

ORIENTAÇÃO A OBJETOS: CONCEITOS INICIAIS

Programação Orientada a Objetos – POOS3

Prof. Dr. Lucas Bueno R. Oliveira



INSTITUTO FEDERAL
São Paulo
Campus São Carlos

O QUE JÁ VIMOS NAS AULAS ANTERIORES

Uma breve introdução ao IntelliJ IDEA.

Uma breve introdução à plataforma Java.

Sintaxe e lógica de programação com linguagem Java.

Alguns programas em linguagem Java.

AO FINAL DESTA AULA, VOCÊ ESTARÁ APTO A...

Compreender o conceito de desenvolvimento de software a partir de classes de objetos.

Implementar, instanciar e compor classes de objetos.

Declarar, utilizar e associar objetos.

Aplicar o conceito de mensagem na invocação de métodos.

Entender a importância do encapsulamento e o papel dos modificadores de acesso.

Desenvolver e utilizar classes com membros estáticos.

Ler e interpretar classes de objeto em notação diagramática.

Utilizar convenções de nome em linguagem Java.

PARADIGMAS DE PROGRAMAÇÃO

Fornecem e determinam a visão que o programador possui sobre a estruturação e execução do programa.

São meios de classificar linguagens de programação a partir de suas funcionalidades.

Alguns dos paradigmas mais conhecidos são:

- Imperativo;
- Estruturado;
- Funcional; e
- Orientado a Objetos.

Note que paradigma e linguagem não são sinônimos!

PROGRAMAÇÃO ESTRUTURADA

Forma de programação na qual todos programas podem ser desenvolvidos com a apenas três estruturas:

- **Estruturas de sequência:** Uma tarefa é executada após a outra, linearmente;
- **Estruturas de decisão:** A partir de um teste lógico um determinado trecho de código é executado; e
- **Estruturas de iteração:** Repetição de um bloco por um número finito de vezes.

Procedimentos são implementados em blocos e a comunicação entre eles se dá pela passagem de dados.

A execução de um programa é caracterizada pelo acionamento de procedimentos destinados à manipulação de dados.

PROGRAMAÇÃO ORIENTADA A OBJETOS

Paradigma que busca representar as propriedades e funções das entidades do **mundo real** em unidades de código chamadas **objetos**.

Sistemas de software são desenvolvidos por meio da **associação** e da **composição** de diferentes objetos.

Permite a construção de sistemas de software por meio de **abstrações**, que podem ser modelos para representar os domínios do problema e da solução.

Segue o princípio de **encapsulamento**, ocultando do usuário a complexidade por trás dos objetos utilizados.

PROGRAMAÇÃO ORIENTADA A OBJETOS

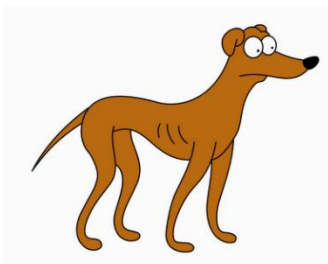
Provê um aumento da **modularidade** e do **reúso** de partes do sistema, de forma sistemática e organizada.

Evolução do reúso de software:

- Reutilização de linhas de código;
- Reutilização de sub-rotinas;
- Reutilização de bibliotecas de funções;
- Orientação a objetos;
- ...

CLASSES DE OBJETOS

“Nosso mundo é um mundo de objetos, para se convencer disso basta tropeçar neles” (Castanheda)



CLASSES DE OBJETOS

Definem de forma conceitual **características** e **comportamentos** comuns a um conjunto de objetos.

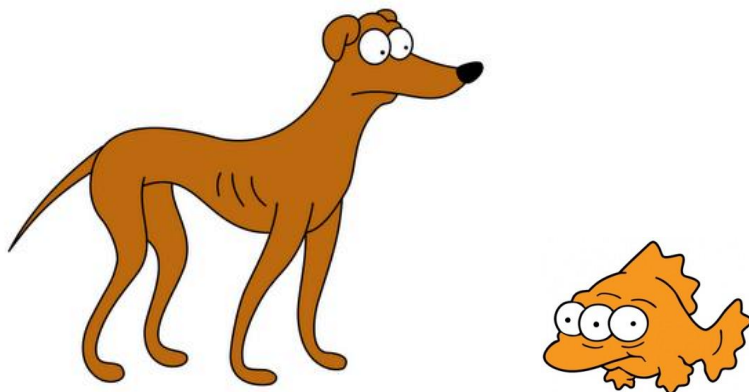
Servem como um **molde** para a criação de diversos **objetos** com características particulares.

