

# Programação Orientada a Objetos

**Aula: Introdução a Orientação a Objetos, Paradigmas de Programação e Linguagem de Programação Java.**



Prof. Anderson Augusto Bosing

# Introdução a Linguagem de Programação Java

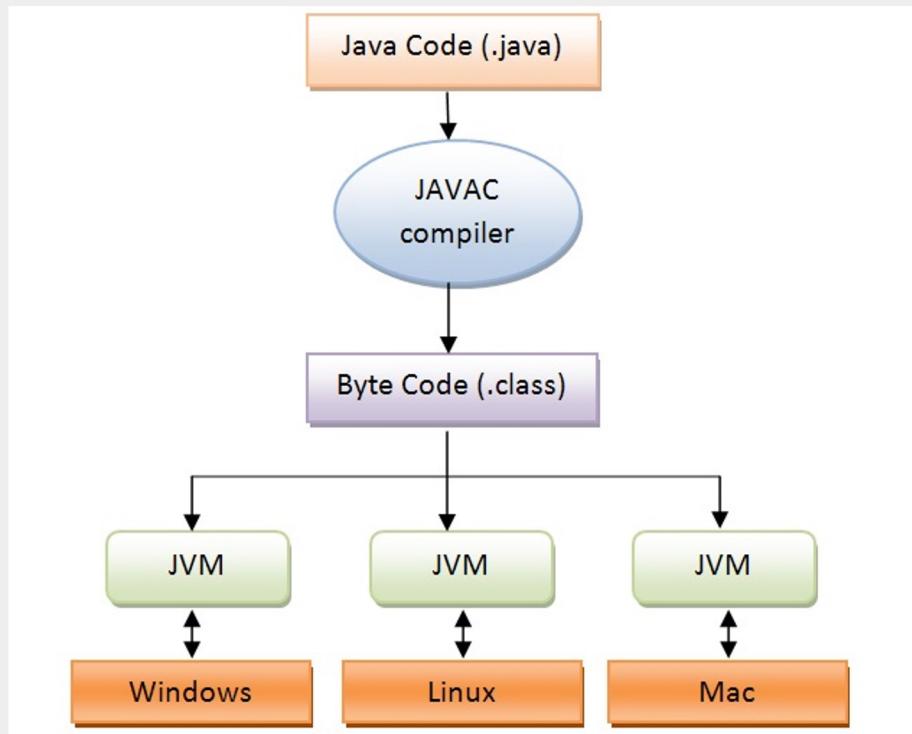
**Java é uma linguagem de programação orientada a objetos que começou a ser criada em 1991, na Sun Microsystems. Teve início com o Green Project, no qual os mentores foram Patrick Naughton, Mike Sheridan, e James Gosling.**

**A linguagem JAVA é multiplataforma. Esta afirmação reporta ao fato de que um programa escrito na linguagem JAVA pode ser executado em qualquer plataforma (sistema operacional) sem necessidade de alterações no código-fonte.**



# Introdução a Linguagem de Programação Java

Tal funcionalidade é possível devido à estrutura de linguagem interpretada que caracteriza a linguagem JAVA e seu processo de compilação do código-fonte.



# Introdução a Linguagem de Programação Java

**Principais características e vantagens da linguagem Java:**

- **Suporte à orientação a objetos;**
- **Portabilidade;**
- **Linguagem Simples;**
- **Alta Performance;**
- **Independente de plataforma;**
- **Tipada (detecta os tipos de variáveis quando declaradas);**
- **Alta aceitação de mercado tanto regional quanto mundial.**



# Tipos de Dados-Java

**Tipos de dados são especificados de diferentes tamanhos e valores que podem ser armazenados na variável. Existem dois tipos de dados em java:**

- **Tipo de dado primitivo.**
- **Tipo de dado não primitivo.**



# Tipos de Dados-Java

## Dados Primitivos

➤ **boolean**

**Exemplo de utilização**

```
boolean eMenorQue = false;
```

**Utilizado para armazenar valores de verdadeiro ou falso(true, false);**



# Tipos de Dados-Java

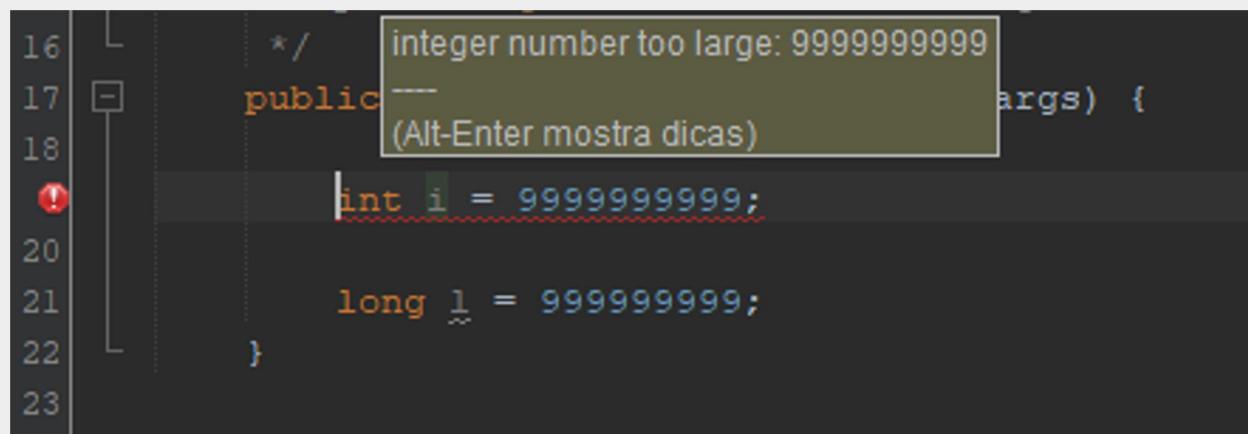
## Dados Primitivos

➤ **long**

**Exemplo de utilização**

```
long a = 100000;
```

**Utilizado para armazenar um números inteiros que não são suportados pelo tipo int.**



A screenshot of an IDE showing Java code. The code is as follows:

```
16 | L      */
17 |     public static void main(String[] args) {
18 |         int i = 999999999; // integer number too large: 999999999
19 |         long l = 999999999;
20 |
21 |
22 |     }
23 | }
```

A tooltip is displayed over the line `int i = 999999999;`. The tooltip contains the text `integer number too large: 999999999` and `(Alt-Enter mostra dicas)`.



# Tipos de Dados-Java

## Dados Primitivos

➤ **double**

**Exemplo de utilização**

```
double i = 345.455555;
```

**Utilizado para armazenar um números com casas decimais chamados de ponto-flutuante.**



# Tipos de Dados-Java

## Dados Não Primitivos

### ➤ String

**Exemplo de utilização**

```
String i = "Professor Anderson";
```

**Utilizado para armazenar valores que podem ser expressados por palavras.**



# Operadores de Atribuição-Java

O operador de atribuição é utilizado para definir o valor inicial ou sobrescrever o valor de uma variável.

```
int idade = 15;  
String nome = "Anderson";  
double salario = 450.00;
```



# Operadores Aritméticos-Java

**Os operadores aritméticos realizam as operações fundamentais da matemática entre duas variáveis e retornam o resultado.**

```
int area = 2 * 2;|
```

+	operador de adição
-	operador subtração
*	operador de multiplicação
/	operador de divisão
%	operador de módulo (ou resto da divisão)



# Operadores de Igualdade-Java

**Os operadores de igualdade verificam se o valor ou o resultado da expressão lógica à esquerda é igual (“==”) ou diferente (“!=”) ao da direita, retornando um valor booleano.**

```
int valorA = 1;
int valorB = 2;

if (valorA == valorB) {
    System.out.println("Valores iguais");
} else {
    System.out.println("Valores diferentes");
}

if (valorA != valorB) {
    System.out.println("Valores diferentes");
} else {
    System.out.println("Valores iguais");
}
```

==	Utilizado quando desejamos verificar se uma variável é igual a outra.
!=	Utilizado quando desejamos verificar se uma variável é diferente de outra.



# Operadores relacionais-Java

Os operadores relacionais, assim como os de igualdade, avaliam dois operandos. Neste caso, mais precisamente, definem se o operando à esquerda é menor, menor ou igual, maior ou maior ou igual ao da direita, retornando um valor booleano.

```
int valorA = 1;
int valorB = 2;

if (valorA > valorB) {
    System.out.println("maior");
}

if (valorA >= valorB) {
    System.out.println("maior ou igual");
}

if (valorA < valorB) {
    System.out.println("menor");
}

if (valorA <= valorB) {
    System.out.println("menor ou igual");
}
```



# Operadores Lógicos - Java

**Os operadores lógicos representam o recurso que nos permite criar expressões lógicas maiores a partir da junção de duas ou mais expressões. Para isso, aplicamos as operações lógicas E (representado por “`&&`”) e OU (representado por “`||`”).**

```
if ((1 == (2 - 1)) && (2 == (1 + 1))) {  
    System.out.println("Ambas as expressões são verdadeiras");  
}
```

<code>&amp;&amp;</code>	Utilizado quando desejamos que as duas expressões sejam verdadeiras.
<code>  </code>	Utilizado quando precisamos que pelo menos um das expressões seja verdadeira.



# Estruturas Condicionais –Java

## If/Else

As estruturas condicionais possibilitam ao programa tomar decisões e alterar o seu fluxo de execução. Isso possibilita ao desenvolvedor o poder de controlar quais são as tarefas e trechos de código executados de acordo com diferentes situações, como os valores de variáveis.

```
//Se a posição for múltipla de 3
if (i % 3 == 0) {
    //Múltiplos de 3
    //Índice da Posição x 30% x Valor Informado pelo Usuário
    array[i] = i * (0.3 * valueUser);

} else {
    //Índice da Posição x 10% x Valor informado pelo Usuário
    array[i] = i * (0.1 * valueUser);
}
```

```
if (somaPares.equals("par")) {
    indice = 0;
} else {
    indice = 1;
}
```



# Estruturas Condicionais –Java Switch/Case

A estrutura condicional switch/case vem como alternativa em momentos em que temos que utilizar múltiplos ifs no código. Múltiplos if/else encadeados tendem a tornar o código muito extenso, pouco legível e com baixo índice de manutenção.

```
int mes = 1;
switch (mes) {
    case 1:
        System.out.println("O mês é janeiro");
        break;
    case 2:
        System.out.println("O mês é fevereiro");
        break;
    default:
        System.out.println("Mês inválido");
        break;
}
```



# Conhecendo nossa IDE



# Estruturas de Repetição – Java

## While

Especifica que um sistema ou programa de computador deve repetir uma instrução ou um conjunto de instruções enquanto a condição que é uma expressão booleana permanece verdadeira.

```
int num = 0;

while (num <= 99) {
    System.out.println("nnum = "+num);

    num++;
}
```

```
-- exec-maven-plugin:1.2.1:exec (default-cli) @ ListaVetores --
nnum = 0
nnum = 1
nnum = 2
nnum = 3
nnum = 4
nnum = 5
nnum = 6
nnum = 7
nnum = 8
nnum = 9
nnum = 10
```



# Estruturas de Repetição – Java

## Do - While

```
int num = 0;

do {
    System.out.println("num = "+num);
    num++;
} while (num < 100);
```

```
-- exec-maven-plugin:1.2.1:exec (default-cli) @ ListaVetores --
num = 0
num = 1
num = 2
num = 3
num = 4
num = 5
num = 6
num = 7
num = 8
num = 9
num = 10
```



# Estruturas de Repetição – Java

## For

O for também é uma instrução de repetição que processa uma instrução ou grupo de instrução zero ou mais vezes. O que munda é a sintaxe, no caso do for a variável de controle, o valor inicial, o incremento e a condição de continuação do loop, são declarados como em uma espécie de cabeçalho.

```
for (int num = 0; num < 10; num++) {  
    System.out.println("num =" + num);  
}
```



# System.out.print – Java

O for também é uma instrução de repetição que processa uma instrução ou grupo de instrução zero ou mais vezes. O que munda é a sintaxe, no caso do for a variável de controle, o valor inicial, o incremento e a condição de continuação do loop, são declarados como em uma espécie de cabeçalho.

```
public static void main(String[] args) {  
    System.out.println("Bem vindos ao segundo ano da UNIPAR");  
}
```

```
[...] --- exec-maven-plugin:1.2.1:exec (default-cli) @ ListaVetores ---  
| Bem vindos ao segundo ano da UNIPAR  
|-----  
| BUILD SUCCESS
```



# Classe Scanner – Java

Como ler dados informados pelo usuário ?

