

Apostila Java Progressivo



www.javaprogressivo.net

Apostila Java Progressivo



www.javaprogressivo.net

Índice

Básico 13

Por onde Começar: instalando o JDK e o NetBeans 14

Instalando o necessário para programar em Java 14

Criando o primeiro programa em Java 17

Como programar em Java do básico 17

Possíveis problemas com o NetBeans 21

Código comentado do primeiro programa 23

O que são Classes e Métodos em Java 23

Código comentado do primeiro programa em Java 24

Saídas Simples usando print, println e printf 28

Como exibir um texto em Java 28

Exercícios de Saída Simples 31

Exercícios sobre print, printf e println em Java 31

Comentários e Delimitadores em Java 35

Como fazer comentários nos códigos Java 35

Como usar os delimitadores /* */ em Java 36

Tipos Numéricos - int (inteiro), float e double (decimais ou reais) | Java Progressivo 38

Declarando variáveis em Java 38

Inicializando uma variável 39

Recebendo dados do usuário: a classe Scanner 41

Importando (import) classes (class) e pacotes (packages) 41

Recebendo dados do usuário: new Scanner(System.in) 42

Recebendo outros tipos de dados, float: nextFloat(); 45

Operações Matemáticas: Adição, Subtração, Multiplicação, Divisão, Resto da Divisão (módulo) e precedência dos operadores 48

Operações Matemáticas em Computação 48

Adição, Subtração e Multiplicação 48

Formatação com printf 49

Divisão 51

Resto da divisão 52

Precedência dos operadores e uso dos parênteses 53

Fazendo comparações: os operadores maior (>), menor (=), menor igual (<=), igual (==) e diferente (!=) 55

Como fazer comparações em Java 55

O tipo char: armazenando e representando caracteres 58

Tipo char: O que é? Onde é usado ? 58

Declaração do tipo char 58

Como armazenar um caractere (char) que o usuário digitou 60

O tipo boolean: a base da lógica 62

Booleanos em Java 62

Declaração de tipos boolean 63

Operadores lógicos e de negação: && (E ou AND), || (OU ou OR) e o ! (negação) 64

Operadores lógicos E (&&) e OU (||) em Java 64

Formalizando 64

Negando declarações: ! 65

Testes e Laços 66

Como usar IF e ELSE: Testando e Escolhendo o que executar 67

Como usar o IF em Java 67

Como usar ELSE em Java 68

Questões de IF e ELSE 70

Programa em Java que calcula as raízes e resolve uma equação do segundo grau 71

Passo 1: 71

Passo 2: 71

Passo 3: 71

Passo 4: 71

Passo 5: 72

Aplicativo em Java 73

Parte 1: Tendo a certeza que o usuário vai digitar uma nota válida 74

Parte 2: Checando se passou direto 75

Parte 3: Checando se você ainda há esperanças 77

Exercício: 78

Operadores Matemáticos de Incremento (++) e Decremento (--) 80

Operadores de incremento e decremento em Java 80

Somando e Subtraindo variáveis em Java 81

a++ e a-- 82

Operadores de Atribuição: +=, -=, *=, /= e %= | Java Progressivo 84

Operadores de atribuição em Java 84

Atalhos que salvam vidas 85

Operadores de atribuição e de incremento e decremento: diferenças entre a=++b e a=b++ 87

Diferença de a=++b e a=b++ em Java 87

Explicações 1. a = b++ 87

2. a = ++b 87

Laço WHILE 89

Como usar o laço while em Java 89

Contando até 10 em Java, com o laço while 90

Fazendo um PA (progressão aritmética) com o laço while, em Java 90

Fazendo um PG (progressão geométrica) com o laço while, em Java 91

Questões envolvendo laço WHILE 92

Loop infinito, controlando laços e loopings com o while 94

Loop infinito em Java 94

Criando um loop infinito simples 94

Controlando os laços e looping 95

Laço FOR: tendo um maior controle sobre as repetições 98

Diferenças entre o while e o for: para que serve o laço for em Java 98

Como usar o laço for em Java 99

Exemplo 1: Contando até 10, com laço for 99

Exemplo 2: contagem regressiva, usando o laço for 100

Exemplo 3: contagem progressiva e regressiva no mesmo laço for 100

Questões usando o laço FOR 102

Exercícios sobre o laço FOR em Java 102

Solução 09: 104

Passo 1: 104

Passo 2: 104

Passo 3: 104

Passo 4: 104

Segunda maneira: 105

O laço do ... while: O laço que sempre acontece...pelo menos uma vez 106

Como usar o laço DO WHILE em Java 106

Exemplos de uso 106

Os comandos break e continue: interrompendo e alterando fluxos e loopings 109

Para que servem os comandos BREAK e CONTINUE em Java 109

O comando break 110

Exemplo de uso: 110

O comando continue 112

O comando switch: fazendo escolhas em Java 114

O comando SWITCH em Java 114

Declaração e Sintaxe do comando switch 114

Exemplo de uso do comando SWITCH em Java: 115

O real funcionamento do comando switch: sem o comando break 117

Exemplo da utilidade do comando switch sem o break 118

Mais um exemplo útil do comando switch sem o break 119

Problemas envolvendo laços 122

Exercícios sobre laços em Java 122

Solução do desafio: 123

Parte de cima do diamante 123

Parte de cima do diamante 124

Apostila de Java, Capítulo 3 - Variáveis primitivas e Controle de Fluxo - Tipos primitivos e valores 127

Página 30, Exercício 3.3: Variáveis e Tipos primitivos 127

Enunciados 127

Soluções 128

Página 40, Exercício 3.13: Fixação de Sintaxe 129

Enunciados 129

Soluções 131

Página 41, Exercício 3.14 - Desafios: Fibonacci 139

Orientação a Objetos, parte I: Criando e declarando classes - Construtores 141

Introdução: O que são e para que servem as Classes e Objetos 141

O que são Classes e Objetos em Java 141

Utilidade das Classes e Objetos em Java 142

Agrupar coisas semelhantes 142

Crie uma vez, use quantas vezes quiser 143

Altere uma parte do código, e a mudança se propagará em todo o código 143

Classe à parte, vida à parte 143

Como saber quando usar Classes e Objetos em Java 144

Como criar uma Classe e Declarar Objetos 146

Criando uma classe em Java 146

Declarando um objeto de uma classe em Java 147

Acessando e modificando variáveis de Classes e Objetos 149

Variáveis de métodos e Variáveis de Classes 149

Criando classes com atributos 150

Alterando e Acessando atributos de uma classe 150

Construtor padrão e com parâmetros: o que são, para que servem e como usar
152

O que são construtores/constructor em Java 152

O que é e como utilizar o construtor padrão em Classes 153

Criando um construtor que recebe parâmetros 155

Como iniciar Objetos com parâmetros 157

Classes com mais de um construtor 157

Métodos em Java 160

Métodos: Introdução, o que são, para que servem, como e quando usar os methods 161

161

Introdução aos métodos em Java 161

A utilidade dos métodos em Java 162

Como declarar métodos em Java 163

Usando/chamando métodos 164

Aplicativo: menu simples usando métodos, laços e o comando switch usando um método para exibir um menu de opções 166

Programa em Java: Como criar um menu 166

O comando RETURN: obtendo informações dos métodos 169

Return: Retornando informações úteis em Java 169

Retornando inteiros, floats, doubles... em Java 170

Retornando uma String em Java 171

Retornando boolean em Java 172

Parâmetros e Argumentos: passando informações para os métodos 173

Diferença entre parâmetro e Argumento em Java 173

Declarando métodos com parâmetros 173

Exemplo: Passando argumentos para um método - Função que calcula o quadrado de um número 174

Exemplo: passando uma lista de parâmetros para um método - Cálculo do IMC 176

Exemplo: chamando um método dentro do outro - Cálculo do IMC 178

Classe Math: constantes, principais métodos e chamando métodos de outras classes 180

Usando métodos de outras classes em Java 180

A Classe Math (Class Math) do Java 181

Constantes da classe Math: 181

Exponencial e potenciação na classe Math 181

Calculando a Raiz quadrada em Java através da classe Math 183

Calculando logaritmos naturais em Java através da classe Math 183

Calculando senos, cossenos, tangentes e outras funções trigonométricas em Java através da classe Math 183

Módulo, máximo, mínimo e arredondamento em Java através da classe Math 184

Mais métodos matemáticos em Java 184

Sobrecarga de métodos (method overloading): declarando métodos com o mesmo nome 185

Overloading ou sobrecarga em Java 185

Varargs - passando uma lista de argumentos, de tamanho qualquer, para um

método 187

Sintaxe: 187

Exemplo de uso 187

Exercício: 189

Questões envolvendo métodos 190

Exercícios sobre métodos em Java 190

Jogo: adivinhe o número sorteado pelo computador 195

PASSO 1: Computador gera um número entre 1 e 1000 195

PASSO 2: looping principal do programa 196

PASSO 3: tentativa incrementada 196

PASSO 4: analisando o palpite 196

Orientação a Objetos, parte II: Os métodos set e get - Composição – Enum 199

Auto-referência com o this - Invocando métodos de Classes e Objetos 199

Referenciando membros da classe com this 199

Como invocar métodos de objetos que criamos 201

set e get: o que são e como usar esses métodos de forma correta 204

O que são e para que servem os métodos get e set 204

Como usar os métodos get e set 205

Como invocar métodos de dentro do construtor 207

Uso errado do get e set 208

Exercício: Caixa Eletrônico em Java - Caixa.java 209

Aplicativo: Simulação simples de conta bancária 211

Aplicativo: Conta bancária/Caixa eletrônico simples em Java 211

Use a main só para iniciar o aplicativo 211

Sistema bancário simples em Java 212

Código fonte Java do Aplicativo 213

Composição: trocando informações entre objetos 217

O que é Composição em Java 217

Para que serve a Composição em Java 217

Herança x Composição 218

Exemplo de uso de Composição em Java 219

controleHorario.java 219

Hora.java 220

Funcionario.java 221

Código do nosso programa: 222

controleHorario.java 222

Hora.java 222

Funcionario.java 223

Exercício: Aplicativo Java para um Supermercado 224

Use constantes, e não números - declarando variáveis com o final Suponha que o governo te contratou para criar um aplicativo que, dentre outras coisas, analisa se um cidadão é maior de idade ou não. 226

Por que usar constantes ao invés de números? 226

final - declarando variáveis constantes 228

final em Classes e Objetos 229

enum: A melhor maneira para manusear constantes 230

O que é enum em Java 230

Declarando uma enum em Java 230

Usando enum em Java 231

Exemplo de uso de enum em Java 232

static - Usando membros estáticos em Java 235

O que é static em Java 235

Como declarar uma variável static em Java 236

Quando usar variáveis static em Java 236

Exemplo de código com static 237

Apostila de Java, Capítulo 4 - Orientação a objetos básica 238

Página 51, 4.12 Exercícios: Orientação a Objetos básica 238

Enunciados 238

SOLUÇÕES 243

Página 55, Desafios 247

Enunciados 247

Soluções 248

Apostila de Java, Capítulo 6 - Modificadores de acesso e atributos de classe 251

6.8 - Exercícios: Encapsulamento, construtores e static 251

6.9 - Enunciado dos desafios 253

Solução das questões do capítulo 6 da apostila 254

6.9 - Solução dos desafios 261

Jogo: Campo Minado em Java 263

Como jogar o nosso Campo Minado em Java 263

Código do jogo Campo Minado em Java 264

-->campoMinado.java 264

-->Jogo.java 264

-->Tabuleiro.java 265

Campo Minado em Java: código comentado 269

Campo Minado em Java: A classe Tabuleiro.java 269

Fazendo uma jogada: 271

Campo Minado em Java: A classe Jogo.java 272

Programação Gráfica em Java, parte I: Caixas de Diálogo 274

Programação gráfica em Java, GUI e 2D: Introdução 274

Programação Gráfica, GUI e desenhos 2D em Java 274

Exibindo mensagens através das caixas de diálogo 276

Caixas de diálogo em Java 276

Exibindo mensagens nas caixas de diálogos 276

Recebendo dados do usuário através das caixas de diálogo 279

Como usar as caixas de diálogo para receber dados do usuário em Java 279

Recebendo informações do usuário através das caixas de diálogo 279

Como passar variáveis do tipo String para int, float e double 282

Transformando string em inteiro em Java 282

Transformando string em float 283

Transformando string em double 284

Exercício: 284

Aplicativo gráfico: mostra as raízes de uma equação do segundo grau 285

Exercício: 285

Faça um programa que receba os coeficientes de uma equação do segundo grau e retorne suas raízes, mesmo as complexas, através de caixas de diálogo. 285

Programa em Java 285

Passo 2: 285

Passo 3: 285

Passo 4: 286

Construindo (build) seu projeto Java no NetBeans 288

Transformando seu programa em Java em executável 288

Clean and Build Project no NetBeans 289

Estrutura de Dados, parte I: Array e ArrayList 290

Introdução aos tipos de Estrutura de Dados em Java 290

Estrutura de dados em Java 290

Estrutura de dados em Java: Array e ArrayList 291

Estrutura de dados em Java: Lista 292

Estrutura de dados em Java: Fila 292

Estrutura de dados em Java: Pilha 293

Arrays: como declarar, usar e acessar os elementos de um array 294

Declarando Arrays(vetores) em Java 294

A ordem de numeração dos elementos 295

Como usar os elementos de um array em Java 296

Exemplos de código Java usando Array 297

O laço for para Arrays: o laço For each 300

Fazendo o for percorrer todo o Array/Vetor 300

Exemplo de uso do foreach em Java: 302

Array Multidimensional ou Matriz: Array de arrays 303

Matrizes ou Vetores multidimensionais: Conjunto de vetores ou arrays em Java 303

Arrays de uma ou mais dimensões em Java 304

Matrizes: declarando vetores/arrays multidimensionais em Java 305

Exemplos de códigos: 305

Arrays em métodos: passagem por valor e passagem por referência 308

Passando arrays/vetores para methods/métodos 308

Passagem por valor e passagem por referência 310

Os tipos de referência em Java 311

Exemplo de como funciona a passagem dos tipo referência 312

Exemplo de passagem do tipo referência na vida real 314

Classe Arrays (Arrays Class): aprenda a manusear (copiar, ordenar, buscar e manipular) Arrays 316

Como usar a Arrays Class 316

Ordenando e buscando um elemento em um Array no Java 317

Métodos da classe Arrays (Array class methods): 318

Como usar ArrayList em Java: principais métodos 320

O que são ArrayList em Java 321

Como declarar e usar ArrayList em Java 321

Exemplo de uso do ArrayList 321

Apostila de Java, Capítulo 5 - Um pouco de Arrays 324

Página 71, Exercícios 5.5: Arrays 324

Enunciados 324

Soluções 327

Página 75, Desafio 333

Enunciado 333

Solução 333

Jogo: Batalha Naval em Java 334

Regras do Jogo Batalha Naval em Java 335

Como jogar: 335

Para os programadores Java: 335

Legenda do tabuleiro: 335

Métodos: 336

Lógica do problema 336

Jogos com programação gráfica em Java 342

Orientação a Objetos, parte III: Herança e Polimorfismo 343

Herança em Java - o que é, para que serve, exemplos e quando usar 343

O que é herança em Java - Superclasse e subclasse 344

Quando e como saber que é hora de usar herança em Java - Relação 'é um' 346

Hierarquia - subclasse de subclasse 347

Outra vantagem da Herança - Organização 347

Herança de construtores e Override 349

Sintaxe de declaração da Herança 349

Herança: o construtor da Subclasse sempre invoca o construtor da Superclasse 350

Herança: quando a superclasse não tem método construtor 353

O que é e como fazer Override em Java 354

super: Chamando o construtor da Superclasse 356

Herança ou Composição: Qual o melhor? 358

Herança ou Composição? Qual usar? 359

Herança: vantagens e desvantagens 359

Composição: vantagens e desvantagens 360

Delegando na Composição 361

Onde estudar mais sobre Herança e Composição 362

Interface em Java (implements) - O que é, para que serve e como implementar 362

O que é uma Interface 363

Interface em Java 363

Interface de um restaurante 364

Interface de um programa real 364

Como declarar uma Interface 365

Implementando uma classe - implements 366

Como comparar objetos - Classe abstrata Comparable e o método compareTo 368

Comparando objetos em Java 369

A classe abstrata Comparable e o método compareTo() 370

Exemplo de código - Comparando Objetos 371

Carro.java 373

classeComparable.java 374

Exercício de Java 374

private, public e protected: Protegendo suas informações em Java 375

Herança de atributos public 376

Herança de atributos private 376

Herança de atributos protected 378

A classe Object: o que é, uso e principais métodos 381

O que é e para que serve a classe Object 382

Métodos da classe Object: 382

Outros métodos: 383

Exemplo dos métodos da classe Object: 383

Polimorfismo em Java: o que é, pra que serve, como e onde usar 385

Definição de polimorfismo em Java 386

Exemplo 1 de Polimorfismo: Aumento no preço dos carros 386

Exemplo 2 de Polimorfismo: animais mugindo, latindo, berrando... 387

Polimorfismo: Classes abstratas e Métodos abstratos 389

O que são classes abstratas em Java 389

Classes abstratas no mundo real 390

Para que servem as classes abstratas e os métodos abstratos em Java 391

Exemplo de código em Java: Polimorfismo e abstração dos animais 392

Importância do Polimorfismo e Abstração em softwares 395

Exemplos de polimorfismo e abstração em aplicações: 395

Jogos de Carro, Flight Simulator, plugins e linguagens de programação 395

Manipulando polimorficamente subclasses 398

Superclasse abstratas se tornando subclasses concretas 398

Código Java de como manipular polimorficamente as subclasses 399

Descobrimo a classe de um objeto: instanceof 402

Apostila de Java, capítulo 07 - Herança, reescrita (override) e polimorfismo 405

7.7 Exercícios: Herança e Polimorfismo 405

Questão 01: 405

Questão 02: 405

Questão 03: 406

Questão 04: 407

Questão 05: 407

Questão 06 (opcional): 408

Questão 07 (opcional): 408

Questão 08 (Opcional): 409

Questão 09 (Opcional): 409

Questão 10 (Opcional, Trabalhoso): 409

Solução comentada das questões do capítulo 7 da apostila de Java 410

Questões 01, 02, 03 e 04 410

Questão 05: 412

Questão 06 e 07: 412

Questão 08: 413

Questão 09: 414

Questão 10: 414

Jogo da Velha em Java 418

Como Jogar 418

Para os programadores Java: 419

Como criar um Jogo da Velha em Java 419

Código Java do Jogo da Velha, modo texto Humano x Humano 419

Código comentado sobre como criar um Jogo da Velha em Java 427

Classe JogoDaVelha.java 427

Classe Tabuleiro.java 427

Classe Jogo.java 428

Classe Jogador.java 429

Classe Humano.java 430

Classe Computador.java 431

Strings e Caracteres: Escrevendo em Java 432

Java: A Classe String 432

Java: A Classe StringBuilder 435

Java: A Classe Character 437

Java: Expressões Regulares (regex) em Java 439

Programação Gráfica em Java, pt II: desenhos, fontes e figuras geométricas em 2D 443

JFrame e JPanel - Introdução ao estudo de GUI 443

O que é JFrame 443

O que é possível fazer com JFrame 444

O que é JPanel 445

JFrame e JPanel: como criar uma aplicação gráfica em Java 446

JFrame: Como criar Janelas em Java 446

JPanel: Inserido elementos em um JFrame 448

Desenhando Linhas 451

Desenhando Linhas em Java 451

Exemplo de código: Desenhando uma linha em Java 452

Criando o JPanel 452

Criando o JFrame 452

O que são e como usar os métodos getWidth() e getHeight() 455

Os métodos getWidth() e getHeight() 455

Exemplo de código: Desenhando duas linhas em Java, usando getWidth() e getHeight() 456

Criando alguns desenhos interessantes apenas com Linhas 458

Primeiro desenho: 458

Segundo desenho: 460

Exercício: 462

Como desenhar retângulos e quadrados 463

O método drawRect: como desenhar retângulos e quadrados em Java 463

Desenhando um Cubo 465

Exercícios sobre linhas & retângulos: 467

Como colocar um menu de cores: Usando o JcolorChooser 468

JColorChooser - Escolher cores em Java 468

- Mais métodos 469

Como usar as fontes em Java 473

A classe FONT: Escolhendo Fontes e efeitos em Java 473

Como desenhar Polígonos e Polilinhas em Java 476

Desenhando polígonos e Polilinhas 476

`public void drawPolygon(int[] xPoints, int[] yPoints, int points)` 476

`public void drawPolyline(int[] xPoints, int[] yPoints, int points)` 476

`public void drawPolygon(Polygon p)` 476

`public void fillPolygon(int[] xPoints, int[] yPoints, int points)` 476

`public void fillPolygon(Polygon p)` 477

`public Polygon()` 477

`public Polygon(int[] xValues, int[] yValues, int numberOfPoints)` 477

`public void addPoint(int x, int y)` 477

A incrível API Java 2D 478

API Java 2D 478

Programação Gráfica em Java, pt III: GUI - Graphic User Interface 480

JLabel - Como criar rótulos com textos e imagens em frames 480

JLabel - Exibindo rótulos (Textos e Imagens) 480

JLabel só com String 481

GUI.java 481

Rotulo.java 482

JLabel só com imagem 482

Rotulo.java 483

JLabel com imagem e texto 484

Rotulo.java 485

JButton - Como criar botões em aplicativos Java 487

JButton - O que são e para que servem os botões 487

JButton - Como usar botões em Java 488

GUI.java 488

Botao.java 489

Layout e organização de uma aplicação GUI 489

Botao.java 490

Imagens em JButton 491

O que um JButton faz - Eventos e Ações 491

Tratando evento e ações em GUI - Event Handling, ActionListener, ActionEvent e actionPerformed 493

GUI - Controlando eventos e ações 493

Como tratar eventos - A interface ActionListener e o método actionPerformed 494

Event Handling - Criando um tratador de eventos 495

ButtonHandler.java 496

O método addActionListener - Adicionando um tratador de eventos aos componentes 497

Botao.java 497

Main.java 498

Tratamento de eventos - Extends e Implements, Classe Interna e Objeto anônimo - Como mudar a cor de um JFrame 500

Estendendo o JFrame e implementando a ActionListener 500

public class Botao extends JFrame implements ActionListener 501

Alterando a cor do background (fundo) de um JFrame e o método repaint() 501

Main.java 502

Botao.java 503

Classe Interna em Java 504

Main.java 505

Botao.java 505

Objeto anônimo 506

Main.java 507

Botao.java 508

Afinal, qual a melhor maneira de tratar eventos em Java? 509

JTextField e JPasswordField - Como usar caixas de texto e de senha em Java 511

JTextField - Caixas de Texto 511

JTextField - Como criar e usar uma caixa de texto em Java 512

Exemplo sobre JTextField 512

Main.java 513

CaixaDeTexto.java 513

JPasswordField - Como criar caixas de senha 515

Main.java 515

CaixaDeTexto.java 516

Exercício de Java 517

JCheckBox - Como Usar Botões de Checagem (CheckBox ou Caixa de Seleção) (Tutorial de Java) 518

JCheckBox em Java - O Que É e Para Que Serve 518

Tutorial de JCheckBox - Como Usar Caixas De Seleção em Java 519

MyCheckBox.java 520

Main.java 520

Eventos em JCheckBox - Handler e Listener 521

MyCheckBox.java 522

JRadioButton - Botão de Rádio ou de Opção (Tutorial de Java GUI) 524

JRadioButton em Java - O Que É e Para Que Serve 524

Como Usar Radio Button em Java - JRadioButton 525

RadioButton Handler - Eventos com os Botões de Rádio 526

Código Fonte do Tutorial de Java 526

Main.java 527

RadioButton.java 527

A Classe ButtonGroup do Java 528

Código Fonte do Tutorial de Java 529

RadioButton.java 529

Arquivos (Files): Escrevendo (writing) , lendo (reading) , anexando (appending) e manipulando 532

Arquivos (Files) em Java 532

O que são Arquivos em Java 532

Para que estudar Arquivos em Java 533

Entrada, Saída e Tipos de dados - Fluxo (stream) e o pacote Java.io 535

Tipos de dados dos arquivos em Java 535

Fluxo (stream) - Entrada(in) e Saída(out) de dados 536

O pacote java.io 536

Como ler caracteres, Strings e Bytes de um arquivo em Java 538

Como ler caracteres de um arquivo em Java 538

Lendo mais de um caractere - Fim de arquivo 539

Como ler Strings de um arquivo em Java 540

Como ler bytes de um arquivo em Java 541

Fechando arquivos - O método close() 543

Java: Class File (a classe File) - Obtendo informações de arquivos e diretórios 544

Java: Class Formatter (a classe Formatter) - Escrevendo em arquivos 548

- Criando um arquivo de texto 548

Java: Class Scanner (a classe Scanner) - Lendo e Recebendo dados de arquivos 551

Mercado de Trabalho 554

Como se tornar um programador Java profissional 554

1. Estude. Estude mais, e de novo. Não pare. 555

2. Prove que estudou 555

Curso de Java Online com Certificado da Brava Cursos 557

Obter Certificado do Curso Online de Java da Brava Cursos 558

Básico

Essa sessão visa ensinar os conceitos básicos de Java e de Programação, de um modo geral.

Os seguintes tópicos são voltados para aqueles que nunca tiveram contato com programação, ou tiveram e já esqueceram.

Se não tem a mínima ideia do que fazer, por onde começar, o que baixar, clicar, digitar...calma, vamos te ensinar tudo, bem do básico mesmo.

Basicamente, você só precisa ser capaz de ligar seu computador e entrar na internet, o resto a gente te ensina com toda calma e nos mínimos detalhes, como programar em Java.

Vamos lá ?

Por onde Começar: instalando o JDK e o NetBeans

Ok, você está convencido!

Java é a melhor linguagem do mundo e você está perdendo tempo de vida a cada segundo que não estudar Java.

Mas, por onde começar? Aqui no site tem o curso.

Mas que programa usar? Tem que pagar algo? Onde que eu escrevo os códigos? O que tenho que instalar? Tem que baixar algo?

JDK? NetBeans? Hã?

Agora que você já leu sobre os artigos de programação, linguagens de programação e sobre a linguagem Java e está aqui, assumo que optou por iniciar o curso de Java.

Parabéns, muito inteligente você. Vejo que quer garantir seu lugar no mercado.

Mas não posso dizer que ao final do curso você já vai sair fazendo programas em Java.

Na verdade, durante o curso iremos fazer bastante isso...odeio só teoria.

Entre os diversos artigos daremos sempre uma pausa para fazer aplicativos como calculadoras, jogos, calcularemos a chance de ganhar na mega-sena, criaremos um banco de dados etc.

O mais importante é fazermos algo útil, não é?

Instalando o necessário para programar em Java

Bom, vamos instalar o Java para poder programar nele.

Provavelmente você já tem a JRE, que é o Java Runtime Environment, que serve para rodar as aplicações em Java, como o site daquele banco ou aquele jogo on-line.

Mas usar o Java é coisa do passado, agora você vai programar o Java.

Pra isso você vai precisar do JDK, o Java Development Kit, que é o Kit de Desenvolvimento em Java, que já contém o JRE!

Há duas maneiras de rodar aplicativos em Java, porém usarei só uma (por IDE), que é a mais usada e a mais simples.

A outra é por linha de comando, ou seja, através daquela telinha preta.

Pra rodar assim, você tem que ir pelo DOS ou pelo Terminal até a pasta que você criou os arquivos de código, compilar e rodar, usando os comandos 'javac' e 'java', mas é um processo lento e cansativo.

Vamos usar uma IDE, o NetBeans, que é um programa do próprio site da Oracle, detentora dos direitos do Java.

Esse programa já compila e roda automaticamente os códigos que você escreve, além de checar erros, auto-completar seus códigos, mostrar visualmente os arquivos de seu programa (se for um projeto grande, terá dezenas de arquivos) além diversas de outras funcionalidades.

Além da instalação ser totalmente automatizada. Bem mais simples, não?

Há diversos tipos de Java, como o para celular, o ME, o Card etc.

Usaremos o Java SE, para criar aplicações para computadores.

Note que você pode baixar o Java separadamente e instalar em seu computador e depois outra IDE.

No nosso curso, recomendarei a instalar o pacote 'Java + NetBeans', pois ele já vai instalar o Java e a IDE NetBeans e configurar automaticamente pra você.

Digite no Google: “jdk + netbeans”

O primeiro site que deve aparecer é o da Oracle, como:

<https://www.oracle.com/technetwork/pt/java/javase/downloads/jdk-netbeans-jsp-3413153-ptb.html>

PS: No momento que escrevo o artigo, as URL são estas. Vai mudando a medida que vão atualizando tanto a JDK como o o Netbeans.

No momento que você lê, as versões e URLs podem ser outras. Caso sejam outras, baixem a versão mais recente.

Evite dores de cabeça e sempre baixe dos sites oficiais!

Aceite os termos e condições clicando em 'Accept License Agreement'.

Depois escolha seu SO (Windows, Linux ou Mac) e baixe o arquivo.

Pronto, você já pode desenvolver e criar aplicações em Java.

Pra isso, simplesmente inicie o NetBeans.

Se digitar: “jdk” no Google, vai achar o site oficial para baixar somente a JDK e depois ir atrás de outra IDE (programinha pra programar), como o Eclipse.

Criando o primeiro programa em Java

Neste artigo iremos começar, de fato a programar.

Veremos o código inicial que será necessário escrever para iniciarmos nossos estudos na linguagem de programação Java.

É um dos tutoriais mais importantes de nossa apostila de Java, e é importante que você estude com atenção.

Como programar em Java do básico

Agora que já instalou o JDK e o NetBeans, está na hora de criar o seu primeiro programa em Java!

Abra seu NetBeans.

Vá em File, depois em New.

Você verá uma série de opções.

São as possibilidades de se programar em Java, os tipos de aplicações etc.

No nosso caso é 'Java' em 'Categories' em 'Projects' escolha 'Java Application'.

Clique em Next.

Dê o nome ao seu projeto em 'Project Name'. Isso é importante.

Escolha nomes fáceis, sem acentuação e sem espaços. Vamos escolher 'Primeiro'.

Em 'Project Location' ficará a pasta onde ficará guardado seus projetos. Então escolha um lugar fácil e bacana também. Crie um local se precisar. Clique em Finish.

New Java Application

Steps

1. Choose Project

2. **Name and Location**

Name and Location

Project Name:

Project Location:

Project Folder:

☐ Use Dedicated Folder for Storing Libraries

Libraries Folder:

Different users and projects can share the same compilation libraries (see Help for details).

☒ Create Main Class

☒ Set as Main Project

Pronto, você vai iniciar sua primeira aplicação em Java.

Note que apareceu uma tela com várias coisas escritas.

No futuro isso vai ser útil para você, mas por hora, não.

Vamos passar o código para você escrever, compilar e ver resultado. Depois explicaremos cada detalhe do código, como funciona, para que serve e tal.

Vamos lá, digite exatamente o código a seguir (sim, exatamente, pois a linguagem Java é case sensitive, ou seja, main é algo totalmente diferente de Main, e isso vale para classes, objetos, métodos, variáveis etc, como veremos ao longo do curso):

```
package primeiro;
```

```
public class Primeiro {
```

```
    public static void main(String[] args) {
```

```
        System.out.println("Meu primeiro programa em Java!");
```

```
    }
```

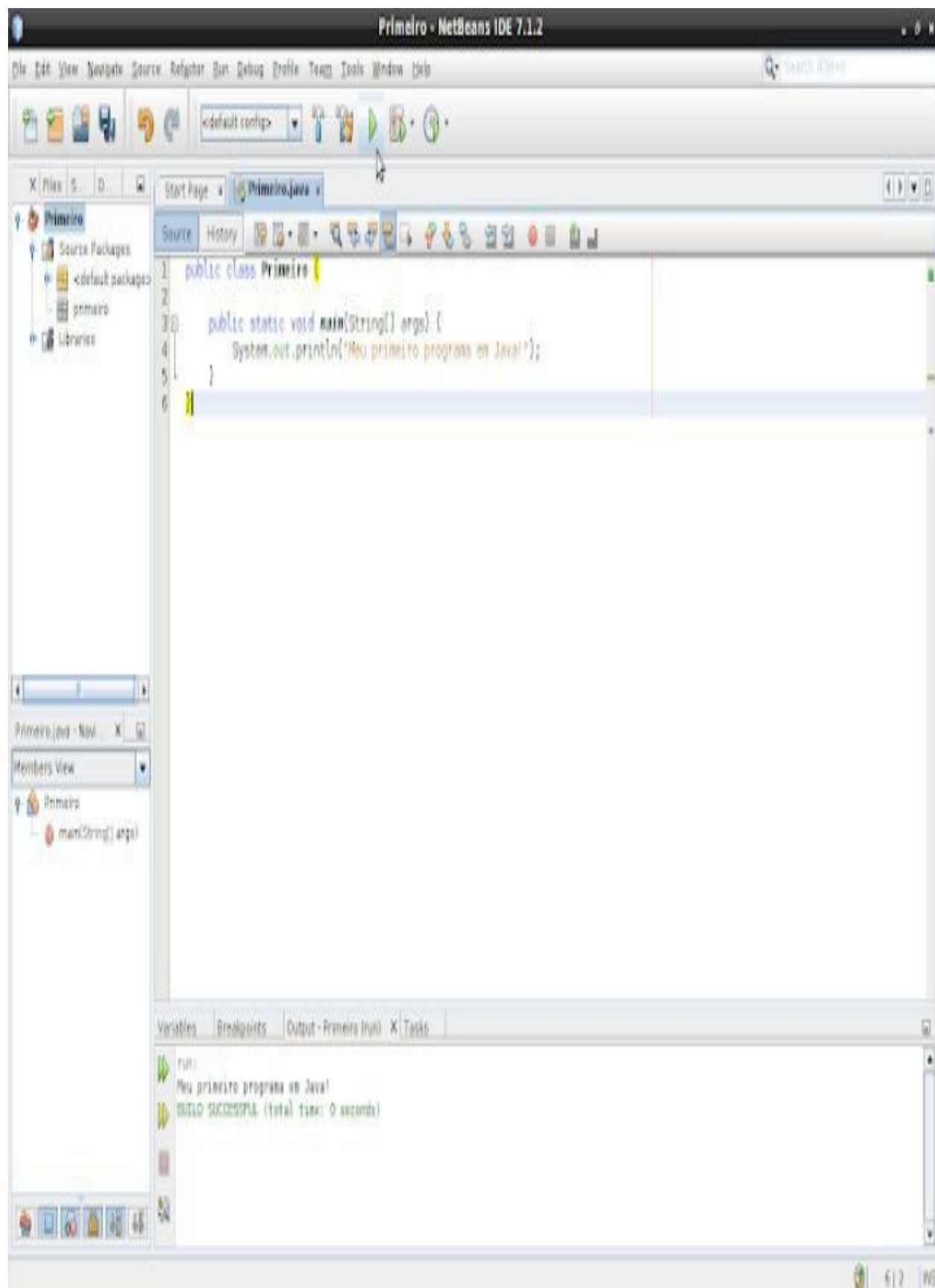
```
}
```

Está vendo aquela setinha verde ali? Do lado de um martelo e uma vassoura? Se

você pousar o mouse em cima verá 'Run Main Project'.

Clique nela.

Seu projeto será compilado e irá rodar.



O resultado do seu programa irá aparecer na tela debaixo, a frase: 'Meu primeiro programa em Java!'"

Caso não apareça, certifique-se de que não escreveu nada de errado.

Aqui vai uma dica preciosa: embora eu vá te mostrar os códigos, sempre, mas SEMPRE digite os seus códigos!

É um habito que tem que ter! Não fique copiando e colando, pois não vai aprender nada!

Porém, se digitar tudo que for passado aqui, aos poucos irá memorizando e sem fazer esforço algum, irá aprender naturalmente.

Caso tenha dado algum erro, poste na área de comentários.

Caso tenha dado tudo ok, parabéns, você já programou em Java. No próximo artigo explicarei o que você fez, o motivo disso tudo ter acontecido, faremos algumas alterações, erraremos de propósito para o NetBeans nos avisar do erro e tentarmos consertar etc.

PS: Note que, ao criar um projeto no NetBeans, na verdade aparece isso:

/*

* To change this template, choose Tools | Templates

* and open the template in the editor.

```

*/

package primeiro;

/**
 *
 * @author JavaProgressivo
 */

public class Primeiro {

    /**

    * @param args the command line arguments

    */

    public static void main(String[] args) {

        // TODO code application logic here

    }

}

```

Um erro comum é escolher o nome do projeto como "Primeiro" e digitar "public class primeiro". "Primeiro" é diferente de "primeiro".

O que vai nos interessar é do "public class..." em diante.

O que aparece nas letras em verde ou azul, são apenas comentários e não são levados em conta pelo Java na hora de rodar suas aplicações.

Porém, se você usar outra IDE, como Eclipse, irá aparecer outra coisa:

```
public class Primeiro {
```

```
    /**
```

```
    * @param args
```

```
    */
```

```
public static void main(String[] args) {
```

```
    // TODO Auto-generated method stub
```

```
    }
```

```
}
```

```
-----
```

Porém, como disse, o que vai importar (por hora) é o que está dentro do main().

O "package primeiro;" pode ficar ali, sem problemas, e se refere ao nome do seu projeto.

É como se seus aplicativos fizessem parte de um pacote, no caso, o nosso pacote é chamado de "primeiro".

Poderíamos criar um pacote chamado "matematica", com várias fórmulas e funcionalidades matemáticas.

Depois poderíamos criar um pacote chamado "Estatiticas" e usar seu pacote, previamente feito, "matematica" nesse pacote de estatística. Assim evitaria ter que programar tudo de novo.

Em breve, ao longo de nosso curso de Java online, aprenderemos sobre comentários e pacotes.

Possíveis problemas com o NetBeans

Bem, alguns leitores entraram em contato relatando alguns problemas.

Como pode ser dúvida de mais gente, vou relatar um aqui e sua solução.

O leitor Raphael, através da área de contato, disse que quando rodava o seu primeiro programa não aparecia nada, e quando me mandou um print screen, aparecia um erro, dizendo que não havia encontrando a classe main:

"Erro: não foi possível localizar nem carregar a classe primeiro. Primeiro"

Pois bem, como eu expliquei pra ele ao ver seu print:

O que pode acontecer é que, as vezes, estamos com vários projetos abertos no NetBeans, aí quando clica pra rodar, ele não sabe qual rodar e dá esse problema.

Vá ali no menu, escolha o projeto que quer rodar, clica com o botão direito em cima dele e vai em "Run".

Quando há algum erro em nossos programas, o programa avisa...note que ali no começo da linha tem umas lâmpadas...pousa o mouse ali em cima delas (tanto na lampada amarela com o sinal vermelho, como a lampada amarela), vai exibir algum aviso de problema ou sugerir alguma solução.

Por favor, quaisquer dúvidas e problemas que venham a ter, entrem em contato.

Suas dúvidas podem ser de outras pessoas, e isso faz crescer a quantidade de informação de nosso tutorial em Java, o curso Java Progressivo.

Código comentado do primeiro programa

Agora que você já é um programador Java - pois já fez um programa -, vamos entender o que você fez, o que é aquela sopa de letrinhas e para que serve cada parte daquele código bizarro.

Isso inclui uma explicação automobilística sobre Classes e Métodos.

O que são Classes e Métodos em Java

Já neste primeiro exemplo vou falar de dois dos mais importantes conceitos de Java, que você vai ouvir pelo resto de sua vida de programador Java:

classes e métodos.

Classe é a base de tudo, em Java. Classe é algo mais abstrato, é uma definição geral. Um exemplo ou instância de uma classe é um objeto.

Classe é algo mais genérico, objeto é algo mais específico.

Daí vem o fato de Java ser orientado à objetos.

Parece ser complicado e confuso, mas vou explicar e você vai ver que é simples.

Na verdade foi feito pra ser simples, pois imita nossa vida.

O carro, por exemplo, é uma classe. É algo mais geral.

Um exemplo de objeto, é o fusca. Um Honda Civic é um objeto da classe carro, mas é um objeto diferente do objeto fusca.

Bem óbvio, né?

As classes possuem atributos que todos os objetos possuem, mas que não são, necessariamente, iguais. Mas podem ser. Como assim?

Voltemos para os carros.

A classe carros tem o item 'motor'. Então o objeto fusca tem motor e o objeto Honda Civic também tem motor, embora esses motores sejam diferentes, pois a característica dos motores são diferentes.

A classe carros tem o item portas. No fusca, esse valor vale 2, pois só tem duas portas (a não ser que você tenha turbinado seu fusca ou uma porta tenha caído). Porém, outros objetos da classe Carro tem 4 portas.

Isso ajuda muito na hora de criar grandes projetos. Imagina que você tenha um trabalho numa empresa, com milhares de funcionários e serviços.

Fácil fácil seu programa em Java vai possuir milhares de objetos, mas se você se organizar bem só vai ter algumas dezenas de Classes.

Assim, você organiza o 'mundo' (que no caso é a empresa), em blocos, as Classes.

Então, quando fizer uma alteração, faz nas classes que todos os objetos, automaticamente, herdarão essa mudança.

Passe a ver o mundo como Classes e Objetos.

Por exemplo, antigamente só existiam sexo masculino e feminino.

Hoje em dia, é comum ter bem mais opções, como indefinido.

E aí, vai em cada um dos milhares de funcionários e colocar mais essa categoria?

Não ué, vai lá na classe 'funcionarios' e adiciona a opção 'indefinido' como mais uma opção de gênero.

Pronto, todos os objetos (ou seja, os funcionários, as pessoas), herdarão e terão essa característica, pois todos os objetos são instâncias da classe. Eles são a classe. Se a classe mudou, o objeto mudou.

Código comentado do primeiro programa em Java

No nosso caso, a nossa classe é 'Primeira'.

Agora vamos aos métodos!

Métodos são...métodos! Ou seja, são meios, ou jeitos de se fazer uma coisa. Em outras linguagens, são chamados de funções (C, C++) ou sub-rotinas (Perl).

O que o nosso método faz? Ele imprime (escreve, mas se acostume com a palavra imprimir) o texto 'Meu primeiro programa em Java!'.

Métodos podem calcular soma, subtração, integração etc. Existem métodos de formatação do HD, métodos de backup, métodos de invasão, métodos pra mostrar um texto, um menu, métodos que sorteiam um número...métodos são tarefas.

É uma porção de código que faz algo bem definido.

É um método pra fazer algo.

No nosso caso, o sistema ('System') joga pra fora ('out'), que no caso é a tela, na forma de escrita ('print') a mensagem 'Meu primeiro programa em Java!'.

Se você gosta de se questionar, então a resposta é sim. Assim como sai ('out'), pode entrar ('in'), que é quando o sistema ('System') recebe dados de você usuário ou de um arquivo - 'System.in', que você verá no futuro.

E sim, ele não escreve só na tela. Também é comum escrevermos, ou printarmos, para arquivos, para criarmos um registro de erros e eventos (logs), por exemplo.

Por hora, não se estresse com os 'public', 'static', 'void', 'String[]', 'args', eles são esmiuçados detalhadamente aos poucos por partes no âmago de suas entranhas íntimas e pessoais.

Agora as partes mais importantes desse começo:

1. O seu programa, pra rodar, precisa ter a 'main'.

A primeira coisa que o Java faz pra rodar é procurar o 'public static void main(String[] args)'. É daí que ele inicia o seu programa.

2. A classe que contém o método main, 'Primeira', precisa ter o mesmo nome do projeto.

Lembre-se que o nome do nosso projeto é 'Primeira' e a classe é 'public class Primeiro {}'

Ou seja, o método main tem que estar em algum lugar dentro desse par de colchetes aí.

Estes são os erros mais comuns que vejo em fóruns, que os iniciantes cometem. Esquecerem da main.

Agora vou usar com você o melhor método de aprendizado, o teste. Testem:

1. Em vez de main, escreva Main e compile/rode.
2. Em vez de 'public class Primeiro...' escreva 'public class primeiro' e

compile/rode

3. Tirem o ';', um '{' ou '}' e compile/rode

4. Que a diferença entre os códigos abaixo?

Esse:

```
public class Primeiro
```

```
{
```

```
public static void main(String[] args) {
```

```
System.out.println("Meu primeiro programa em Java!");
```

```
}
```

```
}
```

Pra esse:

```
public class Primeiro
```

```
{
```

```
public static void main(String[] args)
```

```
{
```

```
System.out.println("Meu primeiro programa em Java!");  
  
    }  
  
}
```

E em relação ao original?

```
public class Primeiro {  
  
  
  
  
public static void main(String[] args) {
```

```
System.out.println("Meu primeiro programa em Java!");  
  
    }  
  
}
```

Se você achava que tinha amigos, te apresento o que é amigo de verdade:
debugger

Esse cara te aponta os erros que você cometeu. As vezes mostra até a linha e exatamente o que você errou.

Sempre leia os erros e tente entender o que ele está dizendo.

Façam os testes e obterão as respostas, é o primeiro exercício. Leia a mensagem

de erro e interprete.

Se fizerem isso e constatarem os erros e a diferença, caso exista, com seus próprios olhos, irão aprender por experiência própria, que é o melhor jeito de se aprender algo.

E parabéns, tem muita gente 'avançada' que ainda erra essas coisas básicas.

Saídas Simples usando print, println e printf

Vamos falar a respeito das maneiras de mostrar mensagens na tela, que incluem:

`System.out.println`

`System.out.print`

`System.out.printf`

Não, isso não é bobagem...desde o momento que você liga o computador, até nos seus jogos e Facebook, mensagens são mostradas na sua tela.

São muito importantes e uma das funções mais usadas pelos dispositivos digitais.

Como exibir um texto em Java

Pra começar, vá no seu projeto 'Primeiro' e altere a linha:

```
System.out.println("Meu primeiro programa em Java!");
```

Por:

```
System.out.print("Meu segundo programa em Java!");
```

Além do fato de não ser mais o primeiro, mas segundo programa que você criou, qual outra diferença você notou após compilar e rodar? É capaz de descobrir só olhando?

É fácil ver que ao final da frase, a mensagem 'BUILD SUCCESSFUL (total time: 0 seconds)' não está mais abaixo da 'Meu segundo programa em Java!', e sim ao lado.

Já sei! Depois da frase, vou dar um enter! Vai ficar assim:

```
System.out.print("Meu segundo programa em Java!"  
+ "");
```

(Esse + "" apareceu sozinho)

Compilei, rodei e...deu na mesma.

Sim, por quê? Pro Java, dá na mesma. Pode apertar enter mil vezes que vai continuar igual.

O 'ln' de 'println' é de 'line', pois essa função imprime uma linha, e linha inclui uma quebra de linha (ou newline, ou \n, ou [enter], ou parágrafo).

Ou seja, a função 'print' não inclui essa quebra. Como colocar essa quebra no 'print'?

Existe um símbolo especial para isso, é o '\n'. Teste aí:

```
System.out.print("Meu segundo programa em Java!\n");
```

Agora foi né? Ok!

Então, como faríamos para aparecer na tela:

Programação

Progressiva

Assim:

```
System.out.println("Programação");
```

```
System.out.println("Progressiva");
```

Ou assim?

```
System.out.print("Programação\n");
```

```
System.out.print("Progressiva\n");
```

Qual destes códigos é o certo? Qual é o melhor? Por que usar um e não o outro?

Aqui vem um aspecto bacana da programação. Não exige maneira certa de escrever o código.

Se as duas formas tem o mesmo resultado, podemos usar as duas formas.

Vai depender de sua criatividade.

É bem comum quebrarmos a cabeça, fazermos um programa de centenas de centenas de linhas, aí vem alguém e resolve em 20 ou 30 linhas.

Programação depende do seu raciocínio, do seu jeito de pensar. É algo pessoal.

Teste agora:

```
System.out.printf("Programação Progressiva");
```

E depois:

```
System.out.printf("Programação Progressiva\n");
```

E por fim:

```
System.out.printf("Programação Progressiva"  
+ "");
```

Notou a diferença? Calma, não te trollei.

O 'f' de 'printf' é referente a formatação, serve pra quando você for colocar números, strings (textos), alinhar e fazer outros tipos de formatação usando as saídas.

Veremos isso melhor em outros artigos.

Então está na hora de praticar, pois você já está apto a resolver os 10 exercícios sobre saídas.

Depois que resolver, já pode se considerar mestre nas artes de mostrar mensagens na tela.

Exercícios de Saída Simples

As seguintes questões foram extraídas do material '300 ideias para Programar', de Virgílio Vasconcelos Vilela 10 questões pra você resolver, sobre saída.

Use print, println, printf...use a criatividade. Só não esqueça de me enviar seu código, junto com seu nome.

Exercícios sobre print, printf e println em Java

1. Frase na tela - Implemente um programa que escreve na tela a frase "O primeiro programa a

gente nunca esquece!".

2. Etiqueta - Elabore um programa que escreve seu nome completo na primeira linha, seu endereço na segunda, e o CEP e telefone na terceira.

3. Frases assassinas - Faça um programa que mostre na tela algumas frases assassinas, que são aquelas que fazem com muitas idéias sejam perdidas antes que amadureçam ou seja aprofundadas.

Eis alguns exemplos (bole também os seus):

"Isto não vai dar certo"

"Você nunca vai conseguir"

"Você vai se estrepar"

"Não vai dar em nada"

"Está tudo errado!"

4. Mensagem - Escreva uma mensagem para uma pessoa de que goste. Implemente um programa

que imprima essa mensagem.

5. Ao mestre - Escreva um bilhete ao seu professor, informando seus objetivos nesta disciplina e o

que espera dela e do professor. Implemente um programa que mostra seu bilhete na tela.

6. Quadrado - Escrever um programa que mostre a seguinte figura no alto da tela:

XXXXXX

X X

X X

X X

XXXXXX

7. Tabela de notas - Escreva um programa que produza a seguinte saída na tela:

ALUNO(A)	NOTA
----------	------

=====	=====
-------	-------

ALINE	9.0
-------	-----

MÁRIO	DEZ
-------	-----

SÉRGIO	4.5
--------	-----

SHIRLEY	7.0
---------	-----

8. Letra grande - Elabore um programa para produzir na tela a letra J, de Java Progressivo, usando a própria. Se fosse

‘L’, seria assim:

L

L

L

LLLLL

9. Menu - Elabore um programa que mostre o seguinte menu na tela:

Cadastro de Clientes

0 - Fim

1 - Inclui

2 - Altera

3 - Exclui

4 - Consulta

Opção:

10. Pinheiro - Implemente um programa que desenhe um "pinheiro" na tela, similar ao abaixo.

Enriqueça o desenho com outros caracteres, simulando enfeites.

```

    X
  XXX
XXXXX
XXXXXXX
XXXXXXXXX
XXXXXXXXXX
XXXXXXXXXXX
XXXXXXXXXXXX
  XX
    XX
      XXXX
```

Crie um novo tópico no fórum do Java Progressivo, e mostre seus códigos!

Segue o Pinheiro e o quadrado, bem caprichados que ele fez:

```
public class pinheiro {

public static void main(String[] args) {

System.out.print("    X\n");
```

```
System.out.print("   XXX\n");
```

```
System.out.print("   XXXXX\n");
```

```
System.out.print("   XXXXXXXX\n");
```

```
System.out.print("   XXXXXXXXXXX\n");
```

```
System.out.print("   XXXXXXXXXXXXX\n");
```

```
System.out.print("   XXXXXXXXXXXXXXXX\n");
```

```
System.out.print("   XXXXXXXXXXXXXXXXXXXX\n")
```

```
System.out.print("   XX\n");
```

```
System.out.print("   XX\n");
```

```
System.out.print("   XXXX\n");
```

```

    }
}

public class Quadrado {

    public static void main(String[] args) {

        System.out.println("XXXXXX");

        System.out.println("X    X");

        System.out.println("X    X");

        System.out.println("X    X ");

        System.out.println("XXXXXX");

    }
}

```

E finalizamos com a bela mensagem que ele fez, que tem tudo a ver com o nosso curso! Continuem estudando!

```
public class Mensagem {
```

```
public static void main(String[] args) {
```

```
System.out.println("Má, não deixe ninguém te dizer que há alguma coisa que  
não possa fazer!");
```

```
}
```

```
}
```


Comentários e Delimitadores em Java

Comente seus códigos, deixe-os bem explicados e claros para quando outros programadores forem ler.

Daqui alguns meses, quando estiver mais avançado em nosso curso de Java, você vai notar que esses 'outros' incluem você, quando não se lembrar mais o motivo de ter implementado o método daquele jeito.

Um comentário é sempre bom para explicar o que um trecho no seu código significa, assim você ou outra pessoa não perde tempo tentando decifrar.

Basta ler o comentário que explica seu funcionamento.

Como fazer comentários nos códigos Java

Bom, nessa altura do campeonato você já deve ter criado vários projetos e saídas.

Crie um projeto chamado 'Teste'.

Vamos testar uma coisa, vamos colocar essa linha aqui embaixo da 'main' e ver o que acontece:

```
//Olá, o que acontece se eu escrever isso?
```

Teste:

```
public class Teste {
```

```
    //Olá, o que acontece se eu escrever isso?
```

```
public static void main(String[] args) {
```

```
    System.out.println("Não sei! O que acontece se escrever aquilo lá?");
```

```
    }
```

```
}
```

O que fizemos foi um comentário.

Tudo o que colocarmos na mesma linha e depois das duas barras, '//', não surtirá efeito na compilação/execução. Você pode até xingar o Java que ele vai rodar do mesmo jeito.

Pra que ser isso então, se não vai influenciar em nada?

Bom, em grandes projetos, principalmente os acadêmicos e os que envolvem complexa lógica, fica complicado entender o que outro programador programou.

As vezes parecem hieróglifos.

As vezes o sujeito faz uma magia matemática e você não consegue entender o que ele fez.

As vezes você cria um algoritmo complexo e longo, passa meses sem mexer nesse código e depois quando vai olhar de volta não consegue se lembrar como criou aquilo.

Os comentários servem para isso, escreva algo do tipo, antes:

```
//o seguinte método faz isso, isso e aquilo
```

```
//essa classe é usada pra aqueles
```

```
//esse algoritmo recebe esses números, faz esses cálculos e retorna essa operação
```

Porém, evite comentar tudo. Não comente o óbvio:

```
//me sentei
```

//pausa pra ler o globo.com

//a função exibe 'Bom dia' na tela

Use comentários para coisas que não entende, como coisas bizarras, por exemplo:

//A saída simples a seguir é uma famosa música sertaneja universitária brasileira, de 2012

```
System.out.println("Tchê tchê-rê-rê tchê-tchê tchê tchê tchê");
```

PS: Caso você saiba o que isso significa, por favor, me avise.

Como usar os delimitadores /* */ em Java

Vamos supor que você vai criar um algoritmo complexo, ou copiou uma questão na IDE mesmo (pra não ter que ficar olhando pro livro, pro Java Progressivo ou pro pdf).

Supondo que esse comentário tenha várias linhas, dezenas. E aí?

Vai criar dezenas de '/' ?

```
// Questão 08
```

```
// do site: Java Progressivo
```

```
// sobre Saídas Simples
```

```
// do curso de Java
```

```
// desenhe a letra P de forma gigante com os caracteres P
```

Claro que não, isso é muito cansativo.

Pra isso, existem os delimitadores '/*' e '*/'

Tudo o que você escrever dentro dele será desconsiderado. Veja:

```
/* Cavalo de Fogo
```

```
No meu sonho eu já vivi
```

```
Um lindo conto infantil
```

Tudo era magia,
Era um mundo fora do meu
E ao chegar desse sono acordei

Foi quando correndo eu vi
Um cavalo de fogo alí
Que tocou meu coração
Quando me disse, então
Que um dia rainha eu seria
Se com a maldade pudesse acabar
No mundo dos sonhos pudesse chegar */

```
public class Teste {
```

```
public static void main(String[] args) {
```

```
System.out.println("Tenho saudades dos desenhos da minha infância");
```

```
}
```

```
}
```

Note que o NetBeans já realça em uma cor diferente a parte que será delimitada, que é um comentário.

Tipos Numéricos - int (inteiro), float e double (decimais ou reais) | Java Progressivo

Já sabemos como mostrar mensagens através dos 'print'.

Agora vamos fazer algumas operações com números.

Iremos mostrar como trabalhar com números inteiros e reais (decimais), além de fazer uma explicação sobre os tipos de dados (int, float e double, no caso).

Declarando variáveis em Java

O Java é uma linguagem fortemente tipada, ou seja, para usarmos os tipos de informações, temos que declará-los.

Vamos declarar um inteiro:

```
int idade;
```

As declarações seguem essa sintaxe: [tipo] nome_da_variável;

Isso é necessário pois o Java seleciona uma parte na memória (aloca) para esta variável, porém os tipos de variáveis ocupam diferentes tamanhos na memória.

O tipo 'int', por exemplo, armazena 32 bits, ou qualquer inteiro entre -2.147.483.648 e 2.147.483.647

O tipo 'float', que armazena números decimais (quebrados, ou com vírgula) também armazenam 32 bits.

Já os 'long' armazenam 64 bits, assim como 'double' (que é um 'float' maior), ou seja, qualquer número inteiro de -9.223.372.036.854.775.808L até 9.223.372.036.854.775.807L.

Vamos declarar um tipo 'long':

```
long idade_do_universo;
```

Podemos fazer:

long idade;

para armazenar a idade de uma pessoa? Sim, podemos, mas é óbvio um desperdício de memória, pois não usaremos um número grande para representar nossa idade, mesmo que você fosse a Dercy Gonçalves.

Inicializando uma variável

Poderíamos atribuir o valor a uma variável de duas maneiras, uma na declaração:

```
int idade=21;
```

Outro meio é depois da declaração:

```
int idade;
```

```
idade=21;
```

Vamos mostrar como imprimir o valor de uma variável na 'print', lembrando que o Java só permite isso depois que você declara e inicializa sua variável:

```
System.out.println(idade);
```

Teste o seguintes código:

```
public class Soft {
```

```
public static void main(String[] args) {
```

```
    int idade=21;
```

```
System.out.println("idade");
```

```
System.out.println(idade);
```

```
}
```

```
}
```

Notou a diferença? Quando colocado entre aspas, saí o texto. Sem aspas, sai o valor armazenado.

Vamos usar os dois, "idade" e o valor idade.

Como vimos, para sair em forma de texto, escreva "entre aspas" e use o sinal de somar '+' para adicionar mais informações, no caso, o valor da variável inteira 'idade'.

Veja o resultado do código:

```
public class Soft {
```

```
public static void main(String[] args) {
```

```
    int idade=21;
```

```
System.out.println("Minha idade é: " + idade);
```

```
}
```

```
}
```

Um texto entre aspas é chamado de string. Então, estamos printando a string "idade" e um inteiro.

Substitua e teste, agora com valores decimais:

```
float dinheiro=1.99f;
```

```
System.out.println("Só tenho R$" + dinheiro + " na minha carteira");
```

Dois detalhes importantes:

1. Usamos vírgula em países da América não-inglesa e na Europa, mas para representar valores decimais, em computação, usamos o ponto '.' como separador, e não vírgula!
2. Por padrão, o Java assume valores decimais como double. Pra especificar que é um 'float', coloque aquele 'f' ao final. Ou 'F'.

Ao final do tipo 'long' coloque 'l' ou 'L'.

Para armazenar inteiros, também existem os tipos 'byte', que armazena 8 bits e 'short', que armazena 16 bits.

Porém, vamos desconsiderar estes, devido suas limitações (muito pequenos).

Recebendo dados do usuário: a classe Scanner

Até o presente momento definimos as variáveis no momento em que estávamos programando.

Mas e se quiséssemos obter essa informação do usuário?

Por exemplo, para perguntar a idade ou para criar uma calculadora? Iríamos depender do que fosse digitado.

Usaremos a classe Scanner para receber esses dados.

Importando (import) classes (class) e pacotes (packages)

Existem milhares de funcionalidades no Java. Essas classes foram agrupadas em pacotes, os packages.

E pacotes para a mesma funcionalidade são chamados de API (Application Programming Interface). Por exemplo, temos uma seção sobre a API Java 2D, para fazer desenhos em 2D.

Ou seja, são uma série de pacotes para desenhar.

Porém, todos esses pacotes não estão simplesmente prontos para serem utilizados, pois são muitos.

Inclusive, você pode criar (e vai) os seus pacotes, pode baixar, reutilizar, compartilhar, vender etc.

Se todos estes estivessem prontos para utilização, demoraria MUITO para rodar um programa em Java.

Qual a solução então?

Vamos dizer ao Java quais funcionalidades queremos usar. Pra isso, usamos a função 'import':

```
import pacote_que_voce_quer_importar;
```

Por exemplo: para usar print, printf e println, não precisa dizer nada ao Java.

São métodos tão comuns que podem ser usadas automaticamente em qualquer aplicação.

Esses métodos fazem parte de um pacote chamado 'java.lang'.

Recebendo dados do usuário: new Scanner(System.in)

Para receber dados do usuário, temos que usar a classe Scanner, que faz parte do pacote 'java.util'.

Vamos dizer ao Java que usaremos essa classe na nossa aplicação

Para isso, adicione essa linha no começo do programa:

```
import java.util.Scanner;
```

Bom, temos a classe. Vamos declarar o nosso objeto do tipo Scanner.

Vamos chamá-lo de 'entrada'. Sua declaração é feita da seguinte maneira:

```
Scanner entrada = new Scanner(System.in);
```

Pronto, o objeto 'entrada' será usado para ler entradas do sistema.

Lembre-se que há uma forte tipagem por trás dos panos. Ou seja, o Java está lidando com bytes, blocos de memória e outras coisas mais complicadas.

Então, para ele, há muita diferença entre inteiros, float, doubles e outros tipos. Portanto, precisamos ser bem claros quanto a isso.

Assim, a nossa entrada será bem tipada. Vamos iniciar por inteiros.

Para receber um número inteiro do usuário, com nosso objeto 'entrada', usaremos

a seguinte sintaxe:

```
inteiro = entrada.nextInt();
```

Explicações dadas, vamos ver a coisa funcionando. Esse é um exemplo bem simples que pergunta a idade do usuário, espera ele digitar (e dar enter) e exibe essa mensagem na tela:

```
import java.util.Scanner;
```

```
public class Entrada {
```

```
public static void main(String[] args) {
```

```
    Scanner entrada = new Scanner(System.in);
```

```
    int idade;
```

```
    System.out.println("Digite sua idade: ");
```

```
    idade = entrada.nextInt();
```

```
    System.out.printf("Sua idade é " + idade + "\n");
```

```
    }
```

```
}
```

Mas isso de 'qual sua idade?...19...você tem 19 anos', é meio tele-tubbie.

Você é programador Java, vulgo Paladino das Artes Computacionais.

O seguinte programa usa uma função que obtém o ano atual (do computador do usuário) e calcula o ano que o usuário nasceu, que é algo mais útil que repetir o que você acabou de digitar.

Para isso, usaremos a classe 'Calendar', que tem métodos para trabalharmos com dias, horas, dias da semana, minutos, segundos, anos etc.

Para saber mais sobre, acesse a documentação:

<http://docs.oracle.com/javase/1.4.2/docs/api/java/util/Calendar.html>

Pra usar a 'Calendar', é necessário importar:

```
import java.util.Calendar;
```

Usaremos o método get(Calendar.YEAR), que retorna um inteiro com o ano, e vamos armazenar esse inteiro em uma variável 'ano'.

Então, nosso programa fica assim:

```
import java.util.Scanner;
```

```
import java.util.Calendar;
```

```
public class Entrada {
```

```
public static void main(String[] args) {  
  
    Scanner entrada = new Scanner(System.in);  
  
    int idade;  
  
    int ano_atual;  
  
    int ano_nascimento  
  
    // pergunta a idade e armazena  
  
    System.out.println("Digite sua idade: ");  
  
    idade = entrada.nextInt();  
  
    //Criando um objeto do tipo Calendar, o 'calendario' e armazenando o ano  
    atual  
  
    Calendar calendario = Calendar.getInstance();  
  
    ano_atual=calendario.get(Calendar.YEAR);  
  
    ano_nascimento= ano_atual - idade;  
  
    System.out.printf("Você nasceu em " + ano_nascimento + "\n");  
  
}
```

```
}  
  
}
```

O ano de nascimento é calculado e armazenado através da operação de subtração:

```
ano_nascimento = ano_atual - idade;
```

Você aprenderá outras operações matemáticas em breve.

Aqui vale uma ressalva.

Eu fiz assim por ser um tutorial básico e por questão de organização, mas a variável 'ano_nascimento' não seria necessária.

Poderíamos ter usado '(ano_atual - idade)' direto no printf assim:

```
System.out.printf("Você nasceu em " + (ano_atual - idade) + "\n");
```

Mas tem que ser entre parênteses.

Assim você não precisa mais da variável 'ano_nascimento'.

Aliás, também não precisaríamos da variável 'ano_atual', poderíamos ter feito diretamente assim:

```
System.out.printf("Você nasceu em " + ( calendario.get(Calendar.YEAR) -  
idade) + "\n");
```

Quer saber? Esqueça da variável 'idade', faça só:

```
import java.util.Scanner;
```

```
import java.util.Calendar;
```

```
public class Entrada {
```

```
public static void main(String[] args) {
```

```
    Scanner entrada = new Scanner(System.in);
```

```
    System.out.println("Digite sua idade: ");
```

```
    Calendar calendario = Calendar.getInstance();
```

```
    System.out.printf("Você nasceu em " + (calendario.get(Calendar.YEAR) -  
    entrada.nextInt()) + "\n");
```

```
    }
```

```
}
```

Notou como fomos tirando as variáveis e foi ficando cada vez menor?

Cada vez que tiramos, o programa fica menor e ocupa menos espaço.

Porém perde em legibilidade. Note que agora está mais difícil de entender o que fizemos.

Com 'idade', 'ano_atual' e 'ano_nascimento' fica bem mais organizado.

Não existe um método ou jeito melhor de se fazer as coisas. Programar é algo pessoal.

Você terá que encontrar um meio termo entre eficiência (programa rápido e ocupado pouco espaço), mas que seja de fácil entendimento por outros usuários (e por você no futuro). Pois se escrever de forma muito compacta e complicada, pode ser que nem você entenda o que fez futuro.

Recebendo outros tipos de dados, float: nextFloat();

No exemplo anterior, usamos o tipo inteiro.

Se você for perspicaz notará o 'Int' de 'nextInt()'.

Sim, pra float, será 'nextFloat':

```
import java.util.Scanner;
```

```
public class Entrada {
```

```
    public static void main(String[] args) {
```

```
        Scanner entrada = new Scanner(System.in);
```

```
        float preco;
```

```
        System.out.println("Quanto custa algo em uma loja de R$ 1,99? ");
```

```
        preco = entrada.nextFloat();
```

```
        System.out.println("Hã? " + preco + "?");
```

```
}  
  
}
```

Programa bobo ,não?

Não. É uma pegadinha. Digite 1,99 pra você ver.

O certo é 1.99 ;)

Exercício:

Crie um aplicativo que pergunte o ano de nascimento e diga a idade atual do indivíduo.

Use a classe Calendar e o método get(Calendar.YEAR) desta classe.

PS: não chame o usuário de indivíduo

PS2: Poderá existir alguns problemas no cálculo de sua idade

1. Por conta do mês que você faz aniversário.

Isso será resolvido deixando o problema mais completo, perguntando o mês e dia do aniversário e usando testes condicionais, que aprenderemos mais adiante.

2. Se o usuário não digitar um número, ou digitar negativo ou outro absurdo.

Nesse momento você sentirá vontade de chamar ele de indivíduo, energúmeno e outras coisas.

Mas se acostume.

Existe uma coisa chamada tratamento de erros e exceções, muito usada justamente pra tentar imaginar os possíveis erros e exceções que podem ocorrer.

No caso, exceções, é o nome bonito que se dá aos absurdos que os usuários podem digitar.

Por exemplo, você já deve ter visto em cadastros:

Insira sua data de nascimento na forma dd/mm/aaaa:

Aí o usuário vai e digita 5/janeiro/89 e não sabe qual foi o erro.

Experimente agora digitar algo que não seja um número em seus programas.

Para saber mais sobre a relação de de usuários x programadores visite:

<http://www.vidadesuporte.com.br>

<http://www.vidadeprogramador.com.br>

VIDA DE PROGRAMADOR

.COM.BR

/* HISTÓRIA REAL
ENVIADA POR
RAFAEL ROSSIGNOL */



#595

CARA, EU FICO INDIGNADO COM
ESSES RELATOS DE ERROS ONDE
ESCREVEM SÓ: "TEM UM ERRO"...

FAZ ASSIM: ESCREVA UM
E-MAIL PARA O USUÁRIO
PARA QUE ELE SÓ POSSA
TE RESPONDER A OPÇÕES
DE FORMA NÃO-AMBIGUA...



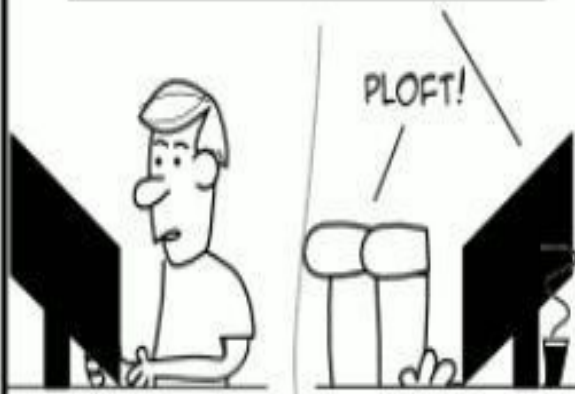
"POR FAVOR, ME DIGA O QUE
ACONTECEU PARA GERAR O ERRO:
A) VOCÊ ENTROU NA TELA E O
PEDIDO SUMIU SEM VOCÊ
FAZER NADA; OU
B) VOCÊ MANDOU FATURAR O
PEDIDO E DEU ALGUMA
MENSAGEM DE ERRO"



DEPOIS...

Re: Re: Erro no sistema

"Isso mesmo"



→ ISTO É UM E-MAIL

Operações Matemáticas: Adição, Subtração, Multiplicação, Divisão, Resto da Divisão (módulo) e precedência dos operadores

Quanto é $1 + 2 \times 2$?

Para nós, pode ser 5 ou 6. Pro Java é sempre 5.

Se fizer a soma primeiro, é 6: $1+2 \times 2 = 3 \times 2 = 6$

Se fizer o produto primeiro, é 5: $1 + 2 \times 2 = 1 + 4 = 5$

E o computador? Como faz? E se ele se confundir também?

E 5 dividido por 2 ?

2.5 ? Pro Java pode ser 2.5 ou 2.

Operações Matemáticas em Computação

Primeiro, esqueça o 'x' como sinal de multiplicação, de agora em diante é '*'.

Segundo, divisão se representa com o '/'. Por exemplo, $4/2 = 2$

Terceiro, tudo em computação é matemática.

Aliás, computação vem de computar, ou seja, contar.

Até quando você escreve em Java, a JVM(Java Virtual Machine) troca informações com sua máquina na forma de matemática (em bits).

Até as imagens, são divididas em pequenos blocos, são numerados e cada bloquinho é pintado com uma cor devidamente identificada.

Adivinhe como é essa identificação? Claro, por meio de números.

Ou achou que a máquina ia dividir em 'rosa emo', 'azul bebê' ou 'azul piscina'?

Adição, Subtração e Multiplicação

Essas três não tem muitas dificuldade. São 'zona'. Você pode somar, subtrair e multiplicar o que quiser.

Á rigor, tudo é uma soma.

Subtração é uma soma com números negativos: $a - b = a + (-b)$

Multiplicação é uma soma repetida várias vezes: $a * b = a + a + a + a \dots + a$ (o 'a' se repete 'b' vezes).

O seguinte código pede dois números inteiros (se não for inteiro, terá erros) e mostra o resultado da soma, subtração e multiplicação, sem segredo:

```
import java.util.Scanner;
```

```
public class Operacoes {
```

```
public static void main(String[] args) {
```

```
    Scanner entrada = new Scanner(System.in);
```

```
    int num1;
```

```
int num2;
```

```
System.out.print("Digite o primeiro número: ");
```

```
num1 = entrada.nextInt();
```

```
System.out.print("Digite o segundo número: ");
```

```
num2 = entrada.nextInt();
```

```
System.out.println();
```

```
System.out.println(num1 + " + " + num2 + " = " + (num1 + num2) );
```

```
System.out.println(num1 + " - " + num2 + " = " + (num1 - num2) );
```

```
System.out.println(num1 + " * " + num2 + " = " + (num1 * num2) );
```

```
}
```

```
}
```

Caso não for usar inteiros (ou for) aconselho a usar 'printf', pois você pode formar a saída.

Formatação com printf

Uma forma diferente de exibir números é:

```
System.out.printf("número: %d", num1);
```

O exemplo passado ficaria:

```
import java.util.Scanner;
```

```
public class Operacoes {
```

```
public static void main(String[] args) {
```

```
    Scanner entrada = new Scanner(System.in);
```

```
    int num1;
```

```
    int num2;
```

```
System.out.print("Digite o primeiro número: ");
```

```
    num1 = entrada.nextInt();
```

```
System.out.print("Digite o segundo número: ");
```

```
num2 = entrada.nextInt();
```

```
System.out.printf("\n%d + %d = %d\n", num1, num2, num1 + num2);
```

```
System.out.printf("%d - %d = %d\n", num1, num2, num1 - num2);
```

```
System.out.printf("%d * %d = %d\n", num1, num2, num1 * num2);
```

```
}
```

```
}
```

Ou seja, o '%d' será substituído pelo valor de %d.

Para substituir valores de variáveis float, use %f

Com multiplicação e divisão de números decimais, a coisa pode ficar bagunçada por conta das vírgulas. Por isso, vamos adicionar esse '.2' entre o '%' e o 'f', ficando %.2f.

Isso quer dizer que, após o ponto decimal, imprimir somente duas casas decimais. Veja:

```
import java.util.Scanner;
```

```
public class Operacoes {
```

```
public static void main(String[] args) {
```

```
    Scanner entrada = new Scanner(System.in);
```

```
    float num1;
```

```
    float num2;
```

```
    System.out.print("Digite o primeiro número: ");
```

```
    num1 = entrada.nextFloat();
```

```
    System.out.print("Digite o segundo número: ");
```

```
    num2 = entrada.nextFloat();
```

```
    System.out.printf("\n%.2f + %.2f = %.2f\n", num1, num2, num1 + num2);
```

```
    System.out.printf("%.2f - %.2f = %.2f\n", num1, num2, num1 - num2);
```

```
    System.out.printf("%.2f * %.2f = %.2f\n", num1, num2, num1 * num2);
```

```
    System.out.printf("%.2f / %.2f = %.2f\n", num1, num2, num1 / num2);
```

}

}

Experimente tirar o '.2' e veja a bagunça que fica na hora de multiplicar e dividir.

Divisão

Não divida nada por zero. Isso não existe.

Não é infinito (ou -infinito). É uma indefinição matemática.

Outra detalhe que você deve atentar é a divisão de inteiros. Quanto é $5/2$?

Vamos ver o que o Java nos dizer com o seguinte programa:

```
import java.util.Scanner;
```

```
public class Operacoes {
```

```
public static void main(String[] args) {
```

```
    Scanner entrada = new Scanner(System.in);
```

```
    int num1=5;
```

```
    int num2=2;
```

```
    System.out.println(num1/num2);
```

```
}
```

```
}
```

run: 2 BUILD SUCCESSFUL (total time: 0 seconds)

2 ?

"Putz, como meu computador é burro!"

Calma, agora teste:

```
import java.util.Scanner;
```

```
public class Operacoes {
```

```
public static void main(String[] args) {
```

```
    Scanner entrada = new Scanner(System.in);
```

```
    float num1=5;
```

```
    float num2=2;
```

```
System.out.println(num1/num2);
```

```
}
```

```
}
```

run: 2.5 BUILD SUCCESSFUL (total time: 0 seconds)

Ou seja, divisões de inteiros retornam resultados inteiros. Mesmo se o resultado 'de verdade' fosse 2.999999, o Java retornaria somente a parte inteira, que é 2, e não 3.

Em termos técnicos, o Java trunca a parte decimal. Simplesmente descarta ela.

Resto da divisão

Voltando ao $5 / 2$: caso a operação seja feita com inteiro, você sabe que o resultado é 2. Porém, terá um resto, que será 1.

Pois: $5 = 2 * 2 + 1$

O símbolo de resto da divisão é %.

Assim, $8 \% 3 = 2$. Pois: $8 = 3*2 + 2$

Em matemática chamamos de 'mod', de módulo: $8 \bmod 3 = 2$

No Brasil, é aquele 'resto' que deixávamos lá embaixo nas continhas do colégio:

$$\begin{array}{r} 2009 \overline{) 19} \\ 109 \\ \hline 95 \end{array}$$

14

Resto da divisão

Teste:

```
public class Operacoes {
```

```
public static void main(String[] args) {
```

```
System.out.println(2009%19);
```

```
}
```

```
}
```

Veremos e usaremos bastante o resto da divisão em várias questões/algoritmos.

Por exemplo, todo número par deixa resto da divisão por 2 igual à 0. Isso é uma informação valiosíssima.