Representam **entidades** do mundo real, tais como: pessoas, animais, móveis, imóveis, etc.

Representam também **processos** que ocorrem no cotidiano, tais como: compra, venda, aluguel, etc.

CLASSES DE OBJETOS

Classe Animal e seus objetos:



○ que estes objetos têm em comum?

○ que estes objetos fazem em comum?

CLASSES DE OBJETOS

Atributo: são as características compartilhadas por todos os objetos pertencentes a classe em um dado modelo do domínio. Cada objeto da classe pode apresentar valores específicos para tais características.

Método: define as habilidades que todos os objetos pertencentes a classe possuem. Métodos implementam operações que podem atuar sobre o valor dos atributos dos objetos da classe.

CLASSES DE OBJETOS

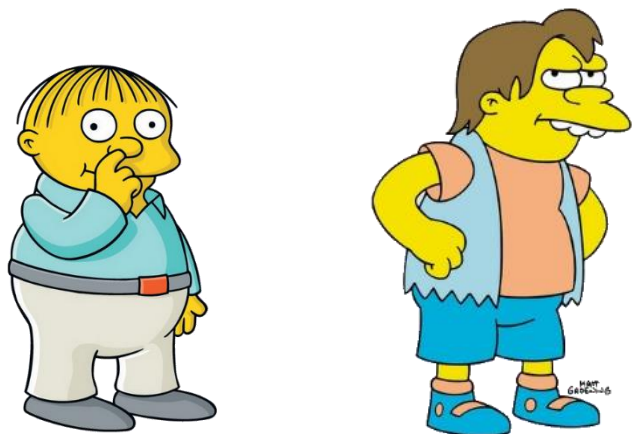
A classe Pessoa e seus atributos:



CLASSES DE OBJETOS

A classe Pessoa e seus métodos:

Falar



Trabalhar



Comer



CLASSES DE OBJETOS

Objetos concretizam as representações definidas pelas classes as quais pertencem.

Os objetos são exemplares únicos de uma classe, que possuem as seguintes propriedades:

- **Identidade:** são únicos em um programa, sendo identificados por uma instância na memória;
- **Estado:** possuem valores armazenados nos atributos definidos pela classe em um dado momento, que descrevem as características particulares deste objeto em sua classe; e
- **Comportamento:** são as ações que um objeto pode tomar com base nos métodos definidos na classe ao qual pertence.

Diferentemente das classes, que são conceituais, objetos representam o ciclo de vida de entidades concretas no domínio da aplicação.

Ou seja, objetos são criados, mudam de estado e deixam de existir ao longo da execução de um programa orientado a objetos.

CLASSES DE OBJETOS

Em termos de programação, definir uma classe significa formalizar um tipo de dado e todas as operações associadas a esse tipo, enquanto declarar objetos significa criar variáveis do tipo definido e atribuir/modificar os valores dessas variáveis por meio das operações.

Veja os objetos da classe Pessoa:



Nome: Bart Simpson
Peso: 42 Kg
Idade: 14
Profissão: Estudante



Nome: Homer Simpson
Peso: 102 Kg
Idade: 48
Profissão: Operador de Usina



Nome: Lisa Simpson
Peso: 38 Kg
Idade: 12
Profissão: Estudante

CLASSES DE OBJETOS

Exemplo de classe em Java:

```
public class Pessoa {  
    String nome, profissao;  
    int idade;  
    long identidade;  
  
    public String falar( ) {  
        return "bla bla bla";  
    }  
    public void trabalhar(int horas) {  
        ...  
    }  
    public void comer( ) {  
        ...  
    }  
}
```

← Nome da classe

← Atributos da classe

← Métodos da classe

MÉTODO CONSTRUTOR

O construtor é um método especial que permite a criação de uma instância (objeto) de uma classe.

Métodos construtores não possuem tipo de retorno.

Uma mesma classe pode possuir múltiplos construtores.

Construtores devem possuir o mesmo nome da classe e toda classe tem pelo menos um construtor.

Se (e somente se) o programador não implementar um construtor, o Java criará um construtor padrão (que não recebe argumentos) de forma implícita.

