

Programação Orientada a Objetos

Professor Filipe Dwan Pereira

Aula 5 – Modificadores de Acesso e Atributos de Classe

“Tens visto a um homem que é sábio aos seus próprios olhos? Maior esperança a no tolo do que nele.”

Provérbios 26:12

Disclaimer

- Este slide foi baseado nas seguintes fontes principais:
 - SOFTBLUE. Professor Carlos Eduardo Gusso Tosin. Fundamentos de Java. <http://www.softblue.com.br/>.
 - Slides professor Horácio Oliveira – UFAM.
 - CAELUM. Java e Orientação a Objetos. Disponível em: <https://www.caelum.com.br/apostila-java-orientacao-objetos/>
 - K19. Java e Orientação a Objetos. Disponível em: <http://www.k19.com.br/cursos/orientacao-a-objetos-em-java>.

Objetivo desta aula

- 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.

Controlando o acesso

Um dos problemas mais simples que tínhamos no nosso sistema de contas é que o método `saca` permitia 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;  
    }  
}
```

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 que tenhamos uma conta com estado inconsistente

Apesar do if 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.

```
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:

```
// 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;
}
```

Contudo uma solução mais plausível seria 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;  
    }  
}
```

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

Controle de Acesso

- Cada classe é responsável por controlar os seus atributos.
- Centralizar funcionalidades e facilitar futuras mudanças.
- Basta saber o que o método faz e não como exatamente ele o faz.
- A palavra chave `private` também pode ser usada para modificar o acesso a um método.
- Sempre devemos expor o mínimo possível de funcionalidades

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 modificadores 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.

Modificadores de Acesso: friendly (padrão)

- Se nenhum modificador for utilizado, Java considera o acesso “padrão”.
- Este acesso “padrão”, que não possui palavra-chave reservada, é conhecido como modificador “friendly” ou “package access”.
 - Classes, atributos e métodos com acesso “padrão”:
 - só podem ser acessadas por código do próprio pacote onde elas foram declaradas;
 - se duas classes não pertencem a nenhum pacote mas estão no mesmo diretório, elas são consideradas “friendly” e, portanto, podem acessar suas classes, métodos e atributos “friendly”.

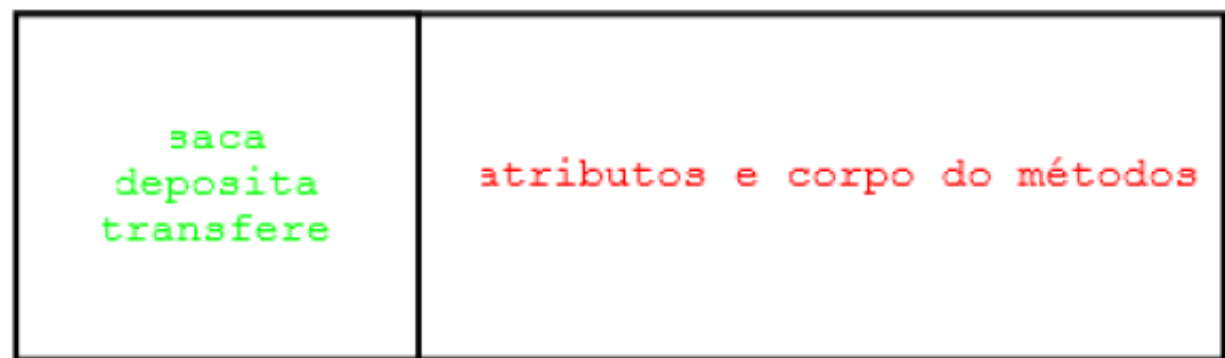
Controle de acesso

- É muito comum que seus atributos sejam *private* e quase todos seus métodos sejam *public* (não é uma regra!).
- Assim, toda conversa de um objeto com outro é feita por troca de mensagens, isto é, acessando seus métodos.

Imagine cobrar CPMF de cada saque

Encapsulamento

- Esconder todos os membros de uma classe;
- Esconder como funcionam as rotinas (no caso métodos) do nosso sistema;
- Encapsular é **fundamental** para que seu sistema seja suscetível a mudanças;



interface
(importante)

implementação
(não preciso me preocupar muito,
já que posso mudar a qualquer
hora!)

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.*

Encapsulamento

- Agrupam um conjunto de classes.
 - grupo de classes relacionadas e, possivelmente, cooperantes.
 - o pacote de uma classe é definido pela palavra-chave *package*.