A classe Scanner apareceu a partir do Java 5 com o objetivo de facilitar a entrada de dados no modo Console. Uma das características mais interessante da classe Scanner é a possibilidade de obter o valor digitado diretamente no formato do tipo primitivo que o usuário digitar. Para isso basta utilizarmos os métodos next do tipo primitivo no formato nextTipoDado().

```
Scanner scanner = new Scanner(System.in);
System.out.print("Qual o seu nome: ");
String nome = scanner.next();
System.out.println("Seja bem vindo " + nome + "!");
```

```
Qual o seu nome: Anderson
Seja bem vindo Anderson!
-----
BUILD SUCCESS
```



# Variáveis em Java

```
String nome = "Professor Anderson";
int idade;
int Idade;
idade = 29;
```



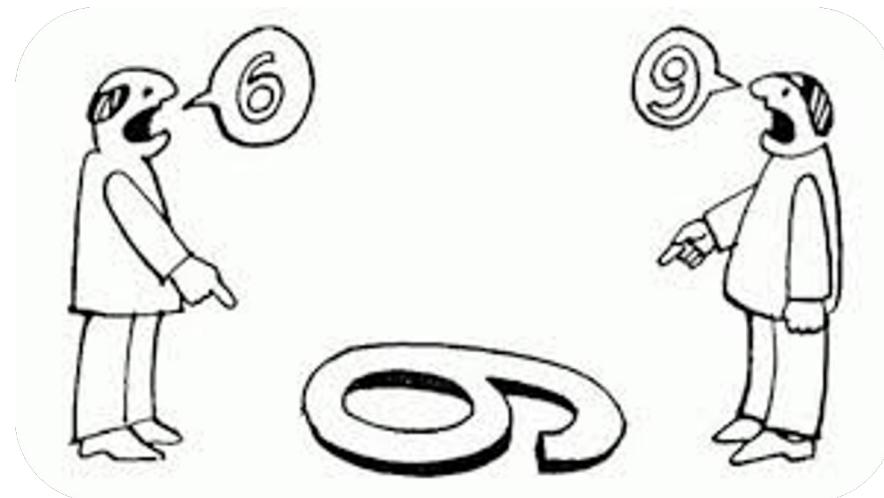
# Programação Orientada a Objetos

**Aula: Introdução a Classes  
e Objetos.**



Prof. Anderson Augusto Bosing

# O que é um Paradigma?



# O que é um **Paradigma**?

**Um exemplo que serve como modelo ou padrão.**



# O que são Paradigmas da Programação?



# O que são Paradigmas da Programação?

**Um paradigma é um estilo de programação, um modelo, uma metodologia.**

**Não se trata de uma linguagem, mas a forma como você soluciona problemas usando uma determinada linguagem de programação.**



# **Paradigmas da Programação**

## **Procedural**

**Consiste em um modelo de programação que funciona como uma espécie de lista de instruções que são executadas em forma de passo a passo.**

**Ex:**

**C**

**Pascal**



# **Paradigmas da Programação**

## **POO**

**Este paradigma é o que mais reflete os problemas atuais. Um programa OO consistem em objetos que enviam mensagens uns para os outros. Estes objetos no programa correspondem diretamente a objetos atuais, tais como pessoas, máquinas, departamentos, documentos e assim por diante.**

**Ex:**

**Java**

**C#**



# O paradigma de Orientação a Objetos

Historicamente antes de 1975 a maioria das empresas de software não usava nenhuma técnica específica, cada indivíduo trabalhava do seu próprio jeito.

Grandes avanços foram feitos aproximadamente entre 1975 e 1985, com o desenvolvimento assim chamado paradigma clássico ou estruturado. Essa abordagem parecia extremamente promissora para a época. Todavia à medida que o tempo foi passando, constatou-se que ela ficou aquém do esperado em dois principais aspectos:

1. Algumas vezes o paradigma e suas técnicas eram incapazes de lidar com o tamanho cada vez maior dos produtos de software, isso é, as técnicas clássicas eram adequadas para elaborar produtos com escala pequena ou média.
2. O paradigma clássico não estava a altura das expectativas iniciais durante a manutenção pós-entrega.

# O paradigma de Orientação a Objetos

A principal razão para o sucesso limitado desse paradigma clássico foi que as técnicas são orientadas a operações ou a atributos(dados), mas não a ambos ao mesmo tempo.

Em contraste, o paradigma de orientação a objetos considera igualmente importantes tanto os atributos quanto as operações. Uma maneira simplista de compreender um objeto é vê-lo como um artefato de software unificado, que incorpora tanto os atributos quanto as operações realizadas sobre os atributos



# Evolução Histórica

Bezerra (2015) apresenta um breve resumo histórico da evolução das técnicas de desenvolvimento, para explicar como chegamos ao cenário atual.

**Décadas de 1950/1960:** Os sistemas eram bem simples, e o seu desenvolvimento era direto ao assunto, não tinha um planejamento inicial, como dizem, “ad hoc”. Como os sistemas eram significativamente mais simples, as técnicas de modelagem também: Eram usados fluxogramas e diagramas de módulos.

**Década de 1970:** Começaram a surgir computadores mais avançados e acessíveis. Houve grande ampliação do mercado computacional. Sistemas mais complexos começavam a surgir. Consequentemente, modelos mais robustos foram propostos. Os autores Larry Constantine e Edward Yourdon são grandes colaboradores nessas técnicas.



# Evolução Histórica

**Década de 1980:** Nessa fase os computadores se tornaram ainda mais avançados e mais baratos. Surge a necessidade por interfaces mais sofisticadas, o que originou a produção de sistemas de softwares mais complexos. A Análise Estruturada surgiu no início desse período, com os trabalhos de Edward Yourdon, Peter Coad, Tom De Marco, James Martin e Chris Gane.

**Década de 1990:** No início dessa década é o período em que surge um novo paradigma de modelagem, a Análise Orientada a Objetos, como resposta a dificuldades encontradas na aplicação da análise estruturada em algumas aplicações. Grandes colaboradores desse paradigma são Sally Shlaer, Stephen Mellor, Rebecca Wirfs-Brock, James Rumbaugh, Grady Booch e Ivar Jacobson. Já no fim da década, o paradigma da orientação a objetos atinge sua maturidade. Os conceitos de padrões de projeto, frameworks, componentes e qualidade começam a ganhar espaço. Surge a Linguagem de Modelagem Unificada UML.



# Evolução Histórica

**Década de 2000:** O reúso por meio de padrões de projetos e frameworks se solidifica. As denominadas metodologias ágeis começam a ganhar espaço. Técnicas de testes automatizados e refatoração começaram a se difundir entre os desenvolvedores que trabalham com orientação a objeto. Grandes nomes dessa fase são: Rebecca Wirfs-Brock, Martin Fowler e Eric Vans.

Como percebemos, durante a década de 90 surgiram várias propostas e técnicas para modelagem de sistema segundo o paradigma da orientação a objeto. Era comum, durante essa década, duas técnicas possuírem diferentes notações gráficas para modelar a mesma perspectiva de um sistema. Todas tinham pontos fortes e fracos em relação à notação utilizada, mas via-se a necessidade de uma notação que viesse a se tornar um padrão para a modelagem de sistemas orientados a objeto, e que fosse amplamente aceita, nas indústrias e na academia.

Alguns esforços surgiram em 1996 para essa padronização, o que resultou na definição da UML (Unified Modeling Language), que detalharemos a seguir em um tópico específico, mas antes falaremos na Modelagem Estruturada e da Modelagem Orientada a Objetos.



# Evolução Histórica

Pode-se construir uma casa para um cachorro apenas juntando algumas tábuas, alguns pregos e algumas ferramentas básicas. Com um planejamento mínimo, em poucas horas e com um pouco de habilidade, nosso cachorro terá uma casa (BOOCH et al., 2000).



# Introdução à Tecnologia de Objetos

Hoje, como a demanda por software novo e mais poderoso está aumentando, construir softwares de maneira rápida, correta e econômica continua a ser um objetivo indefinido.

Objetos ou, mais precisamente, as classes de onde os objetos são essencialmente componentes reutilizáveis de software.

Há objetos data, objetos data/hora, objetos áudio, objetos vídeo, objetos automóvel, objetos pessoas etc.

Quase qualquer substantivo pode ser razoavelmente representado como um objeto de software em termos dos **atributos** (por exemplo, nome, cor e tamanho) e **comportamentos** (por exemplo, calcular, mover e comunicar).



# Introdução à Tecnologia de Objetos

**Quando programamos estamos modelando aspectos do ‘mundo real’ utilizando uma linguagem de programação. Assim sempre temos um aspecto do ‘mundo real’, a representação deste aspecto no ‘mundo computacional’ (em tempo de execução) e a descrição deste aspecto no ‘mundo linguístico’ (em uma linguagem de programação).**



# O automóvel como um objeto

Vamos supor que você queira dirigir um carro e fazê-lo andar mais rápido pisando no pedal acelerador. O que deve acontecer antes que você possa fazer isso?



# O automóvel como um objeto

Vamos supor que você queira dirigir um carro e fazê-lo andar mais rápido pisando no pedal acelerador. O que deve acontecer antes que você possa fazer isso?

Alguém precisa projetá-lo.

Criar uma especificação que vai servir como base para fabricação dos carros que vão ser utilizados nas ruas.



# **Considerando um software para um banco.**

**O que toda conta corrente tem que é importante para nós?**



# **Considerando um software para um banco.**

**O que toda conta corrente tem que é importante para nós?**

- Número da conta
- Nome do titular da conta
- Saldo
- Limite



# **Considerando um software para um banco.**

**O que toda conta corrente faz que é importante para nós?**

**O que pedimos à conta corrente?**



# **Considerando um software para um banco.**

**O que toda conta corrente faz que é importante para nós?**

**O que pedimos à conta corrente?**

- **Saca uma quantidade x;**
- **Deposita uma quantidade x;**
- **Consultar o saldo atual;**
- **Imprimir extrato.**
- **Transfere uma quantidade x para uma outra conta y;**



# Considerando um software para um banco.

De acordo com o que levantamos então temos uma especificação de uma conta. Sendo ela:

## Atributos de Uma conta

- Número da conta
- Nome do titular da conta
- Saldo
- Limite

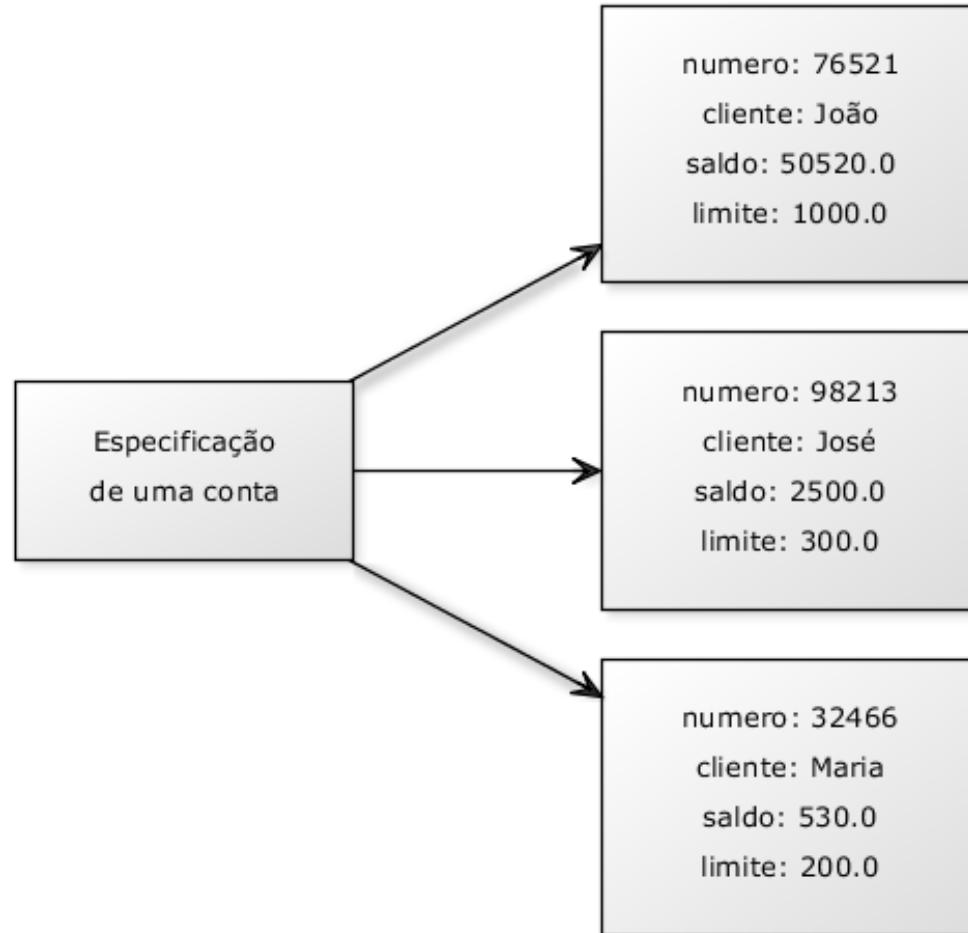
## Métodos de uma Conta

- Saca uma quantidade x;
- Deposita uma quantidade x;
- Consultar o saldo atual;
- Imprimir extrato.
- Transfere uma quantidade x para uma outra conta y;

Mas o que temos ainda é o projeto dela, antes de a utilizarmos precisamos construir uma conta.



