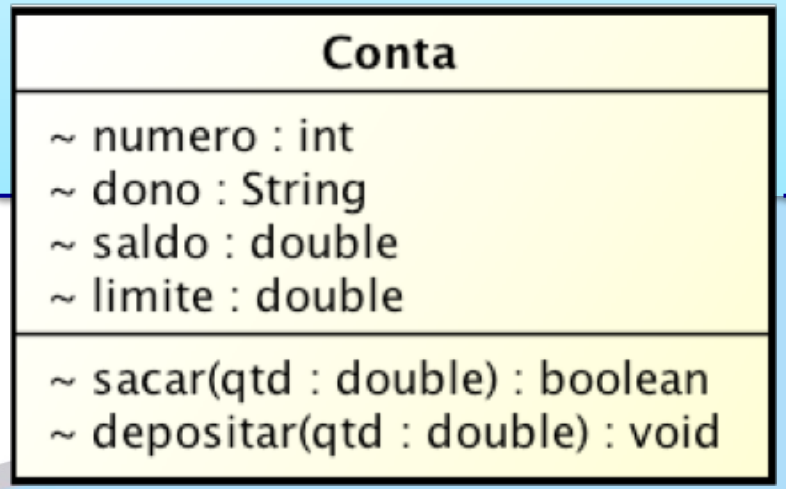
```
Pessoa p = new Pessoa();
p.cpf = "321.654.987-00"; // Cadê a validação?
```

Regras de negócio

- Para uma conta corrente:

```
class Conta {
  int numero;
  String dono;
  double saldo;
  double limite;
  // ...
}
```

RN1: quando negativo, o valor absoluto do saldo não pode ser superior ao do limite.



UNIVERSIDADE FEDERAL
DO ESPÍRITO SANTO

Implementando a regra de negócio

```
class Conta {
  // Restante da classe...
  void sacar(double qtd) {
    double novoSaldo = this.saldo - qtd;
    this.saldo = novoSaldo;
  }
}
public class TesteConta {
  public static void main(String[] args) {
    Conta c = new Conta();
    c.saldo = 1000.0;
    c.limite = 1000.0;
    c.sacar(5000);  // Vai gerar inconsistência!
  }
}
```

Implementando a regra de negócio

```
public class TesteConta {
  public static void main(String[] args) {
    Conta c = new Conta();
    c.saldo = 1000.0;
    c.limite = 1000.0;

    // Vamos verificar antes de sacar...
    double valorASacar = 5000.0;
    if (valorASacar < c.saldo + c.limite)
      c.sacar(valorASacar);
  }
}
```

A responsabilidade está com a classe certa?

UNIVERSIDADE FEDERAL
DO ESPÍRITO SANTO

Implementando a regra de negócio

Conta

```
boolean sacar(double qtd) {
   double novoSaldo = this.saldo - qtd;
   if (novoSaldo >= -limite) {
      this.saldo = novoSaldo;
      return true;
   }
   else return false;
}
```

```
   Conta c = new Conta();
   c.saldo = 1000.0;
   c.limite = 1000.0;

   // Agora sim!
   if (c.sacar(5000)) System.out.println("Consegui");
   else System.out.println("Não deu...");

   c.saldo = -3000.0;  // Só que não...
```

UNIVERSIDADE FEDERAL
DO ESPÍRITO SANTO

Falta encapsulamento...

- Permitir o acesso direto aos atributos:
  - *Exige disciplina dos clientes da classe `Conta`;*
  - *Pode levar a inconsistências;*
- Solução: impedir o acesso externo ao atributo:
  - *Atributo privativo;*
  - *Externo = qualquer outra clase, exceto a proprietária do atributo (ex.: `Conta` para o atributo `saldo`).*
- Vantagens:
  - *Objetos trocam mensagens com base em contratos;*
  - *Modificações na implementação não afetam clientes (ex.: adicionar CPMF nos saques de conta-corrente).*

