CAPÍTULO 6

Modificadores de acesso e atributos de classe

"A marca do homem imaturo é que ele quer morrer nobremente por uma causa, enquanto a marca do homem maduro é querer viver modestamente por uma."
— J. D. Salinger

Ao término desse capítulo, 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.

6.1 - 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. 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;
  }
```

```
}
```

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.

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? 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[]) {
    // a Conta
    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;
    }
  }
}
```

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`:

```
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;
   }
}

TesteAcessoDireto.java:5 saldo has private access in Conta
                minhaConta.saldo = 1000;
                       ^
1 error
```

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.

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.

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;
    }
  }
}
```

E quando não há modificador de acesso?

Até agora, tínhamos declarado variáveis e métodos sem nenhum modificador como `private` e `public`. Quando isto acontece, o seu método ou atributo fica num estado de visibilidade intermediário entre o `private` e o `public`, que veremos mais pra frente, no capítulo de pacotes.

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!

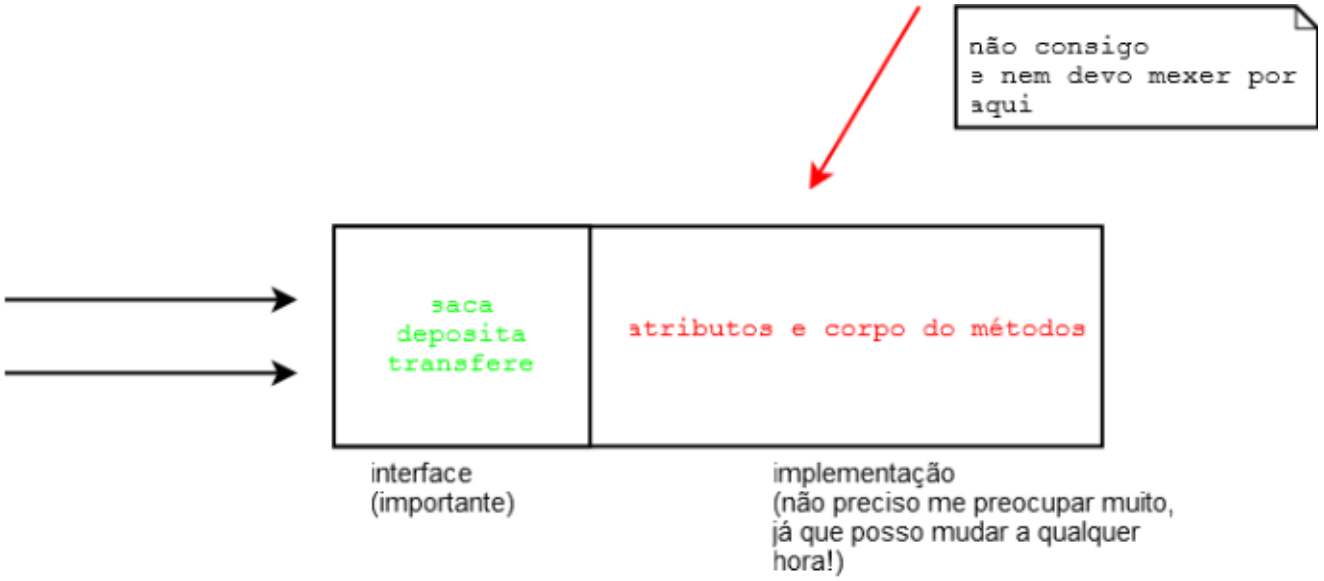
Melhor ainda! O dia em que precisarmos mudar como é realizado um saque na nossa classe `Conta`, adivinhe onde precisaríamos modificar? Apenas no método `saca`, o que faz pleno sentido. Como exemplo, imagine cobrar CPMF de cada saque:

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.

6.2 - Encapsulamento

O que começamos a ver nesse capítulo é a ideia de **encapsular**, isto 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)



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.

Programando voltado para a interface e não para a implementação

É sempre bom programar pensando na interface da sua classe, como seus usuários a estarão utilizando, e não somente em como ela vai funcionar.

A implementação em si, o conteúdo dos métodos, não tem tanta

importância para o usuário dessa classe, uma vez que ele só precisa saber o que cada método pretende fazer, e não como ele faz, pois isto pode mudar com o tempo.

Essa frase vem do livro Design Patterns, de Eric Gamma et al. Um livro cultuado no meio da orientação a objetos.

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

- 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).

- Todos os celulares fazem a mesma coisa (interface), eles possuem maneiras (métodos) de discar, ligar, desligar, atender, etc. O que muda é como eles fazem (implementação), mas repare que para o usuário comum pouco importa se o celular é GSM ou CDMA, isso fica encapsulado na implementação (que aqui são os circuitos).

Já temos conhecimentos suficientes para resolver aquele problema da validação de 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
  }

  // ..
}
```

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**. 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!

6.3 - 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:

```
class Conta {

  private double saldo;

  // outros atributos omitidos
```

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

  // deposita() saca() e transfere() omitidos
}
```

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.pegaSaldo());
   }
}
```

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 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;
  private double limite;
  private Cliente titular;

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

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

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

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

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

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

```
  }
}
```

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`? Repare:

```
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;
  }
}
```

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.

