

# JAVA – BREVE HISTÓRICO



# 1991 – Sun Microsystems

2

- Início do Projeto Green
  - Foco: programas para pequenos dispositivos eletrônicos
    - PDA's
    - Eletrodomésticos em geral
  - Problema: desenvolver programas específicos para cada dispositivo
  - Solução: desenvolvimento de um SO e uma linguagem
    - GreenOS
    - Oak

# 1993 – Conclusão do Star7

3

- Conclusão do PDA
- Perda da concorrência
- Mudança de foco: Internet
  - A linguagem Oak passa a se chamar Java

# 1994 – Apresentação do Java

4

- Criação do WebRunner: primeiro navegador que utilizava Java
- Apresentação do ambiente Java, juntamente com o navegador, rebatizado de HotJava

# 1995

5

- Primeira versão do Netscape com suporte à Java
  - Execução de pequenos programas (Applets)
- Outras empresas lançam versões de seus navegadores com suporte à Java

# 1996 – Distribuição do JDK

6

- Sun inova e distribui gratuitamente o seu kit de desenvolvimento
  - Inicialmente para Solaris, Windows 95 e Windows NT
  - Posteriormente para outros SO's
    - OS/2
    - Linux
    - Macintosh

# 1996 até hoje

7

- A aceitação do Java cresceu rapidamente
  - Porque (dentre outras coisas) Java é:
    - Simples
    - Orientada a objetos
    - Robusta
    - Segura
    - Independente de plataforma
    - Portável

# TECNOLOGIA JAVA





# Tecnologia Java

9

- Composta de duas partes
  - Linguagem de Programação Java
  - Plataforma Java

# Linguagem de Programação Java

10

- Originalmente foi criada como parte de um projeto de criação de um software avançado para diversos dispositivos e plataformas
- No início a linguagem escolhida para o projeto foi o C++
- Devido às dificuldades encontradas com o C++ foi decidido que era necessária a criação de uma linguagem nova
- Assim, a linguagem Java foi feita para suprir os seguintes desafios:
  - Operar em ambientes heterogêneos
  - Operar em ambientes de rede distribuídos
  - Operar com segurança
  - Consumir o mínimo de recursos disponíveis
  - Ser dinamicamente expansível

# Linguagem de Programação Java

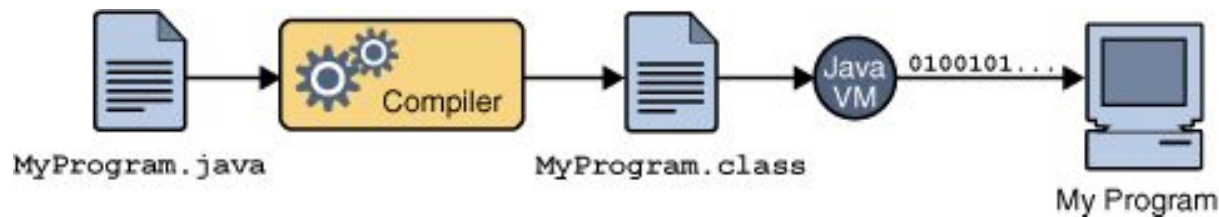
11

- Objetivos principais da linguagem
  - Simples, orientada a objetos e familiar
  - Segura e robusta
  - Arquitetura neutra e portátil
  - Alta performance
  - Interpretada, Multithread e dinâmica

# Linguagem de Programação Java

12

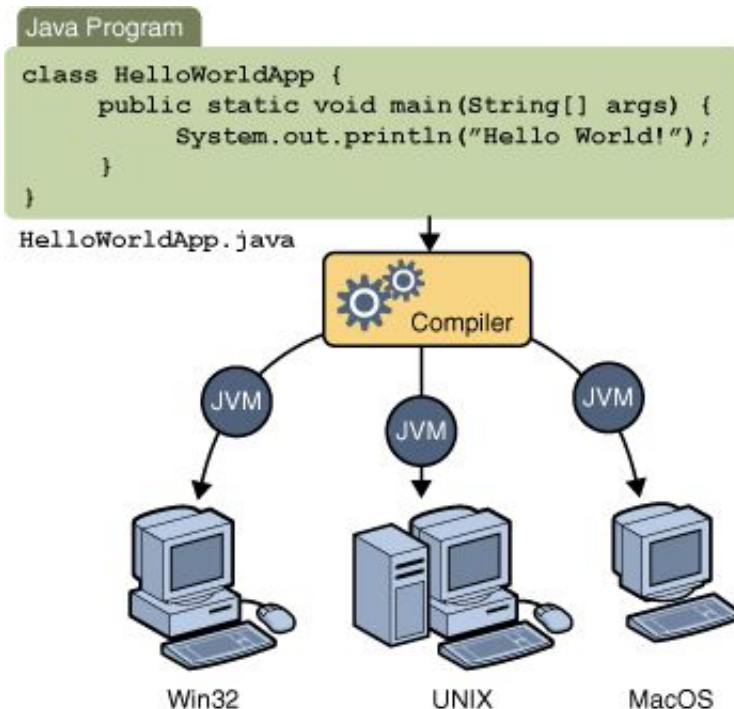
- O processo de desenvolvimento em Java consiste em:
  - Os programas são escritos em arquivos texto com a extensão `.java`
  - Utilizando um programa que vem com o JDK do Java SE chamado `javac` estes arquivos são compilados em arquivos com a extensão `.class`
  - Estes arquivos não contém código nativo compilado para a plataforma. Ele contém os chamados bytecodes que são instruções a serem executadas por uma Virtual Machine
  - Depois o programa `java` launcher executa os bytecodes em uma máquina virtual



# Linguagem de Programação Java

13

- Pelo fato de ser interpretada pela Java Virtual Machine os programas em bytecodes são portáveis em várias arquiteturas



# Plataforma Java

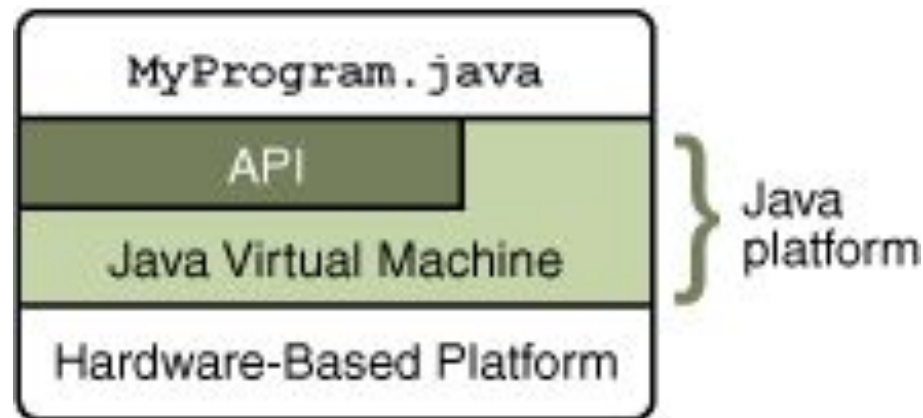
14

- Uma plataforma é um ambiente onde um programa é executado
  - Hardware
  - Software
- A Plataforma Java é composta de
  - Java Virtual Machine
  - Java Application Program Interface – API
- A API Java é um conjunto de componentes prontos para ajudar na programação em Java
- São agrupados em bibliotecas de classes e interfaces que são conhecidos como pacotes

# Plataforma Java

15

- Abaixo é mostrada uma figura com o contexto de um programa e a Plataforma Java
- Apesar de poder ser um pouco mais lento por ser interpretado os avanços na arquitetura da JVM estão trazando a performance perto do código nativo sem sacrifício da portabilidade.



# INSTALAÇÃO





# Ambientes Java

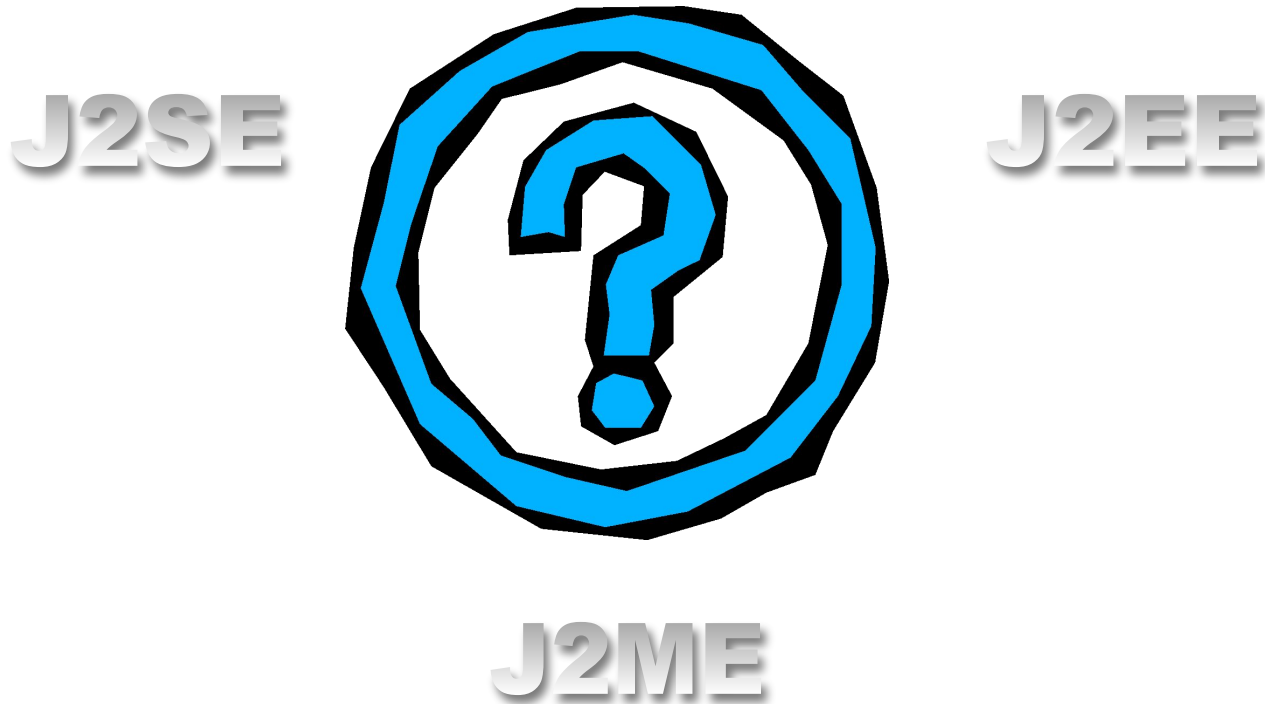
17

- JDK – Java Development Kit
  - Ambiente de Desenvolvimento Java
- JRE – Java Runtime Environment
  - Ambiente de Execução Java

# Toolkits (kits de desenvolvimento)

18

- Primeira dificuldade para novatos em Java:



# J2SE – Java 2 Platform Standard Edition

(Edição Padrão da Plataforma Java 2)

19

- J2SE – Java 2 Platform Standard Edition  
(Edição Padrão da Plataforma Java 2)
  - Ideal para o desenvolvimento da maioria das aplicações Java de pequeno e médio porte.
- J2EE – Java 2 Enterprise Edition  
(Edição Empresarial da Plataforma Java 2)
  - Voltada para aplicações mais robustas, distribuídas e de grande porte.
- J2ME – Java 2 Platform Micro Edition  
(Edição Micro da Plataforma Java 2)
  - Vasto conjunto de soluções para o desenvolvimento de pequenos dispositivos (eletrodomésticos, cartões inteligentes, palms e celulares)

<http://java.sun.com/javase/downloads/index.jsp>

## Java SE Downloads



### Download the complete platform and runtime environment

Download the Java SE-JavaFX bundle, and use your creative talents to design a winning application. » [Get the bundle](#)

Overview Technologies Documentation Community Support **Downloads**

**Latest Release** | Next Release (Early Access) | Embedded Use | Real-Time | Previous Releases



Java Platform (JDK)  
» [JDK](#) » [JRE](#)



JDK + JavaFX Bundle



JDK + NetBeans Bundle



JDK + Java EE Bundle

Here are the Java SE downloads in detail.