UNIVERSIDADE FEDERAL
DO ESPÍRITO SANTO

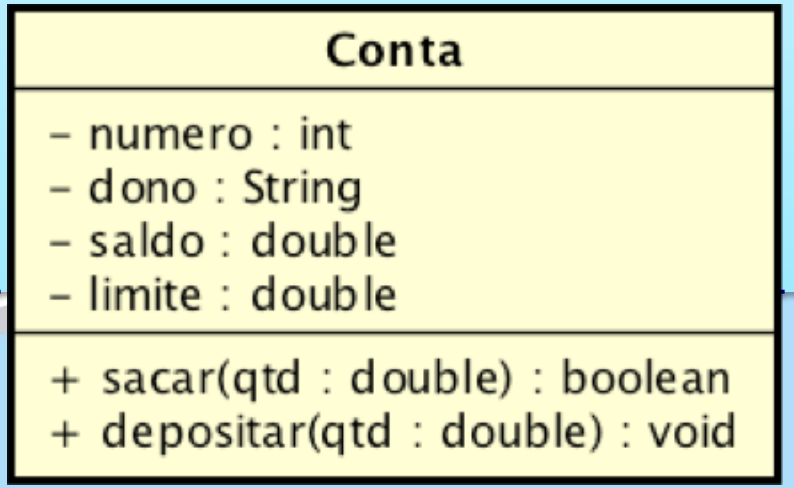
Encapsulamento

- Usamos objetos sem saber seu funcionamento interno;
- Assim também deve ser em nossos sistemas OO:
  - *Maior manutenibilidade;*
  - *Maior reusabilidade.*

Implementando o encapsulamento

```
class Conta {
  private int numero;
  private String dono;
  private double saldo;
  private double limite;

  public boolean sacar(double qtd) {
     // ...
  }

  // ...
}
```

Modificador de acesso / visibilidade!



UNIVERSIDADE FEDERAL
DO ESPÍRITO SANTO

Modificadores de acesso

- Determinam a visibilidade de um determinado membro da classe com relação a outras classes;
- Há quatro níveis de acesso:
  - *Público (public);*
  - *Privado/privativo (private);*
  - *Protegido (protected);*
  - *Amigo ou privativo ao pacote (friendly ou package-private).*

Regra de bolso do encapsulamento: atributos são privativos, métodos são públicos.

Regra de bolso das regras: toda regra tem suas exceções...

UNIVERSIDADE FEDERAL
DO ESPÍRITO SANTO

Palavras-chave

- Três palavras-chave especificam o acesso:
  - *public*
  - *private*
  - *protected*
- O nível de acesso *package-private* é determinado pela ausência de especificador;
- Devem ser usadas antes do nome do membro que querem especificar;
- Não podem ser usadas em conjunto.

UNIVERSIDADE FEDERAL
DO ESPÍRITO SANTO

Modificadores de acesso

Acesso	Público	Protegido	Amigo	Privado
A própria classe	Sim	Sim	Sim	Sim
Classe no mesmo pacote	Sim	Sim	Sim	Não
Subclasse em pacote diferente	Sim	Sim	Não	Não
Não-subclasse em pacote diferente	Sim	Não	Não	Não

Testando o encapsulamento

Conta

```
class Conta {
  // ...
  private double saldo;

  public boolean sacar(double qtd) {
     // ...
  }
}
```

TesteConta

```
    Conta c = new Conta();
    c.depositar(1000.0);
    c.saldo = -3000.0;

    // Não compila!
    // error: saldo has private access in Conta
    // c.saldo = -3000.0;
    //  ^
```

UNIVERSIDADE FEDERAL
DO ESPÍRITO SANTO

Separando interface e implementação

- Em OO é fundamental o ocultamento de informação:
  - *Estrutura interna fica inacessível;*
  - *Interface do objeto é pública.*

- O que é uma pilha?
  - *Uma lista?*
  - *Um vetor?*
  - *Uma estrutura que me permite empilhar e desempilhar itens?*

UNIVERSIDADE FEDERAL
DO ESPÍRITO SANTO

Ocultamento de informações

```
import java.util.*;

class Pilha {
  private Vector elems = new Vector(10, 10);

  public void empilha(Object obj) {
    elems.add(obj);
  }

  public Object desempilha() {
    Object obj = elems.get(elems.size() - 1);
    elems.remove(elems.size() - 1);
    return obj;
  }
}
```

UNIVERSIDADE FEDERAL
DO ESPÍRITO SANTO

Mudando a implementação

```
import java.util.*;

class Pilha {
  private LinkedList elems = new LinkedList();

  public void empilha(Object obj) {
    elems.addFirst(obj);
  }

  public Object desempilha() {
    return elems.removeFirst();
  }
}
```

Programe para interfaces, não para implementações!
(Design Patterns, de Eric Gamma et al.)