Precedência dos operadores e uso dos parênteses

Para não existir confusão e mais de um resultado possível, o Java adota uma precedência de seus operadores:

1o: Multiplicação (*), Divisão (/) e resto da divisão (%) são calculados primeiro. Caso existam mais de um na mesma expressão, o Java impõe a precedência como sendo da esquerda pra direita na expressão.

2o: Adição (+) e Subtração(-) depois dos operadores acima serem calculados, esses são calculados. Caso existam mais de um na mesma expressão, o Java impõe a precedência como sendo da esquerda pra direita na expressão.

Para não existir confusão e para deixar seu código sempre mais claro, separe as expressões com parênteses. Por exemplo, como se calcula média dos números a e b?

$a + b / 2$ -> errado

$(a + b) / 2$ -> certo

Os parentêses formam um bloco. É como se o que estivesse ali dentro fosse calculado separadamente.

No exemplo de inicio deste capítulo: $1 + 2 * 2$

Dependendo do que você quer, divida em em parantêses que ficará implícito.

Veja:

$1 + 2 * 2 = 5$ (precedência do $*$ perante ao $+$, em Java)

$(1 + 2) * 2 = 6$

$1 + (2 * 2) = 5$

É fácil ver que fica mais claro e organizado com o uso dos parênteses.

Exercícios:

- 1. Crie um programa que recebe suas três notas (colégio, faculdade) e calcule a média final.**
- 2. Crie um programa que receba a altura e o peso do usuário, e diga seu IMC (Índice de Massa Corporal), dado pela fórmula:**

INC



peso (em quilos)



altura² (em metros)

Fazendo comparações: os operadores maior (>), menor (<), maior igual (>=), menor igual (<=), igual (==) e diferente (!=)

O que te faz escolher um canal ou outro da televisão?

O que te faz escolher entre uma loira ou morena? Ou homem e mulher? Ou os dois, caso seja emo?

Por que estudar Java, e não Haskell?

Seja lá quais forem os motivos, mas todos passam por um estágio: a comparação.

Como fazer comparações em Java

Em Java, quando fazemos uma comparação ele retorna "true", 'false' ou dá erro. E se ele errou, meu amigo, é porque você errou.

Esses operadores, junto com as condicionais (que veremos em breve) e as operações matemáticas já vistas, são o alfabeto de Java.

São o básico, mas são os assuntos mais essenciais.

Esse artigo é bem curto e de fácil entendimento.

Talvez você já saiba o significado de todos esses símbolos, então já adianto um programa que faz e mostra todos os testes:

```
import java.util.Scanner;
```

```
public class Comparando {
```

```
public static void main(String[] args) {
```

```
    float num1, num2;
```

```
    Scanner entrada = new Scanner(System.in);
```

```
System.out.print("Digite o primeiro número: ");
```

```
    num1 = entrada.nextFloat();
```

```
System.out.print("Digite o segundo número: ");
```

```
    num2 = entrada.nextFloat();
```

```
System.out.printf("%.2f > %.2f -> %s\n", num1, num2, num1 > num2);
```

```
System.out.printf("%.2f >= %.2f -> %s\n", num1, num2, num1 >= num2);
```

```
System.out.printf("%.2f < %.2f -> %s\n", num1, num2, num1 < num2);
```

```
System.out.printf("%.2f <= %.2f -> %s\n", num1, num2, num1 <= num2);
```

```
System.out.printf("%.2f == %.2f -> %s\n", num1, num2, num1 == num2);
```

```
System.out.printf("%.2f != %.2f -> %s\n", num1, num2, num1 != num2);
```

```
    }
```

```
}
```


Maior que: >

$a > b$ -> retorna 'true' caso 'a' seja maior que 'b', e 'false' caso seja menor

Menor que: <

$a < b$ -> retorna 'true' caso 'a' seja menor que 'b', e 'false' caso seja maior

Maior ou igual a: >=

$a \geq b$ -> retorna 'true' caso 'a' seja maior ou igual à 'b', e 'false' caso seja menor

Menor ou igual a: <=

$a \leq b$ -> retorna 'true' caso 'a' seja menor ou igual à 'b', e 'false' caso seja maior

Comparação de igualdade: ==

$a == b$ -> retorna 'true' caso 'a' seja igual a b, e 'false' caso contrário

Comparação de negação: !=

$a != b$ -> retorna 'true' caso 'a' seja diferente de b, e 'false' caso contrário

Importante: Note a diferença entre

$a == b$

e

`a = b`

O primeiro é um TESTE de comparação! É como se fosse uma pergunta pro Java: "'a' é igual a 'b'?"

O segundo, é uma afirmação, ou seja, 'a' vai receber o valor de 'b' ! Neste caso o Java retorna sempre o último valor, que no caso é 'b'. Não é comparação!

Caso fosse `a=b=c`, ele retornaria 'c'.

O tipo char: armazenando e representando caracteres

O tipo de dado char serve para armazenar um caractere (o char vem de character, que é caractere em inglês), e por essa função simples tem o tamanho do tipo short.

Naturalmente vem a pergunta, para que armazenar só um caractere?

Se há tipos para armazenar números, strings e textos, o que vamos fazer com um caractere?

Tipo char: O que é? Onde é usado ?

"Pressione S para sim ou N para não"

"Pressione alguma tecla para para continuar..."

Ou em jogos...nunca notou que no counter-strike que para ir para o lado esquerdo basta pressionar uma das setas (ou 'a' ou 'd').

E para o Mário Bros pular ou soltar o fogo?

Muitas vezes não precisamos de um arquivo de texto inteiro ou string, mas de simplesmente um caractere.

Declaração do tipo char

```
char nome_do_char = 'a';
```

onde poderíamos substituir 'a' por qualquer caractere alfanumérico, caso tenhamos declarado assim.

Sim, podemos declarar de outra maneira. Os caracteres podem ser representados por inteiro também.

```
char letra_J = 74;
```

```
char letra_P = 80;
```

E dentro do printf, representamos o char por %c, para efeitos de formatação.

Vamos imprimir as iniciais do site, veja:

```
public class Tipos {
```

```
public static void main(String[] args){
```

```
    char letra_J = 74;
```

```
    char letra_P = 80;
```

```
System.out.printf("%c %c\n",letra_J, letra_P);
```

```
}
```

```
}
```

run: J P BUILD SUCCESSFUL (total time: 0 seconds)

Mas como eu adivinhei que 'J' é 74 e 'P' é 80? Qual a Matemática ou magia pra saber isso?

Tabela ASCII, que são uma forma universal de representação do caracteres por meio de números, seja decimal, hexadecimal ou octal.

Esses números são a representação decimal dos caracteres, confira a tabela abaixo:

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040	 	Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	\$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	((72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051))	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EH (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[[123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174	|	
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135]]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177		DEL

Source: www.LookupTables.com

Na verdade, essa representação é humana. Para diversos tipos de numeração.

Existe extensão dessa tabela para os chineses, para os indianos e outros povos com diferentes caracteres. E pros computadores?

Computador é macho, pra ele é tudo número, é tudo bit!

Representação é coisa humana.

Misturei tudo, veja só:

```
public class Tipos {
```

```
public static void main(String[] args){
```

```
    int numero_J = 74;
```

```
    char letra_J = (char) numero_J;
```

```
    char letra_P = 80;
```

```
        char letra_i = 'i';
```

```
System.out.printf("%c%c%c%c %c%c%c%c%c%c%c%c%c%c%c\n",
```

```
        letra_J,97,118,97, letra_P, 'r',
```

```
        111,103,114,101,'s','s',letra_i,
```

```
        118, 111);
```

```
}
```


}

Não poderíamos simplesmente igualar um char a um inteiro:

letra_J = numero_J

Então fizemos um cast: `char letra_J = (char) numero_J;`

Que nada mais é que dizer ao Java para representar o próximo tipo de uma outra forma.

No caso, estamos dizendo 'Hey, Java, brother, não veja o `numero_J` como inteiro, veja como char. Eu sei que declarei como char, mas é só dessa vez, ok?'.

Para ver mais caracteres e magias numéricas:

<http://www.asciitable.com/>

<http://www.tamasoft.co.jp/en/general-info/unicode.html>

<http://pt.wikipedia.org/wiki/ASCII>

Como armazenar um caractere (char) que o usuário digitou

Usamos, obviamente, a classe Scanner para receber um caractere do usuário.

Porém, há uma pequena diferença. Usaremos o método 'nextLine()' que recebe uma string do usuário, porém como queremos um caractere, usaremos o 'charAt(0)' para receber apenas um caractere (ou o primeiro elemento da string).

Veja como se faz:

```
Scanner entrada = new Scanner(System.in);
```

```
char caractere;
```

```
caractere = entrada.nextLine().charAt(0);
```

O tipo boolean: a base da lógica

Booleano é um tipo de dado que permite apenas dois valores, true (verdadeiro) ou false (false).

Java, programação, computação, dois valores...binário?

Sim, tem a ver com algo mais complexo, a Álgebra Booleana (números e lógica binária), que é a base de toda a Engenharia Elétrica, Eletrônica e de Telecomunicações, por exemplo.

Booleanos em Java

Mas não será necessário irmos tão à fundo em nossos estudos para desenvolver em Java.

Como já dito antes, pro computador é tudo número, tudo bit mesmo.

Isso de verdadeiro ou falso é para os humanos e programadores Java criarem código com mais facilidade (programador Java não é humano, é Paladino das Artes Computacionais, é um estágio acima).

E onde usamos 'true' e 'false'? Em todo canto.

"Li e aceito os termos de condições'.

Se não selecionou, esta opção está como 'false' e você não consegue instalar o programa pirata em seu Windows (alguém já leu algum termo de condição na vida?).

Quando você marca, fica 'true' e é possível prosseguir.

Se um formulário pede uma data na forma: dd/mm/aaaa

Enquanto não é preenchido corretamente, o formulário está como 'false' para continuar.

Só fica 'true' se preencher corretamente. Por isso, se colocar: 5/jan/90

não vai de jeito nenhum, não vai ficar 'true'.

Ou seja, usamos os tipos booleanos em Java quando quisermos ter a certeza que alguma condição é verdadeira ou falsa.

Em uma linguagem mais matemática, os tipos booleanos estão relacionados com os operadores lógicos AND (E) e OR (OU), que veremos em seguida.

Declaração de tipos boolean

```
boolean nome_bool = true;
```

```
boolean nome_bool2 = false;
```

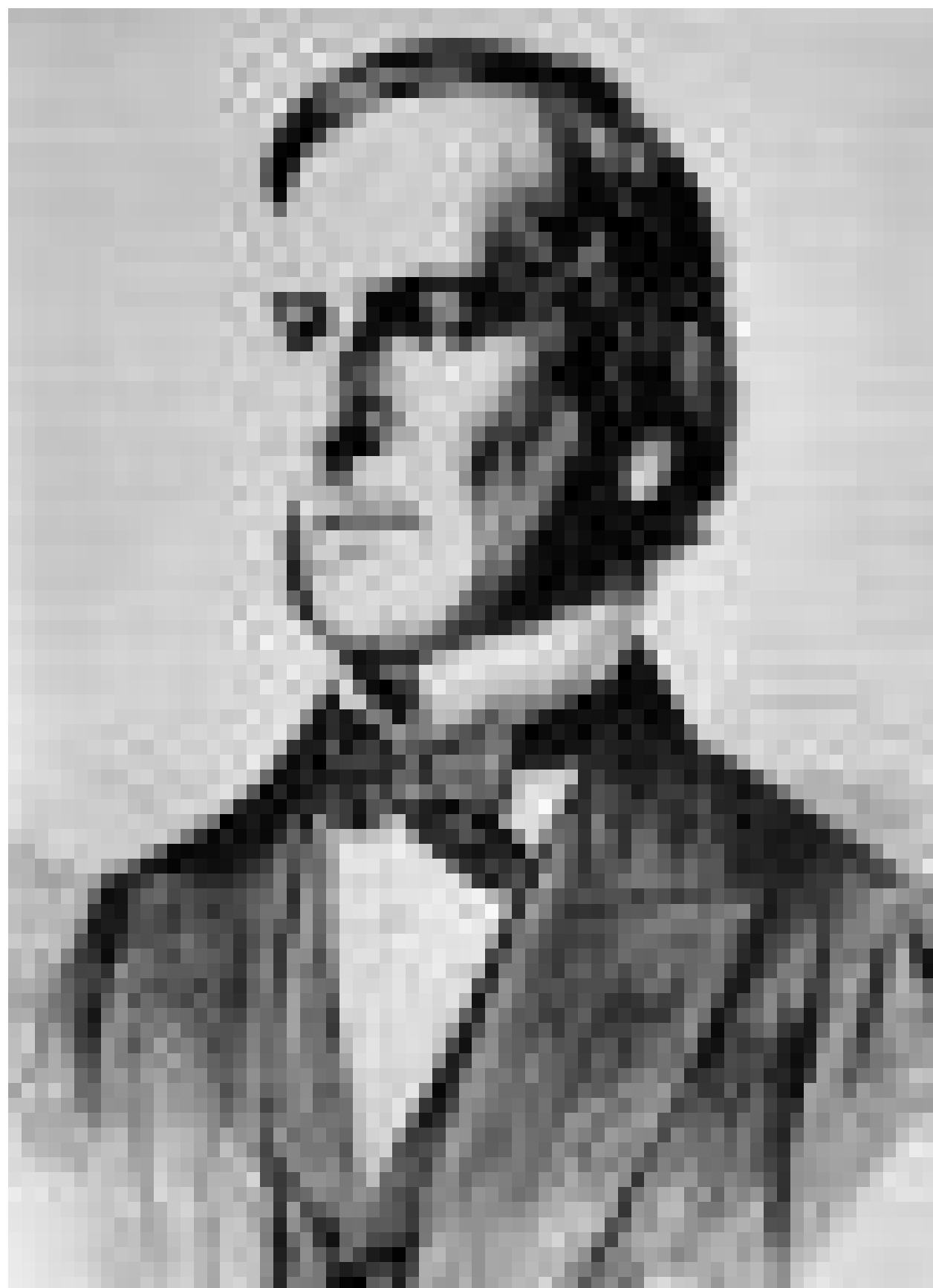
Embora possamos fazer uma analogia com o sistema binário, diferente de outras linguagens, não podemos fazer a conversão com inteiros para true = 1 e false = 0.

Ou seja, boolean é boolean e inteiro é inteiro em Java, nem com cast vai.

Em outras, linguagens, por exemplo, 'false' pode ser representado por 0, string vazia, ponteiro apontando pra 'null' ou algo vazio e 'true' por qualquer outro valor.

Em SQL (linguagem para se trabalhar com banco de dados), o booleano pode ser true, false, unknown (desconhecido) ou null (nulo).

Por hora, não temos muito o que mostrar. Usaremos MUITO os booleanos em condições de controle e loopings, mais a frente e durante todo o nosso curso.



George Boole, seus trabalho são a base de todo o sistema digital do mundo

<http://pt.wikipedia.org/wiki/Booleano>

**Operadores lógicos e de negação: && (E ou AND), || (OU ou OR) e o !
(negação)**

Completando seus conhecimentos sobre álgebra booleana, explicaremos um pouco sobre E e OU.

Mas não se assuste com a palavra 'álgebra', se você sabe português e tem bom senso, já sabe o que será explicado.

Operadores lógicos E (&&) e OU (||) em Java

Muitas vezes queremos testar se algumas condições são verdadeiras.

Geralmente queremos testar mais de umas condições.

Por exemplo: "para você passar de ano você precisa ter média maior que 6.0 em todas as matérias E média maior que 7 em pelo menos 3."

Notou o E? Ele quer dizer que as duas condições tem que ser satisfeitas.

Agora o mesmo exemplo, usando o OU.

"Para você passar de ano você precisa ter média maior que 6.0 em todas as matérias OU média maior que 7 em pelo menos 3."

Agora ficou mais fácil, pois você só precisa satisfazer uma das condições. Você pode tirar 7 em Matemática, Física e Química e zerar todas as outras, pois não tem mais que tirar média 6.0 em todas.

Mas nada te impede de satisfazer as duas condições.

Formalizando

'E' e 'OU' são nossas analogias para linguagem humana.

O computador fala na linguagem dos bits, e tem sua linguagem especial.

Em Java, para representar o 'E' e o Java se comunicar com sua máquina, você vai usar a representação: &&

Para representar o 'OU', use ||

Então, vamos formalizar os exemplos. Sejam as condições :

condicao_A = todas médias maiores que 6.0

condicao_B = tirar média maior que 7 em pelo menos 3 matérias

O primeiro exemplo é representado por: (condicao_A && condicao_B)

O segundo exemplo é representado por: (condicao_A || condicao_B)

Negando declarações: !

Para negar uma declaração ou condição, simplesmente colocamos um símbolo de exclamação (!) antes de tal declaração/condição.

Por exemplo, um bom aluno tem suas notas representadas por: `condicao_A`

Já um péssimo aluno tem suas notas representadas assim: `!condicao_A`

`!condicao_A` quer dizer que nem todas as médias são maiores que 6.0

Qual o contrário de verdade? Falso, obviamente.

Em Java, as seguintes formas são iguais:

`true = !false`

`false = !true`

Testes e Laços

Agora que você já estudou e exercitou seus conhecimentos básicos em Java sobre tipos, variáveis, classe Scanner, print, operadores e outros conceitos essenciais para o aprendizado desta linguagem de programação, vamos seguir adiante em nossos estudos.

Nesta seção iremos aprender sobre os testes condicionais (if else e switch case) bem como os loopings ou laços (for, while e do while), que fazem parte de toda linguagem de programação.

Até o momento, nossos programas em Java simplesmente rodavam do começo ao fim, automaticamente e sempre da mesma maneira.

Ao final destes tutoriais de nossa Apostila de Java, você irá aprender a criar programas que terão um fluxo diferente, baseado nas opções e escolhas feitas por um usuário.

Ou seja, se o usuário fornecer um determinado valor, acontece algo. Se ele fornecer outro valor diferente, uma outra coisa poderá ser acionada.

Por exemplo, ao executar uma calculadora, você pode optar por fazer soma, e ela fará.

Caso opte por fazer uma multiplicação, o aplicativo irá fornecer outros resultados.

E caso queira uma divisão, seu programa vai rodar de uma maneira diferente, de

modo a operar divisões, tudo de acordo com o que o usuário escolher.

Através dos laços de repetição, ou loopings, iremos ver como executar qualquer tarefa quantas vezes quisermos, de maneira automática e totalmente controlada por você programador, ou pelo usuário.

Com os conceitos destes tutoriais, nossos programas ficarão bem mais funcionais, robustos e interessantes, não só de programar, mas como também de usar.

Como usar IF e ELSE: Testando e Escolhendo o que executar

Até agora na apostia Java Progressivo, nossos códigos foram procedurais. Ou seja, seguiram um ordem de execução: do começo ao fim, executando tudo.

Mas executamos tudo, sempre que usamos um programa?

Claro que não, escolhemos. Fazemos opções.

É isso o que o if else faz, nos permite optar por executar determinadas coisas.

Como usar o IF em Java

Em inglês quer dizer 'se'.

A sintaxe é:

```
if ( condição ){  
    caso a condição seja verdadeira  
    esse bloco de código será executado  
}
```

E se não for verdade?

Não executa, ué.

Vamos testar a condição ' 1 é igual a 2?', que obviamente retornará 'false' e depois testar se 1 é igual 1.

Mas isso é em nossa língua. Como perguntamos isso pro computador?

Assim: `if(1 == 2)`

e depois: `if(1 == 1)`

Veja bem, é `1 == 2`, e não `1 = 2`.

1 == 2 é uma comparação, é uma pergunta, ela retorna um valor lógico!

Aqui que está a chave, só podemos colocar no if o que retorna valor lógico.

1 = 2 é uma atribuição de valores simplesmente!

Vamos aos códigos:

```
public class Ifelse {
```

```
public static void main(String[] args) {
```

```
if (1 == 2){
```

```
    System.out.println("Você nunca lerá essa mensagem, mwahuahauha");
```

```
    }
```

```
if (1 == 1){
```

```
    System.out.println("1 é igual a 1? Jura? ");
```

```
    }
```

```
}
```

```
}
```

Como usar ELSE em Java

Do inglês: senão

O else só vem acompanhado do if, e ele só ocorre quando a condição do if é falsa.

A sintaxe é a seguinte:

```
if ( condição ){  
    caso a condição seja verdadeira  
    esse bloco de código será executado  
} else {  
    caso a condição seja falsa  
    esse bloco de código que será executado  
}
```

O else não recebe condição. Ele executa quando o if não executa.

Ou seja:

```
if (verdade)
```

faz isso

else

faz isso

Aprender if else não é aprender pro Java, é aprender para todas as outras linguagens.

Aliás, é aprender lógica. É matemática, é raciocínio.

Essa mesma base lógica é extremamente usada em Engenharia (Elétrica, Eletrônica, Telecomunicações), em Física, Matemática Discreta, todo tipo de Ciência, em chips, no Kernel dos Sistemas Operacionais, nos aviões e em tudo que envolver lógica e mundo digital.

Mas vamos fazer algo útil:

Problema: Crie um programa que recebe uma nota (pela classe Scanner) e checa se você passou direto, ficou de recuperação ou foi reprovado na matéria, e exiba tal mensagem:

A regra é a seguinte:

Nota 7 ou mais: passou direto

Entre 5 e 7: tem direito de fazer uma prova de recuperação

Abaixo de 5: reprovado direto.

É importante que tente. Mesmo que não consiga ou consiga só uma parte do problema.

É assim que se evolui. Tentando..já passei minutos em alguns problemas, horas, dias em outros...alguns eu nem sabia por onde começar e corri pra ver a solução. Outros eu acordo de madrugada tendo uma ideia de como resolver.

No próximo tutorial, vamos ver como resolver esse problema.

Questões de IF e ELSE

0. Escreva um programa que receba o raio de uma circunferência e mostre o diâmetro, comprimento e área desta.

1. Faça um programa que receba três inteiros e diga qual deles é o maior e qual o menor. Consegue criar mais de uma solução?

2. Escreva um programa em Java que recebe um inteiro e diga se é par ou ímpar

Use o operador matemático % (resto da divisão ou módulo) e o teste condicional if.

3. Escreva um programa que pede os coeficientes de uma equação do segundo grau e exibe as raízes da equação, sejam elas reais ou complexas.

Desafio 1: Crie um programa que recebe uma nota (pela classe Scanner) e checa se você passou direto, ficou de recuperação ou foi reprovado na matéria.

A regra é a seguinte:

Nota 7 ou mais: passou direto

Entre 5 e 7: tem direito de fazer uma prova de recuperação

Abaixo de 5: reprovado direto

Desafio 2: Escreva um aplicativo Java que gere um número aleatório inteiro entre 1 e 10, e através de testes condicionais você tem que adivinhar que número é esse.

Qual a melhor técnica, a que adivinha em menos chances possíveis?

Para gerar um número aleatório na variável inteiro 'num_aleatorio', adicione no seu programa:

```
import java.util.Random;
```

Para gerar números aleatórios, crie um tipo Random 'randomGenerator':

```
Random randomGenerator = new Random();
```

E declare a variável para receber o número aleatório assim:

```
num_aleatorio = randomGenerator.nextInt(10) + 1;
```

Questão 03:

Crie um programa, em Java, que receba os coeficientes de uma equação do 2o grau e retorne suas raízes.

Um equação do segundo grau é uma equação onde:

$ax^2 + bx + c = 0$, com 'a' diferente de 0

Programa em Java que calcula as raízes e resolve uma equação do segundo grau

Passo 1:

A primeira parte do programa recebe os três coeficientes da equação, que são 'a', 'b' e 'c' e serão representados pelo tipo float.

Passo 2:

Uma equação do 2o grau só é válida se 'a' for diferente de 0, então, se for igual a 0 o programa deverá terminar.

Ou seja, nosso programa irá acontecer dentro do 'if' que checa que 'a' é diferente de 0.

Passo 3:

Determinando o valor de delta: $\text{delta} = b^2 - 4ac$

Determinando a raiz quadrada de delta: `sqrtdelta = (float)Math.sqrt(delta);`

Onde sqrt significa 'square root', ou raiz quadrada, em inglês.

Em Java, calculamos a raiz quadrada do número x com o método: `Math.sqrt(x);`

Esse método retorna um tipo 'double'. Como usamos float em nossos coeficientes, usamos o cast '(float)' para transformar o double em float.

Passo 4:

Se delta for maior ou igual a zero, as raízes são dadas por:

$$\text{raiz1} = (-b + \text{sqrtdelta}) / 2a$$

$$\text{raiz2} = (-b - \text{sqrtdelta}) / 2a$$

Passo 5:

Se delta for menor que zero, suas raízes serão complexas e as raízes serão da forma:

$$\text{raiz1} = (-b + i.\text{sqrt}(-\text{delta}))/2a$$

$$\text{raiz2} = (-b - i.\text{sqrt}(-\text{delta}))/2a$$

Formatei a saída da seguinte forma, para ficar mais legível, que é a mesma coisa das equações anteriores:

$$\text{raiz1} = (-b)/2a + i.\text{sqrt}(-\text{delta})/2a$$

$$\text{raiz2} = (-b)/2a - i.\text{sqrt}(-\text{delta})/2a$$

Código Java:

```
import java.util.Scanner;
```

```
public class Bhaskara {
```

```
public static void main(String[] args) {
```

```
    float a, b, c,    //coeficientes
```

```
        delta,    //delta
```

```
    sqrtdelta, //raiz quadrada de delta
```

```
    raiz1,raiz2; //raízes
```

```
Scanner entrada = new Scanner(System.in);
```

```
//Passo 1: Recebendo os coeficientes
```

```
System.out.println("Equação do 2o grau:  $ax^2 + bx + cx = 0$ ");
```

```
System.out.print("Entre com o valor de a: ");
```

```
    a = entrada.nextFloat();
```

```
System.out.print("Entre com o valor de b: ");
```

```
    b = entrada.nextFloat();
```

```
System.out.print("Entre com o valor de c: ");
```

```
    c = entrada.nextFloat();
```

```
//Passo 2: Checando se a equação é válida
```

```
if(a != 0){
```

//Passo 3: recebendo o valor de delta e calculando sua raiz quadrada

delta = (b*b) - (4*a*c);

sqrtdelta = (float)Math.sqrt(delta);

//Passo 4: se a raiz de delta for maior que 0, as raízes são reais

if(delta >=0){

raiz1 = ((-1)*b + sqrtdelta)/(2*a);

raiz2 = ((-1)*b - sqrtdelta)/(2*a);

System.out.printf("Raízes: %.2f e %.2f", raiz1, raiz2);

}

//Passo 5: se delta for menor que 0, as raízes serão complexas

else{

delta = -delta;

sqrtdelta = (float)Math.sqrt(delta);

System.out.printf("Raíz 1: %.2f + i.%.2f\n", (-b)/(2*a), (sqrtdelta)/(2*a));

System.out.printf("Raíz 2: %.2f - i.%.2f\n", (-b)/(2*a), (sqrtdelta)/(2*a));

```
    }  
  
    }  
  
else{  
  
    System.out.println("Coeficiente 'a' inválido. Não é uma equação do 2o grau");  
  
    }  
  
}  
  
}
```

Desafio 01:

Problema: Crie um programa que recebe uma nota (pela classe Scanner) e checa se você passou direto, ficou de recuperação ou foi reprovado na matéria.

A regra é a seguinte:

Nota 7 ou mais: passou direto

Entre 5 e 7: tem direito de fazer uma prova de recuperação

Abaixo de 5: reprovado direto

Aplicativo em Java

Vamos fazer isso usando condições, testando com IF ELSE em Java o que foi digitado pelo usuário. No caso, a nota.

Vamos colocar if dentro de if, essa técnica se chama aninhar (nested).

Vou dar uma importante dica que você vai usar por toda sua vida de programador, que é uma das

maiores lições que sei e posso passar:

"Um problema difícil nada mais é que uma série de pequenos problemas fáceis."

E qual pra transformar um problema difícil em fáceis?

Usando a técnica de Jack, o estripador: por partes. Quebre o programa.

Todo e qualquer projeto, desde esse que vou passar até os da NASA, são divididos em área, partes...na criação de um jogo, tem gente especializada até no desenho das árvores, no barulho das armas e se o eco vai ser realista.

Ok, vamos lá.

Criar um programa que recebe uma nota do usuário você já sabe, é só usar a

classe Scanner.

Mas o nosso curso vai ser focado nas boas práticas, vamos focar em criar aplicações robustas.

O que é isso?

Vamos minimizar o máximo possível os possíveis problemas que possam aparecer quando usuário for usar o problema.

Parte 1: Tendo a certeza que o usuário vai digitar uma nota válida

Você pensou nisso? E se o usuário digitar 11 ou -1.1 ?

Não pode.

O primeiro passo da nossa aplicação é ter a certeza que ele vai digitar uma nota ENTRE 0.0 e 10.0!

Se for 0.0 até 10.0, ok, e se não for? Acaba o programa.

Fica assim:

```
public class PasseiOuNao {  
  
public static void main(String[] args) {  
    float nota; //vai armazenar a nota  
  
    Scanner entrada = new Scanner(System.in);  
  
    System.out.print("Digite sua nota [0.0 - 10.0]: " );  
  
    nota = entrada.nextFloat();  
  
    if( (nota <= 10.0) && (nota >= 0.0) ){
```

```
System.out.println("Nota válida");
```

```
    } else {
```

```
System.out.println("Nota inválida, fechando aplicativo");
```

```
    }
```

```
}
```

```
}
```

O que fizemos foi certificar que a nota é menor que 10.0 E maior 0.0 !

Os dois casos precisam ser satisfeitos, por isso o operador lógico &&.

Teste. Coloque 10.1, 10.5, 21.12, -19, -0.000001, "java progressivo".

Você tem que fazer muito isso: testar.

Nas grandes empresas tem gente contratada só pra isso: testar (estagiários ou bolsistas).

Eliminada as possibilidades de notas absurdas, prossigamos.

Parte 2: Checando se passou direto

Vamos colocar mais um if else que vai checar se o aluno passou direto ou não.

Por hora, esqueça o código feito.

Como seria esse código pra testar se ele passou direto? Fácil:

```
if( nota >= 7.0 ){  
  
    System.out.println("Parabéns, você passou direto. Já sei, você programa em  
    Java?");  
  
}  
  
else {  
  
    System.out.println("Não passou direto");  
  
}
```

Vamos colocar esse trecho de código abaixo daquele print que diz 'nota válida'.

Então, vamos pensar como o Java pensa:

Primeiro ele recebe o resultado.

Checa se está entre 0.0 e 10.0. Se não está, pula pro else e terminou o programa.

Se estiver entre 0.0 e 10.0, diz que a nota é válida e depois faz outro teste em outro if, pra checar se a nota é maior que 7.0, se for diz passou direto, se não for diz que não passou direto.

O código ficaria assim:

```
import java.util.Scanner;
```

```
public class PasseiOuNao {
```

```
public static void main(String[] args) {
```

```
    float nota; //vai armazenar a nota
```

```
    Scanner entrada = new Scanner(System.in);
```

```
    System.out.print("Digite sua nota [0.0 - 10.0]: " );
```

```
    nota = entrada.nextFloat();
```

```
    if( (nota <= 10.0) && (nota >= 0.0) ){
```

```
System.out.println("Nota válida");
```

```
if( nota >= 7.0 ){
```

```
System.out.println("Parabéns, você passou direto. Por acaso você programa em  
Java?");
```

```
    }
```

```
else {
```

```
System.out.println("Não passou direto");
```

```
    }
```

```
}
```

```
else {
```

```
System.out.println("Nota inválida, fechando aplicativo");
```

```
    }
```

```
}
```

}

Teste.

Note que o Java não perdoa. Se tirar 6.99 não passa direto!

Parte 3: Checando se você ainda há esperanças

Bom, agora vamos checar se o aluno ainda pode fazer recuperação, ou seja, se a nota está entre 5.0 e é menor que 7.0.

Caso seja exatamente 7.0, passará automaticamente.

O código pra essa parte é o seguinte:

```
if( nota >= 5.0 ){
```

```
    System.out.println("Vai ter que fazer recuperação");
```

```
}
```

```
else {
```

```
    System.out.println("Reprovado. Ainda bem que é só simulação, hein?");
```

```
}
```

Note que se não for maior ou igual a 5.0, é menor, então vai pro else.

Ora, se é menor, foi reprovado. Então o else manda mensagem de reprovação.

A pergunta é, onde vamos inserir esse código?

No else do trecho 'Não passou direto'

Então, o código completo fica:

```
import java.util.Scanner;
```

```
public class PasseiOuNao {
```

```
    public static void main(String[] args) {
```

```
        float nota; //vai armazenar a nota
```

```
        Scanner entrada = new Scanner(System.in);
```

```
        System.out.print("Digite sua nota [0.0 - 10.0]: " );
```

```
        nota = entrada.nextFloat();
```

```
        if( (nota <= 10.0) && (nota >= 0.0) ){
```

```
            if( nota >= 7.0 ){
```

```
                System.out.println("Parabéns, você passou direto. Por acaso você programa em Java?");
```

```
            }
```

else {

if(nota >= 5.0){

System.out.println("Vai ter que fazer recuperação");

}

else {

System.out.println("Reprovado. Ainda bem que é só simulação, hein?");

}

}

}

else {

System.out.println("Nota inválida, fechando aplicativo");

}

```
}  
  
}
```

O Java lê e interpreta seguindo esses passos:

1; Primeiro checa se a nota é válida (se está entre 0 e 10)

Se não está, vai pro else, que termina o programa.

Se está, vai pro próximo if, que é o passo 2.

2. Checa se tirou mais que 7.0 através do próximo if

Se sim, ele passa direto e termina o programa.

Se não tirou mais que 7.0, vai pro else, que é o passo 3.

3. Checa se tirou mais que 5.0 através do próximo if

Se sim, diz que ficou de recuperação e termina o programa.

Se não, vai pro else, que diz que foi reprovado e terminou o programa.

Por questão de organização, coloque os pares if-else na mesma linha vertical, veja como fica mais organizado, e assim você vai saber à qual if o else pertence (sim, você pode se confundir).

Pintei os pares correspondentes para ficar mais claro.

Isso se chama indentação!

Exercício:

Refaça o mesmo problema.

Teste se a nota está entre 0.0 e 10.0.

Depois se foi reprovado direto, se sim, termina o programa.

Se não, vai pra outro if pra saber se está de recuperação. Se estiver, termina.

Se não estiver de recuperação, vai pro else que diz que ele passou direto.

Ou seja, é o mesmo problema, mas com a lógica ao contrário.

Obviamente, deve funcionar da mesma maneira.

Isso vai mostrar como um mesmo problema pode ser feito de várias maneiras.

Operadores Matemáticos de Incremento (++) e Decremento (–)

Nesta lição iremos aprender duas 'mão na roda', especialmente úteis para estudar laços while e for.

Esses operadores simplesmente somam e subtraem uma unidade.

Parece simples, não?

Ao fim do curso, você será fã número 1 dos operadores de incremento e decremento.

Operadores de incremento e decremento em Java

Em muitas aplicações Java temos a necessidade de contar, ou controlar coisas:

quantas coisas foram digitadas

quantos tiros você deu no jogo

quanto tempo se passou

quantos alunos foram cadastrados em um sistema

etc.

Geralmente, contamos de um em um. Imagine, gerenciar e contar um sistema de cadastro com milhões de inscritos, como o do Enem?

Ainda bem que você é programador Java e não se assustou, pois sabe que não vai fazer isso.

Você é paladino das artes computacionais, vai programar o computador pra fazer isso ;)

Então, boa parte das contas são feitas utilizando os 'laços', que vai aprender já já e vai utilizar pra sempre.

Com ele, você vai fazer com que o computador faça o tanto de contas que quiser...10, 20, mil, 1 milhão...o Google realiza trilhões por dia.

Você pode até fazer um looping, que é um laço sem fim e travar seu computador. Uns chamam de hackear, eu prefiro chamar de estudo e vou mostrar como fazer.

Esses laços, porém, tem que ser incrementado. Geralmente eles tem um início, uma condição e um fim.

Como esse fim é atingido? Através de uma ou mais variável que vai crescendo ou diminuindo.

Quando atinge um certo ponto, o laço vai parar.

É aí que entra os operadores de incremento e decremento.

Somando e Subtraindo variáveis em Java

O incremento quer dizer:

$a = a + 1$

Ainda lembra como resolver isso?

Passa o 'a' pro outro lado: $a - a = 1$

Aí fica: $0 = 1$

Ou seja, não é isso o que você tinha em mente.

$a = a + 1$, em Java, aliás, em Programação, é algo totalmente diferente da Matemática convencional.

Vamos ver na prática, que depois explico.

Abra seu NetBeans, declare um inteiro, atribua um valor e imprima.

Agora faça:

$a = a + 1;$

Logo após atribuir 1 ao 'a', imprime, veja o resultado e tente descobrir o que aconteceu.

Substitua por: $a = a + 2$;

Ou $a = a + 3$;

```
public class incremento {
```

```
public static void main(String[] args) {
```

```
    int a=1;
```

```
    System.out.println(a);
```

```
}
```

```
}
```

Hackers, pessoas que descobrem as coisas sozinhas (e não criminosos ou vândalos, como pensam, erroneamente), fazem isso.

Testam, pensam e descobrem como as coisas funcionam.

Explicação:

Quando fazemos $a =$

é porque vamos atribuir um valor ao 'a', um novo valor, INDEPENDENTE do que ele tinha antes.

E qual o valor que vamos atribuir? $a + 1$!

Ou seja, o a vai ser agora o seu antigo valor mais 1!

Se $a=2$, e fazemos $a = a + 3$, o que estamos fazendo?

Estamos dizendo que o a vai mudar, vai ser seu antigo valor (2), mais 3! Ou seja, agora $a=5$

Faz sentido, não?

a++ e a--

Usaremos muito, mas MUITO mesmo o incremento e o decremento de unidade:

```
a = a + 1;
```

```
a = a - 1;
```

Porém, é chato ou da preguiça ficar escrevendo `a=a+1` e `a=a-1` o tempo inteiro.

Então, inventaram atalhos para isso!

`a = a + 1` pode ser representado por `a++` ou `++a`

`a = a - 1` pode ser representado por `a--` ou `--a`

Existe uma diferença entre `a--` e `--a`, ou entre `a++` e `++a`.

Mas isso é assunto pra outro artigo, por hora, fique com um código que mostra o uso do incremento e decremento.

Não deixem de rodar.

Como já dizia o filósofo: só creio compilando e vendo.

```
public class incremento {
```

```
public static void main(String[] args) {  
    int a=1;  
    int b=1;  
  
    System.out.println("Valor inicial de a = " + a);  
  
    System.out.println("Valor inicial de b = " + b);  
  
    System.out.println("Incrementando: a++");  
    a++;  
  
    System.out.println("Decrementando: b--");  
    b--;  
  
    System.out.println("Agora a = " + a);  
  
    System.out.println("Agora b = " + b);  
    }  
  
}
```

Exercício:

Teste com ++a e --b e veja se há diferença ou não.

Formate com printf, ao invés de println.

Operadores de Atribuição: +=, -=, *=, /= e %= | Java Progressivo

Vimos na aula passada de nossa apostila de Java sobre operadores de incremento ++ e decremento --, que são nada mais que atalhos para atribuições do tipo:

```
a = a + 1
```

```
a = a - 1
```

Porém, para evitar ficar escrevendo 'à toa', não foram criado somente estes atalhos e nem somente existem estes tipos de atribuições.

Vamos aprender mais alguns atalhos e dicas em Java que nos farão poupar tempo e escrita.

Operadores de atribuição em Java

Se você já sabe como funciona o: $a = a + 1$

Já parou pra pensar se existe e como funcionaria o: $a = a * a$?

Vamos lá, seja $a = 2$

Quanto seria: $a = a * a$?

Pela lógica, o 'a' receberia o produto de seus antigo valores, $2*2 = 4$. Correto

Você já estudou PG, progressão geométrica?

Esse seria um belo exemplo onde usaríamos essa atribuição. Note que o valor era 'a', agora é 'a²'

E seu professor, que ficava mandando você calcular PA, PG...logo logo você programar o computador pra fazer isso para você através do bendito Java.

Usaremos atribuição para fazer um aplicativo que calcula juros também.

Eu tenho um que, embora simples, é bem útil. Ele pergunta o tanto que vou botar no banco, o juros, o tanto de meses que vou deixar, e ele me informa o que vai render.

Ou eu digo o quanto que quero que renda e ele me diz os meses que devo deixar lá, para que renda o que eu preciso.

Sabendo programação, você vai fazer as coisas que você necessita. Esse foi meu exemplo.

Qual o seu? O que você precisa calcular todo dia?

Em que você perde tempo fazendo conta, perguntando pra alguém ou pesquisando?

Ao término da seção de Controle de Fluxo você já será capaz de fazer uma gama enorme de aplicativos em Java para seu propósito.

Atalhos que salvam vidas

Você já aprendeu que é bem melhor escrever `a++` ao invés de `a = a + 1`

E em relação as outras atribuições? Não achou que iríamos perder tempo escrevendo tanto assim né? Claro, que não.

Veja só os truques que são usados em Java, e em outras linguagens:

`a = a + b`, fazemos: `a +=b`

`a = a - b`, fazemos: `a -=b`

`a = a * b`, fazemos: `a *=b`

`a = a / b`, fazemos: `a /=b`

`a = a % b`, fazemos: `a %=b`

A lógica é simples. Você sempre vai ver dois números, `x` e `y`, por exemplo.

E um operador matemático, vamos chamar de `[]`

E vai ver o sinal de igualdade, `=`, que é o sinal de atribuição.

E vai ver a fórmula: `x [] = y`

Isso sempre vai representar isso: `x = x [] y`

Ou seja, o primeiro número, sempre vai receber a operação matemática [] de seu antigo valor com o número y.

E para não perder o costume, um código em Java para mostrar como é que se faz:

```
public class Atribuicao {  
  
public static void main(String[] args) {  
  
    int a=1;  
  
    int b=2;  
  
    System.out.println("Valor inicial a = " + a);  
  
    System.out.println("Valor inicial b = " + b);  
  
    System.out.println("Fazendo a +=b");  
        a +=b;  
  
    System.out.println("Agora a = " + a);  
  
    System.out.println();  
}
```

```
System.out.println("Fazendo a -=b");
```

```
    a -=b;
```

```
System.out.println("Agora a = " + a);
```

```
System.out.println("Fazendo a *=b");
```

```
    a *=b;
```

```
System.out.println("Agora a = " + a);
```

```
System.out.println("Fazendo a +=2 ");
```

```
    a +=2;
```

```
System.out.println("Agora a = " + a);
```

```
System.out.println();
```

```
System.out.println("Fazendo a /=b");
```

```
    a /=b;
```

```
System.out.println("Agora a = " + a);
```

```
System.out.println();
```

```
}
```

```
}
```

Operadores de atribuição e de incremento e decremento: diferenças entre $a=++b$ e $a=b++$

Já vimos o uso dos operadores de decremento e incremento, em Java.

E também como usar os operadores de atribuição: $+=$, $-=$, $*=$, $/=$ e $\%=$

Porém, devemos tomar alguns cuidados no uso deles, como, por exemplo, na diferença entre $a=++b$ e $a=b++$

Diferença de `a=++b` e `a=b++` em Java

Você já estudou e testou que usar isoladamente:

`a++` e `++a` não surte efeito.

Porém, surte na hora da atribuição.

Vamos lá, ao nosso 'teste hacker':

Seja `b=2`;

Só olhando e raciocinando, você seria capaz de descobrir quanto seria 'a' em cada caso:

`a = b++;`

`a = ++b;`

Explicações 1. a = b++

Mais uma vez, é um atalho em Java, uma forma mais simples de escrever as seguintes linhas:

```
a = b;
```

```
b++;
```

2. `a = ++b`

Analogamente, é o atalho que os programadores Java usam para representar as seguintes linhas de código:

```
++b;
```

```
a = b;
```

'Tenho que decorar todos esses atalhos em Java, então? Maldito Java!'

Não. Você é programador, você não decora, você pensa.

Não foi à toa que coloquei o 'teste hacker', foi pra aguçar seu raciocínio.

Em ambos os casos, 'b' é incrementado. A diferença é que 'a' recebe o valor de 'b' antes do incremento em um e depois do incremento em outro.

Como eu disse, em Java, você pode descobrir apenas olhando e pensando um pouco.

Veja:

No primeiro exemplo: `a = b++`

Qual a ordem, 'b' ou incremento? Primeiro o 'b'. Depois o de incremento ocorre, '++'.

Ou seja, 'a' vai receber o valor de 'b' primeiro, então a=2

Só depois que 'b' vai ser incrementado e vai se tornar b=3.

No segundo exemplo: a = ++b

Qual a ordem: 'b' ou incremento?

Primeiro o incremento, '++', depois que aparece o 'b', então só depois acontece a atribuição...Assim, primeiro ocorre o incremento, então b=3.

Só depois é que esse valor é atribuído para a.

Como diz a bíblia do programador Java: compilaís e verás a verdade.

Eis o código:

```
public class Increment {
```

```
public static void main(String[] args) {
```

```
    int a,
```

```
    b=1;
```

```
System.out.println("b = " + b);
```

```
System.out.println("a = b++ ");
```

```
a = b++;
```

```
System.out.println("Então: a = " + a);
```

```
System.out.println();
```

```
System.out.println("b = " + b);
```

```
System.out.println("a = ++b ");
```

```
a = ++b;
```

```
System.out.println("Então: a = " + a);
```

```
}
```

```
}
```

Laço WHILE

Nos últimos artigos aprendemos vários operadores e atalhos utilizados pelos programadores Java, como ++, +=, *= dentre outros.

Está na hora de colocá-los em ação e aprender um dos comandos mais importantes em qualquer linguagem de programação: os laços

Nesse artigo, explicaremos o que é e como usar o laço while.

Como usar o laço while em Java

While, do inglês 'enquanto' é um laço de repetição.

Cada ciclo de repetição é chamado de iteração.

Sua sintaxe é a seguinte:

```
while ( condição ){  
    códigos  
}
```

Ou seja, enquanto (while) a condição for verdadeira, os códigos entre chaves serão repetidos.

Mas qual a vantagem de ter um código repetido? Isso não parece absurdo?

Bom, quando pensamos em repetido, pensamos em algo estático, mas não isso o que acontece.

É aí que entra os operadores de incremento, decremento, atribuição e outros.

Nas linhas de código, geralmente as coisas vão 'mudando' a condição e à cada iteração a condição é testada.

Se a condição retornar um valor lógico verdadeiro, o código entre chaves é executado.

Caso o valor retornado seja falso, o laço while é então terminado e o programa Java segue normalmente.

Vamos ver um exemplo simples, que mostra os números de 1 até 10.

Contando até 10 em Java, com o laço while

O programa é simples.

Definimos uma variável inteira, 'count'.

Enquanto essa variável for menor que 10, seu valor será imprimido e será incrementado.

Ou seja será, será impresso 1, 2, 3, ..., 9, 10...opa!

Quando chegar em 11, a condição dentro do laço não será mais verdadeira, e o laço terminará!

Teste:

```
public class contando {  
  
    public static void main(String[] args) {  
  
        int count=1;  
  
        while(count<=10){  
  
            System.out.println(count);  
  
            count++;  
  
        }  
  
    }  
}
```

}

Fazendo um PA (progressão aritmética) com o laço while, em Java

O programa a seguir mostra os termos 'an' de uma PA, de termo inicial 'inicial' e razão 'razao', até um valor máximo 'valor_max'.

A diferença deste programa para o anterior é que os termos de uma PA crescem através da adição da razão ao termo anterior.

Em programação, poderíamos representar isso assim: $a_n = a_n + \text{razao}$

Porém, como vimos alguns operadores em Java, faremos isso: $a_n += \text{razao}$

Veja o resultado:

```
public class pa {  
  
    public static void main(String[] args) {  
  
        int inicial=1,  
  
        razao=3,  
  
        an=inicial,  
  
        valor_max=20;  
  
        System.out.printf("Elementos da PA, de valor inicial %d e razão %d, menores  
que %d\n", inicial, razao, valor_max );  
  
        while(an<=valor_max){
```



```
System.out.println(an);
```

```
an += razao;
```

```
}
```

```
}
```

```
}
```

Fazendo um PG (progressão geométrica) com o laço while, em Java

O programa a seguir mostra os termos 'gn' de uma PG, de termo inicial 'inicial' e quociente 'quociente', até um valor máximo 'valor_max'.

Os produtos de uma PG, progressão geométrica, crescem através do produto do termo anterior com o quociente.

Em programação, poderíamos representar isso assim: $gn = gn * \text{razao}$

Porém, como vimos alguns operadores em Java, faremos isso: $gn *= \text{razao}$

Veja o resultado:

```
public class pg {  
    public static void main(String[] args) {  
        int inicial=1,  
        quociente=2,  
        gn=inicial,  
        valor_max=32;  
  
        System.out.printf("Elementos da PG, de valor inicial %d e razão %d, menores  
que %d\n", inicial, quociente, valor_max );  
  
        while(gn<=valor_max){  
  
            System.out.println(gn);  
  
            gn *= quociente;  
        }  
    }  
}
```

}

}

}

Questões envolvendo laço WHILE

Usando o laço while, faça as seguintes questões em Java:

0. Programa em Java dos patinhos da Xuxa

Xuxa, a rainha dos baixinhos, criou uma música que tem o seguinte formato:

n patinhos foram passear

Além das montanhas

Para brincar

A mamãe gritou: Quá, quá, quá, quá Mas só n-1 patinhos voltaram de lá.

Que se repete até nenhum patinho voltar de lá.

Ao final, todos os patinhos voltam:

A mamãe patinha foi procurar

Além das montanhas

Na beira do mar

A mamãe gritou: Quá, quá, quá, quá

E os n patinhos voltaram de lá.

Crie um programa em Java que recebe um inteiro positivo do usuário e exibe a música inteira na tela, onde o inteiro recebido representa o número inicial n de patinhos que foram passear.

1. Programa em Java que mostra os números ímpares

Escreva um aplicativo em Java mostra todos os números ímpares de 1 até 100.

2. Programa em Java que mostra os números pares

Escreva um aplicativo em Java mostra todos os números pares de 1 até 100.

3. Programa em Java que mostra os números pares e ímpares

Escreva um aplicativo em Java que recebe inteiro e mostra os números pares e ímpares (separados), de 1 até esse inteiro.

4. Programa em Java que calcula a média das notas de uma turma

Escreva um programa que pergunte ao usuário quantos alunos tem na sala dele.

Em seguida, através de um laço while, pede ao usuário para que entre com as notas de todos os alunos da sala, um por vez.

Por fim, o programa mostra a média, aritmética, da turma.

5. Achando o maior número

Achar o maior, menor, média e organizar números ou sequências são os algoritmos mais importantes e estudados em Computação. Em Java não poderia ser diferente.

Em nosso curso, obviamente, também não será diferente.

Escreva um programa em Java que solicita 10 números ao usuário, através de um laço while, e ao final

mostre qual destes números é o maior.

6. Achando os dois maiores números

Escreva um programa em Java que solicita 10 números ao usuário, através de um laço while, e ao final

mostre os dois maiores números digitados pelo usuário.

7. Quadrado de asteriscos

Escreva um programa que lê o tamanho do lado de um quadrado e imprime um quadrado daquele tamanho com asteriscos. Seu programa deve funcionar para quadrados com lados de todos os tamanhos entre 1 e 20.

Para lado igual a 5:

8. Quadrado de asteriscos e espaços em branco

Escreva um programa que lê o tamanho do lado de um quadrado e imprime um quadrado daquele tamanho com asteriscos e espaços em branco. Seu programa deve funcionar para quadrados com lados de todos os tamanhos entre 1 e 20.

Para lado igual a 5:

* *

* *

* *

Loop infinito, controlando laços e loopings com o while

Usar laços, muitas vezes, pode ser uma tarefa perigosa.

Isso por conta de uma coisa chamada loop infinito.

Acontece sem querer, mas as vezes acontece de propósito...principalmente por pessoas más intencionadas.

Nesse artigo vamos estudar mais sobre esse tipo de loop, como acontecem, como fazer e como ter mais controle sobre laços.

Loop infinito em Java

No nosso estudo sobre o laço while em Java, vimos que ele só é executado se a condição que ele testa, a cada iteração, for verdadeira, e que dentro do código de execução, geralmente, ocorre uma alteração dessa condição para que uma hora ou outra, o laço pare.

Mas, e se o laço não parar? Ele vai rodar infinitamente?

Sim, vai.

Você é o programador. Você diz, o Java faz. Ele não vai te questionar.

E para que alguém iria querer um programa que não parasse?

E se você quiser montar um servidor que vai funcionar 24h/dia para sempre (até parar de funcionar)?

Ora, vai usar um loop sem fim, vai funcionar sempre.

Em muitos programas de pesquisas científicas, os computadores simplesmente não param de calcular. Se programa as tarefas, e deixa-se que elas façam todo o trabalho.

Controlar um laço e deixar o computador em um loop é uma forma de ter um controle maior sobre sua máquina.

Criando um loop infinito simples

```
public class chato {  
    public static void main(String[] args) {  
        while(true){  
            System.out.println("Loop infinito!");  
        }  
    }  
}
```


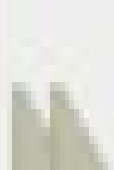

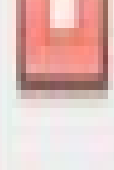

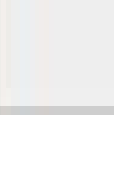
Esse é bem simples de entender.

Qual a condição testada? 'true'? Sim, ela retorna sempre o valor lógico 'true', então o código do while será sempre executado.

Em outras palavras, você pode entender o código como: enquanto 'verdade' -> executar o código

Rode e veja como fica lento.

Para parar o looping, clique no botão vermelho, o sinal universal de 'stop', que aparece na figura:

Variables	Breakpoints	On
	Loop infinito!	
	Loop infinito!	
	Loop infinito!	
	Loop infinito!	
	Loop infinito!	
	Loop infinito!	

O que esse laço while faz é simplesmente mostrar, infinitamente, uma mensagem na tela.

Mesmo uma coisa tão simples, é chata. E o pior, prejudicial ao computador.

Embora nossas máquinas atuais sejam bem potentes, ocorre um decréscimo considerável na memória RAM e gera uma lentidão no sistema.

Em sistemas antigos, poderia fazer um estrago.

Loops mais 'sofisticados' podem facilmente travar sua máquina, principalmente se você usar Windows.

Um vírus bem simples de ser criado é criar um loop que gere arquivos de texto, ou que fique adicionando linhas de string a esse arquivo de texto.

Em poucos minutos seu HD estará entupido.

Controlando os laços e looping

No exemplo simples de looping, a condição é sempre verdadeira. Aí não tem muito o que fazer. Ela foi definida assim e vai continuar assim.

Em Java, como em outras linguagens, não usamos esse operador booleano diretamente.

Ao invés disso, declaramos uma variável do tipo 'boolean' que vai definir se o laço deve continuar ou não.

E como isso vai ser decidido? Por você, durante a execução do programa.

No exemplo a seguir, criei a variável 'boolean' chamada 'continuar'. Inicialmente ela é 'true' para o while começar.

Se eu tivesse iniciado ela como 'false' o laço nem iniciaria e não existiria programa.

Declarei um tipo char 'opcao' que vai receber um caractere do usuário. Vai ser tipo uma senha. Só vai escapar do while se o usuário digitar essa senha, ou caractere.

Muito bem, o programa inicia e diz que para você sair da Matrix precisará digitar um caractere especial.

Como você é programador Java, está vendo que no teste condicional a senha é 'j', de java.

Se o usuário adivinhar a letra, o if retorna verdadeiro e altera o valor de

'continuar' para false, aí na próxima iteração o while não irá executar e o programa sai do loop.

Caso ele erre - o que é provável, visto que existem 26 letras e 10 números no teclado - a variável 'continuar', continua sendo true (pois nada foi alterado).

É o seu primeiro programa de 'segurança' em Java, pois exige uma senha. É possível transformar em executável ou inviabilizar que o usuário não veja essa senha, assim só o programador sabe a senha de acesso.

Você pode alterar de 'j' para a inicial de seu nome.

Quando estudarmos Strings, poderemos usar palavras ou frases, em vez de caracteres. Mas calma, já chegaremos lá. Por hora teste esse programe e entenda como funciona, usaremos bastante esse tipo de controle de fluxo em nossas aplicações Java:

```
import java.util.Scanner;

public class Matrix {

    public static void main(String[] args) {

        boolean continuar = true;

        char opcao;

        Scanner entrada = new Scanner(System.in);

        while(continuar){

            System.out.println("Você está na matrix;");
```

```
System.out.print("Digite o caractere especial para sair da matrix: ");  
  
opcao = entrada.next().charAt(0);  
  
if(opcao=='j'){  
  
    continuar=false;  
  
    System.out.println("Parabéns! Você conseguiu sair da Matrix!");  
  
}  
  
else{  
  
    System.out.println("Você não está autorizado a sair da Matrix. Estude Java.");  
  
};  
  
}  
  
}
```

Laço FOR: tendo um maior controle sobre as repetições

Veremos agora o tipo de laço mais importante e usado em Java, que será sempre visto nos códigos de nosso curso: o laço for

Sua fama se deve ao poder que temos por conta dos detalhes que o laço for é capaz de receber, sendo possível ter um maior controle sobre as repetições.

Diferenças entre o while e o for: para que serve o laço for em Java

Geralmente, o que é possível fazer usando o for, é possível fazer usando o o laço while em Java.

Então, naturalmente vem a pergunta: por que e para que serve, então, o laço for?

É o mesmo que perguntar 'por que usar `a++`' se podemos usar '`a=a+1`'.

Simple: por questão de simplicidade e praticidade.

Nós, programadores Java, devemos procurar sempre a máxima eficiência. Fazer as coisas da maneira mais simples possível.

O uso do for se deve aos seguintes dois fatos, que você deve ter notado, caso tenha feito os exercícios sobre o laço while e caso tenha visto os exemplos do artigo while:

1. Geralmente existe um ponto de partida, um contador inicial para o laço while
2. Geralmente esse contador, ou outro, muda. De modo incrementativo ou decrementativo, até que a condição do while não seja mais satisfeita e o laço termine.

O que o laço for faz é automatizar isso.

Já que na grande maioria das vezes precisaremos desses dois fatores (inicial e de mudança), o laço for irá prover isso.

Como usar o laço for em Java

A sintaxe do laço for é a seguinte:

```
for(cond inicial; teste condicional ; apos iteracao){
```

```
//código
```

```
}
```

O laço funciona da seguinte maneira:

O laço se inicia pela condição inicial. Geralmente se inicia o contador. Esse primeiro estágio SEMPRE acontece.

O segundo estágio é o teste da condição do laço, um simples teste condicional. Caso seja verdade, o código é executado.

Ao término de execução do código, ocorre o fator de mudança, que geralmente é um incremento ou decremento, sempre após cada iteração do looping,

Depois a condição é testada novamente. Caso retorne 'true', o código é executado.

Ao término de execução do código, sempre ocorre o fator de mudança...e assim sucessivamente.

Faremos 3 exemplos que mostram bem o exemplo do uso e flexibilidade laço for.

É importante que você aprenda bem, pois eles serão muito usados em nosso curso de Java.

Exemplo 1: Contando até 10, com laço for

```
public class for1 {  
  
public static void main(String[] args) {  
  
for(int count=1 ; count <= 10 ; count++){  
  
    System.out.println(count);  
        }  
    }  
  
}
```

Nesse exemplo, fiz questão de mostrar uma importante característica do laço for: a declaração dentro do for

Note que, na condição inicial do contador, foi declarado o inteiro 'count', que foi usado no laço.

Como funciona:

A primeira coisa que acontece é a condição inicial. No nosso caso, é criado um inteiro, 'count' e inicializado com o valor 1.

Depois, a condição é testada. Como ela é verdadeira, ele imprime o valor de 'count', que é 1.

Depois 'count' é incrementada e vira 2.

Depois é testada. Como continua ser menor que 10, ela é imprimida. Depois incrementada, testada, imprimida...ela é imprimida até ser 10, pois depois disso ela será incrementada e 'count' será 11, então a condição não será mais verdadeira e o laço for terminará.

Exemplo 2: contagem regressiva, usando o laço for

```
public class for2{  
  
public static void main(String[] args) {  
  
for(int count=10 ; count >= 1; count--){  
  
    System.out.println(count);  
  
        }  
  
    }  
  
}
```

Esse exemplo serve para mostrar que podemos usar o decremento, não somente o incremento, dentro do laço for.

Como funciona:

A primeira coisa que acontece é a condição inicial. No nosso caso, é criado um inteiro, 'count' e inicializado com o valor 10.

Depois, a condição é testada. Como 'count' é maior ou igual a 1, é verdadeira e se imprime o valor de 'count', que é 10.

Após essa execução, 'count' se torna 9, e continua a ser maior ou igual a 1, e impressa...quando 'count' for 1, ela será impressa, por 1 é maior ou igual a 1.

Porém, ao ser decrementada, se tornará 0 e o laço for terminará.

Exemplo 3: contagem progressiva e regressiva no mesmo laço for

Outra característica do laço for, é que não precisamos usar só uma variável de controle ('count') ou testar somente uma condição. O código Java para isso é:

```
public class for3 {  
  
public static void main(String[] args) {  
  
for(int sobe=1, desce=10 ; sobe<=10 && desce>=1; sobe++, desce--){  
  
    System.out.printf("%d \t %d \n", sobe, desce);  
  
        }  
  
    }  
  
}
```

Como funciona:

Declaramos duas variáveis, a 'sobe', que vai ser incrementada e impressa de 1 até

10,

e a variável 'desce' que será decrementada e impressa de 10 até 1.

Dentro do printf usamos o caractere '\t', que é o código ASCII para a tabulação, ou seja, o nosso famoso 'tab'.

Compile e rode esse simples código, verá como é maravilhoso ser programador Java e sua vida fará mais sentido.

Questões usando o laço FOR

Exercícios sobre o laço FOR em Java

Usando o laço for, faça as seguintes questões em Java:

0. Programa em Java dos patinhos da Xuxa

Xuxa, a rainha dos baixinhos, criou uma música que tem o seguinte formato:

n patinhos foram passear

Além das montanhas

Para brincar

A mamãe gritou: Quá, quá, quá, quá

Mas só n-1 patinhos voltaram de lá.

Que se repete até nenhum patinho voltar de lá.

Ao final, todos os patinhos voltam:

A mamãe patinha foi procurar

Além das montanhas

Na beira do mar

A mamãe gritou: Quá, quá, quá, quá

E os n patinhos voltaram de lá.

Crie um programa em Java que recebe um inteiro positivo do usuário e exibe a música inteira na tela, onde o inteiro recebido representa o número inicial n de patinhos que foram passear.

1. Programa em Java que mostra os números ímpares

Escreva um aplicativo em Java mostra todos os números ímpares de 1 até 100.

2. Programa em Java que mostra os números pares

Escreva um aplicativo em Java mostra todos os números pares de 1 até 100.

3. Programa em Java que mostra os números pares e ímpares

Escreva um aplicativo em Java que recebe inteiro e mostra os números pares e ímpares (separados), de 1 até esse inteiro.

4. Programa em Java que calcula a média das notas de uma turma

Escreva um programa que pergunte ao usuário quantos alunos tem na sala dele.

Em seguida, através de um laço for, pede ao usuário para que entre com as notas de todos os alunos da sala, um por vez.

Por fim, o programa mostra a média, aritmética, da turma.

5. Achando o maior número

Achar o maior, menor, média e organizar números ou sequências são os algoritmos mais importantes e estudados em Computação. Em Java não poderia ser diferente.

Em nosso curso, obviamente, também não será diferente.

Escreva um programa em Java que solicita 10 números ao usuário, através de um laço for, e ao final

mostre qual destes números é o maior.

6. Achando os dois maiores números

Escreva um programa em Java que solicita 10 números ao usuário, através de um laço for, e ao final

mostre os dois maiores números digitados pelo usuário.

7. Quadrado de asteriscos

Escreva um programa que lê o tamanho do lado de um quadrado e imprime um quadrado daquele tamanho com asteriscos. Seu programa deve funcionar para quadrados com lados de todos os tamanhos entre 1 e 20.

Para lado igual a 5:

8. Quadrado de asteriscos e espaços em branco

Escreva um programa que lê o tamanho do lado de um quadrado e imprime um quadrado daquele tamanho com asteriscos e espaços em branco. Seu programa deve funcionar para quadrados com lados de todos os tamanhos entre 1 e 20.

Para lado igual a 5:

* *

* *

* *

9. Construa um aplicativo em Java para gerar 20 números de 1000 a 1999 e mostrar aqueles que divididos por 11 deixam resto 5.

Solução 09:

Para gerar números aleatório, temos que importar a classe Random, que provém os métodos para gerar números aleatórios:

```
import java.util.Random;
```

Passo 1:

Precisamos criar um objeto do tipo Random, o objeto 'randomGenerator'.

Ele que irá 'cuidar' da geração e dos tipos (inteiros, decimais etc).

Passo 2:

Um laço de 20 iterações, onde iremos gerar 20 números inteiros.

Passo 3:

O trecho: `randomGenerator.nextInt(1000)`

Ir  gerar n meros aleat rios num intervalo de 1000 n meros, de 0 at  999.

Como queremos que gere n meros entre 1000 at  1999, adicionamos 1000, ficando: `randomGenerator.nextInt(1000) + 1000`

Passo 4:

Usaremos o operador '%' (módulo ou resto da divisão) e um condicional 'if' para imprimir somente aqueles números que deixam resto 5, quando divididos por 11.

O código fica:

```
import java.util.Random;
```

```
public class aleatorio1 {
```

```
public static void main(String[] args) {
```

```
    // Passo 1: preparando o gerador
```

```
Random randomGenerator = new Random();
```

```
    // Passo 2: gerando 20 números
```

```
for(int count=1 ; count <= 20 ; count++){
```

```
// Passo 3: gerando um número entre 1000 e 1999
```

```
int num_aleatorio = randomGenerator.nextInt(1000) + 1000;
```

```
// Passo 4: imprimindo somente os que deixam resto 5 na divisão por 11
```

```
if(num_aleatorio % 11 == 5)
```

```
System.out.println(num_aleatorio);
```

```
    }
```

```
}
```

```
}
```

Segunda maneira:

Declaramos o inteiro só uma vez, e 'alimentamos' o gerador à cada iteração:

```
import java.util.Random;
```

```
public class aleatorio2 {
```

```
public static void main(String[] args) {
```

```
    int num_aleatorio;
```

```
    // Passo 1: gerando 20 números
```

```
for(int count=1 ; count <= 20 ; count++){
```

```
    // Passo 2: preparando o gerador
```

```
Random randomGenerator = new Random();
```

```
// Passo 3: gerando um número entre 1000 e 1999
```

```
num_aleatorio = randomGenerator.nextInt(1000) + 1000;
```

```
// Passo 4: imprimindo somente os que deixam resto 5 na divisao por 11
```

```
if(num_aleatorio % 11 == 5)
```

```
System.out.println(num_aleatorio);
```

```
    }
```

```
}
```

```
}
```

O laço do ... while: O laço que sempre acontece...pelo menos uma vez

Veremos nesse artigo mais um importante laço em Java, o laço do...while, que sempre executa o código do laço pelo menos uma vez, independente da condição do while ser true ou false.

Um dos maiores problemas do laço while é que ele só é executado se a condição contida nele for true.

Porém, muitas vezes, em nossos aplicativos Java, não podemos garantir que essa condição será sempre verdadeira ou queremos que o laço seja executado PELO MENOS UMA VEZ, e caso o usuário decida, o laço vai continuar ou não.

Então, o diferencial do laço do while é que ele sempre executa, sempre inicia o laço.

Como usar o laço DO WHILE em Java

A sintaxe do laço do .. while em Java, é a seguinte

do

{

//esse código será executado pelo menos uma vez

} while(condição);

Seu uso é bem simples e intuitivo, visto que o laço while nós já conhecemos.

O 'do' do inglês quer dizer 'faça'.

Ou seja, esse laço nos diz 'faça isso enquanto 'condição' for verdadeira'

Exemplos de uso

Um dos exemplos de uso é para exibir o menu. Ora, o menu tem que ser exibido pelo menos uma vez.

Se depois, você quiser sair, é opção sua.

Mas que ele tem que ser exibido ao menos uma vez, ele tem. E uma boa saída para fazer isso, em Java, é usar o laço do ... while.

No exemplo a seguir, o menu é mostrado.

Para parar de exibir o menu, basta digitar 0, que o boolean irá se tornar 'false' e o laço não mais se repetirá, pois a condição dentro do while será falsa.

Caso a entrada do usuário seja qualquer outro número que não 0, a variável boolean continua 'true', como foi declarada e o laço continuará a rodar (ou seja, o menu continua a ser exibido).

O trecho: `System.out.printf("\n\n\n\n\n\n");`

É simplesmente para limpar a tela.

Teste para ver:

```
import java.util.Scanner;
```

```
public class DoWhile {
```

```
public static void main(String[] args) {
```

```
    boolean continuar=true;
```

```
    int opcao;
```

```
    Scanner entrada = new Scanner(System.in);
```

```
do
```

```
    {
```

```
        System.out.println("\t\tMenu de opções do curso Java Progressivo:");
```

```
        System.out.println("\t1. Ver o menu");
```

```
        System.out.println("\t2. Ler o menu");
```

```
        System.out.println("\t3. Repetir o menu");
```

```
        System.out.println("\t4. Tudo de novo");
```

```
System.out.println("\t5. Não li, pode repetir?");
```

```
System.out.println("\t0. Sair");
```

```
System.out.print("\nInsira sua opção: ");
```

```
    opcao = entrada.nextInt();
```

```
if(opcao == 0){
```

```
    continuar = false;
```

```
System.out.println("Programa finalizado.");
```

```
    }
```

```
else{
```

```
System.out.printf("\n\n\n\n\n\n");
```

```
    }
```

```
    } while( continuar );
```

```
}
```

}

Embora este seja um exemplo simples, ele será a base para vários menus que iremos fazer em nosso Curso de Java.

Por exemplo, como um sistema de cadastro de alunos e um esquema de caixa eletrônico.

Os comandos break e continue: interrompendo e alterando fluxos e loopings

Nesta parte de nossa apostila online iremos ver os comandos break e continue, do Java, que servem para nos propiciar um maior controle sobre o fluxo dos aplicativos, especialmente das iterações dos loopings.

Para que servem os comandos BREAK e CONTINUE em Java

Geralmente, em nossos programas de Java, iremos usar os laços while, for e do ... while para procurar algum item, número ou checar alguma condição.

Por exemplo, imagine que você foi contratado para fazer um programa, em Java, claro, para um banco.

Em um momento do aplicativo, o cliente insere o número de sua conta e o programa vai buscar esse número no banco de dados do sistema, que tem milhões de clientes cadastrados.

Ele vai fazer essa busca através de um looping.

Porém, imagine que o programa encontra os dados do cliente logo no começo, logo nas primeiras posições.



BREAK

© 2008 Microsoft Corporation. All rights reserved.



CONTINUE

© 2008 Microsoft Corporation. All rights reserved.

E aí, vai checar todo o resto do banco de dados?

Claro que não, isso seria perda de tempo.

É aí que entram os comandos break e continue. Esse seria um bom exemplo onde daríamos um 'break' no laço, pois já encontramos o que queríamos.

O comando break

Break significa quebrar, parar, frear, interromper. E é isso que se faz.

Quando o Java encontra esse comando pela frente, ele interrompe o laço/estrutura de controle ATUAL, como o while, for, do ... while e o switch (que veremos no próximo artigo da apostila).

Vamos mostrar um exemplo do uso do break através de um exemplo matemático.

Exemplo de uso:

Suponha que você é um cientista e quer saber se entre os números 1 e um milhão existe um número que é múltiplo de 17 e 19, ao mesmo tempo.

Ou seja, queremos saber se existe um número entre 1 e um milhão que deixa resto 0 na divisão por 17 e por 19.

Caso exista, o imprima. E só imprima o menor.

Poderíamos fazer um laço de 1 até 1 milhão, e testar isso através de um for.

Ok, é uma solução. O principal na vida profissional de um programador é saber resolver o problema.

Vamos usar um método chamado 'currentTimeMillis()', que retorna um tipo 'long' com o tempo atual do sistema em mili segundos. Vamos declarar esse tipo no início e ao fim do laço, e depois subtrair esses valores, assim teremos o tempo de execução do programa.

Após imprimir o menor número, o boolean se torna falso, assim só o primeiro número é impresso. Veja:

```
public class breakTest {
```

```
public static void main(String[] args) {
```

```
    long i = System.currentTimeMillis();
```

```
        boolean imprimir = true;

        for(int count=1 ; count <=1000000 ; count++){

            if((count % 17 == 0) && (count % 19 == 0))

                if(imprimir){

                    System.out.println(count);

                        imprimir=false;

                    }

                }

            System.out.println("Tempo de execução, em milisegundos: "+
            (System.currentTimeMillis() -i));

        }

    }
```

Em minha máquina deu:

26 mili segundos

Mas se você quiser ser um bom profissional, não basta só saber resolver. Tem que resolver e da melhor maneira possível.

Note que que o menor número achado é 323, e o laço vai até 1 milhão! Ou seja, ele percorre de 324 até 1 milhão à toa, pois já achou o número desejado!

Ora, se ele já achou o número, 323, vamos fazer com que o laço pare, usando o comando break:

```
public class break {
```

```
public static void main(String[] args) {
```

```
    long i = System.currentTimeMillis();
```

```
    for(int count=1 ; count <=1000000 ; count++){
```

```
        if((count % 17 == 0) && (count % 19 == 0)){
```

```
            System.out.println(count);
```

```
            break;
```

```
        }
```

```
}
```

```
System.out.println("Tempo de execução, em milisegundos: "+  
(System.currentTimeMillis() -i));
```

```
}
```

```
}
```

8 mili segundos

Menos de um terço do tempo!

Você pode pensar "Ah, de 26 mili segundos para 8 mili segundos a diferença é insignificante".

Concordo com você.

Porém, no futuro você fará aplicações maiores e que levam mais tempo, e esses pequenos detalhes farão a diferença.

Por exemplo, existem métodos que, durante um game são facilmente chamados milhões de vezes.

Multiplique essa diferença 'insignificante' de alguns mili segundos por milhões de vezes e terá um belo de um 'lag', ou lentidão. Isso se chama otimizar: fazer da maneira mais eficiente possível.

Daremos bastante enfoque para as otimizações em nosso curso de Java, como verá no exemplo a seguir.

O comando continue

Como o nome diz, ele 'continua' o laço. O comando break interrompe o laço, já o continue interrompe somente a iteração atual.

Não basta porém ser um bom profissional. Você está no curso Java Progressivo, você tem que ser um dos melhores!

Nós vamos otimizar o código passado.

Note uma coisa, queremos achar um número que seja múltiplo de 17 e 19. Ora, tal número não pode ser par, pois 17 e 19 são ímpares.

Para cada número 'count', estamos fazendo dois testes: se é múltiplo de 17 e se é múltiplo de 19.

Vamos otimizar da seguinte maneira, vamos fazer um só teste: vamos checar se é múltiplo de 2. Caso seja, nem adianta testar se é múltiplo de 17 e 19, podemos pular essa iteração.

E como pulamos uma iteração? Com o comando continue!

Veja como fica:

```
public class continueTest {
```

```
public static void main(String[] args) {  
    long i = System.currentTimeMillis();  
  
    for(int count=1 ; count <=1000000 ; count++){  
  
        if(count % 2 == 0){  
  
            continue;  
        }  
  
        if((count % 17 == 0) && (count % 19 == 0)){  
  
            System.out.println(count);  
  
            break;  
        }  
  
    }  
  
    System.out.println("Tempo de execução, em milisegundos: "+
```

```
(System.currentTimeMillis() -i));
```

```
}
```

```
}
```

E temos o impressionante tempo de 1mili segundo!

De 26 mili segundos, fomos para 1 mili segundo! Impressionante esse Java, não?

O comando switch: fazendo escolhas em Java



No artigo sobre o laço do ... while de nossa apostila online Java Progressivo, mostramos como criar um menu simples.

Porém, o Java possui um recurso específico para esse propósito, que nos permite optar por algo, dependendo do que tenhamos digitado, é o comando switch.

O comando SWITCH em Java

Colocamos várias opções e vários comandos dentro do comando switch, todas as possibilidades de nosso aplicativo ou todas as opções ou rumos que nossos programas possam tomar.

O switch vai funcionar como um interruptor, pois dependendo da entrada que você der a ele, ele vai acionar somente certo(s) comando(s) dentre os que você disponibilizou.

É como se você criasse um menu, ou cardápio, e com o switch você escolhesse o que vai querer.

Declaração e Sintaxe do comando switch

Em Java, usamos e declaramos o comando switch da seguinte maneira:

```
switch( opção )
```

```
{
```

case opção1:

comandos caso a opção 1 tenha sido escolhida

```
break;
```

case opção2:

comandos caso a opção 2 tenha sido escolhida

```
break;
```

case opção3:

comandos caso a opção 3 tenha sido escolhida

```
break;
```

default:

comandos caso nenhuma das opções anteriores tenha sido escolhida

}

A variável 'opção' geralmente é um inteiro ou caractere (também pode ser byte ou short), que o usuário digita através da classe Scanner.

Se 'opção' receber 'opção1' como entrada, são os códigos contido na 'case opção1' que serão executados.