#### Java Platform, Standard Edition

##### JDK 6 Update 20 ([JDK](#) or [JRE](#))

This release contains critical security updates to the Java runtime. Please update now to take advantage of these enhancements. » [Learn more](#)

[Download JDK](#)

[Docs](#) ▼

#### Free Java Download

- » Check downloads for **all operating systems**
- » Read more about **Supported System Configurations**

#### Java for Business

- » Access to critical fixes
- » Long-term support
- » Enterprise features
- » JRE or JDK 6, 5.0, or 1.4.2

#### Regional Downloads

[Japanese](#)  
[日本語版](#)



Java EE SDK  
Fuels Efficiency

#### Related Resources

# Ferramentas para o desenvolvimento Java

21

- Editor
- Compilador (javac)
- Interpretador (java)

# AloMundo.java

22

```
/**
 * Classe AloMundo: mostra uma mensagem na tela
 */
class AloMundo {

    public static void main(String[] args) {

        System.out.println("Alo Mundo Java!"); // Imprime
    }

}
```

# Elementos Básicos

23

- Inicialmente os elementos que aparecem na aplicação básica são:
  - Comentários
    - Ignorados pelo compilador
    - Existem três tipos em Java:
      - Comentário simples: `/* texto */`
      - Comentário documentador – utilizado pela ferramenta javadoc: `/** texto */`
      - Comentário de linha – o compilador ignora tudo depois do início do comentário - `// texto`
  - Definição de classe
    - **palavra chave** `class` + `NomeDaClasse` `{ ... }`  
`class NomeDaClasse` `{`  
`}`

# Elementos Básicos

24

- Método `main`
  - Em aplicações console Java é necessário que exista o método `main`.
  - Isto não é requisito para outros tipos de aplicações internet, por exemplo.
  - O formato para o método `main` é:

```
public static void main(String[] args) {  
    // Conteúdo do programa  
}
```
  - O `public` e `static` podem vir em qualquer ordem mas este é o padrão.
  - O argumento pode ter qualquer nome mas normalmente é utilizado `args`.



# Elementos Básicos

25

- A única instrução deste programa é:

```
System.out.println("Alo Mundo Java!"); // Imprime
```

- Esta instrução utiliza uma biblioteca da API do Java para imprimir uma mensagem na saída padrão.
- Se estivermos executando a aplicação a partir da linha de comando, a saída padrão então será o console.
- Utilizando-se a API do Java é possível alterar a saída padrão para a impressora ou arquivo por exemplo.
- Exemplos de utilização da API do Java serão vistos mais adiante.

# ESTRUTURAS BÁSICAS PARA PROGRAMAÇÃO

JAVA

Tipos de dados, variáveis, constantes,  
operadores e controle de fluxo



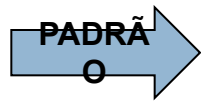
# Tipos de Dados

27

- Inteiros
- Ponto flutuante ou real
- Textuais
- Lógicos

# Tipos de Dados - Inteiros

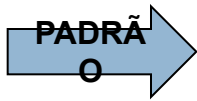
28



Tipo	Tamanho	Mínimo	Máximo
byte	1 byte	-128	127
short	2 bytes	-32.768	32.767
int	4 bytes	-2.147.483.648	2.147.483.647
long	8 bytes	-9.223.372.036. 854.755.808	9.223.372.036. 854.755.808

# Tipo de Dados - Reais

29



Tipo	Tamanho	Precisão
float	4 bytes	6-7 dígitos
double	8 bytes	15 dígitos

# Tipos de Dados - Textuais

30

<b>Tipo</b>	<b>Tamanho</b>	<b>Conteúdo</b>
<b>char</b>	<b>2 bytes</b>	<b>1 caractere</b>
<b>String</b>	<b>Variado</b>	<b>cadeia de caracteres</b>

# Tipos de Dados - Lógicos

31

Tipo	Conteúdo
boolean	true ou false

# Valores Literais

32

Tipo de Dado	Representação	Exemplo
Inteiros base decimal	<b>v</b>	<b>11</b>
Inteiros base hexadecimal	<b>0xv</b>	<b>0xB</b>
Inteiros base octal	<b>0v</b>	<b>013</b>
Inteiros longos	<b>vL</b>	<b>11L</b>
Reais precisão simples	<b>v.vf</b>	<b>24.2f</b>
Reais precisão dupla	<b>v.v</b>	<b>24.2</b>
Lógicos	<b>v</b>	<b>true</b>
Caracteres	<b>'v'</b>	<b>'H'</b>
Texto	<b>"v"</b>	<b>"Ana Carolina"</b>



# Literais.java

33

```
public class Literais {  
    public static void main (String[] args) {  
        System.out.println("Inteiro - decimal: " + 11);  
        System.out.println("Inteiro - hexadecimal: " + 0xB);  
        System.out.println("Inteiro - octal: " + 013);  
        System.out.println("Inteiro - longo: " + 11L);  
        System.out.println("Real - precis o simples: " + 24.2f);  
        System.out.println("Real - precis o dupla: " + 24.2);  
        System.out.println("Tipo L gico:" + true);  
        System.out.println("Caractere : " + 'H');  
        System.out.println("Texto: " + "Ana");  
    }  
}
```

# Variáveis - Declaração

34

```
tipoDaVariavel nomeDaVariavel;
```

- Em Java não existe um bloco específico para declaração de variáveis.

- Exemplo:

```
int idade;  
double valor;  
boolean ligado;
```

# Variáveis - Nomes

35

- Cuidados referentes ao nome das variáveis:
  - Pode começar com letra minúscula ou maiúscula, underscore(\_) ou símbolo dólar (\$)
  - Não pode conter espaços
  - Não pode ser palavras reservadas da linguagem
  - Apesar do tamanho ser ilimitado, evitar nomes muito grandes, por questões de legibilidade do código
- Exemplo de nomes válidos
  - `codigo`, `quantidade`, `indice1`, `numeroTelefone`
- Exemplo de nomes inválidos:
  - `código`, `1teste`, `numero do telefone`

# Variáveis - Atribuição

36

```
nomeDaVariavel = valorLiteral ou outraVariavel;
```

OU

```
tipoDaVariavel nomeDaVariavel = valorLiteral ou outraVariavel;
```

- Quando você atribui uma variável à outra, o Java faz uma cópia do conteúdo da variável origem para a variável destino.
- Exemplo:

```
int j = 10;  
int i = j;  
i=15;
```

- Aqui o Valor final de i é 15 e o valor final de j é 10.

# Constantes

37

## □ Declaração

- Semelhante à declaração de variável acrescentando-se a palavra reservada final no início da instrução.

Exemplo:

```
final double CPMF = 0.38;
```

## □ Atribuição

- Obrigatoriamente feita na declaração

# VariaveisConstantes.java

38

```
public class VariaveisConstantes {  
    public static void main(String[] args) {  
        int qtdeIrmaos;  
        char sexo='F';  
        double minhaAltura=1.65;  
        String meuNome="Rosi Teixeira";  
        final int MES_NASCIMENTO = 12;  
  
        System.out.println("Olá! Meu nome é " + meuNome);  
        System.out.println("Sexo : " + sexo);  
        System.out.println("Altura : " + minhaAltura);  
        System.out.println("Mes de Nascimento : " +  
MES_NASCIMENTO);  
    }  
}
```

# Atividade de Classe

39

- No programa anterior, existe uma variável que não foi inicializada. Qual foi?
- Faça as seguintes alterações no programa anterior, salve, compile e, caso algum problema aconteça, descreva o que aconteceu e qual o seu palpite para a solução do mesmo.
  - Incluir a impressão da variável `qtdelrmaos`.
  - Alterar o tipo da variável `minhaAltura` para `float`.
  - Alterar o tipo da variável `sexo` para `String`

# Operadores

40

- Aritméticos
  - Utilizados para cálculos matemáticos
- Relacionais
  - Utilizados para comparar igualdade e a ordem entre variáveis, constantes, valores literais e expressões,
  - Retornam sempre um valor lógico



# Operadores Aritméticos

41

Operador	Função	Operador	Função
+	Adição	-	Subtração
*	Multiplicação	/	Divisão
%	Resto da Divisão	++	Incremento
--	Decremento	+=	Atribuição Aditiva
-=	Atribuição Subtrativa	*=	Atribuição de Multiplicação
/=	Atribuição de Divisão	%=	Atribuição de Resto

Para compreender o funcionamento destes operadores execute o programa OperadoresAritmeticos.java (exercício de fixação 01)

# Operadores Lógicos

42

## ■ Relacionais

Operador	Função	Operador	Função
==	Igual a	!=	Diferente de
>	Maior que	<	Menor que
>=	Maior ou igual a	<=	Menor ou igual a

## ■ Lógicos

Operador	Função	Operador	Função
	OR lógico		OR dinâmico
&	AND lógico	&&	AND dinâmico
^	XOR lógico	!	NOT unário lógico
=	Atribuição de OR	&=	Atribuição de AND
^=	Atribuição de XOR	?:	If-then-else ternário

Para compreender o funcionamento destes  
operadores execute os programas  
exercício de fixação 02 e 03

# Controle de Fluxo

43

- Estruturas de decisão
  - if - else
  - switch - case
- Estruturas de repetição
  - while
  - for
  - do - while

# if

44

## □ Sintaxe:

```
if (<condição>) {  
    <instruções>  
}
```

<condição>  
• Deve retornar true ou false.

## • Exemplo

```
int idade = 15;  
if (idade < 18) {  
    System.out.println("É menor de idade!");  
}
```

# if - else

45

utilize a clausula else para indicar o que deve ser executado caso a condição do if seja falsa:

```
int idade = 15;
if (idade < 18) {
    System.out.println("É menor de idade!");
}
else {
    System.out.println("É maior de idade!");
}
```

# switch - case

46

## Sintaxe:

<expressão ou variável>  
• Tem que ser int ou char.

```
switch (<expressão ou variável>) {  
    case <valor1>:  
        <instrução1>;  
        break;  
    case <valor2>:  
        <instrução2>;  
        break;  
    case <valor3>:  
        <instrução3>;  
        break;  
    default:  
        <instrução default>;  
}
```

break

• Não é obrigatório mas a ausência dele provoca a execução de todos os cases.

<valorN>

• Tem que ser do mesmo tipo de <expressão ou variável>.  
• Tem que ser valores literais ou constantes.

default

• Não é obrigatório mas é uma boa prática de programação.

# switch - case

47

□ Exemplo:

```
int diaDaSemana=1;
switch (diaDaSemana) {
    case 1:
        System.out.println("Domingo");
        break;
    case 2:
        System.out.println("Segunda-Feira");
        break;
    case 3:
        System.out.println("Terça-Feira");
        break;
    case 4:
        System.out.println("Quarta-Feira");
        break;
    case 5:
        System.out.println("Quinta-Feira");
        break;
    case 6:
        System.out.println("Sexta-Feira");
        break;
    case 7:
        System.out.println("Sábado");
        break;
    default:
        System.out.println("Dia da semana Inválido!");
}
```

# while

48

## Sintaxe:

```
while (<condição>) {  
    <instruções>  
}
```

- **Exemplo:**

```
int idade = 14;  
while (idade < 18) {  
    System.out.println(idade + " anos: ainda é menor!");  
    idade = idade+1;  
}
```



# for

49

## Sintaxe:

```
for
(<inicialização>;<condição>;<incremento>) {
    <instruções>
}
```

- **Exemplo**

```
System.out.println("Contando de zero a
10...");
for (int i=0;i<11;i++) {
    System.out.println(i);
}
```

# Quebras de laço - break

50

## break

- Interrompe a execução do loop
- Exemplo:

```
int x=10;
int y=30;
System.out.println("Intervalo [" + x + ", " + y + "]");
for (int i=x;i<y;i++) {
    System.out.println("Repetição [" + i + "]");
    if (i % 19 == 0) {
        System.out.println("Achei um número divisível por 19 no intervalo.");
        break;
    }
}
```

# Quebras de laço - continue

51

## Continue

- Obriga o loop a executar o próximo laço
- Exemplo:

```
for (int i=0;i<100;i++) {  
    if (i > 50 && i < 60) {  
        continue;  
    }  
    System.out.println("Repetição [" + i + "]");  
}
```

# Operadores

52

<b>Categoria</b>	<b>Operadores</b>
<b>Unários</b>	<code>++ -- + - ! ~ ()</code>
<b>Aritméticos</b>	<code>/ * % + -</code>
<b>Deslocamento</b>	<code>&lt;&lt; &gt;&gt; &gt;&gt;&gt;</code>
<b>Comparação</b>	<code>&lt; &lt;= &gt; &gt;= == != isinstance</code>
<b>Bitwise</b>	<code>&amp; ^  </code>
<b>Curto-circuito</b>	<code>&amp;&amp;   </code>
<b>Condicional</b>	<code>?:</code>
<b>Atribuição</b>	<code>= "op="</code>

# Operadores

53

## □ Ordem de Avaliação

- A ordem de avaliação é sempre da esquerda para a direita no Java.
- A ordem de execução depende da precedência dos operadores e a associatividade.
- Para atribuições a associatividade é da direita para a esquerda.

```
class Associacao {  
    public static void main(String[] args) {  
        int[] a = {1,2,3};  
        int b = 1;  
        a[b] = b = 0;  
        System.out.println("a[0]: " + a[0]);  
        System.out.println("a[1]: " + a[1]);  
        System.out.println("a[2]: " + a[2]);  
    }  
}
```

```
a[0]: 1  
a[1]: 0  
a[2]: 3
```

# Operadores

54

- Operadores unários
  - Só aceitam um argumento.
    - **Incremento e Decremento: ++, --**
    - **Sinal unário: +, -**
    - **Inversão bitwise: ~**
    - **Complemento booleano: !**
    - **Cast: ()**

# Operadores

55

- Incremento e decremento: ++, --
  - Alteram o valor da expressão somando ou subtraindo uma unidade:
    - **int i = 0;**
    - **int a = ++i;**
    - **int b = --i;**
  - Pré-fixado: operação de incremento/decremento é feita e a expressão é avaliada.
  - Pós-fixado: a expressão é avaliada e a operação de incremento/decremento é feita.