# Considerando um software para um banco.



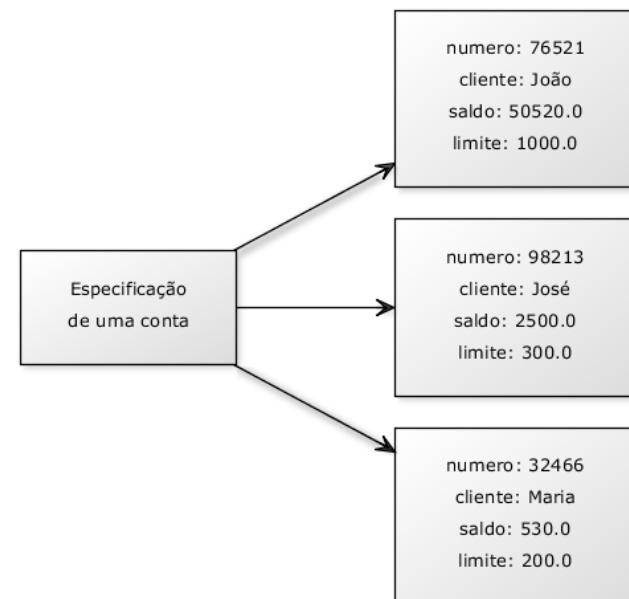
# Considerando um software para um banco.

Na figura anterior podemos observar que ao lado esquerdo temos a especificação de uma conta, mas essa especificação é uma conta ?

Nós depositamos e sacamos dinheiro desse papel?

Não. Utilizamos a especificação da Conta para poder criar instâncias que realmente são contas, nas quais podemos realizar as operações que criamos.

Apesar de declararmos que toda conta tem um saldo, um número e uma agência no pedaço de, são nas instâncias desse projeto em que realmente há espaço para armazenar esses valores.

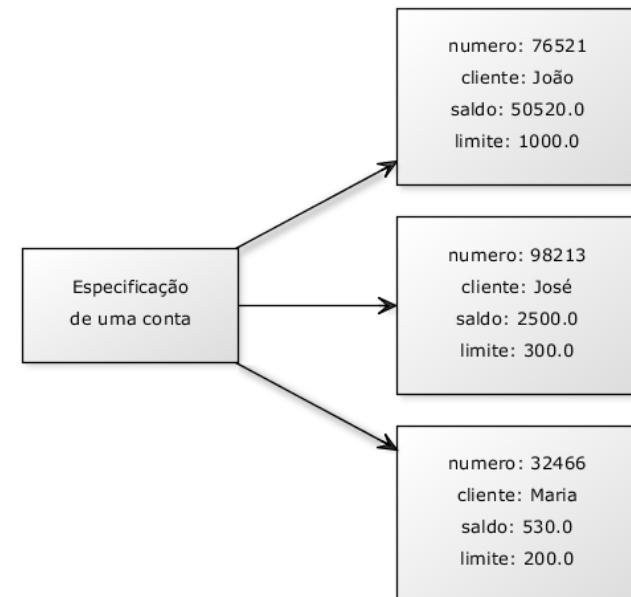


# Considerando um software para um banco.

Ao projeto da conta, isto é, à especificação da conta, damos o nome de **classe**. Ao que podemos construir a partir desse projeto que são as contas de verdade, damos o nome de **objetos**.

As características da conta damos o nome de **atributos**(Número, Saldo, Cliente).

E aos comportamentos desta conta damos o nome de **métodos**(Sacar, depositar, imprimir extrato).



# Um outro exemplo: uma receita de bolo

A pergunta é certeira: você come uma receita de bolo? Não.

Precisamos instanciá-la e fazer um **objeto bolo** a partir dessa **especificação (a classe)** para utilizá-la.

Podemos criar centenas de bolos com base nessa **classe (a receita, no caso)**. Eles podem ser bem semelhantes, alguns até idênticos, mas são **objetos diferentes**.



# Um outro exemplo: planta de uma casa

A planta de uma casa é uma casa? Definitivamente, não.

Não podemos morar dentro da planta de uma casa nem podemos abrir sua porta ou pintar suas paredes.

Precisamos, antes, construir instâncias a partir dessa planta. Essas instâncias, sim, podemos pintar, decorar ou morar dentro.



# Definições Técnicas

## Classe

Assim como os objetos do mundo real, uma classe agrupa os objetos pelos seus **comportamentos e atributos comuns**.

Uma classe define os **atributos e comportamentos comuns compartilhados por uma tipo de objeto**. Os objetos de certo tipo ou **classificação compartilham os mesmos comportamentos e atributos**.



# Definições Técnicas

## Objeto

**Um objeto é uma instância de uma classe.**

## Atributo

**Atributos são as características de uma classe visíveis externamente. A cor de um carro e o seu modelo são exemplos de atributos.**

## Método

**Métodos definem as habilidades dos objetos**



# Vamos codificar.

- Criar uma classe câmera.
- Criar uma classe gato.



# Vamos Praticar?

- . Lista de Exercícios em Orientação a Objetos com Java disponível no classroom.



# Programação Orientada a Objetos

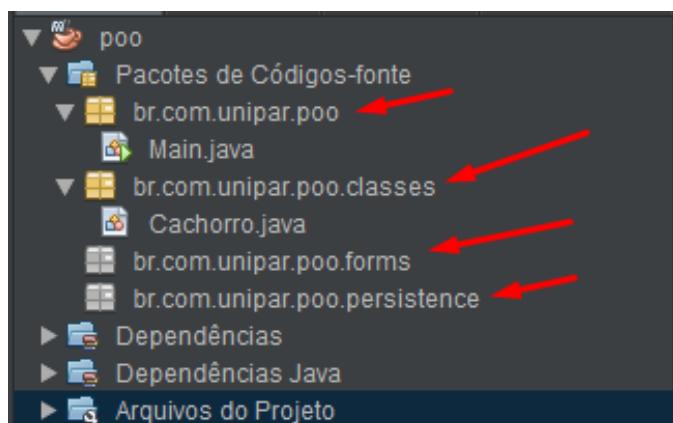
**Aula: Modificadores de  
acesso,  
Métodos Acessores.**



Prof. Anderson Augusto Bosing

# Pacotes

Pacotes java são utilizados para organizar as classes da sua aplicação. Um programa pode, facilmente, ter mais de centenas de classes. Então é muito importante que todos os seus componentes fiquem organizados. Podemos pensar nos pacotes como uma pasta do seu sistema de arquivos.



Nome	Data de modificação	Tipo	Tamanho
classes	05/04/2022 18:30	Pasta de arquivos	
forms	05/04/2022 18:30	Pasta de arquivos	
persistence	05/04/2022 18:30	Pasta de arquivos	
Main.java	05/04/2022 18:29	Arquivo Fonte Java	1 KB

# Modificadores de Acesso a Nível de Classe - public

Modificador **public** torna uma classe visível:

Para qualquer outra classe e em qualquer pacote.

```
 */
public class Cachorro {  
    |  
}
```



# Modificadores de Acesso a Nível de Classe - default

Modificador vazio ou default torna uma classe visível:

Torna uma classe visível apenas para classes do mesmo pacote.

```
/*  
class Cachorro {  
}
```



# Modificadores de Acesso a Nível de Atributos e Métodos(Membros) - **public**

**public** torna um membro acessível:

Em qualquer lugar e a qualquer outra classe que possa visualizar a classe que contém o membro.

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

```
/*  
 * @param args the command line arguments  
 */  
public static void main(String[] args) {  
  
    Cachorro dog = new Cachorro();  
    dog.nome = "bob";  
  
}  
}
```

# Modificadores de Acesso a Nível de Atributos e Métodos(Membros) - **protected**

**protected** torna um membro acessível às classes:

Do mesmo pacote.

Os membros herdados não são acessíveis a outras classes fora do pacote em que foram declarados.

```
/*
public class Cachorro {

    protected String nome;
}
```

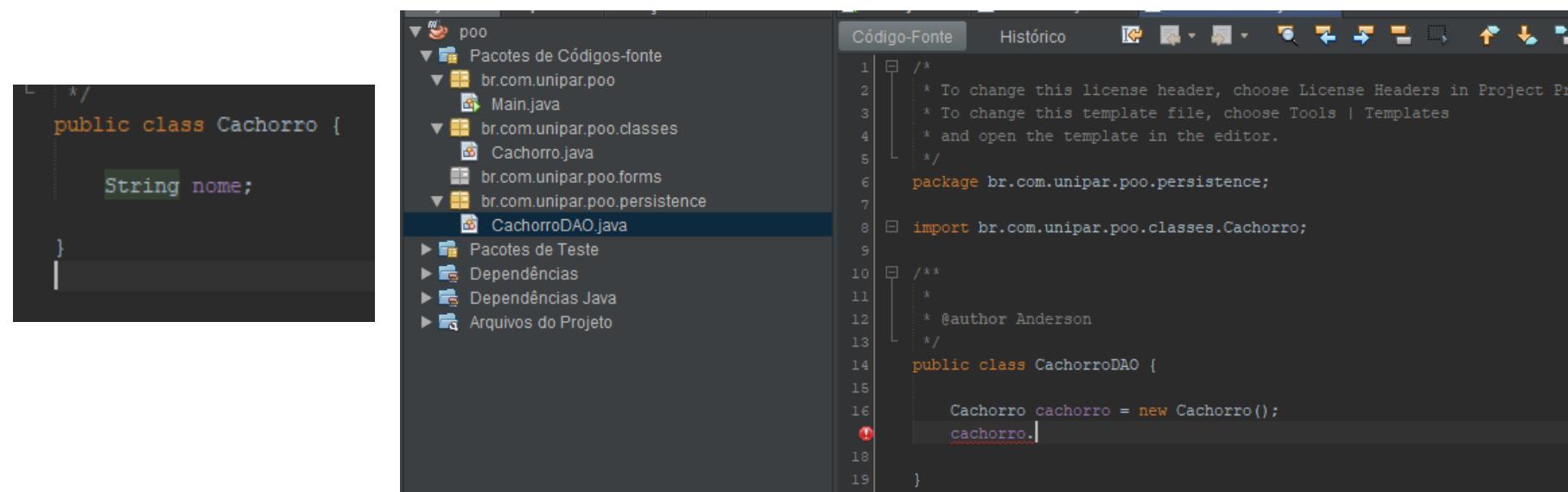
```
/*
public static void main(String[] args) {
    Cachorro dog = new Cachorro();
    dog.nome = "bob";
}
```

```
/*
public static void main(String[] args) {
    Cachorro dog = new Cachorro();
    dog.nome = "bob";
}

Cachorro.nome has protected access in Cachorro
--(Alt-Enter mostra dicas)
```

# Modificadores de Acesso a Nível de Atributos e Métodos(Membros) - default

**default (sem modificador explícito)** torna um membro acessível apenas para classes do mesmo pacote.



The screenshot shows a Java development environment with a project tree on the left and a code editor on the right.

**Project Tree:**

- poo
- Pacotes de Códigos-fonte
  - br.com.unipar.poo
    - Main.java
  - br.com.unipar.poo.classes
    - Cachorro.java
  - br.com.unipar.poo.forms
  - br.com.unipar.poo.persistence
    - CachorroDAO.java
- Pacotes de Teste
- Dependências
- Dependências Java
- Arquivos do Projeto

**Code Editor (Código-Fonte tab):**

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package br.com.unipar.poo.persistence;

import br.com.unipar.poo.classes.Cachorro;

/**
 * @author Anderson
 */
public class CachorroDAO {

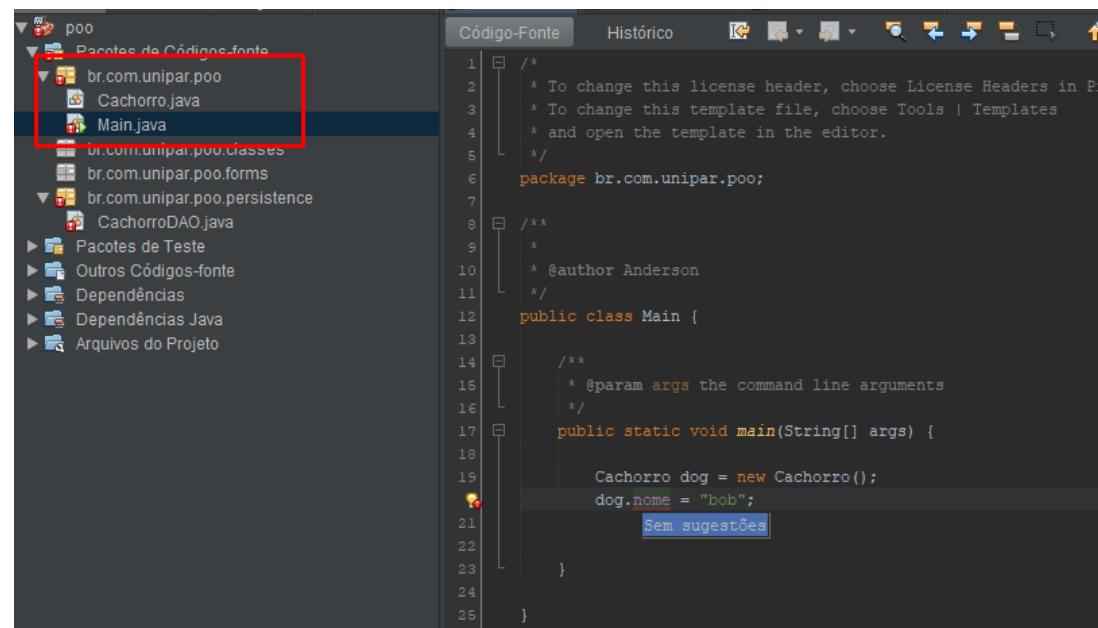
    Cachorro cachorro = new Cachorro();
    cachorro.

}
```

# Modificadores de Acesso a Nível de Atributos e Métodos(Membros) - **private**

**private** torna um membro acessível apenas para a classe que o contém.

