Inatel
Instituto Nacional de Telecomunicações

Programação Orientada a Objetos

Cap.5 - Modificadores de Acesso e Atributos de Classe

Prof. Me. Renzo P. Mesquita

Objetivos

1. Controlar o acesso aos nossos Métodos e Atributos por meio dos modificadores public e private;
2. Escrever métodos de acesso do tipo getters e setters;
3. Escrever Construtores para as Classes;
4. Utilizar Variáveis e Métodos estáticos;

Capítulo 5
Modificadores de Acesso e Atributos de Classe

5.1. Controlando o acesso às Classes;
5.2. Encapsulamento de Dados;
5.3. Getters e Setters;
5.4. Construtores;
5.5. Atributos de Classe;

5.1. Controlando o acesso às Classes

Lembra da nossa Classe Conta? Ela tinha um método que permitia sacar dinheiro de um determinado Objeto Conta, não tinha? Vamos relembrar.

```
public class Conta {
    int numero;
    Cliente titular;
    float saldo;
    float limite;

    public void saca(float quantia)
    {
        saldo = saldo - quantia;
    }
    // resto da classe..
}
```

Se na nossa Classe Principal (ou seja, que possui o método main) tivéssemos um Objeto Conta e resolvêssemos sacar 1.000.000 dele sem que ele possuísse saldo suficiente, seria possível?

Em caso afirmativo, o que poderíamos fazer para resolver este problema?

5.1. Controlando o acesso às Classes

No caso anterior, podemos simplesmente incluir um if dentro do nosso método para resolver o problema.

Mas e se o usuário resolver utilizar não o método saca(), mas acessar diretamente o seu atributo saldo para mudá-lo, seria possível?

Sem dúvidas! E isso poderia trazer muitos problemas.

A melhor forma de resolver isso seria forçar quem usa a Classe Conta a invocar o método saca() para alterar um saldo e não permitir o acesso direto ao atributo.

Para fazer isso no Java, basta declarar que os atributos não podem ser acessados de fora da Classe. Fazemos isso utilizando da palavra chave *private* .

Exemplo:

```
private float saldo;
private float limite;
// outros atributos...
```

5.1. Controlando o acesso às Classes

" 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, por exemplo, não compile.

```
Conta minhaConta = new Conta();
// OPS! Impossível acessar,
// pois o atributo é privado
minhaConta.saldo = 1000000;
```

Na Orientação a Objetos, é prática quase que obrigatória proteger nossos atributos com *private*.

5.1. Controlando o acesso às Classes

Cada Classe é responsável por controlar o acesso a seus atributos e este acesso não deve ser controlado por quem está usando a Classe, e sim, por ela mesma.

Repare que, quem invoca o método saca() não faz a menor idéia 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 ele faz.

Exemplo:

```
public void saca(float quantia)
{
    if((saldo - quantia + limite) >= 0)
    {
        saldo = saldo - quantia;
        System.out.println("Saque realizado com Sucesso! Novo Saldo:"+saldo);
    }
    else
    {
        System.out.println("Saque não realizado! Saldo insuficiente! :(");
    }
}
```

5.1. Controlando o acesso às Classes

A palavra chave private também pode ser usada para modificar o acesso a um método.

Tal funcionalidade por ser utilizada, por exemplo, quando existir um método que serve apenas para auxiliar a própria Classe internamente, e não pode ser visto externamente.

Exemplo:

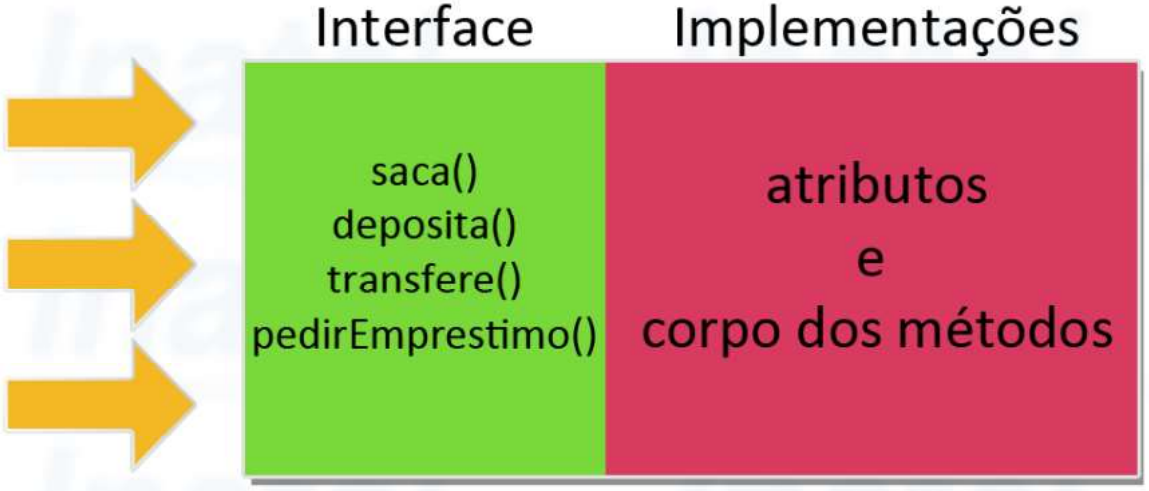
```
public class Conta {
    private void calculaImpostos(float valor){...6 linhas }
    private void calculaJuros(float valor){...7 linhas }
    private boolean verificaSerasa(Cliente cli){...9 linhas }

    public void pedirEmprestimo(float valor, Cliente cli)
    {
        calculaImpostos(valor);
        calculaJuros(valor);
        verificaSerasa(cli);
        // resto da lógica do método...
    }
}
```