UNIVERSIDADE FEDERAL
DO ESPÍRITO SANTO

O exemplo do CPF

```
class Cliente {
  private String nome;
  private String endereco;
  private String cpf;
  private int idade;

  public void mudaCPF(String cpf) {
     validaCPF(cpf);
     this.cpf = cpf;
}

  private void validaCPF(String cpf) {
     // série de regras aqui, falha caso não seja válido
  }

  // ...
}
```

E se um dia eu não precisar mais validar CPF para pessoas com idade acima de 60 anos?

UNIVERSIDADE FEDERAL
DO ESPÍRITO SANTO

Mas e se eu precisar acessar um atributo?

- Basta usar um método para isso!

```
class Conta {
  private int numero;
  private String dono;
  private double saldo;
  private double limite;

  public double verSaldo() {
    return saldo;
  }
  public void alterarLimite(double limite) {
    this.limite = limite;
  }
  // ...
}
```

UNIVERSIDADE FEDERAL
DO ESPÍRITO SANTO

POJOs (o Bom e Velho Objeto Java)

- Atributos devem ser privativos;
- Se precisarem ser lidos ou alterados, prover métodos get/set (para booleanos, pode-se usar o prefixo `is`):

```
public class Cliente {
  private String nome;          // ...

  public String getNome() {
    return  nome;
  }

  public void setNome(String nome) {
    this.nome = nome;
  }
}
```

IDEs geram esses métodos automaticamente.

Planeje bem a interface da classe

- É importante observar que:
  - *O método `getAtr()` não tem que necessariamente retornar apenas o atributo `atr`;*
  - *Não crie automaticamente métodos get/set para todos os atributos! Para alguns não faz sentido...*

```
// O limite faz parte do saldo (só que cobra juros)!
public double getSaldo() {
   return saldo + limite;
}

// Não há método para mudar o saldo. Tem que sacar()
// public void setSaldo(double saldo)
```

Sugestão de leitura:
http://blog.caelum.com.br/nao-aprender-oo-getters-e-setters/

UNIVERSIDADE FEDERAL
DO ESPÍRITO SANTO

Inicialização

- Neologismo criado para indicar tarefas que devem ser efetuadas ao iniciarmos algo;
- Quando criamos objetos, podemos querer inicializá-lo com alguns valores;
- Poderíamos criar um método para isso:

```
class Aleatorio {
  private int numero;

  public void inicializar() {
    Random rand = new Random();
    numero = rand.nextInt(20);
  }
}
```

UNIVERSIDADE FEDERAL
DO ESPÍRITO SANTO

Construtores

- Problema do método `inicializar()`: podemos esquecer de chamá-lo!
- Por isso, Java provê o mecanismo de construtores:
  - *São chamados automaticamente pelo Java quando um objeto novo é criado;*
  - *Construtores não tem valor de retorno e possuem o mesmo nome da classe.*

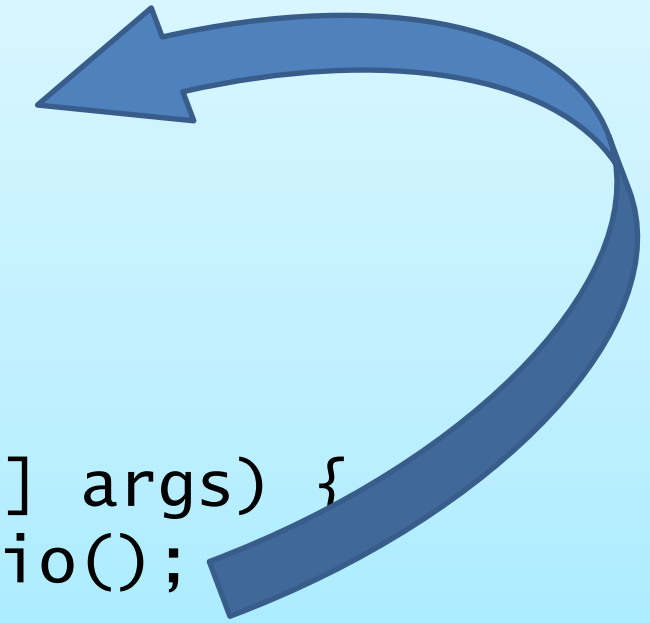
new → construtor

UNIVERSIDADE FEDERAL
DO ESPÍRITO SANTO

Construtores

- Quando um novo objeto é criado:
  1. *é alocada memória para o objeto;*
  2. *o construtor é chamado.*

```
class Aleatorio {
  private int numero;
  public Aleatorio() {
    Random rand = new Random();
    numero = rand.nextInt(20);
  }
}
public class Teste {
  public static void main(String[] args) {
    Aleatorio aleat = new Aleatorio();
  }
}
```