```
public class Cachorro {  
    private String nome;  
}
```



The screenshot shows a Java development environment. On the left, the project tree displays a package named 'poo' containing a 'br.com.unipar.poo' folder which holds 'Cachorro.java' and 'Main.java'. Both files are highlighted with a red box. The right side shows the code editor with the 'Main.java' file open. The code is as follows:

```
/*  
 * To change this license header, choose License Headers in Project  
 * Properties; you can find them in the file menu.  
 */  
package br.com.unipar.poo;  
  
/**  
 * @author Anderson  
 */  
public class Main {  
    /**  
     * @param args the command line arguments  
     */  
    public static void main(String[] args) {  
        Cachorro dog = new Cachorro();  
        dog.nome = "bob";  
    }  
}
```

In the code editor, the line 'dog.nome = "bob";' is selected, and a tooltip 'Sem sugestões' (No suggestions) appears at the bottom of the editor window.

**Objetos não devem alterar os estados dos outros ou seus atributos.**

**Exemplos:**

**Pessoa e um Carro.**

**Pessoa e um Porta.**



**Professor, mas se um objeto não pode alterar diretamente os atributos e o estado de outro objeto, como fazemos nosso código abaixo?**

```
/*  
 *  
 */  
public class Cachorro {  
  
    String nome;  
  
}
```

```
public class Main {  
  
    /**  
     * @param args the command line arguments  
     */  
    public static void main(String[] args) {  
  
        Cachorro dog = new Cachorro();  
        dog.nome = "bob";  
  
    }  
  
}
```



**Professor, mas se um objeto não pode alterar diretamente os atributos e o estado de outro objeto, como fazemos nosso código abaixo?**

Através dos métodos acessores.



# Métodos Acessores(Get e Set)

Também conhecidos como getter e setters, são métodos utilizados para que seja possível acessar e modificar os valores de variáveis com modificador de acesso private.



# Getters

O propósito de um getter é obter o valor de uma variável declarada como private e permitir sua leitura a partir de outra classe.



# Setters

**Um setter é um método que permite modificar o valor de um atributo da classe que não seja acessível diretamente por ser privado.**



# Getters e Setters

```
1  /*
2  public class Cachorro {
3      //Atributo privado
4      private String nome;
5
6      //No caso do setter o retorno do metodo é void ou seja sem retorno pois
7      //usamos o metodo para setar o valor a um atributo privado
8      //Acessor do metodo + Tipo de Retorno + nomeMetodo(Tipo e Parametro de Entrada)
9      public void setNome(String nome) {
10          this.nome = nome;//this identifica o contexto do que está sendo setado
11              //nesse caso o nosso contexto é a própria classe
12              //fazendo com que this.nome seja diferente da variavel de entrada nome
13      }
14
15      //No caso do getter como apenas queremos buscar o valor do atributo privado não temos
16      //parametros de entrada no metodo mas temos o tipo de retorno
17      //nesse caso como o atributo se trata de uma string
18      //o tipo de retorno é uma string
19      //Acessor do metodo + Tipo de Retorno + nomeMetodo()
20      public String getNome() {
21          return nome; //return é um comando reservado que identifica no java qual será o retorno do metodo
22      }
23  }
```



# Vamos codificar.

- Criar uma
- Criar uma



# Vamos codificar.



# Programação Orientada a Objetos

**Aula: Encapsulamento,  
Construtores e Destrutores.**



Prof. Anderson Augusto Bosing

# Encapsulamento

**Classes (e seus objetos)** encapsulam, isto é, contêm seus atributos e métodos. Os atributos e métodos de uma classe (e de seu objeto) estão intimamente relacionados.

Os objetos podem se comunicar entre si, mas eles em geral não sabem como outros objetos são implementados — os detalhes de implementação permanecem ocultos dentro dos próprios objetos. (Deitel, 2016).



# Encapsulamento

Encapsular é tornar o código dentro de uma classe acessível ou inacessível para objetos fora da classe. A lógica que suporta este comportamento é que cada classe deve ter um significado e por si só deve descrever o comportamento de um objeto.



# A palavra chave this

Palavra reservada do Java que referencia atributos e métodos da própria classe.



# Objetos e Mais Objetos

É comum que sejam criadas classes para representar objetos do mundo real, e tão comum quanto isso são que os atributos das classes também seja identificados como outras classes.

Vamos a um exemplo.

Se nós temos uma classe que representa um carro, esse carro tem uma marca que tem seus próprios atributos. E essa marca consequentemente pode possuir um endereço que também possui seus próprios atributos.

Mas professor, como isso seria codificado?



# Objetos e Mais Objetos

Vamos a outro exemplo.

Um banco possui muitas agencias e estas agencias possuem muitos correntistas, sendo contas correntes ou contas poupanças. Cada correntista possui um endereço para onde deve ser enviado o cartão.

Mas e professor, como isso seria codificado?



# ArrayList

O Java ArrayList, é, basicamente, um **array dinâmico** que encontramos no `java.util` — um pacote que contém uma grande variedade de recursos, como estruturas de coleções, modelos de data e hora, recursos para internacionalização, entre muitas outras facilidades para o desenvolvimento de aplicações em Java.

Esses **arrays redimensionáveis** são muito úteis quando utilizados para implementações em que precisamos manipular listas.



# ArrayList

A dinamicidade do recurso possibilita ao desenvolvedor a criação de coleções — arrays, classes e objetos — sem precisar se preocupar com o redimensionamento dos vetores. Caso algum haja necessidade de uma posição adicional em um array, o **ArrayList** realiza a operação de maneira autônoma.



# Quais as principais características da classe ArrayList Java?

Entre as interfaces e classes das estruturas disponibilizados para uso durante o desenvolvimento de aplicações em Java, certamente, o ArrayList é uma das principais delas. Sua característica de construção dinâmica de arrays possibilita manipulá-las com o uso de métodos para adicionar ou retirar objetos.



# Quais as diferenças entre array e arrayList em java?

Quando utilizamos o array em Java, estamos trabalhando com tamanhos fixos de coleções. Para inserirmos algum elemento naquele grupo de objetos, precisamos criar outro que contemple aquele novo tamanho, com determinado número de posições a mais — isso não acontece com o ArrayList.



# Quais as diferenças entre array e arrayList em java?

É muito comum a confusão de que arrays e ArrayList são a mesma coisa, mas saiba que eles são muito diferentes. O ArrayList não é um array padrão, ele apenas utiliza essa funcionalidade para realizar o armazenamento dos objetos contidos nas listas manipuladas por ele. Nem ao menos os atributos desse array, que o ArrayList utiliza, é possível ser acessado durante o processamento.



# Quais as diferenças entre array e arraylist em java?

Como a classe ArrayList é um agrupamento dinâmico de objetos, isso quer dizer que podemos adicionar ou retirar elementos de, por exemplo, uma lista, sem que seja necessário criar uma nova, mantendo a original com um número diferente dos objetos ali contidos.



# Quais as diferenças entre array e arraylist em java?

Desde o momento em que criamos um array, seu tamanho não pode ser mudado:

```
String [] meuStringArray = new String[3];
```

Repare que, nesse caso, independentemente de termos objetos ou não dentro de nosso novo array, sempre teremos para ele três posições disponíveis, nunca mais do que isso.



# Quais as diferenças entre array e arrayList em java?

O que é bem diferente para o ArrayList.

Aqui, trazemos um exemplo prático da aplicação:

```
List lista = new ArrayList();
lista.add("Pessoa 1");
lista.add("Pessoa 2");
lista.add("Pessoa 3");
```

Isso porque a classe ArrayList é independente do número de objetos contidos nela. Podemos aumentar ou diminuir seu tamanho, apenas inserindo ou retirando mais objetos.



# **Quais os principais métodos usados com a classe ArrayList?**

- **new ArrayList():** cria um novo ArrayList. Por padrão, essa classe tem a capacidade inicial de 10 elementos;
- **add(item):** é o método utilizado para adicionar novos elementos ao ArrayList. Os elementos são colocados no fim da lista, por padrão;
- **remove(posição):** remove um item de determinada posição na lista criada;
- **set(item, lista):** usamos para definir um elemento em determinado index;
- **get(item):** retorna o objeto ligado a determinado índice;
- **clear():** limpa todos os elementos contidos na lista.



# Vamos Praticar?

- . **Vamos praticar, vamos criar uma agenda de contatos com endereço e telefones para cada contato.**



# Vamos Praticar?

- . Vamos praticar, vamos criar uma agenda de consultas para uma clinica médica.



# Vamos Praticar?

- . **Vamos praticar, vamos criar um sistema para o cinema de toledo. Onde devem ser gerenciadas as cadeiras, salas de cinema e a ocupação delas pelos clientes.**



# Por que Utilizar o Encapsulamento?

**Organização de código.**

**Quando criamos classes, cada uma delas possui uma função específica.** Ou seja, elas descrevem um objeto específico, sendo assim, classes coesas não realizam várias funções.

**Manutenção de código.**

**Depois de pronto, todo código, principalmente os mais extensos, são propensos a sofrer manutenções.**

**Com o encapsulamento, isso passa a ser mais fácil,** uma vez que, com a devida proteção de acesso aos dados, a pessoa programadora achará mais rápido algum ponto onde o código precisa ser melhorado.



# Por que Utilizar o Encapsulamento?

## Reuso de Código.

Com o encapsulamento, o programa terá mais chances de ter o código reutilizado em outros projetos, poupando bastante tempo da equipe de desenvolvimento.

## Simplificação da codificação.

O encapsulamento transforma a implementação de alguns códigos em uma espécie de caixa preta. Na prática, isso significa que as classes externas não precisam acessar alguns dados de forma direta. Assim, o desenvolvimento dos sistemas passa a ficar simplificado e acelerado.



# Construtores

**Construtores são os métodos responsáveis por criar uma instância da classe em memória, ou seja instanciar um objeto em memória.**

**O método construtor é muito semelhante a um método comum, porém ele se difere dos demais métodos por alguns pontos bem específicos e importantes para a linguagem Java.**



# Construtores

**Em primeiro lugar o método construtor deve possuir o mesmo nome da classe, sendo assim, é o único método por padrão Java que será nomeado com a primeira letra em maiúscula.**

**Outro ponto importante é que um construtor nunca terá um tipo de retorno, poderá ser do tipo private, public, protected ou default, mas nunca terá um tipo de retorno nem que ele seja do tipo void.**



# Construtores

## Classe sem construtor explícito

```
/*
public class Animal {

    private double peso;
    private String grupo;

    public double getPeso() {
        return peso;
    }

    public void setPeso(double peso) {
        this.peso = peso;
    }

    public String getGrupo() {
        return grupo;
    }

    public void setGrupo(String grupo) {
        this.grupo = grupo;
    }
}
```

```
public class Main {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {

        Animal animal = new Animal();

    }
}
```



# Construtores

## Classe com construtor explícito

```
/*
public class Animal {

    public Animal() {
    }

    private double peso;
    private String grupo;

    public double getPeso() {
        return peso;
    }

    public void setPeso(double peso) {
        this.peso = peso;
    }

    public String getGrupo() {
        return grupo;
    }

    public void setGrupo(String grupo) {
        this.grupo = grupo;
    }
}
```

```
public class Main {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {

        Animal animal = new Animal();
    }
}
```



# Construtores

## Classe com construtor explícito e Parâmetros de entrada

```
public class Animal {  
  
    public Animal(double peso, String grupo) {  
        this.peso = peso;  
        this.grupo = grupo;  
    }  
  
    private double peso;  
    private String grupo;  
  
    public double getPeso() {  
        return peso;  
    }  
  
    public void setPeso(double peso) {  
        this.peso = peso;  
    }  
  
    public String getGrupo() {  
        return grupo;  
    }  
  
    public void setGrupo(String grupo) {  
        this.grupo = grupo;  
    }  
}
```

```
public class Main {  
  
    /**  
     * @param args the command line arguments  
     */  
    public static void main(String[] args) {  
  
        Animal animal = new Animal(2, "Hunter");  
  
    }  
}
```



# Destruidores

**Antes de começarmos a falar sobre destrutores, primeiro eu preciso dizer que o Java não possui métodos destrutores.**



# O que é Garbage Collector?

O garbage collection é o processo pelo qual os programas Java executam o gerenciamento automático de memória. Os programas Java compilam para bytecode que pode ser executado em um Java Virtual Machine (JVM).

Quando os programas Java são executados na JVM, os objetos são criados no heap, que é uma parte da memória dedicada ao programa. Eventualmente, alguns objetos não serão mais necessários. O garbage collector localiza esses objetos não utilizados e os exclui para liberar memória.



# O mais próximo dos Destruidores

O método `finalize`, é um método `protected` definido na classe `Object`, e que pode ser redefinido pelo programador em sua classe. Ele é executado automaticamente quando a área do objeto da classe que a contém estiver para ser liberada pelo coletor.



# O mais próximo dos Destruidores

```
public class Animal {  
  
    public Animal(double peso, String grupo) {  
        this.peso = peso;  
        this.grupo = grupo;  
    }  
  
    private double peso;  
    private String grupo;  
  
    public double getPeso() {  
        return peso;  
    }  
  
    public void setPeso(double peso) {  
        this.peso = peso;  
    }  
  
    public String getGrupo() {  
        return grupo;  
    }  
  
    public void setGrupo(String grupo) {  
        this.grupo = grupo;  
    }  
  
    @Override  
    protected void finalize() throws Throwable {  
        System.out.println("Finalizar");  
        super.finalize();  
    }  
}
```

```
public static void main(String[] args) {  
  
    Animal animal = new Animal(2, "Hunter");  
    animal = null;  
    System.gc();  
}
```

```
Building Teste 1.0  
-----[ jar ]-----  
--- exec-maven-plugin:1.2.1:exec (default-cli) @ Teste ---  
| Finalizar  
|-----  
BUILD SUCCESS  
-----
```



# Sobrecarga de Métodos

A sobrecarga de métodos (overload) consiste basicamente em criar variações de um mesmo método, ou seja, a criação de dois ou mais métodos com nomes totalmente iguais em uma classe mas que possuam assinaturas diferentes podendo ter retornos diferentes, parâmetros de entradas diferentes.

A assinatura de um método em Java é composta pelo nome do método e pela lista de tipos dos parâmetros que o método aceita. Em outras palavras, a assinatura de um método consiste no nome do método junto com os tipos dos parâmetros em ordem específica.