Se 'opção' receber 'opção2' como entrada, são os códigos contido na 'case opção2' que serão executados.

Se 'opção' receber 'opção3' como entrada, são os códigos contido na 'case opção3' que serão executados.

Se 'opção' receber qualquer outra coisa que não seja 'opção1', 'opção2' ou 'opção3', são os códigos contido em 'default' que serão executados.

Exemplo de uso do comando SWITCH em Java:

Vamos criar uma calculadora que faz as operações básicas usando o comando switch.

É um aplicativo Java simples que recebe 3 dados: dois números e um caractere.

Esse caractere poderá ser '+', '-', '*' ou '/' , e representarão a operação matemática que você deseja realizar entre os números.

Vejamos:

```
import java.util.Scanner;
```

```
public class switchTest {
```

```
public static void main(String[] args) {
```

```
    float numero1, numero2;
```

```
    char operacao;
```

```
    Scanner entrada = new Scanner(System.in);
```

```
System.out.print("Escolha sua operação [+ - * / ]: ");
```

```
operacao = entrada.nextLine().charAt(0);
```

```
System.out.print("Entre com o primeiro número: ");
```

```
numero1 = entrada.nextFloat();
```

```
System.out.print("Entre com o segundo número: ");
```

```
numero2 = entrada.nextFloat();
```

```
System.out.println();
```

```
switch( operacao )
```

```
{
```

```
case '+':
```

```
System.out.printf("%.2f + %.2f = %.2f", numero1, numero2, numero1 +  
numero2);
```

```
break;
```


case '-':

```
System.out.printf("%.2f - %.2f = %.2f", numero1, numero2, numero1 -  
numero2);
```

```
break;
```

case '*':

```
System.out.printf("%.2f * %.2f = %.2f", numero1, numero2, numero1 *  
numero2);
```

```
break;
```

case '/':

```
System.out.printf("%.2f / %.2f = %.2f", numero1, numero2, numero1 /  
numero2);
```

```
break;
```

default:

```
System.out.printf("Você digitou uma operação inválida.");
```

```
}
```

```
}
```

```
}
```

Caso a 'opção' seja um char, coloque entre aspas simples ", caso seja string coloque entre aspas duplas "" e caso seja um número, não é necessário colocar nenhum tipo de aspas.

O real funcionamento do comando switch: sem o comando break

Omiti, propositalmente, uma informação importante.

Só é realizado um 'case' de cada vez por conta do 'break' contido em cada comando dentro dos 'case'.

Na verdade, o switch seleciona o 'case' através da 'opção' e faz com que todos os 'case' a partir daquele sejam executados.

O exemplo a seguir ilustra o real funcionamento do comando switch em Java, experimente digitar a vogal 'a':

```
import java.util.Scanner;
```

```
public class switchTest2 {
```

```
public static void main(String[] args) {
```

```
    char vogal;
```

```
    Scanner entrada = new Scanner(System.in);
```

```
    System.out.print("Digite uma vogal minúscula: ");
```

```
    vogal = entrada.nextLine().charAt(0);
```

```
switch( vogal )
```

```
{
```

```
case 'a':
```

```
System.out.println("Você está no case da vogal 'a'");
```

```
case 'e':
```

```
System.out.println("Você está no case da vogal 'e'");
```

```
case 'i':
```

```
System.out.println("Você está no case da vogal 'i'");
```

```
case 'o':
```

```
System.out.println("Você está no case da vogal 'o'");
```

```
case 'u':
```

```
System.out.println("Você está no case da vogal 'u'");
```

default:

```
System.out.println("Você não digitou uma vogal minúscula");
```

```
}
```

```
}
```

```
}
```

Exemplo da utilidade do comando switch sem o break

À priori parece ser estranho essa propriedade do comando switch 'rodar' todos os cases.

Porém, isso é uma mão na roda e se bem utilizado, só vai nos ajudar. E caso não precise, simplesmente coloque um break.

Vamos pegar o exemplo passado e deixá-lo mais robusto, à prova de usuários que tentarão digitar as vogais maiúsculas.

É uma prática bem comum em Java, é essencial que aprenda:

```
import java.util.Scanner;
```

```
public class switchTest3 {
```

```
public static void main(String[] args) {
```

```
    char vogal;
```

```
    Scanner entrada = new Scanner(System.in);
```

```
    System.out.print("Digite uma vogal: ");
```

```
    vogal = entrada.nextLine().charAt(0);
```

```
switch( vogal )
```

```
{
```

```
case 'a': case 'A':
```

```
System.out.println("Você digitou 'a' ou 'A' ");
```

```
break;
```

```
case 'e': case 'E':
```

```
System.out.println("Você digitou 'e' ou 'E' ");
```

```
break;
```

```
case 'i': case 'I':
```

```
System.out.println("Você digitou 'i' ou 'I' ");
```

```
break;
```

case 'o': case 'O':

System.out.println("Você digitou 'o' ou 'O' ");

break;

case 'u': case 'U':

System.out.println("Você digitou 'u' ou 'U' ");

break;

default:

System.out.println("Você não digitou uma vogal !");

}

}

}

Note que estamos acumulando os case minúsculos e maiúsculos de cada vogal para gerarem o mesmo comando. Essa é a prática comum. No caso, o comando é print.

Mais um exemplo útil do comando switch sem o break

Suponha que você atrasou uma conta. A cada mês que você deixa de pagar, será cobrado 1% de juros no valor inicial.

Ou seja, se você atrasar um mês, irá pagar 1%. Se atrasar 3 meses, irá pagar 3% etc.

Vamos supor que você pode atrasar, no máximo, 5 meses.

O programa pede, como entrada, dois valores:

- um float: com o valor de sua dívida inicial (valor_i)
- um inteiro: de 0 até 5, que são os meses de atraso.

Nosso programa ficaria assim:

```
import java.util.Scanner;
```

```
public class switchTest4 {
```

```
public static void main(String[] args) {
```

```
    float valor_i, valor_f, juros=0;
```

```
int meses;
```

```
Scanner entrada = new Scanner(System.in);
```

```
System.out.print("Qual o valor inicial da dívida: ");
```

```
valor_i = entrada.nextFloat();
```

```
System.out.print("Você vai atrasar quantos meses [0-5]?: ");
```

```
meses = entrada.nextInt();
```

```
switch( meses )
```

```
{
```

```
case 5:
```

```
    juros++;
```

```
case 4:
```

```
    juros++;
```

```
case 3:
```

```
    juros++;
```

case 2:

juros++;

case 1:

juros++;

break;

default:

System.out.println("Você não digitou um valor válido de meses");

}

System.out.println("Juros: "+juros+"%");

valor_f=(1 + (juros/100))*valor_i);

System.out.printf("Valor final da dívida: R\$ %.2f", valor_f);

}

}

Se cair no case 5, vai contar todos os case, até o break, somando juros=5

Se cair no case 4, vai contar todos os case, até o break, somando juros=4

Assim, analogamente, para case 3, 2 e case 1.

Ou seja, nós utilizamos o fato dos case irem se acumulando.

Exercício:

Crie um programa que receba um inteiro, de 1 até 12, representando os meses do ano e retorne o número de dias do mês.

Use switch e não use break. Acumule os case.

Solução:

Inicialmente, a variável 'dias' é declarada como tendo 31 dias.

Caso seja o mês 4, 6, 9 ou 11, é subtraído 1 da variável 'dias' e o programa informa que esses meses possuem 30 dias.

Caso seja o mês 2, é subtraído 2 de 'dias', ficando 28 dias para o mês de fevereiro.

Caso não seja nenhum desses meses, não cai no switch, então continua com 31 dias (que são os meses 1, 3, 5, 7, 8, 10 e 12).

Código:

```
import java.util.Scanner;
```

```
public class months {
```

```
public static void main(String[] args) {
```

```
    int mes, dias=31;
```

```
    Scanner entrada = new Scanner(System.in);
```

```
    System.out.print("Digite o mês [1-12]: ");
```

```
    mes = entrada.nextInt();
```

```
    if(mes>12 || mes<1){
```

```
        System.out.println("Mês inválido");
```

```
    return;
```

```
    }
```

```
    switch( mes )
```

```
{  
    // fevereiro: subtraímos 2 dias aqui e 1 dia no próximo case  
  
    case 2:  
        dias -=2;  
  
        //meses que possuem 30 dias: só subtraímos 1 dia  
  
    case 4: case 6: case 9: case 11:  
        dias--;  
    }  
  
    System.out.printf("O mês %d possui %d dias", mes, dias);  
    }  
}
```

Problemas envolvendo laços



Dizer 'use isso' ou 'use aquilo' é fácil.

Agora teste sua criatividade Javeana e invente seu jeito para resolver os seguintes problemas, mas se lembre de usar:

[if else](#)

[while](#)

[for](#)

[do ... while](#)

switch.

para exercitar seu aprendizado.

Exercícios sobre laços em Java

0. Escreva um programa em Java que recebe 'n' números do usuário, e recebe o número 'n' também, e determine qual destes números é o menor.

1. Escreva um programa em Java que recebe um inteiro 'n' do usuário e calcula o produto dos números pares e o produtos dos números ímpares, de 1 até n.

2. Faça um programa em Java que recebe um inteiro do usuário e calcula seu fatorial.

O fatorial de 'n' é dado por:

$$n*(n-1)*(n-2)...*3*2*1$$

e é representado por n!

3. Crie um aplicativo bancário em Java que pede o valor do depósito inicial, o valor do investimento mensal e o número de meses que o dinheiro vai ficar rendendo na poupança.

Após isso, calcule o lucro obtido, sabendo que o juros da poupança é de 0,5%.

Desafio do diamante de asteriscos:

Escreva um aplicativo Java que peça um número inteiro ímpar ao usuário e desenhe um diamante no seguinte formato:

```

*
***
*****
*****
*****
*****
*****
***
*

```

Nesse caso, o número é 9, pois há 9 colunas e 9 asteriscos na linha central.

Solução do desafio:

Há duas coisas na figura: espaços em branco e asteriscos.

Vamos resolver esse desafio analisando o padrão do número de asteriscos (variável asteriscos) e do número de espaços em branco (variável espacos).

Antes de desenhar o diamante, temos que checar se o número que o usuário forneceu é ímpar, através de um teste condicional:

```
if(numero%2 != 0)
```

Se não for, o programa cai no else e nossa aplicação Java é finalizada.

Vamos desenhar o dito cujo.

Parte de cima do diamante

Qual o número inicial de espaços?

Note que é sempre: $(\text{numero}-1)/2$

...e essa é a lógica do problema, a parte mais difícil, que é notar esse padrão.

E o número de asteriscos inicial?

Fácil, é 1 e vai crescendo de 2 em 2 até que o número de asteriscos impressos seja igual a numero.

Pois bem, vamos desenhar linha por linha, imprimindo primeiro o número correto de espaços e depois o de asteriscos.

Esse controle de linhas é feito pela linha de código:

```
for(int linha=1 ; espacos > 0 ; linha++)
```

Ou seja, vamos imprimir da linha 1 até a linha central (note que a linha central é a que tem $\text{espacos}=0$).

Agora, dentro desse looping vamos imprimir, em cada linha, primeiro o número de espaços e depois os asteriscos.

Sabemos que o número inicial de espaços é $(\text{numero}-1)/2$ e o número inicial de asteriscos é 1.

Para imprimir o número correto de espaços e asteriscos vamos usar a variável count:

```
for(int count=1 ; count <= espacos ; count++)
```

```
for(int count=1 ; count <= asteriscos ; count++)
```

Após o looping maior, o das linhas, temos que decrementar o número de espaços em 1, e incrementar o número de asteriscos em 2, além da quebra de linha.

Parte de cima do diamante

A lógica é a mesma da de cima, porém o número de espaços aumenta em 2, e o número de asteriscos é decrementado em 2.

Outra diferença é que vamos imprimir uma linha a menos, pois a linha central já foi impressa. Assim, essas linhas são impressas desde a primeira abaixo da linha central, até enquanto houver asteriscos (até ter 1, que é o último):

```
for(int linha=1 ; asteriscos > 0 ; linha++)
```

Logo, nosso código Java fica assim:

```
import java.util.Scanner;
```

```
public class DesafioDoDiamante {

    public static void main(String[] args) {

        int numero,

            espacos,

            asteriscos;

        Scanner entrada = new Scanner(System.in);

        System.out.print("Insira um número ímpar: ");

        numero = entrada.nextInt();

        if(numero%2 != 0){

            //Imprimindo a parte de cima do diamante

            asteriscos = 1;

            espacos = (numero-1)/2;

            for(int linha = 1 ; espacos > 0 ; linha++){
```



```
//Espaços
```

```
for(int count = 1 ; count <= espacos ; count++){
```

```
System.out.print(" ");
```

```
}
```

```
//Asteriscos
```

```
for(int count = 1 ; count <= asteriscos ; count++){
```

```
System.out.print("*");
```

```
}
```

```
espacos--;
```

```
asteriscos += 2;
```

```
System.out.println();
```

```
}
```

```
//Imprimindo a parte de baixo do diamante
```

```
for(int linha=1 ; asteriscos > 0 ; linha++){
```

```
    //Espaços
```

```
    for(int count = 1 ; count <= espacos ; count++){
```

```
        System.out.print(" ");
```

```
    }
```

```
    //Asteriscos
```

```
    for(int count = 1 ; count <= asteriscos ; count++){
```

```
        System.out.print("*");
```

```
    }
```

```
    espacos++;
```

```
    asteriscos -= 2;
```

```
    System.out.println();
```

```
    }

    }else{

System.out.println("Não é ímpar!");

    }

}

}
```

Lembrando que essa é apenas uma solução, é possível fazer usando menos linhas, porém o entendimento é mais complicado. Tente!

Apostila de Java, Capítulo 3 - Variáveis primitivas e Controle de Fluxo - Tipos primitivos e valores

Nesse artigo iremos comentar e resolver os exercícios propostos no capítulo 3, sobre Variáveis primitivas e Controle de Fluxo - Tipos primitivos e valores da apostila FJ-11: Java e Orientação a Objetos, da Caelum.

[Clique aqui para saber sobre a Caelum e sua apostila.](#)

[Clique aqui para baixar a apostila.](#)

Recomendamos que tentem ao máximo resolver os exercícios, e só quando conseguir (ou depois de MUITO tentar) veja o código. Caso tenha dificuldades, veja a Solução, que conterà uma explicação a respeito do código, e tente criar seu código.

Programação é criatividade, logo existem mais de uma solução para o mesmo exercício.

Página 30, Exercício 3.3: Variáveis e Tipos primitivos

Enunciados

QUESTÃO 1:

Na empresa onde trabalhamos, há tabelas com o quanto foi gasto em cada mês. Para fechar o balanço do primeiro trimestre, precisamos somar o gasto total. Sabendo que, em Janeiro, foram gastos 15000 reais,

em Fevereiro, 23000 reais e em Março, 17000 reais, faça um programa que calcule e imprima o gasto total

- a) Crie uma classe chamada BalancoTrimestral com um bloco main, como nos exemplos anteriores;
- b) Dentro do main (o miolo do programa), declare uma variável inteira chamada gastosJaneiro e inicialize-a com 15000;
- c) Crie também as variáveis gastosFevereiro e gastosMarco, inicializando-as com 23000 e 17000, respectivamente, utilize uma linha para cada declaração;
- d) Crie uma variável chamada gastosTrimestre e inicialize-a com a soma das outras 3 variáveis:

```
int gastosTrimestre = gastosJaneiro + gastosFevereiro + gastosMarco;
```

- e) Imprima a variável gastosTrimestre.

QUESTÃO 2:

Adicione código (sem alterar as linhas que já existem) na classe anterior para imprimir a média mensal

de gasto, criando uma variável `mediaMensal` junto com uma mensagem. Para isso, concatene a String

com o valor, usando `"Valor da média mensal = "+ mediaMensal`.

Soluções

QUESTÃO 1:

Esse exemplo é bem simples e não há muito o que explicar, o passo-a-passo do enunciado é bem claro.

Mostramos uma maneira diferente de declarar várias variáveis do mesmo tipo (no caso, inteiro), bastante organizada e muito utilizada por programadores.

Ao invés de:

```
int variavel1;
```

```
int variavel2;
```

Fizemos:

```
int variavel1,
```

```
variavel2;
```

```
public class BalancoTrimestral {
```

```
public static void main(String[] args) {
```



```
gastosFevereiro = 23000,
```

```
gastosMarco = 17000;
```

```
int gastosTrimestre = gastosJaneiros + gastosFevereiro + gastosMarco;
```

```
System.out.println(gastosTrimestre);
```

```
int mediaMensal = gastosTrimestre/3;
```

```
System.out.println("Valor da média mensal = "+ mediaMensal);
```

```
}
```

```
}
```

Página 40, Exercício 3.13: Fixação de Sintaxe

Enunciados

QUESTÃO 1:

Imprima todos os números de 150 a 300.

QUESTÃO 2:

Imprima a soma de 1 até 1000.

QUESTÃO 3:

Imprima todos os múltiplos de 3, entre 1 e 100.

QUESTÃO 4:

Imprima os fatoriais de 1 a 10.

O fatorial de um número n é $n * n-1 * n-2 \dots$ até $n = 1$. Lembre-se de utilizar os parênteses.

O fatorial de 0 é 1

O fatorial de 1 é $(0!) * 1 = 1$

O fatorial de 2 é $(1!) * 2 = 2$

O fatorial de 3 é $(2!) * 3 = 6$

O fatorial de 4 é $(3!) * 4 = 24$

Faça um for que inicie uma variável n (número) como 1 e fatorial (resultado) como 1 e varia n de 1 até 10:

```
int fatorial = 1;

for (int n = 1; n <= 10; n++) {

}
```

QUESTÃO 5:

No código do exercício anterior, aumente a quantidade de números que terão os fatoriais impressos, até 20, 30, 40. Em um determinado momento, além desse cálculo demorar, vai começar a mostrar respostas completamente erradas. Por quê? Mude de int para long, e você poderá ver alguma mudança.

QUESTÃO 6:

Imprima os primeiros números da série de Fibonacci até passar de 100. A série de Fibonacci é a seguinte: 0,1,1,2,3,5,8,13,21, etc... Para calculá-la, o primeiro elemento vale 0, o segundo vale 1, daí por diante, o n-ésimo elemento vale o (n-1)-ésimo elemento somado ao (n-2)-ésimo elemento (ex: 8= 5 + 3).

QUESTÃO 7:

Escreva um programa que, dada uma variável x (com valor 180, por exemplo), temos um novo

x de acordo com a seguinte regra:

- se x é par, $x = x / 2$

- se x é ímpar, $x = 3 * x + 1$
- imprime x
- O programa deve parar quando x tiver o valor igual a 1. Por exemplo, para $x = 13$, a saída será:

40 -> 20 -> 10 -> 5 -> 16 -> 8 -> 4 -> 2 -> 1

QUESTÃO 8:

Imprima a seguinte tabela, usando fors encadeados:

1

2 4

3 6 9

4 8 12 16

n $n*2$ $n*3$ $n*n$

[Clique aqui para aprender sobre o laço for em Java.](#)

[Clique aqui para aprender sobre o laço while em Java.](#)

Soluções

QUESTÃO 1:

O que fizemos foi usar a variável 'numero' iniciar em 150 e terminar em 300.

Dentro do for fazemos ela começar em 150, terminar em 300 e ser incrementada em 1, cada iteração.

No laço while, a idéia é semelhante. Porém a variável tem que ser inicializada antes do laço, e ser incrementada dentro do próprio laço while.

Usando o laço for:

```
public class Caelum1 {
```

```
public static void main(String[] args) {
```

```
int numero;
```

```
for(numero = 150 ; numero <= 300 ; numero++){
```

```
System.out.println(numero);
```

```
}
```

```
}
```

```
}
```

Usando o laço while:

```
public class Caelum1 {
```

```
public static void main(String[] args) {
```

```
int numero=150;
```

```
while(numero<=300){
```

```
System.out.println(numero);
```

```
    numero++;
```

```
}
```

```
}
```



```
}
```

QUESTÃO 2:

Definimos duas variáveis inteiras, a 'numero' que irá percorrer do número 1 até 1000. A variável 'soma' é inicializada em 0 e servirá para armazenar o valor de todos os números, de 1 até 1000.

Isso acontece no código: soma = soma + numero;

Ou seja, o valor da variável 'soma' é seu antigo valor somado com o valor da variável 'numero'.

Usando o laço for:

```
public class Caelum2 {
```

```
public static void main(String[] args) {
```

```
int numero,
```

```
    soma=0;
```

```
for(numero = 1 ; numero <= 1000 ; numero++){
```

```
    soma = soma + numero;
```

```
}
```

```
System.out.println("Soma: "+soma);
```

```
}
```

```
}
```

Usando o laço while:

```
public class Caelum2 {
```

```
public static void main(String[] args) {
```

```
int numero=1,
```

```
    soma=0;
```

```
while(numero<=1000){
```

```
    soma = soma + numero;
```

```
    numero++;
```

```
}
```

```
System.out.println("Soma: "+soma);
```

```
}
```

```
}
```

QUESTÃO 3:

Assim como nos exemplos anteriores, vamos usar a variável inteira 'numero' para percorrer do número 1 até 100. Porém, não vamos imprimir todos os números, somente os que são divisíveis por 3.

Para tal, vamos usar o teste condicional if else, que fará com que somente os números que deixam resto 0 na divisão por 3: `numero % 3 == 0`

[Clique aqui para estudar o símbolo % \(módulo ou resto da divisão\).](#)

Assim, colocamos o print dentro dos colchetes do if, para que somente os número divisíveis por 3 que sejam impressos na tela.

Note que no laço while, o incremento (`numero++`) é feito fora do teste condicional if, pois a variável deve crescer sempre, não somente 'numero' for divisível por 3.

Usando o laço for:

```
public class Caelum3 {
```

```
public static void main(String[] args) {
```

```
int numero;
```

```
for(numero = 1 ; numero <= 100 ; numero++){
```

```
if( numero%3 == 0){
```

```
System.out.println(numero);
```

```
    }
```

```
}
```

```
}
```

```
}
```

Usando o laço while:

```
public class Caelum3 {
```

```
public static void main(String[] args) {
```

```
int numero=1;
```

```
while(numero <= 100){  
  
    if( numero%3 == 0){  
  
        System.out.println(numero);  
  
        }  
  
        numero++;  
  
    }  
  
}  
  
}
```

QUESTÃO 4:

Essa questão é bem interessante! O segredo dela é dado no próprio enunciado.

Por exemplo, $5! = 5 \times 4 \times 3 \times 2 \times 1$, ok?

Se você é iniciante e tentou, provavelmente fez assim. Porém isso não é necessário.

Porque calcular tudo isso se você, ao calcular '5!' já possui o valor de '4!'?

Inicialmente fazemos `fatorial=1`

Essa variável, como o próprio nome diz, irá armazenar o valor dos fatoriais de 1 até 10.

A chave de tudo é o: $\text{fatorial} = \text{fatorial} * n$

Essa linha diz: o novo valor de 'fatorial' é o valor antigo dessa variável - que é $(\text{fatorial}-1)!$ - multiplicado por um número 'n', onde esse n vai de 1 até 2.

Ora, inicialmente fazemos $\text{fatorial}=1$.

Então a primeira iteração é: $\text{fatorial} = (1!) * 1 = 1$

Depois, para $n=2$: $\text{fatorial} = (1!) * 2 = 2!$

Para $n=3$: $\text{fatorial} = (2!) * 3 = 3!$

Para $n=4$: $\text{fatorial} = (3!) * 4 = 4!$

...

Para $n=10$: $\text{fatorial} = (9!) * 10 = 10!$

```
public class CaelumFatorial {  
  
public static void main(String[] args) {  
  
int n,  
  
    fatorial=1;  
  
    for(n=1 ; n<=10 ; n++){  
  
        fatorial = fatorial * n;
```

```
System.out.println("O fatorial de "+ n + " é (" +(n-1)+"!) * "+n+" = "+fatorial);

    }

}

}
```

QUESTÃO 5:

Fatorial é um cálculo um pouco trabalhoso para o Java, pois é um número que cresce assustadoramente.

[Segundo a documentação oficial, a variável do tipo int pode variar de -2.147.483.648 até o máximo 2.147.483.647](#)

Note que:

$12! = 479.001.600$

$13! = 6.227.020.800$

Ou seja, até $n=12$, tudo ok. Passou disso, o Java imprime valores errados, pois extrapola o máximo que a variável inteira pode suportar.

Já o tipo long pode armazenar inteiros de tamanho até

9.223.372.036.854.775.807

Ou seja, suporte até $20! = 2.432.902.008.176.640.000$

QUESTÃO 6:

Vamos chamar o n -ésimo termo da série de 'numero', o $(n-1)$ -ésimo termo de 'ultimo' e o $(n-2)$ -ésimo de 'penultimo'.

Assim, temos a fórmula geral: $\text{numero} = \text{ultimo} + \text{penultimo}$;

Até aqui, sem segredos. Imprimimos o 'numero' e tudo ok.

O problema é na hora de descobrir o próximo número.

Para calcularmos o próximo termo da série ('numero'), temos que mudar os valores de 'ultimo' e 'penultimo'.

Para tal, o valor de 'penultimo' agora será o valor de 'ultimo', e o valor de 'ultimo' será agora o valor de 'numero'. Após fazer isso, podemos calcular o novo valor de 'numero', que é $\text{ultimo} + \text{penultimo}$.

Por exemplo, chegamos em um ponto que:

$\text{numero} = 21$

$\text{ultimo} = 13$

$\text{penultimo} = 8$

Para calcular o próximo termo, fazemos ' $\text{penultimo} = \text{ultimo}$ ', ou seja, agora:

penultimo = 13

Analogamente, fazemos 'ultimo=numero', e temos:

ultimo=21

De posse desses novos valores, temos o novo valor de 'numero', que é: $21 + 13 = 34$, que é o próximo termo da série.

```
public class caelumFibonacci {
```

```
public static void main(String[] args) {
```

```
int numero,
```

```
    ultimo=1,
```

```
    penultimo=0;
```

```
    numero = ultimo + penultimo;
```

```
while(numero<=100){
```

```
    System.out.println(numero);
```

```
    penultimo=ultimo;

    ultimo=numero;

    numero = ultimo + penultimo;

}

}

}
```

QUESTÃO 7:

A primeira coisa a se fazer é declarar o inteiro 'x' e inicializá-lo com algum valor (13, por exemplo).

Após isso, o aplicativo entra um loop através do laço while, que só termina quando x for 1.

Fazemos isso assim: while(x !=1)

Isso quer dizer: "enquanto x não for 1, continue nesse looping".

A cada iteração, o valor de x irá mudar.

Se x for par, ele mudará seu valor para 'x/2', e caso seja ímpar mudará seu valor para '3*x + 1'.

Para controlar como ocorrerão essas mudanças de valores, vamos usar o teste condicional if else.

Vamos checar se x é par através do seguinte código: if (x % 2 == 0)

Ou seja, se x deixa resto 0 na divisão de por 2, é porque x é par.

Se isso for verdade, x assumirá a metade de seu valor: $x = x/2$;

Mas ora, um número só poderá ser par ou ímpar. Se o if for falso, é porque o número é ímpar e o que vai ocorrer nesse caso será o que está dentro do bloco do else : $x = 3*x + 1$

Após cada mudança, apenas imprimimos o valor de x.

```
public class caelum7 {
```

```
public static void main(String[] args) {
```

```
int x=13;
```

```
while(x != 1){
```

```
if( x%2== 0){
```

```
    x = x/2;
```

```
    } else {
```

```
        x = 3*x +1;
```

```
    }
```

```
System.out.print(" -> "+x);
```

```
    }  
  }  
}
```

QUESTÃO 8:

Como é pedido, vamos usar dois laços for para resolver essa questão. O primeiro laço é para controlar quantas linhas existem no nosso resultado final. Assim, se $n=5$, existirá 5 linhas.

O segundo laço vai imprimir o resultado de cada linha, suas colunas.

Assim, se $n=5$, o resultado é: $5*1$ $5*2$ $5*3$ $5*4$ $5*5$

Ou seja, a linha é a 5, e as colunas são 1, 2, 3, 4, e 5 (sempre variam de 1 até o valor da linha).

O grande segredo de tudo é revelado na última linha do enunciado: $n*1$ $n*2$ $n*3$... $n*n$

Essa é a fórmula geral. Onde 'n' é a linha atual, e em cada linha é impresso o produto de 'n' por um número que varia de 1 até o próprio valor da linha, o 'n'. Construímos isso com os laços for encadeados:

```
for(linha=1 ; linha <= n ; linha++){  
  for(coluna=1 ; coluna <= linha ; coluna++){  
    System.out.print(linha*coluna + " ");  
  }  
  System.out.println();  
}
```

```
}
```

Após cada linha ser expressa (ou seja, após cada iteração do primeiro laço, temos que imprimir uma quebra de linha, por isso o println ali no final.

```
public class Caelum8 {
```

```
public static void main(String[] args) {
```

```
int n=5,
```

```
    linha,
```

```
    coluna;
```

```
for(linha=1 ; linha <= n ; linha++){
```

```
for(coluna=1 ; coluna <= linha ; coluna++){
```

```
System.out.print(linha*coluna + " ");
```

```
    }
```

```
System.out.println();
```

}

}

}

Página 41, Exercício 3.14 - Desafios: Fibonacci

QUESTÃO 1:

Faça o exercício da série de Fibonacci usando apenas duas variáveis.

Solução:

Essa questão é um pouco mais complicada, vai precisar de conhecimentos em Programação e um pouco de Matemática, mas não deixa de ser simples e engenhosa.

Na seção passada usamos três variáveis: 'numero', 'ultimo' e 'penultimo', para descrever o n -ésimo, $(n-1)$ -ésimo e o $(n-2)$ -ésimo termo, respectivamente.

Vamos excluir a variável 'numero', e trabalhar somente com o 'ultimo' e 'penultimo'.

Vamos supor que em um determinado ponto temos:

ultimo=5

penultimo=3

É fácil ver que o próximo termo é sempre (ultimo + penultimo), que nesse caso é 8.

Nosso objetivo então é fazer com que as variáveis tenham o valor ultimo=8 e

penultimo=5, concorda?

Para que consigamos gerar mais números de Fibonacci, fazemos com que 'ultimo' receba seu antigo valor somado com o valor de 'penultimo', pois assim a variável 'ultimo' terá a soma dos dois números anteriores, como diz a regra de Fibonacci. Em Java, fica assim:

```
ultimo = ultimo + penultimo
```

Agora ultimo = 8, e penultimo = 3.

Nosso objetivo agora é fazer com que penultimo receba o valor 5. Mas como, se nenhuma das variáveis possui valor 5? De onde obter esse valor?

De uma relação matemática da série de Fibonacci! Note que sempre: ultimo = numero - penultimo

Veja como obter 5: $5 = 8 - 3$, ou seja, é o novo valor de 'ultimo' subtraído do atual valor de 'penultimo'.

Em Java fica assim:

```
penultimo = ultimo - penultimo
```

Note no desenho como variam os valores das variáveis no decorrer da execução:

0

1

1

2

3

5

8

13

21

penultimo

ultimo

inicialmente

penultimo

ultimo

Após fazer: $\text{ultimo} = \text{ultimo} + \text{penultimo}$

penultimo

ultimo

Após fazer: $\text{penultimo} = \text{ultimo} - \text{penultimo}$

Veja como fica o resultado:

```
public class Caelum_Desafio_Fibonacci {
```

```
public static void main(String[] args) {
```

```
int ultimo=1,
```

```
    penultimo=0;
```

```
while( (ultimo + penultimo) <=100){
```

```
System.out.println(ultimo + penultimo);
```

```
    ultimo = ultimo + penultimo;
```

```
    penultimo = ultimo - penultimo;
```

```
}
```

```
}
```

```
}
```

Orientação a Objetos, parte I: Criando e declarando classes - Construtores

Introdução: O que são e para que servem as Classes e Objetos

O conceito de orientação a objetos, é, sem dúvida, o mais importante em Java.

E é por isso que ensinaremos desde o início, em nosso curso de Java.

Por ser um conceito um pouco abstrato, é normal que demore um pouco até entender tudo. E até entender bem, vai demorar mais ainda.

Porém, vamos ensinar, ao longo de vários tutoriais, divididos em várias seções, estes tão importantes detalhes.

Nesta parte, como só vimos o básico e laços da programação Java, vamos apenas ensinar como declarar as classes, instanciar os objetos (com e sem construtores) e como usar testes condicionais dentro das classes que criamos.

Isso dará uma idéia do que são classes e objetos, e é importante que se habitue a estas idéias.

O que são Classes e Objetos em Java

Como havíamos comentado em nosso artigo com a explicações sobre o primeiro programa que criamos, classes podem ser vistas como abstrações ou definições maiores das coisas e objeto já é algo mais real, mais concreto, é um elemento ou tipo daquela classe.

Usando essas definições, é realmente difícil entender os conceitos, por isso vamos usar exemplos.

Por exemplo, podemos ver "Pessoa" como uma classe. É uma classe que representa seres humanos, que possuem cabeça, coração, cérebro etc. É uma generalização maior.

Podemos declarar você, caro leitor, como um objeto dessa classe, com seu nome e características específicas de uma "Pessoa". Você pertence a classe "Pessoa". Eu também, pois possuímos cabeça, coração, cérebro etc.

Nós temos essas características gerais, que todos da classe "Pessoa" possuem. Ou seja, em Java, dizemos que somos instâncias da classe "Pessoa".

Utilidade das Classes e Objetos em Java

Esse tipo de representação vai nos ajudar muito em programação Java.

Por exemplo, imagine que você foi contratado para criar um aplicativo para uma empresa - em Java, claro.

Você tem que cadastrar os milhares de funcionários da empresa.

É claro que você não vai declarar milhares de strings para armazenar o nome de cada um, nem de inteiros para armazenar seus números nem de floats para armazenar seus salários.

Seria humanamente impossível fazer isso.

Agrupar coisas semelhantes

Aí que entra a vantagem da programação orientada a objetos. Podemos ver todos esses funcionários de uma mesma maneira: como a classe Funcionario.

O que a classe "Funcionario" tem em comum?

Tem um nome, uma idade, uma data de contratação, um salário, um setor em que trabalham e outras coisas específicas da empresa.

Pronto.

Você pode ver essa classe como um tipo de dado.

Assim como 'int' ou 'float', agora existe o tipo 'Funcionario'. Toda vez que entrar alguém novo na empresa, você declara esse elemento como fazendo parte do tipo 'Funcionario'. Ou seja, estará criando um objeto dessa classe.

O objeto, diferente da classe, é algo mais específico, ele que terá as informações pessoais de cada funcionário.

Crie uma vez, use quantas vezes quiser

A grande vantagem desse tipo de 'visão', é que, ao declarar a classe, você declara dentro dela os tipos: string, float, int etc, que estão dentro da classe.

Então, quando for criar um objeto, automaticamente esses dados estarão criados!

Aí que reside a beleza do Java e da orientação a objetos. É muito, mas muito útil e prático. É um novo jeito de pensar e ver o mundo. Dizemos que é um tipo de paradigma de programação diferente.

Altere uma parte do código, e a mudança se propagará em todo o código

Ok, você criou seu aplicativo usando a classe "Funcionario".

Porém, a pessoa que te contratou - que muitas vezes não são da área de TI - esqueceu de te informar que os funcionários devem ter uma informação no cadastro: se possuem carro ou não.

E aí? Alterar tudo? Começar do zero?

Claro que não. Simplesmente vá na classe e coloque esse atributo (informação), e automaticamente todos os objetos passarão a ter esse dado, "carro".

Então é só pedir para os funcionários preencherem esse dado no seu aplicativo de cadastro.

Classe à parte, vida à parte

Uma coisa interessante que a programação orientada a objetos nos proporciona é a divisão das partes do programa. Dois programadores podem programar duas classes de forma totalmente independente e fazer com que elas funcionem perfeitamente.

Coisa que em outros paradigmas de programação é quase impossível.

Por exemplo, você criou a classe "Funcionario".

Nessa classe você precisa a informação do salário de cada funcionário, porém você não tem acesso aos detalhes financeiros da empresa. Ora, nem precisa e nem deve ter, é algo mais restrito.

Outro programador, responsável pelo setor financeiro, pode ter criado a classe "Salario" que recebe os dados de cada pessoa, sua posição na empresa, bônus, horas extras etc etc, e te dá somente o resultado final: o número. Esse dado é o que você vai usar na sua classe "Funcionario".

Isso todo pode ser feito de uma maneira totalmente eficiente, segura e independente, tanto por você como pelo programador que fez a outra classe.

A única troca de informação é que você pega um float dele e ele pega o nome do

seu funcionário ou código dele da empresa.

O mais importante disso é: em nenhum momento foi necessário um ver o código do outro!

Na hora de trabalhar, isso nos diz muito em termos de rendimento!

Como saber quando usar Classes e Objetos em Java

Como você pode notar, através dos exemplos dos Carros, das Pessoas e dos Funcionários, as Classes são nada mais que um grupo de informações. Sempre que quiser usar essas informações, declare um Objeto daquela classe.

Ou seja, sempre que quiser generalizar ou criar um grupo com características parecidas, não tenha dúvida, use Classe e Objetos.

Em um jogo, Worms Armageddon ou Counter-Strike, por exemplo. Existem vários jogadores. Ora, eles são parecidos, tem semelhanças e características em comum. Não perca tempo declarando seus atributos individualmente, use classes e objetos.

Na verdade, em Java, tudo são classes e objetos, então não há muito essa preocupação.

Nosso programa principal, que contém a 'main()', é uma classe. Para receber dados, usamos a classe Scanner.

[Em C, não existe Orientação a Objetos.](#)

[Em C++, você escolhe se usa ou não.](#)

Adiante, veremos que uma classe pode se parecer com uma e outra, pegar 'emprestado' detalhe de outras, implementar ou estender outras.

Existem vários recursos para serem usados nas Classes, inclusive de segurança, que permitem que alguns objetos tenham acesso a alguns dados, outros não.

Recursos gráficos: por exemplo, as janelas dos aplicativos Java, botões, menu e tudo mais são classes.

Como dissemos, Java gira em torno de orientação a objetos, e Java é uma linguagem de programação riquíssima e ilimitada. Logo, o que é possível fazer com as classes e objetos também é bem vasto.

Como criar uma Classe e Declarar Objetos

Agora que você já leu no artigo passado sobre a utilidade das classes e objetos em Java, vamos ensinar nesse tutorial como criar uma classe em sua IDE e como instanciar os objetos dessa classe.

Para isso, vamos usar o conceito de Orientação a Objeto, criando uma classe chamada Aluno, que usaremos em nosso próximo artigo para conter o nome e notas de alunos, bem como preencher e acessar os dados desses objetos.

Criando uma classe em Java

Sem mais delongas, crie seu projeto de Java.

Eu chamei o meu de PrimeiraClasse.

Ao criar esse, note que se chamará PrimeiraClasse.java

Nesse tutorial vamos criar outra classe, a classe "Aluno", que o Java irá chamar de "Aluno.java".

Para isso, se estiver no NetBeans vá em File -> New File

Em Categories, escolha Java Class, e Next.

Dê o nome de Aluno para sua nova classe.

Pronto. Sua nova classe está criada.

Note que apareceu outra aba, ao lado da "PrimeiraClasse.java", e se chama "Aluno.java"

Nesse novo arquivo, deverá ver algo como:

```
public class Aluno{
```

```
}
```

Que é nossa classe, e que, ainda, está em branco.

Criada a classe, vamos aprender como criar os objetos dessa classe.

Declarando um objeto de uma classe em Java

Se lembrar bem, você já declarou objetos de classe em nosso curso de Java.

Mais especificamente, declarou o objeto 'entrada', da classe 'Scanner', várias vezes.

Agora, vamos fazer o mesmo com nossa classe Aluno.

Vamos criar um objeto, um aluno da classe "Aluno".

A sintaxe é (digite isso na aba da PrimeiraClasse.java):

```
Aluno donaFifi = new Aluno();
```

```
Aluno patropi = new Aluno();
```

```
Aluno programador = new Aluno();
```

Pronto, criamos três objetos do tipo Aluno.

Poderíamos ter feito:

```
Aluno donaFifi;
```

```
donaFifi = new Aluno();
```

O `new Aluno()` é o que cria o objeto. E atribuímos ele à variável `donaFifi`.

Essa variável, porém, não é um objeto. Ela contém uma referência ao objeto. É como se ela apontasse, como se tivesse o endereço da localização do objeto. Então, sempre que mudamos essa variável, estamos mudando diretamente o objeto.

Porém, fica muito chato dizer 'declare uma referência ao objeto `Aluno`'.

No dia-a-dia, simplesmente dizemos: criamos um objeto do tipo `Aluno`.

Mas na verdade essas variáveis não são objetos, e sim referências a objetos. O importante é saber disso.

Qualquer coisa, métodos ou dados que colocarmos na classe "`Aluno`", fará parte dos objetos "`donaFifi`", "`patropi`" e "`programador`".

No próximo tutorial você verá bem isso.

Por exemplo, se quisermos adicionar o campo para armazenar o nome completo deles, criamos uma string dentro da classe "`Aluno`", assim, todos os objetos terão essa string.

Qual a vantagem disso?

Ora, criamos esse campo somente uma vez! Na classe! E todos os objetos herdaram isso!

Imagine numa situação real! Em uma escola, com milhares de alunos. Você declara a string uma vez, e ela passa a fazer parte da ficha de cadastro de milhares de alunos.

Muito útil esse Java e a Orientação a Objetos, não?

Acessando e modificando variáveis de Classes e Objetos

Visto para que servem os objetos e classe em Java, como declarar, criar uma classe e instanciar um objeto, vamos agora, nesta seção, colocar algumas variáveis dentro de nossa classe e aprender como atribuir valores para elas, bem como acessá-las.

Este tutorial é o primeiro passo para criarmos atributos (características) em nossas classes.

Variáveis de métodos e Variáveis de Classes

A diferença das variáveis declaradas nas classes daquelas declaradas no método 'main' ou em outros métodos (que você aprenderá a criar numa seção futura do curso), é que nos métodos elas são locais e temporárias.

Ou seja, as variáveis locais método podem ser acessadas somente dentro daquele método, e quando este acaba, seu valor é perdido.

Quando declaramos um objeto de uma classe, suas variáveis passam a existir a partir daquele momento até enquanto o objeto existir.

Quanto ao seu acesso, o Java provém artifícios interessantíssimos de segurança. Podemos declarar variáveis de uma classe como 'public' (que podem ser acessados de qualquer lugar do programa), 'private' (só elementos da própria classe podem acessar esses dados) e 'protected' (só elementos da própria classe e subclasses que podem ver essa variável).

Veremos mais isso em Encapsulamento, numa seção futura do nosso curso online de Java, quando falarmos mais sobre Orientação a Objetos. Por hora, saiba apenas que existe e para que servem - segurança e organização.

Criando classes com atributos

Voltando a nossa classe "Aluno", vamos criar 3 variáveis: "Nome", que vai receber o nome do aluno e "notaMat" e "notaFis" que vão receber a nota de matemática e física do aluno.

A nossa classe ficará:

```
public class Aluno {  
  
    public String nome;  
  
    public double notaMat,  
    notaFis;  
  
}
```

Sim, simplesmente isso.

Declaramos as variáveis como 'public' porque vamos acessá-las de outra classe, a classe "PrimeiraClasse", que contém o método 'main'.

Quando estudarmos Métodos, mais adiante em nosso curso online de Java, veremos como declarar esses atributos de forma 'private', de modo que eles fiquem inacessíveis à outras classes. E como vamos acessar esses dados então? Através de métodos dessa classe.

Assim podemos controlar como essas variáveis serão usadas.

Alterando e Acessando atributos de uma classe

Variáveis criadas, vamos dar valor a elas. Vamos usar a classe Scanner para perguntar ao usuário o nome do aluno e suas notas.

Note que, se estas variáveis estivesse na classe principal, que contém o método main, simplesmente atribuiríamos valores da seguinte forma:

```
notaMat = [numero]
```

Porém, essas variáveis não estão nessa classe. Estão em outra.

Então temos que acessar através do nome do objeto.

No meu caso, declarei o objeto Aluno com o nome "patropi", então para acessar seus elementos, simplesmente vamos usar:

```
patropi.notaMat
```

Ou seja: nomeDoObjeto.nomeDaVariavel;

Então, nossa classe principal, que instância um objeto, preenche o nome e duas notas do Aluno, depois mostra esses valores e a média será:

PrimeiraClasse.java


```
import java.util.Scanner;
```

```
public class PrimeiraClasse {
```

```
public static void main(String[] args) {
```

```
    Aluno patropi = new Aluno();
```

```
    Scanner entrada = new Scanner (System.in);
```

```
    System.out.print("Nome do aluno: ");
```

```
    patropi.nome = entrada.nextLine();
```

```
    System.out.print("Nota em Matemática: ");
```

```
    patropi.notaMat = entrada.nextDouble();
```

```
    System.out.print("Nota em Física: ");
```

```
    patropi.notaFis = entrada.nextDouble();
```

```
    //Exibindo os dados
```

```
    System.out.printf("O aluno \"%s\" tirou %2.2f em Matemática, "
```

```
        + "%2.2f em Física e obteve média %2.2f\n", patropi.nome,
```

```
        patropi.notaMat,  
        patropi.notaFis,  
        (patropi.notaMat+patropi.notaFis)/2);  
    }  
}
```

Aluno.java

```
public class Aluno {  
  
public String nome;  
  
public double notaMat,  
        notaFis;  
  
}
```

Dentro do printf usamos \" para exibir aspas duplas.

Construtor padrão e com parâmetros: o que são, para que servem e como usar

Você reparou que, no tutorial passado, sobre como declarar e criar classe e instanciar objetos, nós fizemos:

```
Aluno fulano = new Aluno();
```

Você reparou como repetimos 'Aluno' duas vezes?

Por que? Não é inútil ou perda de tempo repetir isso?

Em Java - aliás, em programação - nada é à toa (se for, logo eles mudam). Tudo tem um motivo.

Nesse artigo você irá aprender sobre os Construtores (constructors) e saberá o motivo dessa repetição e como usufruir e ganhar muito tempo a partir deles

O que são construtores/constructor em Java

Quando usamos a keyword 'new' para criar um objeto de uma determinada classe estamos alocando um espaço na memória. Ao fazer isso, o Java requer que algumas variáveis sejam iniciadas com algum valor.

Esse ato de inicializar, ou construir, é feito pelos construtores, que são métodos - que iremos estudar mais a fundo em nossa apostila de Java logo mais.

O Java automaticamente inicializa as variáveis globais quando criamos um objeto de uma classe.

No artigo passado de nossa apostila, fizemos um exemplo com a classe "Aluno", onde criamos três variáveis: "nome", "notaMat" e "notaFis", que são consideradas variáveis globais da classe, pois não fazem parte de nenhum método.

Logo, elas devem ser inicializadas pelo Java. Para você ver como isso realmente acontece, vá na sua classe principal, que contém a 'main', e após a criação do objeto "patropi" da classe "aluno" adicione as seguintes linhas de código, de print, para você ver como essas variáveis da classe já possuem um valor, sem nem mesmo você ter atribuído nada:

```
Aluno patropi = new Aluno();
```

```
System.out.println(patropi.nome);
```

```
System.out.println(patropi.notaMat);
```

```
System.out.println(patropi.notaFis);
```

O resultado é:

null

0.0

0.0

Ou seja, por padrão, as strings são iniciadas com valores 'null' (nulos) e valores zeros para números.

Isso é o que ocorre automaticamente, quando nenhum construtor é especificado. Vamos mostrar como especificar construtores.

O que é e como utilizar o construtor padrão em Classes

Construtor é um método, em Java.

Embora não tenhamos estudado métodos ainda, diremos como utilizar. É bem simples, e em breve você aprenderá tudo sobre método em nosso curso de Java.

Métodos são um trecho de código que fazem uma coisa específica, em programação. Se já estudou C ou C++, lá eles são chamados de funções ou de rotinas ou sub-rotinas em outras linguagens, como Perl.

Podemos criar vários construtores. O padrão é aquele que não recebe nenhum parâmetro.

Em termos mais simples, é aquele que não recebe nenhuma informação. Ele sempre vai ser executado quando você criar um objeto.

Para criar um construtor, crie um método com o mesmo nome da classe.

Métodos tem a seguinte sintaxe:

```
nomeDoMetodo( tipoDoParametro nomeDosParametros) {  
  
    // código dos métodos  
  
}
```

Como nossa classe se chama "Aluno", nosso método é um construtor dessa classe e ela é padrão (não recebe parâmetros) e é 'public', ficará assim:

```
public Aluno(){  
  
}
```

Para melhor visualizar a utilidade do construtor padrão, vamos adicionar um print:

```
public Aluno(){  
  
    System.out.println("Objeto criado");  
  
}
```

Ao rodar seu programa, você verá a mensagem "Objeto criado" sendo exibida logo após ter criado o objeto "patropi".

Esse método é chamado construtor default. Ao contrário do que muitos pensam, default não é padrão (embora usemos muito como se fosse).

Default é omissão. Quando estamos omitindo outros construtores - que recebem parâmetro - é esse construtor 'padrão' (default) que será chamado.

Note que ao criar o objeto, usamos "new Aluno();" - logo, não passamos nenhum parâmetro. Então, chamamos claramente o construtor padrão.

Criando um construtor que recebe parâmetros

No nosso aplicativo que cadastro um aluno, seu nome e duas notas, estamos atribuindo os valores dessas variáveis direto no objeto:

```
patropi.nome = entrada.nextLine();  
  
patropi.notaMat = entrada.nextDouble();  
  
patropi.notaFis = entrada.nextDouble();
```

O que vamos fazer agora, para ilustrar como funciona os construtores que recebem parâmetros, é pedir esses dados antes e iniciar o objeto "patropi" com esses dados.

Por exemplo, vamos criar as variáveis "Nome", "NotaMat", "NotaFis" na 'main' e receber esses dados pela Scanner:

```
Nome = entrada.nextLine();  
  
NotaMat = entrada.nextDouble();  
  
NotaFis = entrada.nextDouble();
```

Agora vamos criar nosso construtor na classe "Aluno" que recebe esses três parâmetros: uma string e dois doubles.

Ele vai ficar assim:

```
public Aluno(String Nome, double NotaMat, double NotaFis){
```



```
nome=Nome;  
  
notaMat=NotaMat;  
  
notaFis=NotaFis;  
  
}
```

Isso pode parecer um pouco estranho ou complicado agora, mas quando estudar melhor métodos, fará todo o sentido do mundo e verá como é simples.

O que esse método faz é receber uma string "Nome", dois doubles "NotaMat" e "NotaFis" e atribuir eles aos valores da classe aluno.

Veja que nossa classe tem uma variável string "nome" e dois doubles "notaMat" e "notaFis", que seriam inicializados com 'null' e '0.0', conforme mostramos no início deste tutorial.

O que estamos fazendo com esse método é inicializar estas variáveis com outros valores! E que valores são estes? Ora, são os valores das variáveis "Nome" e "NotaMat" e "NotaFis", que definimos a partir da classe Scanner no método 'main'.

Embora estejamos engatinhando na orientação a objeto, vamos fazer uma coisa interessante e mostrar o quão poderosa e útil essa técnica é. Vamos adicionar na classe uma variável, um double de nome "media", e no construtor vamos adicionar uma linha de código:

```
media = (notaMat + notaFis)/2;
```

Ou seja, nosso construtor vai ficar:

```
public Aluno(String Nome, double NotaMat, double NotaFis){  
    nome=Nome;  
    notaMat=NotaMat;  
    notaFis=NotaFis;  
  
    media = (notaMat + notaFis)/2;  
}
```

O que quer dizer isso?

Que quando iniciarmos nosso objeto com as notas, esse método vai automaticamente calcular a média dessas notas!

Você pode pensar "Mas já fazíamos isso na 'main', durante o print. Não é a mesma coisa?".

Imagine que você vai preencher os dados de 20 alunos.

Então vai colocar 20 vezes essa fórmula? Claro que não.

Aí é que reside a beleza das Classes & Objetos. Você declara essa fórmula uma única vez, e poderá usar ela sempre que criar um objeto!

Criou o objeto, iniciou ele com as variáveis, ele vai calcular automaticamente a média!

Para acessar essa variável basta fazer: `patropi.media`

E ela estará lá, bonitinha para você usar.

Como iniciar Objetos com parâmetros

Agora que criamos nossos métodos construtores padrão e com parâmetros, vamos mostrar como iniciar um objeto com dados.

No construtor padrão, vínhamos fazendo:

```
Aluno patropi = new Aluno();
```

Agora, que já temos as variáveis "Nome", "NotaMat" e "NotaFis" e um construtor que está preparado para receber esses três tipos de dados, podemos criar um objeto e lançar essas variáveis:

```
Aluno patropi = new Aluno( Nome, NotaMat, NotaFis);
```

Note que você só deve criar esse objeto depois de ter essas variáveis com os valores corretos, senão vai iniciar o objeto com valores errados.

Um detalhe importante é que o nome dessas variáveis, "Nome", "NotaMat" e "NotaFis", não precisam ser iguais aos nomes que declaramos lá no construtor da classe "Aluno". Poderíamos passar:

```
Aluno patropi = new Aluno( name, math, phys);
```

Desde que "name" seja uma string e "math" e "phys" doubles. Quando elas 'chegam' no construtor, elas 'chegam' com os nomes "Nome", "NotaMat" e

"NotaFis".

Você se acostumará melhor sobre isso quando estudar métodos.

Classes com mais de um construtor

O que ocorre se deixarmos o construtor: `Aluno()`

e o construtor: `public Aluno(String Nome, double NotaMat, double NotaFis)`
juntos, na classe "Aluno"?

Nada. Aliás, é até recomendável. É um excelente costume.

Se você iniciar um objeto dessa classe sem nenhum parâmetro, o Java inicia o método construtor padrão (que mostra a mensagem "Objeto criado").

Se iniciar um objeto com 3 parâmetros, ele atribui esses valores que você passou aos valores do objeto e ainda calcula a média.

Você pode ainda criar outros métodos, que recebem só o nome do aluno, só a nota de matemática ou só de a física.

Enfim, os construtores são feitos para inicializar os atributos e preparar o objeto para o uso, seja lá qual for o uso que você queira dar. Se você utilizar vários construtores, o Java é inteligente o suficiente para saber qual método utilizar dependendo dos parâmetros que você passar.

Obviamente se criar dois construtores com a mesma lista de parâmetros, terá problemas.

Para mostrar um exemplo do uso de dois construtores - o padrão e o que recebe 3 parâmetros - , vamos criar dois objetos, o "donaFifi", logo no início, e veremos a mensagem "Objeto criado".

Depois pedimos os dados do aluno patropi e inicializamos o objeto "patropi" com esses dados. Em seguida, usamos as variáveis do objeto desse aluno, inclusive sua média, que foi calcula pelo construtor.

Veja como ficou o código:

PrimeiraClasse.java

```
import java.util.Scanner;
```

```
public class PrimeiraClasse {
```

```
public static void main(String[] args) {
```

```
    Aluno donaFifi = new Aluno();
```

```
String nome;
```

```
Double notaMat,
```

```
    notaFis;
```

```
Scanner entrada = new Scanner (System.in);

System.out.print("Nome do aluno: ");

    nome = entrada.nextLine();

System.out.print("Nota em Matemática: ");

    notaMat = entrada.nextDouble();

System.out.print("Nota em Física: ");

    notaFis = entrada.nextDouble();

    Aluno patropi = new Aluno(nome, notaMat, notaFis);

System.out.printf("O aluno \"%s\" tirou %2.2f em Matemática, "
    + "%2.2f em Física e obteve média %2.2f\n", patropi.nome,
        patropi.notaMat,
        patropi.notaFis,
        patropi.media);

}

}
```


Aluno.java

```
public class Aluno {
```

```
public String nome;
```

```
public double notaMat,
```

```
    notaFis,
```

```
    media;
```

```
public Aluno(){
```

```
    System.out.println("Objeto criado");
```

```
}
```

```
public Aluno(String Nome, double NotaMat, double NotaFis){
```

```
    nome=Nome;
```

```
    notaMat=NotaMat;
```

```
    notaFis=NotaFis;
```

```
    media = (notaMat + notaFis)/2;
```

```
}
```

}

Métodos em Java

Nesta série de tutoriais de Java, iremos aprender a usar uma das ferramentas mais importantes, poderosas e flexíveis da linguagem: os métodos.

Métodos nada mais são que trechos de códigos, que fazem algo específico e que podem ser executados em qualquer lugar de seu código, sempre que for necessário, sem existir a necessidade de repetir aqueles comandos em Java.

Vimos como automatizar nossas tarefas, criando loopings que executavam códigos quantas vezes quiséssemos através de laços, como o `for` e o `while`.

Porém, uma vez que esses laços terminavam, eles ficavam para trás.

Com os métodos não, os métodos podem rodar sempre, de qualquer lugar.

Outra vantagem dos métodos, é a reutilização de código.

Por exemplo, você pode criar um método que faz uma soma, outro que calcula o quadrado de um número, outro que calcula o delta da equação do segundo grau, e qualquer método com funções em específicas.

No futuro, quando quiser esses códigos para fazer uma calculadora, por exemplo, eles já estarão prontos na forma de métodos, e você não terá que programar tudo de novo.

Não, basta chamar os métodos, a qualquer instante, e quando quiser, sem ser necessário repetir uma linha de código.

Aprenda bem o que são métodos, faça os seus com objetivos bem simples e definidos, e o principal: guarde e organize seus métodos, você irá usá-los pelo resto de sua vida como profissional de Java.

Métodos: Introdução, o que são, para que servem, como e quando usar os methods

Para ganhar uma seção na apostila online "Java Progressivo", você já deve suspeitar da importância dos métodos.

Nesse artigo, daremos uma descrição sobre o que são métodos, para que servem, como declarar, como usar e quando usar.



Introdução aos métodos em Java

Até o presente momento, criamos aplicações que seguiram sempre a seqüência padrão do código.

Ou seja, de cima pra baixo.

Com o uso dos laços (while, for, do ... while), testes condicionais if else e os comandos switch, break e continue, passamos a ter um pouco mais de controle e alteramos um pouco o fluxo, ou seja, o jeito que os programas ocorriam.

Porém, eles ainda ocorrem de cima pra baixo.

Métodos nada mais são que um bloco de códigos que podem ser acessados a qualquer momento e em qualquer lugar de nossos programas.

Ou seja, não precisam estar na ordem 'de cima pra baixo' no código.

A utilidade dos métodos em Java

As utilidade são duas:

1. Organização

Tudo que é possível fazer com os métodos, é possível fazer sem.

Porém, os programas em Java ficariam enormes, bagunçados e pior: teríamos que repetir um mesmo trecho de código inúmeras vezes.

Uma das grandes vantagens, senão a maior, que mostraremos a seguir, de se utilizar os métodos em Java é que escrevemos um trecho de código uma vez (que são os ditos cujos métodos) e podemos acessá-los várias vezes.

2. Reusabilidade

Quando você for iniciar um grande projeto, dificilmente você terá que fazer tudo do zero.

Se fizer, muito provavelmente é porque você se organizou muito mal ou não fez o Curso Java Progressivo.

Aqui vai uma das dicas mais importantes do curso Java Progressivo:

Crie métodos que façam coisas específicas e bem definidas.

Isso será útil para reusabilidade.

O que queremos dizer com métodos específicos?



Em vez de criar um método que constrói uma casa, crie um método que cria um quarto, outro método que cria a sala, outro que cria o banheiro, um método pra organizar a casa etc.

Assim, quando tiver que criar um banheiro, já terá o método específico para aquilo.

Veja os métodos como peças de um quebra-cabeça. Porém, são peças 'coringa', que se encaixam com muitas outras peças.

Reusabilidade é tempo. E tempo é dinheiro.

Por exemplo, no estudo deste curso, você vai criar um método que retorna as raízes de uma equação do segundo grau.

Guarde esse método com um nome que você lembre.

Em trabalhos futuros, você precisará desse método. E aí, vai sempre digitar tudo de novo?

Claro que não! Se organize! Se lembre que já fez esse método, onde guardou, vá lá, copie e coloque em seu novo projeto.

Como declarar métodos em Java

Há várias, mas várias maneiras mesmo de se declarar um método. Alguns detalhes só iremos explicar melhor quando você souber Orientação a Objetos.

A sintaxe 'geral' dos métodos é a seguinte:

```
[características do método] nome_do_metodo (tipos parâmetros) {  
    // código  
    // do seu  
    // método  
}
```

Usando/chamando métodos

Vamos criar um método que simplesmente imprime a mensagem na tela: "Curso Java Progressivo".

Vamos chamar esse método de 'mensagem'.

Ele será um método público (public) e não irá retornar nada (void).

Um método que calcula o delta da equação do segundo grau, retorna o valor do delta, ou seja, um float.

No nosso caso, o método vai apenas imprimir uma mensagem na tela.

O 'static', assim como o 'public' e o 'void' serão explicados no decorrer da apostila.

Ele ficará assim:

```
public static void mensagem(){  
    System.out.println("Curso Java Progressivo!");  
}
```

Colocamos esse método dentro de alguma classe. No nosso caso, dentro da

classe que tem o método 'main'.

Pronto. Código escrito, método declarado. Como usar o método?

Chamando! Como se chama algo? Pelo nome, ué!

Basta escrever 'mensagem();' em seu programa, onde quiser, que o método será chamado e executado.

Teste:

```
public class metodoTeste {
```

```
public static void mensagem(){
```

```
System.out.println("Curso Java Progressivo!");
```

```
}
```

```
public static void main(String[] args) {
```

```
System.out.print("Exibindo a mensagem uma vez: ");
```

```
    mensagem();
```

```
System.out.println("Exibindo a mensagem 3 vezes:");
```

```
for(int count=1 ; count<=3 ; count++){  
    mensagem();  
}  
}  
}
```

Note que nossa classe agora tem dois métodos.

Um é o método 'main', que serve pra inicia os aplicativos Java. O outro exibe uma mensagem.

Já aqui notamos um detalhe importante: podemos, e com frequência, invocamos um método dentro de outro.

No caso, estávamos na 'main()', quando invocamos o 'mensagem();'.

Note outro detalhe também: em vez de escrever várias vezes:

```
System.out.println("Curso Java Progressivo!");
```

Escrevemos só uma vez, no método, e chamamos o método. Mais prático, não?

Esse Java!

Mostraremos, no próximo exemplo, como criar um menu interativo, usando os métodos.

Você vai ver que com os `methods` as coisas ficam mais organizadas, economiza-se linhas de códigos e fica mais fácil de entender quando se olha os códigos. Além da questão da reusabilidade do código, já que você poderá sempre usar esses métodos em outros aplicativos.

Aplicativo: menu simples usando métodos, laços e o comando switch usando um método para exibir um menu de opções

PROGRAMMING MENU

Focando o objetivo de nosso curso, que é fazer e mostrar coisas úteis, vamos mostrar uma utilidade de métodos que simplesmente exibem mensagens na tela.

Programa em Java: Como criar um menu

Vamos usar o exercício 9, sobre saídas simples ([Clique aqui para ver a questão](#)).

Basicamente, vamos usar nossos conhecimentos em laços (do ... while), o comando switch (para escolher entre as opções do menu) e 5 métodos.

Um método para cada opção (inclui, altera, exclui, consulta) e outro método que mostra esse menu de opções.

Como as opções são números, usaremos um tipo inteiro, chamado 'opcao', para receber as opções do usuário.

Como queremos que o menu seja exibido ao menos uma vez, usamos o do...while.

Ao entrar nesse laço, o menu é exibido com o comando 'menu();' que chama o método que exibe o menu.

Logo após, o programa espera a entrada do usuário. Dependendo do que foi digitado, o método específico - inclui(), altera(), exclui() ou consulta() - é selecionado pelo switch.

O programa só termina se o usuário digitar 0.

```
import java.util.Scanner;
```

```
public class menu {
```

```
public static void menu(){
```

```
System.out.println("\tCadastro de clientes");
```

```
System.out.println("0. Fim");
```

```
System.out.println("1. Inclui");
```

```
System.out.println("2. Altera");
```

```
System.out.println("3. Exclui");
```

```
System.out.println("4. Consulta");
```

```
System.out.println("Opcao:");
```

```
}
```

```
public static void inclui(){
```

```
System.out.println("Você entrou no método Inclui.");
```

```
}
```

```
public static void altera(){
```

```
System.out.println("Você entrou no método Altera.");
```

```
}
```

```
public static void exclui(){
```

```
System.out.println("Você entrou no método Exclui.");
```

```
}
```

```
public static void consulta(){
```

```
System.out.println("Você entrou no método Consulta.");
```

```
}
```

```
public static void main(String[] args) {  
    int opcao;  
    Scanner entrada = new Scanner(System.in);  
  
    do{  
        menu();  
        opcao = entrada.nextInt();  
  
        switch(opcao){  
  
            case 1:  
                inclui();  
  
            break;  
  
            case 2:  
                altera();  
  
            break;
```

case 3:

exclui();

break;

case 4:

consulta();

break;

default:

System.out.println("Opção inválida.");

}

} while(opcao != 0);

}

}

O comando RETURN: obtendo informações dos métodos

RETURN
TO WORK



Nesta seção do curso Java Progressivo iremos mostrar para que serve e como usar o comando `return` nos métodos.

Return: Retornando informações úteis em Java

Embora seja possível não retornar nada dos métodos (como simplesmente mostrar uma mensagem ou um menu de opções, como fizemos no artigo passado), o mais comum é que os métodos retornem algo.

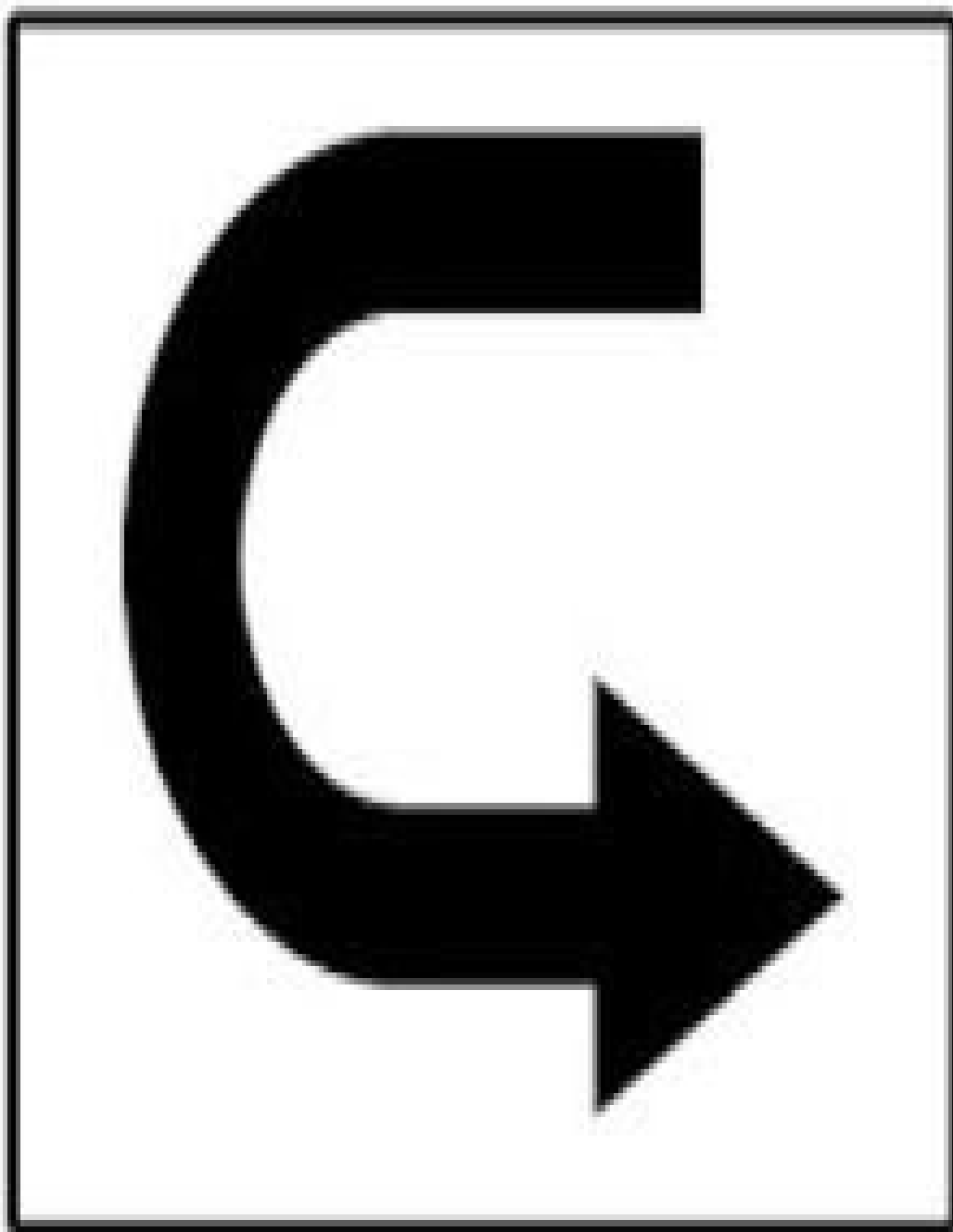
Como assim 'retornar'?

Retornar no sentido de resultado. Por exemplo, uma soma.

O método pode ser simplesmente um trecho de código que calcula a soma de dois números e retorna esse resultado.

Ou pode ser um método que recebe os coeficientes de uma equação do segundo grau e retorna suas raízes.

O retorno pode ser um inteiro, um float, uma string, pode ser simplesmente uma decisão (como um boolean) ou outra coisa mais elaborada que você desejar.



RETURNS

Mas como eu disse, e mais uma vez vou repetir, crie métodos simples e os mais específicos possíveis.

Não crie métodos que fazem mil coisas. Ao invés disso, crie mil métodos, onde cada um faz uma coisa bem determinada. Isso é importante para você se organizar, não se perder e poder reutilizar seus códigos no futuro.

Retornando inteiros, floats, doubles... em Java

No exemplo do artigo passado onde criamos um método que simplesmente mostrava uma mensagem na tela, ele não retornava nada. Isso poderia ser visto por um detalhe na declaração do método, o 'void'.

Para retornar um inteiro, usaremos 'int'.

Por exemplo, vamos criar um método que retornar o valor de '1+1'.

Para fazer isto basta escrever 'return' e o que quisermos retornar após. Veja:

```
public static int soma(){  
  
    return 1+1;  
  
}
```

E agora, como usamos um retorno de inteiro?

É o mesmo que perguntar 'o que podemos fazer com um inteiro?'

Já que o método retorna um inteiro, você pode ver esse method como um inteiro.

Ou seja, você pode imprimir, somar com outro inteiro, com um float, ou simplesmente atribuir a um tipo inteiro.

Veja:

```
public class returnTest {
```

```
public static int soma(){
```

```
return 1+1;
```

```
}
```

```
public static void main(String[] args) {
```

```
System.out.print("Declarando a variável 'res_soma' e recebendo o método  
soma(): ");
```

```
    int res_soma=soma();
```

```
System.out.println(res_soma);
```

```
System.out.println("Imprimindo diretamente o resultado do return: " + soma());
```

```
System.out.println("Usando em uma soma: 2 + soma() = " + (2 + soma()));
```



```
System.out.println("Usando em um produto: 3 * soma() = " + (3 * soma()));  
  
    }  
  
}
```

Se quisermos retornar um float: return 1.1*3.3

Devemos declarar o método como 'public static float soma()' ou obteremos um erro.

Retornando uma String em Java

Vamos refazer o programa do artigo passado. Porém, ao invés de imprimir mensagem na tela, ele vai retornar a string:

```
public class returnString {
```

```
    public static String mensagem(){
```

```
        return "Curso Java Progressivo!";
```

```
    }
```

```
    public static void main(String[] args) {
```

```
        System.out.println("Exibindo a mensagem uma vez: "+ mensagem());
```

```
        System.out.println("Exibindo a mensagem 3 vezes:");
```

```
        for(int count=1 ; count<=3 ; count++){
```

```
System.out.println(mensagem());  
  
    }  
  
}  
  
}
```

Mais uma vez, você poderia ter declarado uma variável do tipo String para receber o 'return'.

Note que, dentro do método, seria a mesma coisa se tivéssemos feito:

```
public static String mensagem(){  
  
String mensagem;  
  
    mensagem="Curso Java Progressivo!";  
  
return mensagem;  
  
}
```

Retornando boolean em Java

Se você é um brasileiro e defensor ferrenho da língua portuguesa, pode definir os seguintes métodos:

```
public static boolean verdade(){
```

```
    return true;
```

```
}
```

```
public static boolean falso(){
```

```
    return false;
```

```
}
```

E pronto. Agora poderá usar

if (verdade()) ao invés de if(true)

ou *if (falso())* no lugar de *if(false)*

:)

Até o momento nossos métodos não são flexíveis, pois sempre imprimem e retornam a mesma coisa.

Isso vai mudar quando você aprender como os métodos são utilizados em programas reais, através de argumentos e parâmetros, que será o assunto do próximo artigo de nossa apostila Java Progressivo.

Parâmetros e Argumentos: passando informações para os métodos

Em nosso curso de Java, até o momento, os nossos métodos faziam coisas bem específicas, como exibir uma string, um menu ou realizavam cálculos matemáticos.

Isso, na prática, não é muito útil.

Diferença entre parâmetro e Argumento em Java

Parâmetro e argumentos são os meios na quais nós passaremos dados para o método, e estes métodos irão trabalhar especificamente em cima dessas informações que nós demos.

E, geralmente, vão nos retornar algum resultado.

Declarando métodos com parâmetros

Sempre que um method receber dados, estes serão recebidos através dos parâmetros.

Em Java, assim como em C ou C++, você precisa deixar bem claro quais e qual o tipo dos dados que você vai passar.

Em algumas linguagens, como Perl, isso não é necessário.

Isso tem a ver com tipagem, alocação de memória e outras coisas de mais baixo nível.

(Quando falamos em baixo nível, em Java ou computação, estamos no referindo ao nível de abstração. Quanto mais baixo mais próximo do hardware ou dos bits estamos falando).

A sintaxe da declaração fica assim:

```
[dados_do_método] tipo_de_retorno nome_do_método (tipo  
nome_do_parametro1, tipo nome_do_parametro2){
```

```
    //código
```

```
    // do
```

```
    // method
```

```
}
```


Uma função que recebe um inteiro e retorna um inteiro teria a seguinte 'cara':

```
public int metodo(int numero){
```

```
    //código
```

```
    return intNumber;
```

```
}
```

Esse tipo, no caso 'int', que vem logo antes do nome do método se refere ao tipo do retorno.

Se seu method retorna float, deverá ser: `public float metodo(...)`

Se for String, deverá ser: `public static String metodo(...)`

O static será explicado mais adiante em nosso curso Java Progressivo, na seção de Orientação a Objetos da apostila online.

Até então estamos usando pois estamos chamando os métodos a partir da main, que é um static também.

Exemplo: Passando argumentos para um método - Função que calcula o quadrado de um número

Pela declaração, vemos que o seguinte método recebe um inteiro, o 'num' e retorna um inteiro também.

Dentro de métodos podemos declarar tipos. No caso, declaramos o tipo 'quadrado', que recebe o valor de 'num' elevado ao quadrado, ou seja: $\text{num} * \text{num}$ e retorna esse 'quadrado'.

```
public static int quadrado(int num){  
    int quadrado;  
    quadrado = num * num;  
  
return quadrado;  
  
}
```

Para usarmos um método que recebe um inteiro como parâmetro precisamos passar um argumento ao método. Por argumento, entenda um valor, algo real.

No programa, a chamada do método seria a seguinte: `quadrado(numero);`

Nos exemplos passados não colocávamos nada entre parênteses, pois os métodos que criamos nos artigos passado de nosso curso não tinham parâmetro. Agora, temos que passar um número, ficaria algo do tipo: quadrado(2), quadrado(3) etc.

Veja como ficaria esse método em um aplicativo Java que pede ao usuário um número inteiro e retorna o quadrado desse valor:

```
import java.util.Scanner;
```

```
public class quadrado {
```

```
public static int quadrado(int num){
```

```
    int quadrado;
```

```
    quadrado = num * num;
```

```
return quadrado;
```

```
}
```

```
public static void main(String[] args) {
```

```
    int numero, numero_quadrado;
```

```
    Scanner entrada = new Scanner(System.in);
```

```
System.out.print("Entre com um inteiro: ");

    numero = entrada.nextInt();

    numero_quadrado=quadrado(numero);

System.out.printf("%d elevado ao quadrado é %d", numero, numero_quadrado);

    }

}
```

Note que, na 'main', declaramos o valor 'numero', fizemos com que ele receba um valor inteiro e passamos esse valor inteiro ao método.

Você pode achar estranho o fato de usarmos 'numero' na 'main' e 'num' no método 'quadrado()'. Porém, é assim mesmo.

Na 'main', o número que o usuário forneceu fica armazenado na variável 'numero', mas quando passamos este valor ao método, estamos passando um número.

Este número, agora, será armazenado na variável 'num' do método 'quadrado()'.

Chamamos essa variável, 'num', de variável local, pois ela só existe dentro do método. Se você tentar usar ela fora do método, obterá um erro.

Teste, adicione a linha seguinte ao programa:

```
System.out.println(num)
```

Você nem vai conseguir compilar, pois para a 'main', não existe 'num'.

