INSTITUTO FEDERAL DE
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
BAHIA

Modificadores de Acesso e Atributos de Classe

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**;

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

    // ..

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

Controlando o acesso

- 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!!
    }
}
```

- **Colocamos** 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**.

Controlando o acesso

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA BAHIA

- No entanto 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 **idéia simples** seria **testar** se não estamos **ultrapassando** o **limite** toda vez que formos **alterar o saldo**

Controlando o acesso


```
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;
        }
    }
}
```

- 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**.

Controlando o acesso

- 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**).

Controlando o acesso

- Marcando um **atributo** como **privado**, **fechamos** o **acesso** ao mesmo em **relação** a **todas** as outras **classes**;

- Esse código compila?

```
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;
   }
}
```

- Na **orientação a objetos**, é **prática quase** que **obrigatória proteger** seus **atributos** com **private**.

Controlando o acesso

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

- Tal f**uncionalidade** 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;

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

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**;

- Imaginem se um dia **precisarmos** mudar como é **realizado** um saque na nossa classe Conta, adivinhe onde **precisaríamos modificar**?

Encapsulamento

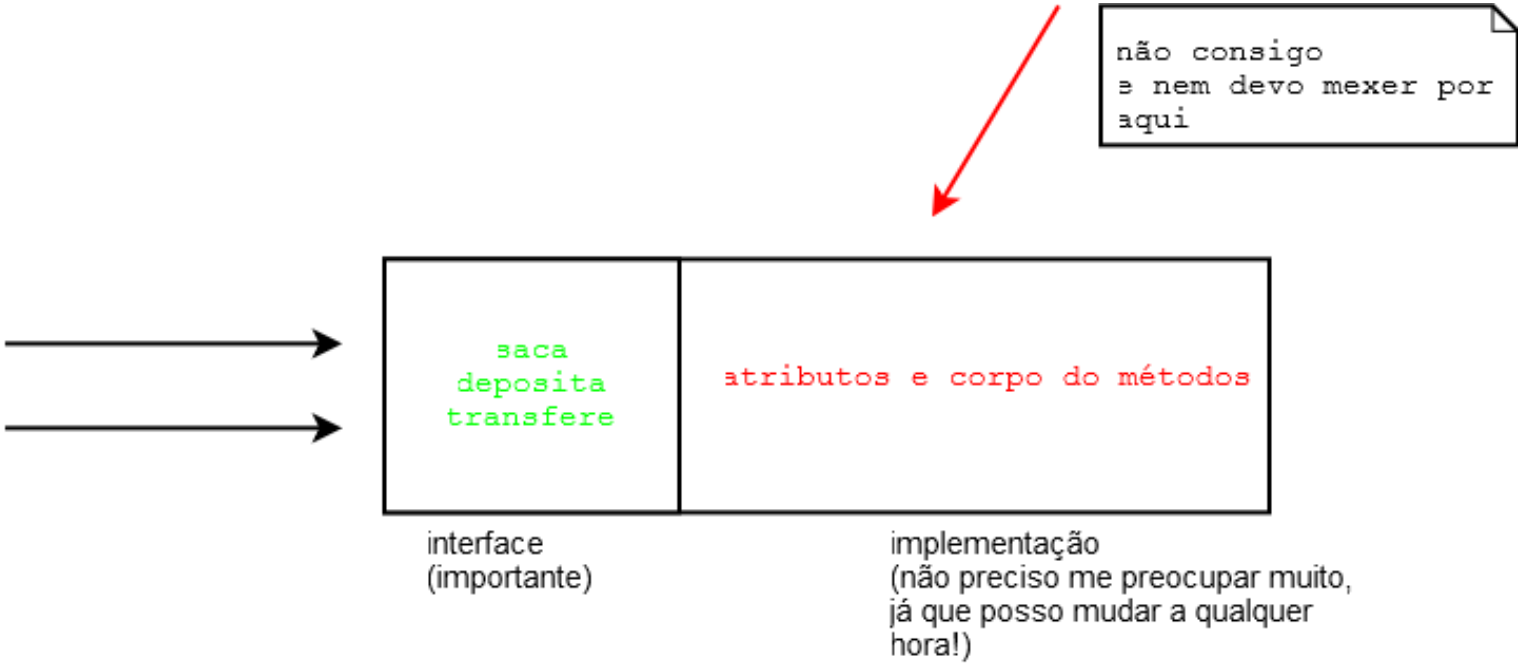
INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA BAHIA

- O que começamos a ver é a idéia de **encapsular**, isto é, **esconder todos os membros de uma classe**, além de esconder como **funcionam** as **rotinas** (no caso **métodos**).

- Encapsular é **fundamental** para que seu sistema seja **sucetível** 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**.

Encapsulamento



- 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**.

Programação para interface

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

- Sempre que vamos **acessar um objeto**, utilizamos sua **interface**.

Programação para interface

- Imaginem que **não queremos** que as pessoas **alterem** o atributo **CPF diretamente** na classe cliente;

```
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 nao 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**

Programação para interface

- E o dia que você não precisar **verificar** o **CPF** de quem tem **mais** de 60 anos?

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

- Podemos perceber que 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

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA BAHIA

- 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**!

Getters e Setters


```
public class Conta {

    private double saldo;

    // outros atributos omitidos

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

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

- Acessando o saldo na conta:

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

Getters e Setters

- 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.

Getters e Setters

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA BAHIA

```
public 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;
    }
}
```

Getters e Setters

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA BAHIA

- 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.

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.");
    }

    // ..
}
```

Construtores

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA BAHIA

- Então, quando fizermos:

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

- A mensagem "construindo uma conta" aparecerá.

- O **construtor** é como uma **rotina de inicialização** que é **chamada** sempre que um **novo objeto** é **criado**.

Construtores

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA BAHIA

- Lembrem-se um construtor pode **parecer**, mas **não é um método.**

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

Construtores

- 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**.

Construtores

```
Cliente carlos = new Cliente();
carlos.nome = "Carlos";

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

- Para que **utilizamos** um **construtor**?

- Dar **possibilidades** ou **obrigar** o usuário de uma classe a **passar argumentos** para o objeto durante o processo de **criação** do mesmo.

- Você **pode ter mais de um construtor** na sua classe e, no momento do **new**, o **construtor apropriado** será **escolhido**.

Construtores

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA BAHIA

- Construtor: um **método especial**?
- Um construtor **não é um método**, já que não possui **retorno** e só é **chamado** durante a **construção** do objeto.
- 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**, para não ter de ficar copiando e colando;

Construtores

```
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;
    }

    //..
}
```

Atributos de classe

- Imaginem que o nosso sistema bancario também quer controlar a **quantidade** de **contas existentes no sistema.**

- Como poderíamos fazer isto?

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

- No entanto, desta maneira estamos **espalhando um código por toda aplicação**, e **quem garante que vamos conseguir lembrar de incrementar a variável totalDeContas toda vez**?

Atributos de classe

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA BAHIA

- O que **acham** desta **alternativa**?

```
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.**

Atributos de classe

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA BAHIA

- O interessante séria 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**,

Atributos de classe

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA BAHIA

```
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.**

Atributos de classe

- 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 o 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

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA BAHIA

- **Métodos** e atributos estáticos só podem acessar outros métodos e atributos estáticos da mesma **classe**;

- Isso 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**.