Linguagem para Aplicações de Internet I

Modificadores de Acesso e Atributos de Classe

Prof. Ânderson Kanegae Soares Rocha
https://about.me/kanegae

Objetivo da Aula

- Ao final da aula, o aluno será capaz de compreender e utilizar modificadores de acesso e atributos de classe.

Controlando o Acesso

```
class Conta {
  int numero;
  Cliente titular;
  double saldo;
  double limite;

  // ...

  void sacar(double valor) {
    this.saldo = this.saldo - valor;
  }
}
```

Controlando o Acesso

```
class TestaContaEstouro1 {
  public static void main(String args[]) {
    Conta minhaConta = new Conta();
    minhaConta.saldo = 1000.0;
    minhaConta.limite = 1000.0;
    // saldo + limite = 2000!
    minhaConta.sacar(50000.0);
  }
}
```

Controlando o Acesso

```
class TestaContaEstouro2 {
  public static void main(String args[]) {
    Conta minhaConta = new Conta();
    minhaConta.limite = 100.0;
    // saldo abaixo dos 100 de limite!
    minhaConta.saldo = -200.0;
  }
}
```

Controlando o Acesso

```
class TestaContaEstouro3 {
  public static void main(String args[]) {
    Conta minhaConta = new Conta();
    minhaConta.limite = 100.0;
    minhaConta.saldo = 100.0;
    double novoSaldo = -200.0;
    if (novoSaldo < -minhaConta.limite) {
      System.out.println("Saldo inválido!");
    } else {
      minhaConta.saldo = novoSaldo;
    }
  }
}
```

Momento da Reflexão

- Resolvemos o problema?
  - Quais os benefícios e malefícios dessa solução?

Controlando o Acesso

```
class Conta {
  private int numero;
  private Cliente titular;
  private double saldo;
  private double limite;

  // ...

  void sacar(double valor) {
    this.saldo = this.saldo - valor;
  }
}
```

Controlando o Acesso

```
class TestaAcessoDireto {
    public static void main(String args[]) {
        Conta minhaConta = new Conta();
        minhaConta.saldo = 1000.0; // Não compila!
    }
}
```

Controlando o Acesso

```
class Conta {
  private double saldo;
  private double limite;

  // ...

  public void sacar(double valor) {
    if (valor > this.saldo + this.limite) {
      System.out.println("Operação não permitida!");
    } else {
     this.saldo = this.saldo - valor;
    }
  }
}
```

Controlando o Acesso

- E quando não tiver modificador de acesso?
  - O método ou atributo fica num estado de visibilidade intermediário entre o `private` e o `public`.
  - Vamos estudar esse estado de visibilidade intermediário nas próximas aulas, quando estudarmos os pacotes.

Encapsulamento

- Em poucas palavras, a ideia é esconder os atributos das classes, além de esconder como funcionam os métodos do nosso sistema.
- Encapsular é fundamental para que o sistema seja suscetível a mudanças:
  - Regra de negócio centralizada e encapsulada!
  - Lembram do nosso método **`sacar`**?
    - Não é necessário que todos conheçam a regra de negócio para realizar um saque!

Encapsulamento

```
class Cliente {
  private String nome;
  private String cpf;

  public void mudarCPF(String cpf) {
    validarCPF(cpf);
    this.cpf = cpf;
  }

  private void validarCPF(String cpf) {
    // validação do CPF aqui!
  }

  // ...
}
```

Momento da Reflexão

- O modificador de acesso `private` faz com que apenas a própria classe seja capaz de modificar, acessar ou ler seus atributos.
  - Como realizar alguma operação em um atributo privado a partir de outra classe quando necessário?

Getters e Setters

```
class Conta {
  private double saldo;

  public double pegarSaldo() {
    return this.saldo;
  }

  public void depositar(double valor) {
    this.saldo = this.saldo + valor;
  }
}
```

Getters e Setters

```
class TestaAcesso {
    public static void main(String args[]) {
        Conta minhaConta = new Conta();
        minhaConta.depositar(1000.0);
        System.out.println("Saldo: " +
            minhaConta.pegarSaldo());
    }
}
```

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 e outro que muda o valor.
- A convenção para esses métodos é colocar a palavra **`get`** ou **`set`** antes do nome do atributo.
- Sendo assim, esses métodos são conhecidos como **`getters`** e **`setters`**.