UNIVERSIDADE FEDERAL
DO ESPÍRITO SANTO

Construtores podem ter argumentos

- Se definidos argumentos, devem ser passados na criação do objeto com new:

```
class Aleatorio {
  private int numero;
  public Aleatorio(int max) {
    Random rand = new Random();
    numero = rand.nextInt(max);
  }
}
public class Teste {
  public static void main(String[] args) {
    Aleatorio aleat1 = new Aleatorio(20);
    Aleatorio aleat2 = new Aleatorio(50);
  }
}
```

UNIVERSIDADE FEDERAL
DO ESPÍRITO SANTO

Pode haver múltiplos construtores

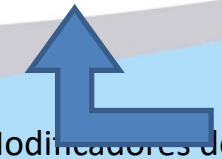
- Nossas classes podem ter quantos construtores quisermos (com argumentos diferentes):

```
class Aleatorio {
  private int numero;
  public Aleatorio() {
     Random rand = new Random();
     numero = rand.nextInt(20);
  }
  public Aleatorio(int max) {
     Random rand = new Random();
     numero = rand.nextInt(max);
  }
}
public class Teste {
  public static void main(String[] args) {
     Aleatorio aleat1 = new Aleatorio();
     Aleatorio aleat2 = new Aleatorio(50);
```

UNIVERSIDADE FEDERAL
DO ESPÍRITO SANTO

Construtor *default*

- Quando não especificamos construtores, Java provê um construtor *default* para nossa classe:
  - *Toda classe precisa de um construtor;*
  - *Sem parâmetros e sem implementação.*
- Quando especificamos construtores, o construtor *default* não é provido automaticamente:
  - *Se você escreveu um construtor, Java assume que você sabe o que está fazendo e não provê um;*
  - *Chamar o construtor sem o parâmetro gera erro se ele não for definido explicitamente.*



Isso é bom! Mas por que? (Apostila Caelum, sec. 6.5)

Construtores chamando construtores

- Seria interessante não haver duplicação de código:

```
class Aleatorio {
  private int numero;

  public Aleatorio() {
    Random rand = new Random();
    numero = rand.nextInt(20);
  }

  public Aleatorio(int max) {
    Random rand = new Random();
    numero = rand.nextInt(max);
  }
}
```

UNIVERSIDADE FEDERAL
DO ESPÍRITO SANTO

Construtores chamando construtores

- Usamos novamente a palavra-chave `this`, com outro significado: chamar outro construtor:

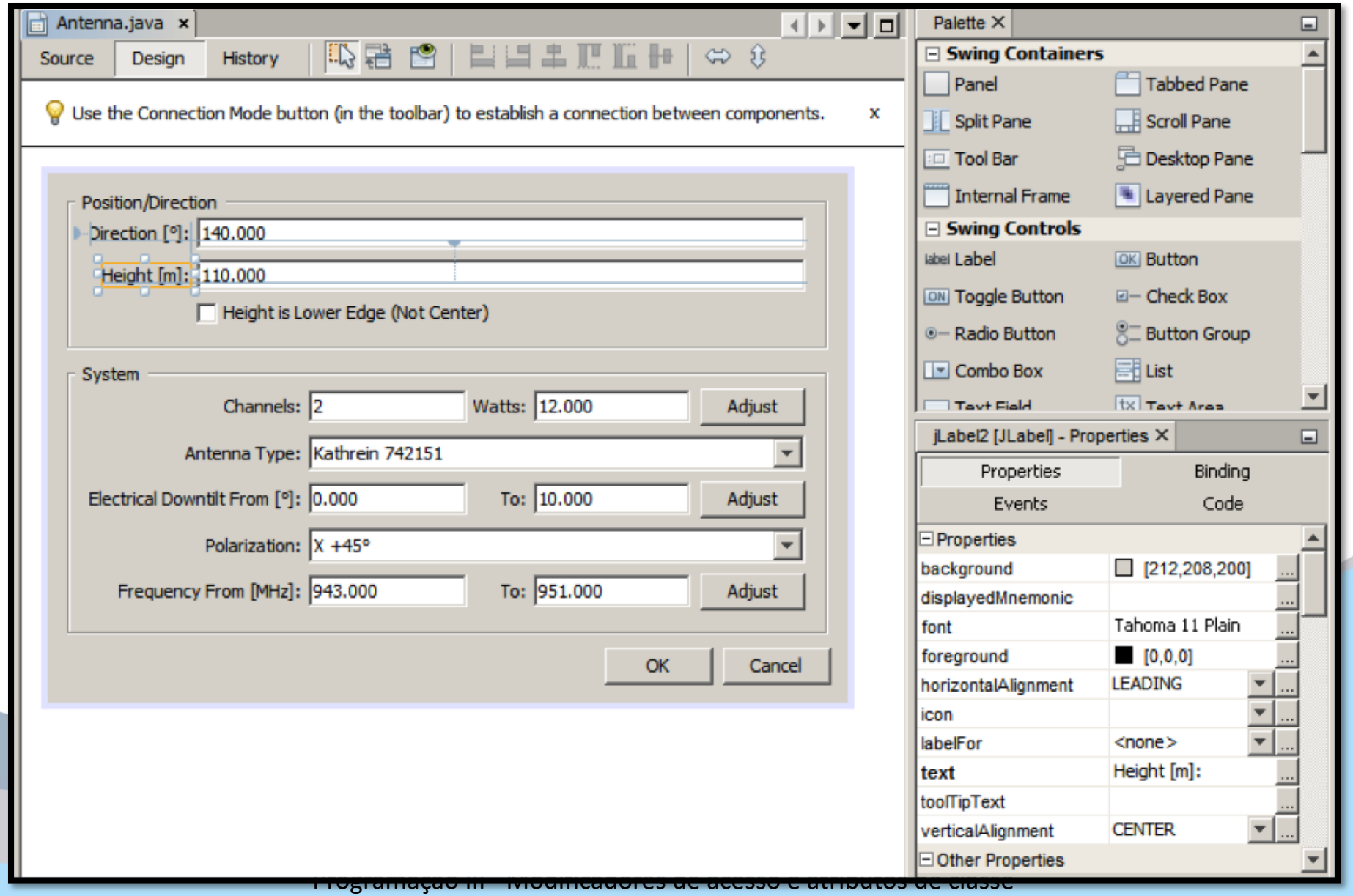
```
class Aleatorio {
  private int numero;

  public Aleatorio() {
    // Chama o outro construtor com argumento 20.
    this(20);
  }

  public Aleatorio(int max) {
    Random rand = new Random();
    numero = rand.nextInt(max);
  }
}
```

Construtores chamando construtores

- A chamada `this()`:
  - *Deve especificar os argumentos do construtor a ser chamado;*
  - *Deve ser a primeira linha do construtor que a utiliza;*
  - *Não pode ser usada fora de construtores.*

Depois que o objeto foi construído, não é mais possível chamar o construtor para ele.

O padrão JavaBean



Atributos de Classe

UNIVERSIDADE FEDERAL
DO ESPÍRITO SANTO

Atributos independentes de objetos

- Vimos até agora que atributos pertencem aos objetos:
  - *Não se faz nada sem antes criar um objeto (new)!*
- No entanto, há situações que você quer usá-los sem ter que criar objetos:

```
public class TesteConta {
  public static void main(String[] args) {
    int qtdContas = 0;
    Conta c1 = new Conta();
    qtdContas++;

    Conta c2 = new Conta();
    qtdContas++;
    // ...
  }
}
```

A responsabilidade está com a classe certa?