Declaramos dentro do method 'quadrado()' a variável 'quadrado' somente para efeitos didáticos, para mostrar que podemos declarar variáveis dentro de um método, mas à rigor, ela não é necessária, poderíamos simplesmente ter feito:

```
public static int quadrado(int num){
```

```
    return num * num;
```

```
    }
```

Nesse exemplo, 'num' seria um parâmetro e o valor de 'numero' seria o argumento.

Exemplo: passando uma lista de parâmetros para um método - Cálculo do IMC

No artigo sobre operações Matemáticas em Java, passamos como exercício o cálculo do IMC (índice de massa corporal). Que nada mais é um número, que é calculado pelo peso (em kilogramas) dividido pelo quadrado da altura (em metros) da pessoa.

O nosso método IMC recebe dois valores float, o peso e a altura, e retorna outro float, que é o valor do IMC. A declaração de um método com mais de um parâmetro ficaria assim:

```
public static float IMC(float peso, float altura)
```

Nosso programa ficaria assim:

```
import java.util.Scanner;
```

```
public class IMC {
```

```
public static float IMC(float peso, float altura){
```

```
    float imc;
```

```
    imc = peso/(altura*altura);
```

```
return imc;
```

```
}
```

```
public static void main(String[] args) {
```

```
    float peso, altura, imc;
```

```
    Scanner entrada = new Scanner(System.in);
```

```
    System.out.print("Entre com seu peso, em kilos: ");
```

```
    peso = entrada.nextFloat();
```

```
    System.out.print("Entre com sua altura, em metros: ");
```

```
    altura = entrada.nextFloat();
```

```
    imc = IMC(peso, altura);
```

```
    System.out.printf("Seu IMC vale: %.2f",imc);
```

```
}
```

```
}
```

Note que agora invocamos o método IMC e passamos dois argumentos: peso e altura

Note que na 'main', usei exatamente as mesmas variáveis que usei dentro do method IMC(), fiz isso pra mostrar que não há problemas em fazer isso.

Porém, quando usamos essas variáveis dentro da 'main', os valores serão os que o usuário passou. Dentro do método, esses valores podem ser alterados e poderão representar outros valores.

Cuidado para não confundir. Embora tenham o mesmo nome, estamos usando em lugares diferentes de nosso aplicativo Java.

Exemplo: chamando um método dentro do outro - Cálculo do IMC

Lembra que falei várias vezes sobre organização, fazer métodos simples e diretos?

Pois é, vamos mostrar, na prática, como fazer isso.

Notou que no cálculo do IMC elevamos um número ao quadrado, a altura?

Notou que o primeiro método calcula um número ao quadrado?

Você deve estar pensando:

'Ora, vamos usar o primeiro exemplo para fazer um cálculo dentro do segundo.'

Quase. Note que o primeiro exemplo usamos inteiros! Mas a altura, que usamos no segundo exemplo desse artigo, se refere a float!

Então vamos modificar um pouco nossa função que calcula o quadrado de um número, para receber e retornar decimais e usar dentro do método IMC();

Os métodos ficariam assim:

```
public static float quadrado(float num){
```

```
return num*num;
```

```
}
```

```
public static float IMC(float peso, float altura){
```

```
    float imc;
```

```
    imc = peso/quadrado(altura);
```

```
return imc;
```

```
}
```

E o aplicativo Java ficaria assim:

```
import java.util.Scanner;
```

```
public class IMC {
```

```
public static void main(String[] args) {
```

```
    float peso, altura;
```

```
    Scanner entrada = new Scanner(System.in);
```

```
    System.out.print("Entre com seu peso, em kilos: ");
```

```
    peso = entrada.nextFloat();
```

```
System.out.print("Entre com sua altura, em metros: ");  
  
    altura = entrada.nextFloat();  
  
System.out.printf("Seu IMC vale: %.2f", IMC(peso, altura));  
  
}
```

```
public static float IMC(float peso, float altura){
```

```
    return peso/quadrado(altura);
```

```
    }
```

```
public static float quadrado(float num){
```

```
    return num*num;
```

```
    }
```

```
}
```

Pronto, usamos um método - quadrado() - dentro de outro, IMC().

Note que a ordem em que declaramos os métodos dentro de uma classe não

importa.

Classe Math: constantes, principais métodos e chamando métodos de outras classes

Vamos estudar uma importante classe, a Class Math, que provém várias funcionalidades matemáticas que nos ajudarão bastante em nosso curso de Java.

Aliado a essa grande e poderosa classe com a nossa seção de Métodos, vamos aprender como usar os methods que estão declarados em uma classe diferente daquela que estamos fazendo a chamada.

Usando métodos de outras classes em Java

Até o presente momento em nossa apostila online Java Progressivo, chamamos os métodos da seguinte maneira:

```
nome_do_metodo(argumentos);
```

Fizemos isso porque definimos esses métodos dentro da mesma classe (até o momento só fizemos aplicativos Java contendo uma classe. Na próxima seção estudaremos classes).

Porém, quando o método não está presente na classe em que estamos fazendo a chamada, a sintaxe é diferente, ela é a seguinte:

```
Nome_da_classe.Nome_do_metodo(argumentos);
```

Vale salientar que a declaração é feita dessa maneira pois esses métodos são estáticos (static). Ou seja, não foi necessário criar um objeto da classe Math (diferente da classe Scanner, por exemplo).

Na próxima seção do curso, sobre Classes, você entenderá mais sobre static.

A Classe Math (Class Math) do Java

Para ilustrar o uso de métodos de outras classes e, de quebra, para nos auxiliar nos cálculos de constantes (como do número pi, do número de euler), no cálculo de funções trigonométricas (senos, cossenos, tangentes etc) e outras funcionalidades, vamos apresentar e usar a classe Math.

Essa classe já está na package (pacote) java.lang. Ou seja, não precisamos importar, que nem fazemos com a classe Scanner.

Constantes da classe Math:

Vamos imprimir o valor de Pi e da constante de euler, o 'e' dos números exponenciais.

Esses valores estão armazenados nas constantes PI e E, da classe Math. As constantes de outras classes são acessadas da mesma maneira que os métodos, ou seja:

Math.PI

Math.E

Vamos imprimir esses valores:

```
public class constantes {
```

```
public static void main(String[] args){
```

```
System.out.println("O valor de pi é: " + Math.PI);
```

```
System.out.println("O valor de E é: " + Math.E);
```

```
}
```

```
}
```


Exponencial e potenciação na classe Math

Para calcular valores do tipo: e^x

usamos o método: `exp()` , que recebe um `double` e retorna um `double`.

```
numero = Math.exp(argumento)
```

Para calcular qualquer tipo de potências, da forma: a^b

onde `a` e `b` são do tipo `double`, usamos o método `pow()` da classe `Math`

```
numero = Math.pow(a,b)
```

Veja:

```
public class Mathtest {
```

```
public static void main(String[] args){
```

```
System.out.println("'e' elevado ao quadrado = "+ Math.exp(2));
```

```
System.out.println("2 elevado ao cubo = " + Math.pow(2, 3));
```

```
}  
  
}
```

Lembre-se que estes métodos retornam double.

Caso você tenha declarado um float e queira receber o resultado no sua variável float, use o cast.

Por exemplo, no caso dos nossos methods para o cálculo de IMC, o 'quadrado' retorna um float, então fazemos:

```
(float)Math.pow(altura,2);
```

Assim, o método Math.pow() irá retorna um float, ao invés do double.

Veja como ficariam os nossos métodos:

```
public static float IMC(float peso, float altura){
```

```
    return peso/(float)Math.pow(altura, 2);
```

```
    }
```

```
public static float quadrado(float num){
```

```
    return Math.pow(num,2);
```

```
}
```

Na verdade, nem precisaríamos mais do método `quadrado()`, pois o `pow()` já faz isso.

Calculando a Raiz quadrada em Java através da classe Math

Para calcular a raiz quadrada de um número positivo, usamos o método `sqrt()`, de square root (raiz quadrada em inglês).

Que recebe e retorna um double.

Por exemplo, um programa que mostra a raiz quadrada de PI:

```
public class RaizDePi {
```

```
public static void main(String[] args){
```

```
System.out.println("A raiz quadrada de Pi é = "+ Math.sqrt( Math.PI ) );
```

```
}
```

```
}
```

Calculando logaritmos naturais em Java através da classe Math

Calcula logaritmos naturais (ou seja, de base 'e') através do método: `log()` que recebe e retorna um `double`.

Por exemplo, um programa que mostra o logaritmo de natural de 10 e do número 'e':

```
public class logaritmos {
```

```
public static void main(String[] args){
```

```
System.out.println("O logaritmo natural de 10 é = "+ Math.log(10) );
```

```
System.out.println("O logaritmo natural de 'e' é = "+ Math.log( Math.E ) );
```

```
}
```

```
}
```

Calculando senos, cossenos, tangentes e outras funções trigonométricas em Java através da classe Math

As principais funções trigonométricas são seno, cosseno e tangente, e são calculadas através dos métodos:

`Math.sin()`

`Math.cos()`

`Math.tan()`

Que recebem e retornam valores do tipo `double`. Porém, os valores que estas funções recebem devem ser em **RADIANOS**!

```
public class Trigonometricas {
```

```
    public static void main(String[] args){
```

```
        System.out.println("O seno de 90 é = "+ Math.sin( (Math.PI)/2 ) );
```

```
        System.out.println("O cosseno de 0 é = "+ Math.cos(0) );
```

```
        System.out.println("A tangente de 45 é= "+ Math.tan( (Math.PI)/4 ));
```

```
    }
```

}

Módulo, máximo, mínimo e arredondamento em Java através da classe Math

Para calcular o módulo de um número 'numero' usamos: `Math.abs(numero)`

Para calcular o valor mínimo de dois números 'num1' e 'num2', usamos:
`Math.min(num1,num2)`

Para calcular o valor máximo de dois números 'num1' e 'num2', usamos:
`Math.max(num1,num2)`

Para arredondar um número 'numero' para cima, usamos: `Math.ceil(numero)`

Para arredondar um número 'numero' para baixo, usamos: `Math.floor(numero)`

Estes métodos, assim como todos os outros, recebem e retornam double. Caso deseje receber a passar outro tipo, use cast conforme foi explicado no método `pow()`.

Mais métodos matemáticos em Java

Acesse a documentação:

<http://docs.oracle.com/javase/6/docs/api/java/lang/Math.html>

Sobrecarga de métodos (method overloading): declarando métodos com o mesmo nome



Em nosso artigo sobre parâmetros e argumentos em Java, mostramos um exemplo sobre um método que recebe um número e retorna o quadrado desse número.

Porém esse método tem uma limitação: ele só recebia inteiros.

E se você quiser criar métodos com float? double?

Faça uma sobrecarga.

Overloading ou sobrecarga em Java

O termo sobrecarga vem do fato de declararmos vários métodos com o mesmo nome, estamos carregando o aplicativo com o 'mesmo' método.

A única diferença entre esses métodos são seus parâmetros e/ou tipo de retorno.

Por exemplo, vamos declarar, num mesmo aplicativo, dois métodos quadrados: um que recebe e retorna inteiros e outro que recebe e retorna double.

```
public static int quadrado(int num){
```

```
    int quadrado;
```

```
    quadrado = num * num;
```

```
return quadrado;
```

```
}
```

```
public static double quadrado(double num){
```

```
    double quadrado;
```

```
    quadrado = num * num;
```

```
return double;
```

```
}
```

Quando declaramos estes dois métodos, estamos fazendo uma sobrecarga de métodos, ou method overloading em Java.

O que acontece, então, quando fazemos o chamado da função? Já que existem duas.

Embora os nomes sejam os mesmos, elas atuam de forma diferentes e ocupam espaços diferentes em memória, pois estão lidando com tipos diferentes de variáveis.

Quando invocamos o método com um inteiro como um argumento, o Java é inteligente o suficiente para invocar corretamente o método que foi declarado com inteiro como parâmetro.

Caso invoquemos o método usando um double como um argumento, o método a ser executado será aquele que foi declarado com o tipo double em seu parâmetro.

Veja:

```
public class Sobrecarga {
```

```
public static int quadrado(int num){
```

```
int quadrado;  
  
quadrado = num * num;
```

```
return quadrado;
```

```
}
```

```
public static double quadrado(double num){
```

```
double quadrado;  
  
quadrado = num * num;
```

```
return quadrado;
```

```
}
```

```
public static void main(String[] args){
```

```
System.out.println("Inteiro 2 ao quadrado: " + quadrado(2));
```

```
System.out.println("Double PI ao quadrado: " + quadrado( Math.PI ));
```

```
}
```

```
}
```

Varargs - passando uma lista de argumentos, de tamanho qualquer, para um método

Até o momento, em nosso curso de Java, sempre especificamos o número exato de argumentos que um método pode receber.

Quando estudarmos Arrays e ArrayLists logo mais veremos que podemos passar uma quantidade qualquer de valores aos métodos.

Há, porém, uma outra maneira de fazer isso em Java, através das reticências: ...

Com o uso desse artifício, podemos passar um, dois, três ou um número qualquer de argumentos para um mesmo método que ele vai saber como tratar esses dados.

Sintaxe:

Na hora de declarar seu método, use as reticências logo após o tipo do parâmetro:

```
tipo_de_retorno nomeDoMétodo( tipo... nomeDoParametro ){  
//código do seu method  
}
```

Pronto.

O Java é esperto o bastante para saber que 'nomeDoParametro' é, na verdade, uma lista de valores.

Exemplo de uso

Código Java: Crie um aplicativo que receba 5 valores do usuário e calcule a média dos 5, dos primeiros 4, 3 e 2 valores inseridos. Use apenas um método que receba uma lista de argumentos de tamanho qualquer.

```
import java.util.Scanner;
```

```
public class medias{
```

```
public static float media(float... valores){
```

```
    float media=0;
```

```
    for(float valor: valores){
```

```
        media +=valor;
```

```
    }
```

```
return media/valores.length;
```

```
}
```

```
public static void main(String[] args){  
  
    float valor1, valor2, valor3, valor4, valor5;  
  
    Scanner entrada = new Scanner(System.in);  
  
    System.out.print("Entre com o valor 1: ");  
  
    valor1 = entrada.nextFloat();  
  
    System.out.print("Entre com o valor 2: ");  
  
    valor2 = entrada.nextFloat();  
  
    System.out.print("Entre com o valor 3: ");  
  
    valor3 = entrada.nextFloat();  
  
    System.out.print("Entre com o valor 4: ");  
  
    valor4 = entrada.nextFloat();  
  
    System.out.print("Entre com o valor 5: ");  
  
    valor5 = entrada.nextFloat();  
  
    System.out.println("A média dos 5 números é:  
"+media(valor1,valor2,valor3,valor4,valor5));  
}
```

```
System.out.println("A média dos 4 primeiros números é:  
"+media(valor1,valor2,valor3,valor4));
```

```
System.out.println("A média dos 3 primeiros números é:  
"+media(valor1,valor2,valor3));
```

```
System.out.println("A média dos 2 primeiros números é:  
"+media(valor1,valor2));
```

```
}
```

```
}
```

Nesse exemplo, usamos um tipo especial de laço for, que usaremos bastante em Arrays.

Mas não se assuste, ele é bem fácil de entender: ele simplesmente percorre todos os elementos de uma lista de variáveis.

No nosso caso, nossa lista de variáveis é 'valores'. Por que lista de valores?

Ora, porque não é um só valor. Podemos mandar um, dois, três...mil valores.

Usando esse tipo de laço for, ele vai percorrer, sempre, TODOS os valores dessa lista de variáveis.

A cada iteração, cada valor dessa lista será atribuído a variável 'valor'.

A lógica do laço é somar cada elemento da lista, ou seja, soma 'valor' à variável 'media'.

Ao final do cálculo, dividimos o valor dessa variável 'media' pelo número de elementos da lista.

Esse número está armazenado em um método chamado length e pode ser acessado através de: `media.length`

Essa é uma propriedade especial dessa lista de valores. Ela já vai com um conjunto de informações. Uma dessas informações é o seu número de elementos.

Você aprenderá mais sobre essas 'listas' de valores quando estudar Arrays.

Sábio, esse Java, não?

Exercício:

Um professor, muito legal, fez 3 provas durante um semestre mas só vai levar em conta as duas notas mais altas para calcular a média.

Faça uma aplicação em Java que peça o valor das 3 notas, mostre como seria a média com essas 3 provas, a média com as 2 notas mais altas, bem como sua nota mais alta e sua nota mais baixa. Essas médias devem ser calculadas usando o mesmo método, pois você é um programador Java e não vai ficar criando métodos à toa.

Crie outro método que receba as 3 notas e retorna a maior delas. E outro que retorna a menor.

Questões envolvendo métodos



Fique à vontade para treinar sua criatividade, usando os laços que quiser (while, for, do ... while), comandos (switch, break e continue), testes condicionais if else ou na raça mesmo ;)

Porém, como estamos na seção de métodos, é importante que treine e use métodos, o comando return, além de parâmetros e argumentos em todas as questões abaixo:

Exercícios sobre métodos em Java

0. Crie um método que receba um valor e informe se ele é positivo ou negativo através de um retorno com boolean.

Declare como: `boolean isPositive(float num)`

1. Crie um método que receba um valor e diga se é nulo ou não.

Declare como: `boolean isZero(float num)`

2. Crie um método que receba três valores, 'a', 'b' e 'c', que são os coeficientes de uma equação do segundo grau e retorne o valor do delta, que é dado por 'b² - 4ac'

3. Usando os 3 métodos acima, crie um aplicativo que calcula as raízes de uma equação do 2o grau:

$$ax^2 + bx + c = 0$$

Para ela existir, o coeficiente 'a' deve ser diferente de zero.

Caso o delta seja maior ou igual a zero, as raízes serão reais. Caso o delta seja negativo, as reais serão complexas

e da forma: $x + iy$

4. Crie um método que receba 2 números e retorne o maior valor.

5. Crie um método que receba 2 números e retorne o menor valor.

6. Crie um método que receba 3 números e retorne o maior valor, use o método `Math.max()`.

7. Crie um método que receba 3 números e retorne o menor valor, use o método `Math.min()`.

8. Crie um método chamado `Dado()` que retorna, através de sorteio, um número de 1 até 6.

9. Use o método da questão passado e lance o dado 1 milhão de vezes. Conte quantas vezes cada número saiu.

A probabilidade deu certo? Ou seja, a porcentagem dos números foi parecida?

10. Crie um aplicativo de conversão entre as temperaturas Celsius e Fahrenheit.

Primeiro o usuário deve escolher se vai entrar com a temperatura em Célsius ou Fahrenheit, depois a conversão escolhida é realizada através de um comando `switch`.

Se C é a temperatura em Célsius e F em fahrenheit, as fórmulas de conversão são:

$$C = 5 \cdot (F - 32) / 9$$

$$F = (9 \cdot C / 5) + 32$$

11. Um professor, muito legal, fez 3 provas durante um semestre mas só vai levar em conta as duas notas mais altas para calcular a média.

Faça uma aplicação em Java que peça o valor das 3 notas, mostre como seria a média com essas 3 provas, a média com as 2 notas mais altas, bem como sua nota mais alta e sua nota mais baixa. Essas médias devem ser calculadas usando o mesmo método, pois você é um programador Java e não vai ficar criando métodos à toa.

Crie um método que receba as 3 notas e retorna a maior delas. E outro que retorna a menor.

Desafio: Ache todos os números primos até 1000

Número primo é aquele que é divisível somente por 1 e por ele mesmo.

Desafio: Escreva um programa em Java que recebe dois inteiros e retorna o MDC, máximo divisor comum.

Desafio: Ache todos os números perfeitos até 1000.

Número perfeito é aquele que é a soma de seus fatores. Por exemplo, 6 é divisível por 1, 2 e 3 ao passo que $6 = 1 + 2 + 3$.

Desafio: Crie um programa em Java que receba um número e imprima ele na ordem inversa.

Ou seja, se recebeu o inteiro 123, deve imprimir o inteiro 321.

Solução 11:

Se está acompanhando nosso curso de Java, já viu como funciona nosso método `média`, que recebe uma lista de argumentos de um número qualquer, em nosso artigo `Passando uma lista de argumentos`, de tamanho qualquer, para um método.

A idéia para descobrir o maior número, entre três números dados, é exatamente a mesma para descobrir o menor número. Vamos apenas trocar o operador `>` pelo `<`.

No método `maiorNota`, queremos descobrir qual o valor maior.

Vamos testar, primeiro, se `'nota1'` é o maior valor. Como fazemos isso?

Ora, vamos comparar com `'nota2'` e `'nota3'`. Se `nota1` for maior que estes dois, é claro que ele é o maior.

Vamos checar isso com dois testes condicionais `if` consecutivos. Caso o resultado seja `true` para os dois, retornamos o `'nota1'` como o maior número e acaba aí o método.

Porém, logo após o primeiro `if`, colocamos um `'{'`.

Pois, caso `'nota1'` não seja maior que `'nota2'`, é porque `'nota2'` é maior (óbvio). Bom, vamos aproveitar que sabemos que `'nota2'` é maior que `'nota1'` e vamos ver se é maior que `'nota3'` também. Pois se for, retornaremos `'nota2'` como o número maior e acaba aí o método.

Caso não seja, chega ao fim os testes condicionais.

E aí? Ora, se nem 'nota1' nem 'nota2' foi maior, é óbvio que 'nota3' é a maior e retornamos ela, e acaba aí o método.

```
import java.util.Scanner;
```

```
public class parametrosVariaveis{
```

```
public static float media(float... valores){
```

```
    float media=0;
```

```
    for(float valor: valores){
```

```
        media +=valor;
```

```
    }
```

```
return media/valores.length;
```

```
    }
```

```
public static float maiorNota(float nota1, float nota2, float nota3){
```

```
    if(nota1 >= nota2){
```

```
        if(nota1 >= nota3)
```

return nota1;

 }else{

if(nota2 >= nota3)

return nota2;

 }

return nota3;

}

public static float menorNota(float nota1, float nota2, float nota3){

if(nota1 <= nota2){

if(nota1 <= nota3)

return nota1;

 }else{

```
if(nota2 <= nota3)
```

```
    return nota2;
```

```
    }
```

```
return nota3;
```

```
}
```

```
public static void main(String[] args){
```

```
    float nota1, nota2, nota3;
```

```
    Scanner entrada = new Scanner(System.in);
```

```
    System.out.print("Entre com o valor 1: ");
```

```
    nota1 = entrada.nextFloat();
```

```
    System.out.print("Entre com o valor 2: ");
```

```
    nota2 = entrada.nextFloat();
```

```
    System.out.print("Entre com o valor 3: ");
```



```
nota3 = entrada.nextFloat();
```

```
System.out.println("Sua maior nota foi: "+maiorNota(nota1,nota2,nota3));
```

```
System.out.println("Sua menor nota foi: "+menorNota(nota1,nota2,nota3));
```

```
System.out.println("Sua média com as três notas é: " +  
media(nota1,nota2,nota3));
```

```
System.out.println("Sua média sem a menor nota é: "+ (nota1+nota2+nota3 -  
menorNota(nota1,nota2,nota3))/2);
```

```
}
```

```
}
```

Jogo: adivinhe o número sorteado pelo computador

Jogar é bom...mas jogar um jogo que você mesmo criou, é um prazer que poucos tem.



A linguagem de programação Java é conhecida por ser bastante usada para criação de jogos. Muito provavelmente você já jogou algum jogo, mesmo que seja aqueles online, feito em Java.

Com base nos conhecimentos obtidos em nosso curso online de Java, já é possível criarmos um jogo.

Embora bem simples, é bem engenhoso e legal:

O computador vai sortear um número, entre 1 e 1000 e você terá que adivinhar que número é esse.

A cada rodada você entra com um número e o computador vai te dar as dicas, dizendo se o número que você digitou é maior ou menor ao número que ele sorteou.

Termina quando você acerta, e ele guarda o número de tentativas que você fez. Consegue bater seu próprio record?

Sabe qual é a melhor estratégia?

PASSO 1: Computador gera um número entre 1 e 1000

Para gerar um número, precisamos importar a classe Random:

```
import java.util.Random;
```

Precisamos criar um objeto do tipo Random para que ele possa gerar os números, vamos chamar de 'geradorDeAleatorios':

```
Random geradorDeAleatorios = new Random();
```

Agora precisamos que esse gerador nos forneça um número aleatório no intervalo de 1000 números. Vamos armazenar esse valor na variável inteira 'sorteado':

```
sorteado = geradorDeAleatorios.nextInt(1000);
```

Porém, dessa maneira, ele vai gerar um número entre 0 e 999. Para ser entre 1 e 1000, somamos 1:

```
sorteado = geradorDeAleatorios.nextInt(1000) + 1;
```

PASSO 2: looping principal do programa

Número gerado!

Agora o usuário vai dar os palpites. Como o usuário vai tentar adivinhar o número pelo menos uma vez, o looping precisa rodar pelo menos uma vez. Portanto, usamos o `do ... while`.

Esse laço ocorre enquanto não adivinharmos o número, ou seja, enquanto nosso palpite for diferente do número sorteado:

```
do{
```

```
// laço
```

```
}while( palpite != sorteado);
```

PASSO 3: tentativa incrementada

A cada tentativa que damos, a variável 'tentativas', que contabiliza nossas tentativas é incrementada em uma unidade:

```
tentativas++;
```

PASSO 4: analisando o palpite

Vamos mandar 3 números para o método dica(): o nosso palpite, o número sorteado pelo computador e o número de tentativas que fizemos.

Para o nosso jogo dar a dica certa, usaremos os testes condicionais if else.

O método testa se o nosso palpite foi maior que o número gerado pelo computador, se for, dá tal dica:

```
if(palpite > numero){
```

```
System.out.println("Seu palpite é maior que o número sorteado.");
```

Case não seja, o palpite será menor ou igual ao número sorteado. Então vai pro else.

Caso o palpite do usuário seja menor que o número sorteado pelo computador, tal dica é exibida:

```
if(palpite < numero){
```

```
System.out.println("Seu palpite é menor que o número sorteado.");
```

Por fim, se o palpite não for maior nem menor que o número sorteado é porque é igual.

Ora, se é igual, você acertou e é exibida a mensagem de acerto e o número de

tentativas realizadas:

```
System.out.println("Parabéns, você acertou! O número era " + numero);
```

```
System.out.println("Você tentou " + tentativas + " vezes antes de acertar!");
```

Quando você acerta, o método dica não é mais chamado, pois 'palpite != sorteado' retornará um valor falso, já que estes números serão iguais (palpite = sorteado), então o laço acaba, acabando o nosso jogo em Java.

Nosso jogo/aplicativo Java ficará assim:

```
import java.util.Random;
```

```
import java.util.Scanner;
```

```
public class JogoAdivinhaNumero {
```

```
public static void dica(int palpite, int numero, int tentativas){
```

```
if(palpite > numero){
```

```
System.out.println("Seu palpite é maior que o número sorteado.");
```

```
    } else {
```

```
if(palpite < numero){
```

```
System.out.println("Seu palpite é menor que o número sorteado.");
```

```
    } else {
```

```
System.out.println("Parabéns, você acertou! O número era " + numero);
```

```
System.out.println("Você tentou " + tentativas + " vezes antes de acertar!");
```

```
    }
```

```
}
```

```
}
```

```
public static void main(String[] args) {
```

```
    int palpite=0,
```

```
        sorteado,
```

```
        tentativas=0;
```

```
    Scanner entrada = new Scanner(System.in);
```

```
Random geradorDeAleatorios = new Random();
```

```
sorteado = geradorDeAleatorios.nextInt(1000) + 1;
```

```
System.out.println("Número entre 1 e 1000 sorteado!");
```

```
do{
```

```
System.out.printf("\n\n\n\n-----\n");
```

```
System.out.println("Número de tentativas: " + tentativas);
```

```
System.out.print("Qual seu palpite: ");
```

```
    palpite = entrada.nextInt();
```

```
    tentativas++;
```

```
    dica(palpite,sorteado, tentativas);
```

```
    }while (palpite != sorteado);
```

```
}
```

```
}
```

Orientação a Objetos, parte II: Os métodos set e get - Composição – Enum

Auto-referência com o this - Invocando métodos de Classes e Objetos

No tutorial de Java passado relacionado a Orientação a Objetos, falamos como criar construtores, com ou sem parâmetros, em nossas classes.

Nesta pequena aula, iremos aprender como referenciar membros de um objeto através da keyword 'this', que é uma ferramenta bastante usada por programadores Java em métodos dentro de Classes.

Referenciando membros da classe com this

Imagine que criemos uma classe chamada "Funcionario", onde seu construtor padrão recebe uma String com o nome do funcionário, um inteiro com seu número de identificação e um double com seu salário.

Se esses dados, dentro da classe são:

```
private String nome;
```

```
private int ID;
```

```
private double salario;
```

E o cabeçalho do construtor é:

```
public Funcionario( String nome, int ID, double salario)
```

Como faríamos a atribuição? Ora, do mesmo jeito que fizemos antes:

```
nome = nome;
```

```
ID = ID;
```

```
salario = salario;
```

Epa! Notou que as variáveis da classe e as variáveis do cabeçalho tem o mesmo nome?

E agora, como o Java vai saber que as variáveis da esquerda se referem as variáveis 'private' da classe a as da direita são as que o usuário mandou pra criar o objeto?

Já sei! Basta criar o método com nomes de variáveis diferentes, como fizemos antes:

```
public Funcionario( String Nome, int id, double Salario){  
  
    nome = Nome;  
  
    ID = id;  
  
    salario = Salario;  
  
}
```

Ok, isso funcionaria perfeitamente. Mas seria extremamente incômodo, e desorganizado, criar dois nomes pra um mesmo tipo de variável.

Não pode parecer problema agora, com essa simples aplicação. Mas em uma situação real, em que seu programa em Java terá centenas de variáveis e você tiver que criar e decorar nomes de variáveis, isso vai ser um baita problema.

Para isso, existe o 'this', que referencia - ou seja, aponta - a própria classe!

'this' em inglês, significa 'isso', 'isto'. É bem fácil seu uso, veja como ficaria nosso construtor:

```
public Funcionario( String nome, int ID, double salario){
```

```
this.nome = nome;  
  
this.ID = ID;  
  
this.salario = salario;  
  
}
```

Pronto.

Agora ficou óbvio que 'this.nome' é a variável 'nome' da classe "Funcionario" e 'nome' é a variável que a classe recebeu para criar um objeto!

Usamos o 'this' dentro da classe. Assim, sempre que colocarmos 'this.' antes de uma variável, fica implícito ao Java que estamos nos referindo aos atributos daquela Classe.

Podemos usar, inclusive, em um print, caso esteja dentro da classe. Em um método, por exemplo, como veremos a seguir.

Outra utilidade do 'this' é passar o objeto atual como parâmetro.

```
public Object getObjeto(){  
  
    return this;  
  
}
```

Outro exemplo disso é criar o método...:


```
public String toString()
```

...na sua classe, e usar 'this' dentro de um print. O Java entenderá que deve ser impresso o que estiver dentro desse método 'toString()'.

Outra utilidade do 'this' é invocar outros construtores. Para invocar um construtor dentro de outro, essa chamada deve ser o primeiro comando do construtor atual.

Por exemplo, fazendo simplesmente:

```
this;
```

Estamos chamando o construtor padrão, que não recebe parâmetros.

Fazendo:

```
this(2112);
```

Estamos invocando o construtor que recebe um inteiro como parâmetro.

Lembrando que quando criamos um construtor que não é o padrão, o Java não vai mais criar o construtor padrão vazio.

Esse construtor padrão vazio só é criado automaticamente quando não criamos nenhum construtor.

Como invocar métodos de objetos que criamos

Vamos criar um método, dentro da classe "Funcionário", que exibe todas as informações de um objeto dessa classe. Vamos chamar de 'exibir':

```
public void exibir(){  
  
    System.out.printf("O funcionário %s, de número %d recebe %.2f por mês",  
        this.nome,this.ID,this.salario);  
  
}
```

Para invocar, basta colocar '.exibir()' após o nome do objeto, que fará com que este método rode.

Note, porém, que conforme explicamos em nosso artigo sobre Classes e Objetos, essa classe é apenas uma abstração.

Ou seja, esse método não existe de verdade! Ele só vai passar a existir quando criarmos um objeto dessa classe!

(Na verdade ele pode existir, caso a classe fosse estática. Estudaremos isso em breve).

Para ilustrar a chamada de métodos de um objeto e o uso do 'this', vamos criar um funcionário - um objeto - de nome 'chefe'.

O código ficará assim:

thisMetodo.java

```
public class thisMetodo{
```

```
public static void main(String[] args){
```

```
String nome = "Neil Peart";
```

```
    int ID=2112;
```

```
    double salario = 1000000;
```

```
    Funcionario chefe = new Funcionario(nome, ID, salario);
```

```
    chefe.exibir();
```

```
}
```

```
}
```

Funcionario.java

```
public class Funcionario {
```

```
private String nome;
```

```
private int ID;
```

```
private double salario;
```

```
public Funcionario(){
```

```
    System.out.println("Método construtor padrão invocado!");
```

```
    }
```

```
public Funcionario( String nome, int ID, double salario){
```

```
    this();
```

```
    System.out.println(this);
```

```
    this.nome = nome;
```

```
this.ID = ID;
```

```
this.salario = salario;
```

```
}
```

```
public String toString(){
```

```
return "Foi usado : System.out.println(this)";
```

```
}
```

```
public void exibir(){
```

```
System.out.printf("O funcionário %s, de número %d recebe %.2f por mês",  
this.nome,this.ID,this.salario);
```

```
}
```

set e get: o que são e como usar esses métodos de forma correta

Vamos aprender nesta aula de nosso curso online de Java os dois métodos mais usados pelos programadores Java: os getters e setters.

O que são e para que servem os métodos get e set

get e set nada mais são que métodos, que freqüentemente vemos em classes de Java.

Eles servem para pegarmos informações de variáveis da classe que são definidas como 'private', porém esses métodos são definidos como 'public'.

Daí surge uma pergunta natural: por que criar métodos para acessar variáveis, se podemos acessar elas diretamente?

Simples: questão de segurança.

As variáveis 'private' só podem ser acessadas de dentro da Classe. É como se elas fossem invisíveis fora do escopo da classe/objeto. Assim, evitamos que outros métodos, classes ou hackers tenham acesso aos dados de determinada classe, que muitas vezes podem ser dados privados, como é caso de aplicações para empresas e bancos.

Como tratar, então, essas variáveis?

Aí que entra a questão dos métodos. Vamos permitir o acesso a essas variáveis, claro (senão não haveria sentido em usar esses atributos).

Porém, vamos ter um total controle sobre essas variáveis através dos métodos.

Por exemplo: suponha que você tenha que criar uma aplicação para um banco, onde vai limitar o número de saques de um cliente.

Os valores do saldo desse cliente ficarão armazenados na variável 'saldo'. Se você der acesso total a essa variável, o cliente poderia usar de forma descontrolada os serviços do banco.

Porém, podemos criar um método em que, cada vez que o cliente saque dinheiro ele ative um contador:

```
public void setSaque(double valor){  
  
    saldo -= valor;  
  
    contador++;  
  
}
```

Nesse trecho de código, quando o usuário tenta sacar, através do menu do caixa eletrônico, ele vai para o método 'saque()'.

Então cada vez que ele saca, seu saldo diminui em 'valor' e um contador (que é uma variável da classe) é incrementado. É impossível fugir desse método!

Sacou? O contador roda! Sempre!

Sabemos que temos um limite diário de saque, vamos supor que seja três. Então vamos limitar o número de saques:

```
public void setSaque(double valor){  
  
    if(contador <=3){  
  
        saldo -= valor;  
  
        contador++;  
  
    }  
  
}
```



```
} else {  
  
System.out.println("Você atingiu o limite de saques diários");  
  
}  
  
}
```

Pronto. Se você sacar 3x, o contador terá valor 4 (supondo que seu valor inicial seja 1), e agora é impossível ele sacar mais, pois sempre vai cair no else.

No próximo artigo criaremos uma aplicação que simula um banco.

Como usar os métodos get e set

Usamos get para obter informações. Esse tipo de método sempre retorna um valor.

Usamos set para definir valores. Esse tipo de método geralmente não retorna valores.

Usando o exemplo da classe "Funcionario", do artigo passado sobre this e uso de métodos em Classe e Objetos, temos o código:

getSet.java

```
public class getSet{
```

```
public static void main(String[] args){
```

```
String nome = "Neil Peart";
```

```
    int ID=2112;
```

```
    double salario = 1000000;
```

```
    Funcionario chefe = new Funcionario();
```

```
        chefe.setNome(nome);  
        chefe.setID(ID);  
        chefe.setSalario(salario);  
  
        chefe.exibir();  
    }  
  
}
```

Funcionario.java

```
public class Funcionario {  
  
    private String nome;  
  
    private int ID;  
  
    private double salario;  
  
    public void exibir(){
```

```
System.out.printf("O funcionário %s, de número %d recebe %.2f por mês",  
getNome(),getID(),getSalario());
```

```
}
```

```
public void setNome( String nome ){
```

```
    this.nome = nome;
```

```
}
```

```
public void setID( int ID ){
```

```
    this.ID = ID;
```

```
}
```

```
public void setSalario( double salario ){
```

```
    this.salario = salario;
```

```
}
```

```
public String getNome(){
```

```
return this.nome;
```

```
}
```

```
public int getID(){
```

```
return this.ID;
```

```
}
```

```
public double getSalario(){
```

```
return this.salario;
```

```
}
```

```
}
```

Como invocar métodos de dentro do construtor

Outra maneira de inicializarmos as variáveis de um Objeto, é usando os métodos 'set', direto do construtor:

setGet.java

```
public class setGet{
```

```
public static void main(String[] args){
```

```
String nome = "Neil Peart";
```

```
int ID=2112;
```

```
double salario = 1000000;
```

```
Funcionario chefe = new Funcionario(nome, ID, salario);
```

```
chefe.exibir();
```

```
}
```

```
}
```

Funcionario.java

```
public class Funcionario {
```

```
private String nome;
```

```
private int ID;
```

```
private double salario;
```

```
public Funcionario( String nome, int ID, double salario){
```

```
    setNome(nome);
```

```
    setID(ID);
```

```
    setSalario(salario);
```

```
}
```

```
public void exhibir(){
```

```
    System.out.printf("O funcionário %s, de número %d recebe %.2f por mês",  
        getNome(),getID(),getSalario());
```

```
}
```

```
public void setNome( String nome ){
```

```
    this.nome = nome;
```

```
}
```

```
public void setID( int ID ){
```

```
    this.ID = ID;
```

```
}
```

```
public void setSalario( double salario ){
```

```
    this.salario = salario;
```

```
}
```



```
public String getNome(){
```

```
    return nome;
```

```
}
```

```
public int getID(){
```

```
    return ID;
```

```
}
```

```
public double getSalario(){
```

```
    return salario;
```

```
}
```

```
}
```

Uso errado do get e set

Qual a diferença entre alterar e pegar o valor de uma variável diretamente (caso ela seja pública), e alterar e pegar o valor dela através de set e get (se ela for privada) ? Não é segurança, já que podemos alterar seu valor como bem quisermos, através dos métodos set.

Definir get e set para todas as variáveis, de forma direta, é um exemplo de péssimo hábito de programação. Infelizmente é uma técnica bem comum e muitos programadores, até os mais experientes, cometem esse ato inútil.

Para que servem então? Quando usar?

Só use set e get quando você for fazer algo além de simplesmente mudar o valor de uma variável 'private'.

Lá no começo de nosso tutorial, usamos o exemplo de contador, que limita o número de saques de um cliente a um caixa eletrônico.

Outro exemplo disso, seria em um aplicativo de vendas.

Ao comprar, o método set ou get automaticamente define o valor de desconto ou juros. Assim esse método faz algo mais além do que mudar a variável. Assim, sua aplicação se torna robusta e segura, pois se torna impossível mudar o valor da variável sem que seja alterada o valor do juros ou desconto.

Sempre que seu método set do tipo for:

```
retorno metodo(tipo valor){  
  
this.nome_da_variavel = valor;  
  
}
```

Muito provavelmente ele é inútil. Para saber quando usar é simples: use quando for fazer algo além de fornecer valor.

Um exemplo util:

```
retorno metodo(tipo valor){  
  
this.nome_da_variavel = valor * juros;  
  
};
```

Vamos treinar!

Exercício: Caixa Eletrônico em Java - Caixa.java

Crie um protótipo de caixa eletrônico em Java. No início, ele pede seu nome e valor \$\$ inicial que tem na conta. O programa deve fornecer um número de 4 dígitos - número da conta - para o usuário (use Random). Esses dados serão usados para criar um objeto da classe "Conta.java"

A seguir, mostra um menu com as opções (esse menu deverá ser um método):

- Extrato: exhibe o nome, número da conta, saldo e quantos saques já foram realizados
- Sacar: recebe o valor a ser sacado, informa se pode ser sacado (não pode ficar negativo) e mostra o saldo
- Depositar: recebe o valor a ser depositado e mostra o novo saldo
- Sair

Esse menu aparece até o usuário escolher sair. As outras opções são métodos que devem fazer parte da "Conta.java" (setters e getters). Note que o usuário pode sacar no máximo 5 vezes por dia.

Aplicativo: Simulação simples de conta bancária

Aplicativo: Conta bancária/Caixa eletrônico simples em Java

Crie um protótipo de caixa eletrônico na linguagem de programação Java. No início, ele pede seu nome e valor \$\$ inicial que tem na conta. O programa deve fornecer um número de até 4 dígitos - número da conta - para o usuário (use Random). Esses dados serão usados para criar um objeto da classe "Conta.java"

A seguir, mostra um menu com as opções (esse menu deverá ser um método):

- Extrato: exibe o nome, número da conta, saldo e quantos saques já foram realizados
- Sacar: recebe o valor a ser sacado, informa se pode ser sacado (não pode ficar negativo) e mostra o saldo
- Depositar: recebe o valor a ser depositado e mostra o novo saldo
- Sair

Esse menu aparece até o usuário escolher sair. As outras opções são métodos que devem fazer parte da "Conta.java" (setters e getters). Note que o usuário pode sacar no máximo 5 vezes por dia.

Nesse exercício não vamos usar explicitamente as palavras 'set' e 'get' nos nomes dos métodos. Mas isso não é necessário. O que vamos usar é sua ideia: a ideia por trás do 'set' é a de alterar valores de variáveis, pra isso vamos usar os métodos 'sacar' e 'depositar', que altera o valor do saldo ; a ideia por trás do 'get' é de simplesmente obter informações das variáveis, como é o caso do método 'extrato'.

Use a main só para iniciar o aplicativo

Inicialmente, no nosso arquivo "Caixa.java", que contém a 'main', criamos uma conta, pedindo um nome e um valor inicial.

Através do comando: `1 + numero.nextInt(9999)` nós sorteamos um número de conta de até 4 dígitos (`nextInt(9999)` gera números de 0 até 9998, somando 1 gera de 1 até 9999).

Com esses dados, criamos uma conta, que na verdade é o objeto 'minhaConta' da classe "Conta.java".

Iniciamos nosso banco ou caixa eletrônico através do método 'iniciar()'.

Note que tentamos enxugar a 'main', pois é uma boa prática. No geral, ela é usada apenas como 'gatilho', pra começar o programa e não pra ser enchida de variáveis e linhas de código.

Vamos pra classe "Conta.java".

Sistema bancário simples em Java

Nossos atributos (variáveis) são: nome, saldo, conta e saques.

Aqui vamos usar a real função do construtor: inicializar as variáveis. Nesse caso é obrigatório, pois não tem como, em um sistema bancário, criar uma conta sem ter - no mínimo - esses dados.

Vamos ver agora os principais métodos desse sistema bancário:

extrato()

Método simples, que exibe todas as informações do usuário.

sacar(int valor)

Esse método altera a variável 'saldo'. No caso, ele reduz ela.

Porém, só faz sentido reduzir (tirar dinheiro), se 'valor' for menor que o 'saldo', por isso é feito um tratamento através do teste condicional if.

Caso seja possível realizar o saque, devemos incrementar a variável 'saques', para termos controle do número de saques realizados. Caso não seja possível, exibimos uma mensagem informando o problema e nada ocorre.

Ou seja, é um belo exemplo de como usar o método set.

depositar(int valor)

Simplesmente adiciona um valor ao saldo atual.

iniciar()

Aqui é a tela inicial de nosso Caixa Eletrônico Progressivo Java.

Ele usa um laço do while que irá rodar o mini-sistema bancário enquanto o usuário não selecionar a opção de sair, que é o número 4 (while(opcao != 4)).

A cada iteração é exibido o menu através do método `exibeMenu()`, é pedido uma entrada (número) ao usuário e esse número é enviado para o método que vai direcionar o programa para a opção escolhida pelo usuário, o `escolheOpcao()`.

exibeMenu()

Um método simples desse sistema bancário é o '`exibeMenu()`', que não recebe nenhum argumento nem retorna nenhuma variável. Como o nome diz, ele simplesmente exibe a lista de opções de nosso sistema.

escolheOpcao(int opcao)

Vamos realmente escolher a opção que queremos no método '`escolheOpcao`', que recebe um número.

Mas que número é esse?

Ora, é o referente ao menu. Você vê as opções, entra com o número e esse método serve pra escolher a opção desejada.

Escolher opção...isso te lembra algo? Sim, o comando switch.

Caso tenha escolhido a opção 1, eles nos envia para o método '`extrato()`'.

Caso seja a 2, deveria ir para o método '`sacar()`', porém não é sempre que podemos sacar. Só podemos se tivermos realizado menos de 3 saques.

Caso seja possível realizar o saque, tanto o caso 2 com o caso 3 devem receber

um valor do usuário, que é o montante que vai ser sacado ou depositado.

O case 4 é para encerrar o sistema e qualquer outra opção cai na default que acusa como erro.

Código fonte Java do Aplicativo

Caixa.java

```
import java.util.Scanner;
```

```
import java.util.Random;
```

```
public class Caixa {
```

```
public static void main(String[] args){
```

```
    // Declarando as variáveis, Scanner e Random
```

```
String nome;
```

```
    double inicial;
```

```
    Scanner entrada = new Scanner(System.in);
```

```
Random numero = new Random();
```

```
    int conta = 1 + numero.nextInt(9999);
```

```
    //Obtendo os dados iniciais do Cliente
```

```
System.out.println("Cadastrando novo cliente.");
```

```
System.out.print("Ente com seu nome: ");
```

```
    nome = entrada.nextLine();
```

```
System.out.print("Entre com o valor inicial depositado na conta: ");
```

```
    inicial = entrada.nextDouble();
```

```
    //Criando a conta de um cliente
```

```
    Conta minhaConta = new Conta(nome, conta, inicial);
```

```
    minhaConta.iniciar();
```

```
}
```

```
}
```

Conta.java

```
import java.util.Scanner;
```

```
public class Conta {
```

private String nome;

private int conta, saques;

private double saldo;

Scanner entrada = new Scanner(System.in);

public Conta(String nome, int conta, double saldo_inicial){

this.nome=nome;

this.conta=conta;

 saldo=saldo_inicial;

 saques=0;

}

public void extrato(){

System.out.println("\tEXTRATO");

```
System.out.println("Nome: " + this.nome);
```

```
System.out.println("Número da conta: " + this.conta);
```

```
System.out.printf("Saldo atual: %.2f\n",this.saldo);
```

```
System.out.println("Saques realizados hoje: " + this.saques + "\n");
```

```
}
```

```
public void sacar(double valor){
```

```
    if(saldo >= valor){
```

```
        saldo -= valor;
```

```
        saques++;
```

```
System.out.println("Sacado: " + valor);
```

```
System.out.println("Novo saldo: " + saldo + "\n");
```

```
    } else {
```

```
System.out.println("Saldo insuficiente. Faça um depósito\n");
```

```
}
```

```
}
```

```
public void depositar(double valor)
```

```
{
```

```
    saldo += valor;
```

```
System.out.println("Depositado: " + valor);
```

```
System.out.println("Novo saldo: " + saldo + "\n");
```

```
}
```

```
public void iniciar(){
```

```
    int opcao;
```

```
do{
```

```
    exibeMenu();
```

```
    opcao = entrada.nextInt();
```

```
    escolheOpcao(opcao);
```

```
}while(opcao!=4);
```

```
}
```

```
public void exhibeMenu(){
```

```
System.out.println("\t Escolha a opção desejada");
```

```
System.out.println("1 - Consultar Extrato");
```

```
System.out.println("2 - Sacar");
```

```
System.out.println("3 - Depositar");
```

```
System.out.println("4 - Sair\n");
```

```
System.out.print("Opção: ");
```

```
}
```

```
public void escolheOpcao(int opcao){
```

```
    double valor;
```



```
switch( opcao ){
```

```
case 1:
```

```
    extrato();
```

```
break;
```

```
case 2:
```

```
if(saques<3){
```

```
    System.out.print("Quanto deseja sacar: ");
```

```
        valor = entrada.nextDouble();
```

```
        sacar(valor);
```

```
    } else{
```

```
        System.out.println("Limite de saques diários atingidos.\n");
```

```
    }
```

```
break;
```

case 3:

```
System.out.print("Quanto deseja depositar: ");
```

```
    valor = entrada.nextDouble();
```

```
    depositar(valor);
```

```
break;
```

case 4:

```
System.out.println("Sistema encerrado.");
```

```
break;
```

```
default:
```

```
System.out.println("Opção inválida");
```

```
    }
```

```
}
```

```
}
```

Treinamento Hacker:

Encontre falhas, brechas e erros no aplicativo acima.

Composição: trocando informações entre objetos

Um dos melhores artifícios do Java é a comunicação entre seus elementos, variáveis, métodos, objetos, classes e coisas que verá mais adiante. Mais que isso, o bacana é que podemos controlar essa comunicação, ela pode ser: pública, privada e protegida.

Nesse tutorial, daremos início a essa comunicação, falando sobre a Composição (Composition), que é simplesmente o ato de passar um objeto para outro, para usar seus métodos ou atributos.

O que é Composição em Java

É instanciar, ou usar, uma classe/objeto em outra(o). É como se elas se comunicassem, trocassem informações. Ou seja, serve para reutilizar dados, sem ter que criar mais código pra isso.

Simplesmente passamos a informação - na forma de Objeto - para outro Objeto, e este se encarrega de obter os dados e como trabalhar em cima dele.

Costuma-se dizer que composição é o ato de delegar trabalho para outro objeto.

Isso deixa seu código mais elegante, menor e mais seguro.

Para que serve a Composição em Java

Como citamos em nosso artigo introdutório sobre Classes e Objetos, a comunicação entre seus dados é essencial.

Talvez não note muito essa necessidade em projetos pequenos e de iniciantes, mas quando for um profissional, vai ter muita dor de cabeça com isso. Através de uma simplificação de projetos reais, vamos mostrar exemplos de códigos e passaremos exercícios para você notar a importância da Composição em aplicações Java.

Demos o exemplo de setores diferentes em uma empresa naquele artigo inicial.

Difícilmente uma empresa de grande porte vai mandar você fazer e ter acesso a todo o sistema, principalmente o setor financeiro.

Geralmente esse é mantido à sete chaves. Mas como expliquei lá, eles podem simplesmente compartilhar certos dados com você, como o salário final de cada funcionário, para você usar em sua classe "Funcionario".

Notou? Você criar sua classe, faz seu projeto e usa dados de outro Objeto/Classe. E o mais importante: é seguro! Você não faz a mínima idéia do que aconteceu lá, de como é calculado as coisas, dos segredos financeiros da empresa etc.

Graças a orientação a objetos - e outras coisas - isso é possível. Assim você não precisa repetir código.

Além da segurança, a Composição em Java faz com que não seja necessário repetir código. Você simplesmente usa o que já foi feito em outra Classe/Objeto, como é o caso do exemplo da empresa.

Imagina se o programador não soubesse Java? Ia ter que repetir todo o código do setor financeiro dentro da classe "Funcionario"?

Acredite, há coisas bizarras sendo feitas por aí.

Herança x Composição

Mais a frente, em nosso curso online de Java, iremos falar sobre uma das mais importantes, e controversas, e usadas funcionalidades do Java: a Herança. Isso nos leva ao encapsulamento de dados.

Não vamos entrar em muitos detalhes agora, mas basicamente herança é capacidade de uma classe herdar dados de outra.

Por exemplo, você cria uma classe mãe. Quer criar uma classe parecida com a mãe - mas com outras funcionalidades, você não precisa reescrever tudo. Basta fazer com que a classe filha herde a classe mãe.

Por exemplo, podemos criar a classe "Carro" com atributos como "motor", "portas", "carburador" etc, pois todos atributos estão presentes em todos carros. Fazemos então a classe "Fusca" e a classe "Ferrari" herdar a classe "Carro".

Ora, é claro que vamos adicionar mais funcionalidades na classe "Ferrari", pois ela possui mais atributos que o padrão genérico generalizado da classe "Carro". Mas não deixa de ser um carro.

Essa generalização, o fato de algumas classes herdadas não precisarem de todos os atributos da classe mãe, também devido ao fato de ao se mudar a classe mãe mudar todas as classes filhas e uma porção de outros detalhes, faz com que herança seja um conceito perigoso e não recomendado. Alguns simplesmente abominam a herança.

Não entraremos em detalhe agora, mas usar Composição é inofensivo e bem mais recomendado. Porém, cada caso é um caso.

□

Exemplo de uso de Composição em Java

Exercício: Descobrindo se um Funcionário chegou atrasado e seu tempo de trabalho.

Nesse exemplo, vamos criar 2 classes adicionais: "Funcionario" e "Hora"

Vamos criar um funcionário, o 'geddyLee', criar um objeto para armazenar a hora que ele chegou 'horaChegada' e a hora que ele saiu 'horaSaida'.

A partir da hora de chegada, vamos verificar se ele chegou atrasado (após as 8h) e usando informações dos dois objetos da classe "Hora", vamos saber quanto tempo ele trabalhou e armazenar essa informação na variável "tempoTrabalhado", do objeto 'geddyLee' da classe "Funcionario", e vamos guardar a informação caso ele tenha chegado atrasado

Ou seja, vamos trocar bastante informações entre esses objetos !

controleHorario.java

Essa é nossa classe principal, que tem o método main.

Na main, simplesmente criamos os objetos de Hora e Funcionário mencionados no enunciado.

Iniciamos os objetos de "Hora" já com seus valores (hora, minuto,segundo). Você pode mudar a seu gosto, para testes ou pedir ao usuário, através da classe Scanner.

A seguir, a título de informação, ele exibe a hora de chegada, de saída e o tempo total trabalhado em horas.

A main é bem pequena e enxuta, como deve ser.

Hora.java

Essa classe recebe três elementos inteiros: horas, minutos e segundos, e faz um tratamento.

O tratamento é: as horas devem ser inteiros entre 0 e 23, os minutos e segundos devem ser inteiros entre 0 e 59.

Caso não seja, é lançada uma exceção (artifício para mostrar que ocorreu um erro e a aplicação é encerrada).

Experimente colocar uma data errada para ver a mensagem de erro.

Os métodos dessa classe são simples getters um toString().

Método toString()

O método toString é um método especial, existente em todos os objetos de Java (pois está na classe Objects e todas classes são derivadas desta).

Como o nome pode sugerir, ele serve para retornar uma String.

Vamos formatar uma string com argumentos, por isso vamos usar o método format, para exibir as horas, minutos e segundos no formato: hh:mm:ss

Uma outra característica especial deste método é que ele se torna o padrão e

automaticamente invocado, caso tente imprimir um 'objeto'.

Isso mesmo, se colocar apenas um objetos em uma função de print, esse método será invocado.

Por isso , para imprimir na main, fizemos apenas:

```
System.out.println("Hora de chegada: " + horaChegada);
```

```
System.out.println("Hora de saída: " + horaSaida);
```

Mas poderíamos ter feito:

```
System.out.println("Hora de chegada: " + horaChegada.toString());
```

```
System.out.println("Hora de saída: " + horaSaida.toString());
```

Funcionario.java

Essa classe que vai receber a informação de outros objetos. No caso, ela recebe objetos da classe "Hora".

Vamos comentar seus métodos:

public double tempoAtraso(Hora horaChegada)

Vamos descobrir o tempo que o funcionário atrasou convertendo a hora de chegada em segundos.

Simplesmente transformamos a hora em que o funcionário chegou em segundos:

`horaChegada.getHour()*60*60 + horaChegada.getMinute()*60 +
horaChegada.getSecond()`

E subtraímos do tempo que ele deveria chegar na empresa (8h00min):

`8*3600.0`

E dividimos por 3600.0, para o resultado ser um double que representa as horas atrasadas.

public double horasTrabalhadas(Hora horaChegada, Hora horaSaida)

Para o cálculo do tempo trabalhado de um funcionário, vamos levar em conta somente as horas trabalhadas.

Primeiro, vamos transformar a hora da saída e a hora da chegada em minutos e subtrair esse valor:

```
(horaSaida.getHour()*60 + horaSaida.getMinute()) -  
(horaChegada.getHour()*60 + horaChegada.getMinute())
```

Agora temos a diferença entre os horários de chegada e saída, em minutos.

Então, dividimos tudo por 60.0 para ter o tempo trabalhado, em horas.

Caso esse tempo seja negativo, é porque você colocou uma data de saída anterior a data de chegada.

Então, um erro, com a explicação, é lançado e aplicação é encerrada.

public double getHorasTrabalhadas()

Um simples exemplo de get, que retorna a variável 'tempoTrabalhado'.

Código do nosso programa:

controleHorario.java

```
public class controleHorario {
```

```
public static void main(String[] args) {
```

```
    Hora horaChegada = new Hora(8, 0, 1);
```

```
    Hora horaSaida = new Hora(9, 30, 0);
```

```
    Funcionario geddyLee = new Funcionario("Geddy Lee", horaChegada,  
horaSaida);
```

```
    System.out.println("Hora de chegada: " + horaChegada);
```

```
    System.out.println("Hora de saída: " + horaSaida);
```

```
    System.out.printf("Horas trabalhadas:  
%.1f\n",geddyLee.getHorasTrabalhadas());
```

```
}
```

}

Hora.java

```
public class Hora {  
  
    private int hours,  
               minutes,  
               seconds;  
  
    public Hora (int hours, int minutes, int seconds){  
        //preenchendo as horas  
  
        if(hours>=0 && hours <24 )  
  
            this.hours = hours;  
  
        else  
  
            throw new IllegalArgumentException("Hora inválida");  
    }  
}
```

```
//preenchendo os minutos
```

```
if(minutes >=0 && minutes < 60)
```

```
this.minutes = minutes;
```

```
else
```

```
throw new IllegalArgumentException("Minutos inválidos");
```

```
//preenchendo os segundos
```

```
if( seconds >=0 && seconds < 60)
```

```
this.seconds = seconds;
```

```
else
```

```
throw new IllegalArgumentException("Segundos inválidos");
```

```
}
```

@Override

public String toString(){

return String.format("%d:%d:%d", getHour(), getMinute(), getSecond());

}

public int getHour(){

return this.hours;

}

public int getMinute(){

return this.minutes;

}

public int getSecond(){

return this.seconds;

}

}

Funcionario.java

```
public class Funcionario {
```

```
private String nome;
```

```
private boolean atraso;
```

```
private double tempoTrabalhado, tempoAtraso;
```

```
public Funcionario(String nome, Hora horaChegada, Hora horaSaida){
```

```
    this.nome=nome;
```

```
    this.tempoAtraso = tempoAtraso(horaChegada);
```

```
    if(this.tempoAtraso > 0)
```

```
        this.atraso=true;
```

```
if(atraso){
```

```
System.out.println("Funcionário " + this.nome + " atrasado. ");
```

```
}
```

```
this.tempoTrabalhado = horasTrabalhadas(horaChegada, horaSaida);
```

```
}
```

```
public double tempoAtraso(Hora horaChegada){
```

```
return ((horaChegada.getHour()*60*60 + horaChegada.getMinute()*60 +  
horaChegada.getSecond()) - 8*3600.0)/3600.0;
```

```
}
```

```
public double horasTrabalhadas(Hora horaChegada, Hora horaSaida){
```

```
double horas = ( (horaSaida.getHour()*60 + horaSaida.getMinute()) -  
(horaChegada.getHour()*60 + horaChegada.getMinute()) )/60.0;
```

```
if(horas < 0)
```



```
throw new IllegalArgumentException("Hora de saída anterior a hora de  
chegada");
```

```
return horas;
```

```
}
```

```
public double getHorasTrabalhadas(){
```

```
return this.tempoTrabalhado;
```

```
}
```

```
}
```

Exercício: Aplicativo Java para um Supermercado

Você foi selecionado para criar um aplicativo - em Java, claro - para um supermercado.

O que o dono do estabelecimento pediu a funcionalidade "Promoção para você", que funciona da seguinte maneira:

Após as 20h, todos os produtos recebem um desconto de 10%.

Aos sábados e domingos esse desconto vale o dia inteiro.

Ou seja, além de ter uma classe para os produtos do supermercado, você deve criar outra com o horário da compra.

Use a técnica de composition para passar o objeto referente ao horário da compra para o objeto referente ao produto que está sendo comprado. Assim, no objeto produto o preço é calculado com base no horário e dia da semana.

Na sua main, peça o preço do produto, dia da semana e horário da compra (hora, minuto e segundos), e devolva o valor final do produto com base naquele horário.

PS: Em uma aplicação real, não seria necessário fornecer esses dados. Eles

seriam obtidos por um leitor de código de barras para saber o preço do produto, e pegaria o horário do sistema, pra preencher os dados referentes ao horário.

Pedimos, porém, por questão de aprendizado, já que você não deve ter um leitor desses em casa...

Use constantes, e não números - declarando variáveis com o final Suponha que o governo te contratou para criar um aplicativo que, dentre outras coisas, analisa se um cidadão é maior de idade ou não.

Como você faria este teste condicional?

`if(idade >= 18)...` ou quem sabe `if(idade > 17)` ?

Ambos corretos, mas absolutamente não recomendáveis.

Nesse artigo ensinaremos uma importante lição de Java: Não usar números, e sim constantes.

Fugindo, um pouco, do escopo da seção "Orientação a Objetos", vamos introduzir uma boa prática na programação Java, que é o uso de constantes.

Essa seção tem por objetivo introduzir conceitos pro próximo artigo, sobre `enum`.

Por que usar constantes ao invés de números?

Imagine que você fez a aplicação. Uma aplicação real assim leva, fácil, milhares de linhas de código.

Durante todo o seu programa você usou esse número mágico, o 18.

Ora, é claro que ele vai aparecer várias vezes, pois a aplicação Java sobre maior idade etc.

Ok, até aí tudo bem.

Como você estudou pelo curso Java Progressivo, seu programa está funcionando que é uma beleza.

Agora imagine a situação: eleições.

Novos governantes, novas leis...a maioria agora é 16.

O que eles fazem?

'Chama o programador!' e lá vai você, sair catando no código o número 18 e substituindo por 16.

Imagina o trabalho...

Um mais esperto pode dizer: 'use regex ou ctrl+f' ou outro atalho qualquer pra achar 18 e substituir por 16.

Mas e se no seu aplicativo o número 18 aparecer e não tem nada a ver com idade?

Vai ter um baita bug.

Ou em vez de 18, você usou os números 0 ou 1 em sua aplicação, e agora precisa mudar.

Não dá pra substituir todos os 1 ou 0, tem que sair catando no código e mudando manualmente !!!

A vantagem de usar constantes é essa: você define uma vez, e usa o NOME da constante!

O certo seria:

```
int maioridade=18;  
  
if( idade >= maioridade)
```

Pronto. A variável 'maioridade' poderia aparecer 1 milhão de vezes no seu código.

No dia que o governo mudar ela, mude só a declaração do valor, de 18 pra 16 e isso será reproduzido em todo seu código.

Outra vantagem é a facilidade pra ler seu código, por você (depois de meses sem ver o código, a gente esquece o que significa aquele amontoado de códigos Java) ou pra outra pessoa que for ler.

Note:

```
if(idade > 17)
```

Não dá pra saber, exatamente o que é isso, só olhando. Tem que ver alguma documentação pra ter certeza.

Porém, se você se deparar com:

```
if(idade > maioridade)
```

...ora, maioridade é maioridade! Já sei do que se trata! Esse 'if' checa se a pessoa é adulta!

Outro exemplo:

$$A = 3.14 * r * r$$

O que é isso? Não sabe?

E agora...

$$A = PI * r * r$$

Agora ficou óbvio que é o número pi, então é algo relacionado com geometria. No caso, a área de um círculo.

O mais indicado seria definir constantes e usar seus nomes.

Fica mais bonito, elegante e profissional.

Ou seja, defina o valor de suas variáveis e use constantes.

final - declarando variáveis constantes

O Java reservou um artifício bem interessante para você usar suas constantes, que é a keyword 'final'.

Quando você define uma variável com o 'final', ela se torna constante.

Se tentar mudar ela, obterá um erro.

A declaração é a seguinte:

```
final tipo nomeDaVariavel
```

Por exemplo:

```
final int SIM = 1;
```

```
final int NAO = 0;
```

Pronto. Agora você pode usar 'SIM' e 'NAO', ao invés de usar números.

Usar um 'OK' ou 'ERRO', ao invés de números estranho e bizarros.

Em classes, seria:

```
private final int STATUS_ON = 1;
```



```
private final int STATUS_OFF = 0;
```

De agora em diante, usaremos pouco números.

Mesmo que suas aplicações sejam simples e curtas - e aparente não haver necessidade de constantes - use-as.

É um costume que levará quando se tornar profissional e criar aplicações reais.

final em Classes e Objetos

Em uma classe, você poderá declarar um atributo como final de duas maneiras possíveis:

Ao inicializar a variável, inicie com seu valor. Só faça isso se quiser que todos os objetos daquela classe possuam essa variável com o mesmo valor.

Você também poderá inicializar suas variáveis através de um método construtor. Assim, cada objeto inicializado poderá ter um valor diferente.

Mas se lembre: após inicializadas em sua declaração ou no construtor, não podem mais ser alteradas.

Certifique-se de que quer mesmo ter uma constante em sua aplicação. Se sim, sempre use a keyword final, pois isso evitará que a variável tenha seu valor alterado por acidente ou má programação.

enum: A melhor maneira para manusear constantes

Conforme vimos no tutorial de Java da aula passada, o uso de constantes em suas aplicações é de vital importância pra reusabilidade e legibilidade do código.

O uso de constantes em Java é tão, mas tão importante que a linguagem possui uma ferramenta especial para você manusear - com muita facilidade - suas constantes: o enum

O que é enum em Java

enum nada mais são que um recurso, muito útil e parecido com Orientação a Objetos, feito para tratar, guardar e usar com facilidade suas constantes.

Embora sejam do tipo referência, possuam construtores, declaração e uso parecido com Objetos/Classes, não são. Mas se parecem, por isso vamos apresentar os enum nesta seção de nossa apostila online de Java.

De uma maneira simplificada, você pode ver enum como uma 'classe' especial para tratar constantes.

Você notará melhor isso na sintaxe de declaração e criação do código das enums.

Declarando uma enum em Java

Você pode declarar uma enum da seguinte maneira:

```
public enum nome_da_enum{  
  
    //código de sua enum  
  
}
```

Por exemplo:

```
public enum Bebidas{  
  
    CocaCola, Suco, Agua;  
  
}
```

Obviamente, essas constantes não servem de nada, pois não atribuímos nada a elas.

De antemão, você pode ver cada uma dessas constantes como um, digamos, 'objeto' da 'classe' enum.

Usando enum em Java

Vamos adicionar algumas funcionalidades a essa enum, como o preço de cada bebida:

```
public enum Bebida{  
  
    CocaCola(2.00),  
  
    Suco(1.50),  
  
    Agua(1.00);  
  
    private double preco;  
  
    Bebida(double preco){  
        this.preco = preco;  
    }  
  
    public double getPreco(){  
        return this.preco;  
    }  
}
```

Primeiro declaramos nossas constantes: CocaCola, Suco e Agua.

Note que inicializamos ela com um double, pois criamos um método construtor que recebe um double: Bebidas(double preco) {...}

Tudo bem parecido com Classes/Objetos, inclusive temos que definir uma variável interna, a 'preco'.

Para invocar um método fazemos: nome_do_enum.nomeDaConstante.metodo()

Por exemplo, para saber quanto custa uma Agua: Bebida.Agua.getPreco();

Notes nomes: Bebida, Agua, getPreco.

Note mesmo, como fica óbvio. Você não precisa se preocupar com o valor numérico, e sim com o nome.

Nesse caso, o nome é bem fácil lembrar, pois usamos palavras óbvias.

Exemplo de uso de enum em Java

Exercício: Aplicativo para Lanchonete

O seguinte aplicativo exibe um menu de uma lanchonete, com 3 tipos de comida e 3 tipos de bebida.

O cliente escolhe o que quer comer e o programa retorna o valor total da conta.

Para tal criamos duas enums: 'Bebida' e 'Comida'.

Cada uma dessas enums recebe dois valores: seu nome (String nome) e seu preço (double preco).

Além disso, as enum tem método construtor, variáveis privadas e métodos que retorna o nome e preço do produto.

Também criamos o método 'menu()', que simplesmente exibe o cardápio.

Através do objeto 'entrada', da classe Scanner, perguntamos a opção desejada ao cliente, e enviamos essa informação para o método 'preco()', que retorna o valor do item pedido pelo cliente.

Quando terminar de pedir, ele deve digitar '0' e a aplicação se encerrará.

EnumTest.java


```
import java.util.Scanner;
```

```
public class EnumTest {
```

```
public enum Bebida{
```

```
    CocaCola("Coca-cola", 2.00),
```

```
    Suco("Suco", 1.50),
```

```
    Agua("Agua", 1.00);
```

```
private double preco;
```

```
private String nome;
```

```
    Bebida(String nome, double preco){
```

```
        this.nome = nome;
```

```
        this.preco = preco;
```

```
    }
```

```
public double getPreco(){
```

```
    return this.preco;
```

```
    }
```

```
public String getNome(){
```

```
    return this.nome;
```

```
    }
```

```
}
```

```
public enum Comida{
```

```
    Sanduiche("Sanduiche", 4.0),
```

```
    HotDog("HotDog", 3.0),
```

```
    Xburger("X-Burger", 3.5);
```

```
private double preco;
```

```
private String nome;
```

```
    Comida(String nome, double preco){
```

```
this.nome = nome;
```

```
this.preco = preco;
```

```
}
```

```
public double getPreco(){
```

```
return this.preco;
```

```
}
```

```
public String getNome(){
```

```
return this.nome;
```

```
}
```

```
}
```

```
public static void menu(){
```

```
System.out.println("\tBebidas");
```

```
System.out.println("1." + Bebida.CocaCola.getNome() + ": R$" +  
Bebida.CocaCola.getPreco());
```

```
System.out.println("2." + Bebida.Suco.getNome() + ": R$" +  
Bebida.Suco.getPreco());
```

```
System.out.println("3." + Bebida.Agua.getNome() + ": R$" +  
Bebida.Agua.getPreco());
```

```
System.out.println("\tComidas");
```

```
System.out.println("4." + Comida.Sanduiche.getNome() + ": R$" +  
Comida.Sanduiche.getPreco());
```

```
System.out.println("5." + Comida.HotDog.getNome() + ": R$" +  
Comida.HotDog.getPreco());
```

```
System.out.println("6." + Comida.Xburger.getNome() + ": R$" +  
Comida.Xburger.getPreco());
```

```
System.out.println("0. Sair");
```

```
System.out.print("Escolha sua opção: ");
```

```
}
```

```
public static double preco(int opcao){
```

```
switch( opcao ){
```

```
case 1:
```

```
return Bebida.CocaCola.getPreco();
```

```
case 2:
```

```
return Bebida.Suco.getPreco();
```

```
case 3:
```

```
return Bebida.Agua.getPreco();
```

```
case 4:
```

```
return Comida.Sanduiche.getPreco();
```

case 5:

return Comida.HotDog.getPreco();

case 6:

return Comida.Xburger.getPreco();

default:

return 0.0;

}

}

public static void main(String[] args) {

double total=0.0;

int opcao=0;

Scanner entrada = new Scanner(System.in);

do{

```
menu();
```

```
opcao = entrada.nextInt();
```

```
total += preco(opcao);
```

```
System.out.println("Opção escolhida: " + opcao);
```

```
System.out.println("Valor de sua conta: " + total + "\n");
```

```
    }while(opcao != 0);
```

```
}
```

static - Usando membros estáticos em Java

Agora que você sabe a importância das constantes em aplicações Java e como manusear as constantes de seus programas com enum, vamos mostrar uma aplicação mais direta das constantes em nosso estudo de Orientação a Objetos.

Esse artigo Java mostra o uso das variáveis do tipo static, artifício muito usado para manter o controle sobre todos os objetos de uma mesma classe, por exemplo.

O que é static em Java

Só pelo nome, já dá pra desconfiar que é algo relacionado com constante, algo 'parado' (estático).

Quando definimos uma classe e criamos vários objetos dela, já sabemos que cada objeto irá ser uma cópia fiel da classe, porém com suas próprias variáveis e métodos em lugares distintos da memória.

Ou seja, o objeto 'fusca' tem suas variáveis próprias, diferentes do objeto 'ferrari', embora ambos tenham o mesmo 'modelo', que é a classe 'Carro'.

Quando definimos variáveis com a palavra static em uma classe ela terá um comportamento especial: ela será a mesma para todos os objetos daquela classe.

Ou seja, não haverá um tipo dela em cada objeto. Todos os objetos, ao acessarem e modificarem essa variável, acessarão a mesma variável, o mesmo espaço da memória, e a mudança poderá ser vista em todos os objetos.

Como declarar uma variável static em Java

Basta colocar a palavra static antes do tipo:

```
static tipo nomeDaVariavel
```

Em uma classe:

```
private static int vendidos;
```

```
public static int totalAlunos;
```

Quando usar variáveis static em Java

Principalmente quando você quiser ter um controle sobre os objetos ou quando todos os objetos devem partilhar uma informação (evitar ter que fazer Composição ou chamar métodos de outros objetos).

Exemplo 1: Para controle de número total de objetos

Imagine que você é dono de uma loja de venda de veículos.

Cada um que vende, é um comprador diferente, dados diferentes etc. Portanto, cada carro será um objeto.

Você cria a variável estática 'total', e no construtor a incrementa (total++). Pronto, saberá quantos carros foram vendidos, automaticamente.

Um exemplo parecido, seria para um aplicativo de uma escola ou empresa, para controle de quantos funcionários existem na empresa, ou em cada setor dela.

Exemplo 2: Para compartilhar uma informação

Muitas aplicações Java, principalmente jogos, usam o static para compartilhar informações sobre o número de objetos. O uso tem a ver algoritmos de sobre inteligência computacional.

Por exemplo, em um jogo de futebol, quando você joga contra o computador.

A máquina tem uma estratégia diferente dependendo do time que você escolher, do modo que você jogar, depende do número de jogadores dele e seu também.

E como saber o número atual de jogadores do time deles?

Ora, um método static. Pois todos os jogadores são objetos de uma mesma classe.

Em um jogo de luta, se existirem 3 inimigos contra 2 personagens seus. Provavelmente o computador vai te atacar.

Mas se você matar dois deles, eles estarão em desvantagem. Existe então um método que faz com que os inimigos corram, caso o número de jogadores dele seja menor que o seu.

Para obter essa informação, os objetos devem partilhar da informação: número de personagens vivos.

Exemplo de código com static

O seguinte código cria uma classe bem simples, a "Carro" que simplesmente informa quando o objeto é criado - através do método construtor padrão main - e incrementa a variável 'total', que vai guardar a informação do número total de objetos/carros criados em sua aplicação.

staticTest.java

```
public class staticTest {  
  
public static void main(String[] args) {  
  
    Carro fusca = new Carro();  
  
    Carro ferrari = new Carro();  
  
    Carro jipe = new Carro();  
  
    }  
  
}
```

Carro.java

```
public class Carro {
```

```
public static int total=0;
```

```
Carro(){
```

```
    total++;
```

```
System.out.println("Objeto criado. Existem "+total+" objetos dessa classe");
```

```
}
```

```
}
```

Apostila de Java, Capítulo 4 - Orientação a objetos básica

Página 51, 4.12 Exercícios: Orientação a Objetos básica

Enunciados

QUESTÃO 01:

Modele um funcionário. Ele deve ter o nome do funcionário, o departamento onde trabalha, seu salário

(double), a data de entrada no banco (String) e seu RG (String).

Você deve criar alguns métodos de acordo com sua necessidade. Além deles, crie um método bonifica

que aumenta o salario do funcionário de acordo com o parâmetro passado como argumento. Crie também um método calculaGanhoAnual, que não recebe parâmetro algum, devolvendo o valor do salário multiplicado por 12..

A ideia aqui é apenas modelar, isto é, só identifique que informações são importantes e o que um funcionário faz. Desenhe no papel tudo o que um Funcionario tem e tudo que ele faz.

QUESTÃO 02:

Transforme o modelo acima em uma classe Java. Teste-a, usando uma outra classe que tenha o main.

Você deve criar a classe do funcionário chamada Funcionario, e a classe de teste você pode nomear como

quiser. A de teste deve possuir o método main.