# Operadores

56

- Sinal unário: +,-
  - Representam o sinal matemático de uma expressão.
  - O sinal de soma (+) não tem sentido a não ser de enfatizar que o valor é positivo.
  - O sinal de subtração (-) inverte o sinal do operando.



# Operadores

57

- Inversão bitwise:  $\sim$ 
  - Produz uma inversão de bits em um valor inteiro. Por exemplo, se aplicado assim:
    - **`int i = 2; //00000010`**
    - **`int f = ~i; //11111101`**
  - Geralmente é usado em conjunto com os operadores de deslocamento de bits (`<<` `>>` `>>>`).
  - O `>>` é deslocamento logico, preenche sempre com zero a esquerda
  - o `>>>` é deslocamento matematico, preenche sempre o sinal a esquerda

# Operadores

58

- Complemento booleano: !
  - Inverte um valor booleano, true -> false, ou vice-versa.
    - **boolean b = true;**
    - **boolean c = !b; //c=false.**
  - Geralmente é usado no if() para negar uma sentença:
    - **if (!x.equals(b)) ...**

# Operadores

59

- Cast: (tipo)
  - Usado para fazer uma conversão explícita de tipos.
  - Pode ser utilizado para fazer conversão de tipos primitivos ou referências.
    - `int b = 10;`
    - `byte x = (byte)(5 + b);`
    - **Neste exemplo o cast é obrigatório pois a expressão retornará um inteiro.**
- Cast
  - É utilizado para referências principalmente com os objetos do package `java.util`.

# Operadores

60

- Operadores Aritméticos
  - São os próximos em precedência depois dos unários.
    - **Multiplicação e divisão: \*, /**
    - **Módulo: %**
    - **Soma e subtração: +, -**
- Multiplicação e divisão: \*, /
  - Dentre os aritméticos, são os operadores com maior precedência.
  - Operam todos os tipos primitivos numéricos e o tipo char.
  - A divisão inteira pode gerar uma exceção quando o dividendo for zero:
    - **ArithmeticException**

# Operadores

61

- Módulo: %
  - Resto da divisão inteira.
  - Se o operador direito for zero é gerada uma exceção do tipo `ArithmeticException`.
  - Procedimento:
    - **Reduzir o operando esquerdo pelo valor do operando direito**
    - **Repetir a operação até que o valor absoluto do resultado seja menor que o valor absoluto do operando direito.**

# Operadores

62

- Módulo: %
  - $17 \% 5 = 2$
  - $21 \% 7 = 0$
  - $-5 \% 2 = -1$
  - $-5 \% -2 = -1$
  - $7.6 \% 2.9 = 1.8$

# Operadores

63

- Soma e subtração: + , -
  - Reproduzem as operações matemáticas de soma e subtração
  - Se o operador + for aplicado a um objeto do tipo String gera um novo objeto String.
  - Não existe sobrecarga de operadores em Java. O sinal de adição é sobrecarregado pela própria linguagem.
  - Não geram exceções.

# Operadores

64

- Shift Operators: `>>`, `<<`, `>>>`
  - Deslocamento de bits.
  - Só podem ser aplicados a inteiros.
  - Só devem ser aplicados a operandos do tipo `int` e `long`. Para tipos menores ocorre conversão para `int`.
  - Geralmente são usados em programas que utilizam manipulação de portas de entrada e saída.



# Operadores

65

- Operadores de comparação.
  - Retornam um valor booleano;
  - <, <=, >, >=, ==, !=
  - Geralmente usados em testes de condições:
    - **if (boolean)**
    - **for (;boolean;)**

# Operadores

66

- Operadores de comparação: de igualdade
  - `==` , `!=`
  - Testam igualdade e desigualdade respectivamente;
  - Para objetos testa-se se a referência é a mesma;
  - O teste do conteúdo de objetos é feito com o método `equals(Object)`;

# Operadores

67

- Operadores Bitwise: `&`, `^` e `|`
  - Provêm as operações de comparação de bits;
  - `&` - AND, `^` - XOR, `|` - OR;
  - São aplicáveis a tipos inteiros;
- Operadores Lógicos de Curto Circuito: `&&` e `||`
  - Trabalham da mesma forma que os bitwise mas são restritos a operandos booleanos;
  - `&&` - AND, `||` - OR;
  - Não existe equivalente para a operação XOR (`^`);
  - Produzem o efeito de curto circuito quando aplicados a operações aninhadas;

# Operadores

68

- Operadores de comparação.
  - Para valores ordinais - testam o valor relativo de operandos numéricos;
  - Para objetos – testa se o tipo do objeto em tempo de execução é de um tipo ou seu subtipo (classe derivada);
  - Igualdade – testam se dois valores são iguais;

# Operadores

69

- Operadores de comparação: valores ordinais
  - `<`, `<=`, `>`, `>=`
  - Podem ser aplicados a todos os tipos numéricos e produzem um valor booleano;
  - As promoções são realizadas para executar comparações de tipos diferentes;

# Operadores

70

- Operadores de comparação: para objetos
  - Operador isinstance;
  - Testa se o operando da esquerda é uma classe do mesmo tipo, ou uma subclasse, do operando da direita;
    - **if (p isinstance Pessoa) {...}**
  - Pode ser utilizado para arrays. Neste caso primeiro é testado se o operando da esquerda é um array. Segundo se o array é do tipo ou subtipo do operando da direita;

# Exercício - Extra

- Dado uma variável  $x$  com valor 13, faça um programa que transforme  $x$  da seguinte forma:
  - se  $x$  é par,  $y=x/2$
  - se  $x$  é ímpar,  $y=3*x+1$
  - imprime  $y$
  - neste ponto o programa deve então jogar o valor de  $y$  em  $x$  e começar tudo de novo enquanto  $y$  seja diferente de 1.

# ORIENTAÇÃO A OBJETOS BÁSICA

Classes, objetos, métodos e atributos



# P00

73

- Em Programação Orientada a Objetos
  - O foco passa a ser:
    - Os objetos que compõe o sistema
    - As características relevantes destes objetos
    - Identificação das ações executadas e serviços prestados por estes objetos

# Classes e Objetos

74

□ Considere um programa para uma fábrica de carros.

□ Algumas características que todo carro pode ter:

- cor
- modelo
- velocidade atual
- velocidade máxima

□ Algumas ações que podemos solicitar a um carro:

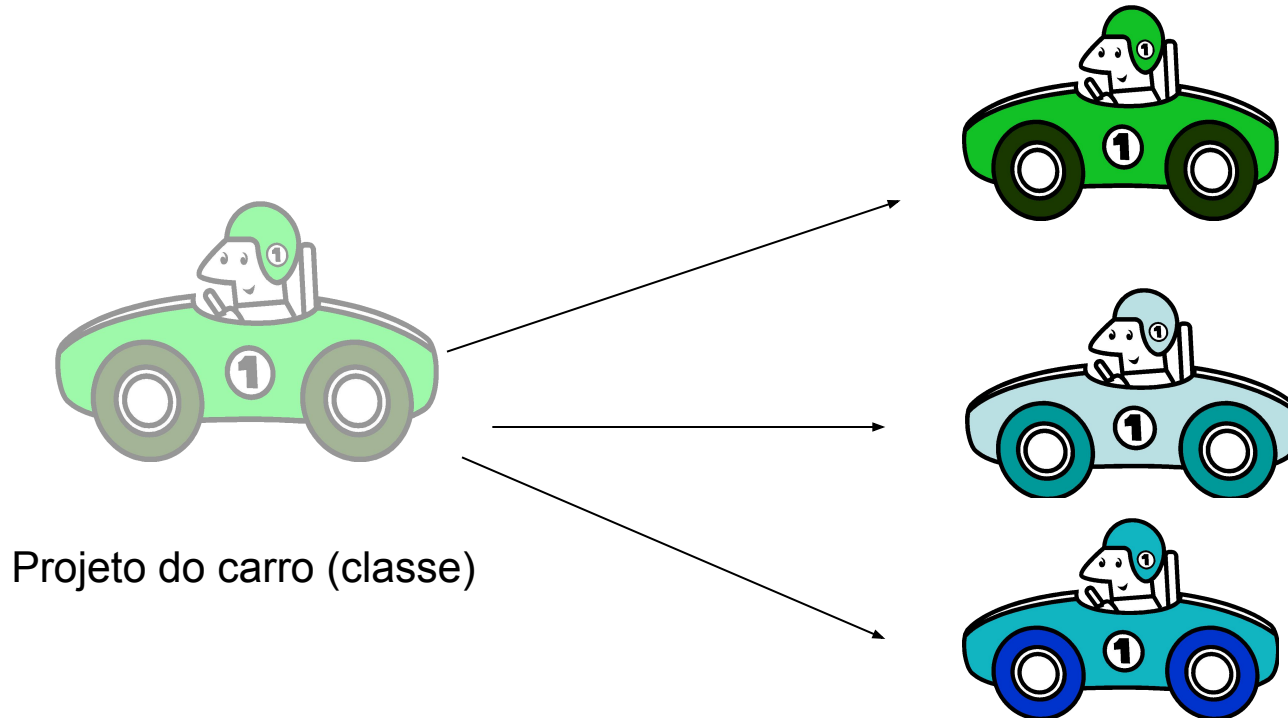
- liga
- desliga
- devolve a marcha atual
- acelera uma quantidade X
- breca uma quantidade X

# Classes e Objetos

75

- Com estas informações temos o projeto de um carro.
- A este projeto damos o nome de classe.

Carros construídos a partir do projeto (objetos)



# Uma classe em Java - atributos

76

Representando o nosso modelo Carro em Java:

```
class Carro {  
    String cor;  
    String modelo;  
    double velocidadeAtual;  
    double velocidadeMáxima;  
}
```

- Até agora declaramos o que todo carro deve ter.
- Observe que estas variáveis foram criados no primeiro escopo da classe. Por isso são chamadas de atributos da classe.

# Uma classe em Java - métodos

77

- Vamos declarar agora o que cada carro faz.

```
class Carro {  
    String cor;  
    String modelo;  
    double velocidadeAtual;  
    double velocidadeMáxima;  
  
    void liga() {  
        System.out.println("Carro ligado!");  
    }  
}
```

- As ações que cada classe pode executar são os métodos
- A palavra **void** indica que este método não retorna nenhuma informação após a execução.

# Uma classe em Java - métodos

78

Outra ação que o carro faz é acelerar

```
class Carro {  
    String cor;  
    String modelo;  
    double velocidadeAtual;  
    double velocidadeMaxima;  
  
    void liga() {  
        System.out.println("Carro ligado!");  
    }  
  
    void acelera(double quantidade) {  
        double velocidadeNova = this.velocidadeAtual + quantidade;  
        this.velocidadeAtual = velocidadeNova;  
    }  
}
```

As variáveis declaradas entre os parêntesis de um método são chamadas de parâmetros ou argumentos do método

A palavra reservada this indica que se trata de um atributo da classe e não uma variável comum.

# Resumindo...

79

□ A sintaxe que vimos até aqui de uma classe em Java é:

```
class <NomeDaClasse> {  
    //Lista de Atributos  
    [<tipoAtributo> <nomeAtributo>];  
  
    // Lista de Métodos  
    [  
        <retorno> <nomeDoMetodo>(<parametros>) {  
            <instruções>  
        }  
    ]  
}
```

# Criando e usando um objeto

80

## Usando a nossa classe Carro

- Para criar, construir ou instanciar um objeto usamos a palavra reservada new.

```
public class Programa {  
    public static void main(String[] args) {  
        Carro meuCarro;  
        meuCarro = new Carro();  
    }  
}
```



# Criando e usando um objeto

81

Através da variável meuCarro podemos acessar o objeto e alterar sua cor, seu modelo, etc.

```
public class Programa {  
    public static void main(String[] args) {  
        Carro meuCarro;  
        meuCarro = new Carro();  
        meuCarro.cor = "Preto";  
        meuCarro.modelo = "Fiesta";  
        meuCarro.velocidadeAtual = 0;  
        meuCarro.velocidadeMaxima = 80;  
    }  
}
```

Observe que utilizamos um ponto para acessar os atributos de meuCarro. O mesmo vale para acessar os métodos.