UNIVERSIDADE FEDERAL
DO ESPÍRITO SANTO

Atributos independentes de objetos

- Se acertamos a responsabilidade, voltamos a depender de um objeto para usar o atributo:

```
class Conta {
  // ...
  public int qtdContas = 0;
  public Conta() {
    qtdContas++;    // Outras inicializações...
  }
}
```

```
    Conta c1 = new Conta();
    Conta c2 = new Conta();

    // Quantas contas foram criadas?
    System.out.println(c2.qtdContas);
```

TesteConta

UNIVERSIDADE FEDERAL
DO ESPÍRITO SANTO

Atributos `static`

- Usando a palavra-chave `static` você define um atributo de classe ("estático"):
  - *Pertence à classe como um todo;*
  - *Pode-se acessá-los mesmo sem ter criado um objeto;*
  - *Objetos podem acessá-los como se fosse um membro de objeto, só que compartilhado.*

```
class Conta {
  // ...
  public static int qtdContas = 0;

  public Conta() {
    qtdContas++;    // Outras inicializações...
  }
}
```

UNIVERSIDADE FEDERAL
DO ESPÍRITO SANTO

Acesso a atributos `static`

- Não precisamos mais de um objeto pra acessar:

```
public class TesteConta {
  public static void main(String[] args) {
    Conta c1 = new Conta();
    Conta c2 = new Conta();

    System.out.println(Conta.qtdContas);

    // ...
  }
}
```

O atributo é da classe.

A visibilidade está certa?

UNIVERSIDADE FEDERAL
DO ESPÍRITO SANTO

Acesso a atributos `static`

- Se acertamos a visibilidade do atributo, precisamos então de um método para acessá-lo:

```
class Conta {
  // ...
  private static int qtdContas = 0;

  public Conta() {
    qtdContas++;    // Outras inicializações...
  }

  public int getQtdContas() {
    return qtdContas;
  }
}
```

Mas não sendo `static`, vou precisar de um objeto pra acessar o atributo de novo!?!

UNIVERSIDADE FEDERAL
DO ESPÍRITO SANTO

Métodos `static`

- Métodos também podem ser `static`:

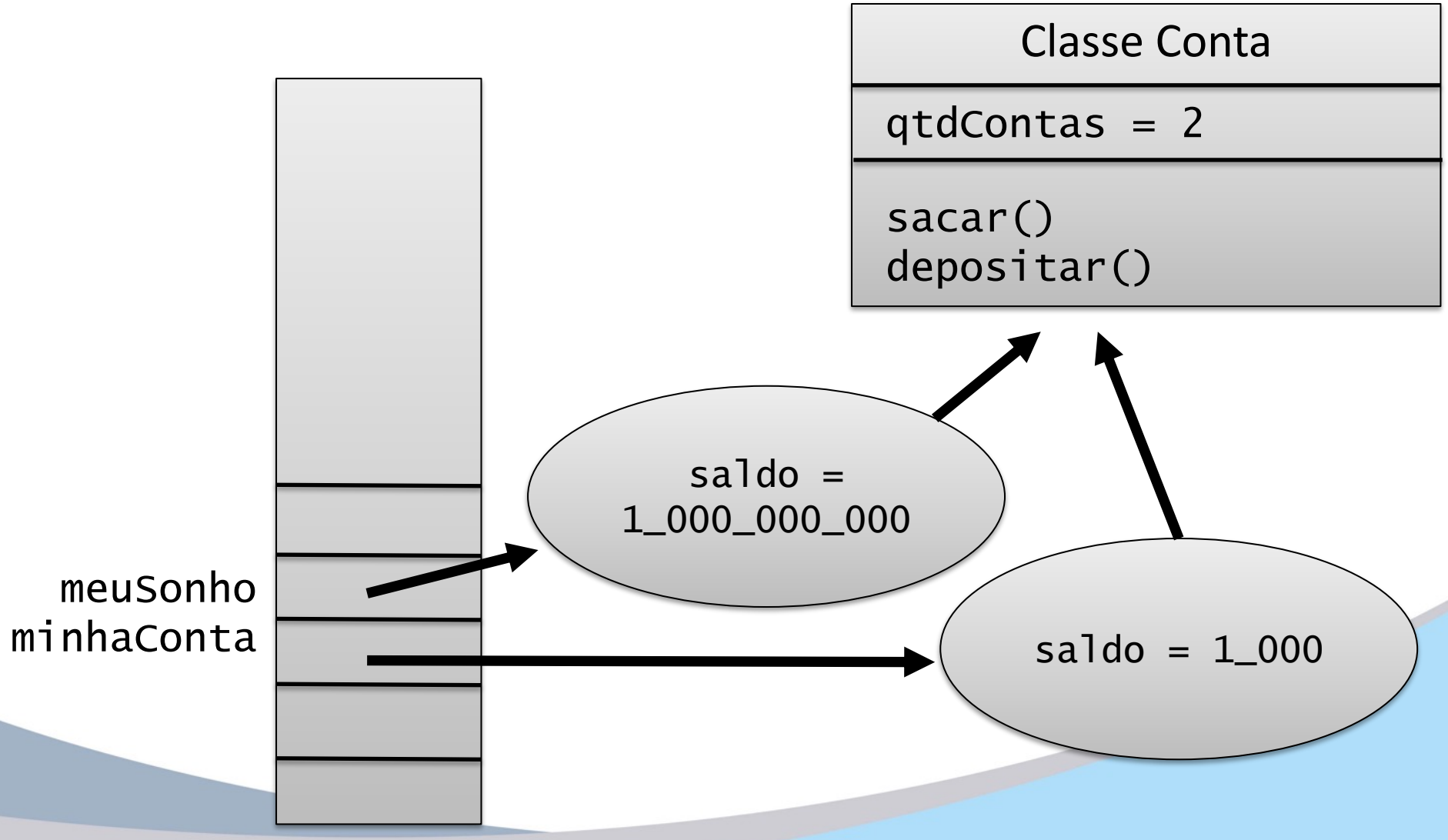
```
class Conta {
  // ...
  private static int qtdContas = 0;
  public Conta() { qtdContas++; /* ... */ }
  public static int getQtdContas() {
    return qtdContas;
  }
}
```

```
    Conta c1 = new Conta();
    Conta c2 = new Conta();

    // Quantas contas foram criadas?
    System.out.println(Conta.getQtdContas());
```