Getters e Setters

```
class Conta {
  private double saldo;
  private double limite;
  private Cliente titular;

  public double getSaldo() {
    return this.saldo;
  }

  public void setSaldo(double saldo) {
    this.saldo = saldo;
  }

  // ...
```

Getters e Setters

```
  // ...

  public double getLimite() {
    return this.limite;
  }

  public void setLimite(double limite) {
    this.limite = limite;
  }

// ...
```

Getters e Setters

```
  // ...

  public Cliente getTitular() {
    return this.titular;
  }

  public void setTitular(Cliente titular) {
    this.titular = titular;
  }
}
```

Getters e Setters

- **`Getters`** e **`Setters`** devem ser criados apenas se forem realmente necessários.
  - Note que, por exemplo, o método **`setSaldo`** não deveria ter sido criado pois queremos que todos usem os métodos **`depositar`** e **`sacar`**.
- **`Getters`** e **`Setters`** não precisam ser métodos exclusivos para acesso aos atributos, eles também podem encapsular regras de negócio.
  - Note que, por exemplo, o método **`getSaldo`** pode retornar o saldo já acrescido do limite.

Getters e Setters

- No caso de atributos booleanos, pode-se usar no lugar do **`get`** o prefixo **`is`**.
  - Dessa maneira, caso tivéssemos um atributo booleano **`ligado`**, em vez de **`getLigado`** poderíamos ter **`isLigado`**.

Construtores

- Quando utilizamos a palavra reservada **`new`**, estamos construindo um objeto.
- Sempre que o **`new`** é invocado, ele executa o construtor da classe.
- O construtor da classe é um bloco de código com o mesmo nome que a classe.
- O construtor não é um método!

Construtores

```
class Conta {
  int numero;
  Cliente titular;
  double saldo;
  double limite;

  // construtor
  Conta() {
    System.out.println("Construindo uma conta!");
  }

  // ...
}
```

Momento da Reflexão

- Até agora, não estávamos implementando construtores em nossas classes.
  - Como era possível utilizar o `new` se ele **obrigatoriamente** executa o construtor?

Construtores

- Quando você não declara nenhum construtor na sua classe, o Java cria um para você.
- Esse construtor é o **construtor *default***, ele não recebe nenhum argumento e o corpo dele é vazio.
- A partir do momento em que você declara um construtor, o **construtor *default*** não é mais fornecido.

Construtores

```
class Conta {
  int numero;
  Cliente titular;
  double saldo;
  double limite;

  // construtor
  Conta(Cliente titular) {
    this.titular = titular;
  }

  // ...
}
```

Construtores

```
class TestaConstrutor {
  public static void main(String args[]) {
    Cliente fulano = new Cliente();
    meuCliente.setNome("Fulano");
    Conta minhaConta = new Conta(fulano);
    // ...
  }
}
```

Construtores

```
class TestaConstrutor {
  public static void main(String args[]) {
    Cliente fulano = new Cliente();
    meuCliente.setNome = "Fulano";
    Conta minhaConta = new Conta(); // Não compila!
    // ...
  }
}
```

Momento da Reflexão

- Até agora, tudo estava funcionando.
  - Para que utilizar construtores?

Construtores

- Se todo objeto de um determinado tipo obrigatoriamente precisa conter outro objeto, podemos utilizar um construtor!
  - Se toda conta obrigatoriamente precisa ter um titular, basta definir um construtor que receba um titular como parâmetro!
  - Não se pode abrir um arquivo para leitura sem informar qual é esse arquivo. Logo, nada mais natural do que receber um parâmetro identificando esse arquivo no construtor!

Construtores

```
class Conta {
  int numero;
  Cliente titular;
  double saldo;
  double limite;

  Conta(Cliente titular) {
    this.titular = titular;
  }

  Conta(int numero, Cliente titular) {
    this(titular); // executa o construtor que recebe um Cliente
    this.numero = numero;
  }
  // ...
}
```

Momento da Reflexão

- Os atributos pertencem às suas classes e é papel delas controlar o acesso a eles.
  - Como obter a quantidade de objetos de uma determinada classe?

Atributos de Classe

```
// alternativa #1
int totalDeContas = 0;
Conta c1 = new Conta();
totalDeContas = totalDeContas + 1;
Conta c2 = new Conta();
totalDeContas = totalDeContas + 1;
Conta c3 = new Conta();
totalDeContas = totalDeContas + 1;
Conta c4 = new Conta();
totalDeContas = totalDeContas + 1;
// ...
```

Atributos de Classe

```
// alternativa #2
class Conta {
  private int totalDeContas;

  // ...

  Conta() {
    this.totalDeContas = this.totalDeContas + 1;
  }
}
```

Atributos de Classe

```
// alternativa #3
class Conta {
  private static int totalDeContas;

  //...

  Conta() {
    Conta.totalDeContas = Conta.totalDeContas + 1;
  }

  public int getTotalDeContas() {
    return Conta.totalDeContas;
  }
}
```

Atributos de Classe

```
// alternativa #3 (continuação)
Conta c = new Conta();
int total = c.getTotalDeContas();
// ...
```

Atributos de Classe

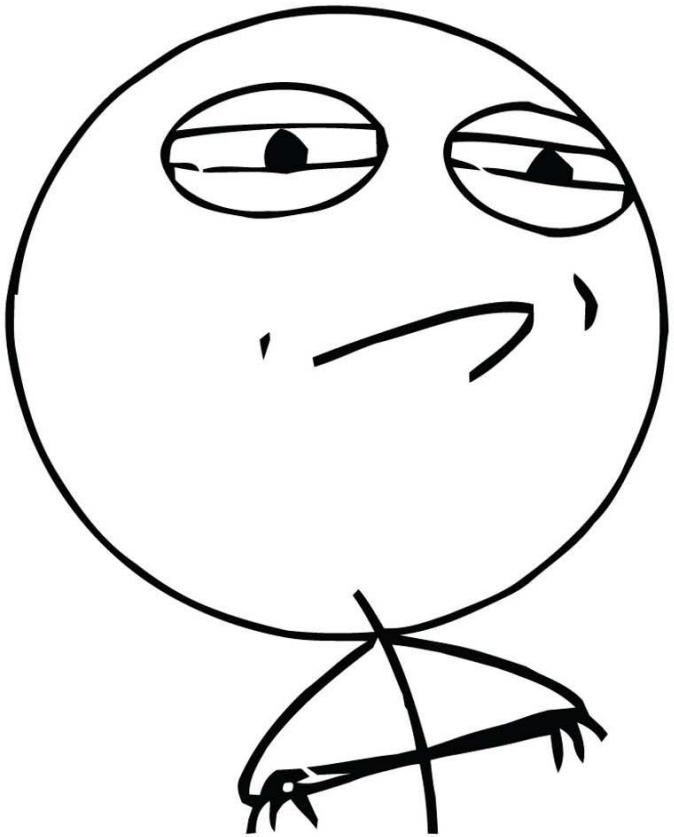
```
// alternativa #4
class Conta {
  private static int totalDeContas;

  //...

  Conta() {
    Conta.totalDeContas = Conta.totalDeContas + 1;
  }

  public static int getTotalDeContas() {
    return Conta.totalDeContas;
  }
}
```

Atributos de Classe

```
// alternativa #4 (continuação)
int total = Conta.getTotalDeContas();
// ...
```

Atributos e Métodos de Classe

- Métodos e atributos estáticos só podem acessar outros métodos e atributos estáticos da mesma classe, o que faz todo sentido já que dentro de um método estático não temos acesso à referência `this`, pois um método estático é chamado através da classe, e não de um objeto.



CHALLENGE ACCEPTED

Exercício de Fixação

- Reescreva a classe **`Conta`** apresentada anteriormente fazendo com que o saldo possa ser alterado apenas por meio dos métodos **`depositar`** e **`sacar`**. Em seguida, implemente uma classe para realizar os devidos testes.

Exercício de Fixação

- Adicione um construtor a classe **`Conta`** que seja capaz de inicializar todos os seus atributos. Em seguida, implemente um programa para instanciar dois objetos do tipo **`Conta`** sendo que apenas um deles deve ser instanciado por meio desse construtor.

Exercício de Fixação

- Implemente uma classe chamada **`Contador`** que seja capaz de contar quantas vezes o seu método **`contar()`** foi executado em uma determinada instancia. Em seguida, implemente um programa que utilize essa classe, execute seu método algumas vezes e exiba a quantidade de vezes que ele foi executado. Utilize um construtor para validar o valor inicial da contagem pois ele não deve ser negativo.

Exercício de Fixação

- Implemente um classe chamada **`Calculadora`** que contenha dois operadores inteiros e seja capaz de realizar operações aritméticas básicas (adição, subtração, multiplicação e divisão). Em seguida, implemente um construtor para informar os operandos e utilize essa classe para realizar as operações aritméticas básicas.

Exercício de Fixação

- Implemente um programa que utilize um *array* (vetor) para armazenar os funcionários de uma empresa. Utilize construtores para que o "tamanho" desse *array* não seja fixo. Não esqueça que é recomendado que os atributos sejam privados e, portanto, implemente os **`getters`** e **`setters`** necessários.

Exercício de Fixação

- Altere o exercício anterior para que a classe **`Funcionario`** possua diversos construtores com parâmetros diferentes, sendo que deve haver reaproveitamento de código entre eles. Em seguida, altere a classe principal para fazer uso desses construtores.

Exercício de Fixação

- Implemente um programa capaz de simular uma fábrica de carros. Esse programa deve conter o método **`produzirCarro`** que instancia e retorna um objeto do tipo **`Carro`**. Esse mesmo programa deve conter um método **`contarCarros`** que exibe o total de objetos do tipo **`Carro`** que foram produzidos por essa fábrica.

Referências e materiais extras

- https://www.caelum.com.br/apostila-java-orientacao-objetos/modificadores-de-acesso-e-atributos-de-classe/
- http://blog.caelum.com.br/nao-aprender-oo-getters-e-setters/