MÉTODO CONSTRUTOR

Exemplo de construtor para a classe Pessoa:

```
public class Pessoa {  
  
    String nome, profissao;  
    int idade;  
    long identidade;  
  
    public Pessoa() { // Construtor sem argumentos  
    }  
  
    public Pessoa(String nome){ // Construtor que recebe 'nome' como argumento  
        this.nome = nome;  
    }  
    ...  
}
```

MÉTODO CONSTRUTOR

Para criar um objeto de uma classe invocamos o seu método construtor de uma forma especial, usando a palavra-chave **new**:

```
<NomeClasse> <nomeObjeto> = new <NomeConstrutor>(<parametros>);
```

Exemplos de instanciação de objetos da classe pessoa:

```
Pessoa homer = new Pessoa(); //construtor padrão  
Pessoa smithers = new Pessoa("Smithers"); //construtor que recebe uma String
```

MÉTODOS E MENSAGENS

Objetos se comunicam por meio do envio e recebimento de mensagens.

O nome, os parâmetros e o tipo de retorno definem a assinatura de um método, ou seja, como as mensagens devem ser enviadas. O envio de mensagens a um método possui a seguinte sintaxe:

```
<nomeObjeto>.<nomeMetodo> (<parametros>) ;
```



homerSimpson : Pessoa



bartSimpson : Pessoa



bartSimpson.serEsganado()

MÉTODOS E MENSAGENS

Exemplo de envio de mensagem para método da classe Pessoa:

```
public class Usina {  
  
    static void Main(String[] args)  
    {  
        // declara e instancia um objeto do tipo Pessoa  
        Pessoa homer = new Pessoa("Homer");  
        homer.falar("Doh!"); // Usina envia mensagem para homer  
  
        // declara um objeto do tipo Pessoa, que pode ser instanciado mais a frente  
        Pessoa burns;  
        //burns = new Pessoa(); // instancia o objeto 'burns'  
        // Erro, pois o objeto não foi instanciado ainda e a instância vale 'null'  
        burns.falar("Smithers!!!");  
    }  
}
```

MÉTODOS E MENSAGENS

Exemplo de envio de mensagem para método de um objeto em um vetor:

```
public static void main(String[] args) {  
    //vetor com três posições <<VAZIAS>> para objetos da classe aluno  
    Aluno[] alunosPOOS3 = new Aluno[3];  
    alunosPOOS3[0] = new Aluno(); //atribui uma instância de aluno na primeira posição  
    //Se alunosPOOS3[0] é um objeto do tipo Aluno(), então posso chamar o método estudar  
    alunosPOOS3[0].estudar("Estudar Java");  
  
    //Erro! Você está chamando um método de um objeto  
    //que ainda não foi instanciado. Null não faz nada!  
    alunosPOOS3[1].estudar("Estudar muito Java");  
}
```

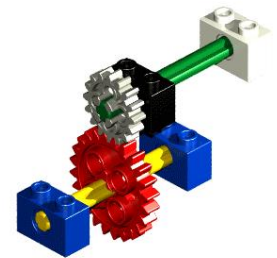
ASSOCIAÇÃO ENTRE OBJETOS

Classes são compostas não apenas por tipos primitivos, mas também por objetos de classes disponíveis na biblioteca padrão (por exemplo, String) ou criadas por desenvolvedores.

Programas orientados a objetos são desenvolvidos a partir da implementação de classes, instanciação de objetos e a associação desses objetos.

A associação (compositiva) entre objetos é um dos mecanismos da orientação a objetos que permite uma maior capacidade de reúso no desenvolvimento de sistemas.

Funciona como um Lego, que você pode usar peças da caixa, montar novas peças e compor peças mais complexas com as peças existentes. O processo de encaixe é a associação.