```
/*
public class Calculadora {

    public int soma(int num1, int num2) {
        return num1 + num2;
    }

    public int soma(int num1, int num2, int num3) {
        return num1 + num2 + num3;
    }

    public String soma(double num1, double num2) {
        return String.valueOf(num1 + num2);
    }
}
```



# Vamos Praticar?

- **Lista de Exercícios em Orientação a Objetos com Java.**



# Programação Orientada a Objetos

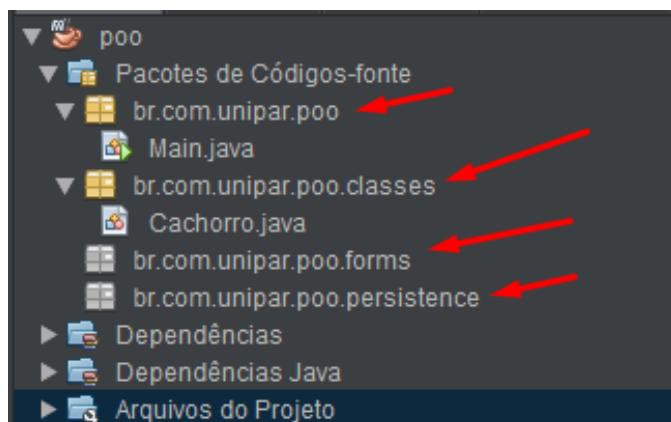
Aula: Packages.



Prof. Anderson Augusto Bosing

# Pacotes(Packages)

Pacotes java são utilizados para organizar as classes da sua aplicação. Um programa pode, facilmente, ter mais de centenas de classes. Então é muito importante que todos os seus componentes fiquem organizados. Podemos pensar nos pacotes como uma pasta do seu sistema de arquivos.



Nome	Data de modificação	Tipo	Tamanho
classes	05/04/2022 18:30	Pasta de arquivos	
forms	05/04/2022 18:30	Pasta de arquivos	
persistence	05/04/2022 18:30	Pasta de arquivos	
Main.java	05/04/2022 18:29	Arquivo Fonte Java	1 KB

# Categorias de pacotes em Java

Existem dois tipos de pacotes em java:

- Pacote definido pelo usuário (criar seu próprio pacote)
- Pacotes integrados(built in) que são pacotes da interface de programação de aplicativos Java que são os pacotes da API Java, por exemplo. como swing, util, net, io, AWT, lang, javax, etc.



# Pacotes integrados(built in)

Pactos Básicos:

**java.lang**- classes fundamentais do java(Boolean, Double, Float, Long, Math, String).

**java.util** – classes utilitárias do java(List, ArrayList, Arrays, Date, Timer, Timezone).

**java.io** – classes de para entrada e saída(File, FileInputStream, FileOutputStream)

**java.net** - Classes para uso de comunicação de rede(TCP/IP, HTTP, PROXY, SOCKET).

**java.sql** - Classes para utilização e acesso de conexões JDBC responsaveis pelo acesso ao banco de dados(Connection, Driver, ResultSet, PreparedStatement).

**java.awt** - Classes para utilização e criação de interfaces desktop em java(Button, Checkbox, Dialog, Event, Font, Frame, Label, Menu, TextField, TextArea).



# Pacotes integrados(built in)

Pactos Básicos:

**java.text** - Classes para formatação e transformação de texto(SimpleDateFormat, NumberFormat, DecimalFormat).

**java.math** - Classes para lidar com numeros precisos(Arredondar, randomizar um numero, abs, max, min).

**java.time** - Classes para utilização em dados que envolvem hora ou tempo.(Year, LocalTime, ZoneId, DayOfWeek, Month).

**java.util.zip** - Classes para lidar com arquivos ZIP's.

**javax.swing** - Classes para utilização e criação de interfaces desktop em java(Button, Checkbox, Dialog, Event, Font, Frame, Label, Menu, TextField, TextArea).



# Pacotes integrados(built in)

<https://docs.oracle.com/javase/8/docs/api/>



# Pacotes Definidos Pelo Usuário

Vamos praticar, pensando que criaremos um novo jogo FPS, quais classes seriam criadas ?



# Pacotes Definidos Pelo Usuário

Como podemos organizar essas classes em pacotes?

**Usuario**

**Pistola**

**Submetralhadora**

**Rifle**

**Armas**

**Pesada**

**Utilitário**

**Mapa**

**Personagem**

**Fila**

**Janela**

**BotaoJogar**

**MenuPrincipal**

**MenuArma**

**MenuUsuario**

**MenuTab**

**MenuKick**



# Pacotes Definidos Pelo Usuário

Vamos praticar, pensando que criaremos uma nova netflix, quais classes seriam criadas ?



# Pacotes Definidos Pelo Usuário

**Como podemos organizar essas classes em pacotes?**

**Filme**

**Serie**

**Anime**

**Novela**

**Documentario**

**Janela**

**Principal**

**Miniatura**

**Video**

**BotaoPlay**

**BotaoPause**

**BotaoVoltar**



# Motivos para Usar Packages

- Separação e Organização das classes.
- Definição das responsabilidades das classes.
- Evitar Conflitos de nomes em caso de classes com nomes idênticos.



# Convenções em Pacotes

- SEMPRE em Lower case(letra minúscula)
- Domínio da aplicação de forma reversa + Nome da Aplicação
- Exemplos:
  - br.unipar.nomeaplicacao
  - br.com.pratidionaduzzi.nomeaplicacao



# Programação Orientada a Objetos

**Aula: Passagem de  
parâmetros, Polimorfismo e  
Herança.**



Prof. Anderson Augusto Bosing

# Passagem de parâmetros

- Por valor
- Por referencia.



# **Herança**

**Herança é uma forma de reutilização de software, onde uma nova classe é criada absorvendo atributos e métodos de uma classe existente.**

**Esta nova classe pode ter características mais específicas ou modificadas em comparação com a classe antiga/absorvida.**

**Com a herança, o tempo de desenvolvimento de um software é reduzido e a depuração é facilitada.**

**A classe nova é chamada de SUBCLASSE, já a classe antiga, que é absorvida pela nova, é chamada de SUPERCLASSE.**



# Herança

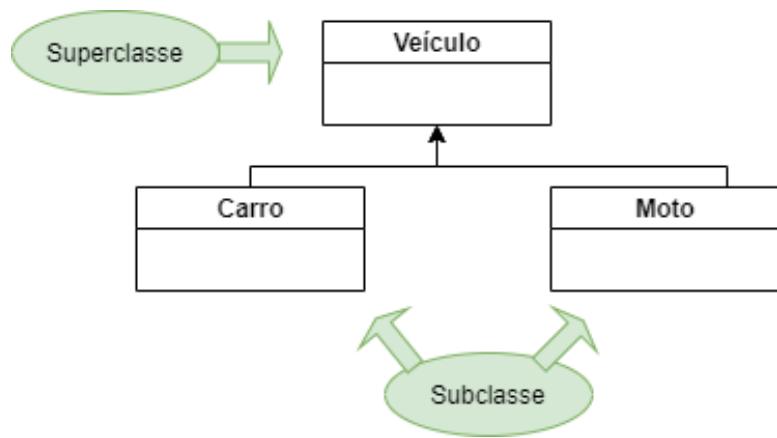
A herança pode se dar em vários níveis, formando uma hierarquia.

A classe imediatamente superior é uma SUPERCLASSE direta.

Uma classe que não seja imediatamente superior é uma SUPERCLASSE indireta.

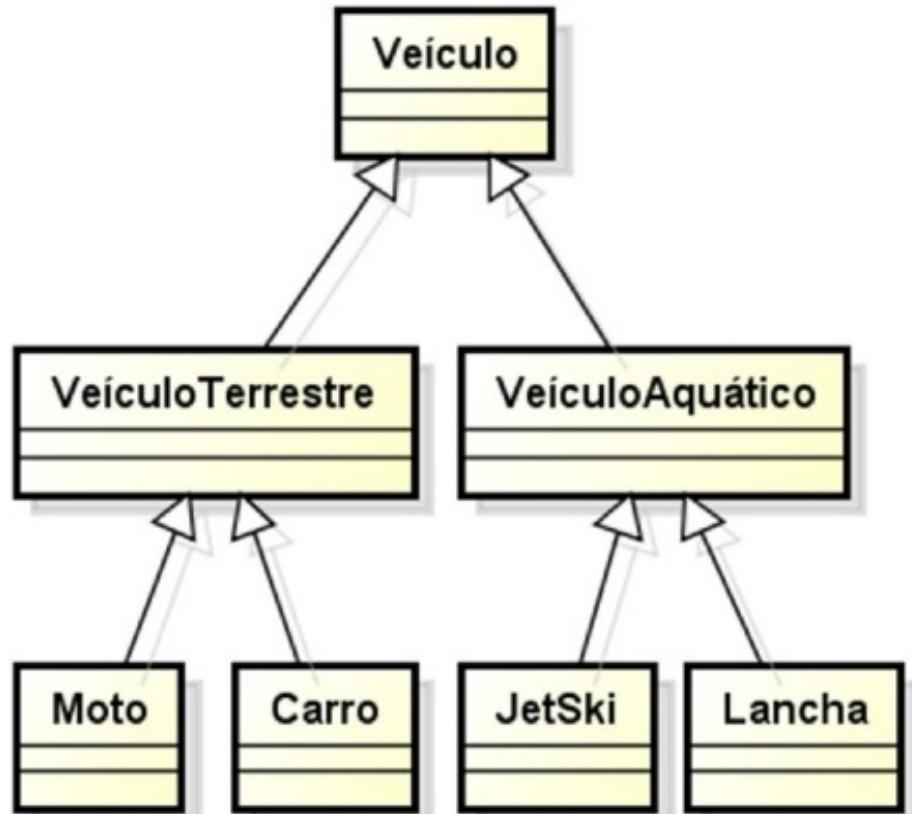


# Herança Exemplo



A classe veiculo é uma **SUPERCLASSE** direta das classes carro e moto que são subclasses de veiculo.

# Herança Exemplo



Alguns apontamentos:

- 1-A Moto é um VeículoTerrestre, mas também é um Veículo.
- 2-A Lancha é um Veículo, porém não é VeículoTerrestre.
- 3-Moto e JetSki são Veículos, porém um é VeículoTerrestre e o outro é VeículoAquático.
- 4-VeículoAquático e VeículoTerrestre compartilham dados da classe veículo.

# Herança

Então, para criarmos uma herança em nosso sistema, utilizamos a palavra reservada **extends**.



# Herança Exemplo

```
/*
 *
 * @author Anderson
 */
public class Veiculo {

    public String marca;
    public String modelo;

    public Veiculo() {
    }

    public Veiculo(String modelo, String marca) {
        this.modelo = modelo;
        this.marca = marca;
    }
}
```

```
/*
 *
 * @author Anderson
 */
public class Carro extends Veiculo {

    public int porta;

    // Chamada implicita para o construtor de Veiculo
    public Carro() {
    }

    // Chamada explicita para o construtor de Veiculo
    public Carro(String marca, String modelo, int porta) {
        super(modelo, marca);
        this.porta = porta;
    }
}
```



# Herança

Para compreender melhor o código acima, é importante conhecer as referências `this` e `super`.

**Referência `this`:** Referência à membros do objeto corrente.

**Referência `super`:** Referência à membros da superclasse, sendo também utilizado para chamar o construtor: Subclasse chama o construtor da superclasse; Primeira instrução no construtor da subclasse.



# Herança Exemplo

```
Pessoa.java | Medico.java | Enfermeira.java | Paciente.java
1 import java.util.Date;
2
3 public class Pessoa {
4     int idCadastro;
5     String nome;
6     String cpf;
7     Date dataNascimento;
8
9     public Pessoa(int idCadastro, String nome, String cpf, Date dataNascimento) {
10        this.idCadastro = idCadastro;
11        this.nome = nome;
12        this.cpf = cpf;
13        this.dataNascimento = dataNascimento;
14    }
15 }
```

```
Pessoa.java | Medico.java | Enfermeira.java | Paciente.java
1 import java.util.Date;
2
3 public class Enfermeira extends Pessoa {
4     public Enfermeira(int idCadastro, String nome, String cpf, Date dataNascimento) {
5         super(idCadastro, nome, cpf, dataNascimento);
6     }
7     private String coren;
8     private String siglaEstadoCoren;
9     private char setorEnfermaria;
10 }
11 }
```

