Aula 15: Modificadores de acesso e atributos de classe

Professor(a): Virgínia Fernandes Mota
http://www.dcc.ufmg.br/∼virginiaferm

ALGORITMOS E ESTRUTURAS DE DADOS - SETOR DE INFORMÁTICA

Modificadores de acesso e atributos de classe

- Ao término desta aula, você será capaz de:
  - controlar o acesso aos seus métodos, atributos e construtores através dos modificadores private e public;
  - escrever métodos de acesso a atributos do tipo getters e setters;
  - escrever construtores para suas classes;
  - utilizar variáveis e métodos estáticos.

Controlando o acesso

- Um dos problemas mais simples que temos no nosso sistema de contas é que o método saca permite sacar mesmo que o limite tenha sido atingido.

```
1  class Conta {
2    int numero;
3    Cliente titular;
4    double saldo;
5    double limite;
6
7    // ..
8
9    void saca(double quantidade) {
10     this.saldo = this.saldo - quantidade;
11   }
12 }
```

- Solução: Podemos incluir um if dentro do nosso método saca().

Controlando o acesso

- Apesar de melhorar bastante, ainda temos um problema mais grave: ninguém garante que o usuário da classe vai sempre utilizar o método para alterar o saldo da conta.
- Como evitar isso? Uma ideia simples seria testar se não estamos ultrapassando o limite toda vez que formos alterar o saldo! (Não muito legal)
- Solução: Forçar quem usa a classe Conta a invocar o método saca e não permitir o acesso direto ao atributo!

Controlando o acesso

- Declarar que os atributos **não podem ser acessados de fora da classe** através da palavra chave **private**:

```
1  class Conta {
2    private double saldo;
3    private double limite;
4    // ...
5  }
```

- private é um modificador de acesso (também chamado de modificador de visibilidade).
- Na orientação a objetos, é prática quase que obrigatória proteger seus atributos com private.
- private também pode ser usada para modificar o acesso a um método.

Controlando o acesso

- Da mesma maneira que temos o private, temos o modificador **public**, que **permite a todos acessarem um determinado atributo ou método**:

```
1  class Conta {
2    //...
3    public void saca(double quantidade) {
4      //posso sacar até saldo+limite
5      if (quantidade > this.saldo + this.limite){
6        System.out.println("Não posso sacar fora do
             limite!");
7      } else {
8        this.saldo = this.saldo - quantidade;
9      }
10   }
11 }
```

- É muito comum que seus atributos sejam private e quase todos seus métodos sejam public (não é uma regra!)

Encapsulamento

- O que começamos a ver nessa aula é a ideia de **encapsular**.
- Encapsular é **fundamental** para que seu sistema seja suscetível a mudanças: não precisaremos mudar uma regra de negócio em vários lugares, mas sim em apenas um único lugar, já que essa regra está **encapsulada**.
- O conjunto de métodos públicos de uma classe é também chamado de **interface da classe**, pois esta é a única maneira a qual você se comunica com objetos dessa classe.

Encapsulamento

- Sempre que vamos acessar um objeto, utilizamos sua interface. Existem diversas analogias fáceis no mundo real:
  - Dirigir um carro: Volante e pedais (interface), Motor (implementação).
  - Celulares: Todos os celulares são iguais (interface), eles possuem maneiras (métodos) de discar, ligar, desligar, atender... O que muda é como fazem (implementação).

Encapsulamento

```
1  class Cliente {
2    private String nome;
3    private String endereco;
4    private String cpf;
5    private int idade;
6
7    public void mudaCPF(String cpf) {
8      validaCPF(cpf);
9      this.cpf = cpf;
10   }
11
12   private void validaCPF(String cpf) {
13     // série de regras aqui, falha caso não seja válido
14   }
```

- Se alguém tentar criar um Cliente e não usar o mudaCPF para alterar um cpf diretamente, vai receber um erro de compilação, já que o atributo CPF é privado!
- O controle sobre o CPF está centralizado: ninguém consegue acessá-lo sem passar por aí, a classe Cliente é a única responsável pelos seus próprios atributos!

Getters e Setters

- Nós protegemos o acesso ao atributo com o private... Mas e agora? Como acessá-lo?
- Vamos criar métodos para isso!
- A convenção para esses métodos é de colocar a palavra *get* ou *set* antes do nome do atributo.

Getters e Setters

```
1  class Conta {
2    private double saldo;
3    private double limite;
4    private Cliente titular;
5
6    public double getSaldo() {
7      return this.saldo;
8    }
9    public void setSaldo(double saldo) { //REALMENTE PRECISA??
10     this.saldo = saldo;
11   }
12   public double getLimite() {
13     return this.limite;
14   }
15   public void setLimite(double limite) {
16     this.limite = limite;
17   }
18   public Cliente getTitular() {
19     return this.titular;
20   }
21   public void setTitular(Cliente titular) {
22     this.titular = titular;
23   }
24 }
```

Getters e Setters

- Você só deve criar um getter ou setter se tiver a real necessidade.
- Nesse exemplo setSaldo não deveria ter sido criado, já que queremos que todos usem deposita() e saca().
- Utilizar getters e setters não só ajuda você a proteger seus atributos, como também possibilita ter de mudar algo em um só lugar... $\rightarrow$ encapsulamento

Construtores

- **new**: Usamos essa palavra chave para *construir* um objeto.
- Sempre quando o new é chamado, ele executa o **construtor da classe**.

```
1  class Conta {
2    int numero;
3    Cliente titular;
4    double saldo;
5    double limite;
6
7    // construtor
8    Conta() {
9      System.out.println("Construindo uma conta.");
10   }
11
12   // ..
13 }
```

Construtores

```
1  Conta c = new Conta();
2  //Vai aparecer a mensagem!
```

- Importante: Um construtor pode parecer, mas não é um método.
- Se você não criar o construtor, o Java cria um **construtor default**: ele não recebe nenhum argumento e o corpo dele é vazio.

Construtores

- Um construtor pode receber um argumento, podendo assim inicializar algum tipo de informação:

```
1  class Conta {
2    int numero;
3    Cliente titular;
4    double saldo;
5    double limite;
6
7    // construtor
8    Conta(Cliente titular) {
9      this.titular = titular;
10   }
11
12   // ..
13 }
```

Construtores

```
1  Cliente carlos = new Cliente();
2  carlos.nome = "Carlos";
3
4  Conta c = new Conta(carlos);
5  System.out.println(c.titular.nome);
```

A necessidade de um construtor

- Tudo estava funcionando até agora. Para que utilizamos um construtor?
- Idéia: Se toda conta precisa de um titular, como obrigar todos os objetos que forem criados a ter um valor desse tipo?
  - criar um único construtor que recebe essa String!
- Construtor: Dar possibilidades ou obrigar o usuário de uma classe a passar argumentos para o objeto durante o processo de criação do mesmo.

Chamando outro construtor

- Um construtor só pode rodar durante a construção do objeto!
- Porém, durante a construção de um objeto, você pode fazer com que um construtor chame outro...

```
1  class Conta {
2    int numero;
3    Cliente titular;
4    double saldo;
5    double limite;
6    // construtor
7    Conta (Cliente titular) {
8      //   faz mais uma série de inicializações e
           configurações
9      this.titular = titular;
10   }
11
12   Conta (int numero, Cliente titular) {
13     this(titular); // chama o construtor que foi
           declarado acima
14     this.numero = numero;
15   }
16   //..
17 }
```

Java Bean

- Observação: Quando criamos uma classe com todos os atributos privados, seus getters e setters e um construtor vazio (padrão), na verdade estamos criando um Java Bean.

Atributos de classe

- Nosso banco também quer controlar a quantidade de contas existentes no sistema.

```
1  Conta c = new Conta();
2  totalDeContas = totalDeContas + 1;
```

- Problemas?

Atributos de classe

- Estamos espalhando um código por toda aplicação, e quem garante que vamos conseguir lembrar de incrementar a variável totalDeContas toda vez?
- Tentamos então, passar para a seguinte proposta:

```
1  class Conta {
2    private int totalDeContas;
3    //...
4
5    Conta() {
6      this.totalDeContas = this.totalDeContas + 1;
7    }
8  }
```

- Problemas?

Atributos de classe

- **O atributo é de cada objeto.**
- Seria interessante então que essa variável fosse **única**, **compartilhada por todos os objetos** dessa classe.
- Para fazer isso em java, declaramos a variável como **static**.

```
1  private static int totalDeContas;
```

- Não é mais um atributo de cada objeto, e sim um **atributo da classe**.

Atributos de classe

- Para acessarmos um atributo estático, não usamos a palavra chave this, mas sim o nome da classe:

```
1  class Conta {
2    private static int totalDeContas;
3    //...
4
5    Conta() {
6      Conta.totalDeContas = Conta.totalDeContas + 1;
7    }
8  }
```

Atributos de classe

- Já que o atributo é privado, como podemos acessar essa informação a partir de outra classe? Precisamos de um getter para ele!

```
1  class Conta {
2    private static int totalDeContas;
3    //...
4
5    Conta() {
6      Conta.totalDeContas = Conta.totalDeContas + 1;
7    }
8
9    public int getTotalDeContas() {
10     return Conta.totalDeContas;
11   }
12 }
```

Atributos de classe

- Então, quantas contas foram criadas?

```
1  Conta c = new Conta();
2  int total = c.getTotalDeContas();
```

- Problema: Precisamos criar uma conta antes de chamar o método! Mas, gostaríamos de saber quantas contas existem sem precisar ter acesso a um objeto conta.
- Solução: Transformar esse método que todo objeto conta tem em um método de toda a classe.

Atributos de classe

- Usamos a palavra static de novo:

```
1  public static int getTotalDeContas() {
2    return Conta.totalDeContas;
3  }
```

- Para acessar esse novo método:

```
1  int total = Conta.getTotalDeContas();
```

Atributos de classe

- Métodos e atributos estáticos só podem acessar outros métodos e atributos estáticos da mesma classe
- O static realmente traz um "cheiro"procedural, porém em muitas vezes é necessário.

Exercícios: Encapsulamento, construtores e static

- 1. Adicione o modificador de visibilidade (private, se necessário) para cada atributo e método da classe Funcionario. Tente criar um Funcionario no main e modificar ou ler um de seus atributos privados. O que acontece?
- 2. Crie os getters e setters necessários da sua classe Funcionario.
- 3. Modifique suas classes que acessam e modificam atributos de um Funcionario para utilizar os getters e setters recém criados.
- 4. Faça com que sua classe Funcionario possa receber, opcionalmente, o nome do Funcionario durante a criação do objeto. Utilize construtores para obter esse resultado.
  Dica: utilize um construtor sem argumentos também, para o caso de a pessoa não querer passar o nome do Funcionario.

Exercícios: Encapsulamento, construtores e static

- 5. Adicione um atributo na classe Funcionario de tipo int que se chama identificador. Esse identificador deve ter um valor único para cada instância do tipo Funcionario. O primeiro Funcionario instanciado tem identificador 1, o segundo 2, e assim por diante. Você deve utilizar os recursos aprendidos aqui para resolver esse problema.
- 6. Crie os getters e setters da sua classe Empresa e coloque seus atributos como private. Lembre-se de que não necessariamente todos os atributos devem ter getters e setters.
- 7. Como garantir que datas como 31/2/2012 não sejam aceitas pela sua classe Data?

Na próxima aula...

Herança, reescrita e polimorfismo