```
package geometrico;  
class Circulo { ... }
```

- O arquivo que contém a classe (Circulo.java) deve estar no diretório geometrico.
- Padronização: começam com letras minúsculas
- Pacotes podem conter outros pacotes:
 - java.lang
 - java.io
 - company.library.graphic

Pacotes

- As classes que estão no mesmo pacote podem ser utilizadas diretamente no código fonte.

```
package geometrico;  
public class Teste {  
    public static void main(String[] args) {  
        Circulo c = new Circulo();  
    }  
}
```

- Para usar as classes de outros pacotes, deve-se indicar onde elas estão:

```
package exemplo;  
import geometrico.Circulo;  
class Teste {  
    public static void main(String[] args) {  
        Circulo c = new Circulo();  
    }  
}
```

Programando voltado para interface e não para implementação

- 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.

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  
    }  
  
    // ..  
}
```

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;  
}
```

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!

Operador this

- Normalmente não é obrigatório
- Usado em basicamente duas situações:
 - Diferenciar um atributo do objeto de um argumento do método
 - Fornecer a referência do próprio objeto para outro método

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 (**get**) e outro que muda o valor (**set**). Por exemplo: a nossa conta com saldo, limite e titular

```
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;  
    }  
}
```


- Um método getX não necessariamente retorna o valor de um atributo que chama X do objeto em questão.
- 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).

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

Classe Conta encapsulada

```
public 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;  
    }  
}
```

Criação de Objetos

- A seguinte sentença realiza três ações:

```
Circulo c = new Circulo();
```

- Declaração
 - Declarações não criam objetos!
- Instanciação
 - *new* é um operador que cria dinamicamente um novo objeto.
- Inicialização
 - Chamada ao construtor da classe Circulo.

Construtores

- 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.");  
    }  
    // ..  
}
```

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.

Um construtor pode receber um argumento:

```
class Conta {  
    int numero;  
    Cliente titular;  
    double saldo;  
    double limite;  
  
    // construtor  
    Conta(Cliente titular) {  
        this.titular = titular;  
    }  
  
    // ..  
}
```

No Slide Anterior o 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);
```

A necessidade de um Construtor

- 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 um Cliente
- Você pode ter mais de um construtor na sua classe e, no momento do new, o construtor apropriado será escolhido.
- Vale ressaltar, que pode parecer, mas construtor não é um método

Chamando um construtor:

```
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;  
    }  
}
```

Exercício em Sala

- Crie uma classe `Lampada` que possui um atributo `ligada`, o qual indica se a lâmpada está ligada ou desligada.
- Ao construir uma lâmpada, o estado dela (ligada ou desligada) deve ser fornecido. Para ligar e desligar a lâmpada, os métodos `ligar()` e `desligar()` devem ser chamados, respectivamente. Aliás, esta é a única forma de alterar o estado da lâmpada, já que o atributo `ligada` não deve ser visível fora da classe.
- A lâmpada também deve possuir um método `imprimir()`. Quando chamado, ele mostra as mensagens “Lâmpada ligada” ou “Lâmpada desligada”, dependendo do estado atual.
- Construa uma aplicação que cria uma lâmpada ligada, muda o estado dela e também imprime o estado atual após cada chamada aos métodos `ligar()` e `desligar()`.

Duração máxima do exercício: 15 minutos;

Destruição de objetos

- Em Java, não podemos destruir objetos!
- Java utiliza um Coletor de Lixo (Garbage Collector).
 - O coletor se encarrega de se livrar (liberar memória) dos objetos que não são mais necessários no programa.
- Principal vantagem: evita “vazamentos de memória”
 - Memory leak
- Vazamentos de memória podem ocorrer:
 - Por existências de blocos alocados mas inacessíveis (sem ponteiros).
 - Por existências de blocos alocados, com referências, mas desnecessários.

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

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!

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?

Resposta: Vai ser 1. Pois cada uma tem essa variável. **O atributo é de cada objeto.**

Declaração de Atributo como static

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.

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

Para acessarmos um atributo estático, não usamos a palavra chave this, mas sim o nome da classe

Já que o atributo totalDeContas é privado podemos fazer um get para ele:

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

Para saber quantas contas foram criadas:

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


Problema

- Precisamos criar um conta antes de chamar o método...
- Podemos transformar esse método que todo objeto conta tem em um método de toda a classe usando a palavra static de novo.

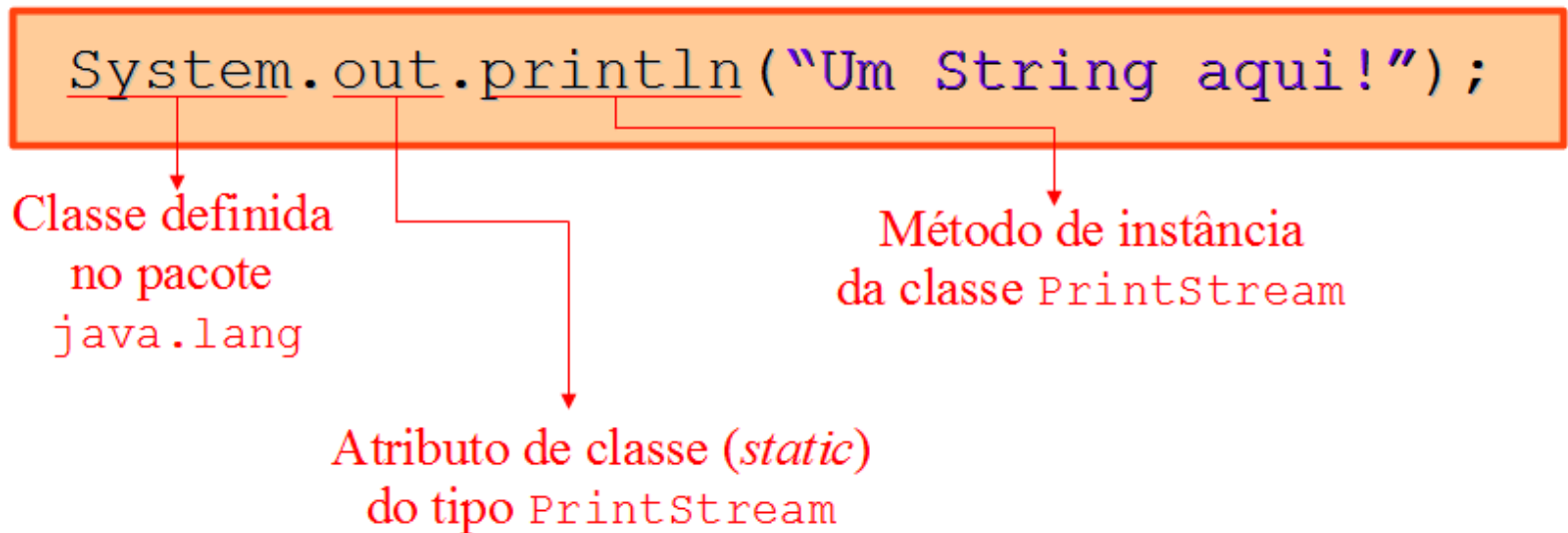
```
public static int getTotalDeContas() {  
    return Conta.totalDeContas;  
}
```

- Para acessar o método:

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

Fim do mistério

- Agora podemos entender melhor a instrução:



- Todas as classes que estão no pacote `java.lang` são automaticamente importadas.

Revisando atributos e métodos estáticos

- Algumas vezes, atributos e/ou métodos podem não estar atrelados a um objeto específico, mas sim à classe
- Atributos ou métodos da classe são assim definidos através do modificador `static`.

Revisando atributos e métodos estáticos

Os valores dos atributos estáticos são compartilhados entre todas as instâncias da classe

```
public class ContaBancaria {  
    private static String banco = "JavaBank";  
  
    private static String getBanco() {  
        return ContaBancaria.banco;  
    }  
}
```

Métodos estáticos só podem acessar atributos ou outros métodos que também sejam estáticos