TesteConta

UNIVERSIDADE FEDERAL
DO ESPÍRITO SANTO

Armazenamento em memória



UNIVERSIDADE FEDERAL
DO ESPÍRITO SANTO

Contexto `static`

- Métodos static não podem acessar membros não-static:

```
public class Teste {
  private int atributo;
  private static int atributoStatic;

  public void metodo() { }

  public static void metodoStatic() { }

  public static void main(String[] args) {
    // ...
    System.out.println(atributoStatic);
    System.out.println(Teste.atributoStatic);
    metodoStatic();
    Teste.metodoStatic();
    // Continua...
```

UNIVERSIDADE FEDERAL
DO ESPÍRITO SANTO

Contexto static

```
    // Não pode:
    // System.out.println(atributo);
    // System.out.println(Teste.atributo);
    // metodo();
    // Teste.metodo();

    // Preciso de um objeto:
    Teste t = new Teste();
    System.out.println(t.atributo);
    t.metodo();
  }
}
```

UNIVERSIDADE FEDERAL
DO ESPÍRITO SANTO

Blocos de inicialização estática

- No exemplo da classe `Aleatorio`, inicializamos uma variável no construtor porque não conseguíamos fazê-lo em uma só linha;
- E se esta variável for `static`?

```
class Aleatorio {
  int numero;

  Aleatorio(int max) {
    Random rand = new Random();
    numero = rand.nextInt(max);
  }
}
```

UNIVERSIDADE FEDERAL
DO ESPÍRITO SANTO

Blocos de inicialização estática

- Resolvemos a questão com blocos de inicialização estática;
- Os blocos estáticos de uma classe são executados quando a classe é usada pela 1ª vez.

```
class Aleatorio {
  static int numero;

  static {
    Random rand = new Random();
    numero = rand.nextInt(20);
  }
}
```

UNIVERSIDADE FEDERAL
DO ESPÍRITO SANTO

Blocos de inicialização estática

- Também podemos fazer blocos de inicialização não-estática;
- Funcionam como os construtores: chamados em cada criação de objeto.

```
class Aleatorio {
  int numero;

  {
    Random rand = new Random();
    numero = rand.nextInt(20);
  }
}
```

UNIVERSIDADE FEDERAL
DO ESPÍRITO SANTO

Cuidado com o `static`

- Membros estáticos são como variáveis/funções globais;
- São mais procedurais do que orientados a objetos;
- No entanto, em alguns casos são necessários...

UNIVERSIDADE FEDERAL
DO ESPÍRITO SANTO

Exercitar é fundamental

- Apostila FJ-11 da Caelum:
  - *Seção 6.8, página 81 (class Funcionario);*
  - *Seção 6.9, página 83 (desafios).*