Programação e Orientação a Objetos

Leonardo Vitazik Neto

Católica de Santa Catarina
Centro Universitário

Agenda

- MODIFICADORES DE ACESSO;
- ENCAPSULAMENTO;
- GETTERS E SETTERS;
- CONSTRUTORES;
- ATRIBUTOS DE CLASSE;
- PACOTES;
- IMPORT;
- Exercício.

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 saldo seja insuficiente. A seguir você pode lembrar como está a classe Conta :

```
class Conta {
    String titular;
    int numero;
    double saldo;

    // ..

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

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

```
class TestaContaEstouro1 {
    public static void main(String[] args) {
        Conta minhaConta = new Conta();
        minhaConta.saldo = 1000.0;
        minhaConta.saca(50000); // saldo é só 1000!!
    }
}
```

CONTROLANDO O ACESSO

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 de 0. 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 faz isso diretamente:

```
class TestaContaEstouro2 {
    public static void main(String[] args) {
        Conta minhaConta = new Conta();
        minhaConta.saldo = -200; //saldo está abaixo de 0
    }
}
```

CONTROLANDO O ACESSO

Como evitar isso? Uma ideia simples seria testar se não estamos sacando um valor maior que o saldo toda vez que formos alterá-lo:

```
class TestaContaEstouro3 {

    public static void main(String[] args) {
        // a Conta
        Conta minhaConta = new Conta();
        minhaConta.saldo = 100;

        // quero mudar o saldo para -200
        double novoSaldo = -200;

        // testa se o novoSaldo é válido
        if (novoSaldo < 0) { //
            System.out.println("Não posso mudar para esse saldo");
        } else {
            minhaConta.saldo = novoSaldo;
        }
    }
}
```

CONTROLANDO O 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 :

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

CONTROLANDO O ACESSO

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.

CONTROLANDO O ACESSO

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 uma verificação para o valor do saque. 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).

CONTROLANDO O ACESSO

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.

CONTROLANDO O 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 valor) {
        //posso sacar até saldo
        if (valor > this.saldo){
            System.out.println("Não posso sacar um valor maior que o saldo!");
        } else {
            this.saldo = this.saldo - valor;
        }
    }
}
```

CONTROLANDO O 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

O que começamos a ver 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.

Programe para interface

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 efetuar uma ligação pouco importa se o celular é iPhone ou Android, 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!

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() e saca() omitidos
}
```

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.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.

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 String titular;
    private double saldo;

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

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

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

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

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.

GETTERS E SETTERS

Podemos encapsular isso em um método e, porque não, dentro do próprio getSaldo ? Repare:

```
class Conta {

    private String titular;
    private double saldo;
    private double limite; // adicionando um limite a conta

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

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

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

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

Encapsulamento

O código anterior 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 saldo menor que 0? 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 {
    String titular;
    int numero;
    double saldo;

    // 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.

CONSTRUTORES

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

```
class Conta {
    String titular;
    int numero;
    double saldo;

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

    // ..
}
```

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

```
String carlos = "Carlos";
Conta c = new Conta(carlos);
System.out.println(c.titular);
```

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.

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 {
    String titular;
    int numero;
    double saldo;

    // construtor
    Conta (String titular) {
        //     faz mais uma série de inicializações e configurações
        this.titular = titular;
    }

    Conta (int numero, String 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.

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;
```

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.

ATRIBUTOS DE CLASSE

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();
```