ASSOCIAÇÃO ENTRE OBJETOS

Exemplo de associação entre classes:

```
public class Personagem {  
    String nome;    int idade;  
    Dublador dublador;  
  
    public Personagem(String nome, int idade) {  
        this.nome = nome; this.idade = idade;  
    }  
    public Dublador getDublador() {  
        return dublador;  
    }  
    public void setDublador(Dublador dublador) {  
        this.dublador = dublador;  
    }  
}
```


ASSOCIAÇÃO ENTRE OBJETOS

Exemplo de associação entre classes:

```
public class Dublador {  
    String nome; int registroProfissional;  
    Personagem personagem;  
  
    public Dublador(String nome, int registroProfissional) {  
        this.nome = nome; this.registroProfissional = registroProfissional;  
    }  
    public Personagem getPersonagem() {  
        return personagem;  
    }  
    public void setPersonagem(Personagem personagem) {  
        this.personagem = personagem;  
    }  
}
```

ASSOCIAÇÃO ENTRE OBJETOS

Exemplo de associação entre classes:

```
public static void main(String[] args) {  
    //cria um personagem do desenho  
    Personagem simpson = new Personagem("Homer Simpson", 45);  
  
    //crio um dublador de personagem  
    Dublador dubl = new Dublador("Carlos A. Vasconcellos", 555);  
  
    //agora que há um personagem e um dublador, posso fazer a associação  
    simpson.setDublador(dubl);  
    dubl.setPersonagem(simpson);  
    //agora o objeto simpson tem uma referência para o objeto dubl e vice-versa  
  
    //imprime o nome do dublador do objeto simpson: simpson -> dubl -> [atributo] nome  
    System.out.println(simpson.getDublador().nome);  
}
```

SOBRECARGA DE MÉTODOS

Métodos podem ser definidos mais de uma vez em uma mesma classe, desde que recebam diferentes conjuntos de tipos parâmetros, esse fenômeno é chamado de **sobrecarga de método**.

Não é permitido definir dois métodos que tenham o mesmo nome e só se diferenciem pelo tipo de retorno ou nome dos parâmetros de entrada.

A sobrecarga de método visa a prover maior flexibilidade na hora de se invocar um determinado comportamento da classe.

SOBRECARGA DE MÉTODOS

Exemplo de sobrecargas de método válidas e inválidas:

```
public String falar(){return "Olá"; } // Válido

public void falar(){ } // Inválido, pois só se diferencia pelo tipo de retorno

public String falar(String nome) { return "Olá, " + nome; } // Válido

// Inválido, pois o nome do argumento muda, mas o tipo é o mesmo
public String falar(String outroNome) { return "Olá, " + outroNome; }

// Válido, pois recebe um conjunto diferente de argumentos
public String falar(int i, String nome) { return i > 0 ? nome : "Blá"; }

// Válido, pois o novo conjunto de argumentos aparece em ordem diferente
public String falar(String nome, int i) { return i > 0 ? nome : "Blá"; }
```

ENCAPSULAMENTO

Consiste na separação de aspectos **internos** e **externos** de um objeto.

Este princípio é utilizado para **impedir o acesso** direto ao estado de um objeto (seus **atributos**) e **detalhes de implementação** de seus métodos.

Um objeto deve ser sempre **coeso** e **autocontido**, de forma a minimizar a complexidade e a comunicação.

Exemplo: você não precisa conhecer os detalhes dos circuitos de um telefone para utilizá-lo. A carcaça do telefone encapsula esses detalhes e você apenas usa os botões.



ENCAPSULAMENTO

Modificadores de acesso viabilizam o encapsulamento de métodos e atributos de uma classe:

- **Público (*public*):** nível de acesso mais permissivo. Atributos e métodos públicos podem ser vistos e acessados de qualquer lugar do código.
- **Privado (*private*):** nível de acesso menos permissivo. Atributos e métodos privados são acessíveis somente dentro do corpo da classe ou por objetos pertencentes a mesma classe.

Boa prática de programação: utilize o modificador *private* a menos que tenha uma boa razão para não fazer isso.

Modificadores de acesso também são aplicados no nível da classe. Algumas classes devem ser marcadas como públicas, outras não necessitam deste modificador (discutiremos isso mais adiante).

ENCAPSULAMENTO

Exemplo de uso de modificadores de acesso:

```
class ContaCorrente {  
    public int var; // var pode ser acessado e alterado  
    private long saldo = 0; // saldo só pode ser alterado pela própria classe  
  
    //Esse método é visível para quem usar objetos da classe ContaCorrente  
    public void deposito(int valor, long conta){  
        if (verificaConta(conta))  
            saldo += valor;  
    }  
  
    // Este método só pode ser visto dentro da própria classe ou por objetos da classe  
    private boolean verificaConta(long conta) {  
        ...  
    }  
}
```

ENCAPSULAMENTO

Atributos privados devem ser acessados e alterados por meio de métodos de acesso *get* e *set*, respectivamente.