Um esboço da classe:

```
class Funcionario {  
  
    double salario;  
  
    // seus outros atributos e métodos  
  
    void bonifica(double aumento) {  
  
        // o que fazer aqui dentro?  
  
    }  
  
    double calculaGanhoAnual() {  
  
        // o que fazer aqui dentro?  
  
    }  
}
```

Você pode (e deve) compilar seu arquivo java sem que você ainda tenha terminado sua classe Funcionario. Isso evitará que você receba dezenas de erros de compilação de uma vez só.

Crie a classe Funcionario, coloque seus atributos e, antes de colocar qualquer método, compile o arquivo java.

Funcionario.class será gerado, não podemos “executá-la” pois não há um main, mas assim verificamos que nossa classe Funcionario já está tomando forma.

Esse é um processo incremental. Procure desenvolver assim seus exercícios, para não descobrir só no fim

do caminho que algo estava muito errado.

Um esboço da classe que possui o main:

```
class TestaFuncionario {  
  
    public static void main(String[] args) {  
        Funcionario f1 = new Funcionario();  
  
        f1.nome = "Fiodor";  
        f1.salario = 100;  
        f1.bonifica(50);  
  
        System.out.println("salario atual:" + f1.salario);  
        System.out.println("ganho anual:" + f1.calculaGanhoAnual());  
  
    }  
}
```

Incremente essa classe. Faça outros testes, imprima outros atributos e invoque os métodos que você criou a mais.

Lembre-se de seguir a convenção java, isso é importantíssimo. Isto é, nomeDeAtributo, nomeDeMetodo, nomeDeVariavel, NomeDeClasse, etc...

QUESTÃO 03:

Crie um método mostra(), que não recebe nem devolve parâmetro algum e simplesmente imprime todos

os atributos do nosso funcionário. Dessa maneira, você não precisa ficar copiando e colando um monte

de System.out.println() para cada mudança e teste que fizer com cada um de seus funcionários, você

simplesmente vai fazer:

```
Funcionario f1 = new Funcionario();
```

```
// brincadeiras com f1....
```

```
f1.mostra();
```

Veremos mais a frente o método toString, que é uma solução muito mais elegante para mostrar a representação de um objeto como String, além de não jogar tudo pro System.out (só se você desejar).

O esqueleto do método ficaria assim:

```
class Funcionario {
```

```
    // seus outros atributos e métodos
```

```
void mostra() {  
    System.out.println("Nome: " + this.nome);  
    // imprimir aqui os outros atributos...  
    // tambem pode imprimir this.calculaGanhoAnual()  
    }  
}
```

QUESTÃO 04:

Construa dois funcionários com o new e compare-os com o ==. E se eles tiverem os mesmos atributos?

Para isso você vai precisar criar outra referência:

```
Funcionario f1 = new Funcionario();  
  
    f1.nome = "Fiodor";  
  
    f1.salario = 100;  
  
Funcionario f2 = new Funcionario();  
  
    f2.nome = "Fiodor";  
  
    f2.salario = 100;
```

```
if (f1 == f2) {  
    System.out.println("iguais");  
  
    } else {
```

```
        System.out.println("diferentes");  
    }
```

QUESTÃO 05:

Crie duas referências para o mesmo funcionário, compare-os com o ==. Tire suas conclusões. Para criar

duas referências pro mesmo funcionário:

```
Funcionario f1 = new Funcionario();
```

```
f1.nome = "Fiodor";
```

```
f1.salario = 100;
```

```
Funcionario f2 = f1;
```

O que acontece com o if do exercício anterior?

QUESTÃO 06:

Em vez de utilizar uma String para representar a data, crie uma outra classe, chamada Data.

Ela possui 3 campos int, para dia, mês e ano. Faça com que seu funcionário passe a usá-la. (é parecido

com o último exemplo, em que a Conta passou a ter referência para um Cliente).

```
class Funcionario {
```

```
    Data dataDeEntrada; // qual é o valor default aqui?
```

```
// seus outros atributos e métodos
```

```
}
```

```
class Data {
```

```
    int dia;
```

```
    int mes;
```

```
    int ano;
```

```
}
```

Modifique sua classe TestaFuncionario para que você crie uma Data e atribua ela ao Funcionario:

```
Funcionario f1 = new Funcionario();
```

```
//...
```

```
Data data = new Data(); // ligação!
```

```
f1.dataDeEntrada = data;
```

Faça o desenho do estado da memória quando criarmos um Funcionario.

QUESTÃO 07:

Modifique seu método mostra para que ele imprima o valor da dataDeEntrada daquele Funcionario:

```
class Funcionario {  
  
    // seus outros atributos e métodos  
  
    Data dataDeEntrada;  
  
    void mostra() {  
  
        System.out.println("Nome: " + this.nome);  
  
        // imprimir aqui os outros atributos...  
  
        System.out.println("Dia: " + this.dataDeEntrada.dia);  
  
        System.out.println("Mês: " + this.dataDeEntrada.mes);  
  
        System.out.println("Ano: " + this.dataDeEntrada.ano);  
  
    }  
}
```

Teste-o. O que acontece se chamarmos o método mostra antes de atribuirmos uma data para este

Funcionario?

QUESTÃO 08:

O que acontece se você tentar acessar um atributo diretamente na classe? Como,

por exemplo:

```
Funcionario.salario = 1234;
```

Esse código faz sentido? E este:

```
Funcionario.calculaGanhoAtual();
```

Faz sentido perguntar para o esquema do Funcionario seu valor anual?

SOLUÇÕES

QUESTÕES 1, 2, 3, 4, e 5

As questões de número 1, 2, 3, 4 e 5 se referem a somente um aplicativo em Java, feito passo-a-passo através das questões.

Basicamente criamos a classe Funcionario com os atributos: nome, departamento, dataEntrada, RG e salario. Além disso, essa classe também possui o método mostra(), que ao ser solicitado, exibe todas as informações de um objeto.

Já a classe CaelumCap4 é a que contém a main, e que cria dois objetos do tipo Funcionario: f1 e f2.

Mostramos os dados ao iniciar, depois fazemos uma bonificação em f1 e mostramos seus dados novamente. Note que o valor do salário e o ganho anual realmente mudaram.

Depois criamos um clone de f1: f2.

Porém, embora possua os mesmos atributos e valores, esses objetos referenciam regiões na memória, portanto não são iguais, como mostra o teste condicional if.

Depois criamos outro objeto: f3

Esse objeto é referenciado para apontar o mesmo local de memória que f1 referencia.

Portanto, ao comparar, vemos que f1 e f2 são iguais.

Nossa solução fica:

--> CaelumCap4.java

```
public class CaelumCap4 {
```

```
public static void main(String[] args) {
```

```
    Funcionario f1 = new Funcionario();
```

```
    Funcionario f2 = new Funcionario();
```

```
    f1.nome = "Programador Java";
```

```
    f1.departamento = "TI";
```

```
    f1.dataEntrada = "21/12/2012";
```

```
    f1.RG = "123456789-0";
```

```
    f1.salario = 2000;
```

```
    f1.mostra();
```

```
    System.out.println("\nApós fazer o curso Java Progressivo, o funcionário obteve  
    bonificação de mil reais.");
```

```
    System.out.println("Os novos dados, após o aumento, são:\n");
```

```
f1.bonifica(1000);
```

```
f1.mostra();
```

```
//Vamos agora preencher os dados de f2
```

```
f2.nome = "Programador Java";
```

```
f2.departamento = "TI";
```

```
f2.dataEntrada = "21/12/2012";
```

```
f2.RG = "123456789-0";
```

```
f2.salario = 2000;
```

```
f2.bonifica(1000);
```

```
if(f1 == f2){
```

```
System.out.println("\nFuncionários 1 e 2 são iguais");
```

```
    } else {
```

```
System.out.println("\nFuncionários 1 e 2 são diferentes");
```

```
    }
```

```
Funcionario f3 = f1;
```

```
if(f1 == f3){
```

```
    System.out.println("Funcionários 1 e 3 são iguais");
```

```
    } else {
```

```
        System.out.println("Funcionários 1 e 3 são diferentes");
```

```
    }
```

```
}
```

```
}
```

```
--> Funcionario.java
```

```
public class Funcionario {
```

```
    String nome,
```

```
        departamento,
```

```
        dataEntrada,
```

```
        RG;
```

```
    double salario;
```

```
void bonifica(double aumento){
```

```
    this.salario += aumento;
```

```
}
```

```
double calculaGanhoAtual(){
```

```
    return (salario * 12);
```

```
}
```

```
void mostra(){
```

```
    System.out.println("Nome do funcionário: " + this.nome);
```

```
    System.out.println("Departamento: " + this.departamento);
```

```
    System.out.println("Entrou em: " + this.dataEntrada);
```

```
    System.out.println("RG: " + this.RG);
```

```
    System.out.println("Salário: " + this.salario);
```

```
System.out.println("Ganha anualmente: " + calculaGanhoAtual());  
  
    }  
  
}
```

QUESTÕES 6, 7 e 8

A criação da classe Data é bem simples e sem mais complicações.

Na classe Funcionario criamos um objeto da classe Data, mas ele só é instanciado na classe principal, a CaelumCap4.

Ao final da main mostramos como é possível acessar e alterar diretamente seus valores.

Nossa classes ficam assim:

--> CaelumCap4.java

```
public class CaelumCap4 {
```

```
    public static void main(String[] args) {
```

```
        Funcionario f1 = new Funcionario();
```

```
        Funcionario f2 = new Funcionario();
```

```
        Data data = new Data();
```

```
f1.nome = "Programador Java";
```

```
f1.departamento = "TI";
```

```
f1.RG = "123456789-0";
```

```
f1.salario = 2000;
```

```
f1.dataEntrada = data;
```

```
f1.mostra();
```

```
System.out.println("\nApós fazer o curso Java Progressivo, o funcionário obteve  
bonificação de mil reais.");
```

```
System.out.println("Os novos dados, após o aumento, são:\n");
```

```
f1.bonifica(1000);
```

```
f1.mostra();
```

```
//Vamos agora preencher os dados de f2
```

```
f2.nome = "Programador Java";
```

```
f2.departamento = "TI";
```

```
f2.RG = "123456789-0";
```

```
f2.salario = 2000;
```

```
f2.bonifica(1000);
```



```
if(f1 == f2){
```

```
    System.out.println("\nFuncionários 1 e 2 são iguais");
```

```
    } else {
```

```
        System.out.println("\nFuncionários 1 e 2 são diferentes");
```

```
    }
```

```
    Funcionario f3 = f1;
```

```
    if(f1 == f3){
```

```
        System.out.println("Funcionários 1 e 3 são iguais");
```

```
    } else {
```

```
        System.out.println("Funcionários 1 e 3 são diferentes");
```

```
    }
```

```
    System.out.println("Acessando e alterando os valores diretamente:");
```

```
        f1.salario = 1234;
```

```
System.out.println(f1.calculaGanhoAtual());
```

```
}
```

```
}
```

--> Funcionario.java

```
public class Funcionario {
```

```
String nome,
```

```
    departamento,
```

```
    RG;
```

```
Data dataEntrada;
```

```
double salario;
```

```
void bonifica(double aumento){
```

```
this.salario += aumento;
```

```
}
```

```
double calculaGanhoAtual(){
```

```
return (salario * 12);
```

```
}
```

```
void mostra(){
```

```
System.out.println("Nome do funcionário: " + this.nome);
```

```
System.out.println("Departamento: " + this.departamento);
```

```
System.out.println("Data de entrada: " + dataEntrada.dia +  
                    "/" + dataEntrada.mes + "/" +  
                    + dataEntrada.ano);
```

```
System.out.println("RG: " + this.RG);
```

```
System.out.println("Salário: " + this.salario);
```

```
System.out.println("Ganha anualmente: " + calculaGanhoAtual());
```

```
}
```

```
}
```

--> Data.java

```
public class Data {
```

```
    int dia,
```

```
        mes,
```

```
        ano;
```

```
}
```

Página 55, Desafíos

Enunciados

QUESTÃO 01:

Um método pode chamar ele mesmo. Chamamos isso de recursão. Você pode resolver a série de fibonacci

usando um método que chama ele mesmo. O objetivo é você criar uma classe, que possa ser usada da

seguinte maneira:

```
Fibonacci fibo = new Fibonacci();
```

```
int i = fibo.calculaFibonacci(6);
```

```
System.out.println(i);
```

Aqui imprimirá 8, já que este é o sexto número da série.

Este método calculaFibonacci não pode ter nenhum laço, só pode chamar ele mesmo como método.

Pense nele como uma função, que usa a própria função para calcular o resultado.

QUESTÃO 02:

Por que o modo acima é extremamente mais lento para calcular a série do que o modo iterativo (que se

usa um laço)?

QUESTÃO 03:

Escreva o método recursivo novamente, usando apenas uma linha. Para isso, pesquise sobre o operador

condicional ternário. (ternary operator)

Soluções

QUESTÃO 01:

Vamos criar um método chamado `calculaFibonacci` que recebe um inteiro como argumento. Chamemos esse número de `n`.

Da definição dos números de Fibonacci, sabemos que se esse número `n` for 0, o valor do número Fibonacci é 0.

Caso `n` seja 1, o valor do número de Fibonacci é 1.

E para TODOS os demais, o número de Fibonacci é a soma dos dois anteriores.

Por exemplo, para `n=3`:

$$\text{calculaFibonacci}(3) = \text{calculaFibonacci}(2) + \text{calculaFibonacci}(1)$$

$$\text{Porém: } \text{calculaFibonacci}(2) = \text{calculaFibonacci}(1) + \text{calculaFibonacci}(0) = 1 + 0 = 1$$

Agora colocamos esse valor na outra equação:

$$\text{calculaFibonacci}(3) = \text{calculaFibonacci}(2) + \text{calculaFibonacci}(1) = 1 + 1 = 2$$

Agora, para calcular `n=4`:

$$\text{calculaFibonacci}(4) = \text{calculaFibonacci}(3) + \text{calculaFibonacci}(2)$$

Mas: $\text{calculaFibonacci}(3) = \text{calculaFibonacci}(2) + \text{calculaFibonacci}(1)$

E: $\text{calculaFibonacci}(2) = \text{calculaFibonacci}(1) + \text{calculaFibonacci}(0)$

Então:

$\text{calculaFibonacci}(4) = \text{calculaFibonacci}(2) + \text{calculaFibonacci}(1) +$
 $\text{calculaFibonacci}(1) + \text{calculaFibonacci}(0)$

Substituindo $\text{calculaFibonacci}(2)$ novamente:

$\text{calculaFibonacci}(4) = \text{calculaFibonacci}(1) + \text{calculaFibonacci}(0) +$
 $\text{calculaFibonacci}(1) + \text{calculaFibonacci}(1) + \text{calculaFibonacci}(0) = 1 + 1 + 1 + 1$
 $+ 1 = 5$

Ou seja, não importa o número que coloquemos, ele será sempre substituído pela soma dos dois elementos anteriores. Cada um desses elementos anteriores serão substituído pela soma de seus respectivos elementos anteriores. E assim vai indo...até sobrar somente uma soma de $\text{calculaFibonacci}(1)$ e $\text{calculaFibonacci}(0)$.

Interessante, não?

Podemos representar em Java essa idéia matemática da seguinte maneira:

```
public class Fibonacci {
```

```
    int calculaFibonacci(int n){
```

```
if(n==0){
```

```
    return 0;
```

```
}
```

```
if(n==1){
```

```
    return 1;
```

```
}
```

```
    return ( calculaFibonacci(n-1) + calculaFibonacci(n-2) );
```

```
}
```

```
}
```

QUESTÃO 02:

Experimente colocar $n=30$, na questão passada. É calculado bem rápido.

Agora tente $n=50$ e verá uma baita demora.

Se você testar obter esses números com a técnica utilizada nos exercícios do

capítulo 3 da apostila da Caelum, que usa somente laços, verá que é muito mais rápido.

Isso porque os métodos são bem mais 'pesados', em termos computacionais.

Se notar bem, o método `calculaFibonacci` é chamado muitas vezes.

Para $n=6$, o método é invocado 25 vezes.

Para $n=20$, o método é invocado 21891 vezes !!!

(para saber quantas vezes o método foi chamado, crie uma variável inteira na classe Fibonacci, e faça 'chamadas++' assim que se inicia o método, e ela guardará quantas vezes esse método foi acessado)

QUESTÃO 03:

O operador condicional ternário tem a seguinte sintaxe:

(teste de condição) ? (retorno isso caso seja true) : (retorna isso caso seja false) ;

Ele faz um teste condicional ($n < 1$, $a == b$, $x \geq 0$ etc). Caso ele seja verdadeiro, retorna o que vem logo após a interrogação '?', e caso seja falso, retorna o que vem após os dois pontos ':'

Sim, é como se fosse um teste condicional IF ELSE, mas usamos apenas '?' e ':'

Voltemos ao Fibonacci.

Você noto que, se $n=0$, o retorno é 0. E se $n=1$, o retorno é 1?

Ou seja: se $n < 2$, o número de Fibonacci é o próprio n .

E se não for? Aí o retorno é a soma dos dois elementos anteriores.

Agora, de posse dessas informações, podemos calcular os números de Fibonacci em apenas uma linha:

```
public class Fibonacci {  
  
    int calculaFibonacci(int n){  
  
return ( n < 2 ? n : ( calculaFibonacci(n-1) + calculaFibonacci(n-2) ) );  
  
    }  
  
}
```

Apostila de Java, Capítulo 6 - Modificadores de acesso e atributos de classe

6.8 - Exercícios: Encapsulamento, construtores e static

1) Adicione o modificador de visibilidade (private, se necessário) para cada atributo e método da classe Funcionario.

Tente criar um Funcionario no main e modificar ou ler um de seus atributos privados. O que acontece?

2) Crie os getters e setters necessários da sua classe Funcionario. Por exemplo:

```
class Funcionario {  
  
    private double salario;  
  
    // ...  
  
    public double getSalario() {  
        return this.salario;  
    }  
  
    public void setSalario(double salario) {  
        this.salario = salario;  
    }  
}
```

Não copie e cole! Aproveite para praticar sintaxe. Logo passaremos a usar o

Eclipse e aí sim teremos procedimentos mais simples para este tipo de tarefa.

Repare que o método calculaGanhoAnual parece também um getter. Aliás, seria comum alguém nomeá-lo de getGanhoAnual. Getters não precisam apenas retornar atributos. Eles podem trabalhar com esses dados.

3) Modifique suas classes que acessam e modificam atributos de um Funcionario para utilizar os getters e setters recém criados.

Por exemplo, onde você encontra:

```
f.salario = 100;
```

```
System.out.println(f.salario);
```

passa para:

```
f.setSalario(100);
```

```
System.out.println(f.getSalario());
```

4) Faça com que sua classe Funcionario possa receber, opcionalmente, o nome do Funcionario durante a criação do objeto. Utilize construtores para obter esse resultado.

Dica: utilize um construtor sem argumentos também, para o caso de a pessoa não querer passar o nome do Funcionario.

Seria algo como:

```
class Funcionario {  
  
    public Funcionario() {  
  
        // construtor sem argumentos  
  
    }  
}
```

```
public Funcionario(String nome) {  
  
    // construtor que recebe o nome  
  
}  
  
}
```

Por que você precisa do construtor sem argumentos para que a passagem do nome seja opcional?

5) (opcional) Adicione um atributo na classe Funcionario de tipo int que se chama identificador. Esse identificador deve ter um valor único para cada instância do tipo Funcionario. O primeiro Funcionario instanciado tem identificador 1, o segundo 2, e assim por diante. Você deve utilizar os recursos aprendidos aqui para resolver esse problema.

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

6) (opcional) Crie os getters e setters da sua classe Empresa e coloque seus atributos como private. Lembre-se de que não necessariamente todos os atributos devem ter getters e setters.

Por exemplo, na classe Empresa, seria interessante ter um setter e getter para a sua array de funcionários?

Não seria mais interessante ter um método como este?

```
class Empresa {  
  
    // ...  
  
    public Funcionario getFuncionario (int posicao) {
```



```
return this.empregados[posicao];
```

```
}
```

```
}
```

7) (opcional) Na classe Empresa, em vez de criar um array de tamanho fixo, receba como parâmetro no construtor o tamanho do array de Funcionario.

Com esse construtor, o que acontece se tentarmos dar `new Empresa()` sem passar argumento algum? Por quê?

8) (opcional) Como garantir que datas como 31/2/2012 não sejam aceitas pela sua classe Data?

9) (opcional) Crie a classe PessoaFisica. Queremos ter a garantia de que pessoa física alguma tenha CPF inválido, nem seja criada PessoaFisica sem cpf inicial. (você não precisa escrever o algoritmo de validação de cpf, basta passar o cpf por um método valida(String x)...)

6.9 - Enunciado dos desafios

1) Porque esse código não compila?

```
class Teste {  
  
    int x = 37;  
  
    public static void main(String [] args) {  
  
        System.out.println(x);  
  
    }  
  
}
```

2) Imagine que tenha uma classe FabricaDeCarro e quero garantir que só existe um objeto desse tipo em toda a memória. Não existe uma palavra chave especial para isto em Java, então teremos de fazer nossa classe de tal maneira que ela respeite essa nossa necessidade. Como fazer isso? (pesquise: singleton design pattern)

Solução das questões do capítulo 6 da apostila

Como é de praxe na apostila de Java da Caelum, as questões são na verdade um programa em Java maior, onde se faz um pouco dessa aplicação em cada questão. Vamos resolver mostrar o código de uma aplicação que resolve as 7 primeiras questões.

Questão 01

Inicialmente, vamos criar 3 atributos: o nome e o setor do funcionário (que são Strings) e o salário, que é do tipo Double. Como são informações pessoais, vamos declarar todas como private.

Criamos a variável 'programador', que é um objeto da classe 'Funcionario', e tentamos definir o nome do programador como "Alonso", porém o atributo 'nome' foi declarado com visibilidade do tipo 'private', ou seja, esse método é privado e somente é visto pela classe/objeto e métodos pertencentes a ela.

Como o método main é de outra classe, não é possível, nem por reza brava, acessar os elementos declarados como private.

Daí obtemos erros do tipo: The field Funcionario.nome is not visible

Obviamente, vamos comentar esse trecho do código, pois não seria possível continuar nosso programa com este erro :)

Questões 02 e 03

Nessa questão da apostila, apenas implementamos os métodos setters e getters.

A única coisa de diferente que fazemos é criar o método `getGanhoAnual()`, que retorna o valor que o funcionário ganha em um anos (12 salários e o 13o terceiro). Para testar, usamos o método `setSalario()` para colocar um valor no atributo 'salario'. Ao fazer isso, automaticamente o método `getGanhoAnual()` retornará 13 vezes o valor deste salário.

Depois preenchemos os campos de nome e setor que trabalham, através dos setters.

Depois exibimos tudo isso em um único print, usando os getters.

Questão 04

Para essa questão da apostila, vamos criar outro funcionário, o 'analista'.

Vamos inicializar ele através da criação do construtor que recebe uma String com seu nome, e dentro deste construtor apenas chamamos o método `setNome()` e passamos a String que esse construtor recebe.

Para inicializar esse tipo de objeto, mas passar uma String como argumento no: `new Funcionario(string_aqui);`

Pois dentro da implementação da classe `Funcionario` já existe um construtor que 'espera' receber uma String como argumento, então quando você manda essa String, o Java automaticamente detecta o construtor pra qual ela deve ir.

Porém, nas questões passadas não passamos nenhuma String. E como criamos

um construtor, o construtor padrão (que não recebe argumentos) não existe, por isso precisamos criar um construtor padrão, para que também possamos criar um objeto da classe Funcionario sem ter que passar argumento algum.

Questão 05

Essa questão da apostila é realmente interessante, pode ativar bastante nossa imaginação e perturbar o juízo de alguns, em busca da solução.

Nossa classe deve ter 'ciência' de todos os objetos que foram criados.

Ela deve ter um atributo que é comum à todas as classes, que é o atributo que iremos chamar de 'numFuncionarios', um inteiro que irá armazenar o número de funcionários (que é o mesmo número de objetos criados dessa Classe).

Para ele ser visto e poder ser acessado de qualquer objeto, ele deve ser do tipo 'static', ou seja, esse inteiro é comum à todos os objetos, pois ele é, na verdade, o mesmo local da memória. Assim, quando alteramos seu valor num objeto, estamos alterando para todos os outros.

Cada vez que criamos um objeto, estamos criando um funcionário. Então devemos incrementar esse atributo.

E o que acontece quando criamos um objeto? Sim, o construtor é sempre chamado.

Então, iremos fazer com que cada vez que algum construtor seja chamado, ele irá incrementar a variável 'numFuncionarios', e somente nos construtores. Assim teremos total controle do número de funcionários.

Após ser criado um funcionário, o valor de 'numFuncionarios' muda (para numFuncionarios + 1), e esse será o valor do identificador do novo funcionário.

Colocamos a variável 'identificador' como 'final'. Ou seja, agora ela é uma constante.

Então, uma vez que um funcionário tem seu identificador, esse atributo não poderá mais mudado, por isso não deve ter um setter.

Seguro e totalmente automatizado nosso aplicativo, não?

Questões 06 e 07

Vamos criar a classe Empresa com quatro atributos: uma string com o nome da empresa, um array com os funcionários (não inicializados, pois não sabemos o tamanho), e dois atributos de controle, um que irá conter a 'capacidade' de funcionários (que o usuário irá dizer) e se o número 'total' de funcionários, que armazena o número atual de funcionários.

O único construtor recebe um número inteiro, que é será a capacidade total da empresa. E esse número é usado para declarar o array de funcionários.

Agora que temos o número exato de funcionários no array, precisamos inicializar esse objetos da classe 'Funcionario'.

E para isso, vamos usar o método 'adiciona()', que recebe um objeto dessa classe.

Porém, antes de adicionar o novo funcionário, precisamos saber se tem lugar pra ele, para isso apenas checamos o número atual de funcionários , na variável 'total', e se estiver dentro da capacidade, fazemos com que um objeto do array de funcionários receba a referência deste novo objeto.

Para ilustrar, na `main()`, criamos uma empresa com capacidade de 3 funcionários.

Adicionamos os objetos 'programador' e 'analista', que já trabalhamos nas questões anterior deste capítulo da apostila.

Agora vamos criar mais um funcionário, o 'estagiario'.

Como já adicionamos 2 objetos, este vai ser o terceiro, então vai ser o elemento de índice 2 do array.

Para pegar esse objeto do array, usamos o método: `JP.getFuncionario(2)`, que vai devolver a referência do objeto de nossa empresa 'JP'.

Então, para definir o setor desse funcionário, vamos usar o método `setSetor()` dele. Fazemos isso assim:

```
JP.getFuncionario(2).setSetor("Faz tudo");
```

Para definir o salário, é análogo, mas usamos o método `setSalario()`, desse mesmo funcionário, dessa mesma empresa:

```
JP.getFuncionario(2).setSalario(0);
```

Note como fica a ordem: `empresa.funcionario.metodoDesseFuncionario()`

Sacou a lógica?

Depois é só exibir tudo.

Assim, o código solução desses 7 primeiras questões é:

Classe: ApostilaJavaCap6.java

```
public class ApostilaJavaCap6 {
```

```
public static void main(String[] args){
```

```
    Funcionario programador = new Funcionario();
```

```
    //Questão 1
```

```
    //programador.nome = "Alonso";
```

```
    //Questão 2 e 3
```

```
System.out.println("Questões 2 e 3 ->");
```

```
    programador.setSalario(2000);
```

```
    programador.setNome("Alonso");
```

```
    programador.setSetor("TI - Programador Java");
```

```
System.out.println("Nome do funcionário: " + programador.getNome() +
```

```
                    "\nSetor de trabalho: " + programador.getSetor() +
```

```
                    "\nSalário: " + programador.getSalario() +
```

```
                    "\nGanho anual: " + programador.getGanhoAtual() +
```



```
"\nID: " + programador.getIdentificador());
```

```
System.out.println("-----");
```

```
//Questão 4
```

```
Funcionario analista = new Funcionario("Bruce Dickinson");
```

```
analista.setSetor("Gestão de projetos");
```

```
analista.setSalario(3000);
```

```
System.out.println("Nome do analista: " + analista.getNome() +
```

```
"\nID: " + analista.getIdentificador());
```

```
//Questão 6 e 7
```

```
System.out.println("\n\nNa nova empresa");
```

```
Empresa JP = new Empresa(3);
```

```
JP.setNome("Empresa Java Progressivo");
```

```
JP.adiciona(programador);
```

```
JP.adiciona(analista);
```

```
JP.adiciona(new Funcionario("Estagiário"));
```

```
JP.getFuncionario(2).setSetor("Faz tudo");
```

```

        JP.getFuncionario(2).setSalario(0);

System.out.println(JP.getNome());

for(int i=0 ; i < 3 ; i++){

System.out.println("Nome do funcionário: " + JP.getFuncionario(i).getNome() +
                    "\nSetor de trabalho: " +
JP.getFuncionario(i).getSetor() +
                    "\nSalário: " + JP.getFuncionario(i).getSalario() +
                    "\nGanho anual: " +
JP.getFuncionario(i).getGanhoAtual() +
                    "\nID: " + JP.getFuncionario(i).getIdificador() +
                    "\n-----");

    }

}

}

```

Classe: Funcionario.java

public class Funcionario {

private String nome,

setor;

private double salario;

private final int identificador;

private static int numFuncionarios;

Funcionario(){

identificador = ++numFuncionarios;

}

Funcionario(String nome){

setNome(nome);

identificador = ++numFuncionarios;

}

public int getIdentificador() {

```
return identificador;
```

```
}
```

```
public Double getGanhoAtual(){
```

```
return 13 * this.salario;
```

```
}
```

```
public String getNome() {
```

```
return nome;
```

```
}
```

```
public void setNome(String nome) {
```

```
this.nome = nome;
```

```
}
```

```
public String getSetor() {
```

```
return setor;
```

```
}
```

```
public void setSetor(String setor) {
```

```
    this.setor = setor;
```

```
}
```

```
public double getSalario() {
```

```
    return salario;
```

```
}
```

```
public void setSalario(double salario) {
```

```
    this.salario = salario;
```

```
}
```

```
}
```

Classe: Empresa.java

```
public class Empresa {
```

private String nome;

private Funcionario[] empregados;

private int total=0,

capacidade;

Empresa(int numFuncionarios){

empregados = new Funcionario[numFuncionarios];

this.capacidade = numFuncionarios;

}

public Funcionario getFuncionario (int posicao) {

return this.empregados[posicao];

}

public void adiciona(Funcionario f){

```
if(this.total < this.capacidade){  
    empregados[total] = f;  
  
    this.total++;  
    }  
}
```

```
public String getNome() {
```

```
    return nome;  
    }
```

```
public void setNome(String nome) {
```

```
    this.nome = nome;  
    }
```

```
}
```

Questão 08

Essa merecia um artigo só para resolver este tipo de aplicativo, mas não é nada

de outro mundo.

Para criar uma aplicação realmente robusta e funcional, precisamos fazer uma série de testes.

Primeiros temos que checar o ano, para saber se é bissexto ou não, pois dependendo disso, o ano vai ter a data 29 de fevereiro ou não.

Depois temos que checar o mês, pois dependendo deste mês, ele vai até dia 28 ou 29 (caso seja fevereiro e dependendo se é ano bissexto), dia 30 ou 31.

Após isso, poderemos saber se o dia é válido ou não.

Tente resolver, é uma questão interessante.

A única dificuldade é criar uma lógica para saber se o ano é bissexto ou não. Em breve vamos resolver esta interessante questão da apostila.

Questão 09

Essa é parecida com a anterior, e vai exibir uma pesquisa no Google sobre 'validação de CPF', pois o Ministério da Fazenda tem regras sobre o número de CPF. Onde, a partir de um algoritmo sobre os 9 primeiros dígitos do CPF, descobrimos os dois últimos dígitos, que são os validadores.

Veja a regra aqui e tente fazer em Java:

http://www.geradorcpf.com/algoritmo_do_cpf.htm

É uma interessante questão, que certamente iremos fazer em breve.

6.9 - Solução dos desafios

Questão 01

Embora a variável 'x' esteja na mesma classe, não é possível acessá-la pois 'x' não é static, já que o método main() é estático e ele só acessa membro static também.

Questão 02

Essa questão da apostila é bem difícil de ser resolvida somente pensando, e provavelmente você vai precisar pesquisar.

Mas se quiser tentar, deverá fazer 3 procedimentos básicos e nada óbvios.

O primeiro é, dentro da classe, declarar um objeto do tipo static da PRÓPRIA CLASSE.

Isso mesmo, se sua classe se chama 'Exemplo', vamos criar um objeto chamado 'objExemplo' dela mesma:

```
private static Exemplo objExemplo;
```

Depois criamos um construtor, mas do tipo private. Criamos ele para que o padrão, que é público, não exista mais.

Assim, só existe um construtor, o private. E como é privado, não pode ser acessado fora da classe.

Ora, sempre que criamos um objeto ele chama o construtor automaticamente.

Nesse caso, não vai ser possível chamar o construtor já que é private, então NÃO SERÁ POSSÍVEL CRIAR OUTRO OBJETO desta classe!

Se não é possível criar um objeto dela, como vamos criar o objeto único dela?

Ué, já foi criado, é aquele que explicamos antes, o 'objExemplo'.

Mas esse objeto precisa ter um 'contato' com o mundo exterior, então vamos criar um método getter, para podermos passar esse objeto para quem pedir.

Vamos chamar esse método de getInstancia().

E aqui vamos fazer outra coisa: note que declaramos nosso objeto, mas não inicializamos ele.

E é nesse método que vamos inicializá-lo.

Ao invocarem o método getInstancia() ele checa se o objeto já foi inicializado (se não tiver sido, ainda aponta para null). Se ainda não tiver sido, ele inicializa dando o 'new'.

Se já tiver sido inicializado, retorna o objeto.

Pronto. Na main, não precisamos declarar nem inicializar o objeto, pois ele já existe (é estático, e está dentro da classe, foi criado automaticamente no ato da compilação).

Como prova disso, apenas acessamos o método 'hello()' desse objeto, que retorna uma string.

Sagaz, não?

Então, nosso código dessa aplicação é:

Classe: Singleton.java

```
public class Singleton {
```

```
public static void main(String[] args) {
```

```
    TestSingleton.getInstancia().hello();
```

```
    }
```

```
}
```

Jogo: Campo Minado em Java

Linha: 8

Coluna: 1

Havia uma mina ! Você perdeu!

Linhas

8	*	-	-	1	2	1	2	*
7	-	-	-	*	2	*	2	1
6	-	2	2	1	2	1	1	0
5	*	*	1	0	0	0	0	0
4	-	4	3	1	0	0	0	0
3	-	*	*	1	0	0	0	0
2	2	3	3	1	0	1	1	1
1	1	*	1	0	0	1	*	1

1 2 3 4 5 6 7 8

Colunas

Dando continuidade aos games que estamos fazendo, e ensinando a fazer, vamos agora mostrar como fazer e programar o famoso jogo Campo Minado em Java.

Aqui mostraremos as classes e código do jogo, no próximo artigo vamos explicar em detalhes como programar o jogo.

Como jogar o nosso Campo Minado em Java

Existe um tabuleiro 8x8, onde estão localizadas 10 minas.

A cada rodada você irá fornecer o número da linha e da coluna (ou seja, números de 1 até 8).

Caso exista uma mina naquele lugar, você perde o jogo.

Caso não exista uma mina naquela posição, será exibido número naquele local e números nos locais vizinhos aquele que você escolheu, exceto onde há minas.

Esses números informam quantos minas existem ao redor daquele local.

Por exemplo, se você escolhe um local e aparece o número '2' lá, é porque existem duas minas nas vizinhanças daquele local. Vale salientar que 'ao redor' e 'vizinhança' significam todos os blocos ao redor, incluindo na diagonais.

Objetivo: Deixar os 10 campos que possuem minas livres. Ou seja, onde você deduzir que existe mina, não marque, simplesmente deixe o '_' lá, pois quando existem 10 underlines você ganhará o jogo.

Código do jogo Campo Minado em Java

-->campoMinado.java

```
public class campoMinado {
```

```
    public static void main(String[] args) {
```

```
        Jogo jogo = new Jogo();
```

```
    }
```

```
}
```

-->Jogo.java

```
public class Jogo {
```

```
private Tabuleiro board;
```

```
boolean terminar = false;
```

```
boolean ganhou = false;
```

```
int[] jogada;
```

```
int rodada=0;
```

```
public Jogo(){
```

```
    board = new Tabuleiro();
```

```
    Jogar(board);
```

```
    jogada = new int[2];
```

```
}
```

```
public void Jogar(Tabuleiro board){
```

```
do{
```

```
    rodada++;
```

```
System.out.println("Rodada "+rodada);
```

```
    board.exibe();
```

```
    terminar = board.setPosicao();
```

```
if(!terminar){
```

```
    board.abrirVizinhas();
```

```
    terminar = board.ganhou();
```

```
}
```

```
    }while(!terminar);
```

```
if(!board.ganhou()){
```

```
System.out.println("Havia uma mina ! Você perdeu!");
```

```
    board.exibeMinas();
```

```
} else {
```

```
System.out.println("Parabéns, você deixou os 8 campos de minas livres em  
"+rodada+" rodadas");
```

```
    board.exibeMinas();
```

```
}
```

```
}
```

```
}
```

-->Tabuleiro.java

```
import java.util.Random;
```

```
import java.util.Scanner;
```

```
public class Tabuleiro {
```

```
private int[][] minas;
```

```
private char[][] tabuleiro;
```

```
private int linha, coluna;
```

```
Random random = new Random();
```

```
Scanner entrada = new Scanner(System.in);
```

```
public Tabuleiro(){
```

```
    minas = new int[10][10];
```

```
    tabuleiro = new char[10][10];
```

```
        iniciaMinas(); // coloca 0 em todas as posições do tabuleiro de minas  
        sorteiaMinas(); //coloca, aleatoriamente, 10 minas no tabuleiro de minas  
        preencheDicas();//preenche o tabuleiro de minas com o número de minas  
vizinhas  
        iniciaTabuleiro();//inicia o tabuleiro de exibição com _  
  
    }
```

```
public boolean ganhou(){
```

```
int count=0;
```

```
for(int line = 1 ; line < 9 ; line++)
```

```
for(int column = 1 ; column < 9 ; column++)
```

```
if(tabuleiro[line][column]=='_')
```

```
    count++;
```

```
if(count == 10)
```

return true;

else

return false;

}

public void abrirVizinhas(){

for(int i=-1 ; i<2 ; i++)

for(int j=-1 ; j<2 ; j++)

if((minas[linha+i][coluna+j] != -1) && (linha != 0 && linha != 9 && coluna != 0 && coluna != 9)){

tabuleiro[linha+i][coluna+j]=Character.forDigit(minas[linha+i][coluna+j], 10);

}

}

public int getPosicao(int linha, int coluna){

```
return minas[linha][coluna];
```

```
}
```

```
public boolean setPosicao(){
```

```
do{
```

```
System.out.print("\nLinha: ");
```

```
    linha = entrada.nextInt();
```

```
System.out.print("Coluna: ");
```

```
    coluna = entrada.nextInt();
```

```
if( (tabuleiro[linha][coluna] != '_') && ((linha < 9 && linha > 0) && (coluna < 9 && coluna > 0)))
```

```
System.out.println("Esse campo já está sendo exibido");
```

```
if( linha < 1 || linha > 8 || coluna < 1 || coluna > 8)
```

```
System.out.println("Escolha números de 1 até 8");
```



```
        }while((linha < 1 && linha > 8) && (coluna < 1 && coluna > 8) ||  
(tabuleiro[linha][coluna] != '_') );
```

```
if(getPosicao(linha, coluna)== -1)
```

```
return true;
```

```
else
```

```
return false;
```

```
}
```

```
public void exhibe(){
```

```
System.out.println("\n  Linhas");
```

```
for(int linha = 8 ; linha > 0 ; linha--){
```

```
System.out.print("    "+linha + " ");
```

```
for(int coluna = 1 ; coluna < 9 ; coluna++){
```

```
    System.out.print(" "+ tabuleiro[linha][coluna]);
```

```
    }
```

```
System.out.println();
```

```
}
```

```
System.out.println("\n      1 2 3 4 5 6 7 8");
```

```
System.out.println("      Colunas");
```

```
}
```

```
public void preencheDicas(){
```

```
    for(int line=1 ; line < 9 ; line++)
```

```
    {
        for(int column=1 ; column < 9 ; column++){
```

```
for(int i=-1 ; i<=1 ; i++)
```

```
for(int j=-1 ; j<=1 ; j++)
```

```
if(minas[line][column] != -1)
```

```
if(minas[line+i][column+j] == -1)
```

```
minas[line][column]++;
```

```
}
```

```
}
```

```
public void exhibeMinas(){
```

```
for(int i=1 ; i < 9; i++)
```

```
for(int j=1 ; j < 9 ; j++)
```

```
if(minas[i][j] == -1)
```

```
tabuleiro[i][j]='*';
```

```
    exibe();  
}
```

```
public void iniciaTabuleiro(){
```

```
    for(int i=1 ; i<minas.length ; i++)
```

```
    for(int j=1 ; j<minas.length ; j++)
```

```
        tabuleiro[i][j]= '_';
```

```
    }
```

```
public void iniciaMinas(){
```

```
    for(int i=0 ; i<minas.length ; i++)
```

```
    for(int j=0 ; j<minas.length ; j++)
```

```
        minas[i][j]=0;
```

```
    }
```

```
public void sorteiaMinas(){
```

boolean sorteado;

int linha, coluna;

for(int i=0 ; i<10 ; i++){

do{

 linha = random.nextInt(8) + 1;

 coluna = random.nextInt(8) + 1;

if(minas[linha][coluna] == -1)

 sorteado=true;

else

 sorteado = false;

 }while(sorteado);

 minas[linha][coluna] = -1;

 }

}

}

Campo Minado em Java: código comentado

Dividimos o jogo em 3 classes, a 'campoMinado' que simplesmente contém a main e cria um objeto do tipo 'Jogo', temos a classe 'Jogo' que irá ministrar toda a lógica e fluxo do jogo e finalmente a classe 'Tabuleiro', que irá gerar os tabuleiros (de minas e visualização) bem como os métodos que lidam com os tabuleiros.

Campo Minado em Java: A classe Tabuleiro.java

Essa classe é a responsável pelo tabuleiro e os métodos que o envolvem.

Na verdade, vamos lidar com dois tabuleiros:

- `int[][] minas`

Esse tabuleiro é inicialmente preenchido com números '0', através do método '`iniciaMinas()`'.

Após isso, vamos sortear 10 locais para colocar as minas, através do método '`sorteiaMinas()`'. Nos locais onde existirem minas, vamos colocar o inteiro '-1'.

Esse método sorteia dois números inteiros, entre 1 e 8, através do uso da classe `Random`. Vamos fazer isso com um looping do `while`, que a cada iteração vai checar se naquela posição sorteada já existe o número '-1'. Se aquele local já tiver sido sorteado, o booleano '`sorteado`' ficará como `true` e o looping irá se repetir, até que tenhamos 10 locais diferentes com o número '-1'.

Depois disso, vamos preencher as dicas, ou seja, vamos colocar em cada bloco desse tabuleiro o número de minas que existem ao redor.

Assim, se uma posição do tabuleiro tiver o número '2', é porque existem duas minas ao redor daquele bloco.

Fazemos isso apenas contando quantas bombas existem ao redor de cada bloco.

Criamos, na verdade, uma matriz 10x10, onde não usaremos a linha 0 nem a linha 9, bem como a coluna 0 e a coluna 9. Por que isso?

Para que, ao calcular quantas bombas existem na vizinhança, apenas contemos quantas bombas existem nos 8 locais ao redor. Isso é necessário pois se usássemos um tabuleiro 8x8, as casas da borda não teriam 8 vizinhos.

Para checar as bombas ao redor, usamos dois laços for:

```
for(int i=-1 ; i<=1 ; i++)
```

```
for(int j=-1 ; j<=1 ; j++)
```

Se queremos checar quantas bombas há ao redor do bloco: `mines[linha][coluna]`, colocamos dentro desses laços:

```
mines[linha+i][coluna+j]
```

Esses dois laços irão percorrer os 8 locais ao redor do bloco `mines[linha][coluna]`.

Porém, só vamos checar os arredores se o local em questão não tiver uma mina:

```
if(minas[line][column] != -1)
```

Após checado isso, checamos se existe mina na posição ao redor, e se tiver, incrementamos o local do tabuleiro 'minas', pois ele também recebe o número de

minas ao redor:

```
if(minas[line+i][column+j] == -1)
```

```
minas[line][column]++;
```

Pronto. Agora temos um tabuleiro 10x10, mas usaremos só o 8x8, e nesse tabuleiro 'interno' temos 10 minas e todos os blocos que não são minas armazenarão quantas minas existem ao seu redor.

- char[][] tabuleiro

É um tabuleiro de caracteres, inicialmente carregado com underline '_' através do método 'iniciaTabuleiro()', isso quer dizer que esse campo ainda não foi escolhido.

Temos ainda o método 'exibe()', que simplesmente exibe todo esse tabuleiro de caracteres de maneira formatada.

O método 'exibeMinas()' coloca um asterisco, '*', em todos os locais onde existem minas. Este método serve para mostrar onde existia minas e será acionado quando o jogador perder a partida.

Fazendo uma jogada:

A jogada é feita através do método 'setPosicao()', que retorna 'true' caso você perca (ou seja, caso exista uma mina onde você jogou) e 'false', caso não exista uma mina no local que você escolheu.

Devemos deixar esse método bem robusto, nos certificando que o jogador não entre com números fora do esperado (ou seja, maior que 8 ou menor que 1):

(linha < 1 || linha > 8 || coluna < 1 || coluna > 8)

Bem como checar se já o local ele jogou já tenha sido escolhido antes.

(tabuleiro[linha][coluna] != '_')

Ainda no método 'setPosicao()', usamos o método 'getPosicao()', que retorna o que existe no bloco quando passamos a linha e a coluna. Caso exista '-1' é porque existe mina, retorna 'true' e o jogo acaba. Caso exista qualquer outro número, o método retorna 'false' e o jogo não acaba.

Após jogar, vamos checar se o jogador ganhou através do método 'ganhou()', que simplesmente conta quantos blocos ainda não foram exibidos, ou seja, quantos underlines existem. Caso existam somente 10, é porque os blocos que sobraram foram justamente os que continham minas, o método retorna 'true' e o jogador ganha. Caso tenha mais de 10, é porque ainda não terminou e retorna 'false'.

Outro método importante, e talvez o mais complicado, é o 'exibirVizinhas()', que vai checar todas as casas vizinhas, no tabuleiro de minas, exceto se essas vizinhas estejam localizadas nas linhas 0 ou 9, ou nas colunas 0 ou 9. Durante a checagem, vamos checar se a vizinha possui uma mina, e caso não possua vamos exibir essa casa no tabuleiro de caracteres:

```
for(int i=-1 ; i<2 ; i++)
```

```
for(int j=-1 ; j<2 ; j++)
```

```
if( (minas[linha+i][coluna+j] != -1) && (linha != 0 && linha != 9 && coluna != 0 && coluna != 9) )
```

```
tabuleiro[linha+i][coluna+j]=Character.forDigit(minas[linha+i][coluna+j], 10);
```

Note que usamos o método 'Character.forDigit(int num,int base)', que recebe um inteiro que representa um número e sua base.

O que esse método faz é pegar um número e transformar em caractere. No nosso caso, os número estão na base decimal (por isso o 10) e esses números nada mais são que os inteiros do tabuleiros de minas. Ou seja, estamos colocando números no tabuleiro de caracteres.

Campo Minado em Java: A classe Jogo.java

Essa classe começa com o construtor padrão criando um tabuleiro, o 'board' e chama o método que irá controlar o jogo, o 'Jogar', que recebe o objeto 'board', do tipo Tabuleiro.

O jogo, como de praxe, é controlado por um laço do while, que só termina quando o booleano 'terminar' for verdadeiro.

Esse booleano só se torna 'true' se o método 'setPosicao()' retornar true, dizendo que encontramos uma mina, ou quando o método 'ganhou()' retornar true, dizendo que o jogador ganhou, pois só restavam 10 blocos no jogo.

Usamos o inteiro 'rodada', que irá contar quantas rodadas o jogo teve.

A jogada é feita em:

```
terminar = board.setPosicao();
```

Caso não tenhamos atingido uma mina:

```
if(!terminar)
```

Vamos abrir os blocos vizinhos desse bloco que escolhemos na 'setPosicao()' para ver as dicas, isso ocorre em:

```
board.abrirVizinhas();
```

Após aberto as casas vizinhas, pode ser que o jogador tenha ganhado, fazemos isso checando:

```
terminar = board.ganhou();
```

Quanto o jogo terminar, ou seja, quando o aplicativo sair do laço do while, vamos mostrar a mensagem de parabéns ou de perda.

Quando o método 'ganhou()' retorna true é porque o jogador ganhou, então exibimos uma mensagem de parabéns bem como em quantas rodadas ele venceu:

```
if(board.ganhou()){
```

```
System.out.println("Parabéns, você deixou os 8 campos de minas livres em  
"+rodada+" rodadas");
```

```
board.exibeMinas();
```

```
}
```

Ora, se não retorna true no condicional if, o else vai ser o caso em que ele tenha perdido. Então dizemos que ele perdeu e mostramos onde havia minas através do método 'exibeMinas()'.

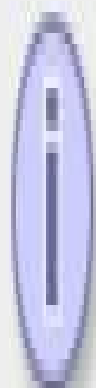
Programação Gráfica em Java, parte I: Caixas de Diálogo

Programação gráfica em Java, GUI e 2D: Introdução

Vamos dar início em nossa apostila online ao uso de elementos gráficos em Java, e vamos iniciar com as tão famosas caixas de diálogo, que são aquelas caixinhas com mensagens e botões, que geralmente vemos com algum aviso de erro ou alerta.



Message



Programação Gráfica - Curso Java Progressivo!



Programação Gráfica, GUI e desenhos 2D em Java

Quando pensamos em programar logo pensamos em criar um jogo em 3D super moderno que exige o máximo de uma placa de vídeo ou em criar um sistema operacional bem seguro.

Mas não importa a linguagem que estudemos, sempre começamos com algumas 'bobagens' básicas e não vemos nada gráfico, só janelas pretas (MS-DOS ou Terminal-Linux) ou aquela janelinha onde aparecem o resultado dos programas nas IDEs, como no NetBeans ou no Eclipse.

Isso não é à toa ou por conta de cursos ou livros mal feitos. Faz parte. Tem que começar por essas coisas mais simples mesmo, como fizemos em nosso curso Java Progressivo.

Para criar aplicações com menus, botões e janelas tem que começar por coisas mais simples e muitas vezes tidas como 'bobas' e inúteis.

Porém, se chegou aqui, já está na hora de ver algumas ferramentas gráficas em ação. De fazer algumas janelas, ver alguns botões e caixas de diálogo.

Ao longo do curso você verá mais dos chamados elementos GUI - Graphics User Interface, ou interface gráfica do usuário.

Mas não se engane. Isso é somente uma 'facilidade' para o usuário.

Na verdade, para o programador, isso só complica. Vai ter que programar do mesmo jeito, imaginar as besteiras que usuário pode fazer e tentar minimizar esses problemas além de prever os possíveis atos do cliente.

Verá que por trás de um clique existe um grande esquema de código que irá tratar o evento, que pode ser um clique, ou algum botão que foi pressionado. E existem muitos botões e muitos locais que podem ser clicados.

Exibindo mensagens através das caixas de diálogo

Caixas de diálogo, ou Dialog Box, são os elementos gráficos mais simples. Mas não menos importantes, e certamente os mais comuns que vemos.

São vistos como mensagens de erros e de alertas nos sistemas operacionais.

Vamos mostrar como declarar e exibir mensagens através das dialog boxes.

Caixas de diálogo em Java

Para usarmos as caixas de diálogo precisamos importar uma classe do pacote (package) javax.swing, que é um pacote que usaremos bastante em nossas aplicações gráficas Java.

Essa classe é estática e se chama JOptionPane.

Se acostume com esses nomes começados com J: JLabel, JButton, JTextField...

A importação, portanto, é:

```
import javax.swing.JOptionPane;
```

Exibindo mensagens nas caixas de diálogos

Para exibir mensagens em uma dialog box, usaremos o método `showMessageDialog`, que recebe dois argumentos.

O primeiro se refere à posição da caixa de diálogo (por hora, não nos preocuparemos com ele e enviaremos o argumento 'null') e o segundo é a string que será exibida.

Um exemplo completo de seu primeiro programa gráfico em Java é:

```
import javax.swing.JOptionPane;
```

```
public class grafico {
```

```
public static void main(String[] args){
```

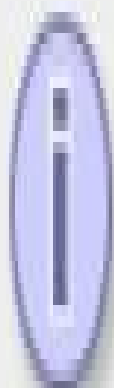
```
    JOptionPane.showMessageDialog(null,"Meu primeiro programa gráfico!\n  
    Obrigado, Curso Java Progressivo!");
```

```
    }
```

```
}
```



Message



Meu primeiro programa gráfico!
Obrigado, Curso Java Progressivo!



Note que não precisamos declarar um objeto da classe JOptionPane, pois a classe é static.

Mais a frente, quando explicarmos melhor Orientação a Objetos e classes, saberá melhor o que isso quer dizer.

Note que podemos exibir uma string tanto do jeito que foi mostrado no programa, ou declarando uma variável do tipo String 'mensagem' e usando:

```
JOptionPane.showMessageDialog(null,mensagem);
```

Por exemplo, para o usuário entrar com uma string através de um objeto 'entrada' da classe 'Scanner', usamos o comando:

```
mensagem = entrada.nextLine();
```

Então, um aplicativo que recebe uma string do usuário e mostra na caixa de diálogo ficaria:

```
import javax.swing.JOptionPane;
```

```
import java.util.Scanner;
```

```
public class caixasDeTexto {
```

```
public static void main(String[] args){
```

```
String mensagem;
```



```
Scanner entrada = new Scanner(System.in);
```

```
System.out.print("Digite a mensagem a ser exibida na caixa de diálogo: ");
```

```
    mensagem = entrada.nextLine();
```

```
    JOptionPane.showMessageDialog(null,mensagem);
```

```
    }
```

```
}
```

Note que, por padrão, nesses exemplos mais simples o Java mostra um botão de 'OK' e as opções de minimizar e fechar, além de um ícone do Java e outro de exclamação.

Tudo isso com pouquíssimo código.

Isso é uma vantagem do Java, a programação gráfica. Podemos facilmente alterar esses ícones e colocar mais botões e funções, como veremos ao longo do curso.

Recebendo dados do usuário através das caixas de diálogo

Vimos no artigo passado que as caixas de diálogo - ou dialog box - são muito usadas para exibir mensagens (principalmente de erros e alertas), e mostramos como fazer isso.

Porém, também podem ser usadas (e vamos mostrar como fazer isso) para receber dados do usuário, através de uma interface mais amigável e gráfica :)

Como usar as caixas de diálogo para receber dados do usuário em Java

Para usarmos as caixas de diálogo precisamos importar uma classe do pacote (package) `javax.swing`, que é um pacote que usaremos bastante em nossas aplicações gráficas Java.

Essa classe é estática e se chama `JOptionPane`.

Se acostume com esses nomes começados com J: `JLabel`, `JButton`, `TextField`...

A importação, portanto, é:

```
import javax.swing.JOptionPane;
```

Recebendo informações do usuário através das caixas de diálogo

É uma espécie de classe 'Scanner', onde o usuário irá digitar algo e essa informação será atribuída a uma string.

Quando mostramos informações, usamos o método `showMessageDialog`.

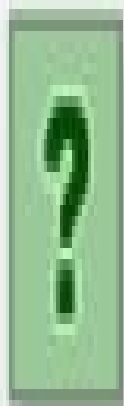
Agora usaremos o método `showInputDialog`.

Note que não faz sentido simplesmente aparecer uma caixa de diálogo que recebe dados do usuário. É preciso que essa Dialog Box exiba alguma informação, como 'Qual seu nome?', 'Login', 'Senha' etc.

No caso do método `showInputDialog`, ele irá receber uma string (digitada pelo usuário) e exibirá uma mensagem na caixa de diálogo, essa mensagem será digitada entre parênteses do método `showInputDialog` e tudo que será digitado pelo usuário será armazenado na string 'nome', pois iremos perguntar o nome do usuário.



Input



Digite seu nome, caro usuário.

Lennogildo

OK

Cancel

Logo após, iremos exibir o nome do usuário. Como exibir?

Ora, através de uma caixa de diálogo que exiba um texto, isso já aprendemos na aula passada.

Nosso aplicativo que pergunta o nome do cliente e exibe uma mensagem gentil é:

```
import javax.swing.JOptionPane;
```

```
public class recebe_dados {
```

```
public static void main(String[] args){
```

```
String nome;
```

```
    nome = JOptionPane.showInputDialog("Digite seu nome, caro usuário.");
```

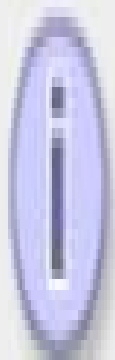
```
    JOptionPane.showMessageDialog(null,nome + "???\nNossa, que nome feio!\n\nPelo menos sabe programar em Java!");
```

```
    }
```

```
}
```




Message



Lennogildo???

Nossa, que nome feio!

Pelo menos sabe programar em Java!



Um método útil da classe String é o format, que formata a string, ou seja, altera, muda a string para o jeito que você quiser.

No nosso caso, seria útil usar esse método assim:

```
nome = String.format(nome + "???\nNossa, que nome feio! \nPelo menos sabe programar em Java!");
```

E usar a string 'nome' na JOptionPane.showMessageDialog assim:

```
JOptionPane.showMessageDialog(null,nome);
```

No próximo artigo mostraremos como transformar as informações, que são do tipo String, em tipos inteiro, float, double etc.

Como passar variáveis do tipo String para int, float e double

Nos dois artigos passados aprendemos como exibir mensagens e receber dados a partir das caixas de diálogos.

Porém, notamos que as caixas de diálogos fazem tudo isso através do tipo String.

Mas palma, palma, palma. Não criemos cânicos. Pois podemos transformar o que foi digitado como string em inteiro, float, double...

Transformando string em inteiro em Java

Suponha que, através do método `showInputDialog` visto no artigo passado, tenhamos pedido um número ao usuário.

Ok. O usuário pediu, mas este foi armazenado numa `String` `'numeroString'`.

Para passar essa string para inteiro, vamos usar o método da classe `String` chamado `parseInt`, da classe estática `Integer`.

Suponha que vamos armazenar esse valor inteiro em `'numInteiro'`, ficaria assim:

```
numInteiro = Integer.parseInt(numeroString);
```

Como exemplo, vamos criar um aplicativo que peça dois inteiros ao usuário e mostre a soma como resultado:

```
import javax.swing.JOptionPane;
```

```
public class somaGrafica {
```

```
public static void main(String[] args){
```

```
String valor, resultado;
```

```
int num1, num2;
```

```
valor = JOptionPane.showInputDialog("Digite o primeiro valor inteiro");
num1 = Integer.parseInt(valor.trim());

valor = JOptionPane.showInputDialog("Digite o segundo valor inteiro");
num2 = Integer.parseInt(valor.trim());

resultado=String.format("Resultado: %d", num1+num2);
JOptionPane.showMessageDialog(null,resultado);
}

}
```

A maioria dos cursos, tutoriais e livros por aí simplesmente se daria por satisfeito em ensinar como:

```
num1 = Integer.parseInt(valor);
```

Sem esse '.trim()'

Faça um teste. Digite '1 ', isso mesmo, com um espaço antes ou depois.

Você terá um baita erro. E aí?

'Que programa ruim!'

Pensaria qualquer usuário que, sem querer, colocasse um espaço e visse essa mensagem de erro.

O curso Java Progressivo tem por objetivo ensinar você a criar aplicações reais, coisas que aprendi na prática e nenhum site me disse nem vi nos livros.

Esse `.trim()` retira os espaços de uma string, pois ESPAÇOS NÃO INTERESSAM EM UM NÚMERO!

Com o `.trim()` você deixa seu aplicativo mais robusto, à prova de bobagens dos usuários! E acredite, eles são bem criativos!

Transformando string em float

```
numFloat = Float.parseFloat(numeroString);
```

Transformando string em double

```
numDouble = Double.parseDouble(numeroString);
```

Exercício:

Propus e fiz um aplicativo que recebe os coeficientes de uma equação do segundo grau e retorna suas raízes, até as complexas, confira:

<http://www.javaprogressivo.net/2012/08/aplicativo-determina-as-raizes-de-uma.html>

Depois propus, nos exercícios de métodos, que fizesse o mesmo, mas quebrando ele em métodos, pra ficar reutilizável:

<http://www.javaprogressivo.net/2012/09/questoes-envolvendo-metodos-java.html>

Agora faça o mesmo. Um programa que receba os coeficientes de uma equação do segundo grau e retorne suas raízes, mesmo as complexas, mas através de caixas de diálogos.

Note que esse é realmente um programa útil.

Na próxima seção vamos ensinar como fazer seus aplicativos se tornarem 'executáveis' para que você possa mandar programas (gráficos, claro) para seus amigos!

Aplicativo gráfico: mostra as raízes de uma equação do segundo grau

Exercício:

Faça um programa que receba os coeficientes de uma equação do segundo grau e retorne suas raízes, mesmo as complexas, através de caixas de diálogo.



Message



As raízes são:

$$-0.50 + 0.87i$$

$$-0.50 - 0.87i$$



Programa em Java

Passo 1:

Primeiro o programa recebe os dados do usuário, que são os 3 coeficientes da equação, 'a', 'b' e 'c':

$$ax^2 + bx + c = 0$$

Passo 2:

Uma equação do 2o grau só é do segundo grau se 'a' for diferente de 0.

Caso o usuário entre com o valor 0 o programa entrará em um loop até que o usuário entre com um valor diferente de 0.

Passo 3:

Se delta for maior ou igual a 0 as raízes serão reais e são facilmente calculadas pela fórmula de Bháskara.

Passo 4:

Caso não seja, as raízes serão complexas.

Para calcular as raízes complexas fazemos delta ser positivo (-delta), tiramos sua raiz e calculamos separadamente a parte real e a parte imaginária.

XXX

```
import javax.swing.JOptionPane;
```

```
public class bhaskara_dialogBoxes {
```

```
public static float delta(float a, float b, float c){
```

```
return (b*b - 4*a*c);
```

```
}
```

```
public static void main(String[] args){
```

```
String valor, raizes;
```



```
float a=0, b, c,    //coeficientes  
delta,            //delta  
sqrtdelta,        //raiz quadrada de delta  
raiz1,raiz2;      //raízes
```

```
//PASSO 1: recebe os coeficientes
```

```
valor = JOptionPane.showInputDialog("Digite o valor de a");  
a = Float.parseFloat(valor.trim());
```

```
valor = JOptionPane.showInputDialog("Digite o valor de b");  
b = Float.parseFloat(valor.trim());
```

```
valor = JOptionPane.showInputDialog("Digite o valor de c");  
c = Float.parseFloat(valor.trim());
```

```
//PASSO 2: checa se 'a' não é 0
```

```
while(a==0){
```

```
if(a==0){
```

```
    valor = JOptionPane.showInputDialog("'a' não pode ser 0. Insira outro
```

```
valor");  
  
    a = Float.parseFloat(valor.trim());  
  
    }  
  
}
```

//PASSO 3: checa se o delta é positivo. Se for, mostra as raízes reais

```
if(delta(a,b,c)>=0){  
  
    raiz1 = (-b + (float)Math.sqrt(delta(a,b,c)))/(2*a);  
    raiz2 = (-b - (float)Math.sqrt(delta(a,b,c)))/(2*a);  
  
    raizes = String.format("As raízes são: %.2f e %.2f", raiz1,raiz2);  
  
    JOptionPane.showMessageDialog(null,raizes);  
  
}
```

//PASSO 4: caso seja negativo, devemos formatar para a exibição para números complexos

```
    } else {  
  
        raizes = String.format("As raízes são:\n" +  
                                "%.2f + %.2fi\n" +
```

```
"%.2f - %.2fi",(-b/(2*a)), ( (float)Math.sqrt(-delta(a,b,c) ) / (2*a) )  
    ,(-b/(2*a)), ( (float)Math.sqrt(-delta(a,b,c) ) / (2*a) )  
    );
```

```
JOptionPane.showMessageDialog(null,raizes);
```

```
}
```

```
}
```

```
}
```

Construindo (build) seu projeto Java no NetBeans

Ok, você criou sua aplicação gráfica, já sabe fazer vários aplicativos e tal.


Mas você quer mostrar ao mundo seus programas, como fazer? Vai mandar a pasta como o projeto pros outros? Vai mandar eles baixarem o NetBeans pra rodarem seus programas em Java?

Claro que não. Você tem que fazer o build e enviar o seu programa prontinho pra ser usado para o mundo.

Transformando seu programa em Java em executável

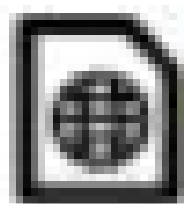
Já notou que sempre vai criar um projeto no NetBeans você escolhe um local e um nome, no campo 'Project Location'?

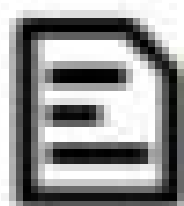
Já teve a curiosidade de ir lá e dar uma fuçada nos arquivos que existem e são criados pro funcionamento de seus aplicativos Java?

 build

 nbproject

 src

 build.xml

 manifest.mf

Sabia que, diariamente, você usa dezenas de aplicativos Java?

E quantas vezes viu essas pastas e arquivos com extensão .java, .xml e .class?

Ora, se nunca precisou usar, não vai precisar ficar mandando isso pros outros.

Vou mostrar agora uma, das várias maneiras, de uma pessoa, sem NetBeans, rodar seus aplicativos Java.

Clean and Build Project no NetBeans

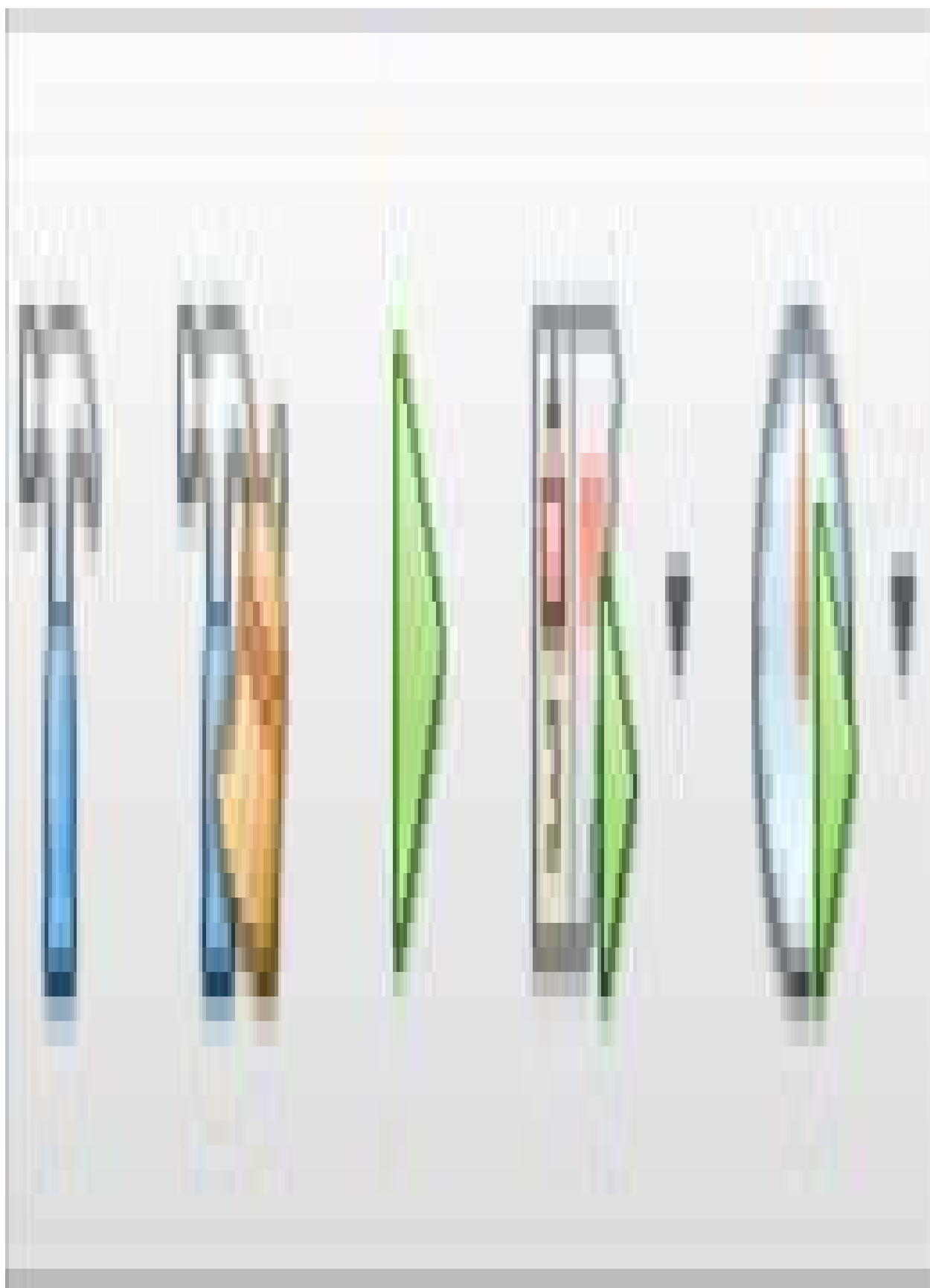
Abra seu projeto. Por exemplo, o aplicativo do exemplo passado, que calcula pra você as raízes de uma equação do segundo grau de modo totalmente gráfico.

Perfeito pra mandar pros seus colegas ou irmão.

Sabe o botão de play que você clica para rodar os programas que você aqui do curso Java Progressivo?

Já se perguntou para que serve para aqueles outros botões ao redor?

Pois é, vamos usar o da 'vassoura' com 'martelo', que fica ao lado, na esquerda:



Essa opção vai 'limpar' e 'construir' seu aplicativo (Clean and Build Main Project), ou seja, vai ajeitar, juntar as classes, figuras e tudo mais que você tenha usado em sua aplicação e colocar tudo em um único arquivo de extensão .jar

Provavelmente você já viu arquivos com extensão .jar quando baixou jogos ou aplicativos para celular.

São os mesmos, mas agora você vai criar um :)

Dentro desse arquivo, vão estar todos os aparatos para uma pessoa que tenha o JRE possa rodar seu aplicativo Java.

Esse arquivo estará localizado dentro da pasta/diretório 'dist' que foi criado durante o processo de building.

Vá lá e clique no .jar, isso deve fazer com que seu programa rode sem o NetBeans.

Agora mande e venda seus aplicativos, fique rico e não esqueça do Java Progressivo :)

Estrutura de Dados, parte I: Array e ArrayList

Introdução aos tipos de Estrutura de Dados em Java

Até o momento, em nossa apostila online Java Progressivo, vínhamos declarando os tipos um por um, notou?

Talvez não, pois é algo normal para você.

Imagine a seguinte situação: você foi contratado para criar um aplicativo Java para um colégio, de mais de 2mil alunos.

Entre outras coisas, vai ter que criar tipos - float, provavelmente - , para armazenar as notas dos alunos, inteiros para armazenar as faltas, strings para os nomes etc.

E aí, vai declarar milhares de variáveis? Uma por uma?

Não! Aprenda estrutura de dados.

Estrutura de dados em Java

Uma das grandes utilidades dos programas, em Java por exemplo, é poder tratar grandes quantidades de informações.

Já imaginou quantos floats, inteiros e strings são declarados nos softwares do governo brasileiro?

Milhões? Claro que não. Milhões é só o número de habitantes.

Cada habitante tem seu nome, cpf, RG, endereço, impostos, ficha criminal, de saúde, de emprego etc etc.

Já pensou como manipular tudo isso?

Até o momento, no curso Java Progressivo, estávamos declarando variável por variável. Obviamente que em uma aplicação como a do governo isso é humanamente impossível.

Aliás, nem precisa ser de um governo. Os dados de uma empresa de médio porte já são grandes o suficiente para não ser possível a criação de cada variável 'manualmente'.

Para isso, usamos meios especiais, as estrutura de dados, como os arrays (também conhecidos como vetores), arraylists, listas, filas e pilhas.

Estrutura de dados em Java: Array e ArrayList

Nesta seção de estrutura de dados, vamos dar uma introdução as tão importantes ferramentas para se trabalhar com muitos dados e informações.



De uma maneira simples, podemos definir os Array - ou vetor- e ArrayList como um conjunto de variáveis do mesmo tipo.

Em vez de declarar um tijolo por vez, vamos usar Arrays para declarar uma parede, com vários blocos de tijolos iguais, uma única vez.

Por exemplo, em vez de declararmos vários inteiros, um por um, até termos declarado 50 variáveis inteiras, vamos usar Array para declarar um bloco de 50 variáveis, todas do mesmo tipo e enumeradas de forma seqüencial.

Array é simplesmente isso. Um bloco de variáveis do mesmo tipo enumeradas em seqüência.

ArrayList são Arrays mais potentes, ou seja, são Arrays mas com utilidades (e complexidade) a mais.

Não entraremos mais em detalhes sobre Array e ArrayList pois esta seção é dedicada ao seu estudo.

Vamos falar um pouco sobre as outras estruturas de dados, que serão tratadas ao longo do curso, em outras seções do site:

Estrutura de dados em Java: Lista



Imagine uma lista, uma lista qualquer. A lista telefônica, por exemplo.

Ela é um registro (e que registro) de dados.

Como funciona? Como você usa?

Não há meio certo para se usar. Você pode usar o começo da lista, o meio ou fim, ir e voltar.

É simplesmente uma lista, uma exposição de informação ao seu dispor.

Assim será em Java.

Estrutura de dados em Java: Fila



Imagine uma fila de banco.

Como ela funciona? (Na verdade, como ela deveria funcionar)

De qualquer jeito, como uma 'lista'?

Claro que não. Na fila, quem estiver na frente é atendido primeiro.

Assim será na programação Java para estrutura de dados.

Quando seus dados estiverem sob essa estrutura, as informações que estão na cabeça da fila serão tratadas primeiro, assim como as pessoas que estão na cabeça de uma fila de banco seriam atendidas primeiro.

Estrutura de dados em Java: Pilha

Agora imagine uma pilha de pratos.

Você comeu uma lasanha, e guardou o prato.

Depois uma macarronada, e guardou o prato.

Por fim, você comeu uma salada e guardou o prato.

Agora você tem uma fila de pratos, que vai levar para a pia para lavar.

Qual prato vai lavar primeiro? O primeiro, o da lasanha?

Não! O último, o da salada, que está no topo da pilha.

Assim será em seus aplicativos em Java: nas informações sob estrutura de pilha, as últimas informações (último prato) que chegaram - ou seja, as mais recentes-, serão os primeiros a serem tratadas (primeiros a serem lavados).

Ou seja, é o oposto das filas.

Aqui, os últimos que chegaram serão atendidos. Nas filas, os primeiros que chegaram é que são atendidos.

Com isso, você já tem uma noção sobre estrutura de dados em Java e que há várias maneiras de se ver e de se tratar informações.

Arrays: como declarar, usar e acessar os elementos de um array

Nesse artigo introdutório de Arrays de nossa apostila Java Progressivo, você irá aprender como declarar e usar os Arrays, ou vetores, em Java.

Aprenderá, também, a importante notação dos colchetes, [], que vamos usar para tratar dos membros de um Array, bem como acessar os elementos através desses colchetes.

Declarando Arrays(vetores) em Java

A sintaxe da declaração e criação do array é a seguinte:

```
tipo[] nome_do_array = new tipo[numero_de_elementos];
```

Ou:

```
tipo[] nome_do_array = { valor1, valor2, ...,valorx};
```

Nesse último caso, o tamanho o array é o tamanho de elementos entre chaves {}.

Por exemplo, se em uma sala de aula tiver 20 alunos e você quiser declarar 20 variáveis do tipo float usando array, ficaria assim:

```
float[] nota = new int[20];
```

```
float[] nota =  
{1,2,3,4,5,6,7,8,90.1,10.02,-11,12.9,13.7,14.5,15.7,16.0,17.5,19.3,20.2};
```

(lembre-se que float, ou decimais, são escritos com ponto, e não com vírgula, como fazemos no Brasil).

E o nome dos alunos, armazenaríamos em Strings:

```
String[] nome = new String[20];
```

Se notar bem, estamos usando a palavra 'new', que é a mesma que usamos na classe Scanner, para criar um objeto. Aqui não é diferente. Estamos criando um objeto de arrays também.

Duas informações importantes:

1. Quando declaramos variáveis numéricas usando array e não inicializamos com valores, estes serão automaticamente nulos (0 ou 0.0).

Se forem Strings, serão 'null' e vazias caso sejam caracteres (char).

2. O tamanho de um array é constante. Uma vez declarado seu valor, não poderá ser mudado.

Se declarou 5 inteiros, o array terá sempre tamanho 5.

Se quiser saber o tamanho de um array, use o método 'length', que está em todo objeto do tipo array e retorna um inteiro com o número de elementos em cada array: nome.length, nota.length etc

A ordem de numeração dos elementos

Vamos mostrar agora uma importante informação, não só de Java, mas de programação.

Em nosso dia-a-dia contamos: 1, 2, 3...