```
Pessoa.java | Medico.java | Enfermeira.java | Paciente.java
1 import java.util.Date;
2
3 public class Medico extends Pessoa {
4
5     public Medico(int idCadastro, String nome, String cpf, Date dataNascimento) {
6         super(idCadastro, nome, cpf, dataNascimento);
7     }
8     private String crm;
9     private String siglaEstadoCRM;
10    private String especialidade;
11 }
```

```
Pessoa.java | Medico.java | Enfermeira.java | Paciente.java
1 import java.util.Date;
2
3 public class Paciente extends Pessoa {
4     public Paciente(int idCadastro, String nome, String cpf, Date dataNascimento) {
5         super(idCadastro, nome, cpf, dataNascimento);
6     }
7     private boolean internado;
8     private char setor;
9     private short leito;
10    private String diagnostico;
11 }
```

# Vamos Praticar?

Criaremos a Classe Professor e Aluno.

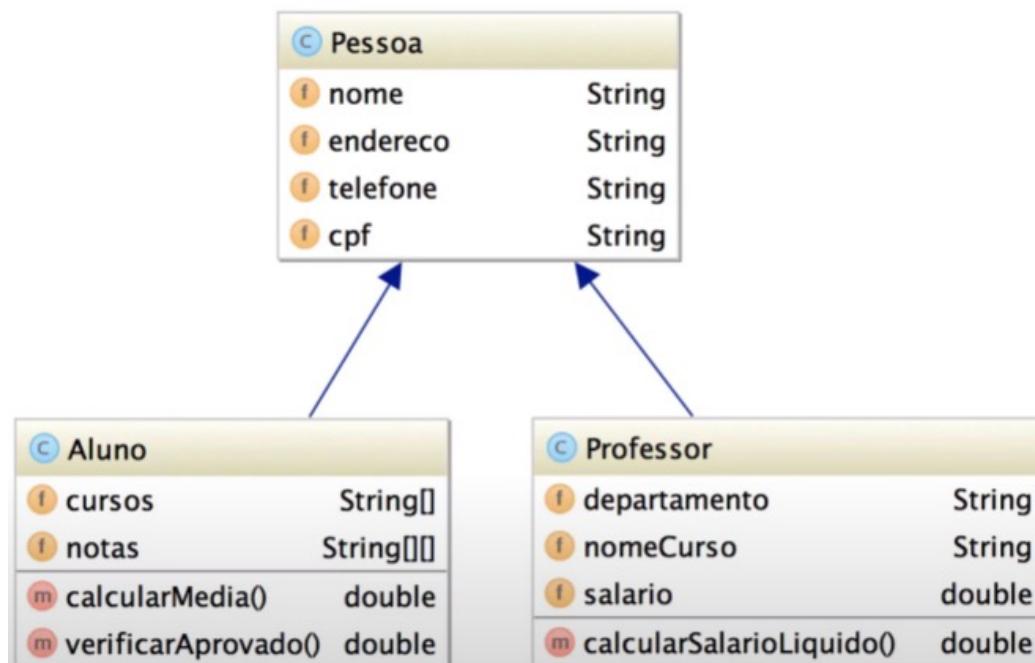
C Aluno	
f nome	String
f endereço	String
f telefone	String
f cpf	String
f cursos	String[]
f notas	String[][]
m calcularMedia()	double
m verificarAprovado()	double

C Professor	
f nome	String
f endereço	String
f telefone	String
f cpf	String
f departamento	String
f nomeCurso	String
f salario	double
m calcularSalarioLiquido()	double



# Vamos Praticar?

Criaremos a Classe Professor, Aluno e Pessoa.



# Herança - Importante

- Uma classe só pode herdar de uma outra classe (herança simples).
- Caso não seja declarada herança, a classe herda da classe Object (Ela define o método `toString()`, que retorna a representação em String do objeto, qualquer subclasse pode sobrescrever o método `toString()` para retornar o que ela deseja.)

Veja os demais métodos da classe Object em:

<https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html>

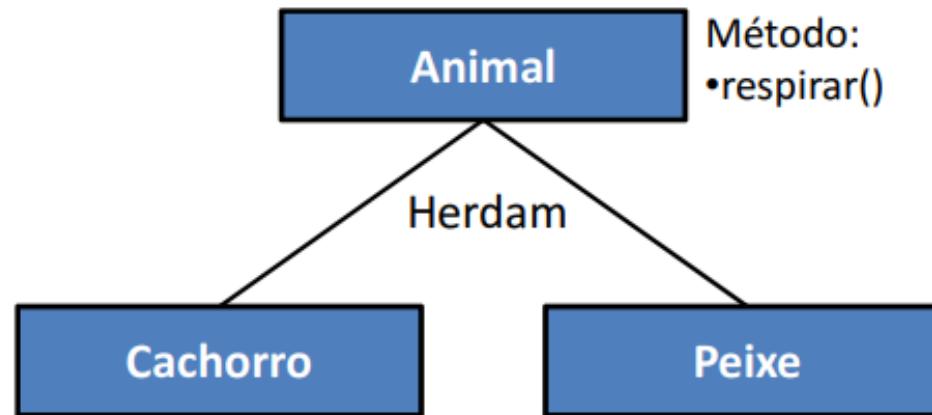


# Polimorfismo

- Significa “várias formas”.
- Habilidade de um mesmo tipo de objeto poder realizar ações diferentes ao receber uma mesma mensagem.
- Criação de múltiplas classes com os mesmos métodos e propriedades, mas com funcionalidades e implementações diferentes.



# Polimorfismo



Cachorro e Peixe respiram da mesma forma?

# Polimorfismo Sobrescrita de Métodos

Importante também falarmos sobre sobrescrita de métodos. A sobrescrita (ou override) está diretamente relacionada à orientação a objetos, mais especificamente com a herança. Com a sobrescrita, conseguimos especializar os métodos herdados das superclasses, alterando o seu comportamento nas subclasses por um mais específico.

A sobrescrita de métodos consiste basicamente em criar um novo método na classe filha contendo a mesma assinatura e mesmo tipo de retorno do método sobrescrito.

Os métodos podem ser sobrescritos, o que é diferente de sobrecarga por terem a mesma assinatura e o tipo de retorno.



# Vamos Praticar?

**Criaremos a Classe Funcionário, Diretor, Gerente e Colaborador.**

**Além de uma classe para calcular o valor de bonificação de cada uma dessas pessoas.**



# Chegou a sua vez?

- . Lista de Exercícios de Herança com Java disponível no classroom.
- . <https://docs.google.com/document/d/1e2w2AJNC-CdG47xMq5DiemIO1pptCXZnChLrUOLkb4U/edit?usp=sharing>



# Programação Orientada a Objetos

**Aula: Interfaces,  
Classes Abstratas e  
ENUM's.**



Prof. Anderson Augusto Bosing

# Classes Abstratas

Quando pensamos em um tipo de classe, supomos que os programas criam objetos desse tipo. Às vezes é útil declarar as classes — chamadas classes abstratas — para as quais você nunca pretende criar objetos. Como elas só são utilizadas como superclasses em hierarquias de herança, são chamadas superclasses abstratas. Essas classes não podem ser utilizadas para instanciar objetos, porque, como veremos mais adiante, classes abstratas são incompletas. As subclasses devem declarar as “partes ausentes” para que se tornem classes “concretas”, a partir das quais você pode instanciar objetos. Do contrário, essas subclasses também serão abstratas.



# Classes Abstratas

```
[-] /**
 * 
 * @author andersonbosing
 */
public abstract class Funcionario {

    public abstract Double retornaBonificacao();

}
```

```
5  * @author andersonbosing
6  */
7  public class Desenvolvedor extends Funcionario{
8
9      @Override
10     public Double retornaBonificacao() {
11         return 1500.00;
12     }
13
14 }
15 }
```



# Classes Abstratas

**Impossibilita a instanciação direta da classe.**

**Possibilita herança de código preservando comportamento.**

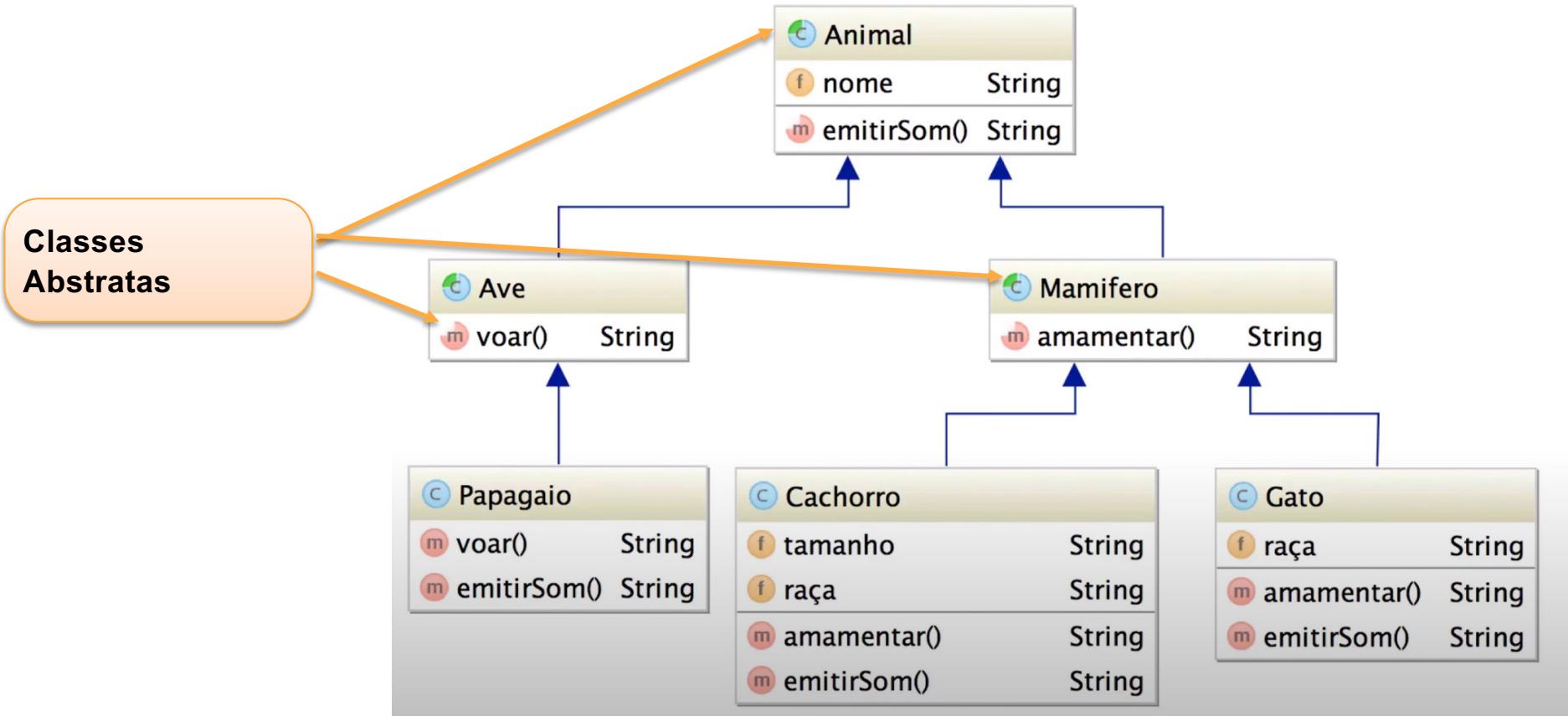
- O que é genérico é herdado.
- O que é específico é implementado nas subclasses.

**Adia especificidades de implementação para**

**Importância de Classes Abstratas Adia especificidades de implementação para fases posteriores de desenvolvimento.**



# Classes Abstratas



# Classes Abstratas

## Exemplos:

- Notas Entrada, Saída;
- Impostos;
- Pessoa, Aluno, Professor;



# Interface

A interface é um recurso muito utilizado em Java, bem como na maioria das linguagens orientadas a objeto, para “obrigar” a um determinado grupo de classes a ter métodos ou propriedades em comum para existir em um determinado contexto, contudo os métodos podem ser implementados em cada classe de uma maneira diferente. Pode-se dizer, a grosso modo, que uma interface é um contrato que quando assumido por uma classe deve ser implementado.



# Interface

**Uma interface não contém atributos, apenas assinaturas de métodos.**

**Não se pode instanciar objetos de interfaces, mas sim de classes que implementam as interfaces.**



# Interface

```
/*
public interface Banco {

    boolean saque(Conta conta, double valor);
    boolean deposito(Conta conta, double valor);
    void extrato(Conta conta);

}
```

```
* @author andersonbosing
*/
public class BancoBrasil implements Banco {

    @Override
    public boolean saque(Conta conta, double valor) {
        throw new UnsupportedOperationException("Not supported yet."); //To change this template, choose Tools | Templates.
    }

    @Override
    public boolean deposito(Conta conta, double valor) {
        throw new UnsupportedOperationException("Not supported yet."); //To change this template, choose Tools | Templates.
    }

    @Override
    public void extrato(Conta conta) {
        throw new UnsupportedOperationException("Not supported yet."); //To change this template, choose Tools | Templates.
    }

}
```



# Interface - Múltiplas Implementações

```
* @author andersonbosig
*/
public class BancoBrasil implements Banco, Comparable {

    @Override
    public boolean saque(Conta conta, double valor) {
        throw new UnsupportedOperationException("Not supported yet."); //To change
    }

    @Override
    public boolean deposito(Conta conta, double valor) {
        throw new UnsupportedOperationException("Not supported yet."); //To change
    }

    @Override
    public void extrato(Conta conta) {
        throw new UnsupportedOperationException("Not supported yet."); //To change
    }

    @Override
    public int compareTo(Object o) {
        throw new UnsupportedOperationException("Not supported yet."); //To change
    }

}
```

<https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html>



# Interface

**Exemplos de Utilização:**

- **BasicDAO, FuncionarioDAO.**
- **BasicControllerRest.**
- **Pessoa implementando Comparable**



# Classes Enumeradas(Enum's)

São tipos de campos que consistem em um conjunto fixo de constantes, sendo como uma lista de valores pré-definidos. Na linguagem de programação Java, pode ser definido um tipo de enumeração usando a palavra chave enum.

Todos os tipos enums implicitamente estendem a classe `java.lang.Enum`, sendo que o Java não suporta herança múltipla, não podendo estender nenhuma outra classe.



# Características dos Tipos Enum

**Em relação às propriedades é preciso tomar os seguintes cuidados:**

**As instâncias dos tipos enum são criadas e nomeadas junto com a declaração da classe, sendo fixas e imutáveis (o valor é fixo).**

**Os construtores são obrigatoriamente privados mesmo embora não seja necessário explicitar isso.**

**Não é permitido criar novas instâncias com a palavra chave new.**

**Acima de tudo são classes então podem ter construtores, atributos e métodos.**

**Seguindo a convenção, por serem objetos constantes e imutáveis, os nomes declarados recebem todas as letras em MAIÚSCULAS por convenção de código.**



# Enum - Exemplos

```
public enum SexoEnum {  
    MASCULINO,  
    FEMININO,  
    NAO_DECLARADO;  
}
```

```
public enum SexoEnum {  
  
    MASCULINO("Masculino", "M"),  
    FEMININO("Feminino", "F"),  
    NAO_DECLARADO("Não Declarado", "NA");  
  
    private String descricao;  
    private String sigla;  
  
    private SexoEnum(String descricao, String  
        this.descricao = descricao;  
        this.sigla = sigla;  
    }  
  
    public String getDescricao() {  
        return descricao;  
    }  
  
    public String getSigla() {  
        return sigla;  
    }  
  
    public void setSigla(String sigla) {  
        this.sigla = sigla;  
    }  
}
```



# Enum - Exemplos

**Categorias de Filmes.**

**Sexo**

**Classes Medicamentos**

**Estado Civil**

**Genero**

**Dias da Semana**



# Chegou a sua vez..

- **Lista de Exercícios de Herança, Enum, Interfaces com Java disponível no classroom.**
- <https://docs.google.com/document/d/1HSGzLcxrGI4JP8oWtqXXjRLAkwiVzVndZb41S5Lik0M/edit?usp=sharing>



# Programação Orientada a Objetos

**Aula: Tratamento de  
Exceções em Java.**



# O que são exceções?



# O que são exceções?

**Uma exceção é uma condição anormal que altera ou interrompe o fluxo de execução.**

**Indicação de um problema que ocorre durante a execução de um programa (DEITEL, 2005).**

**Podem ser causadas por diversas condições:**

- Erros sérios de hardware.
- Erros simples de programação.
- Erros de divisão por zero.
- Valores fora de faixa.
- Valores de variáveis.
- Erro na procura/abertura de um arquivo (entrada/saída).
- Falha na memória.



# Alguns problemas que geram exceções

- Pelo usuário:
  - Divisão por zero
  - Caracteres inválidos
- Pelo código:
  - Senha de acesso ao banco de dados errada
  - Tentativa de abrir um arquivo inexistente
  - Manipulação de variável com valor null (sem objeto)



# Tratamento de Exceções

- Mecanismo de identificar e tratar uma exceção.
- Permite que o programa continue executando.
- Favorece o desenvolvimento de aplicativos tolerante a falhas.

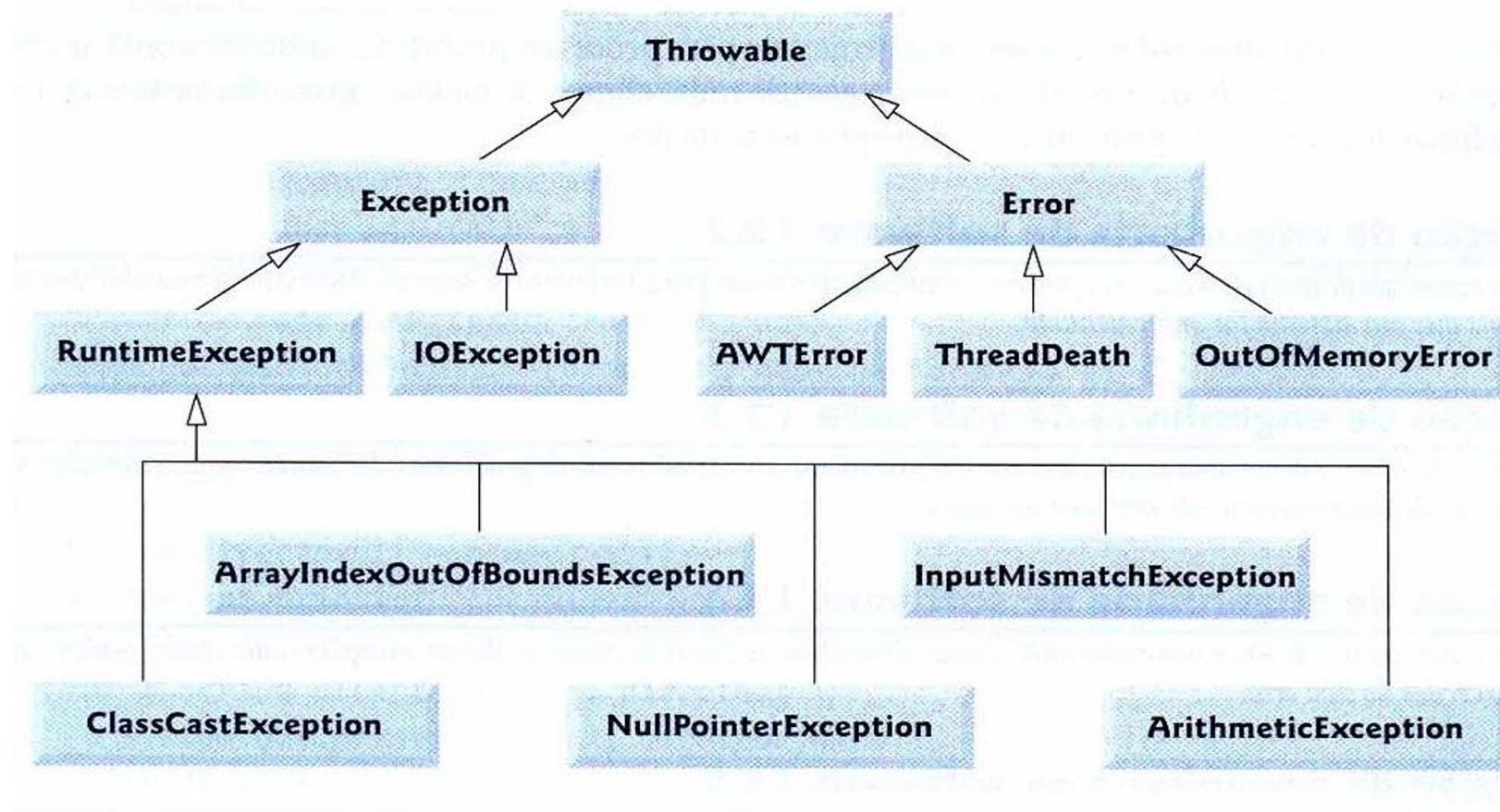


**E uma exceção em Java é um ...?**

**OBJETO**



# Hierarquia de Exceções em Java



# Classes de tratamento de exceções

**Throwable:** Classe base de todas as exceções.

Fornece os métodos:

- `getMessage()` – retorna mensagem do erro.
- `printStackTrace()` – retorna a pilha de métodos chamados.
- `toString` - retorna uma string com a descrição da exception.

**Exception:**

- subclasse de Thworable.
- Por convenção, todas as classes de exceção são subclasses de Exception.



# Bloco try / catch

Como a exceção é lançada por toda a cadeia de classes do sistema, a qualquer momento é possível se “pegar” essa exceção e dar a ela o tratamento adequado.

Para se fazer este tratamento, é necessário pontuar que um determinado trecho de código que será observado e que uma possível exceção será tratada de uma determinada maneira:

O bloco try, é o trecho de código em que uma exceção é esperada e o bloco catch, em correspondência ao bloco try, prepara-se para pegar a exceção ocorrida e dar a ela o tratamento necessário. Uma vez declarado um bloco try, a declaração do bloco catch torna-se obrigatória.

```
public class Main {  
  
    /**  
     * @param args the command line arguments  
     */  
    public static void main(String[] args) {  
  
        try {  
            /* Trecho de código no qual uma exceção pode acontecer.*/  
        } catch (Exception ex) {  
            /* Trecho de código no qual uma exceção do tipo "Exception" será tratada.*/  
        }  
    }  
}
```



# Exemplo em que as linhas abaixo da exception não serão executadas.

```
public class Main {  
  
    /**  
     * @param args the command line arguments  
     */  
    public static void main(String[] args) {  
  
        Scanner s = new Scanner(System.in);  
  
        try {  
            System.out.print("Digite um valor inteiro...:");  
            int numerol = s.nextInt();  
            System.out.print("Digite um valor inteiro...:");  
            int numero2 = s.nextInt();  
  
            System.out.println(numerol + " + " + numero2 + " = " + (numerol + numero2));  
        } catch (Exception ex) {  
            System.out.println("ERRO - Valor digitado nao e um numero inteiro!");  
        }  
  
    }  
}
```



# Palavra-chave throw

Também é possível que você próprio envie uma exceção em alguma situação específica, como em uma situação de login em que o usuário digita incorretamente sua senha. Para realizarmos tal tarefa é necessária a utilização da palavra-chave `throw` da seguinte maneira:

`throw new << Exceção desejada >>();`

```
//  
public class Main {  
  
    public static final String SENHASECRETA = "123456";  
  
    public static void main(String[] args) {  
  
        try {  
            Scanner s = new Scanner(System.in);  
  
            System.out.print("Digite a senha: ");  
  
            String senha = s.nextLine();  
  
            if (!senha.equals(SENHASECRETA)) {  
                throw new Exception("Senha invalida!!!");  
            }  
            System.out.println("Senha correta!!!Bem vindo(a) !!!");  
  
        } catch (Exception ex) {  
            System.out.println(ex.getMessage());  
        }  
    }  
}
```



# Bloco finnaly

A palavra-chave `finally` representa um trecho de código que será sempre executado, independentemente se uma exceção ocorrer.

```
public class Main {  
  
    public static final String SENHASECRETA = "123456";  
  
    public static void main(String[] args) {  
  
        try {  
            Scanner s = new Scanner(System.in);  
  
            System.out.print("Digite a senha: ");  
  
            String senha = s.nextLine();  
  
            if (!senha.equals(SENHASECRETA)) {  
                throw new Exception("Senha invalida!!!!");  
            }  
  
            System.out.println("Senha correta!!!Bem vindo(a) !!!");  
  
        } catch (Exception ex) {  
            System.out.println(ex.getMessage());  
        } finally {  
            System.out.println("Bloco Finally.");  
        }  
    }  
}
```



# Palavra-chave throws

Caso em algum método precise lançar uma exceção, mas você não deseja tratá-la, quer retorna-la para o objeto que fez a chamada ao método que lançou a exceção, basta utilizar a palavra chave throws no final da assinatura do método. Quando utilizamos o throws precisamos também informar qual ou quais exceções podem ser lançadas.