- **Métodos de consulta:** *get* + nome do atributo. Ex: *getIdade*
- **Métodos para alteração:** *set* + nome do atributo. Ex: *setIdade*

O uso de métodos de acesso ajuda a manter a consistência das variáveis que compõem o estado de um objeto.

Por exemplo, o uso de um método de acesso pode garantir que o atributo *idade* nunca seja negativo.

Tanto o método *get* quanto o *set* podem também ser omitidos quando não fizer sentido acessar ou alterar o valor de um atributo, respectivamente.

ENCAPSULAMENTO

Exemplo de uso dos métodos get e set:

```
public class Pessoa {  
  
    private int idade;  
  
    // 'get' sucedido do nome da variável. 'is' sucedido do nome, se a variável for booleana  
    public int getIdade() {  
        return idade;  
    }  
  
    // 'set' sucedido do nome da variável  
    public void setIdade(int idade) {  
        // exemplo de onde usar if ternário  
        this.idade = idade >= 0 ? idade : 0; // "this" desambigua a variável interna da externa  
    }  
}
```

CLASSES E MEMBROS ESTÁTICOS

Em alguns casos, deseja-se utilizar uma funcionalidade ou atributo de uma classe, mas não há interesse em criar ou manipular o estado de objetos dessa classe para isso.

O modificador *static* permite criar métodos e atributos que pertencem a própria classe, ao invés de suas instâncias. Por isso, esses membros são chamados “membros de classe”.

Enquanto uma instância possui cópias individuais dos membros da classe, existe apenas uma cópia de cada membro estático.

```
<modificadorDeAcesso> static <TipoObjeto> <nomeDoObjeto>;
```

```
<modificadorDeAcesso> static <TipoRetorno> <nomeDoMetodo> (<parametros>);
```

CLASSES E MEMBROS ESTÁTICOS

Membros estáticos são construídos logo que a aplicação é criada, por isso, todos os elementos dentro do escopo devem se comportar como estáticos.

Não é possível acessar diretamente uma variável ou método não estático dentro de um método estático sem a criação de uma instância de objeto.

Variáveis estáticas só podem ser declaradas no nível da classe.

Ao se referenciar a um membro estático, deve-se utilizar o caminho completo (nome da classe seguido do nome do membro). Veja o exemplo da classe Math:

```
//Método estático e atributo (constante) estático  
double d = Math.cos(Math.PI);
```

CLASSES E MEMBROS ESTÁTICOS

Exemplo de classe com método estáticos:

```
public class StringUtility { //Provê funcionalidades para manipular Strings

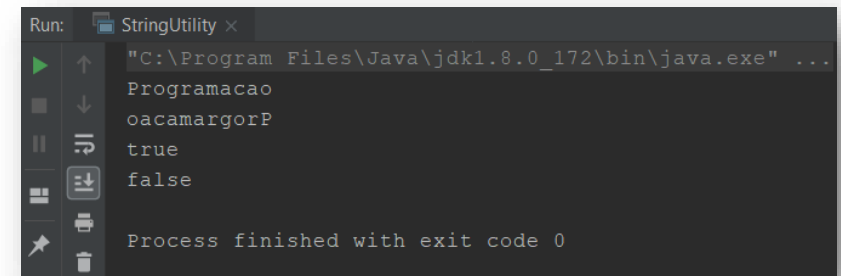
    public static String inverte(String text){
        String invertida = "";
        for (int i = text.length() - 1; i >= 0; --i)
            invertida += text.charAt(i);
        return invertida;
    }

    public static boolean possuiCaracter(String text, char c){
        for (char o : text.toCharArray())
            if (o == c)
                return true;
        return false;
    }
}
```

CLASSES E MEMBROS ESTÁTICOS

Exemplo de classe com método estáticos:

```
public static void main(String[] args) {  
    String texto = "Programacao";  
    System.out.println(texto);  
    System.out.println(StringUtility.Inverte(texto));  
    System.out.println(StringUtility.PossuiCaracter(texto, 'a'));  
    System.out.println(StringUtility.PossuiCaracter(texto, 'b'));  
}
```