MyArray [N]

Item 0

Item 1

Item 2

Item 13

Item (N1)

MyArray [13] →

Não fazemos contagem usando o 0.

Em programação, porém, obviamente isso inclui Java, SEMPRE começamos contando em 0!

Por exemplo, se declaramos um array com o nome 'teste', de 3 elementos, seus elementos serão identificados por:

teste[0]

teste[1]

teste[2]

Logo, sempre que declaramos um array de 'n' índices, o índice desses valores vão de 0 até n-1.

Guarde isso, é importante e usaremos sempre de agora em diante.

Como usar os elementos de um array em Java

Pronto, você declarou um array/vetor. Tem elementos do tipo float. Mas usou só um nome, 'nota'.

Como representar 20 variáveis com um só nome?

A resposta é simples: usando números, ou índices.

'nota' é um Array de Floats.

Se quiser usar um tipo float, use a seguinte sintaxe: nome[indice]

Então, suas variáveis, de forma independente, são chamadas de: nota[0]. nota[1], nota[10] etc.

Esses serão seus nomes. Você pode usar como usaria as variáveis, por exemplo:

Armazenar a nota de um aluno que tirou 10

nota[10]= 10.0 //esse, provavelmente, é programador Java

Somar a nota de dois alunos:

```
float soma = nota[3] + nota[4];
```

Incrementar:

```
nota[5]++;
```

Enfim, pode fazer tudo. São variáveis do tipo float normais.

A diferença é que os nomes das variáveis tem números, que são chamados, em programação, de índice, que são criados automaticamente quando você declara um bloco de vários elementos.

Exemplos de código Java usando Array

Exemplo 1: Faça um programa que peça 3 números inteiros ao usuário, armazene em um Array, depois mostre o valor de cada elemento do array, assim como seu índice.

Passo 1: Primeiro declaramos um array de inteiros, contendo 3 elementos:

```
int[] nota = new int[3];
```

Depois o objeto 'entrada', do tipo Scanner, para receber os valores do usuário.

Passo 2: Armazenar os valores no array

Note que, você é programador e sabe que os índices vão de 0 até 2.

Mas o usuário não. Pro leigo, é número 1, número 2 e número 3, não inicia no 0.

No laço for, o nosso 'índice' vai de 0 até 2.

Porém, ao recebermos o valor de índice 'índice', estamos pedindo ao cliente o valor do número 'índice+1'.

Por exemplo, para armazenar um valor no 'nota[0]', vamos pedir o número '0+1' ao cliente.

Para armazenar um valor no 'nota[1]', vamos pedir o número '1+1' ao cliente.

Para armazenar um valor no 'nota[2]', vamos pedir o número '2+1' ao cliente.

Passo 3: exibindo os valores pro usuário

Usaremos outro laço for para exibir o valor dos índices, que variam de 0 até 2.

Porém, novamente, temos que mostrar 1 ao 3 pro cliente, pois pra ele não faz sentido 'número 0 -> valor 10 ' e sim 'número 1 -> valor 10'.

Então, nosso código Java ilustrando o uso de Arrays será:

```
import java.util.Scanner;
```

```
public class arrayTeste {
```

```
public static void main(String[] args){
```

```
    int[] nota = new int[3];
```

```
Scanner entrada = new Scanner(System.in);
```

```
    //recebendo os números
```

```
for(int indice=0 ; indice < 3 ; indice++){
```

```
System.out.print("\nEntre com o número " + (indice+1) + ": ");
```

```
    nota[indice] = entrada.nextInt();
```

```
}
```

```
//exibindo os números
```

```
for(int indice=0 ; indice < 3 ; indice++){
```

```
    System.out.printf("Número %d -> Valor armazenado: %d\n",indice+1,  
    nota[indice]);
```

```
    }
```

```
}
```

```
}
```

Exemplo 2: Faça um aplicativo Java que peça o nome do aluno, receba duas notas e depois retorne todas essas informações junto com a média dele.

Vamos usar, nesse exemplo, um array de floats com 3 elementos.

Dois elementos para armazenar os valores da nota e o terceiro vai armazenar a média.

O resto, não tem segredo. Veja como ficou nosso código Java:

```
import java.util.Scanner;
```

```
public class arrayTeste2 {
```

```
public static void main(String[] args){
```

```
    float[] nota = new float[3];
```

```
String nome;
```

```
Scanner entrada = new Scanner(System.in);
```

```
System.out.print("Nome do aluno: ");
```

```
    nome = entrada.nextLine();
```

```
System.out.print("Primeira nota: ");
```

```
    nota[0] = entrada.nextFloat();
```

```
System.out.print("Segunda nota: ");
```

```
    nota[1] = entrada.nextFloat();
```

```
//média
```

```
nota[2] = (nota[0] + nota[1])/2;
```

```
System.out.printf("O aluno %s tirou %.1f e %.1f, obtendo média final  
%.2f",nome,nota[0],nota[1],nota[2]);
```

```
}
```

```
}
```

O laço for para Arrays: o laço For each

Em toda sua vida de programador Java, será comum você usar o laço for para percorrer todos os elementos de um Array.

Se isso é tão utilizado e importante, poderia ter um atalho ou meio mais fácil de fazer isso, não?

É, e tem.

Sem contar que, com Arrays e o laço for convencional, é bem fácil de se enrolar e cometer erros.

Aprenda uma maneira especial de usar o laço for e pare de ter problemas: é laço foreach

Fazendo o for percorrer todo o Array/Vetor

A utilização é bem simples e direta. Vamos usar essa variante do for que percorre sempre, do começo ao fim, todos os elementos de um Array.

É bem útil, também, em termos de precaução e organização, pois alguns programadores não gostam de usar o índice 0, usam direto o índice 1 do array, ou as vezes nos confundimos e passamos (durante as iterações) do limite do array. Isso pode ser facilmente evitado usando o laço for modificado para percorrer os elementos de um array.

A sintaxe do for each é a seguinte:

```
for ( tipo variavel_do_tipo_do_seuArray : seuArray){  
  
//seu código  
  
}
```

Se o 'seuArray' for de inteiro, ficaria:

```
for (int count : seuArray){  
  
...  
  
}
```

Como podemos interpretar esse laço foreach?

Muito simples, o inteiro 'count' vai receber, a cada iteração, todos os valores de 'seuArray'.

Ou seja, ele vai percorrer todo o seu Array e receber seus valores, na ordem (do começo para o fim), um por vez. E com esses valores você vai fazer o que desejar.

Por exemplo, suponha que as notas de cada aluno de uma escola é armazenada em um Array.

Como para calcular a média você precisará sempre percorrer todo o Array/Vetor, pode, e deve-se, usar esse laço for especial. Pois ele foi feito para isso, para facilitar nossa vida.

Programadores não devem perder tempo. Devem fazer as coisas sempre da maneira mais simples e prática possível. Não tente fazer arte ou gambiarra.

Se o próprio Java lhe fornece as opções e ferramentas, use-as.

É uma arma 'especial' que não se vê por aí, em qualquer linguagem de programação.

Se você já programa ou programou em outra linguagem deve estar percebendo como Java é cheio dessas ferramentas e pormenores.

Se estudar tudo direitinho, irá se tornar um eficiente programador, pois o Java realmente foi feito para facilitar a vida de desenvolvedores, é realmente uma linguagem voltada para o trabalho.

Exemplo de uso do foreach em Java:

Crie um aplicativo em Java que peça 5 números ao usuário.

Depois, mostre o resultado da soma desses números;

Passo 1:

Vamos declarar o vetor de inteiro e o inteiro 'soma' para receber a soma dos resultados que o usuário fornecer.

Usamos um laço for, convencional, para percorrer o array.

Note que usamos 'numero.length', que é uma boa prática.

Passo 2:

Agora temos que percorrer o array inteiro, e já que temos que fazer isso, vamos fazer usando o laço for modificado que percorre um array inteiro recebendo seus valores.

Como o inteiro 'cont' vai receber os dados de todos os elementos do array 'numero', vamos somar o valor de 'cont' a cada iteração na variável 'soma'.

O código Java de nossa aplicação ficará:

```
import java.util.Scanner;
```



```
public class forParaArray {
```

```
public static void main(String[] args){
```

```
    int[] numero = new int[5];
```

```
    int soma=0;
```

```
Scanner entrada = new Scanner(System.in);
```

```
for(int cont=0 ; cont< numero.length ; cont++){
```

```
    System.out.print("Entre com o número "+(cont+1)+" : ");
```

```
        numero[cont]=entrada.nextInt();
```

```
    }
```

```
    //exibindo e somando
```

```
for(int cont : numero){
```

```
    soma += cont;
```

```
}
```

```
System.out.println("A soma dos números que você digitou é "+soma);  
    }  
}
```

Array Multidimensional ou Matriz: Array de arrays

Já vimos no capítulo passado de nossa apostila que um array é um conjunto de variáveis do mesmo tipo.

Porém, poderíamos também declarar um conjunto de arrays. Nesse caso, teríamos um array multidimensional ou vetor multidimensional, também conhecido por matrizes.

Matrizes ou Vetores multidimensionais: Conjunto de vetores ou arrays em Java

Imagine a situação que vínhamos usando como exemplo em nossa apostila online Java Progressivo, o exemplo dos dados dos alunos de uma escola.

Ora, um aluno faz várias provas todos os anos.

Vamos supor que ele faz 10 provas por ano. Como você já é um sábio programador Java, não vai declarar 10 floats para armazenar as 10 notas. Vai declarar um array de 10 elementos floats.

Porém, existem várias matérias no colégio: Matemáticas, Física, Química, Biologia etc.

Vamos supor que existam 10.

E aí, vai declarar 10 vetores/arrays ?

Claro que não, isso seria muito cansativo.

Declare um array de arrays.

Arrays de uma ou mais dimensões em Java

Nos exemplos passados, declaramos apenas um bloco de variáveis. Dizemos que esse vetor é unidimensional, pois é somente um bloco.

Veja tal array/vetor apenas como uma linha, assim terá a noção de dimensão.

Por exemplo, vamos declarar um array unidimensional com 5 notas de Matemática:

```
int[] notas = {8.0 , 7.5, 8.5 , 9.0 , 8.0};
```

Essa nota pode ser representada por uma matriz 1x5, ou seja, uma linha e 5 colunas:

8.0	7.5	8.5	9.0	8.0
-----	-----	-----	-----	-----

Agora vamos representar as notas em Física, abaixo das de Matemática. Teremos uma matriz 2x5, ou seja, uma matriz de duas linhas e 5 colunas:

8.0	7.5	8.5	9.0	8.0
8.9	9.0	8.6	8.4	8.0

Agora vamos representar as notas de Química, abaixo das notas de Física.

Teremos uma matriz 3x5, ou seja, uma matriz de três linhas e 5 colunas:

8.0	7.5	8.5	9.0	8.0
8.9	9.0	8.6	8.4	8.0
6.8	7.17.17.1	7.07.07.0	7.67.67.6	6.56.56.5

Matrizes: declarando vetores/arrays multidimensionais em Java

A sintaxe é exatamente a mesma do array normal, a diferença está no número de colchetes '[' que iremos usar.

No caso, usamos um par para cada dimensão.

Por exemplo, para declarar a matriz 2x5 do exemplo anterior:

```
float[][] notas = new float[2][5];
```

Ou

```
float[][] notas = { {8.0, 7.5, 8.5, 9.0, 8.0 }, {8.9, 9.0, 8.6, 8.4, 8.0 } };
```

Para declarar a matriz 3x5 do exemplo anterior:

```
float[][] notas = new float[3][5];
```

Ou

```
float[][] notas = { {8.0, 7.5, 8.5, 9.0, 8.0 }, {8.9, 9.0, 8.6, 8.4, 8.0 }, {6.8, 7.1,  
7.0, 7.6, 6.5 } };
```

Note que notas[0] se refere ao array de notas de Matemática.

Note que `notas[1]` se refere ao array de notas de Física.

Note que `notas[2]` se refere ao array de notas de Química.

Por exemplo: qual foi a quarta nota de Física do aluno?

Ora, o vetor de Física é `notas[1]`, e a quarta nota é o elemento `[3]` desse array.

Então a quarta nota de Física do aluno está armazenada em: `nota[1][3]`, que é 8.4

Exemplos de códigos:

Como de costume, em nossa apostila do curso Java Progressivo, vamos apresentar dois exemplos de como usar os arrays dimensionais, ou matrizes, em Java.

Exemplo 1: Crie um aplicativo em Java que peça ao usuário para preencher uma matriz 3x3 com valores inteiros e depois exiba essa matriz.

A grande novidade, e importância, nesse tipo de aplicativo são os laços for aninhados, ou seja, um dentro do outro.

Primeiro criamos um laço que vai percorrer todas as linhas da matriz. Podemos, e devemos, ver cada linha como um vetor de 3 elementos.

Dentro de cada linha, temos que percorrer cada elemento do array e fornecer seu valor. Fazemos isso através de outro laço for, que ficará responsável pelas 'colunas', formando nossos laços aninhados.

Para imprimir, o esquema é exatamente o mesmo. Imprimimos linha por linha, e em cada linha, imprimimos coluna por coluna.

```
import java.util.Scanner;
```

```
public class matrizTeste {
```

```
public static void main(String[] args){
```

```
    int[][] matriz = new int[3][3];
```

```
Scanner entrada = new Scanner(System.in);
```

```
System.out.println("Matriz M[3][3]\n");
```

```
for(int linha=0 ; linha < 3 ; linha++){
```

```
    for(int coluna = 0; coluna < 3 ; coluna ++){
```

```
        System.out.printf("Insira o elemento M[%d][%d]: ",linha+1,coluna+1);
```

```
        matriz[linha][coluna]=entrada.nextInt();
```

```
    }
```

```
}
```

```
System.out.println("\nA Matriz ficou: \n");
```

```
for(int linha=0 ; linha < 3 ; linha++){
```

```
for(int coluna = 0; coluna < 3 ; coluna ++){  
  
    System.out.printf("\t %d \t",matriz[linha][coluna]);  
  
        }  
  
    System.out.println();  
  
        }  
  
    }  
  
}
```

Exemplo 2: Crie um aplicativo em Java que peça ao usuário para preencher uma matriz 3x2 com valores inteiros e depois exiba essa matriz.

No exemplo passado, o número de linhas era igual ao número de colunas, da matriz. Vamos usar um exemplo diferente, para você fixar seu conhecimento em arrays multidimensionais.

Como dito no começo, uma matriz, array multidimensional ou vetor multidimensional nada mais é que um conjunto de arrays ou conjunto de vetores, array de arrays.

Quando fazemos: `int[5]` para declarar um array de inteiros, estamos declarando 5 variáveis do tipo inteiro.

Quando fazemos: `int[10][5]`, estamos declarando 10 arrays, e em cada array desses existem 5 inteiros. Ou seja, 10 arrays do tipo do exemplo passado. Logo, o tamanho desse array – `length` – é 10.

Você pode obter isso com o comando: `array.length`

Assim, uma matriz 3x2 tem tamanho 3.

Uma 4x3 tem tamanho 4, uma 10x123123 tem tamanho 10 etc.

Ou seja, o `length` de arrays multidimensionais é o número de linhas.

Vamos usar esse fato nos laços:

```
import java.util.Scanner;
```

```
public class matrizTeste2 {
```

```
public static void main(String[] args){
```

```
    int[][] matriz = new int[3][2];
```

```
Scanner entrada = new Scanner(System.in);
```

```
System.out.println("Matriz M[3][2]\n");
```

```
for(int linha=0 ; linha < 3 ; linha++){
```

```
    for(int coluna = 0; coluna < 2 ; coluna ++){
```

```
        System.out.printf("Insira o elemento M[%d][%d]: ",linha+1,coluna+1);
```

```
            matriz[linha][coluna]=entrada.nextInt();
```

```
        }
```

```
    }
```

```
    System.out.println("\nA Matriz ficou: \n");
```

```
    for(int linha=0 ; linha < 3 ; linha++){
```

```
        for(int coluna = 0; coluna < 2 ; coluna ++){
```

```
            System.out.printf("\t %d \t",matriz[linha][coluna]);
```

```
        }
```

```
    System.out.println();
```

```
}
```

}

}

Arrays em métodos: passagem por valor e passagem por referência

Vimos em nosso curso de Java uma seção só para métodos, e estamos em outra seção da apostila só para arrays.

São duas coisas importantes, e a interação entre elas também é importante.

Porém, há alguns detalhes que precisam ser esclarecidos e comentados, que veremos agora neste artigo.

Passando arrays/vetores para methods/métodos

Para chamar os métodos usando arrays como argumentos, não há segredo.

Faça como sempre:

```
nome_do_metodo(nome_da_variavel);
```

Não se preocupe com o tipo e com os colchetes, [], pois os métodos já estão preparados e sabem que o que você está mandando é um array e não um tipo comum.

Para declarar, porém, é que há a diferença:

```
public static tipo_do_retorno nomeDoMetodo(tipo_do_parametro[]  
nome_da_variavel);
```

Por exemplo, vamos usar o exemplo do código do artigo passado de nosso curso para criar um método chamado 'exibeMatriz' que exibe uma matriz.

Esse método exibe uma matriz 3x3 de inteiros.

```
public static void exibeMatriz(int[][] Mat){  
  
//codigo
```

```
}
```

Note que nossa matriz tem duas dimensões, então usamos dois pares de colchetes.

Colocando o código que exibe dentro deste método e fazendo o chamado dentro da main, nosso exemplo ficará assim:

Código Java: Preenche e exibe uma Matriz 3x3

```
import java.util.Scanner;
```

```
public class arrayPraMetodo {
```

```
public static void exibeMatriz(int[][] Mat){
```

```
System.out.println("\nA Matriz ficou: \n");
```

```
for(int linha=0 ; linha < 3 ; linha++){
```

```
for(int coluna = 0; coluna < 3 ; coluna ++){
```

```
System.out.printf("\t %d \t",Mat[linha][coluna]);
```

```
}
```

```
System.out.println();
```

```
}
```

```
}
```

```
public static void main(String[] args){
```

```
    int[][] matriz = new int[3][3];
```

```
Scanner entrada = new Scanner(System.in);
```

```
System.out.println("Matriz M[3][3]\n");
```

```
for(int linha=0 ; linha < 3 ; linha++){
```

```
    for(int coluna = 0; coluna < 3 ; coluna ++){
```

```
        System.out.printf("Insira o elemento M[%d][%d]: ",linha+1,coluna+1);
```

```
            matriz[linha][coluna]=entrada.nextInt();
```

```
        }
```

```
    }
```

```
exibeMatriz(matriz);
```

```
}
```

```
}
```

Passagem por valor e passagem por referência

Se você já estudou outras linguagens, como C ou C++ sabe o que são esse tipo de passagem e sua importância, assim como sua confusão.

Muitas vezes queremos passar um valor pro método para que este valor seja alterado. Neste caso, estamos passando por referência, ou seja, é como se estivéssemos passando realmente a própria variável pra ser modificada e receber outro valor dentro do método.

Quando passamos por valor, é como se o método tirasse uma 'xerox' ou uma cópia de nossa variável e trabalhasse com essa cópia, e não alterasse nossa variável. Altera e trabalha apenas com a cópia, deixando a nossa original intacta.

Ambos tipos de passagem são úteis e tem suas funcionalidades dependendo do que se queira fazer.

Porém, na linguagem de programação Java, a passagem é sempre, SEMPRE, feita por valor.

Sempre que criamos uma variável de um tipo, passamos uma cópia de seu valor para o método.

Veja este exemplo, onde definimos um numero de valor 2112, dobramos ele no método e imprimimos de novo. Seu valor não se altera:

Código Java: demonstrativo da passagem por valor, o valor não se altera no método

```
public class passagemPorValor {
```

```
    public static int dobra(int num){
```

```
        return num*2;
```

```
    }
```

```
    public static void main(String[] args){
```

```
        int numero=2112;
```

```
        System.out.println("O valor de numero é: " + numero);
```

```
        System.out.println("Dobrando seu valor.");
```

```
        dobra(numero);
```

```
        System.out.println("Agora o valor de número é: " + numero);
```

```
    }
```

}

A variável 'num', do método 'dobra()' é uma cópia da 'numero' do método 'main'.

O valor do 'numero' não é tocado nem visto pelo method 'dobra()'.

Os tipos de referência em Java

Existem tipos em Java que são chamados de 'tipos de referência, que são variáveis que, por natureza, recebem uma referência, ou seja, elas apontam para outro valor (mais especificamente, elas não armazenam o valor na memória, elas armazenam um endereço de outro bloco de memória - ou seja, tem uma referência desse outro endereço de memória).

Como exemplo, sabemos que o Java trata objetos, de uma maneira geral, e os nosso tema atual de estudo, os arrays, por referência. Logo, as classes e nossos arrays são tipo referência.

Quando declaramos um objeto por exemplo, com o nome 'meuObjeto', na verdade esse 'meuObjeto' vai guardar uma referência do objeto real. Ou seja, em Java, os tipos referências não armazenam na memória o valor, e sim um endereço - quem já tiver estudado C pode pensar logo em ponteiro!

Isso tudo se deve ao fato que objetos e arrays não possuem tamanhos definidos, como os tipos primitivos. Então fica mais 'complicado' pro Java tratar esse tipo de dado, pois eles não tem um tamanho exato.

Então, ao invés de lidar com seus valores exatos (value), ele trata eles por referência (reference).

A confusão reside no fato de que, quando passamos esses tipos de variáveis para os métodos, elas mudam de valor! Mas peraí, por quê?

Como assim? Mas, em Java, a passagem não é sempre por valor?

Na verdade, há dois tipos de chamadas: os que usam cópia dos tipos primitivos e os que usam cópias da referência de objetos. Porém, como o assunto é um pouco avançado e não vimos objetos, darei uma explicação simplificada.

Nos valores do tipo 'primitivo', uma cópia é gerada no método e todas as alterações são feitas nessa cópia, ok?

Nos valores do tipo 'referência' o mesmo, pois Java só passa por valor! Ok?

Então, no método, é criada outra cópia do objeto! Ou seja, temos outra referência apontando pro mesmo local na memória. Nada demais até aqui.

Na passagem do tipo por 'referência', essa segunda cópia, dentro do método, altera a referência. Mas a referência aponta pra um local na memória! Então, alteramos o local na memória quando mudamos a referência!

Ora, se as duas variáveis apontam pro mesmo local e uma delas muda esse local, o local fica mudado para as duas, ué. Esse é o truque!

Exemplo de como funciona a passagem dos tipo referência

Se eu eu declaro o meu tipo referência original com o valor 'x', na verdade essa minha variável aponta pra memória onde esse valor realmente está.

Quando mando pro método, o método cria uma cópia dessa variável do tipo referência, porém a danada dessa cópia também aponta pro mesmo local da memória. Logo, temos duas variáveis apontando pra um mesmo local da memória, o local onde está o 'x'.

Mas o método prossegue e altera o valor 'x' dessa cópia. Na verdade ele alterou foi o valor 'x' na memória. Agora esse valor 'x' é 'x+1', e o método acabou.

De volta pra onde o método foi chamado, minha variável do tipo referência original ainda aponta pro mesmíssimo local da memória. Porém, nesse local não existe mais 'x' e sim 'x+1' e temos a falsa impressão que passamos um valor por referência. Mas passamos sim, por valor!

Veja um exemplo, onde criamos um método que calcula o traço de uma matriz 3x3.

Traço é a soma dos elementos da diagonal principal da matriz, ou seja, os elementos cujo número da linha é igual ao número da coluna:

Código Java: Aplicativo que calcula o traço de uma Matriz

```
import java.util.Scanner;
```

```
public class traco{
```

```
public static void exhibeMatriz(int[][] Mat){
```

```
System.out.println("\nA Matriz ficou: \n");
```

```
for(int linha=0 ; linha < 3 ; linha++){
```

```
for(int coluna = 0; coluna < 3 ; coluna ++){
```

```
System.out.printf("\t %d \t",Mat[linha][coluna]);
```

```
    }
```

```
System.out.println();
```

```
    }
```

```
}
```

```
public static int traco(int[][] Mat){
```

```
    int soma=0;
```

```
for(int linha=0 ; linha<Mat.length ; linha++){
```

```
    soma += Mat[linha][linha];
```

```
}
```

```
return soma;
```

```
}
```

```
public static void main(String[] args){
```

```
    int[][] matriz = new int[3][3];
```

```
Scanner entrada = new Scanner(System.in);
```

```
System.out.println("Matriz M[3][3]\n");
```

```
for(int linha=0 ; linha < 3 ; linha++){
```

```
for(int coluna = 0; coluna < 3 ; coluna ++){
```

```
System.out.printf("Insira o elemento M[%d][%d]: ",linha+1,coluna+1);
```

```
    matriz[linha][coluna]=entrada.nextInt();
```

```
}
```

```
}
```

```
exibeMatriz(matriz);
```

```
System.out.println("\nO traço da Matriz é: "+ traco(matriz));
```

```
}
```

```
}
```

Exemplo de passagem do tipo referência na vida real

Vamos supor que eu anotei um endereço em um papel, o endereço de um prédio.

Em suma: eu tenho uma referência desse prédio.

O papel é minha variável do tipo referência. Ou seja, ela não tem o prédio. Ela tem o endereço dele (referência). Bem claro agora, não?

Agora vamos fazer um 'método'. Ou seja, vamos tirar uma cópia desse papel e dar a um homem-bomba.

Ok, agora o homem bomba tem um papel com a referência também!

Os dois papéis apontam pro mesmo local: o prédio.

É isso que o método faz. Cria uma cópia da referência - por isso é passagem por valor, pois trabalha com a cópia.

Então, agora nosso homem-bomba irá trabalhar com essa cópia (como o método faz).

Ele vai lá e explode o prédio.

Ele mexeu no meu papel? Não, continua intacto, como prediz a passagem por valor.

Porém, ele alterou o lugar para onde meu papel fazia referência.

Ele alterou isso baseado no papel dele. Mas o papel dele tinha o mesmo endereço do meu!

Então, se ele alterou pra onde apontava - pra onde fazia referência - essa mudança vai acontecer pra mim também, pois minha referência também apontava pro prédio!

Em suma, se ficou confuso, aqui vai um resumo do que nos interessa neste ponto do curso:

- se passar tipos primitivos (int, float, double etc), eles não serão alterados pelo métodos
- se passar arrays, eles serão alterados pelos métodos. Bem como todo tipo de Classes

Para ler mais sobre isso, Java in a Nutshell: http://docstore.mik.ua/oreilly/java-ent/jnut/ch02_10.htm

Classe Arrays (Arrays Class): aprenda a manusear (copiar, ordenar, buscar e manipular) Arrays

Uma das grandes vantagens de se programar em Java é que ele nos fornece uma quantidade enorme de APIs. Ou seja, o Java já vem com diversas classes e métodos prontos para serem usados.

Nesse tutorial de nosso curso de Java iremos aprender a usar a Classe Arrays bem como seus principais métodos, que ordenam, comparam, buscam, colocam elementos nos Arrays/vetores dentre outras séries de utilidades.

Como usar a Arrays Class

Para importar, coloque no início de seu código:

```
import java.util.Arrays;
```

Diferente de outras classes, não vamos declarar diretamente os objetos: Arrays
nome_do_objeto;

Na verdade, quando criamos um Array, automaticamente já podemos usar os métodos da classe Arrays nesse vetor(array).

No exemplo a seguir, vamos mostrar como usar os métodos da classe Arrays.

Ordenando e buscando um elemento em um Array no Java

Nesse exemplo de código deste tutorial de java, iremos usar o método 'sort', que significa ordenar em inglês e o método 'binarySearch', que faz uma busca por um determinado elemento e retorna sua localização no Array.

Inicialmente definimos um Array de inteiros quaisquer.

Para exibir os elementos de um Array, em Java, como uma string, usamos o método:

```
Arrays.toString( array );
```

que recebe um array como argumento.

Depois vamos usar o método 'sort', que ordena o Array de ordem crescente. Ou seja, vai trocar a posição dos elementos de modo que estejam organizados do menor para o maior.

Para usar o método sort em Java fazemos:

```
Arrays.sort( array );
```

Depois, mostramos o array novamente, mas na forma de string para você ver como ficou a organização do array depois do método sort ter entrado em ação.

Depois, vamos armazenar em um inteiro, 'posicao', a posição do elemento '2112' no array: `Arrays.binarySearch(array, numero_que_estamos_procurando) ;`

IMPORTANTE: esse método só funciona se o array estiver na ordem crescente! Ou seja, só use o 'binarySearch' após usar o 'sort', pois a Binary Search é um tipo de busca inteligente (ele não sai simplesmente procurando os elementos, um por um, seu algoritmo se baseia na ordenação).

```
import java.util.Arrays;
```

```
public class arraysClass{
```

```
public static void main(String[] args){
```

```
    int[] numeros={1, 4, 0, -13, 2112, 14, 17};
```

```
    int posicao;
```

```
    System.out.println("Os elementos do array são: "+ Arrays.toString(numeros));
```

```
    System.out.println("Ordenando...");
```

```
    Arrays.sort(numeros);
```

```
    System.out.println("Array ordenado: "+Arrays.toString(numeros));
```

```
posicao=Arrays.binarySearch(numeros, 2112);
```

```
System.out.println("Posição do elemento '2112': "+ posicao);
```

```
}
```

```
}
```

Métodos da classe Arrays (Array class methods):

Dependendo dos métodos, podemos usar inteiros, float, double, char, short, List, Object etc.

Para saber exatamente se existe um método para aplicar no que você deseja usar, olhe a documentação da classe Arrays em:

<http://docs.oracle.com/javase/7/docs/api/java/util/Arrays.html>

Arrays.binarySearch:

Serve para encontrar um elemento específico dentro do array. Retorna a posição no array (inteiro).

Caso passe como argumento um array e um valor, a busca é feita em todo o array.

Podemos também passar um intervalo de busca. Por exemplo, para procurar o elemento 'x', no Array 'vetor', a partir do elemento 'daqui' até o elemento 'ate_aqui', faça:

```
Arrays.binarySearch( vetor, daqui, ate_aqui, x);
```

Arrays.copyOf:

Esse método copia um array e retorna outro. Esse que ele retorna é uma cópia do primeiro.

Se receber dois argumentos - um array e um valor, esse array que você passa é aquele que você deseja copiar e o valor é o número de elementos que você deseja copiar (ou o número de elementos que você quer que seu novo array

tenha. Caso deseje ter um array maior, esse método preenche com 0 ou nulls):

```
novoArray[] = Arrays.copyOf( arrayOriginal,  
numero_de_elementos_a_serem_copiados);
```

Você também pode especificar uma faixa de valores:

```
novoArray[] = Arrays.copyOf( arrayOriginal, daqui, ate_aqui);
```

Arrays.equals:

Recebe dois arrays. Retorna true caso sejam iguais e false caso contrário.

Arrays.fill:

Vai preencher os valores de um array com determinado valor.

Caso deseje que todos os elementos de 'array' tenham o valor 'valor':

```
Arrays.fill(array, valor);
```

Para preencher só determinada faixa de valores:

```
Arrays.fill(array, daqui, ate_aqui, valor);
```

Arrays.sort:

Ordena os elementos em ordem crescente:

```
Arrays.sort(array);
```

Para ordenar uma faixa de valores:

```
Arrays.sort(array, daqui, ate_aqui);
```

Arrays.toString:

Retorna todos os elementos de um array na forma de string:

```
Arrays.toString(array);
```

Como usar ArrayList em Java: principais métodos

Até o momento, nessa seção de Estrutura de Dados de nosso curso de Java, vimos o quão útil são os arrays e as operações que podemos fazer com esses arrays, também conhecidos por vetores.

Porém, há um grande problema neles: são estáticos. Ou seja, tem tamanho definido, não podemos adicionar nem excluir nada neles, o que é bastante incômodo e faz com que arrays em Java sejam limitados.

E por que arrays não são muito úteis?

Porque na vida real as coisas não são assim estáticas, elas mudam. O número de funcionários de uma empresa tendem a crescer, em uma faculdade alunos se formam e outros se matriculam, em um jogo de RPG vamos colecionando itens no decorrer do jogo, ao passo que vamos usando outros.

Agora, nesse tutorial do curso Java Progressivo, vamos aprender como usar os ArrayList, que são arrays mais 'flexíveis', ou dinâmicos, já que podemos adicionar ou retirar elementos, além de fazer outras coisas que não são possíveis com os Arrays.

O que são ArrayList em Java

O Java, por padrão, possui uma série de recursos prontos (APIs) para que possamos tratar de estrutura de dados, também chamados de coleções (collections).

Podemos dizer que ArrayList é uma classe para coleções. Uma classe genérica (generic classes), para ser mais exato.

Coleções mesmo, de qualquer tipo de 'coisa', desde que seja um objeto.

Você pode criar seus objetos - através de uma classe - e agrupá-los através de ArrayList e realizar, nessa coleção, várias operações, como: adicionar e retirar elementos, ordená-los, procurar por um elemento específico, apagar um elemento específico, limpar o ArrayList dentre outras possibilidades.

Como declarar e usar ArrayList em Java

Importe:

```
import java.util.ArrayList;
```

Por ser um tipo diferente, sua sintaxe é um pouco diferente do que você já viu até então:

```
ArrayList< Objeto > nomeDoArrayList = new ArrayList< Objeto >();
```

No exemplo a seguir, vamos usar um ArrayList de String para trabalhar com o nome de várias Bandas de música:

```
ArrayList<String> bandas = new ArrayList<String> ();
```

Exemplo de uso do ArrayList

Após declarar a ArrayList 'bandas' que armazenará Strings, vamos adicionar alguns nomes.

Primeiro adicionamos a banda "Rush":

```
bandas.add("Rush");
```

Existe um método do ArrayList chamado 'toArray()' que coloca todos os elementos de um ArrayList em um Array.

Ou seja: `bandas.toArray()` é um Array!

Porém, já vimos que existe um método 'toString' da classe Arrays que retorna uma String com os elementos de um Array. Vamos usar esse método para exibir todos os elementos do ArrayList, que transformamos em Array através do método 'toArray()':

```
Arrays.toString( bandas.toArray() );
```

Vamos adicionar a segunda banda, "Beatles" e imprimir, usando o mesmo método.

Note que quando usamos 'add', sempre adicionamos o elemento pro fim da ArrayList.

Confirme isso agora, vendo que a banda "Iron Maiden" ficará depois de "Beatles".

Vamos pegar o primeiro elemento, o elemento '0', através do método 'get':

```
bandas.get(0);
```

Note que é a banda "Rush", pois ela foi a primeira a ser adicionada.

Vamos adicionar o "Tiririca" na posição do "Rush", ou seja, na posição '0':

```
bandas.add(0,"Tiririca");
```

ou

```
bandas.add( bandas.indexOf("Rush"), "Tiririca");
```

Pois o método 'indexOf' retorna o índice em que ocorre "Rush".

Para saber o tamanho que tem seu ArrayList, basta usar o método 'size()':

```
bandas.size();
```

Feito isso, rapidamente remova o "Tiririca", pois alguém pode ver.

Para tal, use o método 'remove':

```
bandas.remove("Tiririca");
```

Ok. Não quer mais brincar de ArrayList? Remova tudo

```
bandas.clear();
```

```
import java.util.ArrayList;
```

```
import java.util.Arrays;
```

```
public class arrayLists{
```

```
public static void main(String[] args){
```

```
    ArrayList<String> bandas = new ArrayList<String> ();
```

```
    bandas.add("Rush");
```

```
System.out.print( "Adicionando a banda Rush: " );
```

```
System.out.println( Arrays.toString( bandas.toArray() ) );
```

```
    bandas.add("Beatles");
```

```
System.out.print( "Adicionando a banda Beatles: " );
```

```
System.out.println( Arrays.toString( bandas.toArray() ) );
```

```
bandas.add("Iron Maiden");
```

```
System.out.print( "Adicionando a banda Iron Maiden: " );
```

```
System.out.println( Arrays.toString( bandas.toArray() ) );
```

```
System.out.print( "Quem está na índice 0: " );
```

```
System.out.println( bandas.get(0) );
```

```
System.out.print( "Adicionando Tiririca onde estava o Rush: " );
```

```
bandas.add( bandas.indexOf("Rush"), "Tiririca");
```

```
System.out.println( Arrays.toString( bandas.toArray() ) );
```

```
System.out.print( "Número de elementos na lista: " );
```

```
System.out.println( bandas.size() );
```

```
System.out.print( "Removendo o Tiririca: " );
```

```
bandas.remove("Tiririca");
```

```
System.out.println( Arrays.toString( bandas.toArray() ) );
```

```
System.out.print( "Removendo tudo: " );
```

```
bandas.clear();
```

```
System.out.println( Arrays.toString( bandas.toArray() ) );
```

```
}
```

```
}
```

Saída:

Adicionando a banda Rush: [Rush]

Adicionando a banda Beatles: [Rush, Beatles]

Adicionando a banda Iron Maiden: [Rush, Beatles, Iron Maiden]

Quem está na índice 0: Rush

Adicionando Tiririca onde estava o Rush: [Tiririca, Rush, Beatles, Iron Maiden]

Número de elementos na lista: 4

Removendo o Tiririca: [Rush, Beatles, Iron Maiden]

Removendo tudo: []

Para saber mais métodos e mais opções de parâmetros/argumentos, consulte a documentação sobre ArrayList em Java, no site da Oracle:

<http://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html>

Se em vez de Strings, quiser trabalhar com números, use as classes Wrappers:

ArrayList<Integer>, ArrayList<Float>, ArrayList<Double>, ArrayList<Long>
etc.

Apostila de Java, Capítulo 5 - Um pouco de Arrays

Página 71, Exercícios 5.5: Arrays

Enunciados

QUESTÃO 01:

Volte ao nosso sistema de Funcionario e crie uma classe Empresa dentro do mesmo arquivo .java. A Empresa tem um nome, cnpj e uma referência a uma array de Funcionario, além de outros atributos que você julgar necessário

```
class Empresa {  
  
    // outros atributos  
  
    Funcionario[] empregados;  
  
    String cnpj;  
  
}
```

QUESTÃO 02:

A Empresa deve ter um método adiciona, que recebe uma referência a Funcionario como argumento, e guarda esse funcionário. Algo como:

```
void adiciona(Funcionario f) {  
  
    // algo tipo:  
  
    // this.empregados[ ??? ] = f;  
  
    // mas que posição colocar?  
  
}
```

...

Você deve inserir o Funcionario em uma posição da array que esteja livre. Existem várias maneiras para

você fazer isso: guardar um contador para indicar qual a próxima posição vazia ou procurar por uma

posição vazia toda vez. O que seria mais interessante?

É importante reparar que o método adiciona não recebe nome, rg, salário, etc. Essa seria uma ma-

neira nem um pouco estruturada, muito menos orientada a objetos de se trabalhar. Você antes cria um

Funcionario e já passa a referência dele, que dentro do objeto possui rg, salário, etc.

QUESTÃO 03:

Crie uma classe TestaEmpresa que possuirá um método main. Dentro dele crie algumas instâncias de

Funcionario e passe para a empresa pelo método adiciona. Repare que antes você vai precisar criar a

array, pois inicialmente o atributo empregados da classe Empresa não referencia lugar nenhum (null):

```
Empresa empresa = new Empresa();
```

```
empresa.empregados = new Funcionario[10];
```

```
//
```

```
....
```

Ou você pode construir a array dentro da própria declaração da classe Empresa, fazendo com que toda

vez que uma Empresa é instanciada, a array de Funcionario que ela necessita também é criada.

Crie alguns funcionários e passe como argumento para o adiciona da empresa:

```
Funcionario f1 = new Funcionario();
```

```
f1.salario = 1000;
```

```
empresa.adiciona(f1);
```

```
Funcionario f2 = new Funcionario();
```

```
f2.salario = 1700;
```

```
empresa.adiciona(f2);
```

Você pode criar esses funcionários dentro de um loop, e dar valores diferentes de salários:

```
for (int i = 0; i < 5; i++) {
```

```
Funcionario f = new Funcionario();
```

```
f.salario = 1000 + i * 100;
```

```
empresa.adiciona(f);
```

```
}
```

Repare que temos de instanciar Funcionario dentro do laço. Se a instanciação de Funcionario ficasse

acima do laço, estaríamos adicionado cinco vezes a mesma instância de Funcionario nesta Empresa, e

mudando seu salário a cada iteração, que nesse caso não é o efeito desejado.

Opcional: o método adiciona pode gerar uma mensagem de erro indicando quando o array já está cheio.

QUESTÃO 04:

Percorra o atributo empregados da sua instância da Empresa e imprima os salários de todos seus fun-

cionários. Para fazer isso, você pode criar um método chamado mostraEmpregados dentro da classe

Empresa:

...

```
void mostraEmpregados() {
```

```
for (int i = 0; i < this.empregados.length; i++) {
```

```
System.out.println("Funcionário na posição: " + i);
```

```
// preencher para mostrar o salário!!
```

```
}
```

```
}
```

```
...
```

Cuidado ao preencher esse método: alguns índices do seu array podem não conter referência para um

Funcionario construído, isto é, ainda se referirem para null. Se preferir, use o for novo do java 5.0.

Aí, através do seu main, depois de adicionar alguns funcionários, basta fazer:

```
empresa.mostraEmpregados();
```

QUESTÃO 05 (opcional):

Em vez de mostrar apenas o salário de cada funcionário, você pode chamar o método mostra() de cada Funcionario da sua array.

QUESTÃO 06 (opcional):

Crie um método para verificar se um determinado Funcionario se encontra ou não como

funcionário desta empresa:

```
boolean contem(Funcionario f) {
```

```
// ...
```

```
}
```

Você vai precisar fazer um for na sua array e verificar se a referência passada

como argumento se encontra

dentro da array. Evite ao máximo usar números hard-coded, isto é, use o `.length`.

QUESTÃO 07 (opcional):

Caso a array já esteja cheia no momento de adicionar um outro funcionário, criar uma nova maior e copiar os valores. Isto é, fazer a realocação já que java não tem isso: uma array nasce e morre com o mesmo `length`.

Soluções

Vamos começar de baixo, criando a classe Funcionario.

Esta classe é bem simples, pois só vamos usar dois atributos para cada Funcionario: salario e numero.

Onde esse número é de identificação dele na empresa (ou você pode colocar um nome).

Essa classe terá dois métodos:

o getSalario(), que simplesmente retorna o valor do salário do funcionário e o método mostra(), que exibe as informações completas de cada funcionário.

Agora vamos juntar esses funcionários para formar a empresa e criar a classe Empresa.

Esse classe é um pouco mais complexa, pois trata da empresa e de todos os funcionários.

Esse grupo de funcionários estará armazenado no array 'empregados'.

A empresa possui também duas Strings, que representam o nome da empresa e o CNPJ.

A última variável é o inteiro 'numFuncionarios', que é responsável por armazenar o número de funcionários naquela empresa.

No método construtor padrão criamos o array com tamanho 10, inicializamos o número inicial de funcionários (que é 0, pois nenhum foi adicionado ainda), bem o nome da empresa e o CNPJ, que serão passados por argumento.

Nessa classe empresa será necessário ter o método 'adiciona', que preenche o array 'empregados', como pede a questão 2.

Esse método funciona da seguinte maneira: primeiro testamos se o número de funcionários existentes é menor que 10, pois se for igual ou maior, não podemos mais adicionar ninguém (lembre-se que o número de um array é fixo, e o nosso foi criado com tamanho 10).

Caso satisfaça o teste condicional, vamos adicionar esse novo funcionário na posição 'numFuncionarios' e incrementar essa variável 'numFuncionarios', já que adicionamos um funcionário novo.

Conforme pede o enunciado do exercício 4, criamos também a classe 'mostraEmpregados', que simplesmente corre por todos os empregados existentes e exibe suas informações.

Por fim, a classe Empresa possui o método 'contem' que checa se determinado funcionário está presente na empresa.

Para fazer isso, precisamos comparar a informações do funcionário que estamos buscando dentre todos os funcionários da empresa.

Vamos comparar o atributo 'numero' de cada empregado, pois esse número é único (poderia ser por nome também, caso tenha criado esse atributo na classe Funcionario).

Como foi ensinado o enhanced-for (ou foreach), vamos usar ele para fazer essa comparação dentro do método 'contem'.

A variável 'funcionario', do tipo Funcionario, recebe cada objeto (empregado) do array 'empregados[]' e compara o número de cada um. Caso encontre o número, é porque o funcionário existe na empresa e retorna o valor booleano true. Caso não encontre, ao fim do enhanced-for o método retorna false.

Agora vamos criar a classe principal, que contém o método main: CaelumCap5, na qual vamos fazer todos os testes.

Primeiramente criamos uma empresa: a JP, que tem o nome "Java Progressivo" e CNPJ "12345", que é passado ao método construtor padrão da classe Empresa.

Depois, conforme pede o exercício 3, vamos preencher os valores dos salários dos funcionários. Porém, vamos colocar também uma informação a mais: o número de cada funcionário, que será único.

Preenchido os salários e números dos 10 funcionários, vamos invocar o método 'mostraEmpregados', do objeto JP, e veremos todas as informações dos funcionários na tela.

Como é pedido no exercício 5, vamos mostrar essas mesmas informações, porém percorrendo todo o array de funcionários do objeto JP e usando o método 'mostra', de cada funcionário.

Agora vamos fazer dois testes para verificar a funcionalidade do método 'contem': criamos dois funcionários, de números '7' e '11' e checamos se eles estão presentes na empresa (a checagem poderia ser feita usando nomes, por exemplo).

Como é de se esperar, o funcionário 7 está na empresa, mas o 11 não.

Como o décimo primeiro não está lá, vamos adicionar ele à empresa.

Mas a empresa só tem espaço para 10 funcionários, o que nos leva a fazer o que é pedido no exercício 7: criamos outro array, agora com 11 elementos.

Primeiro nos copiamos todas as informações do array antigo para o array novo. Agora este possui 10 funcionários.

Por fim, adicionamos o último funcionário.

Para ver como tudo ocorreu bem, fazemos os testes do 'contem' de novo e mostramos todos os funcionários da empresa, através do método 'mostra', de cada funcionário.

Nosso código, dessas 7 questões, fica assim:

Classe Funcionario.java

```
public class Funcionario {
```

```
    double salario;
```

```
int numero;
```

```
double getSalario(){
```

```
return this.salario;
```

```
}
```

```
void mostra(){
```

```
    System.out.println("Empregado de número " + numero + ": " +  
getSalario());
```

```
}
```

```
}
```

```
Classe Empresa.java
```

```
public class Empresa {
```

```
    Funcionario[] empregados;
```

```
String nome,
```

```
    cnpj;
```

```
int numFuncionarios;
```

```
Empresa(String nome, String cnpj){  
    empregados = new Funcionario[10];  
    numFuncionarios=0;
```

```
    this.nome = nome;
```

```
    this.cnpj = cnpj;  
}
```

```
void adiciona(Funcionario f){
```

```
    if(numFuncionarios < 10){  
        empregados[numFuncionarios] = f;  
        numFuncionarios++;  
    }  
}
```

```
void mostraEmpregados(){
```

```
    System.out.println("Empresa " + this.nome + " CNPJ " + this.cnpj);
```

```
for(int count=0 ; count < numFuncionarios ; count++){  
    System.out.println("Empleado de número " + (count + 1) + ": " +  
    empleados[count].getSalario());  
    }  
}
```

```
boolean contem(Funcionario f){
```

```
for(Funcionario funcionario: empleados){
```

```
if(funcionario.numero == f.numero){
```

```
    return true;
```

```
    }
```

```
}
```

```
return false;
```

```
}
```

```
}
```

Classe CaelumCap5.java

```
public class CaelumCap5 {

    public static void main(String[] args) {

        Empresa JP = new Empresa("Java Progressivo", "12345");

        for (int i = 0; i < 10; i++) {

            Funcionario f = new Funcionario();

            f.salario = 1000 + i * 100;

            f.numero = i+1;

            JP.adiciona(f);

        }

        System.out.println("Método mostraEmpregados:");

        JP.mostraEmpregados();

        System.out.println("\nUsando o método 'mostra', de cada funcionário:");

        for(int i=0 ; i < JP.empregados.length ; i++){

            JP.empregados[i].mostra();

        }

    }

}
```



```
Funcionario f7 = new Funcionario();
```

```
Funcionario f11 = new Funcionario();
```

```
f7.numero = 7;
```

```
f11.numero = 11;
```

```
System.out.println("\nContém o funcionario 7? : " + JP.contem(f7));
```

```
System.out.println("Contém o funcionario 11? : " + JP.contem(f11));
```

```
//Criando um array de 11 elementos
```

```
Funcionario[] empregados2 = new Funcionario[11];
```

```
//Copiando os elementos dos array anterior para o novo array
```

```
for(int i=0 ; i < JP.empregados.length ; i++){
```

```
    empregados2[i] = JP.empregados[i];
```

```
}
```

```
//Adicionando o empregado 11
```

```
empregados2[10] = f11;
```

```
//checando de novo
```

```
System.out.println("\nContém o funcionario 11? : " + JP.contem(f11));
```

```
//Mostrando os empregados
```

```
System.out.println("\nUsando o método 'mostra', de cada funcionário:");
```

```
for(int i=0 ; i < empregados2.length ; i++){
```

```
    empregados2[i].mostra();
```

```
}
```

```
}
```

```
}
```

Página 75, Desafío

Enunciado

QUESTÃO 01:

No capítulo anterior, você deve ter reparado que a versão recursiva para o problema de Fibonacci é lenta porque toda hora estamos recalculando valores. Faça com que a versão recursiva seja tão boa quanto a versão iterativa. (Dica: use arrays para isso)

Solução

QUESTÃO 01:

Vamos criar um array de 70 elementos.

Na posição 0 será armazenado o elemento 0 da série de Fibonacci.

Na posição 1 será armazenado o elemento 1 da série de Fibonacci.

Na posição 2 será armazenado o elemento 2 da série de Fibonacci.

...

Na posição n será armazenado o elemento n da série de Fibonacci.

Onde esse 'n' é o número que o usuário inseriu e, obviamente, deve ser no máximo 69 (pois nosso array é de 70 elementos, ou seja, vai de 0 até 69).

Esse método é mais eficiente, pois uma vez que calculamos um número da série, nós armazenamos ele no array, e para gerar novos números não será necessário recalcular os elementos anteriores, basta usar seus resultados armazenados no array.

```
import java.util.Scanner;
```

```
public class CaelumFibonacci {
```

```
public static void main(String[] args) {  
    //Declarando o array de 100 elementos  
  
    int[] serieFibonacci = new int[100];  
  
    //preenchendo os dois elementos iniciais  
    serieFibonacci[0]=0;  
    serieFibonacci[1]=1;  
  
    int n;  
    Scanner entrada = new Scanner(System.in);  
  
    System.out.print("Que elemento da série deseja calcular [0-99]: ");  
    n = entrada.nextInt();  
  
    //vamos calcular do elemento 2 até o elemento n  
  
    for(int elemento=2 ; elemento<=n ; elemento++){  
        serieFibonacci[elemento] = serieFibonacci[elemento-1] +  
        serieFibonacci[elemento-2];  
    }  
}
```

```
System.out.println("O valor do elemento " + n + " é: " + serieFibonacci[n]);
```

```
}
```

```
}
```

Jogo: Batalha Naval em Java

BATALHA-NAVAL

UMA VITÓRIA E SEU DESENVOLVIMENTO



1884

Agora que aprendemos como passar arrays (de qualquer dimensão) para métodos, já podemos fazer algo mais útil. Vamos fazer o famoso batalha naval, no modo texto.

Usaremos tudo que aprendemos até o momento em nosso curso de Java: if else, laço do while, laço for, métodos, números aleatórios, arrays, arrays multidimensionais e muita criatividade.

Regras do Jogo Batalha Naval em Java

Há um tabuleiro de 5x5, ou seja, 25 blocos. Há 3 navios escondidos (um em cada bloco).

O objetivo do jogar é descobrir onde estão estes navios e acertá-los.

A cada tiro dado é dito se você acertou algum navio. Caso tenha errado, é dito quantos navios existem naquela linha e naquela coluna.

O jogo só acaba quando você descobrir e afundar os 3 navios.

Legenda pro usuário:

~ : água no bloco. Ainda não foi dado tiro.

* : tiro dado, não há nada ali.

X : tiro dado, havia um navio ali.

Como jogar:

A cada rodada, entre com dois números: o número da linha e o número da coluna onde quer dar o tiro.

Depois é só esperar pra ver se acertou, ou a dica.

Para os programadores Java:

O tabuleiro 5x5 é de inteiros. Ele é inicializado com valores '-1'.

A cada tiro ele é atualizado, dependendo se o usuário acertou ou errou. Esses números servirão para exibir '~', '*' ou 'X' para o usuário.

Também servirão para exibir as dicas.

Legenda do tabuleiro:

-1 : nenhum tiro foi dado naquele bloco (~)

0 : o tiro foi dado e não havia nada (*)

1 : o usuário atirou e tinha um navio lá (X)

Dica 9:

linha 4 -> 2 navios

coluna 5 -> 1 navios

Você acertou o tiro (4,5)

Jogo terminado. Você acertou os 3 navios em 9 tentativas

	1	2	3	4	5
1	*	X	~	~	~
2	~	*	*	~	~
3	~	*	~	~	~
4	~	X	~	*	X
5	~	~	~	~	*

Métodos:

void inicializaTabuleiro(int[][] tabuleiro) - inicialmente coloca o valor -1 em todas as partes do tabuleiro

void mostraTabuleiro(int[][] tabuleiro) - recebe o tabuleiro de inteiros e os exibe

void iniciaNavios(int[][] navios) - esse método sorteia 3 pares de inteiros, que são a localização dos 3 navios

void darTiro(int[] tiro) - recebe um tiro (linha e coluna) do usuário, e armazena na variável tiro[]

boolean acertou(int[] tiro, int[][] navios) - checa se o tiro dado acertou um navio

void dica(int[] tiro, int[][] navios, int tentativa) - dá a dica de quantos navios existem naquela linha e naquela coluna onde o tiro foi dado

void alteraTabuleiro(int[] tiro, int[][] navios, int[][] tabuleiro) - após o tiro ser dado, o tabuleiro é alterado, mostrando o tiro que foi dado(se acertou ou errou)

Lógica do problema

Inicialmente, são criadas as variáveis 'tabuleiro[5][5]', que vai armazenar o tabuleiro do jogo, a variável 'navios[3][2]', que vai armazenar a posição (linha e coluna) dos 3 navios que estão escondidos no tabuleiro, a variável 'tiro[2]' que vai armazenar a posição (linha e coluna) do tiro que o jogador vai dar a cada rodada, além da variável 'tentativa', que vai armazenar o número de tentativas que o jogador fez até acertar os 3 navios e, por fim, a variável 'acertos' que contabiliza o número de navios que você acertou.

O método 'inicializaTabuleiro()' é acionado, pra criar o tabuleiro com o número '-1' em todas as posições.

Depois é o método 'iniciaNavios()', que vai sortear a posição de 3 navios (linha e coluna).

Esse método sorteia dois números, entre 0 e 4. Depois ele checa se esse número já saiu, pois não podem existir dois barcos na mesma posição.

Caso já tenha saído, entra num laço do...while que fica sorteando números, e só sai quando sorteia outro par de números que ainda não é a localização de um navio.

Após isso, na main(), vamos iniciar o jogo.

Os jogos geralmente se iniciam através de um laço do...while. No caso, a condição do while é 'acertos!=3'. Ou seja, enquanto você não acertar os 3 navios, o jogo não para.

A primeira coisa que ocorre no laço é mostrar o tabuleiro, através do laço 'mostraTabuleiro()'.

Esse método checa cada posição o 'tabuleiro'. Se for -1 ele exibe água, '~'. Se for 0, ele exibe o tiro que foi dado e errou '*', e se for 1, ele exibe 'X' que indica que você acertou um navio naquela posição.

Após mostrar o tabuleiro, você vai dar seu tiro, através do método 'darTiro()', que recebe dois inteiros.

Note que, o usuário vai digitar números de 1 até 5, pois ele conta de 1 até 5.

Você, como programador Java é que conta de 0 até 4.

Portanto, quando o usuário entrar com a linha e com a coluna, SUBTRAIA 1 de cada valor desses.

Ou seja, se usuário entrou com (1,1), no seu tabuleiro Java 5x5 isso representa a posição (0,0).

Após o tiro ser dado, nosso jogo em Java vai checar se esse tiro acertou algum navio. Isso é feito com o método 'acertou()', que retorna 'true' caso acerte e 'false' caso erre.

No método, ele simplesmente checa se esse par de valores - 'tiro[2]' - que definem seu tiro, batem com algum dos valores que definem a posição dos navios - navios[3][2].

Caso acerte, o valor de 'acertos' aumenta.

Acertando ou errando, 'tentativas' aumenta, pois uma tentativa foi feita.

Acertando ou errando, uma dica também é exibida. Essa dica é exibida através do método 'dica()', que vai olhar o seu 'tiro[2]', e olha se na linha e na coluna que

você tentou existe mais algum navio, catando na 'navios[3][2]'.

Note que, a linha do seu tiro é 'tiro[0]' e a coluna é 'tiro[1]'.

A linha de cada navio é 'navios[x][0]' e a coluna de cada navio é 'navios[x][1]', onde 'x' é o número dos navios. No nosso caso, são 3 navios, ou seja, vai de 0 até 2.

Acertando ou errando, o tabuleiro vai ser alterado. O tiro que você deu vai alterar aquele local do tabuleiro, vai aparecer tiro dado '*' ou tiro que acertou 'X'.

Vamos fazer isso através do método 'alteraTabuleiro()'.

Aqui, usamos um método dentro do outro. Coisa que já explicamos em nosso curso de Java.

Checamos se o tiro dado 'tiro[2]' acertou através do método, que já usamos e explicamos, 'acertou'.

Caso acerte, a posição do tabuleiro referente ao tiro que você deu vai mudar de '-1' para '1'.

Caso erre, a posição do tabuleiro referente ao tiro que você deu vai mudar de '-1' para '0'.

Assim, quando o tabuleiro for exibido, já vai exibir com '*' ou 'X' no lugar desse antigo '~'.

```
import java.util.Random;
```

```
import java.util.Scanner;
```

```
public class batalhaNaval {
```

```
public static void main(String[] args) {
```

```
    int[][] tabuleiro = new int[5][5];
```

```
    int[][] navios = new int[3][2];
```

```
    int[] tiro = new int[2];
```

```
    int tentativas=0,
```

```
        acertos=0;
```

```
    inicializaTabuleiro(tabuleiro);
```

```
    iniciaNavios(navios);
```

```
    System.out.println();
```

```
    do{
```

```
        mostraTabuleiro(tabuleiro);
```

```
        darTiro(tiro);
```

```
        tentativas++;
```

```
    if(acertou(tiro,navios)){
```

```
        dica(tiro,navios,tentativas);
```

```
        acertos++;
```

```
}
```

```
else
```

```
    dica(tiro,navios,tentativas);
```

```
    alteraTabuleiro(tiro,navios,tabuleiro);
```

```
    }while(acertos!=3);
```

```
System.out.println("\n\nJogo terminado. Você acertou os 3 navios em  
"+tentativas+" tentativas");
```

```
    mostraTabuleiro(tabuleiro);
```

```
}
```

```
public static void inicializaTabuleiro(int[][] tabuleiro){
```

```
for(int linha=0 ; linha < 5 ; linha++ )
```

```
for(int coluna=0 ; coluna < 5 ; coluna++ )
```

```
    tabuleiro[linha][coluna]=-1;
```

```
}
```

```
public static void mostraTabuleiro(int[][] tabuleiro){
```

```
System.out.println("\t1 \t2 \t3 \t4 \t5");
```

```
System.out.println();
```

```
for(int linha=0 ; linha < 5 ; linha++ ){
```

```
System.out.print((linha+1)+"");
```

```
for(int coluna=0 ; coluna < 5 ; coluna++ ){
```

```
if(tabuleiro[linha][coluna]==-1){
```

```
System.out.print("\t"+"~");
```

```
    }else if(tabuleiro[linha][coluna]==0){
```

```
System.out.print("\t"+"*");
```

```
    }else if(tabuleiro[linha][coluna]==1){
```

```
System.out.print("\t"+"X");
```

```
    }
```

```
}
```

```
System.out.println();
```

```
    }
```

```
}
```

```
public static void iniciaNavios(int[][] navios){
```

```
    Random sorteio = new Random();
```

```
    for(int navio=0 ; navio < 3 ; navio++){
```

```
        navios[navio][0]=sorteio.nextInt(5);
```

```
        navios[navio][1]=sorteio.nextInt(5);
```

```
        //agora vamos checar se esse par não foi sorteado
```

```
        //se foi, so sai do do...while enquanto sortear um diferente
```

```
for(int anterior=0 ; anterior < navio ; anterior++){
```

```
if( (navios[navio][0] == navios[anterior][0])&&(navios[navio][1] ==  
navios[anterior][1]) )
```

```
do{
```

```
    navios[navio][0]=sorteio.nextInt(5);
```

```
    navios[navio][1]=sorteio.nextInt(5);
```

```
    }while( (navios[navio][0] == navios[anterior][0])&&(navios[navio]  
[1] == navios[anterior][1]) );
```

```
    }
```

```
}
```

```
}
```

```
public static void darTiro(int[] tiro){
```

```
Scanner entrada = new Scanner(System.in);
```

```
System.out.print("Linha: ");
```

```
    tiro[0] = entrada.nextInt();
```

```
    tiro[0]--;
```



```
System.out.print("Coluna: ");
```

```
    tiro[1] = entrada.nextInt();
```

```
    tiro[1]--;
```

```
}
```

```
public static boolean acertou(int[] tiro, int[][] navios){
```

```
    for(int navio=0 ; navio<navios.length ; navio++){
```

```
        if( tiro[0]==navios[navio][0] && tiro[1]==navios[navio][1]){
```

```
            System.out.printf("Você acertou o tiro (%d,%d)\n",tiro[0]+1,tiro[1]+1);
```

```
        return true;
```

```
            }
```

```
        }
```

```
    return false;
```

```
}
```

```
public static void dica(int[] tiro, int[][] navios, int tentativa){
```

```
    int linha=0,
```

```
        coluna=0;
```

```
    for(int fila=0 ; fila < navios.length ; fila++){
```

```
        if(navios[fila][0]==tiro[0])
```

```
            linha++;
```

```
        if(navios[fila][1]==tiro[1])
```

```
            coluna++;
```

```
    }
```

```
    System.out.printf("\nDica %d: \nlinha %d -> %d navios\n" +
```

```
                        "coluna %d -> %d\n",tentativa,tiro[0]+1,linha,tiro[1]+1,coluna);
```

```
    }
```

```
public static void alteraTabuleiro(int[] tiro, int[][] navios, int[][] tabuleiro){
```

```
if(acertou(tiro,navios))  
    tabuleiro[tiro[0]][tiro[1]]=1;  
  
else  
    tabuleiro[tiro[0]][tiro[1]]=0;  
  
}  
}
```

Fica como desafio você fazer um jogo Batalha Naval, mas humano x humano.
Faça assim, primeiro um dos jogadores preenche onde quer colocar seus navios.

Depois o outro.

O jogo será por turnos. Crie uma variável chamada 'turno' que é incrementada a cada rodada, e use o resto da divisão por 2 pra saber de quem é a vez.

Por exemplo, se

turno=1 -> $\text{turno} \% 2 = 1$ -> vez do jogador 1

turno=2 -> $\text{turno} \% 2 = 0$ -> vez do jogador 2

turno=3 -> $\text{turno} \% 2 = 1$ -> vez do jogador 1

turno=4 -> $\text{turno} \% 2 = 0$ -> vez do jogador 2

...

Crie duas variáveis 'tabuleiro', duas 'tentativas', duas 'navios' e duas 'acertos', para armazenar em variáveis diferentes os dados de cada usuário.

O jogo termina quando uma das variáveis 'acertos' é 3.

Ou você pode alterar o tamanho do tabuleiro, o número de navios, a dica etc...sua criatividade é que conta.

Jogos com programação gráfica em Java

Já fizemos um exemplo de jogo em Java usando caixas de diálogo, assim como fizemos o mesmo em modo texto.

Você viu que, a lógica é a mesma, e é, de longe, a parte mais importante.

Entendendo e fazendo o jogo Batalha Naval em Java em modo texto, você saberá como funciona a lógica do game. A parte gráfica usará a mesma lógica. Você irá, simplesmente, colocar alguns recursos visuais/gráficos (GUI - Graphic User Interface) nessa sua aplicação Java.

Muitas pessoas pensam que programar jogos é apenas a parte visual.

Porém, essa é apenas uma parte. Geralmente não é difícil, não é a mais importante nem a que dá mais trabalho.

Programar Jogos é, primeiro, programar. Aprenda antes a linguagem, a lógica, a sintaxe.

Depois, ao longo do curso, você aprenderá a usar os recursos gráficos do Java, como menus, janelas e botões, e iremos ensinar como usar para fazer esse mesmo jogo em uma maneira gráfica, para você enviar para seus amigos :)

Por hora, estude a lógica e tenha calma.

Programar é uma profissão. Não se cria um game ou se aprende a programar em Java em poucas semanas ou meses. Continue estudando...

Orientação a Objetos, parte III: Herança e Polimorfismo

Herança em Java - o que é, para que serve, exemplos e quando usar

Chegamos em um ponto muito importante em nosso curso de Java no que se refere a Programação Orientada a Objetos.

Herança é uma das maiores características desse tipo tão importante e usado tipo de programação.

Através desse recurso, você irá criar classes de uma maneira bem mais rápida, eficiente e fácil de manter: baseando classes em outras.

O que é herança em Java - Superclasse e subclasse

O que é herança, na vida real?

É quando um pessoa deixa seus bens para outra. No Java também.

Geralmente a herança ocorre dentre membros de uma mesma família. No Java também.

Herança, em Java, nada mais é do que criar classes usando outras classes já existentes.

Obviamente, você vai fazer uma classe herdar as características de outra se estas tiverem uma relação (se forem parecidas).

Outro ponto importante é que, quando fazemos uso da herança, nós podemos adicionar mais atributos a classe.

Exemplo 1: Carros e motos

Imagine que você tem uma revenda de veículos: carros e motos.

Todos são veículos. Ora, crie a classe "Veiculo".

O que esses veículos tem em comum?

Motor, preço, marca, nome do cliente que vai comprar, quantos quilômetros fazem com 1 litro de combustível etc.

Todos os objetos da classe "Veiculo" tem essas características.

Porém, existem algumas características que as motos tem que um carro não tem: capacete, somente duas rodas, cilindrada etc.

Também existem características que os carros tem que as motos não tem: podem ter 4 portas, banco de couro, ar-condicionado etc.

Para resolver esse problema, e deixar a aplicação MUITO MAIS ORGANIZADA, vamos fazer duas classes: "Moto" e "Carro".

Cada uma dessas irá herdar a classe "Veiculo", pois também são um tipo de veículos.

Dizemos que "Veiculo" é a superclasse, "Moto" e "Carro" são subclasses.

Faz sentido, não?

Exemplo 2: Alunos de uma escola

Se for criar uma aplicação para um colégio, muito provavelmente você vai criar uma classe chamada "Aluno".

O que todo aluno tem em comum, para colocarmos nessa classe?

Tem seu nome, cursa uma série, tem seu número de matrícula etc.

Mas um aluno do 2o grau tem matérias como Física, Biologia e Química.

Coisa que aluno do ensino fundamental não tem.

Ou seja, se você tentar criar uma classe e colocar as matérias Física, Química e Biologia para um aluno que não cursa essas matérias, estará tendo problemas,

pois muito provavelmente você deveria preencher as notas e faltas dessas matérias no construtor.

E pra calcular a média? A média inclui todas as matérias, ora. Mas os alunos de uma escola não tem as mesmas disciplinas.

E agora? Criar vários métodos diferentes dentro da classe, e acionar o método certo para o cálculo da média através de um booleano?

Claro que não! Vamos criar outras classes: "fundamental" e "medio".

A nossa classe mãe - ou superclasse - será a "Aluno", com dados e disciplinas QUE TODOS os alunos possuem em comum.

Fazemos "fundamental" e "medio" herdarem as características da "Aluno", e dentro de cada subclasse (ou classe filha) dessas irá ter seus métodos e variáveis a mais.

Note que, ao herdar, você é mais eficiente, pois não precisa criar duas classes de alunos diferentes.

Ao herdar, automaticamente as subclasses irão possuir as mesmas variáveis e métodos.

Exemplo 3: Funcionários de uma empresa

Ok, você já sabe que os funcionários da tesouraria, do suporte técnico e do alto escalão são diferentes, fazem diferentes tarefas e tem diferentes características profissional.

Então você cria uma classe para cada tipo de setor: "FuncionarioTesouraria", "FuncionarioSecretaria", "FuncionarioTI" etc.

Uma trabalhadeira...ao término disso tudo, terá dezenas de classes só para representar os funcionários.

Porém, as redes sociais, como Facebook e Twitter estão muito na moda e são usadas, inclusive, em meios profissionais. Sabia que as empresas olham o facebook de um candidato ao emprego antes de contratar?

Então seu chefe pediu para você adicionar o campo "Facebook" no registro de cada funcionário.

Nossa! E agora? Vai em cada uma das dezenas de classes, adicionar esse campo?

Agora não vai mais, pois aprendeu sobre herança aqui no Java Progressivo (e de graça, que bom hein?).

Uma boa solução seria criar a superclasse "Funcionario", com dados que TODOS os funcionários possuem: idade, nome, salário, setor que trabalha etc.

Qual a vantagem disso?

Ora, se você adicionar o campo "facebook" na classe "Funcionario", todas as subclasses passarão a ter esse campo! Pois todas as subclasses (Tesouraria, Secretaria, TI...) são também classes do tipo "Funcionario".

Notou a importância e a mão-na-roda que é herança?

Quando e como saber que é hora de usar herança em Java - Relação 'é um'

Para saber quando usar, e detectar o uso de herança, use a relação 'é um'.

Nos exemplos anteriores:

"Moto" é um "Veiculo", e "Carro" é um "Veículo".

"alunoMedio" é um "Aluno", assim como "alunoFundamental" é um "Aluno".

"FuncionarioTesouraria" é um "Funcionario" e também "FuncionarioTI" é um "Funcionario".

Bem simples e óbvio, não?

Essa é a vantagem do Java e da programação orientada a objetos(POO).

POO é uma imitação do mundo real.

Hierarquia - subclasse de subclasse

Você notou que, por exemplo, dentro do setor "Tesouraria" de uma empresa, podem existir diversos tipos de profissionais, com características únicas e diferentes?

Ora, uma secretária faz uma coisa. O chefe da tesouraria faz outras coisas.

Logo, uma única classe "FuncionarioTesouraria" não seria o suficiente para definir todos os trabalhadores dessa seção?

O que fazer, então?

Crie classes que herdam a classe 'FuncionarioTesouraria' para cada setor da Tesouraria.

Aí teríamos que a 'FuncionarioTesouraria' seria a superclasse da subclasse "TesourariaSecretarias", que tem os atributos que só as secretárias da seção da tesouraria teriam.

Mas "FuncionarioTesouraria" continuaria sendo uma subclasse da "Funcionario".

E "TesourariaSecretarias" também são subclasses da classe "Funcionario", afinal, todos são funcionários.

Então "Funcionario" é superclasse de "FuncionarioTesouraria" e também é superclasse de "TesourariaSecretarias".

Porém, somente a classe "FuncionarioTesouraria" é superclasse direta da "TesourariaSecretarias".

Bem óbvio e parecido com o mundo.

Uma maravilha essa Orientação a Objetos

Outra vantagem da Herança - Organização

Muitas vezes, durante sua carreira de programador, você terá que estudar o código de outras pessoas.

Se o outro programador que você está estudando (ou decifrando) seu código não tiver estudado no Java Progressivo, vai ser muito desorganizado.

Contextualizando com nossos exemplos, será bem comum você ver:

"TesourariaFuncionarios", "TesourariaSecretarias", "CozinheiraExecutivos", "FuncionariosSecretaria", "Chefes" etc.

Ou seja, muitas classes, muitas informações e tudo completamente desorganizado.

Ao criar superclasses e subclasses você terá uma hierarquia.

Sua IDE vai mostrar as superclasses e suas respectivas subclasses em uma árvore, tudo bem bonito e organizado.

Ao chegar e ver isso na sua IDE:

Funcionários:

- Tesouraria:

 - Secretárias

 - Banco

- Chefes

- Secretaria

- Secretárias

- Almoxarifado

- Supervisores

- TI

- Redes

- Office

- Programação

- Banco de dados

- Java

- C

- C++

Você entende como funciona a empresa somente vendo.

Por exemplo, que tipos de funcionários de TI existem?

Os responsáveis pela Rede, outro pra ensinar e tirar dúvidas sobre o Office e os programadores.

E dentre os programadores, que tipos existem?

É óbvio achar e ver que existem os responsáveis pelo Banco de dados, os que fazem aplicações em Java, C e os que fazem em C++.

Organizado, não?

Herança de construtores e Override

Agora que você já sabe o que é herança, sua importância, quando usar e viu vários exemplos práticos do mundo real, vamos mostrar como fazer uma classe herdar as características de outra.

Nesse tutorial de Java ensinaremos também a relação dos construtores com as superclasses e subclasses, e como usar o tão importante `@Override`.

Sintaxe de declaração da Herança

Se quisermos declarar uma classe "Filha" que herda os atributos do pai, simplesmente usamos a palavra "extends" na hora de criar a classe:

```
public class Filha extends Pai{  
  
}
```

Herança: o construtor da Subclasse sempre invoca o construtor da Superclasse

Um fato importante: os construtores são algo único a cada classe, portanto não são herdados.

Porém, é possível invocar os construtores de uma superclasse através da subclasse.

Vale lembrar que, para uma aplicação funcionar corretamente, algumas variáveis devem ser iniciadas. Algumas dessas variáveis são iniciadas em uma superclasse, portanto, sempre que o método construtor de uma subclasse roda, roda também o construtor de sua superclasse.

Usando um dos exemplos de nosso tutorial passado sobre herança, na classe "AlunoMedio", que tem algumas disciplinas (como Física e Química), que não existem na classe dos alunos do ensino médio, ela foi herdada da classe "Aluno".

Porém, no construtor da superclasse "Aluno" é que são inicializadas informações de todos os alunos, como nome e número de matrícula.

Portanto, a primeira coisa que construtor da "AlunoMedio" faz é rodar o construtor da "Aluno" - sua superclasse - pois informações e ações importantes ocorrem no construtor da "Aluno".

Resumindo: a primeira coisa que um construtor faz é rodar o construtor de sua

superclasse, pois ações importantes podem estar acontecendo lá - como inicialização de variáveis que poderão ser usadas na subclasse.

Vamos ver isso na prática, criando duas classes: a Pai e a Filha, e óbvio, fazendo a Filha herdar a classe Pai.

--->**Heranca.java**

```
public class Heranca {
```

```
public static void main(String[] args) {
```

```
new Filha();
```

```
}
```

```
}
```

--->**Pai.java**

```
public class Pai {
```

```
public Pai(){
```

```
System.out.println("Método construtor da classe Pai");  
  
    }  
  
}
```

--->**Filha.java**

```
public class Filha extends Pai {
```

```
public Filha(){
```

```
System.out.println("Método construtor da classe Filha");  
  
    }  
  
}
```

O resultado disso é:

Método construtor da classe Pai

Método construtor da classe Filha

Isso mostra que a primeira coisa que um construtor de uma subclasse faz é invocar o construtor da superclasse. Só depois é que esse construtor da classe Filha faz suas ações.

O exemplo a seguir mostra como a inicialização de variáveis realmente ocorre na superclasse antes de ocorrer na subclasse.

Definimos o nome do pai na classe "Pai" e fazemos a classe "Filha" herdar a pai.

Note que classe "Filha" usamos a variável 'nomePai', mas não declaramos ela na classe Filha.

Porém podemos usar esse atributo pois ele pertence a classe Pai.

Viu que esse valor já vem inicializado? Ou seja, o construtor da classe Pai realmente ocorreu antes da classe Filha.

Heranca.java

```
public class Heranca {  
  
public static void main(String[] args) {  
    Filha filha = new Filha("Mariazinha");  
  
    }  
  
}
```

Pai.java


```
public class Pai {
```

```
public String nomePai;
```

```
public Pai(){
```

```
    this.nomePai = "Neil";
```

```
    }
```

```
}
```

Filha.java

```
public class Filha extends Pai {
```

```
private String nomeFilha;
```

```
public Filha(String nomeFilha){
```

```
    this.nomeFilha = nomeFilha;
```

```
    System.out.println("O nome da filha é " + this.nomeFilha +
```

```
        " e o do pai é " + nomePai + ".");
```

```
}
```

```
}
```

Caso o 'nomePai' fosse private, você poderia ter criado um método público na classe "Pai" que retorna o nome do pai, e usar esse método na classe "Filha" sem problema algum.

Herança: quando a superclasse não tem método construtor

Ok, a chamada da superclasse sempre ocorre. Mas pode ocorrer de duas formas: ou você claramente invoca ela ou ela ocorre sozinha.

Quando ela ocorre sozinha, o método construtor chamado é o padrão, que não recebe argumentos.

Note que sua superclasse pode não conter nenhum construtor. Mesmo assim ele é invocado e nada ocorre, pois o Java sempre cria um construtor vazio, que não faz nada, para as classes que são definidas sem construtor.