# Criando e usando um objeto

82

- Agora o objeto meuCarro é um Fiesta Preto com velocidade atual de 0 e velocidade máxima de 80
- Vamos ligar meuCarro, acelerar 20 e imprimir a velocidade atual.

```
public class Programa {  
    public static void main(String[] args) {  
        Carro meuCarro;  
        meuCarro = new Carro();  
        meuCarro.cor = "Preto";  
        meuCarro.modelo = "Fiesta";  
        meuCarro.velocidadeAtual = 0;  
        meuCarro.velocidadeMaxima = 80;  
  
        meuCarro.liga();  
        meuCarro.acelera(20);  
  
        System.out.println(meuCarro.velocidadeAtual);  
    }  
}
```

# Criando e usando um objeto


83

- Vamos incluir ao nosso carro a ação informar a marcha (já que ele é automático).
- A regra é a seguinte:
  - Se a velocidade atual é 0, a marcha atual é -1
  - Se a velocidade atual é maior ou igual a 0 e menor que 40, a marcha atual é 1
  - Se a velocidade atual é maior ou igual a 40 e menor que 80, a marcha atual é 2
  - Se a velocidade atual é maior ou igual a 80, a marcha atual é 3


# Criando e usando um objeto

84

O método `retornaMarchaAtual()` pode ficar assim:



```
int retornaMarchaAtual() {  
    if (this.velocidadeAtual < 0) {  
        return -1;  
    }  
    if (this.velocidadeAtual >= 0 && this.velocidadeAtual < 40) {  
        return 1;  
    }  
    if (this.velocidadeAtual >= 40 && this.velocidadeAtual < 80){  
        return 2;  
    }  
    return 3;  
}
```



- Aqui temos duas novidades: o tipo int no lugar do void e o return
- Esta mudança na assinatura do método indica que o método `pegaMarcha()` retorna um valor, no caso do tipo `int`, e por isso o return é obrigatório.

# Criando e usando um objeto

85

- Métodos que retornam valor poder ser atribuídos à variáveis.
- Você pode acelerar o carro e em seguida pedir a marcha atual assim:

```
meuCarro.liga();  
meuCarro.acelera(40);  
  
System.out.println(meuCarro.velocidadeAtual);  
System.out.println(meuCarro.retornaMarchaAtual());
```

## ■ Ou assim:

```
meuCarro.liga();  
meuCarro.acelera(40);  
  
System.out.println(meuCarro.velocidadeAtual);  
int marchaAtual = meuCarro.retornaMarchaAtual();  
System.out.println(marchaAtual);
```

```

class Carro {
    String cor;
    String modelo;
    double velocidadeAtual;
    double velocidadeMaxima;

    void liga() {
        System.out.println("Carro ligado!");
    }

    void acelera(double quantidade) {
        double velocidadeNova = this.velocidadeAtual + quantidade;
        this.velocidadeAtual = velocidadeNova;
    }

    int retornaMarchaAtual() {
        if (this.velocidadeAtual < 0) {
            return -1;
        }
        if (this.velocidadeAtual >= 0 && this.velocidadeAtual < 40) {
            return 1;
        }
        if (this.velocidadeAtual >= 40 && this.velocidadeAtual < 80) {
            return 2;
        }
        return 3;
    }
}

```

# Criando e usando um objeto

87

□ Nosso programa pode manter em memória vários objetos ao mesmo tempo:

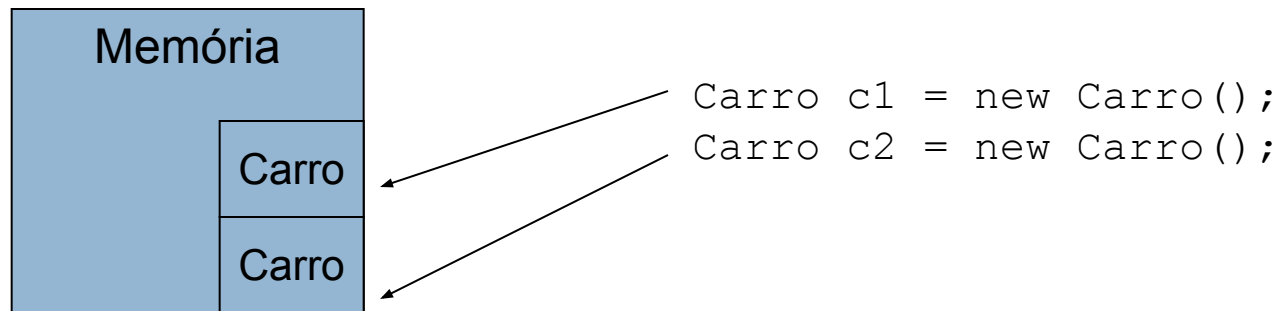
```
Carro meuCarro;  
meuCarro = new Carro();  
meuCarro.cor = "Preto";  
meuCarro.modelo = "Fiesta";  
meuCarro.velocidadeAtual = 0;  
meuCarro.velocidadeMaxima = 80;
```

```
Carro meuSonho = new Carro();  
meuSonho.cor = "Preto";  
meuSonho.modelo = "Pajero";  
meuSonho.velocidadeAtual = 0;  
meuSonho.velocidadeMaxima = 200;
```

# Criando e usando um objeto

88

- Quando declaramos uma variável para associar a um objeto, na verdade esta variável guarda a referência para este objeto



- Experimente agora o seguinte código:

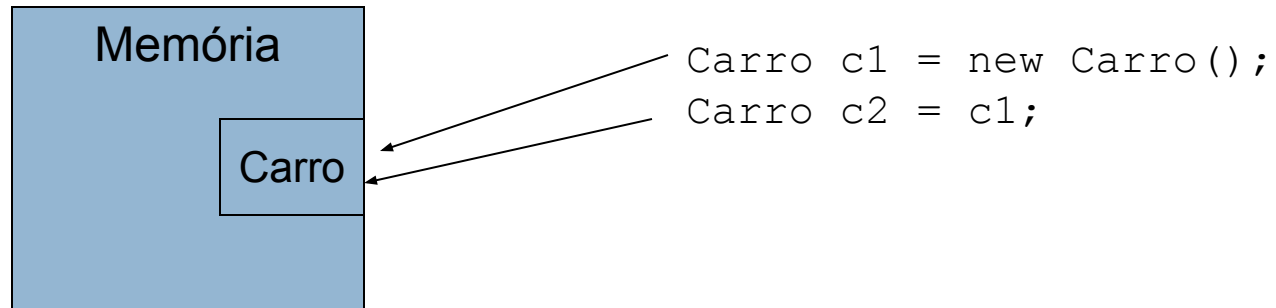
```
Carro c1 = new Carro();  
Carro c2 = c1;  
c1.cor = "Branco";  
System.out.println(c2.cor);
```



# Criando e usando um objeto

89

□ O que aconteceu no exemplo foi:



- c1 e c2 se referenciam ao mesmo objeto.

# Criando e usando um objeto

90

Experimente agora este código:

```
Carro c3 = new Carro();
c3.cor = "Vermelho";
c3.modelo = "Corsa";

Carro c4 = new Carro();
c4.cor = "Vermelho";
c4.modelo = "Corsa";

if (c3 == c4) {
    System.out.println("Os carros são iguais!");
}
else {
    System.out.println("Os carros não são iguais!");
}
```

- == compara o conteúdo de duas variáveis
- Qual é o conteúdo de c3 e c4?

# Criando e usando um objeto

91

Você pode ainda declarar objetos como atributos de outros objetos:

```
class Motor {  
    int potencia;  
    String tipo;  
}
```

```
class Carro {  
    String cor;  
    String modelo;  
    double velocidadeAtual;  
    double velocidadeMaxima;  
    Motor motor;  
    ...  
}
```

## ■ Experimente:

```
Motor meuMotor = new Motor();  
meuMotor.potencia = 10;  
meuMotor.tipo = "Tipo1";
```

```
Carro c5 = new Carro();  
c5.motor = meuMotor;
```

```
System.out.println(c5.motor.potencia);  
System.out.println(c5.motor.tipo);
```

# Exercícios

92

## ContaCorrente

### – Atributos:

- String numero, String tipo, Cliente primeiroTitular, Cliente segundoTitular, double saldo, double limiteTotal

### – Métodos

- credito(double valor) - aumenta o saldo com o valor informado.
- debito(double valor) - diminui o saldo com o valor informado.
- resumo() - Imprime na tela:
  - » Numero da Conta Corrente
  - » Nome do primeiro titular
  - » Nome do segundo titular (se existir)
  - » Saldo Atual
  - » Limite Atual (limite total + saldo atual)
  - » Limite Total
  - » Se estiver devedor mostrar no final do resumo a mensagem: "Procure o seu gerente!"
- estaDevedor() - retorna true se o saldo for menor que zero, retorna false se o saldo for maior ou igual a zero

## • Cliente

### – Atributos:

- String cpf, String nome, String endereço, String anoNascimento

- **Na classe ProgramaBanco.java (com main), crie uma conta corrente, atualize todos os atributos obrigatórios, faça alguns créditos e alguns débitos e depois imprima o resumo da conta corrente.**

# Controlando acesso

93

■ Em nosso sistema de carros o que acontece se fizermos isso:

```
Carro meuCarro= new Carro();  
meuCarro.velocidadeMaxima =  
80;  
meuCarro.acelera(200);
```

- É coerente que o método acelera ultrapasse a velocidade máxima do carro?

# Controlando Acesso

94

## Como resolver isso?

- 1ª solução:
  - No método `acelera()`, antes de alterar a velocidade atual, testar se ela ultrapassa a velocidade máxima.

```
void acelera(double quantidade) {  
    double velocidadeNova = this.velocidadeAtual +  
quantidade;  
    if (velocidadeNova > this.velocidadeMaxima) {  
        System.out.println("Atribuída a velocidade  
máxima.");  
        this.velocidadeAtual = this.velocidadeMaxima;  
    } else {  
        this.velocidadeAtual = velocidadeNova;  
    }  
}
```

# Controlando Acesso

95

Isso melhora muito mas quem garante que ninguém acessará diretamente o atributo?

```
Carro meuCarro= new Carro();  
meuCarro.velocidadeMaxima = 80;  
meuCarro.velocidadeAtual = 200;
```

- Poderíamos testar se não estamos ultrapassando a velocidade máxima antes de alterar diretamente o atributo.

```
Carro meuCarro= new Carro();  
meuCarro.velocidadeMaxima = 80;  
  
if ( (meuCarro.velocidadeAtual + 200) > meuCarro.velocidadeMaxima)  
{  
    meuCarro.velocidadeAtual = meuCarro.velocidadeMaxima;  
} else {  
    meuCarro.velocidadeAtual = meuCarro.velocidadeAtual + 200;  
}
```

# Controlando Acesso

96

- É viável espalhar este teste por todo o código?
- Precisamos na verdade forçar que a alteração da velocidade seja através do método `acelera()`!
- Para isso precisamos limitar o acesso ao atributo `velocidadeAtual`. Para fazer isso em Java utilizamos a palavra-chave `private`.

```
class Carro{
    private String cor;
    private String modelo;
    private double velocidadeAtual;
    private double velocidadeMaxima;
    private Motor motor;
    ...
}
```



# Controlando Acesso

97

- ❑ `private` é um modificador de acesso ou de visibilidade em Java.
- ❑ Os modificadores de acesso servem para restringir o acesso aos atributos, métodos e classes.
- ❑ Os modificadores de acesso disponíveis em Java são:
  - ❑ `private`
  - ❑ `protected`
  - ❑ `public`

# Encapsulamento de Atributos

98

- Em Java é uma prática esconder os atributos das classes
- Assim cada classe é responsável pela atualização de seus atributos.
- Fazemos isto usando o padrão getters e setters.
- O padrão getters e setters consiste em declarar todos os atributos como private e fazer um par de métodos para cada atributo.
  - Um para atualizar o valor do atributo (set...)
  - Um para retornar o valor do atributo (get....)

# Encapsulamento de Atributos

99

## Padrão de nomenclatura:

- set<nome do atributo com a inicial em caixa trocada>
- get<nome do atributo com a inicial em caixa trocada>
- Exemplo
  - Se o nome do atributo é nome, os métodos de get e set dele seriam:
    - setNome
    - getNome
  - Se o nome do atributo é Nome, os métodos de get e set dele seriam:
    - setnome
    - getnome
- Para atributos do tipo boolean ao invés de get, usa-se is no método que retorna o valor do atributo.

# Encapsulamento de Atributos

100

## Exemplo:

```
public class Carro {
    private String cor;
    private String modelo;
    private double velocidadeAtual;
    private double velocidadeMaxima;
    private Motor motor;

    public String getCor()
    {
        return cor;
    }
    public void setCor(String cor)
    {
        this.cor = cor;
    }
    public String getModelo()
    {
        return modelo;
    }
    public void setModelo(String modelo)
    {
        this.modelo = modelo;
    }
    ...
}
```

# Encapsulamento de Atributos

101

- Copie suas classes Carro e Motor para outra pasta e encapsule os seus atributos.

# Construtores

102

- É o bloco que é executado durante a criação do objeto através do `new`.
- A sintaxe dos construtores de uma classe é muito semelhante à sintaxe de um método exceto pelo fato do construtor:
  - Não ter nenhum retorno.
  - Obrigatoriamente ter o mesmo nome da classe

```
public <NomeDaClasse> (<parametros>) {  
    <instruções>  
}
```

# Construtores

103

- Os construtores servem para possibilitar ou obrigar o usuário da classe a passar parâmetros para a classe durante a criação do objeto.
- Quando não definimos um construtor, o Java oferece um construtor padrão sem nenhum parâmetro.