```
public class Main {

    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        try {
            Main et = new Main();

            System.out.print("Digite o valor do dividendo: ");
            double dividendo = s.nextDouble();

            System.out.print("Digite o valor do divisor: ");
            double divisor = s.nextDouble();

            double resultado = et.dividir(dividendo, divisor);

            System.out.println("O resultado da divisão eh: " + resultado);
        } catch (Exception ex) {
            System.out.println(ex.getMessage());
        }
    }

    public double dividir(double dividendo, double divisor) throws Exception {
        if (divisor == 0) {
            throw new Exception("Não é permitido fazer uma divisão por zero!");
        }

        return dividendo / divisor;
    }
}
```



# Tratamento de Exceções

Palavras  
utilizadas  
no  
tratamento  
de  
exceções:

try

catch

finally

throw

throws

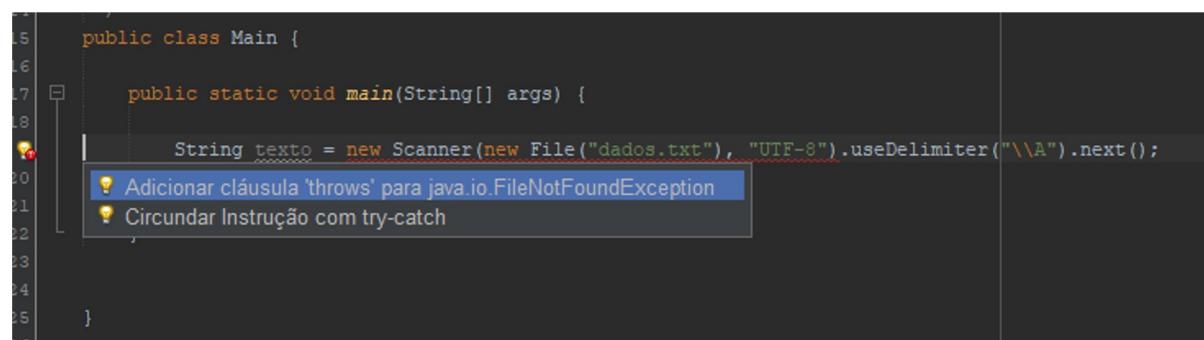


# Checked Exceptions

As checked exceptions, são exceções já previstas pelo compilador. Usualmente são geradas pela palavra chave `throw` (que é discutido mais adiante). Como ela é prevista já em tempo de compilação, se faz necessária a utilização do bloco `try / catch` ou da palavra chave `throws`. A princípio, todas as classes filhas de `Exception` são do tipo checked, exceto pelas subclasses de `java.lang.RuntimeException` (exceção em tempo de execução).



A screenshot of an IDE showing Java code. Line 18 contains the code: `String texto = new Scanner(new File("dados.txt"), "UTF-8").useDelimiter("\n").next();`. A yellow exclamation mark icon is positioned next to the word `Scanner`. A tooltip window is open at the bottom right of the code editor, containing the text: `unreported exception FileNotFoundException; must be caught or declared to be thrown` and `(Alt-Enter mostra dicas)`.



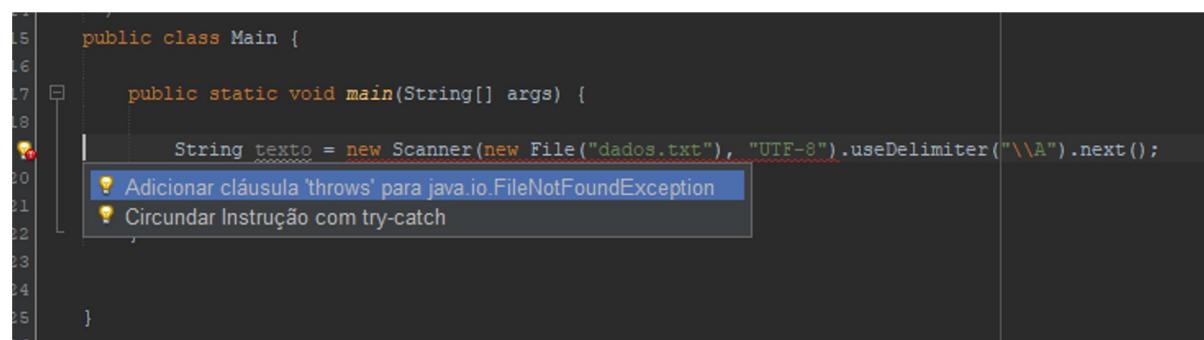
A screenshot of an IDE showing Java code. Line 18 contains the code: `String texto = new Scanner(new File("dados.txt"), "UTF-8").useDelimiter("\n").next();`. A yellow exclamation mark icon is positioned next to the word `Scanner`. A tooltip window is open at the bottom right of the code editor, containing two suggestions: `Adicionar cláusula 'throws' para java.io.FileNotFoundException` and `Circundar Instrução com try-catch`.

# Checked Exceptions

As checked exceptions, são exceções já previstas pelo compilador. Usualmente são geradas pela palavra chave `throw` (que é discutido mais adiante). Como ela é prevista já em tempo de compilação, se faz necessária a utilização do bloco `try / catch` ou da palavra chave `throws`. A princípio, todas as classes filhas de `Exception` são do tipo checked, exceto pelas subclasses de `java.lang.RuntimeException` (exceção em tempo de execução).



A screenshot of an IDE showing Java code. Line 18 contains the code: `String texto = new Scanner(new File("dados.txt"), "UTF-8").useDelimiter("\n").next();`. A yellow exclamation mark icon is positioned next to the word `Scanner`. A tooltip window is open at the bottom right of the code editor, containing the text: "unreported exception FileNotFoundException; must be caught or declared to be thrown" and "(Alt-Enter mostra dicas)".



A screenshot of an IDE showing Java code. Line 18 contains the code: `String texto = new Scanner(new File("dados.txt"), "UTF-8").useDelimiter("\n").next();`. A yellow exclamation mark icon is positioned next to the word `Scanner`. A tooltip window is open at the bottom right of the code editor, containing two suggestions: "Adicionar cláusula 'throws' para java.io.FileNotFoundException" and "Circundar Instrução com try-catch".

# Unchecked Exceptions

Um dos fatores que tornam a `RuntimeException` e suas classes filhas tão específicas em relação as demais subclasses de `Exception` é que elas são exceções não diretamente previstas por acontecer em tempo de execução, ou seja, são unchecked exceptions. Por exemplo:

```
public class Main {  
  
    public static void main(String[] args) {  
        Scanner s = new Scanner(System.in);  
        System.out.print("Digite um numero inteiro...: ");  
        int numero = s.nextInt();  
  
        System.out.println("Numero lido: " + numero);  
    }  
  
}
```

Este tipo de exceção, que acontece somente em tempo de execução, a princípio não era tratado e uma exceção do tipo `java.util.InputMismatchException` será gerada e nosso programa é encerrado. Agora iremos prever este erro, utilizando o bloco `try / catch`:



# Unchecked Exceptions

```
public class Main {  
  
    public static void main(String[] args) {  
        int numero = 0;  
        boolean validado = false;  
        while (!validado) {  
            try {  
                Scanner s = new Scanner(System.in);  
                System.out.print("Digite um numero inteiro..: ");  
                numero = s.nextInt();  
                validado = true;  
            } catch (InputMismatchException ex) {  
                System.out.println("O valor inserido nao e um numero inteiro!");  
            }  
        }  
        System.out.println("O valor lido foi: " + numero);  
    }  
}
```



# Errors

Errors são um tipo especial de Exception que representam erros da JVM, tais como estouro de memória, entre outros. Para este tipo de erro normalmente não é feito tratamento, pois sempre quando um `java.lang.Error` ocorre a execução do programa é interrompida.



# Tratando múltiplas Exceções

É possível se tratar múltiplos tipos de exceção dentro do mesmo bloco try / catch, para tal, basta declará-los um abaixo do outro, assim como segue:

```
public class Main {  
  
    public static void main(String[] args) {  
        try {  
            /* Trecho de código no qual uma exceção pode acontecer. */  
        } catch (InputMismatchException ex) {  
            /* Trecho de código no qual uma exceção  
            do tipo "InputMismatchException" será tratada. */  
        } catch (RuntimeException ex) {  
            /* Trecho de código no qual uma exceção  
            do tipo "RuntimeException" será tratada. */  
        } catch (Exception ex) {  
            /* Trecho de código no qual uma exceção  
            do tipo "Exception" será tratada. */  
        }  
    }  
}
```



# Vamos Praticar?

**Lista de Exercícios no Classroom.**



# Criando sua exceção

Na linguagem Java podemos também criar nossas próprias exceções, normalmente fazemos isso para garantir que nossos métodos funcionem corretamente, dessa forma podemos lançar exceções com mensagens de fácil entendimento pelo usuários, ou que possa facilitar o entendimento do problema para quem estiver tentando chamar seu método possa tratar o erro.

No exemplo abaixo vamos criar uma exceção chamada ErroDivisao que será lançada quando ocorrer uma divisão incorreta, para isso precisamos criar esta classe filha de Exception.

```
/*
 * 
 */
public class ErroDivisao extends Exception {

    public ErroDivisao() {
        super("Divisao invalida!!!!");
    }
}
```



# Criando sua exceção

```
public class Main {  
  
    public static void main(String[] args) {  
        try {  
            Scanner s = new Scanner(System.in);  
            System.out.print("Digite o valor do dividendo: ");  
            int numerol = s.nextInt();  
  
            System.out.print("Digite o valor do divisor: ");  
            int numero2 = s.nextInt();  
  
            Main teste = new Main();  
  
            System.out.println("Resto: " + teste.restoDaDivisao(numerol, numero2));  
        } catch (ErroDivisao ex) {  
            System.out.println(ex.getMessage());  
        }  
    }  
  
    public int restoDaDivisao(int dividendo, int divisor) throws ErroDivisao {  
        if (divisor > dividendo) {  
            throw new ErroDivisao();  
        }  
  
        return dividendo % divisor;  
    }  
}
```



# Biblioteca Lombok



# Sistema de Anúncios de Carros

RQF.001 - MANTER MARCAS

RQF.002 - MANTER MODELOS

RQF.003 - MANTER COMBUSTÍVEL

RQF.004 - MANTER CORES

RQF.005 - MANTER ITENS DO VEÍCULO

RQF.006 - MANTER PROPRIETÁRIO

RQF.007 - MANTER VEÍCULO

RQF.008 - MANTER CLIENTES

RQF.009 - MANTER ANÚNCIO



- **Criando nosso Projeto.**
- **Maven(H2 + Lombok).**
- **Criando Classes Modelo.**
- **Criando classes de negócio.**
- **Simulando operações na classe Main.**
- **Criando classe de conexão com o banco de dados.**
- **Inserindo Registros.**
- **Atualizando Registros.**
- **Deletando Registros.**
- **Consultando Registros.**



# Driver JDBC

Pode-se dizer que é uma API que reúne conjuntos de classes e interfaces escritas na linguagem Java na qual possibilita se conectar através de um driver específico do banco de dados desejado.

Com esse driver pode-se executar instruções SQL de qualquer tipo de banco de dados relacional.

Para fazer a comunicação entre a aplicação e o SGBDs é necessário possuir um driver para a conexão desejada. Geralmente, as empresas de SGBDs oferecem o driver de conexão que seguem a especificação JDBC para caso de algum desenvolvedor querer utilizar.



# Pacote java.sql

**Esse pacote oferece a biblioteca Java o acesso e processamento de dados em uma fonte de dado. As classes e interfaces mais importantes são:**

**DriverManager**

**Connection**

**ResultSet**

**PreparedStatement**



# DriverManager

Cada driver que se deseja a usar precisa ser registrado nessa classe, pois ela oferece métodos para gerenciar um driver JDBC.

Para a aplicação reconhecer a comunicação com o banco de dados escolhido, é preciso obter um arquivo com a extensão .jar que geralmente se consegue através dos sites das empresas que distribuem o SGBD. Esse arquivo tem o objetivo ajudar no carregamento do driver JDBC. Na prática de um desenvolvimento moderno essa biblioteca pode ser buscada e utilizada através do maven.



# Connection

**Representa uma conexão ao banco de dados.**

**Método close -** Geralmente é inserido dentro do bloco finally para realizar sempre a sua execução, pois esse método serve para fechar e liberar imediatamente um objeto Connection.

**Método prepareStatement -** É usado para criar um objeto que representa a instrução SQL que será executada, sendo que é invocado através do objeto Connection.



# PreparedStatement

**Nesta interface que estende a interface base Statement são listados os métodos executeQuery e executeUpdate que são considerados os mais importantes referente a execução de uma query.**

- **executeQuery** - Executa uma instrução SQL que retorna um único objeto ResultSet.
- **executeUpdate** - Executa uma instrução SQL referente a um INSERT, UPDATE e DELETE. Esse método retorna a quantidade de registros que são afetados pela execução do comando SQL.



# ResultSet

Representa o conjunto de resultados de uma tabela no banco de dados. Esse objeto mantém o cursor apontando para a sua linha atual de dados, sendo que seu início fica posicionado na primeira linha.

Além disso, esse objeto fornece métodos getters referentes aos tipos de dados como: `getInt`, `getString`, `getDouble`, `getFloat`, `getLong` entre outros. Com esses métodos são possíveis recuperar valores usando, por exemplo, o nome da coluna ou número do índice.

```
conn = new DatabaseConnection().getConnection();

ps = conn.prepareStatement(FIND_ALL);
rs = ps.executeQuery();

while(rs.next()){

    Cor cor = new Cor();
    cor.setDescricao(rs.getString("descricao"));
    cor.setId(rs.getInt(1));

    listaResultado.add(cor);
}
```



# Classe conexão Base de Dados - H2

```
public class DatabaseConnection {  
  
    public Connection getConnection() throws SQLException {  
        return DriverManager.getConnection("jdbc:h2:~/h2database/database", "", "");  
    }  
  
}
```



# Bibliografia Base

[https://www.java.com/pt-BR/download/help/whatis\\_java.html.](https://www.java.com/pt-BR/download/help/whatis_java.html) Acesso em 19/02/2022.

[https://www.devmedia.com.br/java-historia-e-principais-conceitos/25178.](https://www.devmedia.com.br/java-historia-e-principais-conceitos/25178) Acesso em 19/02/2022.

**DEITEL, Harvey; DEITEL, Paul. Java: Como programar. 10. ed. São Paulo: Pearson Education do Brasil, 2017.**



# Perguntas?



# Conclusão