5.2. Encapsulamento de Dados

O que vimos até agora foi a idéia de ENCAPSULAMENTO de dados, ou seja, esconder todos os membros de uma classe (atributos), além de esconder como funcionam suas rotinas (métodos).

O conjunto de métodos públicos de uma Classe é também chamado de interface da classe, pois esta é a única maneira no qual podemos comunicar com objetos de uma Classe.



Vamos ver algumas analogias de interfaces no mundo real?

5.2. Encapsulamento de Dados

Exemplos de analogias de interfaces no mundo real:

- Quando dirigimos um carro, o que importa são os pedais e o volante (interface) e não o motor que o carro está usando (implementação). Claro que um motor diferente pode oferecer melhores resultados, mas o que ele faz é a mesma coisa que um motor menos potente faz;
- Todos os celulares fazem a mesma coisa, eles possuem maneiras de discar, ligar, desligar, atender etc. (interface). O que muda é como eles fazem (implementação);

5.3. Getters e Setters

O modificador private faz com que ninguém consiga modificar, nem mesmo ler, um atributo qualquer de uma classe externamente. Mas, teria alguma forma de prover isso de uma forma segura?

Para permitir o acesso aos atributos de maneira controlada, a prática mais comum é criar dois métodos, um que retorna o valor e outro que muda o valor, para cada uma das variáveis do tipo private que se deseja visualizar/modificar externamente.

Estes métodos são chamados de GETTERS e SETTERS.

Exemplo:

```
public float getSaldo() {
    return saldo;
}
```

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

A convenção para o nome desses métodos é de colocar a palavra get ou set antes do nome do atributo.

5.3. Getters e Setters

É uma má prática criar uma Classe e, logo em seguida, criar Getters e Setters para todos os atributos.

Devemos criar Getters e Setters apenas quando houver necessidade. No nosso exemplo anterior, por exemplo, não seria interessante existir o método setSaldo() do jeito que está.

Lembrando que Getters e Setters também podem ser modificados, a fim de dar a impressão ao usuário que ele esteja acessando a variável diretamente mas na verdade ela está sendo tratada dentro dos seus métodos.

5.3. Getters e Setters

Exercício 1 - Poligonos

Polígonos Regulares são aqueles que possuem todos os seus lados e ângulos idênticos. Para obtermos a área de qualquer polígono regular, basta realizarmos os seguintes cálculos:

1. Área (A)
*A = (P*a)/2;*

2. Perímetro (P)
*P = c * n;*

3. Apótema (a)
a = c / 2tan(180/n);

c - comprimento dos lados; *n - número de lados do Polígono;*

Baseado nas fórmulas acima e no diagrama ao lado, faça um programa que peça ao usuário para entrar com o número e comprimento dos lados de um Polígono Regular para que assim possa realizar o cálculo de sua área.

PoligonoRegular
-n: int -c: float -areaPoligono: float
-calculaPerimetro(): float -calculaApotema(): float +calculaArea(nLados: int, cLados: float) +getAreaPoligono(): float

...

5.4. Construtores

Quando usamos a palavra chave 'new', estamos criando um Objeto. Sempre quando o 'new' é chamado, ele executa o construtor da Classe.