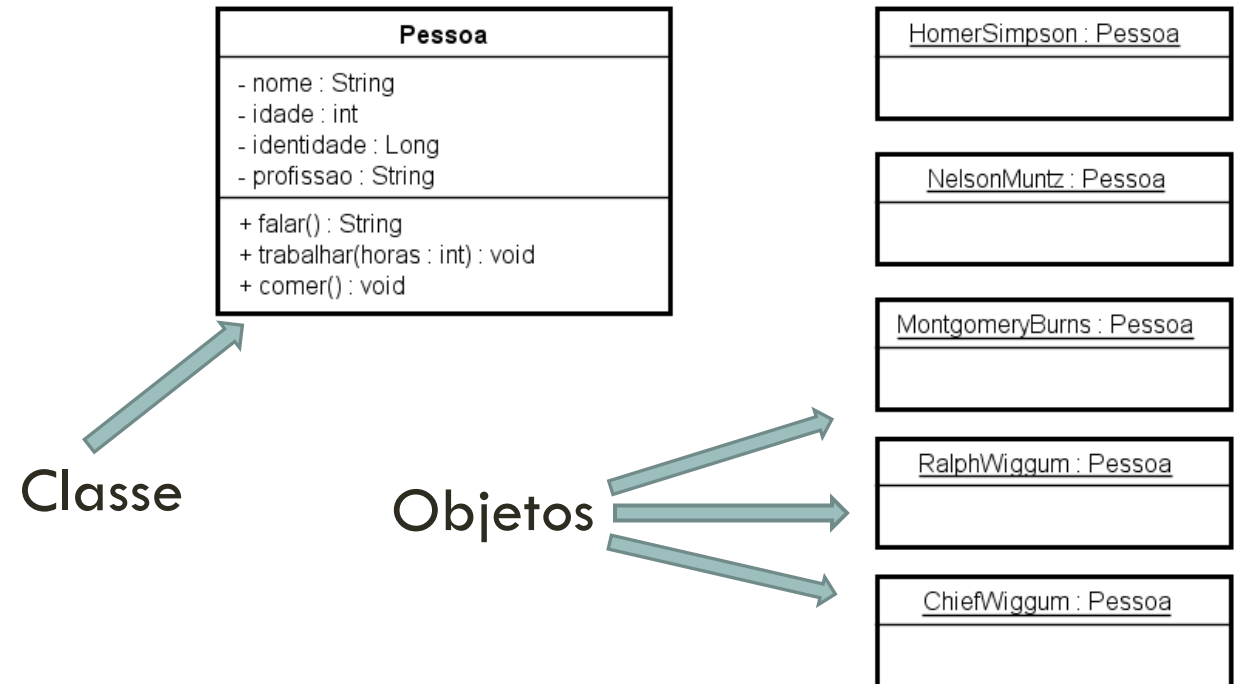
The screenshot shows a 'Run' console window for a class named 'StringUtility'. The command executed is 'C:\Program Files\Java\jdk1.8.0_172\bin\java.exe ...'. The output consists of five lines: 'Programacao', 'oacamargorP', 'true', 'false', and 'Process finished with exit code 0'.

```
Run: StringUtility x  
"C:\Program Files\Java\jdk1.8.0_172\bin\java.exe" ...  
Programacao  
oacamargorP  
true  
false  
Process finished with exit code 0
```