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;
}
```

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.

PACOTES

Quando um programador utiliza as classes feitas por outro, surge um problema clássico: como escrever duas classes com o mesmo nome?

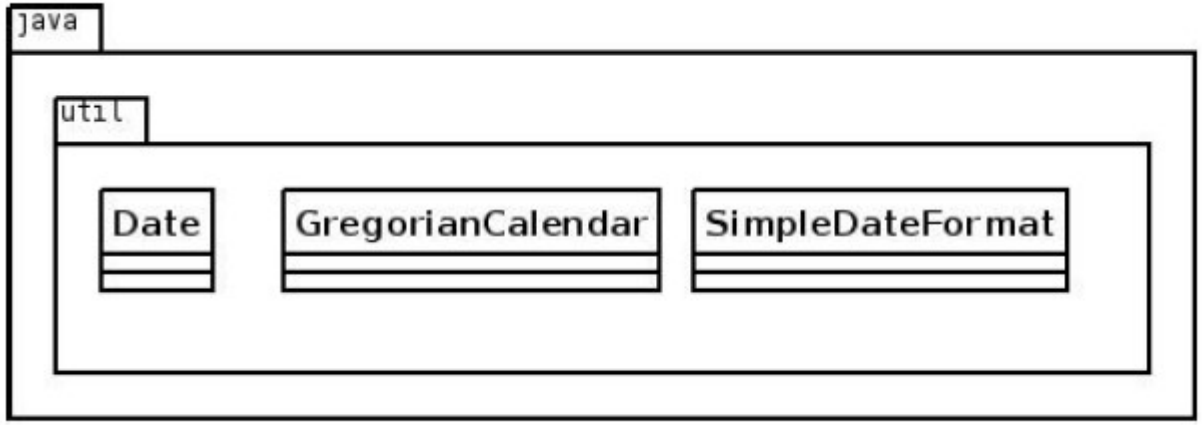
Por exemplo: pode ser que a minha classe de Data funcione de um certo jeito, e a classe Data de um colega, de outro jeito. Pode ser que a classe de Data de uma biblioteca funcione ainda de uma terceira maneira diferente.
Como permitir que tudo isso realmente funcione? Como controlar quem quer usar qual classe de Data?

Pensando um pouco mais, notamos a existência de um outro problema e da própria solução: o sistema operacional não permite a existência de dois arquivos com o mesmo nome sob o mesmo diretório, portanto precisamos organizar nossas classes em diretórios diferentes.

PACOTES

Os diretórios estão diretamente relacionados aos chamados pacotes e costumam agrupar classes de funcionalidades similares ou relacionadas.

Por exemplo, no pacote java.util temos as classes Date , SimpleDateFormat e GregorianCalendar ; todas elas trabalham com datas de formas diferentes.



PACOTES

Se a classe Cliente está no pacote contas , ela deverá estar no diretório com o mesmo nome:
contas . Se ela se localiza no pacote br.org.catolicasc.banco.contas, significa que está no diretório br/org/catolicasc/banco/contas.

A classe Cliente , que se localiza nesse último diretório mencionado, deve ser escrita da seguinte forma:

```
package br.org.catolicasc.banco.contas;

public class Cliente {
    String nome;
    String sobrenome;
    String cpf;
}
```

Fica fácil notar que a palavra chave package indica qual o pacote/diretório contém esta classe.
Um pacote pode conter nenhum ou mais subpacotes e/ou classes dentro dele.

PADRÃO DA NOMENCLATURA DOS PACOTES

O padrão da sun para dar nome aos pacotes é relativo ao nome da empresa que desenvolveu a classe:

br.com.nomedaempresa.nomedoprojeto.subpacote
br.com.nomedaempresa.nomedoprojeto.subpacote2
br.com.nomedaempresa.nomedoprojeto.subpacote2.subpacote3

Os pacotes só possuem letras minúsculas, não importa quantas palavras estejam contidas nele. Esse padrão existe para evitar ao máximo o conflito de pacotes de empresas diferentes.

As classes do pacote padrão de bibliotecas não seguem essa nomenclatura, que foi dada para bibliotecas de terceiros.

IMPORT

Para usar uma classe do mesmo pacote, basta fazer referência a ela como foi feito até agora simplesmente escrevendo o próprio nome da classe. Se quisermos que a classe Banco fique dentro do pacote br.org.catolicasc.banco.contas, ela deve ser declarada assim:

```
package br.org.catolicasc.banco.contas;

public class Banco {
    String nome;
}
```

Para a classe Cliente ficar no mesmo pacote, seguimos a mesma fórmula:

```
package br.org.catolicasc.banco.contas;

public class Cliente {
    String nome;
    String sobrenome;
    String cpf;
}
```

IMPORT

A novidade chega ao tentar utilizar a classe Banco (ou Cliente) em uma outra classe que esteja fora desse pacote, por exemplo, no pacote br.org.catolicasc.banco.main :

```
package br.org.catolicasc.banco.contas.main;

public class TesteDoBanco {

    public static void main(String[] args) {
        br.org.catolicasc.banco.contas.Banco banco = new br.org.catolicasc.banco.contas.Banco();
        banco.setNome("Bradesco");
        System.out.println(banco.getNome());
    }
}
```

IMPORT

Repare que precisamos referenciar a classe Banco com todo o nome do pacote na sua frente. Esse é o conhecido Fully Qualified Name de uma classe. Em outras palavras, esse é o verdadeiro nome de uma classe, por isso duas classes com o mesmo nome em pacotes diferentes não conflitam.
Mesmo assim, ao tentar compilar a classe anterior, surge um erro reclamando que a classe Banco não está visível.

Acontece que as classes só são visíveis para outras no mesmo pacote e, para permitir que a classe TesteDoBanco veja e acesse a classe Banco em outro pacote, precisamos alterar essa última e transformá-la em pública:

```
public class Banco {
    String nome;
}
```

IMPORT

A palavra chave public libera o acesso para classes de outros pacotes. Do mesmo jeito que o compilador reclamou que a classe não estava visível, ele reclama que o atributo/variável membro também não está. É fácil deduzir como resolver o problema: utilizando novamente o modificador public :

```
package br.org.catolicasc.banco.contas;

public class Banco {
    String nome;
}
```

IMPORT

Podemos testar nosso exemplo anterior, lembrando que utilizar atributos como público não traz encapsulamento e está aqui como ilustração.
Voltando ao código do TesteDoBanco , é necessário escrever todo o pacote para identificar qual classe queremos usar? O exemplo que usamos ficou bem complicado de ler:

INCLUIR AQUI O MESMO EXEMPLO COM PACKAFULL

IMPORT

Existe uma maneira mais simples de se referenciar a classe Banco : basta importá-la do pacote

```
package br.org.catolicasc.banco.contas.main;

import br.org.catolicasc.banco.contas.Banco;

public class TesteDoBanco {

    public static void main(String[] args) {
        Banco banco = new Banco();
        banco.setNome("Bradesco");
        System.out.println(banco.getNome());
    }
}
```

Isso faz com que não precisemos nos referenciar utilizando o fully qualified name, podendo utilizar Banco dentro do nosso código em vez de escrever o longo br.org.catolicasc.banco.contas.Banco .

PACKAGE, IMPORT, CLASS

É muito importante manter a ordem! Primeiro, aparece uma (ou nenhuma) vez o package; depois, pode aparecer um ou mais imports; e, por último, as declarações de classes.

IMPORT X.Y.Z.*;

É possível "importar um pacote inteiro" (todas as classes do pacote, exceto os subpacotes) através do coringa * :

```
import java.util.*;
```

Importar todas as classes de um pacote não implica em perda de performance em tempo de execução, mas pode trazer problemas com classes de mesmo nome! Além disso, importar de um em um é considerado boa prática, pois facilita a leitura para outros programadores. Uma IDE como o Eclipse já vai fazer isso por você, assim como a organização em diretórios.

ACESSO AOS ATRIBUTOS, CONSTRUTORES E MÉTODOS

Os modificadores de acesso existentes em Java são quatro, e até o momento já vimos três, mas só explicamos dois.

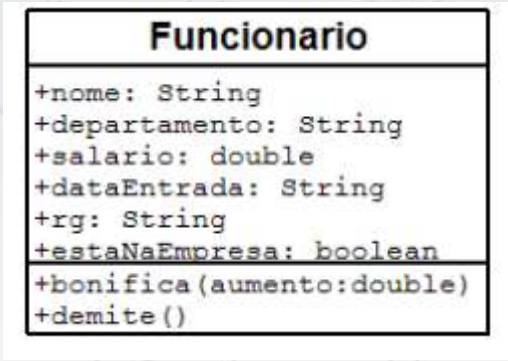
public - Todos podem acessar aquilo que for definido como public . Classes, atributos, construtores e métodos podem ser public .

protected - Aquilo que é protected pode ser acessado por todas as classes do mesmo pacote e por todas as classes que o estendam, mesmo que essas não estejam no mesmo pacote. Somente atributos, construtores e métodos podem ser protected .

padrão (sem nenhum modificador) - Se nenhum modificador for utilizado, todas as classes do mesmo pacote têm acesso ao atributo, construtor, método ou classe.

private - A única classe capaz de acessar os atributos, construtores e métodos privados é a própria classe. Classes, como conhecemos, não podem ser private, mas atributos, construtores e métodos sim.

EXERCÍCIOS



1) Crie uma classe funcionário conforme o diagrama ao lado;
2) 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.
3) Crie os getters e setters necessários da sua classe Funcionario;
4) 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());
```

5) Faça com que sua classe Funcionario possa receber, opcionalmente, o nome do Funcionario durante a criação do objeto. Utilize construtores para obter esse resultado.

6) 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;

7) 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).