# Construtores

104

Você pode definir vários construtores para suas classes.

- Neste exemplo, estamos dando a opção do carro ser construído com uma cor e um modelo.

```
public class Carro {  
    private String cor;  
    private String modelo;  
    private double velocidadeAtual;  
    private double velocidadeMaxima;  
    private Motor motor;  
  
    public Carro() {  
  
    }  
  
    public Carro(String cor,String modelo) {  
        this.cor = cor;  
        this.modelo = modelo;  
    }  
    ...  
}
```



# Construtores

105

Se quisermos forçar que todos os carros criados tenham uma cor e um modelo, basta não declarar o construtor padrão, ou seja, o construtor sem parâmetros.

- Neste exemplo, estamos forçando que todo carro seja construído com uma cor e um modelo.

```
public class Carro {  
    private String cor;  
    private String modelo;  
    private double velocidadeAtual;  
    private double velocidadeMaxima;  
    private Motor motor;  
  
    public Carro(String cor,String modelo) {  
        this.cor = cor;  
        this.modelo = modelo;  
    }  
    ...  
}
```

# Atividade Prática

106

- Inclua na sua classe Carro o atributo chassi e implemente o construtor para forçar que o chassi seja sempre informado no momento da criação do objeto.
- Implemente construtores também na suas classes do exercício de conta-corrente.

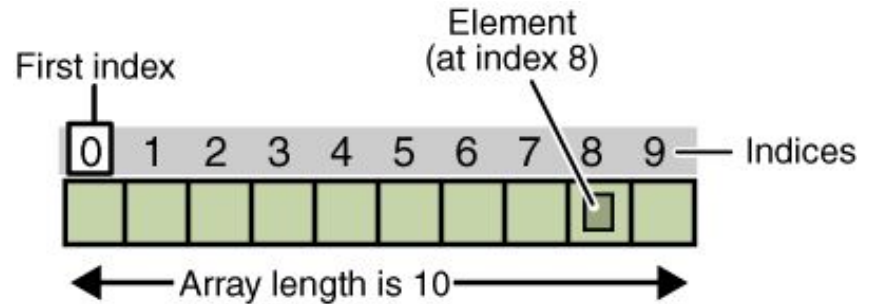
# ARRAYS



# Arrays

108

- Arrays são estruturas de dados que podem armazenar mais que um valor de um mesmo tipo de dado.



- Em Java existem Arrays unidimensionais (Vetores) e multidimensionais (Matrizes).
- Tanto em Vetores, quanto em Matrizes, as operações básicas que você precisa saber fazer são:
  - Declaração
  - Instanciação ou criação
  - Inicialização
  - Consulta

# Vetores - Declaração

109

- A declaração de uma variável tipo Array é semelhante à declaração normal de uma variável.
- Adicionamos apenas os colchetes para indicar que se trata de um Array daquele tipo.
- Sintaxe:
  - `<tipo>[ ] <nome>;` ou `<tipo> <nome>[ ];`
- Exemplo:
  - `int[ ] idades; ou int idades[ ];`

# Vetores – Instanciação ou Criação

110

- Como Arrays são objetos, a criação segue o mesmo padrão da criação de objetos.
- Sintaxe:
  - `<nome> = new <tipo>[<tamanho>];`
    - `<tamanho>` define a quantidade de posições que este vetor vai ter.
- Exemplo:
  - `idades = new int[3];`
- Neste caso criamos um vetor chamado idades que tem 3 posições, ou seja, reservamos 3 espaços na memória para armazenar valores do tipo int.

# Vetores – Inicialização

111

- Assim como declaramos uma variável para armazenar um valor, o sentido de um vetor é receber um conjunto de valores para posterior utilização.
- Inicializar um vetor é na verdade inicializar cada uma de suas posições.
- A sintaxe é:
  - `<nome> [posição] = <valor>;`
- Exemplos:
  - `idades[0] = 8;`
- Neste caso estamos colocando o valor literal 8 na posição 0 do vetor idade. Os índices dos Arrays em Java vão de 0 a n-1, onde n é o tamanho do Array.

# Vetores – Inicialização

112

- Arrays: Inicialização
  - Os elementos de um array são inicializados para seus valores default na construção:
    - **boolean:false;**
    - **byte, short, int, long:0;**
    - **float e double: 0.0f ou 0.0d;**
    - **char:'\u0000';**
    - **referência: null.**



# Vetores – Consulta

113

- A sintaxe para a recuperação de dados em um Vetor é:
  - `<nome>[posição]`
- Exemplo:
  - `System.out.println(idades[0]);`
  - `int idadeNova=idades[0];`

# Vetores - Resumo

114

Resumindo, estas são as operações básicas que envolvem um Vetor:

<b>Declaração</b>	<code>&lt;tipo&gt;[ ] &lt;nome&gt;;</code> ou <code>&lt;tipo&gt; &lt;nome&gt;[ ];</code>	<code>int[ ] numeros;</code> ou <code>int numeros[ ];</code>
<b>Instanciação</b>	<code>&lt;nome&gt; = new &lt;tipo&gt;[tamanho];</code>	<code>numeros = new int[5];</code>
<b>Inicialização</b>	<code>&lt;nome&gt;[posição]=&lt;valor&gt;;</code>	<code>numeros[1]=10;</code>
<b>Consulta</b>	<code>&lt;nome&gt;[posição];</code>	<code>System.out.println(numeros[1]);</code> <code>int umNumero = numeros[1];</code>

# Vetores

115

- Um atributo importante na manipulação de objetos do tipo Array é o length. Ele retorna o tamanho do Array.
- Com isso é possível navegar pelo Array dinamicamente.
- Veja estes dois exemplos:

```
int[] umAteCinco = new int[5];
umAteCinco[0]=1;
umAteCinco[1]=2;
umAteCinco[2]=3;
umAteCinco[3]=4;
umAteCinco[4]=5;
System.out.println(umAteCinco[0]);
System.out.println(umAteCinco[1]);
System.out.println(umAteCinco[2]);
System.out.println(umAteCinco[3]);
System.out.println(umAteCinco[4]);
```

```
int[] umAteCinco = new int[5];
umAteCinco[0]=1;
umAteCinco[1]=2;
umAteCinco[2]=3;
umAteCinco[3]=4;
umAteCinco[4]=5;
for (int i=0;i<umAteCinco.length;i++) {
    System.out.println(umAteCinco[i]);
}
```

- Estes dois códigos são equivalentes!

# Vetores

116

- Teste seus conhecimentos com o exercício de fixação 04.
- Crie uma lista para armazenar o nome de seus familiares. Declare, instancie, atribua e consulte os dados na estrutura array.

# Matrizes

117

- O processo de declaração, instanciação, inicialização e consulta de uma Matriz é muito semelhante aos vetores.
- Temos apenas que acrescentar um par de colchetes para representar a outra dimensão do Array.

<b>Declaração</b>	<code>&lt;tipo&gt;[ ][ ]&lt;nome&gt;;</code> ou <code>&lt;tipo&gt; &lt;nome&gt;[ ][ ];</code>	<code>int[ ][ ] numeros;</code> ou <code>int numeros[ ][ ];</code>
<b>Instanciação</b>	<code>&lt;nome&gt; = new &lt;tipo&gt;[linhas][colunas];</code>	<code>numeros = new int[5][3];</code>
<b>Inicialização</b>	<code>&lt;nome&gt;[linha][coluna]=&lt;valor&gt;;</code>	<code>numeros[1][0]=10;</code>
<b>Consulta</b>	<code>&lt;nome&gt;[linha][coluna];</code>	<code>System.out.println(numeros[1][0]);</code> <code>int umNumero = numeros[1][0];</code>

# Matrices

118

```
class ArrayOfArrayDemo {  
    public static void main(String[] args) {  
        int[][] aMatrix = new int[4][]; //four lines  
        // populate the matrix  
        for (int i = 0; i < aMatrix.length; i++) {  
            aMatrix[i] = new int[5]; // create sub-array  
            for (int j = 0; j < aMatrix[i].length; j++) {  
                aMatrix[i][j] = i + j;  
            }  
        }  
        // print matrix  
        for (int i = 0; i < aMatrix.length; i++) {  
            for (int j = 0; j < aMatrix[i].length; j++) {  
                System.out.print(aMatrix[i][j] + " ");  
            }  
            System.out.println();  
        }  
    }  
}
```

0	1	2	3	4
1	2	3	4	5
2	3	4	5	6
3	4	5	6	7

# Matrizes

119

- Teste seus conhecimentos
  - Uma rainha requisitou os serviços de um monge e disse-lhe que pagaria qualquer preço. O monge, necessitando de alimentos, indagou à rainha sobre o pagamento, se poderia ser feito com grãos de trigo dispostos em um tabuleiro de xadrez, de tal forma que o primeiro quadro conter um grão e os quadros subsequentes o dobro do quadro anterior. A rainha achou o trabalho barato e pediu que o serviço fosse executado, sem se dar conta de que seria impossível efetuar o pagamento. Faça um programa em Java para calcular o número de grãos que o monge esperava receber.
  - Em qual casa você terá overflow usando respectivamente int e long?

# PROGRAMAÇÃO ORIENTADA A OBJETOS

Atributos e métodos estáticos, herança,  
polimorfismo, abstração e interfaces





# Atributos e métodos estáticos

121

- Considere que no nosso sistema de carros será necessário contar quantos carros foram criados até agora.
- A primeira coisa que podemos fazer é criar um contador para ir incrementando ao longo do programa.
- Veja o código disso:

# Atributos e métodos estáticos

122

```
public class ProgramaCarro {  
    public static void main(String[] args) {  
        int qtdeCarros = 0;  
  
        Carro carro1 = new Carro();  
        qtdeCarros++;  
        Carro carro2 = new Carro();  
        qtdeCarros++;  
        Carro carro3 = new Carro();  
        qtdeCarros++;  
        Carro carro4 = new Carro();  
        qtdeCarros++;  
  
        System.out.println("Quantidade de carros criados:  
"+qtdeCarros);  
    }  
}
```

- Este programa informa corretamente a quantidade de carros criada, mas teríamos que espalhar isso por todo código!
- O ideal é que este controle ficasse no construtor do carro.

# Atributos e métodos estáticos

123

```
public class Carro {  
    //demais atributos da classe  
    private int qtde;  
  
    public Carro() {  
        this.qtde++;  
    }  
  
    public int getQtde() {  
        return qtde;  
    }  
    //demais métodos da classe  
}
```

- Vamos criar então um atributo na classe Carro chamado qtde que será incremento cada vez que um novo carro é construído.
- Como precisaremos recuperar esta quantidade, vale a pena também criar o método getQtde()

# Atributos e métodos estáticos

124

□ Agora podemos refazer o código da classe ProgramaCarro utilizando o método getQtde() para retornar o número de carros criados.

- A dúvida é: vamos substituir a variável qtdeCarros pelo método getQtde() de qual carro?
- Se imprimirmos o getQtde() de cada instância do carro criado, qual será o resultado?

```
public class ProgramaCarro {  
    public static void main(String[] args) {  
        Carro carro1 = new Carro();  
        Carro carro2 = new Carro();  
        Carro carro3 = new Carro();  
        Carro carro4 = new Carro();  
  
        System.out.println("Quantidade de carros criados: "+carro1.getQtde());  
        System.out.println("Quantidade de carros criados: "+carro2.getQtde());  
        System.out.println("Quantidade de carros criados: "+carro3.getQtde());  
        System.out.println("Quantidade de carros criados: "+carro4.getQtde());  
    }  
}
```

# Atributos e métodos estáticos

125

- O exemplo anterior nos mostrou que a quantidade de carros será sempre igual a 1.
- Isso acontece porque getQtde() é um método de cada instância da classe Carro.
- Precisamos que este contador seja um atributo da classe Carro e não de suas instâncias.
- Atributos e/ou métodos que pertencem à classe são chamados de estáticos.
- Para marcar um atributo e/ou métodos como estáticos utilizamos a diretiva static logo após a visibilidade do mesmo.

# Atributos e métodos estáticos

126

- Atributos e métodos estáticos não precisam ser acessados a partir de um objeto criado.
- Eles podem ser chamados diretamente na classe
- Veja como ficou nossa classe Carro e ProgramaCarro.

```
public class Carro {  
    //demais atributos da classe  
    private static int qtde;  
  
    public Carro() {  
        Carro.qtde++;  
    }  
  
    public static int getQtde() {  
        return qtde;  
    }  
    //demais métodos da classe  
}
```

Chamada estática do  
atributo qtde

```
public class ProgramaCarro {  
    public static void main(String[] args) {  
        Carro carro1 = new Carro();  
        Carro carro2 = new Carro();  
        Carro carro3 = new Carro();  
        Carro carro4 = new Carro();  
  
        System.out.println("Quantidade de  
        carros criados: "+Carro.getQtde());  
    }  
}
```

Chamada estática do  
método getQtde()

# Atributos e métodos estáticos

127

- Para o nosso exemplo, era suficiente declarar o atributo qtde como estático. Poderíamos acessar getQtde() por um dos carros e o valor retornado estaria correto.
- Mas declarando o método getQtde() como estático criamos a possibilidade dele ser chamado diretamente na classe, o que é mais coerente com o significado de qtde no nosso exemplo.
- Quando o método é estático todas os atributos e métodos acessados dentro dele tem que ser estáticos também.