CLASSES DE OBJETOS: NOTAÇÃO DIAGRAMÁTICA

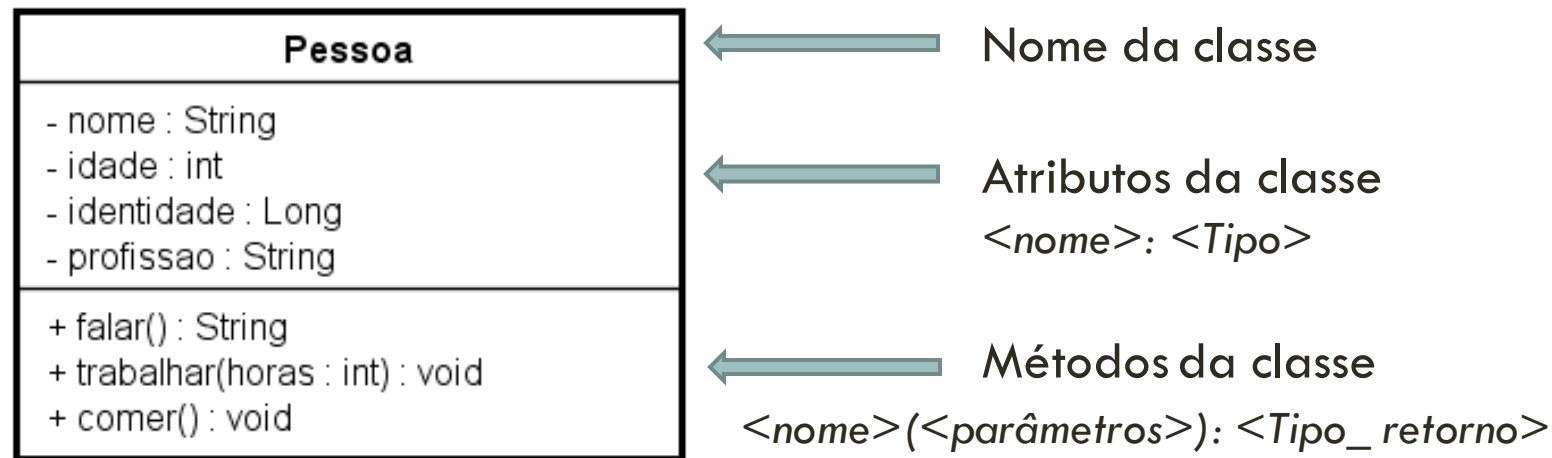
A UML (*Unified Modeling Language*) é uma linguagem que permite representar modelos orientados a objeto de forma independente de linguagem de programação.

Nessa disciplina, serão introduzidos alguns conceitos iniciais da UML para facilitar a comunicação, mas esse assunto será abordado em detalhes na disciplina Desenvolvimento Orientado a Objetos (DOOS4)



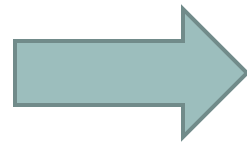
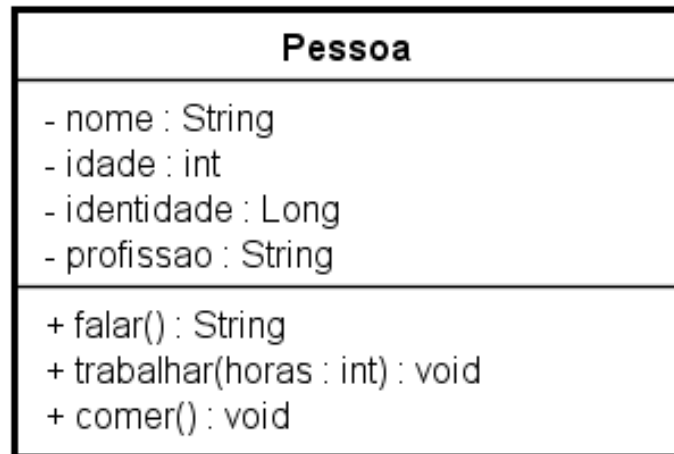
CLASSES DE OBJETOS: NOTAÇÃO DIAGRAMÁTICA

Uma classe em UML é representada da seguinte forma:



CLASSES DE OBJETOS: NOTAÇÃO DIAGRAMÁTICA

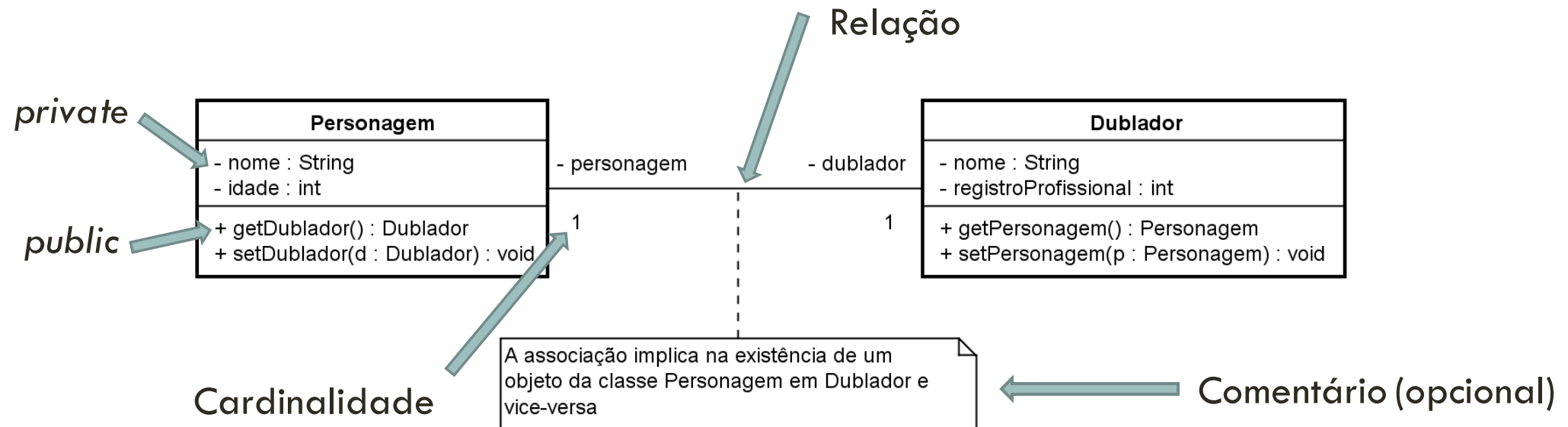
Uma classe em UML e seu respectivo código-fonte em Java:



```
public class Pessoa {  
    private String nome;  
    private int idade;  
    private long identidade;  
    private String profissao;  
  
    public String falar(){...}  
  
    public void trabalhar(int horas){...}  
  
    public void comer(){...}  
}
```


CLASSES DE OBJETOS: NOTAÇÃO DIAGRAMÁTICA

Uma relação em UML é representada da seguinte forma:



CONVENÇÕES DE NOME EM JAVA

Para que desenvolvedores possam reutilizar código e trabalhar em conjunto é preciso haver padronização.

Cada linguagem possui suas convenções de nome, que estabelecem uma forma unificada de declarar elementos.

Em Java as convenções de nome são:

- Nomes de classes, interfaces e enumerações começam com letra maiúscula;
- Variáveis que declaram objetos ou primitivos e métodos começam com letra minúscula;
- Variáveis não devem começar com '_' ou '\$'.
- Variáveis com um único caractere devem ser evitadas, a menos que sejam temporárias;
- Nomes compostos utilizam a forma *camelCase*; e
- Constantes possuem todos os caracteres em maiúsculo, sendo os nomes compostos separados por '_'.

CONVENÇÕES DE NOME EM JAVA

Exemplo de classe com método estáticos:

```
private Pessoa pessoa = new Pessoa("Homer"); // Atributo em minúsculo
private Pessoa alunoEspecial = new Pessoa("Burns");

private int numMatricula = 0; // Atributo com camelCase
private enum Genero { MASC, FEM }; // Enumeração em maiúsculo

// Constante em maiúsculo. Caso seja composta, usar UMA_CONSTANTE_COMPOSTA.
public static final String CONST = "A culpa é minha e eu coloco em quem quiser (Simpson, H.)";

public void cadastrar(Pessoa p) { } // Método em maiúsculo
public void cadastrarAluno(Pessoa a) { } // Método em CamelCase
```

RESUMO DA AULA

Programas orientados a objeto são projetados pensando no domínio da aplicação.

Classes são “moldes” conceituais e objetos são entidades concretas com um ciclo de vida.

Construtores são métodos especiais que criam instâncias de uma classe.

O desenvolvimento OO envolve a definição e associação entre classes de objetos.

É possível definir diferentes assinaturas para um método e isso se chama sobrecarga.

O encapsulamento oculta a complexidade dos detalhes de implementação de uma classe.

Atributos estáticos pertencem a classe e métodos estáticos não dependem de uma instância.

INFORMAÇÕES ADICIONAIS

Comparação entre paradigmas: <https://www.youtube.com/watch?v=cgVVZMfLjEI> (hard)

Construtores: <https://docs.oracle.com/javase/tutorial/java/javaOO/constructors.html>

Sobrecarga: <https://www.javaworld.com/article/3268983/java-challengers-1-method-overloading-in-the-jvm.html>

Modificadores de acesso: <https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html>

Membros estáticos: <https://www.baeldung.com/java-static>

EXERCÍCIO

Implemente o seguinte diagrama UML em linguagem Java:

