

Orientação a Obejtos Classica

Namom Alves Alencar



CONTROLE DE 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. A seguir você pode lembrar como está a classe Conta:

```
class Conta {  
    int numero;  
    Cliente titular;  
    double saldo;  
    double limite;  
  
    // ..  
  
    void saca(double quantidade) {  
        this.saldo = this.saldo - quantidade;  
    }  
}
```

CONTROLE DE ACESSO

- A classe a seguir mostra como é possível ultrapassar o limite usando o método saca:

```
class TestaContaEstouro1 {  
    public static void main(String args[]) {  
        Conta minhaConta = new Conta();  
        minhaConta.saldo = 1000.0;  
        minhaConta.limite = 1000.0;  
        minhaConta.saca(50000); // saldo + limite é só 2000!!  
    }  
}
```

- Podemos incluir um if dentro do nosso método saca() para evitar a situação que resultaria em uma conta em estado inconsistente, com seu saldo abaixo do limite. Fizemos isso no capítulo de orientação a objetos básica.

CONTROLE DE 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. O código a seguir ultrapassa o limite diretamente:

```
class TestaContaEstouro2 {  
    public static void main(String args[]) {  
        Conta minhaConta = new Conta();  
        minhaConta.limite = 100;  
        minhaConta.saldo = -200;  
        //saldo está abaixo dos 100 de limite  
    }  
}
```

- Como evitar isso?

CONTROLE DE ACESSO

- Uma ideia simples seria testar se não estamos ultrapassando o limite toda vez que formos alterar o saldo:

```
class TestaContaEstouro3 {  
    public static void main(String args[]) {  
        Conta minhaConta = new Conta();  
        minhaConta.limite = 100;  
        minhaConta.saldo = 100;  
        // quero mudar o saldo para -200  
        double novoSaldo = -200;  
        // testa se o novoSaldo ultrapassa o limite da conta  
        if (novoSaldo < -minhaConta.limite) { //  
            System.out.println("Não posso mudar para esse saldo");  
        } else {  
            minhaConta.saldo = novoSaldo;  
        } } }
```

CONTROLE DE ACESSO

- Esse código iria se repetir ao longo de toda nossa aplicação e, pior, alguém pode esquecer de fazer essa comparação em algum momento, deixando a conta na situação inconsistente. A melhor forma de resolver isso seria forçar quem usa a classe Conta a invocar o método `saca` e não permitir o acesso direto ao atributo. É o mesmo caso da validação de CPF.
- Para fazer isso no Java, basta declarar que os atributos não podem ser acessados de fora da classe através da palavra chave `private`.

CONTROLE DE ACESSO

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

- Private é um modificador de acesso (também chamado de modificador de visibilidade).
- Marcando um atributo como privado, fechamos o acesso ao mesmo em relação a todas as outras classes, fazendo com que o seguinte código não compile:

```
class TestaAcessoDireto {  
    public static void main(String args[]) {  
        Conta minhaConta = new Conta();  
        //não compila! você não pode acessar o atributo privado de outra classe  
        minhaConta.saldo = 1000;  
    }  
}
```

CONTROLE DE ACESSO

- Na orientação a objetos, é prática quase que obrigatória proteger seus atributos com `private`. (discutiremos outros modificadores de acesso em outros capítulos).
- Cada classe é responsável por controlar seus atributos, portanto ela deve julgar se aquele novo valor é válido ou não! Esta validação não deve ser controlada por quem está usando a classe e sim por ela mesma, centralizando essa responsabilidade e facilitando futuras mudanças no sistema. Muitas outras vezes nem mesmo queremos que outras classes saibam da existência de determinado atributo, escondendo-o por completo, já que ele diz respeito ao funcionamento interno do objeto.

CONTROLE DE ACESSO

- Repare que, quem invoca o método saca não faz a menor ideia de que existe um limite que está sendo checado. Para quem for usar essa classe, basta saber o que o método faz e não como exatamente ele o faz (o que um método faz é sempre mais importante do que como ele faz: mudar a implementação é fácil, já mudar a assinatura de um método vai gerar problemas).
- A palavra chave `private` também pode ser usada para modificar o acesso a um método. Tal funcionalidade é utilizada em diversos cenários: quando existe um método que serve apenas para auxiliar a própria classe e quando há código repetido dentro de dois métodos da classe são os mais comuns. Sempre devemos expôr o mínimo possível de funcionalidades, para criar um baixo acoplamento entre as nossas classes.

CONTROLE DE ACESSO

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

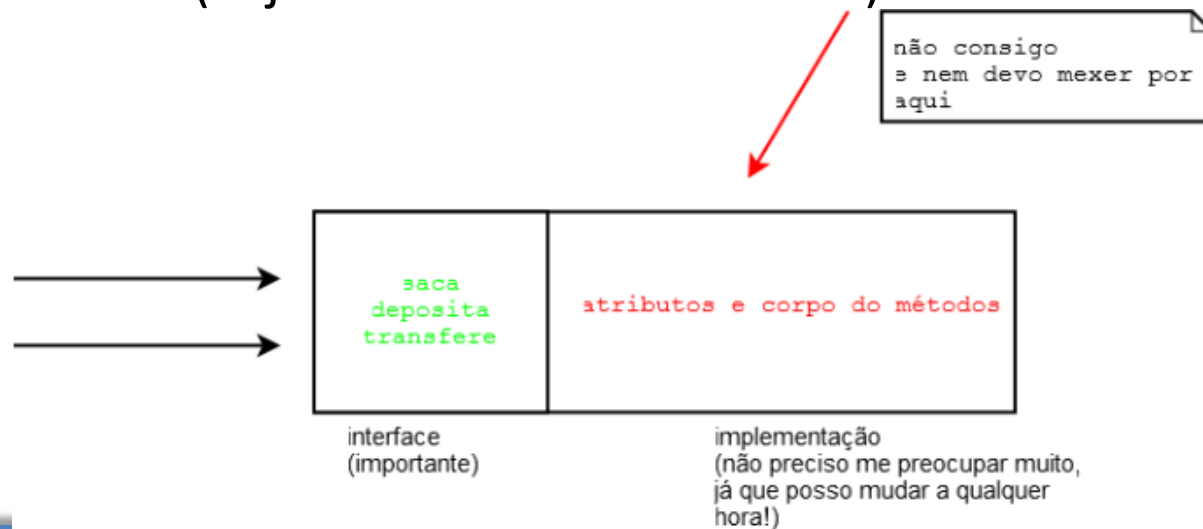
```
class Conta {  
    //...  
    public void saca(double quantidade) {  
        //posso sacar até saldo+limite  
        if (quantidade > this.saldo + this.limite){  
            System.out.println("Não posso sacar fora do limite!");  
        } else {  
            this.saldo = this.saldo - quantidade;  
        } } }  
}
```

CONTROLE DE ACESSO

- É muito comum, e faz todo sentido, que seus atributos sejam `private` e quase todos seus métodos sejam `public` (não é uma regra!). Desta forma, toda conversa de um objeto com outro é feita por troca de mensagens, isto é, acessando seus métodos. Algo muito mais educado que mexer diretamente em um atributo que não é seu!
- Melhor ainda! O dia em que precisarmos mudar como é realizado um saque na nossa classe `Conta`, adivinhe onde precisaríamos modificar? Apenas no método `saca`, o que faz pleno sentido. Como exemplo, imagine cobrar CPMF de cada saque: basta você modificar ali, e nenhum outro código, fora a classe `Conta`, precisará ser recompilado. Mais: as classes que usam esse método nem precisam ficar sabendo de tal modificação! Você precisa apenas recompilar aquela classe e substituir aquele arquivo `.class`. Ganhamos muito em esconder o funcionamento do nosso método na hora de dar manutenção e fazer modificações.

ENCAPSULAMENTO

- Encapsular é esconder todos os membros de uma classe (como vimos acima), além de esconder como funcionam as rotinas (no caso métodos) do nosso sistema.
- 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. (veja o caso do método saca)



ENCAPSULAMENTO - INTERFACE

- 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.
- Sempre que vamos acessar um objeto, utilizamos sua interface. Existem diversas analogias fáceis no mundo real:

Exemplo: Quando você dirige um carro, o que te importa são os pedais e o volante (interface) e não o motor que você está usando (implementação). É claro que um motor diferente pode te dar melhores resultados, mas **o que ele faz** é o mesmo que um motor menos potente, a diferença está em **como ele faz**. Para trocar um carro a álcool para um a gasolina você não precisa reaprender a dirigir! (trocar a implementação dos métodos não precisa mudar a interface, fazendo com que as outras classes continuem usando eles da mesma maneira).