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.

> **Cuidado com os getters e setters!**
>
> Como já dito, não devemos criar getters e setters sem um motivo explicito. No blog da Caelum há um artigo que ilustra bem esses casos:
>
> http://blog.caelum.com.br/2006/09/14/nao-aprender-oo-getters-e-setters/

6.4 - 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 fizermos:

```
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.

O construtor default

Até agora, as nossas classes não possuíam nenhum construtor. Então como é que era possível dar `new`, se todo `new` chama um construtor **obrigatoriamente**?

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 que você declara um construtor, o construtor default não é mais fornecido.

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

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

  // construtor
  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 carlos = new Cliente();
carlos.nome = "Carlos";

Conta c = new Conta(carlos);
System.out.println(c.titular.nome);
```

6.5 - A NECESSIDADE DE UM CONSTRUTOR

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.

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.

Construtor: um método especial?

Um construtor não é um método. Algumas pessoas o chamam de um método especial, mas definitivamente não é, já que não possui retorno e só é chamado durante a construção do objeto.

Chamando outro construtor

Um construtor só pode rodar durante a construção do objeto, isto é, você nunca conseguirá chamar o construtor em um objeto já construído. Porém, durante a construção de um objeto, você pode fazer com que um construtor chame outro, para não ter de ficar copiando e colando:

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

  // construtor
  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;
  }
```

```
    //..
}
```

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.

Java Bean

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 (mas não confunda com EJB, que é Enterprise Java Beans).

6.6 - 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;
  }
}
```

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;
```

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;
  }
}
```

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();
```

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.

Métodos e atributos estáticos

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.

O `static` realmente traz um "cheiro" procedural, porém em muitas vezes é necessário.

6.7 - Um pouco mais...

1. Em algumas empresas, o UML é amplamente utilizado. Às vezes, o programador recebe o UML já pronto, completo, e só deve preencher a implementação, devendo seguir à risca o UML. O que você acha dessa prática? Quais as vantagens e desvantagens.

2. Se uma classe só tem atributos e métodos estáticos, que conclusões podemos tirar? O que lhe parece um método estático em casos como esses?

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

6.8 - 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`. Por exemplo:

```
class Funcionario {
  private double salario;

  // ...

  public double getSalario() {
    return this.salario;
  }

  public void setSalario(double salario) {
    this.salario = salario;
  }
}
```

Não copie e cole! Aproveite para praticar sintaxe. Logo passaremos a usar o Eclipse e aí sim teremos procedimentos mais simples para este tipo de tarefa.

Repare que o método `calculaGanhoAnual` parece também um getter. Aliás, seria comum alguém nomeá-lo de `getGanhoAnual`. Getters não precisam apenas retornar atributos. Eles podem trabalhar com esses dados.

3. Modifique suas classes que acessam e modificam atributos de um `Funcionario` para utilizar os getters e setters recém criados.

Por exemplo, onde você encontra:

```
f.salario = 100;
System.out.println(f.salario);
```

passa para:

```
f.setSalario(100);
System.out.println(f.getSalario());
```

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`.

Seria algo como:

```
class Funcionario {
  public Funcionario() {
    // construtor sem argumentos
  }

  public Funcionario(String nome) {
    // construtor que recebe o nome
  }
}
```

Por que você precisa do construtor sem argumentos para que a passagem do nome seja opcional?

5. (opcional) 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.

Crie um getter para o identificador. Devemos ter um setter?

6. (opcional) 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.

Por exemplo, na classe `Empresa`, seria interessante ter um setter e getter para a sua array de funcionários? Não seria mais interessante ter um método como este?

```
class Empresa {
  // ...

  public Funcionario getFuncionario (int posicao) {
    return this.empregados[posicao];
  }
}
```

7. (opcional) Na classe `Empresa`, em vez de criar um array de tamanho fixo, receba como parâmetro no construtor o tamanho do array de `Funcionario`.

Com esse construtor, o que acontece se tentarmos dar `new Empresa()` sem passar argumento algum? Por quê?

8. (opcional) Como garantir que datas como 31/2/2012 não sejam aceitas pela sua classe `Data`?

9. (opcional) 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)...`)

6.9 - Desafios

1. Porque esse código não compila?

```
class Teste {
  int x = 37;
  public static void main(String [] args) {
    System.out.println(x);
  }
}
```

2. Imagine que tenha uma classe `FabricaDeCarro` e quero garantir que só existe um objeto desse tipo em toda a memória. Não existe uma palavra chave especial para isto em Java, então teremos de fazer nossa classe de tal maneira que ela respeite essa nossa necessidade. Como fazer isso? (pesquise: singleton design pattern)

CAPÍTULO ANTERIOR:

Um pouco de arrays

PRÓXIMO CAPÍTULO:

Herança, reescrita e polimorfismo

Você encontra a Caelum também em:

Blog Caelum

Cursos Online

Facebook

Newsletter

Casa do Código

Twitter