Exemplo:

```
public class Conta {

    private int numero;
    private Cliente titular;
    private float saldo;
    private float limite;

    Conta()
    {
        System.out.println("Construindo uma conta..");
    }
    // ...resto da Classe
}
```

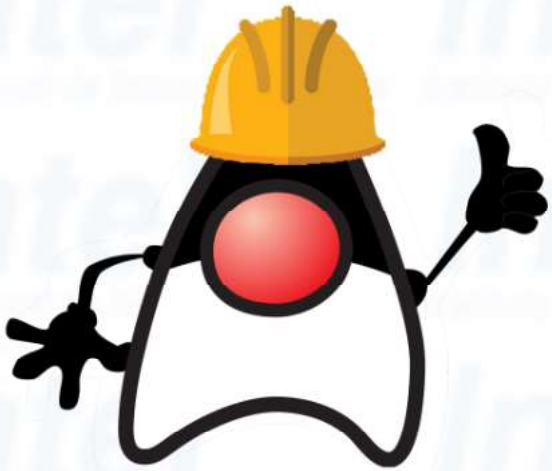
Ou seja, quando fizermos *Conta c = new Conta()* a mensagem "Construindo uma conta.." aparecerá neste caso!

5.4. Construtores

Um Construtor possui o mesmo nome da Classe, e pode parecer, mas não é um método, pois não tem retorno. Ele funciona como uma rotina de inicialização sempre que um novo Objeto é criado.

Até agora, as nossas classes não possuíam nenhum Construtor. Então como era possível dar "new", se todo "new" chama obrigatoriamente um Construtor?

- Quando não criamos um Construtor, o Java cria um automaticamente. Ele é chamado de Construtor Default, e possui o seu corpo vazio;
- A partir do momento que declaramos um Construtor, o Construtor default não é mais fornecido;



5.4. Construtores

Outro detalhe bem interessante dos Construtores é que eles podem receber argumentos, podendo assim, inicializar algum tipo de informação.

Exemplo:

```
public class Conta {

    private int numero;
    private Cliente titular;
    private float saldo;
    private float limite;

    Conta(Cliente cli)
    {
        titular = cli;
    }
    // ...resto da Classe
}
```

E na Classe principal, por exemplo..

```
Cliente novoCliente = new Cliente();
novoCliente.nome = "Duke da Silva Java";

Conta novaConta = new Conta(novoCliente);
```

Nesse caso, o construtor OBRIGA o usuário a passar argumentos para o objeto durante seu processo de criação.

5.4. Construtores

Outro ponto importante, é que uma Classe pode possuir mais de um Construtor. Mas como ela sabe qual Construtor deve executar na hora do "new"?

Simplesmente pelo número e tipo dos parâmetros que são passados em cada um dos Construtores de uma Classe.

Exemplo: Classe Conta com três Construtores.

```
Conta()
{
    System.out.println("Construtor sem parâmetro!");
}

Conta(Cliente cli)
{
    System.out.println("Construtor com um parâmetro e do tipo Cliente!");
}

Conta(int numero, int saldo)
{
    System.out.println("Construtor com dois parâmetros do tipo int!");
}
```

5.5. Atributos de Classe

Imagine que queremos controlar a quantidade de Contas existentes no nosso sistema. Como poderíamos fazer isto?

Ah, essa é fácil! Poderíamos fazer um contador.

Exemplo:

```
int totalDeContas = 0;

Conta novaConta = new Conta();
totalDeContas = totalDeContas + 1;
```

Ou seja, toda vez que uma Conta for criada, incrementamos a variável totalDeContas.

E quem garante que vamos conseguir lembrar de incrementar a variável totalDeContas toda vez que um Objeto for criado?

E se por algum motivo a variável totalDeContas não puder ser vista por outros Objetos? Como poderemos incrementá-la?

Existe uma solução mais adequada para este problema?

5.5. Atributos de Classe

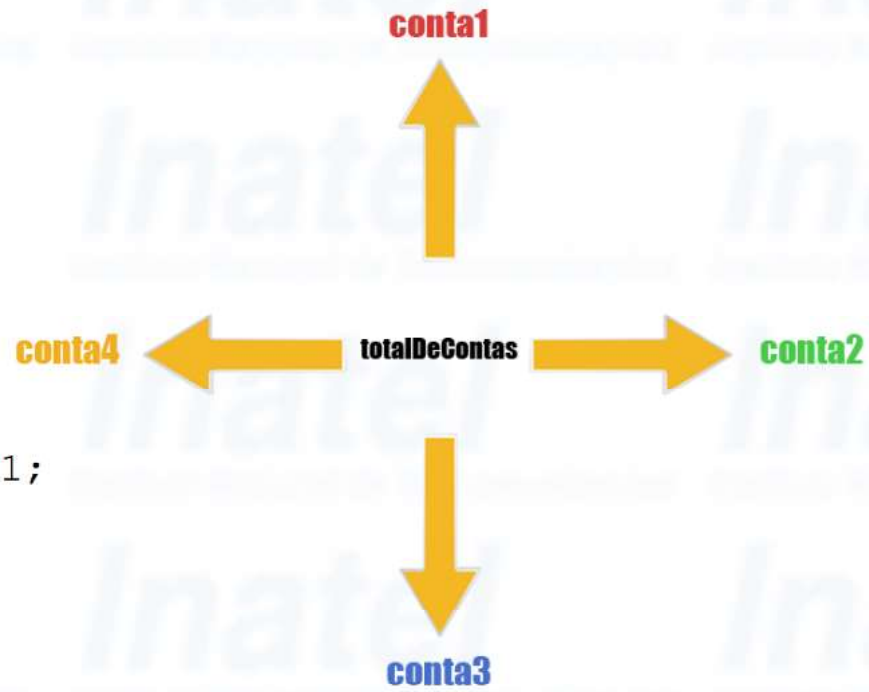
Seria interessante que a variável anterior fosse única, compartilhada por TODOS os objeto de uma mesma Classe.