Então se você não quer que o método construtor da superclasse interfira na subclasse, é só não criar nenhum construtor na superclasse, que nada ocorrerá.

Veja o exemplo a seguir, em que apenas definimos a variável 'nomePai' na classe "Pai". Ela não tem nenhum construtor.

Então, ao ser invocado o construtor da "Pai" na "Filha", nada ocorre na "Filha".

Heranca.java

```
public class Heranca {
```

```
public static void main(String[] args) {
```

```
Filha filha = new Filha("Mariazinha");
```

```
}
```

```
}
```

Pai.java

```
public class Pai {
```

```
public String nomePai;
```

```
}
```

Filha.java

```
public class Filha extends Pai {
```

```
private String nomeFilha;
```

```
public Filha(String nomeFilha){
```

```
this.nomeFilha = nomeFilha;
```

```
System.out.println("O nome da filha é " + this.nomeFilha +  
    " e o do pai é " + nomePai + ".");  
  
}  
  
}
```

O que é e como fazer Override em Java

Suponha que na classe Pai tenha o método nome(), que mostra uma String na tela, com o nome da pessoa:

```
public void nome(){  
  
    System.out.println("O nome do pai é " + this.nomePai + ".");  
  
}
```

Se você usar esse método na classe Filha verá o nome do pai, correto?

Claro, pois a classe filha herda os métodos também da classe pai também.

Mas esse método retorna o nome da pessoa da classe. Não faz sentido ver o nome do pai, quando invocamos esse método na classe filha.

Para isso, vamos criar um método próprio da classe "Filha", que retorne o nome dela, e não o do pai.

Ficará assim:

```
public void nome(){  
  
    System.out.println("O nome da filha é " + this.nomeFilha + ".");  
  
}
```

```
}
```

Porém vamos usar o mesmo nome nesse método, 'nome()'.

E agora? Ao chamar esse método, o que Java vai mostrar? O método da classe Pai ou da classe Filha?

Não vamos nos estressar com o que ele vai mostrar, pois vamos usar um Override, ou seja, vamos sobrescrever um método.

O método original, é claro que é o da classe "Pai".

Porém, quando criarmos uma classe filha e invocarmos o método 'nome()' queremos que apareça o nome da filha, e não o do pai. Então queremos que o método chamado seja o método da subclasse e não o método da superclasse.

Para dizer isso ao Java escrevemos "@Override" antes do método da subclasse.

Então, nosso programa fica assim:

Heranca.java

```
public class Heranca {
```

```
public static void main(String[] args) {
```

```
    Filha filha = new Filha("Mariazinha");
```

```
    Pai pai = new Pai();
```

```
        filha.nome();  
        pai.nome();  
    }  
  
}
```

Pai.java

```
public class Pai {
```

```
    public String nomePai;
```

```
    public Pai(){
```

```
        this.nomePai="Neil";
```

```
    }
```

```
    public void nome(){
```

```
        System.out.println("O nome do pai é " + nomePai + ".");
```

```
    }
```



```
}
```

Filha.java

```
public class Filha extends Pai {
```

```
private String nomeFilha;
```

```
public Filha(String nomeFilha){
```

```
    this.nomeFilha = nomeFilha;
```

```
    }
```

```
@Override
```

```
public void nome(){
```

```
    System.out.println("O nome da filha é " + this.nomeFilha + ".");
```

```
    }
```

```
}
```

Pronto. Com o `@Override`, o Java vai mostrar o nome da filha, caso o objeto seja

do tipo Filha.

E vai mostrar o nome do pai caso o objeto seja apenas do tipo Pai.

super: Chamando o construtor da Superclasse

Muitas vezes, em nossas aplicações Java, criamos um objeto que é subclasse de outra.

Porém passamos 'direto' pela superclasse e vamos usar diretamente o objeto da subclasse.

Mas acontece que a subclasse depende das variáveis inicializadas na superclasse.

E como vai ser inicializada essas variáveis, já que não criamos um objeto da superclasse, e sim da subclasse?

A resposta é: usando a keyword `super`.

Com a `super`, nós chamamos o construtor da superclasse.

Por exemplo, vamos supor que queiramos criar somente um objeto da classe "Filha". Ou seja, não queremos criar a "Pai". Somente a filha!

Mas a filha usa o nome do pai. Então vamos chamar o construtor da Pai no construtor da Filha, passando o argumento para ele (que é o nome do pai).

Nosso código ficará assim:

Heranca.java

```
public class Heranca {
```

```
public static void main(String[] args) {  
    Filha filha = new Filha("Mariazinha", "Neil");  
  
    filha.nome();  
}  
  
}
```

Pai.java

```
public class Pai {  
  
    public String nomePai;  
  
    public Pai(String nomePai){  
  
        this.nomePai=nomePai;  
    }  
  
    public void nome(){
```

```
System.out.println("O nome do pai é " + nomePai + ".");  
  
    }  
  
}
```

Filha.java

```
public class Filha extends Pai {
```

```
private String nomeFilha;
```

```
public Filha(String nomeFilha, String nomePai){
```

```
    super(nomePai);
```

```
    this.nomeFilha = nomeFilha;
```

```
    }
```

```
@Override
```

```
public void nome(){
```

```
    System.out.println("O nome da filha é " + this.nomeFilha + ", e do pai
```

```
" "+nomePai+"");
```

```
}
```

```
}
```

Herança ou Composição: Qual o melhor?

Acabamos de estudar a Herança, que assim como a Composição, é uma forma de tratar a relação entre os objetos, como eles devem ser vistos e trabalhados.

Talvez agora, como iniciante em nossa apostila de Java, isso não seja motivo de muita preocupação para você, mas certamente no futuro será. E para melhor esclarecer isso, nossa apostila de Java irá falar mais sobre as vantagens e desvantagens de cada uma dessas técnicas.

Herança ou Composição? Qual usar?

Quando estudamos os conceitos de Herança pela primeira vez, parece ser um pouco confuso.

Com um pouco mais de dedicação, começamos a entender e ver que as coisas fazem sentido.

E quando vemos exemplos de aplicações reais, como os exemplos e exercícios mostrados por nós, é que vamos o quão interessante é a Herança.

E não é pra menos, pois podemos herdar variáveis, métodos e muitas linhas de código apenas escrevendo uma palavra: `extends`.

Sem dúvida, é uma ideia fantástica e um dos recursos mais utilizados, que fazem com que o paradigma de orientação à objetos seja muitíssimo usado.

Porém, se notar bem, fornecemos duas maneiras de se trabalhar com objetos de classe diferentes, através da Composição e através da Herança. Então naturalmente surge a dúvida? Qual usar?

Qual a melhor de se trabalhar?

A resposta, como a maioria das coisas em programação (e em computação, de um modo geral), é a mesma:

depende. Depende dos seus objetivos, depende da situação.

Nenhuma é inútil, melhor ou pior.

E temos sorte de ter duas boas opções para usar em nossos projetos, algumas linguagens só oferecem uma maneira de resolver as coisas.

□

Herança: vantagens e desvantagens

Já que já mostramos, tanto para Composição como para Herança, seu uso e benefícios, vamos falar de alguns problemas.

Basicamente, as coisas vão girar em torno do acoplamento.

Na sua experiência de vida, o que seria um acoplamento?

Duas peças de carro bem acopladas?

Seriam duas peças totalmente encaixadas e unidas, que praticamente não dá pra desgrudar e funcionam, para aquele propósito, perfeitamente, mas para outros, não.

Afinal, dá muito trabalho usar para outros fins coisas que são bem acopladas.

Herança é isso.

Quando uma classe herda outra, ela está MUITO ligada à ela. Ela está altamente acoplada, interligada, dependente.

Se a classe super muda, ela muda. É um processo estático.

Fez isso? Tá feito, não muda mais, só programando e compilando tudo de novo.

E qual a parte ruim disso? Nem sempre queremos as coisas tão acopladas.

Talvez você queira trabalhar com objeto mais simples, como o "Opala", que é derivado da classe "Carro".

Porém, nessa classe "Carro" você criou diversas variáveis e métodos que não vão existir na classe "Opala", como GPS embutido e injeção eletrônica.

Ou seja, as vezes você herda coisas que não precisa. E para usar essa classe, teria que ter toda a hierarquia acima dela.

Há também problemas de segurança, da preocupação de proteger certos métodos e variáveis.

Pense bem ao estender uma classe: preciso mesmo repassar esse método? E essa variável?

Talvez você coloque uma variável como protected e outra como private, para proteger.

Mas percebe que tinha usado em outra classe esses dados, como se fossem public.

Notou? Pra uns é private, pra outros é public, uns podem acessar, outros não, as vezes depende.

A medida que o projeto cresce, esse acoplamento entre classes pode se tornar complexo.

É ruim? Em alguns casos sim, em outros a Herança pode ser perfeita.

Composição: vantagens e desvantagens

A composição é o contrário, no quesito acoplamento.

Apenas passamos um objeto para outro (ou usamos uma classe em outra).

É uma troca de informações. Uma não depende da outra para existir, não herda nada.

Ou seja, as classes e objetos em uma Composition estão mais livres, mais soltos. Assim é bem mais fácil reutilizar código em uma Composição

São como as peças de um relógio ou de um computador. Funcionam todas juntas, elas se encaixam como um quebra-cabeça.

Mas nada te impede de tirar uma peça e colocar outra.

Ou inventar uma nova peça para acoplar ali. As coisas não são muito estáticas e imutáveis.

Outra vantagem é que quando passamos um objeto, quem o controla pode-se utilizar de apenas um ou outro método ou dado, sem se dar conta do resto.

Não precisa passar todas as informações. Diferente da herança, onde uma classe filha ao herdar, recebe muita coisa da classe mãe. Algumas dessas coisas nem sempre devem ser repassadas.

E a desvantagem? Bom, no caso da herança, como mostramos, as vezes economizamos MUITO trabalho herdando métodos e variáveis.

As vezes simplesmente colocamos um extends, alteramos um ou outro método, mexemos em duas ou três linhas e temos as coisas funcionando.

Delegando na Composição

Quando enviamos um Objeto/Classe para outro, este deverá saber EXATAMENTE o que deve fazer.

Por isso, em composição, é bem comum o conceito de delegação, que é quando você envia informações para objeto(s)/classe(s), onde cada um irá fazer uma coisa diferente e específica.

É como se você fosse o chefe, e vai pedir para o programador começar a desenvolver o aplicativo de um projeto, para o design fazer a arte e para o webmaster criar o site do projeto.

Você está delegando a cada um deles uma tarefa.

Eles vão receber este objeto, o projeto, e irão fazer as alterações nele.

Irão checar dados deste objeto, se já foi criado isso, aquilo, se já foi alterado, se é realmente para fazer esta mudança ou não.

O contrário também ocorre: quando um objeto vai ser enviado, ele pode ir na classe que o vai receber e checar alguns dados e informações.

Esta 'consulta' de dados é chamada de callback.

Quando criamos o Jogo da Velha em Java, criamos uma classe chamada "Jogo",

que vai receber dois "Jogadores" e o "Tabuleiro". É uma composição, pois essa classe Jogo recebe os objetos dos jogadores bem como um tabuleiro, e vai 'mandando' nas coisas.

É como se fosse um gerente, um chefe: olha se o tabuleiro tá vazio, se alguém ganhou, quantas jogadas fez cada jogador, checa se este já decidiu a linha e coluna que vai jogar, se a casa que ele jogou está mesmo vazia etc etc.

Resumindo: criamos a classe "Jogo" com uma série de métodos que irão fazer as coisas funcionarem, onde delegamos uma função à cada um deles, de modo que, quando todos trabalham juntos, o jogo funciona.

Onde estudar mais sobre Herança e Composição

E o assunto rende, cada caso é um caso, e é sempre bom ler mais sobre o assunto e experiências de outros programadores.

Por isso vamos deixar dois ótimos links para vocês se aprofundarem no assunto:

[Blog da Caelum - Como não aprender Orientação a Objetos - Herança](#)

[Macoratti - OOP: Herança x Composição](#)

Interface em Java (implements) - O que é, para que serve e como implementar

Iremos agora estudar um assunto muito importante em nossa apostila de Java, que irá usar principalmente, dentre outras coisas, os conceitos aprendemos sobre Classes e Métodos Abstratos, as interfaces.

Aprenderemos também uma importante palavra reservada do Java, o `implements`.

O que é uma Interface

O conceito de interface é usado em praticamente toda a computação, até mesmo na engenharia do ramo programação em baixo nível (interface de comunicação entre hardware e o kernel do Sistema Operacional).

Podemos definir interface como uma face, um plano, uma divisa que faz a comunicação entre dois meios diferentes.

Por exemplo, quando trabalhamos com eletrônica, basicamente é tudo estado alto ou estado baixo, que são nada mais que níveis de voltagem ou corrente elétrica. Lá se usa registros, CI (circuitos integrados), portas lógicas, equipamentos de elétrica e eletrônica.

Mas quem usa computador como simples usuário, não precisa saber nem se preocupar com eletricidade ou bits se movendo. Por quê?

Porque há uma interface que interpreta essas coisas em baixo nível. Em vez de te mostrar uma série de bits ou voltagens, te mostra textos que você, humano, compreende.

O responsável por isso é o sistema operacional. É como se ele traduzisse: "Opa, recebi um código aqui do hardware "0011100101001011010", isso quer dizer que tenho que fechar a janela".

Ou seja, estamos fazendo uma comunicação entre dois meios: o meio do hardware (em baixo nível, onde se trabalha com bits) e o meio do usuário (que quer ler textos, ver imagens).

Então, há uma interpretação. No fundo, a mensagem é a mesma, só é dita de maneira diferente para que você possa entender.

Interface em Java

No rigor do termo, uma interface em Java nada mais é que uma classe abstrata composta somente por métodos abstratos. E como tal, obviamente não pode ser instanciada.

Ou seja, ela só contém as declarações dos métodos e constantes, nenhuma implementação, só o 'molde'.

E para que raios serve uma classe sem implementação?

Ela serve para que outras classes, baseadas nessa interface, implementem esses métodos para fins específicos.

Parece confuso e complicado, mas é como explicado anterior sobre interface: ela será uma espécie de comunicação entre meios.

Geralmente entre o que é pedido (das funções que ela executa) e a implementação.

Vamos dar alguns exemplos mais práticos.

Interface de um restaurante

Um exemplo de interface em nosso dia-a-dia, é o cardápio de um restaurante.

No menu do restaurante tem dizendo todas as refeições existentes naquele local, inclusive com os ingredientes.

Os clientes que lá forem vão pedir uma determinada comida, e querem receber exatamente aquilo que pediram. Porém, eles não vão ver e nem se importar de como foi feito o prato, só querem receber aquilo que foi solicitado.

Então, o cardápio é uma interface de comunicação entre o cliente os cozinheiros.

Interface de um programa real

Vamos supor que cientistas te contratem para criar um aplicativo Java para uma universidade, para fazer cálculos com matrizes. Eles irão te passar o que você tem que fazer, porém, muito provavelmente não vão estar interessados em COMO você vai fazer isso, só querem saber se vai funcionar.

E é aí que entra a interface, ela servirá de comunicação entre seu código e os cientistas.

Sua interface vai se comunicar com eles, é como se ela dissesse: "Para criar uma matriz, forneçam isso que te devolvo a matriz. Já para calcular o traço da matriz, nos forneça a matriz. Já para calcular a transposta, faça isso que retorno isso. Ah quer essa informação? Ok, me forneça esses dados que te retorno tudo pronto"

Ou seja, vai haver uma comunicação entre a sua implementação em Java e entre os cientistas.

Provavelmente eles nem sabem programar em Java, por isso eles não devem ter acesso ao código, pois não iriam entender, e nem querem entender, só querem que o programa faça o que eles pedem, e essa mediação é feita através da classe abstrata, a interface Java do programa.

Como declarar uma Interface

Embora seja uma classe abstrata, ela é uma classe abstrata especial, pois só possui métodos abstratos e nada de implementação. Só as estruturas, o cabeçalho dos métodos.

E como tal, vamos usar uma palavra reservada do Java para declarar tais tipos de classe: interface.

Por exemplo, para declarar uma interface pública de nome "JavaProgressivo", fazemos:

```
public interface JavaProgressivo{  
  
}
```

Nossa interface do exemplo passado, das matrizes dos cientistas, poderia ser mais ou menos assim:

```
public interface Matriz {  
  
    double Traco(Matriz m);  
  
    Matriz Nula(int linha, int coluna);  
  
    Matriz Transposta(Matriz m);  
  
}
```

}

Vejam que não há implementação, só o cabeçalho dos métodos.

Mas vamos ver como essa interface vai se comunicar com os cientistas?

Por exemplo, é fácil ver que se eles quiserem saber o traço de uma matriz, basta invocar o método "Traco()" fornecendo a matriz como argumento.

Como esse traço é calculado? Não sei, não importa, o importante é que retorna uma variável do tipo double com o resultado.

E para criar uma matriz nula de "linha" linhas e de "coluna" colunas?

É só chamar o método Nula() que ele te retorna tal matriz.

E se eles quiserem a matriz transposta de uma matriz 'm', como fazem?

Ué, chamam o método "Transposta()" passando a matriz como argumento, que o programa em Java retorna a matriz transposta.

Implementando uma classe - implements

Porém, de nada adianta ter uma classe abstrata se ela não for implementada.

Veja a interface como um molde, um esqueleto do que você deve fazer.

É como se seu chefe te desse uma série de tarefas para fazer:

Crie um método que retorne o traço de uma matriz

Crie outro método que retorne uma matriz nula de tamanho "linha x coluna"

E outro que retorne a transposta de uma matriz

E para essas tarefas serem resolvidas, é necessário implementá-las.

É aí que entra a parte do código Java, onde você vai realmente programar tudo que foi pedido.

Vamos supor que sua implementação será através da classe pública "minhaMatriz", a sintaxe para implementar uma classe derivada de uma interface é:

```
public class minhaMatriz implements Matriz{  
  
}
```

Veja como essa sintaxe se assemelha aquela da Herança.

A única diferença que lá a palavra reservada para herdar é `extends`.

De resto, tudo igual.

Agora você cria sua classe "minhaMatriz".

Você pode definir novas variáveis, novos métodos (como setters e getters) e fazer tudo o que quiser, com somente uma condição:

Tudo que estiver na interface, tem que estar, obrigatoriamente, em sua implementação, e da mesma maneira.

Ou seja, se lá tem um método que recebe argumentos específicos e retorna um tipo específico, sua implementação tem que obedecer esta mesma regra: receber os mesmos argumentos específicos e retornar o tipo que diz lá.

Seria algo assim:

```
public class minhaMatriz implements Matriz{
```

```
    Matriz mat;
```

```
    double traco;
```

```
public double Traco(Matriz m){
```

```
        //Implementação do traço de uma matriz

    return traco;

    }

    public Matriz Nula(int linha, int coluna){

        //Zerando todos os elementos da matriz


    return mat;

    }

    public Matriz Transposta(Matriz m){

        //Armazenando a transposta da matriz 'm' na matriz 'mat


    return mat;

    }

}
```

No próximo tutorial vamos implementar a famosa interface "Comparable" e seu método "compareTo()" que serve para comparar dois objetos de quaisquer tipo de classes.

Como comparar objetos - Classe abstrata Comparable e o método compareTo

No tutorial passado de nossa apostila de Java, falamos sobre as Interfaces em Java, que fazem uso de diversos conceitos de Orientação a Objetos que estamos estudando, como Polimorfismo e Classes e Métodos Abstratos.

Como de praxe, iremos ensinar mais uma lição de Java englobando diversos assuntos.

Hoje vamos aprender como comparar objetos (maior, menor, igual etc), e para isso faremos uso da classe abstrata Comparable e de seu método compareTo().

E para ver uma utilidade prática desses assuntos, vamos mostrar um exemplo através da ordenação de elementos de um Array.

Comparando objetos em Java

Como se compara uma coisa com outra?

Analisamos uma característica que deve ser mensurável.

Por exemplo, para comparar dois números, fazemos uso de seus valores.

Mas poderíamos comparar o nome deles, através da ordem alfabética.

Embora 1 seja maior que 2 na comparação de seus valores, "um" é maior que "dois", pois a letra "u" vem depois da letra "b".

Ou seja, não faz sentido comparar duas coisas, e sim duas características específicas e mensuráveis.

Por exemplo, não se compara banana com maçã.

Mas podemos comparar o tanto de vitaminas em cada uma delas, ou seu peso, pois são características mensuráveis.

E objetos? Como comparamos um objeto?

Como comparo um objeto "secretário" com o objeto "gerente", ambos a classe "Funcionários" de uma empresa?

Se tiver captado a ideia, verá que não dá pra se comparar assim.

Mas podemos comparar uma características específica de cada objeto, como o salário de cada funcionário, ou a data de entrada na empresa ou seus números de identificação.

Portanto, quem vai comparar é que escolhe como vai ser a comparação.

A classe abstrata Comparable e o método compareTo()

Existe uma classe especial em Java que é responsável por fazer comparações de objetos, que é a classe Comparable. Ela usada como padrão de comparação.

Por exemplo, vimos em nossos tutoriais sobre Arrays que existe um método chamado sort(), que ordena os elementos de um Array, do menor para o maior.

Porém, como explicamos no tópico passado, não se compara coisas genéricas, como objetos, e sim coisas específicas, que podem ter seus valores medidos.

Ora, se não dá pra comparar, como a classe Comparable serve para comparar?

Se eu quiser comparar um carro com outro, através da classe Comparable, o que ela vai comparar, então?

A resposta está no tutorial passado, sobre classes e métodos abstratos: ela serve para comparar qualquer coisa, mas nada em específico.

Essa comparação é feita através do método compareTo(Object o), que recebe um objeto (se lembre que todos os objetos são derivados da classe Object, logo, todo e qualquer objeto que é possível criar em Java é derivado de Object).

Esse método retorna 3 números: -1, 0 ou 1.

Vamos supor que temos um objeto "X" e queremos comparar com o objeto "Y".

Usamos esse método assim: X.compareTo(Y)

Caso "X" seja maior que "Y", o método retorna 1.

Caso "X" seja menor que "Y", o método retorna -1.

Caso "X" seja igual à "Y", o método retorna 0.

Ok. Já sabemos que essa classe compara objetos, e que o método retorna o valor dessa comparação (-1, 0 ou 1). Mas O QUÊ e COMO essa classe e esse método comparam?

Que característica do objeto?

Ela não diz, pois não pode adivinhar o que raios você vai querer comparar.

Então é você que vai escolher, que característica do objeto comparar.

Ou seja, você vai implementar a comparação.

Implementar, isso te lembra algo? Sim, implements.

A classe Comparable está lá, com seu método compareTo(), declarados e prontos para serem usados.

Mas a maneira que é feita essa comparação, você programador Java que decide.

Vamos criar um exemplo para você ver melhor como as coisas funcionam!

Exemplo de código - Comparando Objetos

Crie diversos objetos do tipo "Carro", onde eles tem um nome e um ano de fabricação.

Coloque esses carros em um Array, e usando o método sort, ordene esses objetos de modo a imprimir a lista de nomes e carros, ordenada por ano de fabricação do carro, do mais antigo para o mais novo.

Primeiramente, vamos criar a classe "Carro".

Ela é bem simples, tem dois atributos: o "nome" do carro e seu "ano", bem como seus métodos getters.

Porém, vamos querer comparar objetos dessa classe.

Logo, os objetos tem que se 'comparáveis' em Java, e para isso, basta fazer com que a classe "Carro" seja um implemento (implements) da interface "Comparable":

```
public class Carro implements Comparable{  
  
}
```

Bom, mas essa classe abstrata tem o método compareTo, que recebe um objeto.

E como todo método de uma classe abstrata, esse método DEVE ser implementado!

Como explicamos, ele tem que retornar um inteiro.

Se quiser usar ele junto com outras funcionalidades do Java, você deve fazer isso de modo que, ao comparar um objeto "X" com um método "Y", tenhamos que fazer: `X.compareTo(Y)`

E, como já havíamos dito, devemos fazer:

Caso "X" seja maior que "Y", o método retorna 1.

Caso "X" seja menor que "Y", o método retorna -1.

Caso "X" seja igual à "Y", o método retorna 0.

Então vamos lá!

Como queremos comparar o ano dos carros, devemos comparar o atributo "ano".

E para isso, basta usar o getter desse atributo.

A única diferença que temos nesse caso é que o método `compareTo` recebe um tipo bem genérico, o tipo `Object`.

Devemos fazer um casting, que é como se estivéssemos dizendo ao Java que objeto específico é esse `Object`, pois vamos usar o método `getAno()`, e o `Object` obviamente não tem esse método.

Vamos chamar esse `Object` recebido pelo método `compareTo()` de "o".

Vamos armazená-lo na variável "car" do tipo "Carro". O casting é feito assim:

```
Carro car = (Carro) o;
```

Pronto! Agora só devemos comparar os anos dos carros.

Na classe "Carro", vamos comparar o ano armazenado na variável "ano" com o ano do carro que recebemos por meio do objeto "o".

Aqui não tem segredo, simplesmente usamos os testes condicionais IF ELSE.

Como queremos comparar o objeto "X" com "Y" através de : X.compareTo(Y)

É fácil ver que dentro da classe, "getAno()" retorna o ano do objeto X, e vamos comparar com o ano do objeto Y, dado por Y.getAno()

Caso "getAno() > car.getAno()", é porque o ano de X é maior, e retornamos 1.

Caso "getAno() < car.getAno()", é porque o ano de Y é maior, e retornamos -1.

Caso contrário, os anos são iguais e retornamos 0.

Veja como fica o código da classe "Carro":

Carro.java

```
public class Carro implements Comparable{
```

```
private int ano=0;
```

```
private String nome;
```

```
public Carro(int ano, String nome){
```

```
    this.ano = ano;
```

```
    this.nome = nome;
```

```
    }
```

```
public int compareTo(Object o){
```

```
    Carro car = (Carro) o;
```

```
    if(getAno() > car.getAno()){
```

return 1;

 }**else{**

if(getAno() < car.getAno()){

return -1;

 }**else{**

return 0;

 }

 }

}

public int getAno(){

return this.ano;

}

public String getNome(){

```
return this.nome;
```

```
}
```

```
}
```

Agora vamos criar nossa classe principal, que tem a main().

Nela, vamos criar um vetor de Carros, de nome "carros" e criar 4 carros nesse array:

um fusca, um gol, um fiat uno e uma hilux.

Em seguida, usamos o método sort que vai ordenar esses carros:

```
Array.sort(carros)
```

Como implementamos a classe Carros para comparar o ano do carro, esse método vai ordenar os carros de acordo com o ano, do mais velho para o mais novo.

Depois, vamos printar cada elemento do carro usando o laço for para arrays, que vai exibir o nome e ano dos carros. Veja como ficou nossa classe principal:

classeComparable.java

```
import java.util.Arrays;
```

```
public class classeComparable {
```

```
public static void main(String[] args) {
```

```
    Carro[] carros = {new Carro(1974, "Fusca"),
```

```
new Carro(2014, "Hilux"),
```

```
new Carro(2000, "Uno"),
```

```
new Carro(1998, "Gol")};
```

```
    Arrays.sort(carros);
```

```
    for(Carro car : carros)
```

```
        System.out.println(car.getNome() + "\t" + car.getAno());
```

}

}

Ao roda, o resultado é como esperávamos:

Fusca 1974

Gol 1998

Uno 2000

Hilux 2014

Exercício de Java

Crie uma classe "Funcionario", que recebe o cargo de cada funcionário bem como seu salário.

Crie um array com 5 objetos da classe "Funcionario" e em seguida os ordene com o método sort da classe Arrays, que vai ordenar os funcionários de acordo com o salário deles.

Exiba o nome e salário de cada um, e ao final o total que essa empresa gasta com esses funcionários em um ano.

private, public e protected: Protegendo suas informações em Java

Já vimos a relação de membros e métodos definidos como public e private, bem como a relação deles com o restante da aplicação.

Sabemos que membros private não podem ser acessados fora da classe em que foram declarados.

Mas em relação as suas subclasses?

E se eu quiser esconder algum dado das subclasses?

Veremos nesse artigo, de nosso curso online de Java, como proteger membros de uma classe, limitar e controlar o acesso aos dados importantes.

Herança de atributos public

Atributos public não tem segredo. Podem ser 'vistos' e acessados de qualquer parte de uma aplicação Java.

Na prática, elementos public são utilizados em variáveis e métodos universais, onde não há problema nem necessidade de segurança daquele dado. Exemplo: horas, constantes, variáveis globais, nome do programa ou da empresa etc.

O nome já fala tudo por si só, é público.

Herança de atributos private

Diferente dos elementos public, os private são informações 'escondidas', ou no mínimo controladas.

Por exemplo, é bem comum ver todas as variáveis de uma classe definidas como priva, e usar métodos public do tipo set, para dar um valor a esta variável.

Ora, se o set() for só para definir o valor da variável, essa atitude é totalmente inútil.

Só devemos usar o set() para manipular variáveis private quando fizemos algo a mais, como checar se o valor recebido é positivo, se está dentro do intervalo que esperamos ou mesmo para contar quantas vezes aquela variável foi acessada.

Isso tudo tem um motivo: variáveis private não podem ser acessadas nem por suas subclasses.

Tenha isso em mente: priva é sinônimo de segurança.

O seguinte exemplo, dando continuidade aos exemplos de Herança de nossa tutorial de Java passado, vamos mostrar que é impossível acessar diretamente a variável 'senhaPai' a partir da subclasse "Filha":

-->Heranca.java

public class Heranca {

public static void main(String[] args) {

Filha filha = new Filha();

}

}

-->Pai.java

public class Pai {

private String senhaPai;

public Pai(){

this.senhaPai = "papaidanadao";

}

}

-->**Filha.java**

```
public class Filha extends Pai {
```

```
public Filha(){
```

```
System.out.println(senhaPai);
```

```
}
```

```
}
```

Obteremos o seguinte erro:

"The field Pai.senhaPai is not visible"

Literalmente: o campo Pai.senhaPai não é visível. Nem pra classe filha a senha da classe pai é vista.

Isso que é segurança, hein?

Pense bem se seus atributos devem ser private ou não. Lembre-se que se não forem, outras pessoas - hackers - podem usar isso para obter informações de sua aplicação.

Herança de atributos protected

Vamos agora introduzir outro modo de declarar seus atributos, o modificador de acesso protected.

O protected é pra quando você não quer deixar um atributo public, livre para todos.

Porém, você quer compartilhar ele com as subclasses. O protected é um intermediário entre public e private.

É um segredo de família.

Por família entende: a superclasse, as subclasses e classes do mesmo package.

Para todas as outras classes, atributos protected são invisíveis. Ou seja, estamos protegendo do acesso externo.

Outro detalhe importante: quando atributos são declarados como public ou protected, eles continuarão como public ou protected.

Lembra que dissemos que subclasses podem ser superclasse também? Ora, basta declarar outra classe que extends a classe Filha, e teremos uma subclasse da subclasse Filha.

Então, vamos chamar essa classe de Neta.

Se declararmos uma variável como protected na classe Pai, ela será protected na

classe Filha, e será vista como protected na classe Neta. O mesmo para public.

Vamos criar dois pacotes: o package "Familia", que irá conter as classes "Heranca", "Pai", "Filha" e "Neta"; e o pacote "Vizinho", que contém a classe "Vizinho".

Para fazer isso basta escrever no início: package Familia ou package Vizinho

A senha do pai poderá ser vista por qualquer um do pacote "Familia", mas obteremos um erro quando tentarmos ver essa senha através da classe "Vizinho", pois o vizinho não faz parte do pacote "Familia".

Nosso exemplo ficará assim:

-->Heranca.java

package familia;

import Vizinho.Vizinho;

public class Heranca {

public static void main(String[] args) {

Neta neta = new Neta();

System.out.println("Senha vista da classe Herança: "+neta.senhaPai);


```
Vizinho vizinho = new Vizinho(neta);
```

```
}
```

```
}
```

-->**Pai.java**

```
package familia;
```

```
public class Pai {
```

```
protected String senhaPai;
```

```
public Pai(){
```

```
this.senhaPai = "papaidanadao";
```

```
}
```

```
}
```

-->Filha.java

package familia;

public class Filha extends Pai {

public Filha(){

System.out.println("Senha vista pela filha: "+senhaPai);

}

}

-->Neta.java

package familia;

public class Neta extends Filha{

public Neta(){

System.out.println("Senha vista pela Neta: " + senhaPai);

}

```
}
```

-->Vizinho.java

```
package Vizinho;
```

```
import familia.Neta;
```

```
public class Vizinho {
```

```
    public Vizinho(Neta neta){
```

```
        System.out.println("Senha vista pelo vizinho: "+neta.senhaPai);
```

```
    }
```

```
}
```

Obtemos o resultado:

Senha vista pela filha: papaidanadao

Senha vista pela Neta: papaidanadao

Senha vista da classe Herança: papaidanadao

Exception in thread "main" java.lang.Error: Unresolved compilation problem:

The field Pai.senhaPai is not visible

Ou seja, podemos ver a senha do Pai por qualquer uma das subclasses e classes do pacote "Familia".

Porém, ao tentarmos ver por uma classe que não faz parte nem do pacote "familia" nem é subclasse de "Pai", "Filha" ou "Neta" obtemos um erro, pois a senha do pai não será visível para o vizinho.

Seguro esse Java, não?

Não é à toa que ele é bastante usado por empresas, como seu bank online. Pode confiar no Java!

A classe Object: o que é, uso e principais métodos

Agora que já sabemos o que é herança, como a herança se comporta em relação aos construtores, uso do `@Override` e vimos os modificadores de acesso `private`, `public` e `protected`, vamos apresentar uma classe de suma importância bem como alguns de seus métodos: a classe `Object`.

O que é e para que serve a classe Object

Essa classe é importante por um motivo bem simples: toda e qualquer classe em Java é subclasse da Object.

Mesmo quando não usamos 'extends Object', o Java automaticamente faz sua classe herdar a Object.

Lembra que, várias vezes, ao longo de nosso curso online de Java usamos o método toString() sem nem ter declarado ele?

Pois é, ele já é declarado na classe Object, por isso podemos usar ele diretamente. Como nossa classe é subclasse da Object, ela também tem os métodos da Object.

Métodos da classe Object:

getClass()

Esse método retorna informações do objeto atual, como o package e o nome da classe.

Muito importante caso você tenha vários tipos de objeto, onde um herda do outro etc.

Ele vai retornar o nome da classe que objeto foi criado, e não sua superclasse.

clone()

Retorna uma referência - ou cópia - de um objeto.

Você deve implementar esse método conforme sua necessidade. Para uma cópia total, você deve implementar a cópia de cada variável corretamente.

Veja sobre a interface Cloneable:

<http://docs.oracle.com/javase/7/docs/api/java/lang/Cloneable.html>

No exemplo ao final do artigo, fizemos um `@Override` do método para que ele retorne uma referência objeto que queremos copiar.

Pode usar tanto:

```
Object copia = original.clone()
```

Ou:

```
NomeDaClasse copia = (NomeDaClasse) original.clone();
```

Mais informações em: [http://en.wikipedia.org/wiki/Clone_\(Java_method\)](http://en.wikipedia.org/wiki/Clone_(Java_method))

toString()

Retorna uma string com a package, nome da classe e um hexadecimal que representa o objeto em questão.

equals(Object obj)

Faz a comparação entre dois Objects, e retorna true se os objetos forem o mesmo, e false se não forem o mesmo.

É útil para saber se dois objetos apontam para o mesmo local na memória.

hashCode()

Esse método retorna um inteiro único de cada objeto, muito usado em Collections.

Falaremos mais dele nos artigos sobre coleções, em breve.

Veja sobre HashMap:

<http://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html>

O hashCode tem uma ligação com o método equals(), conforme você pode ver no exemplo dado, lá embaixo.

É bastante útil quando trabalhamos com arquivos, para agilizar buscas e comparações.

Outros métodos:

Veja a documentação da classe Object:

<http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html>

Exemplo dos métodos da classe Object:

-->objectClass.java

```
package teste;
```

```
public class objectClass{
```

```
    public static void main(String[] args) {
```

```
        classeTeste classe1 = new classeTeste();
```

```
        classeTeste2 classe2 = new classeTeste2();
```

```
        System.out.println("\ngetClass() da classeTeste: " + classe1.getClass());
```

```
        System.out.println("getClass() da classeTeste2: " + classe2.getClass());
```

```
        //      Object classe3 = classe2.clone();
```

```
        classeTeste2 classe3 = (classeTeste2) classe2.clone();
```

```
        System.out.println("\nObjeto classe3 é clone ao classe2?
```

```
" + classe3.equals(classe2));
```

```
System.out.println("\ntoString da classe1: " + classe1.toString());
```

```
System.out.println("toString da classe2: " + classe2.toString());
```

```
System.out.println("\nObjeto classe1 é igual classe2 ? " +  
classe2.equals(classe1));
```

```
System.out.println("Objeto classe1 é igual classe1 ? " + classe1.equals(classe1));
```

```
System.out.println("Objeto classe2 é igual classe3 ? " + classe2.equals(classe3));
```

```
System.out.println("\nHash code da classe 1: " + classe1.hashCode());
```

```
System.out.println("Hash code da classe 2: " + classe2.hashCode());
```

```
System.out.println("Hash code da classe 3: " + classe3.hashCode());
```

```
}
```

```
}
```

-->classeTeste.java

package teste;

public class classeTeste extends Object{

public String s1;

public classeTeste(){

System.out.println("Objeto da classeTeste criado!");

this.s1= "oi";

}

@Override

protected Object clone(){

```
return this;
```

```
}
```

```
}
```

--> classeTeste2.java

```
package teste;
```

```
public class classeTeste2 extends classeTeste{
```

```
public String teste;
```

```
public classeTeste2(){
```

```
System.out.println("Objeto da classe classeTeste2 criado");
```

```
}
```

```
}
```

Polimorfismo em Java: o que é, pra que serve, como e onde usar

Agora que aprendemos os conceitos mais importantes e vimos o uso da Herança em Java, vamos estudar outra característica marcante da programação Java e, de uma maneira mais geral, da programação orientada a objetos: o polimorfismo.

Com o polimorfismo vamos ter um controle maior sobre as subclasses sem ter que nos preocupar especificamente com cada uma delas, pois cada uma terá autonomia para agir de uma maneira diferente.

Definição de polimorfismo em Java

Traduzindo, do grego, ao pé da letra, polimorfismo significa "muitas formas".

Essas formas, em nosso contexto de programação, são as subclasses/objetos criados a partir de uma classe maior, mais geral, ou abstrata.

Polimorfismo é a capacidade que o Java nos dá de controlar todas as formas de uma maneira mais simples e geral, sem ter que se preocupar com cada objeto especificamente.

Mais uma vez, somente pela definição é muito complicado de entender.

Vamos partir para algo comum e de preocupação máxima de nosso curso de Java online: exemplos práticos.

Exemplo 1 de Polimorfismo: Aumento no preço dos carros

Vamos pra nossa loja de carros, onde você é o programador Java de lá.

Lembra que aprendeu, através de exemplos, a criar uma classe bem genérica, chamada "Carro"?

E depois criamos várias subclasses, de fuscas, ferraris, gols etc.

Imagine que, todo ano, todos na empresa tem um aumento.

A Ferrari teve aumento de 5%. o fusca terá aumento de 3% e o gol terá de 1%.

Note que, embora todos sejam "Carro", cada objeto terá que calcular seu aumento de forma diferente, pois terão diferentes valores de aumento. Como criar, então, um método na superclasse que atenda todas essas necessidades diferentes?

Não é na superclasse que se resolve, mas nas subclasses, criando o método 'aumento()' em cada uma.

Ou seja, vai criar vários métodos, e para fazer o aumento realmente ocorrer de maneira correta, é só invocar o método do objeto específico.

Então: objetoFerrari.aumento() é diferente de objetoFusca.aumento().

Note que usamos o mesmo nome do método para todas as subclasses, porém cada método é diferente um do outro.

Isso é o polimorfismo em ação: embora todos os objetos sejam "Carro", eles terão uma forma diferente de agir, pois implementamos os métodos de maneira diferente.

Apenas invocamos, e todo objeto sabe exatamente o que fazer.

Por isso o nome polimorfismo, pois cada objeto terá sua forma própria de como rodar, pois os métodos 'aumento()' dos objetos são diferentes.

Outra vantagem do polimorfismo: você já viu que, criando o método aumento() em toda as subclasses, ela agirão de maneira independente da superclasse e diferente de outros objetos.

Agora, quando chegar outro carro na sua loja você de adicionar o método aumento(), e terá um novo tipo de objeto, sem grandes alterações no código.

Exemplo 2 de Polimorfismo: animais mugindo, latindo, berrando...

Imagine que você é o criador do joguinho Colheita feliz ou Fazenda Feliz (sei lá), onde terá vários bichos nesse jogo.

Como você é esperto, vai logo abstrair e criar uma classe "Animal" com as características que todos tem: idade, peso, espécie etc.

Porém, nesse game, os animais fazem seu som característico: o cachorro late, o gato mia, o pinto pia, a vaca muge etc.

E aí? Como criar um método na superclasse que sirva para todos estes animais?

Ora, não cria, pois cada animal age diferente nesse aspecto.

Veja, polimorfismo é isso: embora objetos sejam da mesma superclasse, vão agir de maneira diferente em algum aspecto. Ou seja, terão várias(poli) formas diferentes (morfismo).

A saída é criar um método chamado 'som()' na superclasse (só o cabeçalho, como veremos nos próximos tutoriais) e em cada subclasse criar um método diferente, que caracterize cada bicho.

Veja que se não fizéssemos isso e invocássemos os métodos: vaca.som(), cachorro.som(), gato.som(), todos iriam fazer o mesmo barulho.

Com o polimorfismo: vaca.som() faria a vaquinha mugir, cachorro.som() faria o cachorro latir e gato.som() faria o objeto miar.

Porém, todos continuam sendo, também, objetos da classe "Animal".

E quando chegar mais animais na sua fazenda, adicione o método som() nesse animal, de modo que ele poderá agir conforme suas características.

Resumindo: polimorfismo permite que uma mesma superclasse possua subclasses com características - ou formas - diferentes.

Polimorfismo: Classes abstratas e Métodos abstratos

Desde que iniciamos nossos tutoriais sobre classes e objetos vínhamos tratando as classes como entidades 'reais' ou 'concretas', pois podíamos instanciar objetos a partir dela.

Nesse artigo vamos introduzir um conceito importante e bastante útil na orientação a objetos: a abstração.

Pode parecer estranho, ou no mínimo confuso, classificar e utilizar algo que se diz ser abstrato, mas veremos que esse tipo de idéia é bem interessante e bem mais realista que nossa noção antiga de classe e objeto.

O que são classes abstratas em Java

Ao pé da letra, a definição de classe abstrata é: classes que não serão instanciadas.

Ou seja, são classes em que não criaremos nenhum objeto dela, diretamente.

Para evitar confusão, é mais correto se referir a esse tipo de classes como superclasses abstratas.

Então, qual o sentido de se ter classes em que nunca vamos instanciar um objeto a partir delas?

Se você for perspicaz, deve ter notado as palavras 'diretamente' e 'superclasse'.

Não vamos criar objetos dessas superclasses abstratas, mas sim de suas subclasses.

Por isso que explicamos em detalhes os conceitos de herança antes de introduzir os de polimorfismo.

Classes abstratas no mundo real

Usamos classes abstratas para representar grupos que tem características comum, mas que, em alguns detalhes específicos, agem de maneira diferente.

Note que as classes abstratas são bastante relacionadas com polimorfismo em Java.

Vamos aos nossos exemplos práticos.

Se você for a uma concessionária para comprar um veículo, você nunca irá escolher um "Carro", você escolhe um gol, um fiat, enfim, um modelo específico.

Se for dono de uma empresa e for contratar alguém, você não contrata simplesmente um "Funcionario", você contrata uma secretária, um técnico de TI, enfim, você contrata alguém com uma profissão específica para fazer uma função específica.

Do tutorial passado sobre os animais no joguinho da Colheita Feliz, você não vai colocar um "Animal" no jogo, você coloca uma vaca, um cão ou uma ovelha. Ou seja, um animal específico.

Na verdade, em nosso mundo real, só existem objetos. As classes são abstratas, servem para classificar grupos de objetos.

Por exemplo, "Carro" é uma classe abstrata que reúne um conjunto de características que classificam os carros: potência do motor, possuem pneus,

possuem portas, se locomovem etc.

"Funcionario" é uma classe abstrata que reúne um conjunto de características, gerais, de um funcionário: tem salário, uma missão na empresa, pagam impostos, tem um chefe etc.

Ou seja, não existe "Carro", existe fusca ou ferrari.

Ninguém pede carne de "Animal" na churrascaria, e sim de boi ou carneiro.

E ninguém se candidata a uma vaga para trabalhar como "Funcionario".

São apenas modelos, abstrações que agrupam e definem um conjunto de objetos com características semelhantes.

Classes abstratas são superclasses que servem como modelo. Esses modelos possuem, no entanto, somente as características gerais.

Por que gerais?

Ora, não podemos dizer qual a potência do motor do "Carro", nem o número de portas ou quantos quilômetros fazem com 1 litro. Isso é uma característica de cada tipo específico de carro.

Também não podemos dizer que "Animal" berra ou late, ou se é herbívoro ou carnívoro.

Muito menos podemos especificar o salário de um "Funcionário" ou que as línguas estrangeiras que sabem ou função desempenhada, pois isso são características específicas.

Vale lembrar que podemos também ter subclasses abstratas.

Por exemplo, "Animal" é abstrato, e as subclasses "Mamíferos" e "Répteis" também seriam abstratas, pois são um modelo para um grupo mais específico e seletivo de animais.

E da classe "Carro", que subclasses, também abstratas porém mais específicas, poderíamos citar?

"Volkswagen", "Toyota" etc.

Logo, classes abstratas servem para criar um modelo, um conjunto de características bem gerais.

Para que servem as classes abstratas e os métodos abstratos em Java

Tínhamos citado, em nosso tutorial de Java passado, que através do polimorfismo as subclasses poderiam se portar de maneira diferente.

Abstração e polimorfismo estão intimamente relacionados.

Vamos criar classes abstratas, e declarar somente o cabeçalho dos métodos. Esses serão os métodos abstratos: somente vamos dizer que tipos de métodos existem, através da declaração deles.

Não podemos definir e criar o código desses métodos por uma razão bem óbvia: os métodos vão se comportar de modo diferente nas subclasses, ora. Porém, eles terão o mesmo nome.

Declarando classes abstratas e métodos abstratos em Java

Usamos, em ambos casos, a keyword 'abstract' logo após o modificador de acesso (ou seja, logo após public, private ou protected):

classes: `public abstract class Animal { ... }`

método: `public abstract void som() { ... }`

Quando for implementar os métodos nas subclasses, não esqueça de usar o `@Override`.

Exemplo de código em Java: Polimorfismo e abstração dos animais

Note que podemos declarar variáveis, criar construtores e métodos que serão o mesmo em todas as subclasses.

Porém, os métodos que serão diferentes em cada subclasse nós apenas declaramos seu cabeçalho.

Vamos usar as variáveis "nome" e "numeroPatas" na superclasse abstrata, porém o método som(), que cada objeto emite, será diferente.

Depois, criamos um objeto para cada tipo diferente de objeto e invocamos o método som() de cada um, e você verá que, embora todos façam parte da superclasse abstrata "Animal", eles vão agir de forma diferente.

É a união da herança, polimorfismo e abstração:

-->**Bichos.java**

```
public class Bichos {
```

```
public static void main(String[] args) {
```

```
    Vaca mimosa = new Vaca();
```

```
    Gato bichano = new Gato();
```

```
    Carneiro barnabe = new Carneiro();
```

```
mimosa.som();
```

```
bichano.som();
```

```
barnabe.som();
```

```
}
```

```
}
```

-->Animal.java

```
public abstract class Animal {
```

```
protected String nome;
```

```
protected int numeroPatas;
```

```
public abstract void som();
```

```
}
```

-->Vaca.java

```
public class Vaca extends Animal {
```

```
public Vaca(){
```

```
    this.nome = "Vaca";
```

```
    this.numeroPatas = 4;
```

```
    }
```

```
    @Override
```

```
public void som(){
```

```
    System.out.println("A " + this.nome + " que tem " + this.numeroPatas + " patas,  
    faz MUUUU");
```

```
    }
```

```
}
```

```
-->Gato.java
```

```
public class Gato extends Animal{
```

```
public Gato(){
```

```
    this.nome = "Gato";
```

```
    this.numeroPatas = 4;
```

```
    }
```

```
    @Override
```

```
public void som(){
```

```
    System.out.println("O " + this.nome + " que tem " + this.numeroPatas + " patas,  
    faz MIAU");
```

```
    }
```

```
}
```

```
-->Carneiro.java
```

```
public class Carneiro extends Animal{
```

```
public Carneiro(){
```

```
this.nome = "Carneiro";
```

```
this.numeroPatas = 4;
```

```
}
```

```
@Override
```

```
public void som(){
```

```
System.out.println("O " + this.nome + " que tem " + this.numeroPatas + " patas,  
faz BÉÉÉ");
```

```
} }
```

Importância do Polimorfismo e Abstração em softwares

O uso dessas duas idéias, ou técnicas, em Java é de vital importância no mundo dos softwares.

Usamos isso o tempo inteiro e nem notamos.

Como programador Java, você deverá ter uma boa capacidade de criatividade e abstração em seus aplicativos.

Vamos mostrar alguns exemplos da utilidade do polimorfismo na abstração em Java.

Exemplos de polimorfismo e abstração em aplicações:

Jogos de Carro, Flight Simulator, plugins e linguagens de programação

Diferente de antes, hoje em dia os jogos são mais personalizáveis. Antes você somente rodava e usava o que o jogo permitia, e nada mais.

Atualmente, os jogos são feitos com elevados níveis de abstração e funcionam à base de polimorfismo.

Por exemplo, um jogo de carro. O software já vem com alguns carros e pistas feitas para você jogar, porém é comum a prática da criação de pistas, carros, manobras, cenários.

No jogo Flight Simulator, de aviões, é possível baixar na internet diversos tipos de aeronaves, helicópteros, cidades, aeroportos, pistas de pouso cor de rosa e até um kombi que voa você pode adicionar.

Aqui a gente se pergunta: como um jogo que foi lançado em uma época, dá suporte e aceita que usemos carros voadores, manobras novas de carro e outros tipos de customização? Não tem como os programadores do jogo adivinhar o que as pessoas vão criar e adicionar ao seu aplicativo. Que magia é essa, então?

Ora, o jogo é feito com abstração e polimorfismo. Ao programar esses tipos de games, os desenvolvedores já pensam que as pessoas vão adicionar novas funcionalidades, então eles focam nisso: programam de modo que o aplicativo possam receber esses novos dados.

Porém, não pode ser qualquer tipo de dado. É necessário uma padronização, que recebem algumas informações, processem e retornem outros dados para o

programa. A entrada e saída - que é a comunicação das novas funcionalidades com o soft - é o mais importante.

Se é uma kombi ou um dinossauro que vai voar, não importa. O jogo Flight Simulator não vai se importar com a figura, pode ser até a foto de seu cachorro. O importante é que seu 'avião' tenha as informações que o jogo necessita, como peso, tamanho, tipo de turbina, asa etc.

Com base nesses dados é que o game sabe se sua 'aeronave' vai subir, cair, explodir, qual velocidade é a necessária para decolar, dentre outros detalhes minuciosos.

A grosso modo, é como se o jogo tivesse diversos métodos abstratos como: 'pousar()', 'decolar()', 'pane()' e outros detalhes necessários para um vôo. Ora, como cada aeronave vai se portar, especificamente, vai depender de quem vai programar as aeronaves (que pode ser qualquer usuário), o importante é que você forneça informações ao jogo sobre o pouso, decolagem e que condições seu avião entra em pane.

O mesmo acontece com alguns jogos modernos de carros. Podemos montar qualquer um, praticamente do zero.

Para esses adicionais funcionarem, o jogo é feito de modo que, se você criar um carro e informar detalhes importantes como: quantos quilômetros fazem com 1 litro, velocidade máxima, peso, potência do motor etc, seu carro vai funcionar e se adaptar perfeitamente no game. O jogo já vem com essa 'abstração'. Você só precisa programar o seu carro, especificamente, para fornecer esses dados. Assim, sua criação se encaixará perfeitamente com a aplicação.

Outro claro exemplo são os plugins, como os de navegadores. Você mesmo pode criar uma funcionalidade para seu Firefox ou Chrome, basta saber como eles 'querem' receber as informações, saber que comandos vai alterar que parte do

programa exatamente.

Uma linguagem de programação, por exemplo, exige alto nível de abstração para ser criada, afim de suportar o polimorfismo que vamos fazer com ela. Afinal, uma linguagem de programação como o Java, tem projetos que foram usados até por robôs da NASA em Marte.

Ao criar a linguagem, se criaram certas funções, sintaxes e comandos. A definição destes especifica para que servem, como usar e o que esperar do resultado de tais funcionalidades.

Sabendo disso, você pode programar o que quiser, do tanto que escreva conforme a sintaxe e use corretamente os comandos. O interpretador de Java saberá exatamente o que executar, desde que você o oriente de forma correta.

Ou seja, se programar corretamente, o Java suportará várias 'formas' de aplicativo e os executará. Isso que é polimorfismo, não?

Portanto, vimos que a abstração e o polimorfismo são essenciais para se criar aplicações dinâmicas, que permitam o crescimento e a interação com o usuários.

Lembre-se: nenhuma aplicação é ilha. Tudo precisa de interação e expansão.

Manipulando polimorficamente subclasses

Agora que você domina completamente os conceitos de herança, polimorfismo e abstração, vamos aos exemplos práticos - códigos em Java.

Vamos mostrar como é possível uma superclasse abstrata manipule suas subclasses concretas.

Superclasse abstratas se tornando subclasses concretas

Uma poderosa arma do polimorfismo e abstração em Java é a capacidade que temos de manipular as subclasses através de uma superclasse.

Como cada subclasse concreta foi criada com métodos diferentes, elas agirão de forma diferente, conforme vimos em nossos tutoriais sobre polimorfismo em Java.

Isso nos dá um incrível poder e versatilidade, afinal podemos usar qualquer objeto que seja uma subclasse de nossa classe abstrata.

Vamos usar o exemplo de nosso tutorial sobre Polimorfismo, Classes e Métodos Abstratos: o exemplo dos animais.

Através da classe abstrata "Animal", criamos 3 objetos: uma vaca, um gato e um carneiro.

Vamos fazer a classe "Animal" receber uma referência de cada um desses objetos e invocar seu método 'som()', que é abstrato e age de forma diferente em cada subclasse.

Ou seja, vamos manipular polimorficamente os animais.

Começamos essa manipulação ao fazer com que um Array de "Animal" receba os objetos da classe "Vaca", "Gato" e "Carneiro":

```
Animal bichos[] = {mimosa, bichano, barnabe};
```

Depois, fazemos com que um objeto da classe "Animal", o 'animal' tenha uma referência para cada um desses objetos (mimosa, bichano e barnabé):

```
for(Animal animal : bichos)

{

    System.out.print("Esse animal é da classe " + animal.getClass().getName() + "
e faz ");

    animal.som();

    System.out.println();

}
```

Note como ficou simples chamar todos as subclasses da classe abstrata "Animal", fazendo com que o objeto 'animal' receba cada um dos tipos ("Vaca", "Gato" e "Carneiro") !

Imagine se fosse uma aplicação real, de um game, com dezenas ou centenas de animal, que a cada 3 segundos tivessem que fazer seu som?

Imagine a trabalhadeira que seria invocar o método de cada objeto desse?

Agora você já sabe que é melhor manipular polimorficamente cada objeto que é subclasse da classe abstrata!

Aproveitamos também para mostrar a utilidade de se saber os métodos da classe Object. No caso, usamos o método getClass(), que retorna 'classe NomeDaClasse' e o método getName() que retorna só o 'NomeDaClasse'

Código Java de como manipular polimorficamente as subclasses

-->zoo.java

public class zoo {

public static void main(String[] args) {

Vaca mimosa = new Vaca();

Gato bichano = new Gato();

Carneiro barnabe = new Carneiro();

Animal bichos[] = {mimosa, bichano, barnabe};

for(Animal animal : bichos)

{

System.out.print(animal.nome + " é da classe " + animal.getClass().getName() +
", tem " + animal.numeroPatras + " patas e faz ");

animal.som();

```
System.out.println();
```

```
}
```

```
}
```

```
}
```

-->Animal.java

```
public abstract class Animal {
```

```
protected String nome;
```

```
protected int numeroPatas;
```

```
public abstract void som();
```

```
}
```

-->Vaca.java

```
public class Vaca extends Animal {
```



```
public Vaca(){
```

```
    this.nome = "Mimosa";
```

```
    this.numeroPatas = 4;
```

```
    }
```

```
    @Override
```

```
public void som(){
```

```
    System.out.print("MUUUU");
```

```
    }
```

```
}
```

```
-->Gato.java
```

```
public class Gato extends Animal{
```

```
public Gato(){
```

```
this.nome = "Bichano";
```

```
this.numeroPatas = 4;
```

```
}
```

```
@Override
```

```
public void som(){
```

```
System.out.print("MIAU");
```

```
}
```

```
}
```

```
-->Carneiro.java
```

```
public class Carneiro extends Animal{
```

```
public Carneiro(){
```

```
this.nome = "Banabé";
```

```
this.numeroPatas = 4;
```

```
}
```

```
@Override
```

```
public void som(){
```

```
System.out.print("BÉÉÉ");
```

```
}
```

```
}
```

Descobrindo a classe de um objeto: instanceof

Nesse rápido, e útil, tutorial ensinaremos como descobrir a classe de um objeto através da keyword:

`instanceof`

Conforme sua aplicação Java for crescendo, facilmente ela atingirá dezenas ou centenas de classes, subclasses, superclasses, classes abstratas e outros nomes da Orientação a Objetos.

Somente pelo nome do objeto pode ficar difícil descobrir a qual classe ela pertence.

Porém, o Java provém uma solução simples pra isso, a keyword: `instanceof`

O nome é bem sugestivo e fácil de entender. Seu uso, mais ainda.

A sintaxe é:

`nomeObjeto instanceof Classe`

Essa é uma expressão, que retorna `true` ou `false`.

Vamos usar o exemplo de nosso tutorial passado, sobre Como Manipular

Subclasses através do Polimorfismo o exemplo dos animais.

-->**zoo.java**

public class zoo {

public static void main(String[] args) {

Vaca mimosa = new Vaca();

Gato bichano = new Gato();

Carneiro barnabe = new Carneiro();

Animal bichos[] = {mimosa, bichano, barnabe};

for(int i=0 ; i < bichos.length ; i++)

{

if(bichos[i] instanceof Vaca){

System.out.print("A vaca tem " + bichos[i].numeroPatas + " patas e faz ");

bichos[i].som();

System.out.println();

```
}
```

```
if(bichos[i] instanceof Gato){
```

```
    System.out.print("O gato tem " + bichos[i].numeroPatas + " patas e faz ");
```

```
        bichos[i].som();
```

```
    System.out.println();
```

```
}
```

```
if(bichos[i] instanceof Carneiro){
```

```
    System.out.print("O carneiro tem " + bichos[i].numeroPatas + " patas e faz ");
```

```
        bichos[i].som();
```

```
    System.out.println();
```

```
}
```

```
}
```

```
}
```

```
}
```

-->Animal.java

```
public abstract class Animal {
```

```
    protected String nome;
```

```
    protected int numeroPatas;
```

```
    public abstract void som();
```

```
}
```

-->Vaca.java

```
public class Vaca extends Animal {
```

```
    public Vaca(){
```

```
        this.nome = "Mimosa";
```

```
        this.numeroPatas = 4;
```

```
}
```

```
@Override
```

```
public void som(){
```

```
System.out.print("MUUUU");
```

```
}
```

```
}
```

```
-->Gato.java
```

```
public class Gato extends Animal{
```

```
public Gato(){
```

```
this.nome = "Bichano";
```

```
this.numeroPatas = 4;
```

```
}
```

```
@Override
```



```
public void som(){
```

```
System.out.print("MIAU");
```

```
    }
```

```
}
```

-->Carneiro.java

```
public class Carneiro extends Animal{
```

```
public Carneiro(){
```

```
this.nome = "Banabé";
```

```
this.numeroPatas = 4;
```

```
    }
```

```
@Override
```

```
public void som(){
```

```
System.out.print("BÉÉÉ");
```

```
}
```

```
}
```

Apostila de Java, capítulo 07 - Herança, reescrita (override) e polimorfismo

7.7 Exercícios: Herança e Polimorfismo

Questão 01:

Vamos criar uma classe Conta, que possua um saldo os métodos para pegar saldo, depositar e sacar.

a) Crie a classe Conta:

```
public class Conta {  
  
}
```

b) Adicione o atributo saldo

```
public class Conta {  
  
    private double saldo;  
  
}
```

c) Crie os métodos getSaldo(), deposita(double) e saca(double)

```
public class Conta {  
  
    private double saldo;  
  
    public void deposita(double valor) {  
  
        this.saldo += valor;  
  
    }  
  
    public void saca(double valor) {  
  
        this.saldo -= valor;  
  
    }  
  
}
```

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

Questão 02:

Adicione um método na classe Conta, que atualiza essa conta de acordo com uma taxa percentual fornecida.

```
class Conta {  
  
    private double saldo;  
  
    // outros métodos aqui também ...  
  
    void atualiza(double taxa) {  
  
        this.saldo += this.saldo * taxa;  
  
    }  
  
}
```

Questão 03:

Crie duas subclasses da classe Conta: ContaCorrente e ContaPoupanca. Ambas terão o método atualiza reescrito: A ContaCorrente deve atualizar-se com o dobro da taxa e a ContaPoupanca deve atualizar-se com o triplo da taxa.

Além disso, a ContaCorrente deve reescrever o método deposita, a fim de retirar uma taxa bancária de dez centavos de cada depósito.

- Crie as classes ContaCorrente e ContaPoupanca. Ambas são filhas da classe Conta:

```
public class ContaCorrente extends Conta {  
  
}
```

```
public class ContaPoupanca extends Conta {  
  
}
```

- Reescreva o método atualiza na classe ContaCorrente, seguindo o enunciado:

```
public class ContaCorrente extends Conta {  
  
    public void atualiza(double taxa) {  
  
        this.saldo += this.saldo * taxa * 2;  
  
    }  
  
}
```


Repare que, para acessar o atributo saldo herdado da classe Conta, você vai precisar trocar o modificador de visibilidade de saldo para protected.

- Reescreva o método atualiza na classe ContaPoupanca, seguindo o enunciado:

```
public class ContaPoupanca extends Conta {  
  
    public void atualiza(double taxa) {  
  
        this.saldo += this.saldo * taxa * 3;  
  
    }  
  
}
```

- Na classe ContaCorrente, reescreva o método deposita para descontar a taxa bancária de dez centavos:

```
public class ContaCorrente extends Conta {  
  
    public void atualiza(double taxa) {  
  
        this.saldo += this.saldo * taxa * 2;  
  
    }  
  
    public void deposita(double valor) {  
  
        this.saldo += valor - 0.10;  
  
    }  
  
}
```

Questão 04:

Crie uma classe com método main e instancie essas classes, atualize-as e veja o resultado. Algo como:

```
public class TestaContas {  
  
    public static void main(String[] args) {  
  
        Conta c = new Conta();  
  
        ContaCorrente cc = new ContaCorrente();  
  
        ContaPoupanca cp = new ContaPoupanca();  
  
        c.deposita(1000);  
  
        cc.deposita(1000);  
  
        cp.deposita(1000);  
  
        c.atualiza(0.01);  
  
        cc.atualiza(0.01);  
  
        cp.atualiza(0.01);  
  
        System.out.println(c.getSaldo());  
  
        System.out.println(cc.getSaldo());  
  
        System.out.println(cp.getSaldo());  
  
    }  
  
}
```

Após imprimir o saldo (`getSaldo()`) de cada uma das contas, o que acontece?

Questão 05:

O que você acha de rodar o código anterior da seguinte maneira:

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

```
Conta cc = new ContaCorrente();
```

```
Conta cp = new ContaPoupanca();
```

Compila? Roda? O que muda? Qual é a utilidade disso? Realmente, essa não é a maneira mais útil do polimorfismo - veremos o seu real poder no próximo exercício. Porém existe uma utilidade de declararmos uma variável de um tipo menos específico do que o objeto realmente é.

É extremamente importante perceber que não importa como nos referimos a um objeto, o método que será invocado é sempre o mesmo! A JVM vai descobrir em tempo de execução qual deve ser invocado, dependendo de que tipo é aquele objeto, não importando como nos referimos a ele.

Questão 06 (opcional):

Vamos criar uma classe que seja responsável por fazer a atualização de todas as contas bancárias e gerar um relatório com o saldo anterior e saldo novo de cada uma das contas.

Além disso, conforme atualiza as contas, o banco quer saber quanto do dinheiro do banco foi atualizado até o momento. Por isso, precisamos ir guardando o saldoTotal e adicionar um getter à classe.

```
public class AtualizadorDeContas {  
  
    private double saldoTotal = 0;  
  
    private double selic;  
  
    public AtualizadorDeContas(double selic) {  
  
        this.selic = selic;  
  
    }  
  
    public void roda(Conta c) {  
  
        // aqui você imprime o saldo anterior, atualiza a conta,  
  
        // e depois imprime o saldo final  
  
        // lembrando de somar o saldo final ao atributo saldoTotal  
  
    }  
  
    // outros métodos, colocar o getter para saldoTotal!
```

}

Questão 07 (opcional):

No método main, vamos criar algumas contas e rodá-las:

```
public class TestaAtualizadorDeContas {  
    public static void main(String[] args) {  
        Conta c = new Conta();  
        Conta cc = new ContaCorrente();  
        Conta cp = new ContaPoupanca();  
        c.deposita(1000);  
        cc.deposita(1000);  
        cp.deposita(1000);  
        AtualizadorDeContas adc = new AtualizadorDeContas(0.01);  
        adc.roda(c);  
        adc.roda(cc);  
        adc.roda(cp);  
        System.out.println("Saldo Total: " + adc.getSaldoTotal());  
    }  
}
```

Questão 08 (Opcional):

Use a palavra chave super nos métodos atualiza reescritos, para não ter de refazer o trabalho.

Questão 09 (Opcional):

Se você precisasse criar uma classe `ContaInvestimento`, e seu método `atualiza` fosse complicadíssimo, você precisaria alterar a classe `AtualizadorDeContas`?

Questão 10 (Opcional, Trabalhoso):

Crie uma classe Banco que possui um array de Conta. Repare que num array de Conta você pode colocar tanto ContaCorrente quanto ContaPoupanca. Crie um método public void adiciona(Conta c), um método public Conta pegaConta(int x) e outro public int pegaTotalDeContas(), muito similar a relação anterior de Empresa-Funcionario.

Faça com que seu método main crie diversas contas, insira-as no Banco e depois, com um for, percorra todas as contas do Banco para passá-las como argumento para o AtualizadorDeContas.

Solução comentada das questões do capítulo 7 da apostila de Java

Questões 01, 02, 03 e 04

A solução destas 4 primeiras questões é bem óbvia e fácil de se fazer, basta seguir o que é dito nos enunciados da apostila.

Como resultado, temos o seguinte código:

Classe Cap7ApostilaCaelum.java

```
public class Cap7ApostilaCaelum {  
  
public static void main(String[] args) {  
  
    Conta programador = new Conta();  
  
    ContaCorrente programadorCC = new ContaCorrente();  
  
    ContaPoupanca programadorCP = new ContaPoupanca();  
  
  
    //Recebendo o salário e atualizando a conta  
  
    programador.deposita(5000);  
  
    programadorCC.deposita(5000);  
  
    programadorCP.deposita(5000);  
  
}
```

```
programador.atualiza(0.01);  
programadorCC.atualiza(0.01);  
programadorCP.atualiza(0.01);
```

```
//Exibindo informações
```

```
System.out.printf("Saldo da Conta Corrente:  
%.2f\n",programadorCC.getSaldo());
```

```
System.out.printf("Saldo da Poupança: %.2f\n", programadorCP.getSaldo());
```

```
//Tirando 5 reais do café em cada conta
```

```
programador.saca(5);  
programadorCC.saca(5);  
programadorCP.saca(5);
```

```
programador.atualiza(0);  
programadorCC.atualiza(0);  
programadorCP.atualiza(0);
```

```
//Exibindo informações
```

```
System.out.println("\nDepois do cafezinho: ");
```

```
System.out.printf("Saldo da Conta Corrente:  
%.2f\n",programadorCC.getSaldo());
```

```
System.out.printf("Saldo da Poupança: %.2f\n", programadorCP.getSaldo());
```

```
}
```

```
}
```

Classe: Conta.java

```
public class Conta {
```

```
protected double saldo;
```

```
public void deposita(double valor) {
```

```
this.saldo += valor;
```

```
}
```

```
public void saca(double valor) {
```

```
    this.saldo -= valor;
```

```
    }
```

```
public double getSaldo() {
```

```
    return this.saldo;
```

```
    }
```

```
    void atualiza(double taxa) {
```

```
        this.saldo += this.saldo * taxa;
```

```
    }
```

```
}
```

Classe: ContaCorrente.java

```
public class ContaCorrente extends Conta{
```

```
    public void atualiza(double taxa) {
```

```
this.saldo += this.saldo * taxa * 2;  
  
    }
```

```
public void deposita(double valor) {
```

```
this.saldo += valor - 0.10;  
  
    }  
  
}
```

Classe: ContaPoupanca.java

```
public class ContaPoupanca extends Conta{
```

```
public void atualiza(double taxa) {
```

```
this.saldo += this.saldo * taxa * 3;  
  
    }  
  
}
```


Questão 05:

Não há o menor problema, em termos de sintaxe, compilação ou lógica de programação.

Criamos 3 tipos de contas:

Conta programador;

Conta programadorCC;

Conta programadorCP;

Em seguida vamos inicializar estes objetos e fazê-los receberem instâncias específicas de cada classe, o que ocorre naturalmente, pois ContaCorrente é Conta também, e ContaPoupança é Conta.

```
programador = new Conta();
```

```
programadorCC = new ContaCorrente();
```

```
programadorCP = new ContaPoupanca();
```

Questão 06 e 07:

Essas duas também são bem simples, pois basicamente pedem para criar uma classe, dois métodos e testar o funcionamento dessa classe. E um deles é um simples getter do atributo 'saldoTotal', que vai armazenar todo o dinheiro existente no banco, que é o somatório do dinheiro das contas.

Ele pede que o método roda() mostre o valor do saldo anterior (um print no método getSaldo() do objeto recebido), que atualize (usando o método atualiza(taxa) do objeto, onde a taxa está armazenada na variável 'selic') e seguida exiba o novo valor armazenado na conta.

Este novo valor deve ser somado à variável 'saldoTotal'.

Assim, nossa classe AtualizadorDeConta fica:

Classe: AtualizadorDeContas.java

```
public class AtualizadorDeContas {
```

```
private double saldoTotal = 0;
```

```
private double selic;
```

```
public AtualizadorDeContas(double selic) {
```

```
    this.selic = selic;
```

```
    }
```

```
public void roda(Conta c) {
```

```
    System.out.printf("Antes da atualização: %.2f\n", c.getSaldo());
```

```
        c.atualiza(this.selic);
```

```
    System.out.printf("Depois da atualização: %.2f\n", c.getSaldo());
```

```
    this.saldoTotal += c.getSaldo();
```

```
    }
```

```
public double getSaldoTotal() {
```

```
    return saldoTotal;
```

```
    }
```

```
}
```

Questão 08:

Nesta questão da apostila é pedido para usar o 'super', que se refere a uma classe mãe (superclasse).

Note que o método atualiza é:

```
this.saldo += this.saldo * taxa * valor;
```

Onde valor=2 na classe ContaCorrente e igual a 3 na classe ContaPoupanca.

Já na classe super, a Conta, não existe esse valor.

Mas podemos, mesmo assim, usar o método atualiza() da classe super nos métodos das classes filhas(subclasses), para isso basta passar para o super.atualiza() um 'valor' diferente, em vez de passar 'taxa', passar: taxa*valor

Assim, na classe ContaCorrente:

```
public void atualiza(double taxa) {
```

```
    super.atualiza(taxa*2);
```

```
}
```

E na classe ContaPoupanca:

```
public void atualiza(double taxa) {
```

```
    super.atualiza(taxa*3);
```

```
}
```

Questão 09:

Não. Uma coisa que este capítulo da apostila de Java da Caelum deixou bem claro foi a utilidade da herança para criar super classes bem gerais, e fazendo as alterações somente em locais específicos.

Nesse caso, não precisamos alterar a classe mais geral, que é a `AtualizadorDeContas`.

Como é uma situação específica, fazemos um override (reescrita) do método `atualiza()` da classe `ContaInvestimento`.

Questão 10:

Como a própria apostila da Caelum diz, essa é realmente mais trabalhosa, mas é bem interessante pois criamos um 'esboço' de um banco. Bom, vamos lá.

Primeiro criamos a classe "Banco" e ela possui 3 atributos:

Um vetor de contas, ou seja, um vetor de objetos da classe "Conta"

Um inteiro 'capacidade' que vai armazenar a capacidade do banco, ou seja, o número máximo de contas que armazena.

Um inteiro 'numContas' que contém o número atual de contas armazenadas em nossa banco.

A 'capacidade' é decidida pelo usuário no momento da criação do objeto do tipo "Banco".

Esse número é necessário para definir o tamanho do array de contas bancárias.

Portanto, é isso que fazemos no único construtor dessa classe: `contas = new Banco(capacidade);`

Depois vamos usar 3 métodos.

Um deles é bem simples, pois é nada mais que um getter, é o `pegaTotalDeContas()` que simplesmente retorna o número de contas que já foram adicionadas ao banco, ou seja, retorna o atributo inteiro 'numContas'.

A adição dessas contas ao banco é feita pelo método 'adiciona()' que recebe uma referência do objeto do tipo "Conta", que pode ter sido declarado por qualquer uma das classes de conta ("Conta", "ContaCorrente" ou "ContaPoupanca").

Ao receber o objeto, precisamos antes checar se tem espaço no banco, bastando checar se o número de contas atual (atributo numContas) é menor que a capacidade de contas (atributo capacidade).

Se for menor, colocamos na posição 'numContas' do array o objeto recebido, e incrementamos essa variável, afinal de contas agora temos uma conta a mais.

E caso esteja cheio, retornamos a referência null.

E por fim, o último método retorna qualquer conta que esteja armazenada em nossa banco.

Para isso, o usuário deve escolher um número entre 0 e (capacidade-1), que nosso aplicativo Java vai retornar a referida conta. Pronto, classe Conta concluída.

Agora vamos testá-la na main(), afinal, não basta criar, tem que mostrar que está funcionando.

Vamos aproveitar as 3 contas que temos: programador, programadorCC e programadorCP, bem como o depósito de R\$ 5000,00 que fizemos em cada uma dessas contas anteriormente, nas outras questões desse capítulo da apostila.

Em seguida criamos nosso Banco, o 'banco', com capacidade para 3 contas, e um objeto do tipo "AtualizadorDeContas", o 'adc', para manusear todos esses objetos, assim como fizemos antes nessa apostila.

Depois adicionamos essas 3 contas ao nosso Banco 'banco', através do método

'adiciona()'.

Agora vamos percorrer todas contas existentes nesse banco na 0 até a ... ?

Ué, até a `banco.pegarTotalDeContas()`, afinal este método retorna o número de contas armazenadas no banco.

Feito isso, mostramos o número de cada funcionário, e pegamos cada funcionário através do método: `banco.pegarConta()`

Pegamos o salário através do método '`getSaldo()`' de cada conta, ficando: `banco.pegarConta(i).getSaldo()`

Também, a cada iteração, temos que adicionar cada conta dessa em nosso `AtualizadorDeContas`, o '`adc`': `adc.roda(banco.pegarConta(i))`.

Concluído este laço `for`, simplesmente mostramos o saldo total, que foi atualizado pelo '`adc`'.

Veja como ficou nosso código da classe que contém a `main()` e da classe "`Banco`" (as demais ficaram como estavam):

Classe: `Cap7ApostilaCaelum.java`

```
public class Cap7ApostilaCaelum {
```

```
public static void main(String[] args) {
```

```
    Conta programador = new Conta();
```

```
Conta programadorCC = new ContaCorrente();
```

```
Conta programadorCP = new ContaPoupanca();
```

```
//Recebendo o salário
```

```
programador.deposita(5000);
```

```
programadorCC.deposita(5000);
```

```
programadorCP.deposita(5000);
```

```
//Objeto que vai atualizar as contas do Banco
```

```
AtualizadorDeContas adc = new AtualizadorDeContas(0.01);
```

```
//Questão 10
```

```
System.out.println("\nCriando um banco com 3 contas");
```

```
Banco banco = new Banco(3);
```

```
banco.adiciona(programador);
```

```
banco.adiciona(programadorCC);
```

```
banco.adiciona(programadorCP);
```

```
for(int i=0 ; i < banco.pegarTotalDeContas() ; i++){
```

```
System.out.println("\nFuncionario de numero: " + (i+1));
```

```
System.out.printf("Saldo: %.2f\n", banco.pegarConta(i).getSaldo());
```

```
        adc.roda(banco.pegarConta(i));
```

```
    }
```

```
System.out.printf("\nSaldo total: %.2f\n", adc.getSaldoTotal());
```

```
    }
```

```
}
```

Classe: Banco.java

```
public class Banco {
```

```
private Conta[] contas;
```

```
private int capacidade,
```

```
        numContas=0;
```

```
public Banco(int capacidade){
```

```
this.capacidade = capacidade;  
    contas = new Conta[this.capacidade];  
}
```

```
public void adiciona(Conta c){
```

```
    if(numContas < capacidade){  
        contas[numContas] = c;  
        numContas++;  
    }else{
```

```
        System.out.println("Número de contas no limite");  
    }
```

```
}
```

```
public Conta pegaConta(int x){
```

```
    if(x < capacidade){
```

```
return contas[x];
```

```
    }else{
```

```
        System.out.println("Essa conta não existe");
```

```
return null;
```

```
    }
```

```
}
```

```
public int pegaTotalDeContas(){
```

```
return this.numContas;
```

```
}
```

```
}
```

Jogo da Velha em Java

Após estudar Herança, Polimorfismo, Classes e Métodos Abstratos chegou a hora de colocarmos nossos conhecimentos em prática para fazer algo realmente útil e bacana:

um jogo da velha.