```
String banco = ContaBancaria.getBanco();
```

O acesso é feito utilizando diretamente a classe. Não é necessário criar um objeto

Outro Exemplo atributos e métodos estáticos

```
public class CriadorDeIds {  
  
    private static int idClasse;  
    private int idObjeto;  
  
    public CriadorDeIds() {  
        this.idObjeto = ++CriadorDeIds.idClasse;  
    }  
  
    public static int getIdClasse() {  
        return CriadorDeIds.idClasse;  
    }  
  
    public int getIdObjeto() {  
        return this.idObjeto;  
    }  
}
```

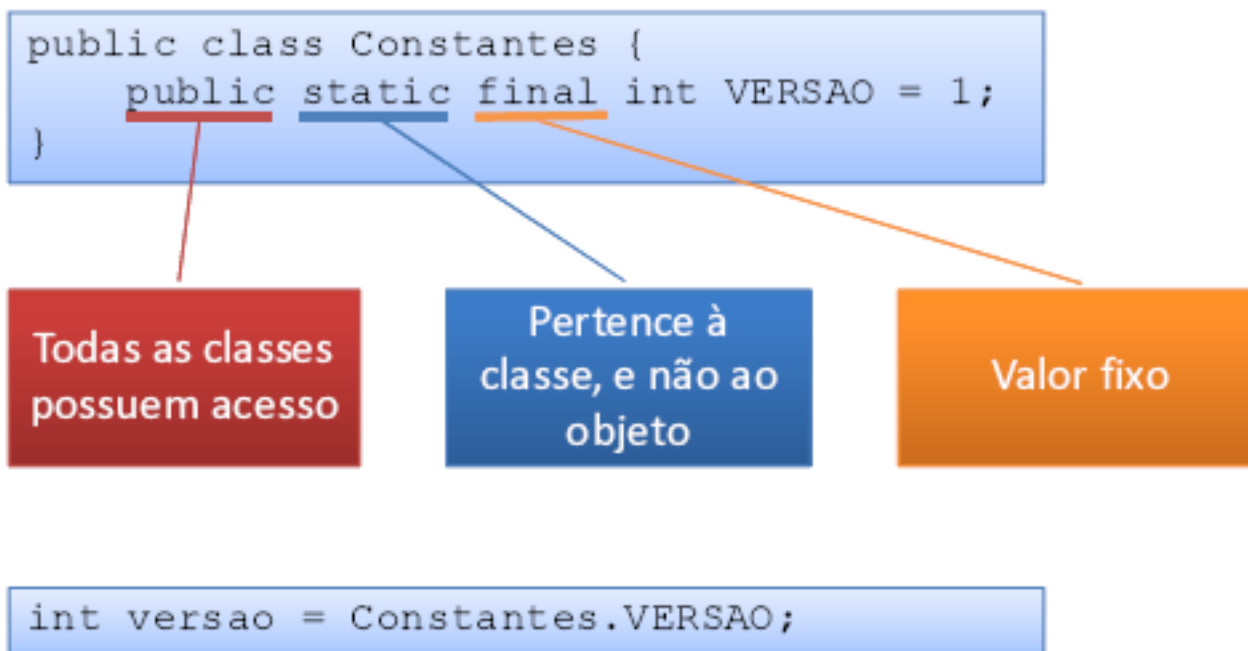
```
public class TestesMod5 {  
  
    public static void main (String[] args) {  
  
        CriadorDeIds obj1 = new CriadorDeIds();  
        CriadorDeIds obj2 = new CriadorDeIds();  
        CriadorDeIds obj3 = new CriadorDeIds();  
  
        System.out.println("Id do obj1: "+obj1.getIdObjeto());  
        System.out.println("Id do obj2: "+obj2.getIdObjeto());  
        System.out.println("Id do obj3: "+obj3.getIdObjeto());  
  