## Incrementando nosso sistema de Carros

128

- Vamos incluir na nossa classe Carro os atributos tanque e precoCusto (incluindo os getters e setters).
- Vamos incluir agora uma nova classe chamada CarroGNV para representar os carros movidos a gás natural. A classe CarroGNV tem os mesmos atributos de Carro e mais estes:
  - int qtdeCilindros
  - String autorizacaoINMETRO



# Incrementando nosso sistema de Carros

129

- Temos agora duas classes muito parecidas: Carro e CarroGNV.
- O que vai acontecer se for necessário incluir mais um atributo comum a todos os carros?
- E se precisarmos criar um outro tipo de carro?
- Estamos repetindo código!
- O ideal seria aproveitar os atributos de Carro em CarroGNV!
  - Em outras palavras, CarroGNV é uma **extensão** de Carro.
  - Chamamos esta relação entre as classes de herança.

# Herança

130

- Para estabelecer uma herança em Java, utilizamos a palavra extends.

- A sintaxe é:

```
public class <nome da subclasse> extends <superclasse> {  
    <código da subclasse>  
}
```

## ■ Exemplo:

```
public class CarroGNV extends Carro {  
    ...  
}
```

# Herança

131

- Na herança, as subclasses herdam os métodos e atributos da superclasse, exceto os atributos privados.
- Qual deve ser a visibilidade dos atributos?
  - Se for public todos as outras classes podem ver os atributos.
  - Se for private nem mesmo as subclasses podem ver seus atributos.
- Existem um tipo de visibilidade que é intermediária: protected
  - Os atributos e métodos marcados como protected (protegido) são visíveis apenas pela própria classe, suas subclasses e classes do mesmo pacote.

# Herança

132

- Vamos implementar agora o método `getPrecoVenda()` na classe `Carro`
  - O preço de venda de um carro é 30% mais alto que o preço de custo, ou seja,  $\text{precoCusto} * 1.3$
- Crie alguns objetos do tipo `Carro` e `CarroGNV` para experimentar o método `getPrecoVenda()`.

# Herança

133

- Considere agora que o preço de venda de um CarroGNV seja 40% a mais que o preço de custo do carro, ou seja,  $\text{precoCusto} * 1.4$ .
- O método `getPrecoVenda` continua atendendo?
- Não!
- Precisamos modificar este método para a classe CarroGNV!

# Polimorfismo

134

- Se modificarmos o método `getPrecoVenda()` na classe `CarroGNV` estamos aplicando o conceito de polimorfismo.
- Polimorfismo é a capacidade que uma classe ou um método tem de mudar de comportamento em função da maneira como é invocado.
- Estas mudanças no comportamento de métodos ou classes são possíveis através da sobrecarga e sobreposição de métodos.

# Polimorfismo

135

- Fazemos uma sobrecarga quando modificamos os atributos de um método ou construtor para criar versões diferentes dele.
  - Fizemos várias sobrecargas quando estávamos aprendendo a usar construtores.
- Experimente a sobrecarga com o exercício de fixação 06.

# Polimorfismo

136

- A sobreposição é a implementação de métodos em subclasses de forma que:
  - Anule o comportamento que ele se apresentava em sua superclasse
  - Ou apenas acrescente novas instruções.
- É justamente o que precisamos fazer em nosso exemplo das classes Carro e CarroGNV. O método getPrecoVenda() precisa ter outra implementação em CarroGNV. Aproveite e faça isso agora.
- Experimente mais um pouco a sobreposição com os exercícios de fixação 07 e 08 do livro.



# Polimorfismo

137

Um exemplo de como herança e polimorfismo pode flexibilizar nossos programas:

```
public class ProgramaCarro {
    public static void main(String[] args) {
        Carro[] carros = new Carro[3];
        carros[0] = new Carro();
        carros[0].setPrecoCusto(100);
        carros[1] = new CarroGNV();
        carros[1].setPrecoCusto(100);
        carros[2] = new Carro();
        carros[2].setPrecoCusto(100);

        for (int i=0; i<carros.length; i++) {
            System.out.println("Valor de venda do carro " + i
+ " -> "+ carros[i].getPrecoVenda());
        }
    }
}
```

# Polimorfismo

138

- Parece estranho colocar um objeto CarroGNV em uma posição de um vetor de Carro.
- Até agora aprendemos a colocar em vetores e variáveis apenas objetos do tipo declarado!
- O que acontece aqui é que, em função da herança, a superclasse pode ser considerada uma generalização da subclasse. Ou seja:
  - Carro tem os atributos e métodos comuns a todos os carros (é uma forma genérica de carro)
  - CarroGNV tem os atributos e métodos específicos dos carros movidos a gás natural (é uma especialização de carro)
- Apesar de ter atributos e métodos próprios, todo CarroGNV é também um Carro.
- A chamada do métodos é feita sempre em tempo de execução, por isso, no nosso exemplo apesar se tratar de um vetor de Carro, a posição 1 foi inicializada com um objeto CarroGNV, ou seja o objeto que está na memória é um CarroGNV. Como em CarroGNV existe uma sobreposição do método getValorVenda(), é esta implementação que será executada.

# Herança e Construtores

139

- Nas regras de herança entre as classes, os Construtores da superclasse não são herdados pelas subclasses.
- Na criação de um objeto da subclasse, primeiro é feita uma chamada do construtor padrão da superclasse e em seguida o construtor da subclasse que foi invocado.
- Como sabemos, o construtor padrão para qualquer classe é o construtor sem parâmetros. Este construtor é associado à classe explicitamente pelo programador ou implicitamente pela linguagem (quando a classe não possui construtores).
- Quando a superclasse possui construtores declarados e não possui o construtor padrão, é preciso fazer uma referência ao construtor da superclasse na primeira linha de cada construtor da subclasse.
- Para fazer referência à superclasse a partir da subclasse, utilizamos a diretiva super.

# Herança e Construtores

140

- Usando o nosso exemplo da lista 03, a classe Trabalhador só tem 1 construtor que recebe nome e sobrenome.
  - Quando fazemos:
- ```
Chefe chefe = new Chefe("João", "Silva", 5000);
```
- O Java tenta executar o construtor padrão de Trabalhador e não consegue porque ele não existe.
  - Por isso é preciso fazer explicitamente uma chamada ao construtor de Trabalhador no construtor de Chefe:

```
public Chefe(String nome, String sobrenome, double salarioBase) {  
    super(nome, sobrenome);  
    this.salarioBase = salarioBase;  
}
```

# Herança e Construtores

141

- Porque não precisamos fazer o `setNome()` e o `setSobrenome()` na classe chefe?
  - Porque o construtor de `Trabalhador` já faz isso!
- O `super` serve apenas pra chamar o construtor da superclasse?
  - Não! O `super` é uma referência para a superclasse.
  - Podemos utilizar o `super` para invocar explicitamente os métodos da mãe, por exemplo:

```
super.gerarSalario();
```

# Classes e métodos abstratos

142

- Durante a confecção do exercício da folha de pagamento, vocês perceberam que não fazia muito sentido fazer uma implementação padrão do método gerarSalario() em Trabalhador, uma vez que este método sempre será reimplementado nas filhas.
- Olhando bem para o nosso exemplo, faz algum sentido criar uma instância de Trabalhador?

```
Trabalhador trab = new Trabalhador();
```

- Todos os trabalhadores possíveis de nosso sistema estão representados nas subclasses de Trabalhador.
- Trabalhador só nos parece útil para utilizarmos o polimorfismo no método gerarSalario() e herdar os atributos e métodos.

# Classes e métodos abstratos

143

- Quando utilizamos uma superclasse apenas para este fim e não existe necessidade dentro do sistema de criação de novas instâncias diretas desta classe, podemos chamar estas classes de abstratas.
- Classes abstratas não podem ser instanciadas diretamente, ou seja, não podemos fazer um new delas.
- Em classes abstratas podemos ainda definir alguns ou todos os métodos como abstratos.
- Os métodos abstratos não podem ser implementados na superclasse.
- Eles são obrigatoriamente implementados nas subclasses.
- Veja como ficaria a nossa classe Trabalhador se ela fosse abstrata.

# Classes e métodos abstratos

144

```
public abstract class Trabalhador {  
    private String nome;  
    private String sobrenome;  
  
    public Trabalhador(String nome, String sobrenome)  
{  
        this.nome = nome;  
        this.sobrenome = sobrenome;  
    }  
  
    public abstract double gerarSalario();  
    //Demais métodos da classe Trabalhador  
}
```

- Utilizamos a palavra abstract para indicar que a classe e/ou método é abstrato.
- Observe também que o bloco do método gerarSalario() foi excluído.
- Faça os seguintes testes e experimente este conceito de abstração:
  - Inclua o abstract na sua classe Trabalhador e no método gerarSalario() mantendo o bloco de implementação do método;
  - Exclua o método gerarSalario() da classe chefe;
  - Tente criar uma instância de Trabalhador em sua classe de teste (new Trabalhador())



# Interfaces

145

- Continuando com o exemplo da folha de pagamento...
- Por uma determinação da empresa, os trabalhadores que tem salário base poderão agora ter plano de saúde.
- Basicamente toda pessoa que tem plano de saúde precisa de tempos em tempos informar a matrícula do plano, a quantidade de dependentes e o nome completo do titular.
- Falando em termos de programação, as classes `TrabalhadorPorComissao` e `Chefe` terão que implementar os métodos `getMatriculaPlano()`, `getQtdeDependentes()` e `getNomeCompletoTitular()`.
- Como fazer isso?

# Interfaces

146

- Faz sentido incluir estes métodos na superclasse?
- Se precisarmos criar uma lista com todos os trabalhadores que tem plano de saúde para posterior impressão, como faremos isso? Teremos que:
  - Implementar estes métodos em Chefe e em TrabalhadorPorComissao;
  - Criar dois arrays para guardar separadamente os trabalhadores.
- Estamos novamente repetindo código!
- E quando outros tipos de trabalhadores forem contemplados com plano de saúde, repetiremos mais código ainda!
- O ideal seria que existisse outra superclasse com estes métodos para fazermos uso da herança!

# Interfaces

147

- Mesmo que exista outra superclasse com estes métodos, as classes Chefe e TrabalhadorPorComissao não poderiam ser subclasses desta nova classe e de Trabalhador. Java não suporta herança múltipla!
- Uma maneira de resolver este problema é utilizando Interfaces.
- Interfaces são uma espécie de contrato que pode ser estabelecido com as classes para garantir que estas classes implementem os métodos necessários para ser acessadas por outras.
- O funcionamento de interfaces é muito parecido com o de classes abstratas.

# Interfaces

148

Declaração de uma Interface é seguinte:

```
public interface <nomeDaInterface> {  
    <métodos da interface>  
}
```

- Considerando que:
  - A visibilidade de uma Interface não pode ser protected ou private.
  - Uma Interface funciona como uma classe abstrata implícita e, por isso, não pode ser instanciada diretamente.
  - Uma Interface pode ser subclasse de outra Interface

# Interface

149

- Como acontece com os métodos abstratos, os métodos definidos nas Interfaces não podem ser implementados nelas.
- Em Interfaces definimos apenas a assinatura do método e este tem que ser obrigatoriamente implementado nas classes que implementam (ou assinam) este contrato (ou Interface)
- A sintaxe da declaração de métodos em Interfaces é:

```
public interface <nomeDaInterface> {  
    public <retorno> <nomeDoMetodo>;  
}
```

- Note que o bloco de implementação do método não existe.

# Interfaces

150

Uma Interface também pode ter atributos. A sintaxe de declaração de um atributo é:

```
public interface <nomeDaInterface> {  
    public <tipo> <nomeDoAtributo> = <inicialização>;  
}
```

- Considerando que:
  - A visibilidade do atributo não pode ser protected ou private.
  - O atributo de uma Interface funciona como uma constante estática implícita, por isso, tem que ser inicializado no momento da declaração e podem ser acessados diretamente da Interface.

# Interfaces

151

□ Para implementar uma interface utilizamos a diretiva implements.

□ A sintaxe é:

```
<encapsulamento> class <nomeDaClasse> [<extends> <nomeDaSuperclasse>]
    implements <nomeDaInterface> {
        <implementaçãoDaClasse>
    }
```

■ Considerando que:

- A parte entre colchetes, que se refere à declaração de herança, não é obrigatória.
- Uma classe pode implementar várias Interfaces separando-as por virgula.

# Interfaces

152

A nossa Interface ficaria assim:

```
public interface SeguroSaude {  
    public String LEGISLACAO = "Lei nº 1111.2 de 1810";  
  
    public String getMatriculaPlano();  
  
    public int getQtdeDependentes();  
  
    public String getNomeCompletoTitular();  
  
}
```



# Interfaces

153

A implementação da nossa Interface SeguroSaude em Chefe, ficaria assim:

```
public class Chefe extends Trabalhador implements
SeguroSaude {
    private double salarioBase;
    private String matriculaPlano;
    private int qtdeDependentes;

    // Construtor de Chefe

    public String getMatriculaPlano() {
        return this.matriculaPlano;
    }

    public int getQtdeDependentes() {
        return this.qtdeDependentes;
    }

    public String getNomeCompletoTitular() {
        return getNome() + " " + getSobrenome();
    }

    // Demais métodos da classe.
}
```

- Para o nosso exemplo, é interessante que os atributos `matriculaPlano` e `qtdeDependentes` sejam criados, bem como os setters.
- Já a implementação do método `getNomeCompletoTitular()` não sugere a criação de atributos pois apenas retorna a concatenação de dois atributos já existentes na classe.
- A criação ou não de atributos para implementação de métodos definidos em Interfaces não é uma obrigatoriedade. Depende da implementação do método.

# Exercitando

154

- Faça a classe `TrabalhadorPorComissao` implementar a interface `SeguroSaude`.
- Teste seus conhecimentos com Interfaces criando um vetor para receber todos os trabalhadores (2 chefes e 2 por comissão) com seguro de saúde e depois imprima:
  - A legislação do seguro
  - E para cada trabalhador:
    - A matrícula do plano, o nome completo do titular e a quantidade de dependentes.
- Recebemos uma nova determinação do setor de RH: todos os funcionários das áreas operacionais (os que não são chefes) poderão se associar um determinado clube.
  - Quais as informações que o sistema do clube pode precisar que nossos funcionários informem?
  - Crie uma solução pra isso utilizando Interfaces.

# Programação Orientada a Objetos

Pacotes



# O problema

156

- Como agrupar todas as classes e interfaces já feitas por vocês neste curso numa mesma pasta?
- Como disponibilizar pra outras pessoas o .class de algum programa feito por vocês?
- Para resolver estes problemas de organização de classes e interfaces em Java utilizamos pacotes.

# Pacotes

157

- Pacotes podem ser entendidos como recipientes utilizados para agrupar, organizar e dividir as classes e interfaces em espaços menores.
- Na prática a estrutura de pacotes é correspondente à estrutura de pastas que precedem a classe ou interface.
- Por exemplo:
  - A classe `br.edu.rosi.trabalhador.Trabalhador` está na pasta `br\edu\rosi\trabalhador`.

# Pacotes

158

- O padrão de nomenclatura de pacotes sugerido pela Sun é:  
    <dominio>.<pacote1>.<pacote2>.<pacoteN>
  - Tudo em letra minúscula!
- Exemplo:
  - br.edu.rosi.cursojava.alunos
- A idéia de utilizar o domínio no começo do pacote é evitar ao máximo o conflito de pacotes de empresas e/ou programadores diferentes.

# Pacotes

159

Para indicar em qual pacote está uma classe ou interface utilizamos a diretiva package.

```
package br.edu.rm.cursojava;

public class AloMundo {
    public static void main(String[] args) {
        System.out.println( "Alo Mundo
Java" );
    }
}
```

- Estamos indicando que a classe AloMundo.java está na pasta br\edu\rm\cursojava

# Pacotes

160

Para utilizar classes ou interfaces que estejam em outro pacote utilizamos a diretiva `import`.

```
package br.edu.rm.cursojava;

import br.edu.rm.cursojava.trabalhador.Chefe;
import br.edu.rm.cursojava.trabalhador.TrabalhadorPorComissao;

public class AloMundo {
    public static void main(String[] args) {
        System.out.println("Alo Mundo Java");

        Chefe chefinho;
        TrabalhadorPorComissao comissionario;
    }
}
```

- Neste exemplo temos dois imports para classes do mesmo pacote,
- Poderíamos importar todas as classes e interfaces do pacote com a seguinte instrução:  
`br.edu.rm.cursojava.trabalhador.*` Mas não é uma boa prática, pois dificulta a leitura de outros programadores.



# package, import, class ou interface

161

- É muito importante manter a ordem!
- As declarações package e import não são obrigatórias numa classe ou interface,
- Mas se existirem, devem ser colocadas na seguinte ordem no arquivo fonte:
  - package
  - import
  - class ou interface

# Distribuindo seus pacotes

162

- Mesmo compreendendo e organizando bem classes e interfaces em pacotes, distribuir para cada cliente uma série de arquivos .class é complicado.
- A maneira mais simples de fazer esta distribuição é compactá-los em um arquivo só. Em Java fazemos isso utilizando arquivos .jar
- O arquivo .jar é uma arquivo compactado no formato zip que pode ser criado por qualquer compactador zip, inclusive o jar que vem junto com o SDK.

# API PADRÃO JAVA

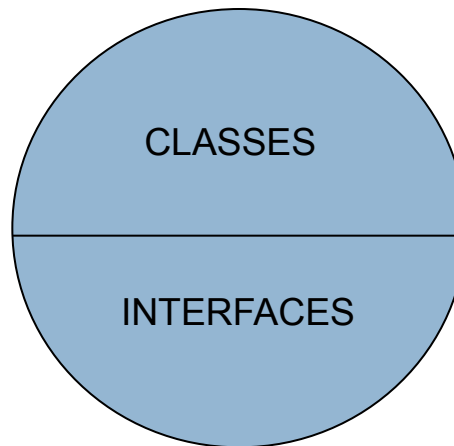
Classes da API Padrão



# Applications Programming Interface

164

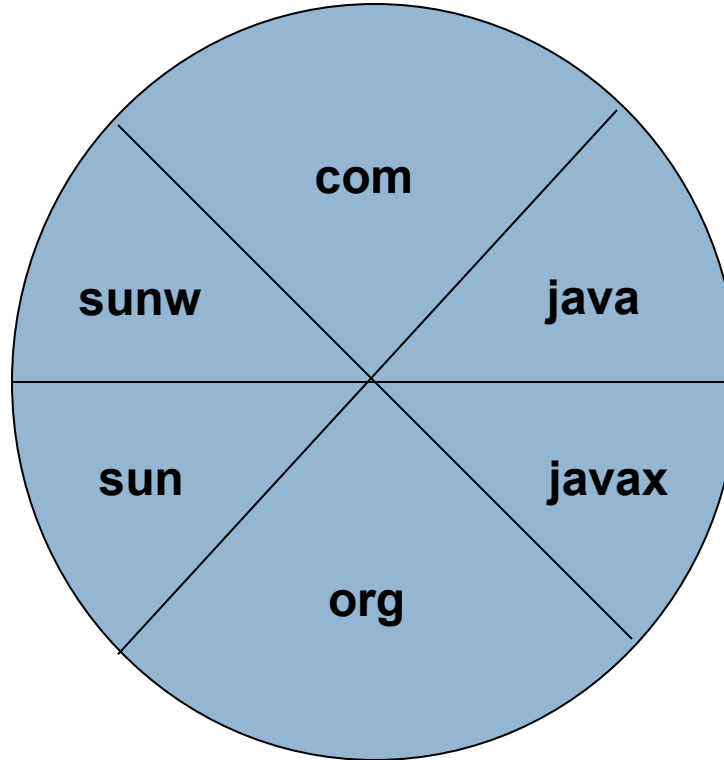
Conjunto de classes e interfaces que dão suporte ao desenvolvimento de aplicativos, trazendo soluções para problemas comuns a várias aplicações.



# Organização

165

Pacotes principais:



- A documentação que acompanha o J2SDK só contempla: java, javax e org

# java.lang

166

- java.lang é o único pacote que é importado automaticamente em todas as classes e interfaces Java.
- Por isso diversas vezes utilizamos as classes String e System e nunca precisamos fazer import delas.

# java.lang.Object

167

- Toda classe em Java é direta ou indiretamente uma subclasse de Object.
- Dois métodos importantes de Object:
  - String toString()
  - boolean equals(Object obj)

# java.lang.Object - toString

168

- A função do método `toString()` é retornar uma `String` que represente a classe.
- A implementação do `toString` em `Object` retorna uma `String` com a seguinte informação:
  - `nomeDaClasse@numeroidentidade`.
- O `toString` é invocado automaticamente quando mandamos imprimir um objeto na saída padrão (`System.out.println`) ou quando concatenamos `Strings`.
- Podemos re-implementar este método para que a `String` retornada contenha informações significativas de nossas classes.
- A assinatura do método `toString()` é:
  - `public String toString()`



# java.lang.Object - toString

169

Exemplo:

```
public abstract class Trabalhador {
    private String nome;
    private String sobrenome;

    public Trabalhador(String nome, String sobrenome){
        this.nome = nome;
        this.sobrenome = sobrenome;
    }

    public String toString() {
        return
        "[Nome: "+nome+", Sobrenome: "+sobrenome+"] ";
    }
    ...
}
```

# java.lang.Object - equals

170

- O método equals de Object compara o estado interno de dois objetos, diferente de com o uso do ==, que compara duas referências de memória.
- Cabe a cada programador reimplementar o equals conforme à regra de comparação existente entre suas classes.
- A assinatura do equals é:
  - `public boolean equals(Object obj)`

# java.lang.Object - equals

171

## Exemplo:

```
public abstract class Trabalhador {
    private String nome;
    private String sobrenome;

    public Trabalhador(String nome, String sobrenome) {
        this.nome = nome;
        this.sobrenome = sobrenome;
    }
    public boolean equals(Object obj) {
        if (this == obj) {
            return true;
        }
        if (!(obj instanceof Trabalhador)) {
            return false;
        }
        final Trabalhador trabalhador = (Trabalhador) obj;

        if (nome != null ? !nome.equals(trabalhador.nome) : trabalhador.nome != null) {
            return false;
        }
        if (sobrenome != null ? !sobrenome.equals(trabalhador.sobrenome) : trabalhador.sobrenome != null){
            return false;
        }
        return true;
    }
    ...
}
```

# java.lang.String

172

- String é a classe que representa os tipos textuais em Java.
- A classe String é uma classe marcada com o qualificador final. Isso quer dizer que ela não pode ter subclasses.
- Pelo fato de ser final, é que a chamada do seu construtor é opcional. Por isso podemos é que essas duas maneiras de instanciar objetos são equivalentes:
  - `String st = "Java";`
  - `String st = new String("Java");`

# TRATAMENTO DE EXCEÇÕES

Estrutura try-catch-finally, throws, throw

# Exceções

174

- Exceções são condições anormais que podem surgir enquanto um programa estiver sendo executado.
- Com certeza você já se deparou com alguma exceção na execução de seus programas Java.
- Vejamos um exemplo:

# Exceções

175

```
package br.edu.rm.cursojava.excecoes;

public class Excecao {
    public static void main(String[] args) {
        if (args.length > 0) {
            int num = Integer.parseInt(args[0]);

            System.out.println("O dobro de "+num+" é
"+num*2);
        }
    }
}
```

- O método `parseInt` da classe `Integer` converte uma `String` em um `int`.
- Se o argumento informado for a letra `A` o que vai acontecer durante a execução?

# try-catch

176

- Para evitar que erros como estes aconteçam, o programador pode isolar o bloco de instruções que podem gerar uma exceção e dar um tratamento padrão quando para estas exceções.
- Este isolamento é feito através da estrutura try-catch.
- A sintaxe do try-catch é:

```
try {  
    <bloco protegido>  
} catch (<tipoDaExceção1> <nomeDaExceção1>) {  
    <tratamento1>  
    [throw <exceçãoLançada1>]  
} catch (<tipoDaExceçãoN> <nomeDaExceçãoN>) {  
    <tratamentoN>  
    [throw <exceçãoLançadaN>]  
} finally {  
    <blocoDeFinalização>  
}
```



# Exemplo

177

```
package br.edu.rm.cursojava.excecoes;
```

```
public class Excecao {  
    public static void main(String[] args) {  
        if (args.length > 0) {  
            try {  
                int num = Integer.parseInt(args[0]);  
  
                System.out.println("O dobro de "+num+" é "+num*2);  
            } catch (NumberFormatException nfe) {  
                System.out.println(args[0]+" não é um inteiro válido!");  
            }  
        }  
    }  
}
```

# Mais um exemplo

□ Tente agora executar este código:

178

```
package br.edu.rm.cursojava.excecoes;

public class Excecoes1 {

    public static void main(String[] args) {
        if (args.length > 0) {
            int num1 = Integer.parseInt(args[0]);
            int num2 = Integer.parseInt(args[1]);
            System.out.println("Resultado:"+num1/num2);
        }
    }
}
```

- **Primeiro execute sem informar argumentos;**
- **Agora informe um só argumento;**
- **Informe dois argumentos não numéricos;**
- **Informe dois argumentos numéricos sendo o segundo zero.**
- **Vamos arrumar estas exceções!**

# finally

179

- O bloco finally é opcional na estrutura try-catch.
- Ele é utilizado para delimitar o bloco que será sempre executado, independente do sucesso ou não na execução das instruções contidas no try.
- Um bloco try pode ter vários catches mas só pode ter um finally.
- Um bom exemplo de utilização do bloco finally é uma tentativa de conexão com um banco de dados: independente do resultado, ela tem que ser encerrada ao final.