Para fazermos isto, declaramos uma variável de uma Classe como *static*.

Exemplo:

```
public class Conta {
    private static int totalDeContas;

    Conta()
    {
        totalDeContas = totalDeContas + 1;
    }
    // ...resto da Classe
}
```



Dessa forma, resolveríamos de forma elegante o problema anterior!

Vale ressaltar que quando declaramos um atributo como static, ele passa a não ser mais um atributo de cada objeto, e sim um ATRIBUTO DA CLASSE.

5.5. Atributos de Classe

Para acessarmos um atributo estático, não precisamos utilizar da variável que aponta para o Objeto, basta colocarmos NomeDaClasse.NomeDaVariável.

Exemplo:

```
System.out.println("Total de Contas: "+Conta.totalDeContas);
```

E se o atributo for privado e estático ao mesmo tempo?

Ele deverá ter um método get para retornar o seu valor e este método também deverá ser estático.

Exemplo:

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

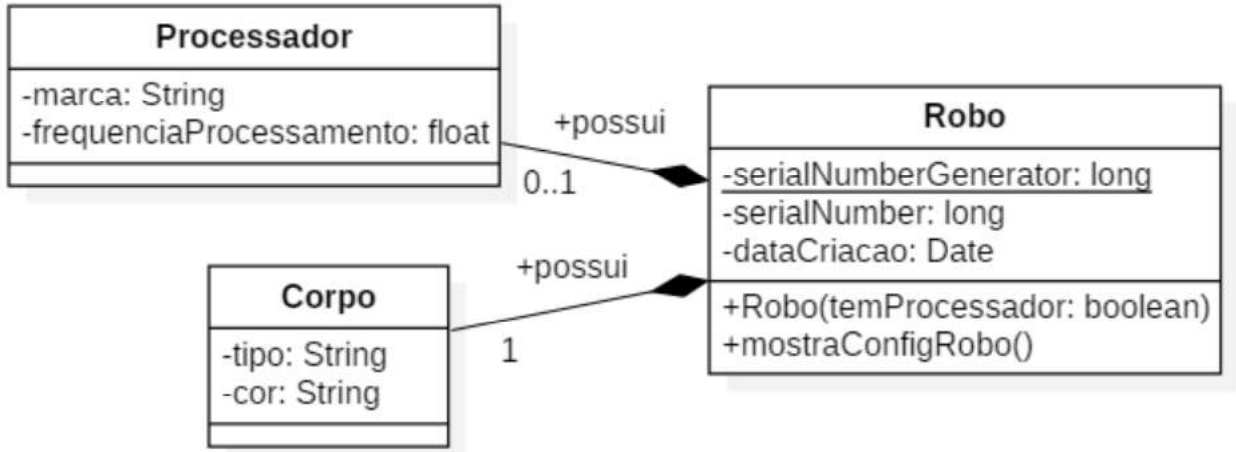
e acessando este método de fora de Classe...

```
System.out.println("Total de Contas: "+Conta.getTotalDeContas());
```

5.5. Atributos de Classe

Exercício 2 - FabricaDeRobos

Crie Classes em Java que atendam a seguinte especificação:



No final, crie pelo menos um Robô e mostre suas informações.

Dicas:

- Vale ressaltar que, toda vez que um Robô for criado, eles já devem automaticamente receber um serialNumber único;
- Robôs podem ou não possuírem um Processador;
- As marcas e frequências de Processadores disponíveis são: Intel, AMD, 500 e 750 (Mhz);
- Os Corpos dos Robôs podem ser dos seguintes tipos: com Pernas, Rodas, Esteira ou Esférico;
- Como pegar a data e hora do sistema? Ex: Date dataCriacao = new Date();

FIM
DO
CAPÍTULO 5

Próximo Capítulo

Herança, Reescrita e Polimorfismo.