ENCAPSULAMENTO

- Já temos conhecimentos suficientes para resolver aquele problema da validação de CPF:

```
class Cliente {  
    //...  
    private String cpf;  
    public void mudaCPF(String cpf) {  
        validaCPF(cpf);  
        this.cpf = cpf;  
    }  
    private void validaCPF(String cpf) {  
        // série de regras aqui, falha caso não seja válido  
    }  
}
```

ENCAPSULAMENTO

- Se alguém tentar criar um Cliente e não usar o mudaCPF para alterar um cpfdiretamente, vai receber um erro de compilação, já que o atributo CPF é privado. E o dia que você não precisar verificar o CPF de quem tem mais de 60 anos? Seu método fica o seguinte:

```
public void mudaCPF(String cpf) {  
    if (this.idade <= 60) {  
        validaCPF(cpf);  
    }  
    this.cpf = cpf;  
}
```

- 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

- O modificador `private` faz com que ninguém consiga modificar, nem mesmo ler, o atributo em questão. Com isso, temos um problema: como fazer para mostrar o saldo de uma Conta, já que nem mesmo podemos acessá-lo para leitura?
- Precisamos então arranjar uma maneira de fazer esse acesso. Sempre que precisamos arrumar uma maneira de fazer alguma coisa com um objeto, utilizamos de métodos! Vamos então criar um método, digamos `pegaSaldo`, para realizar essa simples tarefa:

```
private double saldo;  
//...  
public double pegaSaldo() {  
    return this.saldo;  
}
```


GETTERS E SETTERS

- Para acessarmos o saldo de uma conta, podemos fazer:

```
class TestaAcessoComPegaSaldo {  
    public static void main(String args[]) {  
        Conta minhaConta = new Conta();  
        minhaConta.deposita(1000);  
        System.out.println("Saldo: " + minhaConta.pegarSaldo());  
    }  
}
```

- 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.

GETTERS E SETTERS

- A convenção para esses métodos é de colocar a palavra get ou set antes do nome do atributo. Por exemplo, a nossa conta com saldo, limite e titular fica assim, no caso da gente desejar dar acesso a leitura e escrita a todos os atributos:

```
class Conta {  
    private double saldo;  
    public double getSaldo() {  
        return this.saldo;  
    }  
    public void setSaldo(double saldo) {  
        this.saldo = saldo;  
    }  
}
```

GETTERS E SETTERS

- É uma má prática criar uma classe e, logo em seguida, criar getters e setters para todos seus atributos. Você só deve criar um getter ou setter se tiver a real necessidade. Repare que nesse exemplo `setSaldo` não deveria ter sido criado, já que queremos que todos usem `deposita()` e `saca()`.
- Outro detalhe importante, um método `getX` não necessariamente retorna o valor de um atributo que chama X do objeto em questão. Isso é interessante para o encapsulamento. Imagine a situação: queremos que o banco sempre mostre como saldo o valor do limite somado ao saldo (uma prática comum dos bancos que costuma iludir seus clientes). Poderíamos sempre chamar `c.getLimite()` + `c.getSaldo()`, mas isso poderia gerar uma situação de "replace all" quando precisássemos mudar como o saldo é mostrado. Podemos encapsular isso em um método e, porque não, dentro do próprio `getSaldo`? Veja a seguir:

GETTERS E SETTERS

```
class Conta {  
    private double saldo;  
    private double limite;  
    private Cliente titular;  
    public double getSaldo() {  
        return this.saldo + this.limite;  
    }  
    // deposita() saca() e transfere() omitidos  
    public Cliente getTitular() {  
        return this.titular;  
    }  
    public void setTitular(Cliente titular) {  
        this.titular = titular;  
    }  
}
```

GETTERS E SETTERS

- O código acima nem possibilita a chamada do método `getLimite()`, ele não existe. E nem deve existir enquanto não houver essa necessidade. O método `getSaldo()` não devolve simplesmente o saldo... e sim o que queremos que seja mostrado como se fosse o saldo. Utilizar getters e setters não só ajuda você a proteger seus atributos, como também possibilita ter de mudar algo em um só lugar... chamamos isso de encapsulamento, pois esconde a maneira como os objetos guardam seus dados. É uma prática muito importante.

GETTERS E SETTERS

- Nossa classe está totalmente pronta? Isto é, existe a chance dela ficar com menos dinheiro do que o limite? Pode parecer que não, mas, e se depositarmos um valor negativo na conta? Ficaríamos com menos dinheiro que o permitido, já que não esperávamos por isso. Para nos proteger disso basta mudarmos o método deposita() para que ele verifique se o valor é necessariamente positivo.
- Depois disso precisaríamos mudar mais algum outro código? A resposta é não, graças ao encapsulamento dos nossos dados.

CONSTRUTORES

- Quando usamos a palavra chave `new`, estamos construindo um objeto. Sempre quando o `new` é chamado, ele executa o construtor da classe. O construtor da classe é um bloco declarado com o mesmo nome que a classe:

```
class Conta {  
    int numero; Cliente titular; double saldo; double limite;  
    // construtor  
    Conta() {  
        System.out.println("Construindo uma conta.");  
    } // .. }
```

- Então, quando fazemos `"Conta c = new Conta();"` A mensagem "construindo uma conta" aparecerá. É como uma rotina de inicialização que é chamada sempre que um novo objeto é criado. Um construtor pode parecer, mas **não é** um método.

CONSTRUTORES

- O interessante é que um construtor pode receber um argumento, podendo assim inicializar algum tipo de informação:

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

- Esse construtor recebe o titular da conta. Assim, quando criarmos uma conta, ela já terá um determinado titular.

```
Cliente cliente1 = new Cliente();  
cliente1.nome = "Jose";  
Conta c = new Conta(cliente1);  
System.out.println(c.titular.nome);
```


CONSTRUTORES

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

CONSTRUTORES

- Por exemplo, não podemos abrir um arquivo para leitura sem dizer qual é o nome do arquivo que desejamos ler! Portanto, nada mais natural que passar uma String representando o nome de um arquivo na hora de criar um objeto do tipo de leitura de arquivo, e que isso seja obrigatório.
- Você pode ter mais de um construtor na sua classe e, no momento do new, o construtor apropriado será escolhido.

CONSTRUTORES

```
Conta (Cliente titular) {
```

```
    // faz mais uma série de inicializações e configurações
```

```
    this.titular = titular;
```

```
}
```

```
Conta (int numero, Cliente titular) {
```

```
    this(titular); // chama o construtor que foi declarado acima
```

```
    this.numero = numero;
```

```
}
```

CONSTRUTORES

- Existe um outro motivo, o outro lado dos construtores: facilidade. Às vezes, criamos um construtor que recebe diversos argumentos para não obrigar o usuário de uma classe a chamar diversos métodos do tipo 'set'.
- No nosso exemplo do CPF, podemos forçar que a classe Cliente receba no mínimo o CPF, dessa maneira um Cliente já será construído e com um CPF válido.

ATRIBUTOS DE CLASSE

- Nosso banco também quer controlar a quantidade de contas existentes no sistema. Como poderíamos fazer isto? A ideia mais simples:

```
Conta c = new Conta();
```

```
totalDeContas = totalDeContas + 1;
```

- Aqui, voltamos em um problema parecido com o da validação de CPF. 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:

```
class Conta {
```

```
    private int totalDeContas;
```

```
    Conta() {
```

```
        this.totalDeContas = this.totalDeContas + 1;
```

```
    }}
```

ATRIBUTOS DE CLASSE

- Quando criarmos duas contas, qual será o valor do totalDeContas de cada uma delas? Vai ser 1. Pois cada uma tem essa variável. O atributo é de cada objeto.
- Seria interessante então, que essa variável fosse única, compartilhada por todos os objetos dessa classe. Dessa maneira, quando mudasse através de um objeto, o outro enxergaria o mesmo valor. Para fazer isso em java, declaramos a variável como static.

private static int totalDeContas;

ATRIBUTOS DE CLASSE

- Quando declaramos um atributo como static, ele passa a não ser mais um atributo de cada objeto, e sim um atributo da classe, a informação fica guardada pela classe, não é mais individual para cada objeto.
- Para acessarmos um atributo estático, não usamos a palavra chave this, mas sim o nome da classe:

```
class Conta {  
    private static int totalDeContas;  
    Conta() {  
        Conta.totalDeContas = Conta.totalDeContas + 1;  
    }  
}
```

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!

```
class Conta {  
    private static int totalDeContas;  
    Conta() {  
        Conta.totalDeContas = Conta.totalDeContas + 1;  
    }  
    public int getTotalDeContas() {  
        return Conta.totalDeContas;  
    }  
}
```

Como fazemos então para saber quantas contas foram criadas?

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


ATRIBUTOS DE CLASSE

- Precisamos criar uma conta antes de chamar o método! Isso não é legal, pois gostaríamos de saber quantas contas existem sem precisar ter acesso a um objeto conta. A ideia aqui é a mesma, transformar esse método que todo objeto conta tem em um método de toda a classe. Usamos a palavra static de novo, mudando o método anterior.

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

- Para acessar esse novo método:

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

- Repare que estamos chamando um método não com uma referência para uma Conta, e sim usando o nome da classe.

EXERCICIOS

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.

EXERCICIOS

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.

EXERCICIOS

7. Na classe Empresa, em vez de criar um array de tamanho fixo, receba como parâmetro no construtor o tamanho do array de Funcionario.
8. Como garantir que datas como 31/2/2012 não sejam aceitas pela sua classe Data?
9. Crie a classe PessoaFisica. Queremos ter a garantia de que pessoa física alguma tenha CPF invalido, nem seja criada PessoaFisica sem cpf inicial. (você não precisa escrever o algoritmo de validação de cpf, basta passar o cpf por um método valida(String x)...)

Bons Estudos

Namom Alves Alencar