# throws

180

- A palavra `throws` é utilizada na assinatura do método para indicar que este método lança uma determinada exceção.
- As exceções são lançadas para fora dos métodos para transferir a responsabilidade do tratamento do erro pela classe que está utilizando cada método.
- Por exemplo, se você desenvolve uma classe que represente uma calculadora para ser utilizada por várias outras classes, como é que você vai saber como cada programador quer tratar o erro de divisão por zero? É melhor lançar a exceção para fora do método e deixar que cada um trate com preferir.
- Um método que lança a exceção `Exception` fica com a assinatura assim:

```
public void setIdade (int idade) throws Exception
```

# throw

181

- Quando utilizamos o throws apenas estamos marcando o método como um método que pode lançar uma exceção.
- O lançamento da exceção propriamente dita, é feito dentro do corpo do método com a palavra throw.
- Lançar uma exceção é nada mais, nada menos que criar ou repassar um objeto da família Exception para fora do método.
- Vejamos isso na prática completando o exemplo do slide anterior:

```
public void set Idade (int idade) throws Exception {  
    if (valor < 0) {  
        throw new Exception ("Não existe idade negativa!");  
    }  
    this.idade = idade;  
}
```

# throws e throw

182

- Como podemos ver, a utilização do throws e do throw está completamente ligada.
- As classes que utilizarem o método setIdade poderão tratar a exceção lançada ou repassar a exceção lançada.

# Exemplos

183

## Exceção sendo tratada

```
public void metodoQualquer() {  
    try {  
        setIdade(10);  
    } catch (Exception e) {  
        System.out.println("Erro na tentativa de atualização de idade!"  
            + e.getMessage());  
    }  
}
```

- **Exceção sendo repassada**

```
public void metodoQualquer() throws Exception  
{  
    setIdade(10);  
}
```

# OPERAÇÕES COM BANCO DE DADOS

JDBC





# JDBC – Java DataBase Connectivity

185

- JDBC é a API Java responsável pela conexão com banco de dados.
- Uma das formas de se acessar um banco de dados é através de uma fonte de dados ODBC.
- Basicamente, podemos dizer que o JDBC se utiliza do ODBC para se comunicar com o banco de dados.

# Configurando o ODBC

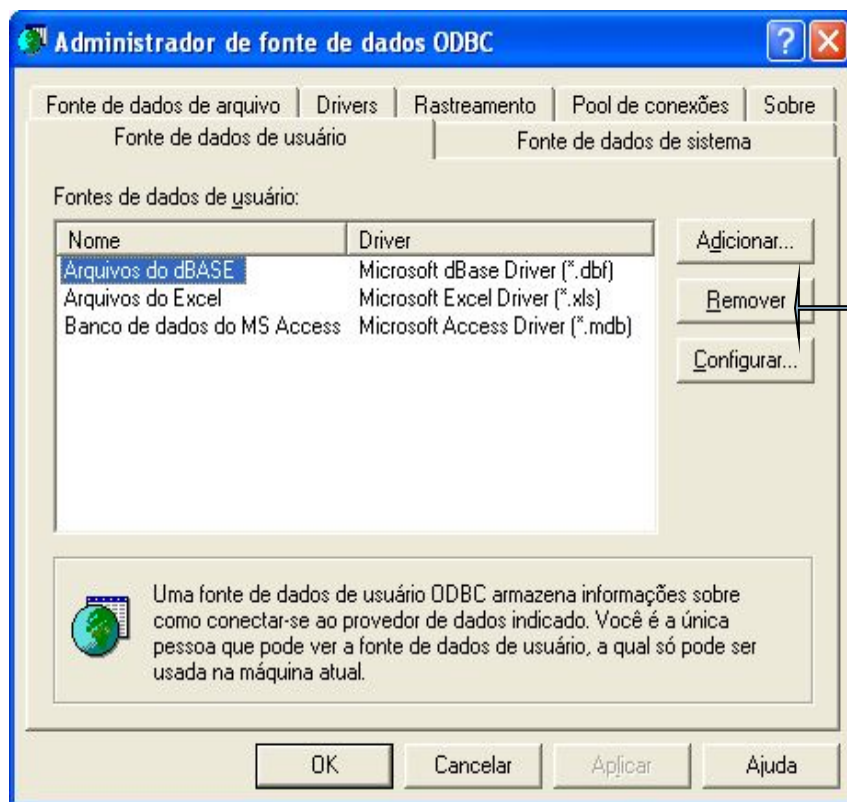
186

- A configuração de uma fonte de dados ODBC no Windows é relativamente simples.
- O “Administrador de fonte de dados ODBC” fica no grupo de “Ferramentas Administrativas” do “Painel de Controle”.
- Segue a instalação de uma fonte de dados para o banco de dados Access:

# Configurando o ODBC

187

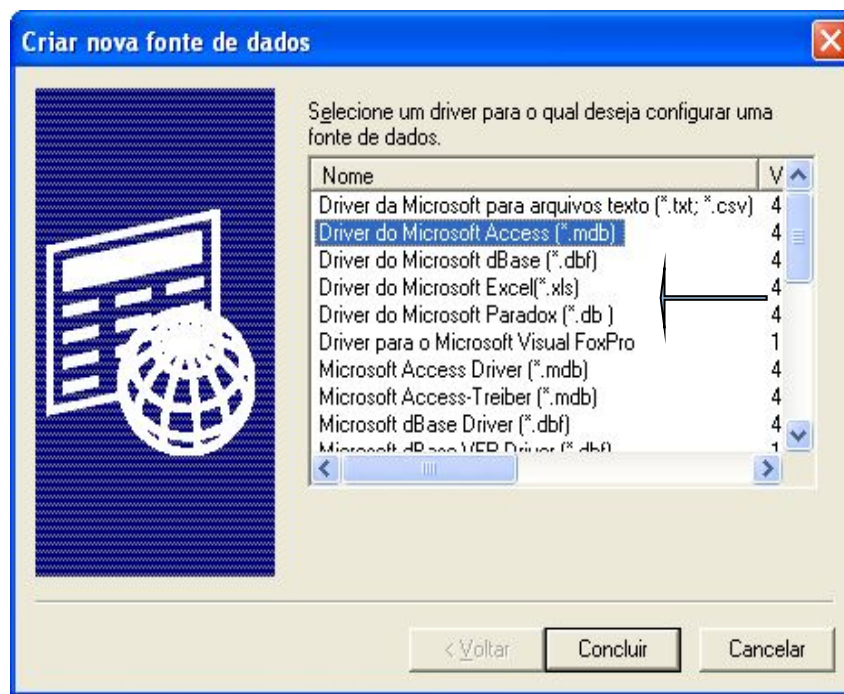
Clique em “Adicionar...”



# Configurando o ODBC

188

Em seguida, escolha o driver do Access e clique em concluir.



# Configurando o ODBC

189

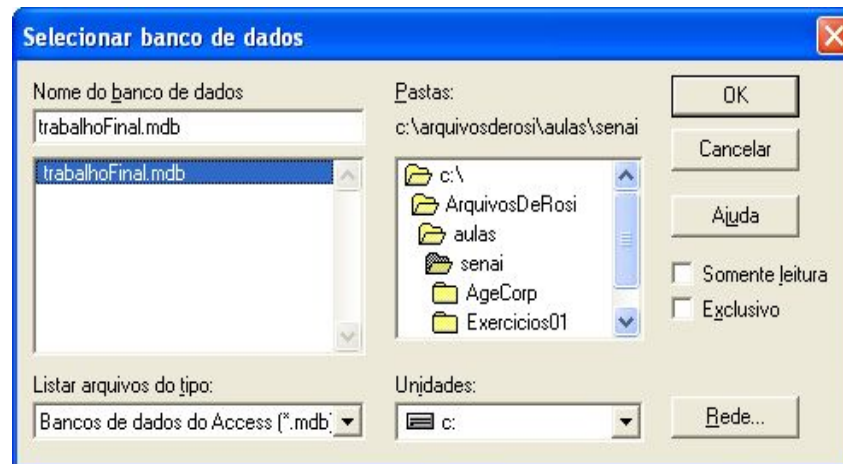
Escreva um nome e uma descrição para o seu banco de dados, em seguida clique em “Selecionar...”



# Configurando o ODBC

190

- Escolha o arquivo (\*.mdb) do seu banco de dados.
- Depois é só confirmar (botão OK) todas as telas seguintes.



# Fazendo uma conexão

191

- As tarefas que envolvem uma conexão com um banco de dados são:
  - Registrar o driver que será utilizado:
    - Para driver ODBC, é preciso registrar a classe `sun.jdbc.odbc.JdbcOdbcDriver` que representa a ligação entre a JDBC e os drivers configurados no SO.
  - Solicitar ao gerenciador de drivers a abertura da conexão
    - A classe que faz o gerenciamento dos drivers JDBC é `java.sql.DriverManager`.
    - O método estático `getConnection` de `DriverManager` recebe como parâmetro uma `String` que representa a fonte de dados com a qual se deseja conectar, e retorna uma instância da interface `java.sql.Connection`, que representa uma conexão aberta com o banco.

# Exemplo

192

```
import java.sql.SQLException;
import java.sql.Connection;
import java.sql.DriverManager;

public class Conexao {
    public static void main(String[] args) {
        Connection conn;
        try {
            //Registrando o Driver
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            //Abrindo a conexão com o banco
            conn = DriverManager.getConnection("jdbc:odbc:TrabalhoFinal");
            System.out.println("Conexão estabelecida!!");
            //Fechando a conexão
            conn.close();
            System.out.println("Conexão fechada!!");
        } catch (ClassNotFoundException cnfe) {
            System.out.println("A classe JdbcObdcDriver não foi encontrada!");
        } catch (SQLException sqle) {
            System.out.println("Conexão falhou!");
        }
    }
}
```



# Operações de Atualização

193

- Para fazer operações de inclusão, alteração e exclusão de registros em tabelas de banco de dados, precisamos da interface `java.sql.PreparedStatement`.
- A interface `PreparedStatement` tem a única função de enviar instruções SQL para o banco e capturar o resultado produzido.
- Para instanciar uma interface `PreparedStatement`, precisamos ter uma conexão aberta e invocar o método estático `prepareStatement`.
- O método `prepareStatement` recebe como parâmetro o comando SQL a ser executado.
- Veja no próximo slide a nossa classe `Conexao` instanciando uma interface `PreparedStatement`.

```

import java.sql.SQLException;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;

public class Conexao {
    public static void main(String[] args) {
        Connection conn;
        PreparedStatement stmt;
        try {
            //Registrando o Driver
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            //Abrindo a conexão com o banco
            conn = DriverManager.getConnection("jdbc:odbc:TrabalhoFinal");
            System.out.println("Conexão estabelecida!!");

            //Instanciando a interface PreparedStatement
            stmt = conn.prepareStatement("select * from pessoa");

            //Fechando a conexão
            conn.close();
            System.out.println("Conexão fechada!!");
        } catch (ClassNotFoundException cnfe) {
            System.out.println("A classe JdbcOdbcDriver não foi encontrada!");
        } catch (SQLException sqle) {
            System.out.println("Ocorreu um erro!");
        }
    }
}

```

— Porque será que é interessante mudar esta mensagem? Talvez o finally comece a fazer sentido agora!

# Operações de Atualização

195

- Com o PreparedStatement devidamente instanciado, já é possível utilizá-lo para atualizar o nosso banco de dados.
- As operações de atualização (inclusão, alteração e exclusão) são enviadas para o banco de dados através do método executeUpdate() da interface PreparedStatement.
- As classes Atualizacao e ProgramaAtualizacao ilustram as operações de insert, update e delete em uma tabela.

# Operações de Consulta

196

- Para consultar os dados armazenados na nossa tabela pessoa, vamos precisar de mais um conjunto da interface `java.sql.ResultSet` e da classe `java.util.ArrayList`.
- Vamos compreender o seu papel observando as classes `Consulta` e `ProgramaConsulta`.
- A novidade neste conjunto de classes é a criação da classe `Pessoa` que é utilizada representar cada registro da tabela pessoa.

# Referências

197

- Santos, Rui Rossi dos. *Programando em Java 2 Teoria e Aplicações*. Rio de Janeiro: Axcel Books, 2004.
- Caelum – Ensino e Soluções em Java. *Java e Orientação a Objetos*. São Paulo: 2004.

É vedada a utilização deste material para ministrar cursos.

# Respostas

198

```
class ChessProblem {
    public static void main(String[] args) {
        long[][] chess = new long[8][8];
        long valor = 1;
        for(int i = 0; i < chess.length; i++) {
            for(int j = 0; j < chess[i].length; j++) {
                if (i == 0 && j == 0) {
                    valor = 1;
                } else if (j == 0) {
                    valor = chess[i-1][7] * 2;
                } else {
                    valor = chess[i][j-1] * 2;
                }
                chess[i][j] = valor;
            }
        }
    }
}
```