        System.out.println("Id do classe CriadorDeIds: "  
            +CriadorDeIds.getIdClasse());  
    }  
}
```

Saída:

Id do obj1: 1
Id do obj2: 2
Id do obj3: 3
Id do classe CriadorDeIds: 3

Criando constantes...

- Atributos estáticos são uma forma bastante usada para criar constantes no Java



Bloco static

- Uma classe **pode** (não é necessário) ter um (apenas um) bloco **static**.
- O bloco **static** é executado quando a classe é **referenciada** pela primeira vez. É útil para:
 - Inicializar atributos estáticos
 - Executar um código antes que a classe seja utilizada

```
public class MinhaClasse {  
    private static int x;  
  
    static {  
        x = 10;  
        Programa.inicializar();  
    }  
}
```

O bloco só é executado uma vez

Em blocos estáticos só devem ter atributos ou métodos estáticos.

Exemplo Bloco static

```
public class A {  
  
    static {  
        System.out.println("SA");  
    }  
  
    public A() {  
        System.out.println("CA");  
    }  
  
    public static void main(String[] args) {  
        A a;  
    }  
}
```

Saída será: **SA**

Exercícios em sala

- Crie uma classe Data que possui dois construtores. O primeiro recebe um dia, mês e ano. O segundo, além destas informações, recebe também uma hora, minuto e segundo (a hora fornecida deve estar entre 0 e 23). É importante que este segundo construtor invoque o primeiro para evitar a duplicação de código.
- Os construtores devem armazenar os dados fornecidos como parâmetros em atributos privados. Estes atributos devem ter métodos getters associados, que irão expor os valores para códigos externos à classe.
- A classe Data deve ter também um método imprimir() utilizado para imprimir a data e a hora representados pelo objeto. Este método recebe como parâmetro o formato de hora que deve ser utilizado para imprimir as horas (12 ou 24h). Se o objeto foi construído sem informação de horário, este parâmetro não afeta a impressão.
- Os formatos da hora são do tipo int, mas devem ser representados por duas constantes na classe Data: `FORMATO_12H` e `FORMATO_24H`.
- Para entender melhor o funcionamento do método imprimir(), observe como ele deve se comportar em diversas situações:

Exercícios em sala

Código	Resultado
<pre>Data d1 = new Data(10, 03, 2000, 10, 30, 10); d1.imprimir(Data.FORMATO_12H); d1.imprimir(Data.FORMATO_24H);</pre>	<pre>10/3/2000 10:30:10 AM 10/3/2000 10:30:10</pre>
<pre>Data d2 = new Data(15, 06, 2000, 23, 15, 20); d2.imprimir(Data.FORMATO_12H); d2.imprimir(Data.FORMATO_24H);</pre>	<pre>15/6/2000 11:15:20 PM 15/6/2000 23:15:20</pre>
<pre>Data d3 = new Data(5, 10, 2005); d3.imprimir(Data.FORMATO_12H); d3.imprimir(Data.FORMATO_24H);</pre>	<pre>5/10/2005 5/10/2005</pre>

Atividade para Entrega pelo AVA

- 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. Não copie e cole! Aproveite para praticar sintaxe.
- 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.

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

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.

Referências Bibliográficas

- DEITEL, Harvey M. e DEITEL, Paul J. Java - Como Programar, 8ª edição. Pearson. 2010.
- BLOCH, Joshua. Effective Java, 2ª edição. Addison-Wesley, 2008.
- CAELUM. Java e Orientação a Objetos. Disponível em: <https://www.caelum.com.br/apostila-java-orientacao-objetos/>
- SOFTBLUE. Professor Carlos Eduardo Gusso Tosin. Fundamentos de Java. <http://www.softblue.com.br/>.
- K19. Java e Orientação a Objetos. Disponível em: <http://www.k19.com.br/cursos/orientacao-a-objetos-em-java>.
- HORSTMANN, CORNELL. Core Java Volume I – Fundamentos, 8ª Edição. São Paulo, Pearson Education, 2010.
- BRAUDE, E. J. Projeto de software - da programação à arquitetura: uma abordagem baseada em Java. Porto Alegre: Bookman, 2005.
- SANTOS, R. Introdução à Programação Orientada a Objetos usando Java. São Paulo: Campus, 2003.
- Slides do Professor Doutor Horácio Fernandes da UFAM.

“Seja a Mudança que você quer ver no mundo”. Ghandi



filipedwan@gmail.com

 filipedwan

 @filipedwan