Rodada 6

É a vez do jogador 2

Linha: 3

Coluna: 1

X		X		O
---	--	---	--	---

X		O		
---	--	---	--	--

O				
---	--	--	--	--

Jogador 2 ganhou!

Essa é uma versão simples, para ser jogada com duas pessoas. Note que, em uma aplicação Java simples como esta, o resultado saiu por volta de 300 linhas de código.

Mas deixamos as classes e tudo pronto para que, em um próximo tutorial de Java, possamos adicionar o computador em dois modos: modo fácil e modo infalível.

Sim, existe uma estratégia que podemos adotar (ou o computador pode adotar) para nunca perder uma partida de jogo da velha:

<http://pt.wikihow.com/Ganhar-no-Jogo-da-Velha>

Como Jogar

Primeiramente, como essa versão é a mais simples, só é possível jogar Humano contra Humano. Em breve ensinaremos como programar o computador para jogar.

Então, escolha Jogador 1 como Humano e Jogador 2 também.

Após isso, basta escolher a linha e a coluna do tabuleiro onde você quer jogar.

Lembrando que o Jogador 1 é sempre o 'X' e o Jogador 2 é sempre o 'O'.

Após isto, basta jogar normalmente. Quando o jogador completar uma linha, coluna ou diagonal o jogo pára.

Ou pára quando o tabuleiro estiver completo e der empate.

Para os programadores Java:

Como criar um Jogo da Velha em Java

Como os códigos deram mais de 300 linhas de código Java, vamos explicar cada classe, método e idéia em outro post:

[Código comentado sobre como criar um Jogo da Velha em Java](#)

Código Java do Jogo da Velha, modo texto Humano x Humano

Crie um projeto com o nome 'JogoDaVelha', e adicione as 6 classes a seguir, rode e seja feliz.

-->JogoDaVelha.java

```
public class JogoDaVelha {
```

```
public static void main(String[] args) {
```

```
Jogo jogo = new Jogo();
```

$$\}$$
$$\}$$

-->Tabuleiro.java

```
public class Tabuleiro {
```

```
private int[][] tabuleiro= new int[3][3];
```

```
public Tabuleiro(){  
    zerarTabuleiro();  
}
```

```
public void zerarTabuleiro(){  
  
    for(int linha=0 ; linha<3 ; linha++)  
  
        for(int coluna=0 ; coluna<3 ; coluna++)  
            tabuleiro[linha][coluna]=0;  
  
}
```

```
public void exhibeTabuleiro(){  
  
    System.out.println();  
  
    for(int linha=0 ; linha<3 ; linha++){  
  
        for(int coluna=0 ; coluna<3 ; coluna++){
```

```
if(tabuleiro[linha][coluna]==-1){
```

```
    System.out.print(" X ");
```

```
}
```

```
if(tabuleiro[linha][coluna]==1){
```

```
    System.out.print(" O ");
```

```
}
```

```
if(tabuleiro[linha][coluna]==0){
```

```
    System.out.print(" ");
```

```
}
```

```
if(coluna==0 || coluna==1)
```

```
    System.out.print("|");
```

```
}
```

```
System.out.println();
```

```
}
```

```
}
```

```
public int getPosicao(int[] tentativa){
```

```
return tabuleiro[tentativa[0]][tentativa[1]];
```

```
}
```

```
public void setPosicao(int[] tentativa, int jogador){
```

```
if(jogador == 1)
```

```
    tabuleiro[tentativa[0]][tentativa[1]] = -1;
```

```
else
```

```
    tabuleiro[tentativa[0]][tentativa[1]] = 1;
```

```
    exhibeTabuleiro();
```

```
}
```

```
public int checaLinhas(){
```

```
for(int linha=0 ; linha<3 ; linha++){
```

```
if( (tabuleiro[linha][0] + tabuleiro[linha][1] + tabuleiro[linha][2]) == -3)
```

```
return -1;
```

```
if( (tabuleiro[linha][0] + tabuleiro[linha][1] + tabuleiro[linha][2]) == 3)
```

```
return 1;
```

```
}
```

```
return 0;
```

```
}
```

```
public int checaColunas(){
```

```
for(int coluna=0 ; coluna<3 ; coluna++){
```

```
if( (tabuleiro[0][coluna] + tabuleiro[1][coluna] + tabuleiro[2][coluna]) == -3)
```


return -1;

if((tabuleiro[0][coluna] + tabuleiro[1][coluna] + tabuleiro[2][coluna]) == 3)

return 1;

}

return 0;

}

public int checaDiagonais(){

if((tabuleiro[0][0] + tabuleiro[1][1] + tabuleiro[2][2]) == -3)

return -1;

if((tabuleiro[0][0] + tabuleiro[1][1] + tabuleiro[2][2]) == 3)

return 1;

```
if( (tabuleiro[0][2] + tabuleiro[1][1] + tabuleiro[2][0]) == -3)
```

```
return -1;
```

```
if( (tabuleiro[0][2] + tabuleiro[1][1] + tabuleiro[2][0]) == 3)
```

```
return 1;
```

```
return 0;
```

```
}
```

```
public boolean tabuleiroCompleto(){
```

```
for(int linha=0 ; linha<3 ; linha++)
```

```
for(int coluna=0 ; coluna<3 ; coluna++)
```

```
if( tabuleiro[linha][coluna]==0 )
```

```
return false;
```

```
return true;
```

```
}
```

```
}
```

-->Jogo.java

```
import java.util.Scanner;
```

```
public class Jogo {
```

```
private Tabuleiro tabuleiro;
```

```
private int rodada=1, vez=1;
```

```
private Jogador jogador1;
```

```
private Jogador jogador2;
```

```
public Scanner entrada = new Scanner(System.in);
```

```
public Jogo(){
```

```
    tabuleiro = new Tabuleiro();
```

```
    iniciarJogadores();
```

```
while( Jogar() );
```

```
    }
```

```
public void iniciarJogadores(){
```

```
    System.out.println("Quem vai ser o Jogador 1 ?");
```

```
    if(escolherJogador() == 1)
```

```
        this.jogador1 = new Humano(1);
```

```
else
```

```
    this.jogador1 = new Computador(1);
```

```
    System.out.println("-----");
```

```
System.out.println("Quem vai ser o Jogador 2 ?");
```

```
if(escolherJogador() == 1)
```

```
this.jogador2 = new Humano(2);
```

```
else
```

```
this.jogador2 = new Computador(2);
```

```
}
```

```
public int escolherJogador(){
```

```
int opcao=0;
```

```
do{
```

```
System.out.println("1. Humano");
```

```
System.out.println("2. Computador\n");
```

```
System.out.print("Opção: ");
```

```
    opcao = entrada.nextInt();
```

```
if(opcao != 1 && opcao != 2)
```

```
System.out.println("Opção inválida! Tente novamente");
```

```
    }while(opcao != 1 && opcao != 2);
```

```
return opcao;
```

```
    }
```

```
public boolean Jogar(){
```

```
if(ganhou() == 0 ){
```

```
System.out.println("-----");
```

```
System.out.println("\nRodada "+rodada);
```

```
System.out.println("É a vez do jogador " + vez() );
```

```
if(vez()==1)
```

```
    jogador1.jogar(tabuleiro);
```

```
else
```

```
    jogador2.jogar(tabuleiro);
```

```
if(tabuleiro.tabuleiroCompleto()){
```

```
    System.out.println("Tabuleiro Completo. Jogo empatado");
```

```
return false;
```

```
    }
```

```
    vez++;
```

```
    rodada++;
```

```
return true;
```

```
    } else{
```

```
if(ganhou() == -1 )
```

```
    System.out.println("Jogador 1 ganhou!");
```

else

System.out.println("Jogador 2 ganhou!");

return false;

}

}

public int vez(){

if(vez%2 == 1)

return 1;

else

return 2;

}


```
public int ganhou(){
```

```
    if(tabuleiro.checaLinhas() == 1)
```

```
        return 1;
```

```
    if(tabuleiro.checaColunas() == 1)
```

```
        return 1;
```

```
    if(tabuleiro.checaDiagonais() == 1)
```

```
        return 1;
```

```
    if(tabuleiro.checaLinhas() == -1)
```

```
        return -1;
```

```
    if(tabuleiro.checaColunas() == -1)
```

```
        return -1;
```

```
if(tabuleiro.checaDiagonais() == -1)
```

```
return -1;
```

```
return 0;
```

```
}
```

```
}
```

-->Jogador.java

```
public abstract class Jogador {
```

```
protected int[] tentativa = new int[2];
```

```
protected int jogador;
```

```
public Jogador(int jogador){
```

```
this.jogador = jogador;
```

```
}
```

```
public abstract void jogar(Tabuleiro tabuleiro);
```

```
public abstract void Tentativa(Tabuleiro tabuleiro);
```

```
public boolean checaTentativa(int[] tentativa, Tabuleiro tabuleiro){
```

```
if(tabuleiro.getPosicao(tentativa) == 0)
```

```
return true;
```

```
else
```

```
return false;
```

```
}
```

```
}
```

-->Humano.java

```
import java.util.Scanner;
```

```
public class Humano extends Jogador{
```

```
public Scanner entrada = new Scanner(System.in);
```

```
public Humano(int jogador){
```

```
    super(jogador);
```

```
    this.jogador = jogador;
```

```
    System.out.println("Jogador 'Humano' criado!");
```

```
    }
```

```
    @Override
```

```
public void jogar(Tabuleiro tabuleiro){
```

```
    Tentativa(tabuleiro);
```

```
    tabuleiro.setPosicao(tentativa, jogador);
```

```
    }
```

@Override

public void Tentativa(Tabuleiro tabuleiro){

do{

do{

System.out.print("Linha: ");

tentativa[0] = entrada.nextInt();

if(tentativa[0] > 3 ||tentativa[0] < 1)

System.out.println("Linha inválida. É 1, 2 ou 3");

}while(tentativa[0] > 3 ||tentativa[0] < 1);

do{

System.out.print("Coluna: ");

```
tentativa[1] = entrada.nextInt();
```

```
if(tentativa[1] > 3 || tentativa[1] < 1)
```

```
System.out.println("Coluna inválida. É 1, 2 ou 3");
```

```
    }while(tentativa[1] > 3 || tentativa[1] < 1);
```

```
    tentativa[0]--;
```

```
    tentativa[1]--;
```

```
if(!checaTentativa(tentativa, tabuleiro))
```

```
System.out.println("Esse local já foi marcado. Tente outro.");
```

```
    }while( !checaTentativa(tentativa, tabuleiro) );
```

```
    }
```

```
}
```

-->Computador.java

```
public class Computador extends Jogador{
```

```
public Computador(int jogador){
```

```
    super(jogador);
```

```
    System.out.println("Jogador 'Computador' criado!");
```

```
}
```

```
    @Override
```

```
public void jogar(Tabuleiro tabuleiro){
```

```
}
```

```
    @Override
```

```
public void Tentativa(Tabuleiro tabuleiro){
```

```
}
```

```
}
```

Código comentado sobre como criar um Jogo da Velha em Java

No tutorial passado mostramos um print do Jogo da Velha em Java, modo texto e Humano x Humano, bem como o código de 6 classes necessárias para rodar o programa:

[Jogo da Velha em Java](#)

Agora vamos comentar, em detalhes, cada classe, método e idéia que usamos para criar o jogo.

Classe JogoDaVelha.java

Bom, essa é a classe que possui o método main. Ou seja, a partir daí a aplicação se inicia.

Note como essa classe e o método está bem enxuto.

É uma boa prática fazer isso: só use a main() pra iniciar sua aplicação, tente ao máximo não encher ela de variáveis, métodos e outras coisas.

Bom, nessa classe apenas iniciamos o jogo, criando um objeto 'jogo' da Classe "Jogo".

Para saber mais, vá para a classe "Jogo" !

Classe Tabuleiro.java

Essa é uma classe bem simples e útil, que ficará encarregada de criar o tabuleiro, desenhá-lo na tela bem como fazer as checagens, para saber se uma linha, coluna ou diagonal foi preenchida por um dos jogadores.

A classe inicia criando uma Matriz 3x3 de inteiros, a matriz 'tabuleiro' que é nosso tabuleiro.

A seguir, o construtor padrão invoca o método "zeraTabuleiro()", que como o nome já diz, faz todos os inteiros da matriz 'tabuleiro' receberem o valor 0.

O código seguinte será invocado várias vezes durante o jogo, é o método que exibe a matriz na tela, o 'exibeTabuleiro()'.

Ele vai percorrer todos os elementos da tela, caso encontre -1, ele imprime um 'X', caso encontre 1 ele imprime o 'O', e caso encontre o número 0 ele não imprime nada (campo vazio no tabuleiro).

Caso nenhuma fila esteja completa, ou seja, caso ninguém tenha ganhado, o método ganhou() retorna 0 e podemos continuar no nosso método "Jogar()".

Os métodos seguintes, getPosicao() e setPosicao() são auto-explicativos. O getter retorna que número (-1, 0, 1) está em determinada posição, e o setter altera um local do tabuleiro para -1 ou 1, conforme o jogador seja X ou O. As posições são sempre representadas por um vetor chamado 'tentativa[2]', onde 'tentativa[0]' representa a linha e 'tentativa[1]' representa a coluna.

Os próximos métodos fazem a checagem de todas as linhas (`checaLinhas()`), colunas (`checaColunas()`) e das duas diagonais (`checaDiagonais()`).

Se alguma linha, coluna ou diagonal estiver cheia e marcada com 'X', então a soma dos elementos dessa fila será -3 ($-1 + -1 + -1 = -3$), e o método retorna -1, sinalizando que o jogador 1 ganhou.

Se alguma linha, coluna ou diagonal estiver cheia e marcada com 'O', então a soma dos elementos dessa fila será 3 ($1 + 1 + 1 = 3$), e o método retorna 1, sinalizando que o jogador 2 ganhou.

O último método, o '`tabuleiroCompleto()`', checa todos os elementos da matriz, caso encontre um número 0, retorna false. Isso indica que a matriz não está totalmente preenchida.

Caso não encontre um número 0 sequer, é porque todos são 1 ou -1. Ou seja, o tabuleiro está completo, retornamos true e o jogo deu empatado.

Classe Jogo.java

Essa classe vai criar, através de variáveis:

um tabuleiro:

```
private Tabuleiro tabuleiro;
```

Números que irão controlar o número da rodada e de quem é a vez de jogar:

```
private int rodada=1, vez=1;
```

Dois jogadores:

```
private Jogador jogador1;
```

```
private Jogador jogador2;
```

E um objeto da classe Scanner para receber dados dos jogadores Humanos.

Sempre que criamos um objeto de uma classe, seu construtor padrão roda. No caso, é o "Jogo()".

Ele vai iniciar o tabuleiro:

```
tabuleiro = new Tabuleiro();
```

(Vá para a classe Tabuleiro e leia como ela funcione. Depois volte)

Em seguida, vai iniciar os jogadores, pra saber quem vai ser o primeiro, segundo e quem vai ser humano ou computador (escolha, por hora, humanos).

A seguir, vamos rodar o jogo a partir do comando:

```
while( Jogar() );
```

O método "Jogar()" é que vai controlar todo o jogo. Ele retorna um booleano.

Se o jogo ainda não tiver terminado, ele retorna 'true', então o laço while irá sempre rodar.

Quando o jogo terminar, alguém ganhando ou dando empate, o método "Jogar()" retornará 'false', o while não irá mais continuar e o construtor termina, terminando a criação do objeto e, conseqüentemente, o jogo.

Lembre-se: nossa main() apenas cria um objeto. Como o objeto foi criado, a aplicação terminou.

Classe Jogador.java

Nessa classe, colocamos em prática nossos conhecimentos de abstração e polimorfismo.

Há dois tipos de jogadores, o humano e o computador.

Portanto, seus métodos de jogar são diferentes. Por isso ela será abstrata, já que terá os métodos abstratos 'jogar()' e 'Tentativa()', já que a estratégia é diferente para cada tipo de jogador, e o computador não fará tentativas que não possa.

Nessa classe definimos o vetor de inteiros 'tentativa[2]', onde tentativa[0] armazenará a linha e tentativa[1] a posição da coluna de onde o jogador tentará marcar seu X ou O.

O método 'jogar()' recebe o atual tabuleiro, assim como o método 'Tentativa()' que também recebe as posições (linha, coluna) que o jogador vai tentar marcar seu X ou O.

O método 'checaTentativa()' vai se certificar que na posição que o Jogador irá marcar no tabuleiro está realmente vazio, ou seja, se aquela posição tem realmente o número 0.

Em breve, vamos criar o jogador Computador nível Easy, que vai jogar simplesmente marcando o tabuleiro aleatoriamente, por isso iremos usar o método 'checaTentativa()' para o computador também (por isso não é abstrato, vai servir tanto pra Humano como pra Computador).

Classe Humano.java

Ela herda a classe abstrata "Jogador".

Cada objeto criado recebe um número, 1 ou 2, que vai caracterizar como jogador 1 ou jogador 2, para as pessoas saberem de quem é a vez de jogar.

A seguir, vamos fazer o Override do método 'jogar()', personalizado para jogadores humanos.

Primeiro, chamamos o método 'Tentativa()', que vai pedir a linha e a coluna, vai checar se esses números são válidos (são 1, 2 ou 3) e vai fazer a alteração da matriz que vemos, pra matriz real do jogo.

Pois como jogador, vemos as linhas e colunas 1, 2 e 3. Porém, ao criamos a matriz tabuleiro[3][3], trabalhamos com linhas e colunas de números 0, 1 e 2.

Assim, se o jogador digitar linha 2 e coluna 3, subtraímos uma unidade de cada e mandamos 'linha 1 e coluna 2' pro Java.

Quando formos implementar o jogo do computador, isso não será necessário, usaremos direto as linhas e colunas 0, 1 e 2.

Essas alterações são apenas para lidas com os humanos.

Então, quando o jogador humano colocar uma dado correto, marcamos o 'X' ou o 'O' no tabuleiro através do método 'setPosicao()', do objeto 'tabuleiro' da classe "Tabuleiro".

Essas informações (tabuleiro e tentativas[2]) são necessárias pro método 'checaTentativa()' saber se na posição desejada já não existe algum X ou O, além de se certificar que o usuário humano não vai colocar um número maior que 3 ou menor que 1.

Classe Computador.java

Ela herda a classe abstrata "Jogador", e nesse tutorial ela não faz nada.

Iremos implementar ela em um próximo tutorial, onde você irá aprender sobre inteligência artificial e poderá, finalmente, brincar com o computador através de uma aplicação que você mesmo fez.

Strings e Caracteres: Escrevendo em Java

Java: A Classe String

Strings é uma sequência de caracteres. Estes podem ser letras, dígitos ou caracteres especiais, como + _ - * > .<

São tão importantes que o Java possui mais de uma classe só para tratar as strings.

Vejamos, por exemplo, construtores para a classe String:

```
char[] blog = {'p', 'r', 'o', 'g', ' ', 'p', 'r', 'o', 'g', 'r', 'e', 's', 's', 'i', 'v', 'a'};
```

```
String stg = new String( "teste" );
```

```
String stg1 = new String();
```

```
String stg2 = new String( stg );
```

```
String stg3 = new String( blog );
```

```
String stg4 = new String( blog, 5, 11 );
```

A saída seria:

stg → teste

stg1 →

stg2 → teste

stg3 → prog progressiva

stg4 → progressiva

Essa classe possui uma série de métodos para formatar strings. No último caso, 'stg4' é iniciada com parte da string 'blog', ela receberá 11 caracteres dessa string, a partir do caractere de índice 4.

Aqui vai uma importante ressalva: String objects são imutáveis! Não é possível mudar seus caracteres depois que você criou!

As strings, por serem da classe String, possuem métodos característicos:

stg.length() -> informa o tamanho da string

stg.charAt(indice) -> mostra determinado caractere da string

stg.getChars(int inicio, int fim, char[] , int inicio2)

Diferente de C/C++, por exemplo, String e Vetor de caracteres são duas coisas diferentes em Java.

No último método, queremos passar os caracteres da String 'stg', a partir do índice 'inicio' até o índice 'fim' para um vetor de caracteres, a partir da posição de índice 'inicio2' nesse vetor

stg.compareTo(stg2) -> 0 se forem iguais; negativo caso stg < stg2 ; positivo caso stg > stg2 , lexicograficamente falando (na forma do dicionário).

A classe String também oferece ferramentas para compararmos strings. Veja alguns desses métodos:

`stg.equals(stg2)` -> retorna 'true' ou 'false', com case sensitive (maiúsculo/minúsculo importam)

`stg.equalsIgnoreCase` -> retorna 'true' ou 'false', sem case sensitive (maiúsculo/minúsculo não importam)

`stg == "teste"` -> retorna 'true' ou 'false' em relação ao fato de serem o mesmo OBJETO, não conteúdo!!! Só retorna 'true' se apontarem pro mesmo endereço de memória!

Se tivéssemos feito:

`stg="teste"` , o método retornaria 'true', pois o Java trata todos os objetos de string literal de mesmo conteúdo como o mesmo objeto, só com mais de uma referência. Por exemplo:

```
string1="Programacao Progressiva"
```

```
string2="Programacao Progressiva"
```

Como ambas possuem o mesmo conteúdo, Java trata elas como o mesmo objeto. Ambas apontam pro mesmo endereço de memória.

Porém, com `stg = new String("hello")` , se cria um novo endereço de memória para a string. O que resultaria em 'false'.

`stg.regionMatches(int a, String stg2, int b, int c)` -> compara somente certas regiões das strings. Começa a partir do índice 'a' da 'stg', compara com a string

'stg2' a partir do índice 'b' desta, e elas comparam 'c' elementos das strings.

Também existem métodos para a busca, concatenação, alteração maiúsculo/minúsculo, substituição total/parcial e extração de trechos de texto ou caractere em strings, presentes na classe String.

Porém, devido ao grande número e flexibilidade dos métodos oferecidos pela classe String, é inviável mostrar todos aqui, deixamos alguns aqui e outros você pode ver na documentação, conforme sua necessidade. Exemplo:

`stg.startsWith()`

`stg.endsWith()`

`stg.lastIndexOf()`

`stg.indexOf()`

`stg.substring()`

`stg.concat()`

`stg.replace()`

`stg.toUpperCase()`

`stg.toLowerCase()`

`stg.toCharArray()`

Também existem métodos para 'quebrar' uma determinada string, cujas partes são separadas por um separador, como o espaço " ", por exemplo.

Escreva algo:

```
String stg= scanner.nextLine();
```

Receba as 'partes', separadas por espaço:

```
String[] pedacos = stg.split(" ");
```

pedacos.length -> retorna o numero de pedaços

Vamos imprimir esses pedaços, que nada mais são que as palavras que voce digitou:

```
for(String palavra : pedacos)
```

```
System.out.println(palavra);
```

Java: A Classe StringBuilder

Essa sim é modificável, diferente da classe String.

Com strings dessa classe podemos usar as strings de forma dinâmica, criando, modificando, copiando, anexando, invertendo, adicionando e retirando caracteres, o tanto que desejarmos.

Porém, devemos ainda usar a classe String, especialmente quando os dados contidos na string não necessitarem de mudança (como em Labels, mensagens de erro, de log etc, pois por conta dessa limitação, essa classe é bem otimizada. Mas para strings que se alteram, usamos a StringBuilder.

Existe ainda a StringBuffer, para caso seja utilizado threads.

Vejamos alguns métodos e características destes:

```
StringBuilder teste = new StringBuilder("123456789");
```

teste.toString() -> retorna a string 123456789

teste.length() -> o tamanho é 9

teste.capacity() -> a capacidade inicial é 25 (9 + 16, toda StringBuilder inicia com 16, por padrão)

teste.ensureCapacity(100) -> garante uma capacidade pra string, para não ser necessário crescer dinamicamente ao longo de um programa, o que pode comprometer a performance.

teste.setLength(5) -> agora, teste é somente "12345"

teste.reverse() -> inverte os caracteres da StringBuilder

teste.charAt(int) -> recebe um inteiro e retorna o caractere referente aquele índice

teste.getChars(a, b, char[] c, d) -> copia caracteres da StringBuilder 'teste' a partir do índice 'a' até o índice 'b' para o vetor de caracteres 'c', a partir do índice 'd' neste vetor.

Teste este método, passando a string para o vetor de caracteres e imprimindo-o:

```
teste.getChars(0, teste.length(), vetorChar, 0);
```

```
for( char caractere : vetorChar )
```

```
System.out.print(caractere);
```

teste.setCharAt(a, 'b') -> insere o caractere 'b' na posição de índice 'a'

teste.insert(a, objeto) -> insere, a partir do índice 'a', o 'objeto' na BuildString. Objeto, em Java, representa uma gama enorme de coisas, como String, vetor de caracteres, booleanos, inteiros, float, doubles, chars etc.

teste.deleteCharAt(a) -> deleta especificamente o caractere da posição de índice 'a'

teste.delete(a, b) -> dele o caractere de índice 'a' até o de índice 'b'

teste.append(variosObjetos) -> anexa, ao fim da String Builder 'teste' algum 'variosObjetos';

É possível, inclusive, a concatenação:

```
String teste2 = new  
StringBuilder().append("Programação").append("\n").append("Progressiva").toSt
```

Com esses métodos, é possível notar a flexibilidade e utilidade da `StringBuilder`.

Java: A Classe Character

Assim como outros tipos, a Character é uma classe do tipo wrapper. Ou seja, não é apenas um tipo e pronto, como int e float.

Ela já vem com uns métodos e funcionalidades que facilitarão bastante a vida do programador Java.

O Java provém, automaticamente, para os caracteres, diversos métodos, como checagem pra saber se o caractere é Unicode, se é letra, se é maiúscula ou minúscula, se é número etc, além de métodos de comparação entre caracteres. Vamos ver:

char caractere;

Character.isDefined(caractere) -> checa se 'caractere' está definido no Unicode; retorna true ou false

Character.isDigit(caractere) -> 'true' se 'caractere' é um dígito, 'false' caso contrário

Character.isLetter(caractere) -> 'true' se 'caractere' for letra, 'false' o contrário

Character.isLowerCase(caractere) -> 'true' se 'caractere' for minúsculo, 'false' o contrário

Character.isUpperCase(caractere) -> 'true' se 'caractere' for maiúsculo, 'false' o contrário

`Character.toUpperCase(caractere)` -> transforma o 'caractere' em maiúsculo

`Character.toLowerCase(caractere)` -> transforma o 'caractere' em minúsculo

`Character.isLetterOrDigit(caractere)` -> 'true' se 'caractere' for letra ou dígito, 'false' o contrário

`Character.isJavaIdentifierStart(caractere)` -> esse método é interessante. Ele identifica se o 'caractere' pode ser o primeiro caractere de um identificador em Java, ou seja, se pode ser letra, underlina `_` ou o sinal de dinheiro `$`.

`Character.isJavaIdentifierPart(caractere)` -> já esse checa se o 'caractere' pode ser parte de um identificador, ou seja, se é letra, número, underline `_` ou dindin `$`; ambos retornam 'true' ou 'false'

Para usarmos os métodos comparativos, vamos declarar os caracteres como objetos 'Character' ao invés do tipo 'char' para explicar melhor alguns métodos da CLasse, pois já mostramos os da wrapper:

```
Character a = 'a';
```

```
Character b = 'b';
```

`a.charValue()` -> retorna o valor do objeto em formta de caractere

`a.toString()` -> retorna o objeto, mas na forma de String

`a.equals(b)` -> compara se o objeto 'a' possui o mesmo conteúdo do objeto 'b', onde o caso sensetivo importa. Ou seja, 'p' é diferente de 'P'

Embora muitas pessoas ainda confundam, mesmo tendo ciência de métodos que envolvam números, caracteres também incluem dígitos!

Portanto, os métodos em Java também provém meios para a conversão entre bases decimais, também conhecidas como 'radix'.

Ambos métodos a seguir são usados principalmente no tratamento de números hexadecimais, ou seja, para radix=16.

`Character.forDigit(numero, base)` -> converte o 'numero' para o character correspondente, na base 'base'

`Character.digit(caractere, base)` -> converte o 'caractere' para número, na base 'base'

Java: Expressões Regulares (regex) em Java

Também conhecidas como regex (regular expressions), expressões regulares são sentenças com códigos que podem representar uma string, de forma genérica.

Por exemplo, o CEP é da forma: ddddd-ddd

Onde 'd' é um dígito qualquer, entre 0 e 9. Essa expressão representa toda a gama de CEP's em nosso país.

São usadas também para a criação de compiladores, através de expressões regulares bem complexas, que podem representar qualquer trecho de código, e acusar erro de sintaxe, caso o código escrito não case com as expressões regulares.

Em nossos aplicativos, usaremos expressões regulares para garantir que a entrada de dados esteja sob uma forma específica.

Por exemplo, caso necessitemos que o usuário digite um CEP e ele escrever:

dddd*-ddd

O que ele digitou não vai casar com a expressão regular que usamos para validar o CEP, pois o número de CEP não aceita asterisco.

As expressões regulares são uma ferramenta bem antiga, usada desde a época do Unix, e é usada nas mais diversas linguagens de programação, como em Perl, e em programas, como o sed.

Vamos a alguns exemplos:

`\d` -> representa qualquer dígitos

`\D` -> representa tudo, menos dígito

`\w` -> representa qualquer caractere de palavra

`\W` -> representa tudo que não seja um caractere de palavra

`\s` -> representa qualquer espaço em branco(" ", tab)

`\S` -> representa tudo que não seja espaço em branco

`[a-z]` -> representa qualquer letra minúscula do alfabeto

`[A-Z]` -> representa qualquer letra maiúscula do alfabeto

`[a-zA-Z]` -> representa qualquer letra do alfabeto, seja maiúscula ou minúscula

`|` -> representa o 'ou'. `"a|b"` casa com 'a' ou com 'b', ou com os dois

Vejamos alguns quantificadores

`.` -> substitui qualquer caractere

`*` -> o caractere anterior aparece nenhuma ou mais vezes

`+` -> o caractere anterior aparece uma ou mais vezes

`{a}` -> o caractere anterior se repete 'a' vezes

`{a,}` -> o caractere anterior se repete pelo menos 'a' vezes

`{a,b}` -> o caractere anterior se repete entre 'a' e 'b' vezes

A representação de dígito é `\d`, mas dentro de strings, para `\` valer, temos que escapá-la.

Logo, podemos representar um CEP como: `"\\d\\d\\d\\d\\d-\\d\\d\\d"`

Ou `"\\d{5}-\\d{3}"`

O método ficaria:

`meu_cep.matches("\\d{5}-\\d{3}")` -> retorna 'true' se o CEP for digitado correto, e 'false' caso errado

Para validar um nome, sabemos que o primeiro caractere é maiúsculo:

`nome.matches("[A-Z][a-zA-Z]*")` -> retorna 'true' se a primeira letra do nome for maiúscula, e 'false' caso errado

O Java provém alguns métodos para o uso de regex e substituições:

`std = std.replaceAll("a", "b")` -> substitui todas as ocorrências de 'a' por 'b'

Também podemos usar expressões regulares no método 'replaceAll':

`std = std.replaceAll("\\w+", "a")` -> substitui todas as palavras por 'a'

`std = std.replaceFirst("\\d","a")` -> substitui a primeira ocorrência de um dígito por 'a'

`String[] pedacos = std.split("a")` -> divide a string 'std' em partes, cujo separador é 'a' e armazena no vetor de strings 'pedacos'. Por exemplo, para separarmos as palavras de uma string, usamos `","` ou `",\\s*"` como separador

É importante frisar que, nos vários métodos e classes para o tratamento de regex, use a classe `String`.

Se usar a `StringBuilder` terá erros.

Caso queira prosseguir no mundo das regex e em Java, esse package possui duas classes que lhe serão bastante úteis: a `Pattern`, para usar uma expressão regular e a `Matcher`, que também usa uma expressão regular mas também uma `CharSequence`, onde você irá procurar o padrão para casar.

Para expressões regulares que serão usadas somente uma vez, se recomenda usar o método `static 'matches'` da `Pattern`. Porém, se for usada mais de uma vez, se aconselha a usar o método `static 'compile'`, que retorna um objeto do tipo `Pattern`, onde posteriormente se pode chamar o método `'matcher'`.

A classe `Matcher` também provém o método `'matches'` e faz a mesma coisa, porém não recebe argumentos. Ela age por encapsulamento no objeto do tipo `Matcher`.

Essa classe também possui outros métodos bastante úteis, como o `'replaceFirst'`, `'replaceAll'`, `'find'`, `'lookingAt'` etc, que, pelo nome, é possível saber o que cada um faz.

Tenha em mente que as possibilidades das expressões regulares são inúmeras e ilimitadas. Compiladores são feitos usando expressões regulares! Isso por si só já

mostra o poder das regex.

Por isso, não entrarei tão afundo no assunto.

Tendo estudado esse artigo, você saberá as possibilidades das Expressões Regulares, e irá se lembrar delas caso um dia necessite delas.

Expressões Regulares são universais. Você pode aprender e usar em vários ramos e linguagens.

Eu mesmo já falei aqui em outros artigos, como no de sed. Lá está até mais completo, se eu falasse mais aqui, estaria simplesmente repetindo o que está escrito lá.

Portanto, dê um pulinho no artigo de Regex da parte de 'sed', é uma continuação daqui:

<http://eeeprogramacaoprogessiva.net/2012/07/sed-parte-vi-expressoes-regulares.html>

Regex é assunto pra livros! E é isso que vou indicar, um site de um brasileiro, o Aurelio Jargas, mestre em Expressões Regulares que publica e divulga GRATUITAMENTE seu material!

<http://aurelio.net/regex/>

Vale ressaltar que o java possui um pacote (package) voltado para expressões regulares.

Vale uma conferida na documentação:

java.util.regex

Programação Gráfica em Java, pt II: desenhos, fontes e figuras geométricas em 2D

JFrame e JPanel - Introdução ao estudo de GUI

Antes de iniciarmos nossos estudos sobre aplicações gráficas em Java, precisamos entender dois importantíssimos e básicos conceitos sobre GUI (Graphic User Interface): JFrame e JPanel.

Não iremos focar na criação códigos ou algo gráfico, mas é essencial ler e entender o que é explicado aqui para prosseguir nessa seção de GUI de nossa apostila de Java.

Iremos dar uma base aqui sobre o assunto, e em seguida iremos, no próximo tutorial falaremos sobre o uso de JFrame e JPanel para fazer desenhos.

O que é JFrame

Frame, em inglês, pode significar várias coisas.

Mas a melhor definição para frame em nosso contexto é: moldura.

Vamos imaginar uma moldura, onde você vai colocar uma foto.

Esse frame é basicamente a base de tudo, para criar aplicações gráficas em Java.

Sabe aquela janela de um programa? É um frame.

Sabe as caixa de diálogo que usamos? É um frame.

As janelas onde ocorrer as aplicações gráficas que são o frame.

E o que faz o frame?

Ele vai armazenar, exibir coisas. Coisas que podem ser imagens, textos, botões, opções para marcar, caixa de texto para escrever e tudo mais o que é possível fazer em aplicativos GUI.

Veja bem, ele exibe, é responsável por controlar e dizer como essas coisas irão aparecer nele. Essas coisas não fazem parte dele.

Por exemplo, uma foto de seu cachorro é desenhada e impressa na sua moldura de madeira? Não, ela é colocada, encaixada lá.

Você pode tirar a foto e colocar outra, a moldura é só um suporte pra ajudar na exibição. Ela em si não tem a informação e detalhes das imagens.

Da mesma maneira é com o JFrame, ele vai servir para exibir e trabalhar com esses elementos (botões, textos, imagens etc).

□

O que é possível fazer com JFrame

Para saber para que serve e o que é possível fazer com o JFrame, basta nos lembrarmos o que é possível fazer com as janelas, ou moldes, das aplicações que conhecemos no dia-a-dia.

Podemos minimizar, maximizar ou fechar.

Quando clicamos em fechar, as vezes ela encerra a aplicação, mas as vezes ela continua rodando no sistema, mas sem ficar visível.

Um JFrame, inclusive, pode chamar outro. E isso é bem normal.

Afinal, são poucos os casos de aplicações 'isoladas'.

Sempre abre uma janela, clicamos em um botão ou menu, irá se abrir outra janela, com outras informações.

Já deve ter notado aplicativos que abrem minimizados, outros no centro da tela, outros em um canto.

Isso também é definido no JFrame.

As janelas também possuem uma cor de fundo, menus, barras de rolagem.

Também há aquele ícone, no canto superior esquerdo, de cada GUI.

Há também programas que é possível mudar o tamanho da janela, outros não.

Tem uns mais chatos que nem nos permitem minimizar.

Notou a importância do JFrame?

Mas falamos que ele serve como moldura, e as fotos?

E o conteúdo dentro desse molde, como colocamos?

Onde criamos?

O que é JPanel

Falamos dos frames, e associamos com molduras.

Então, não é muito difícil adivinhar que o panel é como se fosse a foto, o conteúdo das molduras.

A tradução de panel é bem óbvia: painel.

E é exatamente isso que é (como sempre) um JPanel, um painel.

Este painel vai ser encaixado no molde, no JFrame, e é ele que vai contar os componentes gráficos de nossos programas.

Esses componentes, em Java, são os JComponents, e são muitos!

Podem ser botões, caixas de seleção para você marcar, um bloco de texto para ler (iguais aqueles de "Aceito os termos e condições"), campos de texto para você escrever (para inserir login e senha, por exemplo), imagens e simplesmente tudo que é possível fazer em programação gráfica.

E assim como podemos colocar uma foto em uma moldura, tirar e colocar outra, também podemos colocar um JPanel em um frame, fazer a interação com o usuário, depois colocar outro JPanel lá, ou mandar esse JPanel para outro JFrame, e por aí vai, dependendo do objetivo de sua aplicação.

Agora que você tem uma noção do que é, para que serve a importância do JPanel e do JFrame, vamos, no próximo tutorial de nosso curso de Java, botar a mão na

massa: colocar os códigos e ver as coisas funcionando de verdade.

JFrame e JPanel: como criar uma aplicação gráfica em Java

No artigo passado, de Introdução ao estudo de GUI, demos uma explicação teórica e bem simples de entender sobre JFrame e JPanel. Agora vamos, de fato, usar JFrame e JPanel.

A partir deste ponto de nossa apostila de Java, iremos criar nossa própria 'janela', ou seja, não vamos usar as janelas já feitas, como nas caixas de diálogo.

JFrame: Como criar Janelas em Java

JFrame nada mais é que uma classe, como outra qualquer que estudamos e criamos ao longo de nosso curso.

Porém, essa classe é que será responsável por criar a tela em que iremos desenhar, colocar botões, menus, caixas de texto e tudo mais que existem nas janelas de aplicativos.

Para fazer uso do JFrame, temos que importar essa classe do pacote swing, que contém diversas funcionalidades para programação gráfica:

```
import javax.swing.JFrame;
```

Cada objeto que criamos é um frame, ou janela, diferente. Uma aplicação normal tem dezenas ou centenas de frames.

Nessa seção vamos criar alguns objetos da classe JFrame passando uma String argumento para o construtor padrão, essa string será o título de nossa aplicação.

Esses JFrames, porém, possuem dezenas de funcionalidades e opções, usaremos as funcionalidades: o que ocorre quando clicamos no 'x' de close, adicionar panels, definir o tamanho do frame e se ele será visível ou não.

Como dissemos, o JFrame é uma Classe. E para criar uma classe, criamos um objeto.

Assim, nossa janela será um objeto chamado "janela".

Vamos passar uma string para o construtor dessa classe, que servirá como título de nosso frame/janela.

Porém, para vermos nosso frame, precisamos definir uma funcionalidade: que ele seja visível!

Pode parecer óbvio, mas um programa pode ter muitos, mas muitos frames, e obviamente não podemos exibir todos, senão nossa tela ficaria uma bagunça.

Assim, para exibir esse frame, usamos o método setVisible, que recebe true ou false.

Portanto, um simples programa em Java que cria e exibe um frame será:

```
import javax.swing.JFrame;
```

```
public class framesPanels {
```

```
public static void main(String[] args) {
```

```
    JFrame janela = new JFrame("Meu primeiro frame em Java");
```

```
    janela.setVisible(true);
```

```
}
```

```
}
```

Note que vai ser criado uma janela, mas provavelmente você nem vai notar. Isso porque não fornecemos o tamanho da janela, então ela foi criada com 0 pixel de largura e 0 de altura.

Vamos usar o método `setVisible`, que recebe um tipo boolean. Como queremos a janela visível, colocamos `true`.

Caso queira definir um tamanho pré-definido, use o método `setSize()`, que recebe dois parâmetros com os pixels da janela (horizontal e vertical):

```
import javax.swing.JFrame;
```

```
public class framesPanels {
```

```
public static void main(String[] args) {
```

```
    JFrame janela = new JFrame("Meu primeiro frame em Java");
```

```
    janela.setSize(300,200);
```

```
    janela.setVisible(true);
```

```
    }
```

```
}
```

Para saber mais funcionalidades da classe `JFrame`, veja a documentação em:

<http://docs.oracle.com/javase/6/docs/api/javax/swing/JFrame.html>

JPanel: Inserido elementos em um JFrame

Como vimos, nosso frame está vazio. Só definimos o tamanho da 'moldura' e o título.

Vamos adicionar alguns elementos ao nosso frame, e faremos isso inserindo um JPanel, onde nele podemos colocar uma infinidade de elementos, chamados Jcomponents:

http://www.apl.jhu.edu/~hall/java/Swing-Tutorial/Swing-Tutorial-JPanel.html

Para usar o JPanel, importamos essa classe da package swing:

```
import javax.swing.JPanel;
```

Vamos criar uma classe que será nosso painel, vamos chamar de "Painel".

Para isso, basta fazer com que ela extends a JPanel, como aprendemos em Herança, para que herdemos todas as propriedades do JPanel.

```
import javax.swing.JPanel;
```

```
public class Painel extends JPanel{
```

```
}
```

Na nossa classe principal, vamos criar um objeto do tipo "Painel" e chamar de "meuPainel".

Para adicionar esse JPanel em nosso JFrame, usamos o método add que recebe como argumento um JPanel:

```
Painel meuPainel = new Painel();
```

```
janela.add(meuPainel);
```

Embora tenha adicionado um Panel ao seu Frame, você não vai ver nada quando rodar, pois não adicionou nenhum elemento ao seu Panel.

Nos próximos tutoriais desta seção de GUI, vamos aprender como adicionar elementos no Panel.

Mas para não deixar você 'na mão', vamos mostrar um código da Panel que vai fazer um desenho bem bacana no seu JPanel/JFrame, o código completo é:

framesPanels.java

import javax.swing.JFrame;

public class framesPanels {

public static void main(String[] args) {

JFrame janela = new JFrame("Meu primeiro frame em Java");

Painel meuPainel = new Painel();

janela.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

janela.add(meuPainel);

janela.setSize(600,400);

janela.setVisible(true);

}

```
}
```

Painel.java

```
import java.awt.Graphics;
```

```
import javax.swing.JPanel;
```

```
public class Paine1 extends JPanel{
```

```
public void paintComponent( Graphics g ){
```

```
    super.paintComponent( g );
```

```
        int pixel=0;
```

```
    for(pixel=0 ; pixel <= getHeight() ; pixel += 10){
```

```
        g.drawLine(0, pixel, pixel, getHeight());
```

```
    }
```

```
    for(pixel=getHeight() ; pixel >=0 ; pixel -= 10){
```

```
        g.drawLine(0, pixel, getHeight() - pixel, 0);
```

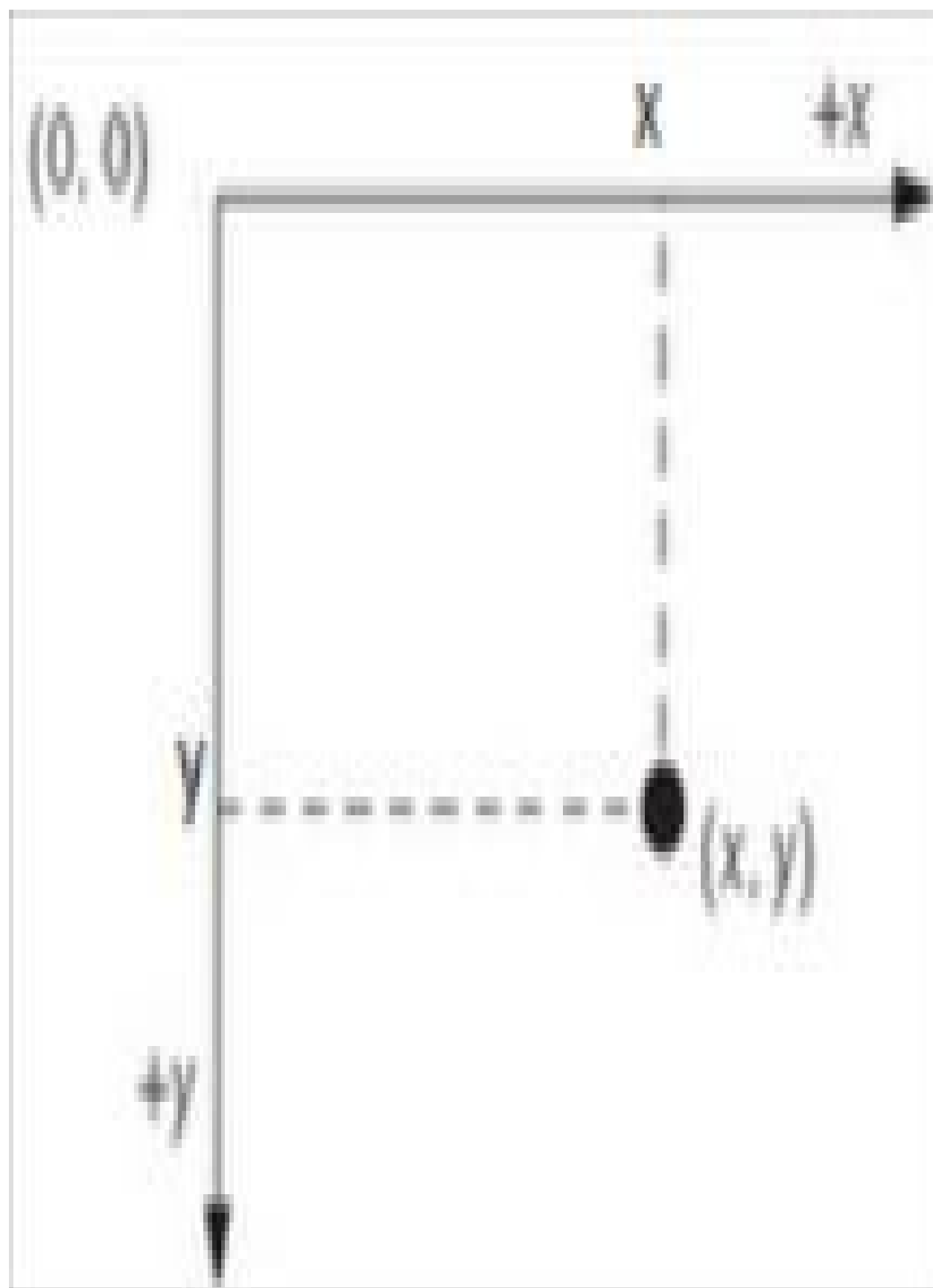
```
    }
```

```
for(pixel=0 ; pixel <= getHeight() ; pixel +=10){  
    g.drawLine(getWidth(), pixel, getWidth() - pixel, getHeight());  
}  
  
for(pixel=getHeight() ; pixel >=0 ; pixel -= 10){  
    g.drawLine(getWidth(), pixel, getWidth() - (getHeight() - pixel), 0);  
}  
  
}  
  
}
```

Para mais detalhes sobre métodos e outras funcionalidades do JPanel, segue a documentação:

<http://docs.oracle.com/javase/1.4.2/docs/api/javax/swing/JPanel.html>

Desenhando Linhas



Vamos começar esta seção de programação gráfica com um conceito muito importante: as coordenadas da tela.

A origem é no canto superior esquerdo da tela.

O eixo x (horizontal) começa na esquerda e cresce em direção a direita.

O eixo y (vertical) começa de cima e cresce para baixo.

Desenhando Linhas em Java

Caso você não se lembre bem de suas aulas de geometria, para definir uma linha precisamos apenas de dois pontos. Ou seja, definindo dois pontos, temos automaticamente uma linha.

Para desenhar uma linha em Java, as coisas serão levemente diferentes.

Iremos fornecer um ponto, que é o ponto de origem, ou seja, dois números: primeiro o do eixo x, depois o número do eixo y.

Guarde bem essa ordem, ela é universal: primeiro a largura (eixo horizontal x), depois a altura (eixo vertical y).

Depois, vamos fornecer mais dois números: a largura e a altura.

Vamos usar esses nomes em inglês: width e height.

Pronto, para desenhar uma linha em Java devemos fornecer 4 números: dois números que caracterizam a origem da linha, e dois que vão dizer até onde ela 'vai', que é o width e height.

O método que vamos usar é o drawLine, que recebe 4 argumentos, que são os 4 números para criar a linha.

Esse método se encontra na classe Graphics, da package awt. Logo, temos que importar tal classe:

```
import java.awt.Graphics;
```

Para mais informações dessa importante classe, acesse a documentação:

<http://docs.oracle.com/javase/1.4.2/docs/api/java/awt/Graphics.html>

Exemplo de código: Desenhando uma linha em Java

Criando o JPanel

Primeiro de tudo, vamos criar nosso panel. Ele será uma classe chamada DrawLines.

Para ser um JPanel basta fazer um extends nessa classe DrawLines.

Como vamos fazer um desenho, usaremos sempre um método chamado paintComponent, que recebe um objeto do tipo Graphics. Costuma se chamar esse objeto de 'g', e é ele que realmente fará os desenhos, através de seus métodos.

Antes, disso porém, precisamos invocar um método da superclasse e passar o objeto que vamos usar para fazer os desenhos, no caso é o objeto 'g', para tal basta fazer isso:

```
super.paintComponent(g);
```

Agora, vamos de fato, desenhar a bendita linha.

Para fazer isso, usamos o método drawLine, que é um nome bem sugestivo e óbvio, se você sabe inglês.

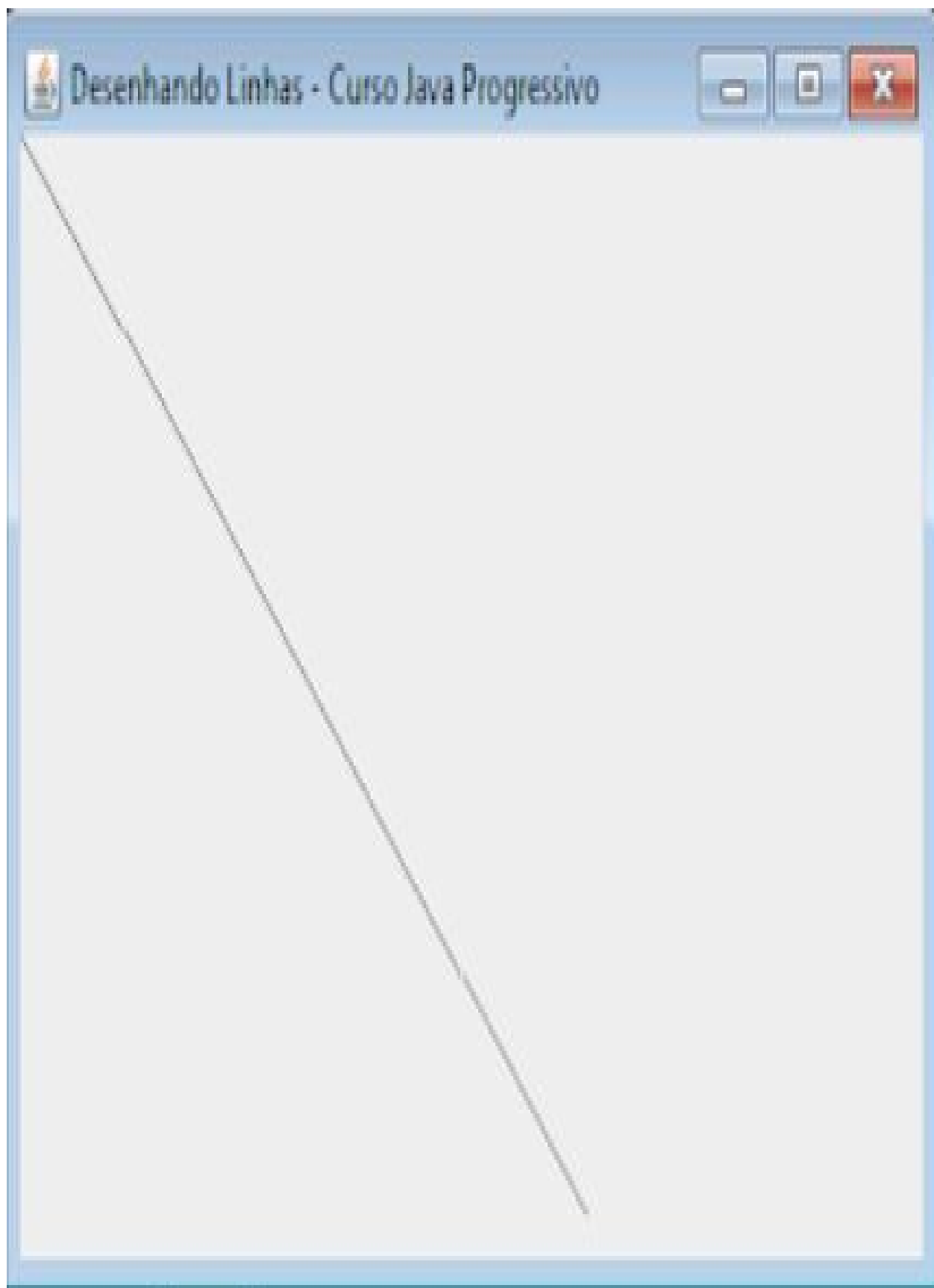
Como já dissemos, é um método que vai receber 4 números. Os dois primeiros são o ponto de origem e os dois últimos o fim da linha.

Criando o JFrame

De panel criado, vamos criar o frame. Vamos fazer isso na classe principal.

Vamos criar um objeto de nome frame, e passar uma string para o construtor.

Esse string ("Desenhando Linhas - Curso Java Progressivo") ficará no título, veja:



Vamos definir algumas coisas em nosso frame:

`frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE)` : define o que acontece no frame quando tentamos fechar (o frame se encerra)

`frame.setSize(250, 250)` : define o tamanho do frame, 250 de largura por 250 de altura

`frame.setVisible(true)` : define se o frame será, naquele instante, visível

`frame.add(panel)` : finalmente vamos adicionar nesse frame o panel que criamos

Por fim, nosso código que cria uma linha que sai da origem do frame até o ponto (250,250) é:

-->**Lines.java**

import javax.swing.JFrame;

public class Lines {

public static void main(String[] args) {

 DrawLines panel = new DrawLines();

 JFrame frame= new JFrame("Desenhando Linhas - Curso Java

```
Progressivo");
```

```
    frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
```

```
    frame.setSize( 250, 250 );
```

```
    frame.setVisible( true );
```

```
    frame.add( panel );
```

```
    }
```

```
}
```

-->DrawLines.java

```
import java.awt.Graphics;
```

```
import javax.swing.JPanel;
```

```
public class DrawLines extends JPanel{
```

```
public void paintComponent( Graphics g ){
```

```
    super.paintComponent( g );
```

```
    g.drawLine( 0, 0, 250, 250 );
```


}

}

O que são e como usar os métodos `getWidth()` e `getHeight()`

No artigo anterior de nossa apostila de Java, mostramos como criar um frame, adicionar um panel nele e criar uma linha nesse panel.

Porém, aplicações gráficas não são estáticas. Ao minimizar, maximizar, mover de lugar, colocar a janela de outro programa por cima de nossa janela e uma infinidade de outros fatores fazem com que nosso aplicativo mude durante sua execução.

Note que no exemplo do código de nosso último artigo, criamos uma linha que vai do ponto (0,0) e vai até o ponto onde a largura (width) é 250 e a altura (height) é 250.

Ou seja, aquela linha liga os pontos (0,0) até o (250,250), e só.

Tente maximizar a tela e veja o que ocorre: a linha só existe de (0,0) até o (250,250).

E se você quiser sempre uma linha que ligue a extremidade superior esquerda até a extremidade inferior direita?

Você precisará das atuais informações de tamanho da tela do aplicativo, mas o Java armazena essas informações e podemos obter esses dados através dos métodos `getWidth()` e `getHeight()`.

Os métodos `getWidth()` e `getHeight()`

Usamos os métodos `getWidth()` e `getHeight()` para pegar as informações atuais da largura e altura da janela que o usuário está visualizando.

Agora, em vez de criar um desenho que liga o ponto (0,0) até o (250,250), vamos ligar a origem ao ponto (`getWidth()`, `getHeight()`).

Ou seja, sempre a linha sai da origem e termina na extremidade inferior direita, criando uma diagonal.

Veja como ficou quando alongamos a tela:

Usando <code>getWidth()</code> e <code>getHeight()</code> , a linha sempre toca a ponta inferior direita
--

Exemplo de código: Desenhando duas linhas em Java, usando getWidth() e getHeight()

Vamos agora, finalmente, desenhar algo em nosso frame/panel.

Vamos criar uma linha que saia da origem e termine no canto inferior direito.

As coordenadas dessa linha serão: origem(0,0), fim(getWidth() , getHeight())

E outra que vai ligar a outra diagonal.

Ou seja a origem é no (0, getHeight()) e o término da linha em (getWidth(), 0).

Concorda?

Caso tenha dúvidas no código, ele foi inteiramente explicado no artigo passado.

Nosso código para desenhar essa linha será:

-->Lines.java

```
import javax.swing.JFrame;
```

```
public class Lines {
```

```
public static void main(String[] args) {  
  
    DrawLines panel = new DrawLines();  
  
    JFrame frame= new JFrame("Desenhando Linhas - Curso Java  
Progressivo");  
  
    frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );  
  
    frame.setSize( 250, 250 );  
  
    frame.setVisible( true );  
  
    frame.add( panel );  
  
} }
```

-->**DrawLines.java**

```
import java.awt.Graphics;  
  
import javax.swing.JPanel;  
  
public class DrawLines extends JPanel{  
  
public void paintComponent( Graphics g ){
```

```
super.paintComponent( g );
```

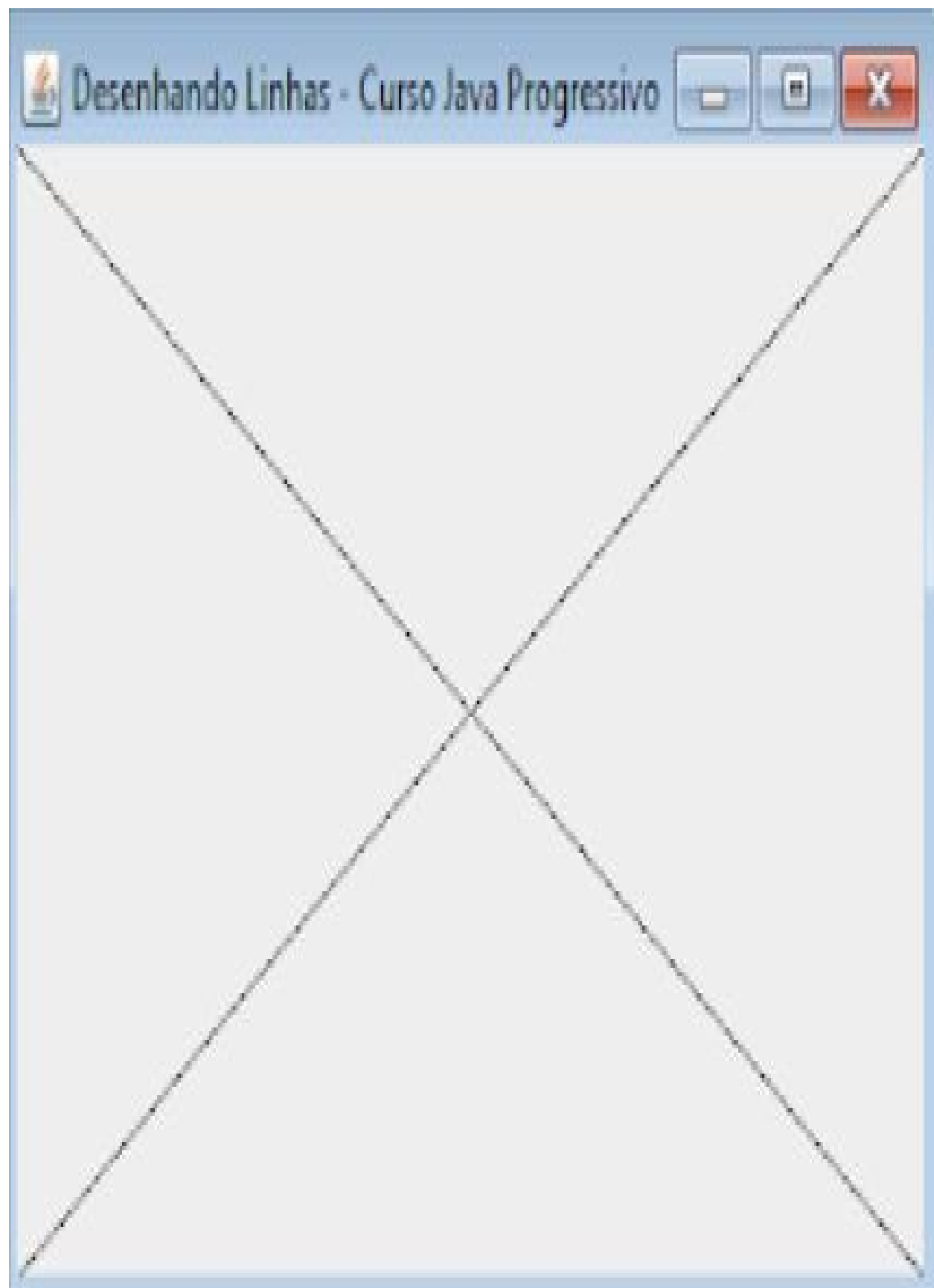
```
g.drawLine( 0, 0, getWidth(), getHeight() );
```

```
g.drawLine( 0, getHeight(), getWidth(), 0 );
```

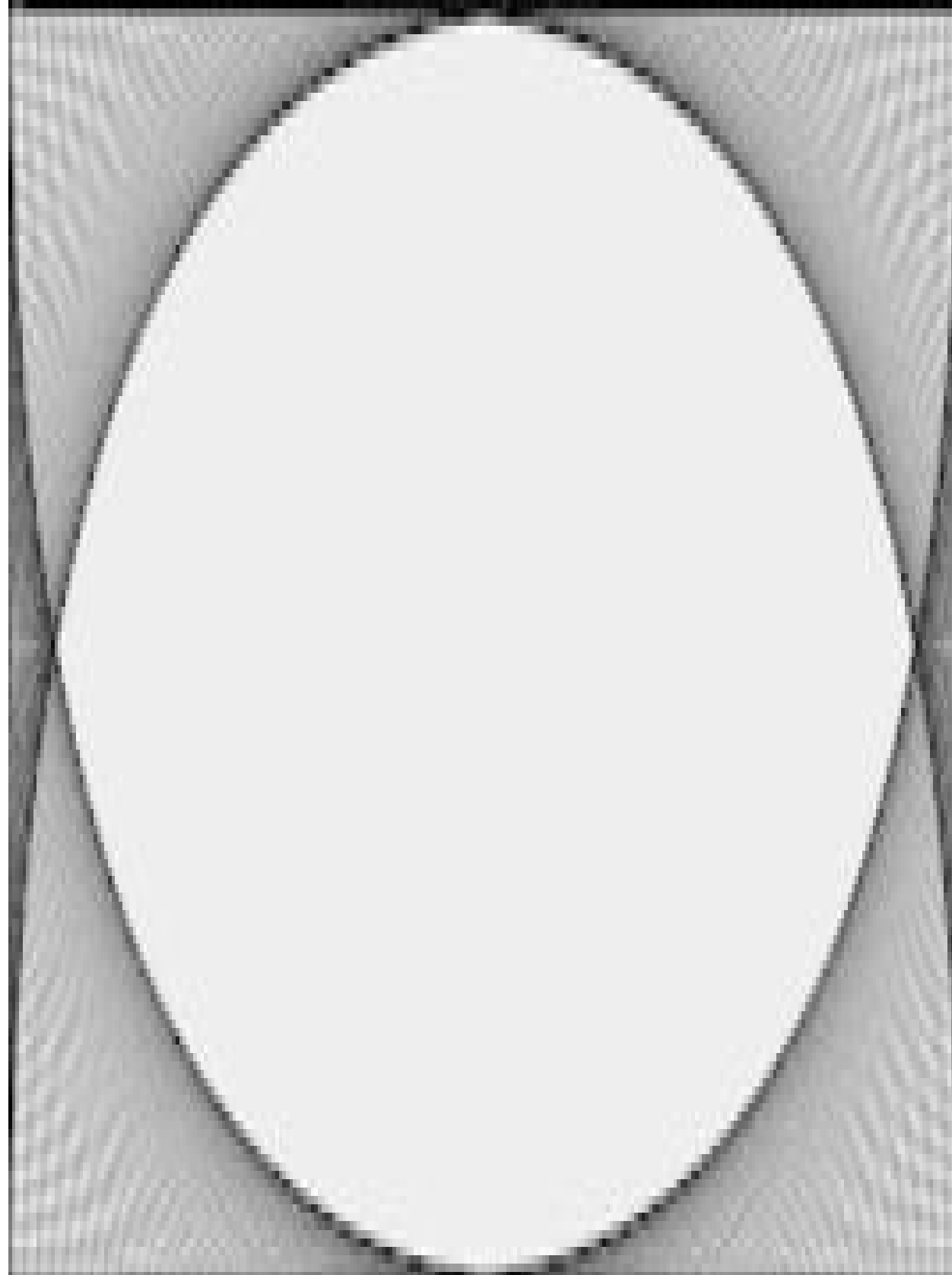
```
}
```

```
}
```

Você deverá ver o seguinte desenho em sua aplicação Java:



Criando alguns desenhos interessantes apenas com Linhas

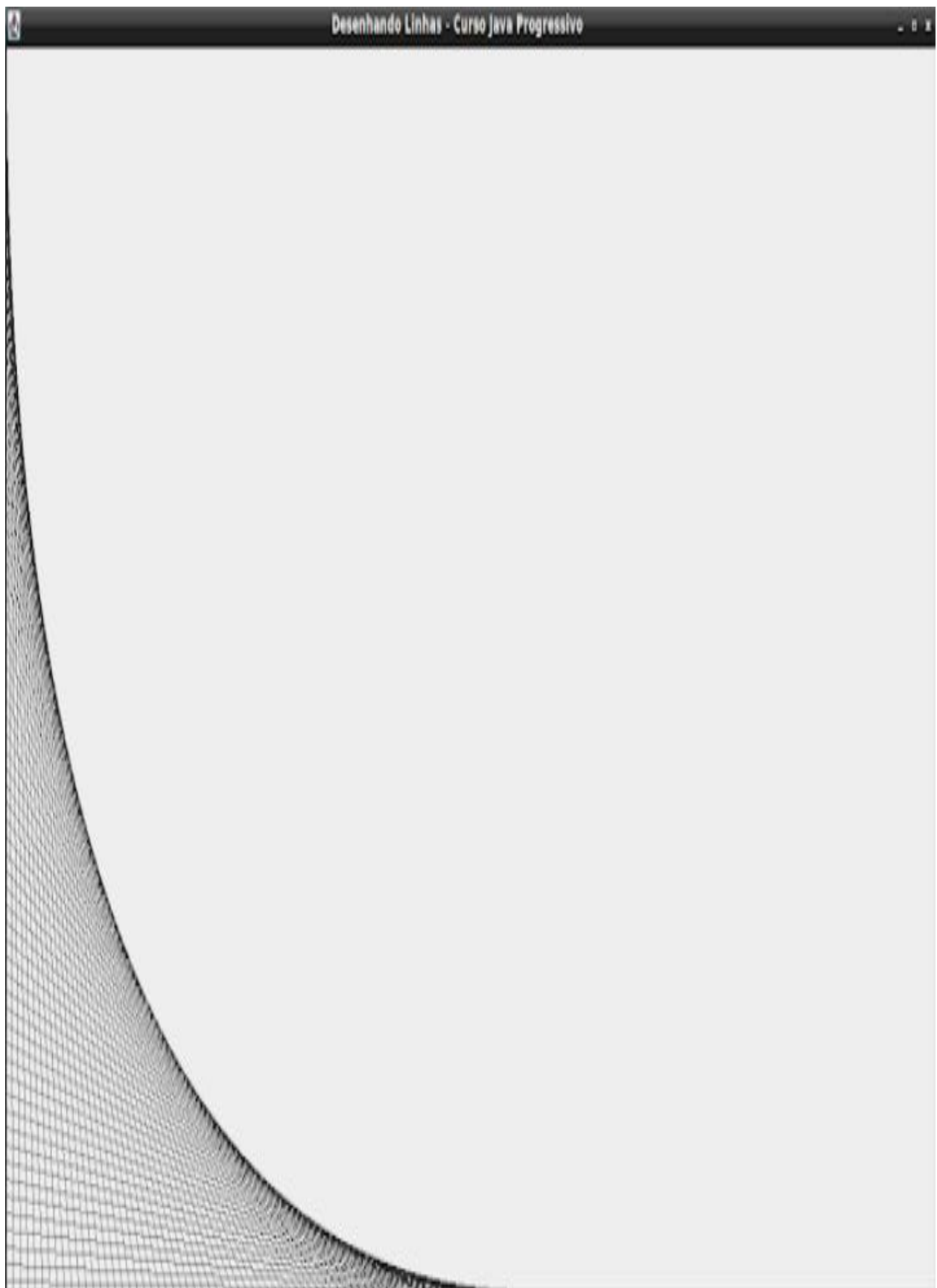


Para mostrar melhor como usar linhas em programação gráfica em Java, vamos ensinar e mostrar o código de diversos programas que geram belas figuras, como essa abaixo. Para isso, vamos usar o código de nosso aplicativo do artigo sobre desenho de linhas.

Caso não tenha visto, pegue o código em nosso tutorial sobre como criar linhas em Java.

Para criar esses desenhos, vamos apenas alterar o conteúdo do método `paintComponent`.

Primeiro desenho:



Bonito não?

Parece complicado, mas é bem simples.

Criamos aqui, várias linhas. Somente linhas.

A primeira linha começa no ponto (0,0) e vai até (0, getHeight()).

A segunda começa um pouco mais abaixo, em (0, 10) e vai até (10, getHeight()).

A terceira começa mais abaixo, em (0,20) e vai até (20, getHeight()).

...

A última linha começa em (0, getHeight()) e termina em (getHeight() , getHeight())

Ou seja, criamos linha que começam na lateral esquerda (width=0), na origem, e tocam a parte inferior do frame (height = getHeight()).

A cada iteração, vamos a origem descer em 10 pixels, ao passo que o final da linha vai crescendo, na horizontal, de 10 em 10 também.

Então, a origem vai decendo de 10 em 10, e o final da linha vai pra direita, de 10 em 10 também.

O código do método paintComponent para produzir tal figura, é:

```
public void paintComponent( Graphics g ) {
```

```
super.paintComponent( g );
```

```
int pixel=0;
```

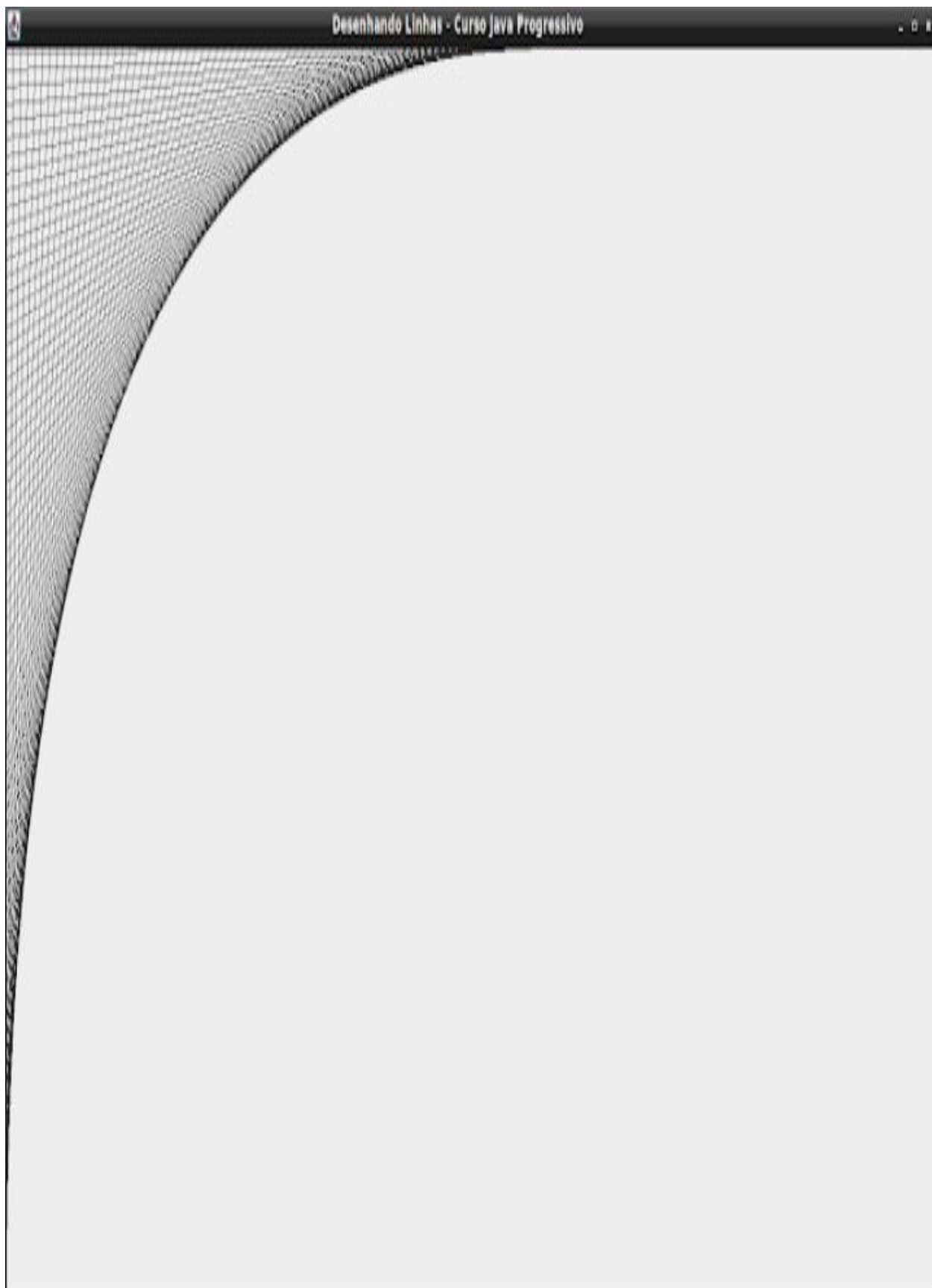
```
for(pixel=0 ; pixel <= getHeight() ; pixel += 10){
```

```
    g.drawLine(0, pixel, pixel, getHeight());
```

```
}
```

```
}
```

Segundo desenho:



O que vamos fazer agora se assemelha ao desenho anterior.

Agora a origem das retas está na borda inferior.

A primeira linha começa em (0, getHeight()) e termina em (0, 0).

Depois essa origem sobe e o fim vai pra direita, e começa em (0, getHeight() - 10), e termina em (10, 0).

A terceira linha continua a subir com sua origem, agora em (0, getHeight() -20) e seu fim vai mais pra direita, terminando em (20, 0).

...

Logo, o código do método paintComponent pra gerar essa figura é:

```
public void paintComponent( Graphics g ){
```

```
    super.paintComponent( g );
```

```
    int pixel=0;
```

```
    for(pixel=getHeight() ; pixel >=0 ; pixel -= 10){
```

```
        g.drawLine(0, pixel, getHeight() - pixel, 0);
```

```
    }
```

```
}
```

Unindo os dois códigos, o método paintComponent será:

```
public void paintComponent( Graphics g ){
```

```
super.paintComponent( g );
```

```
int pixel=0;
```

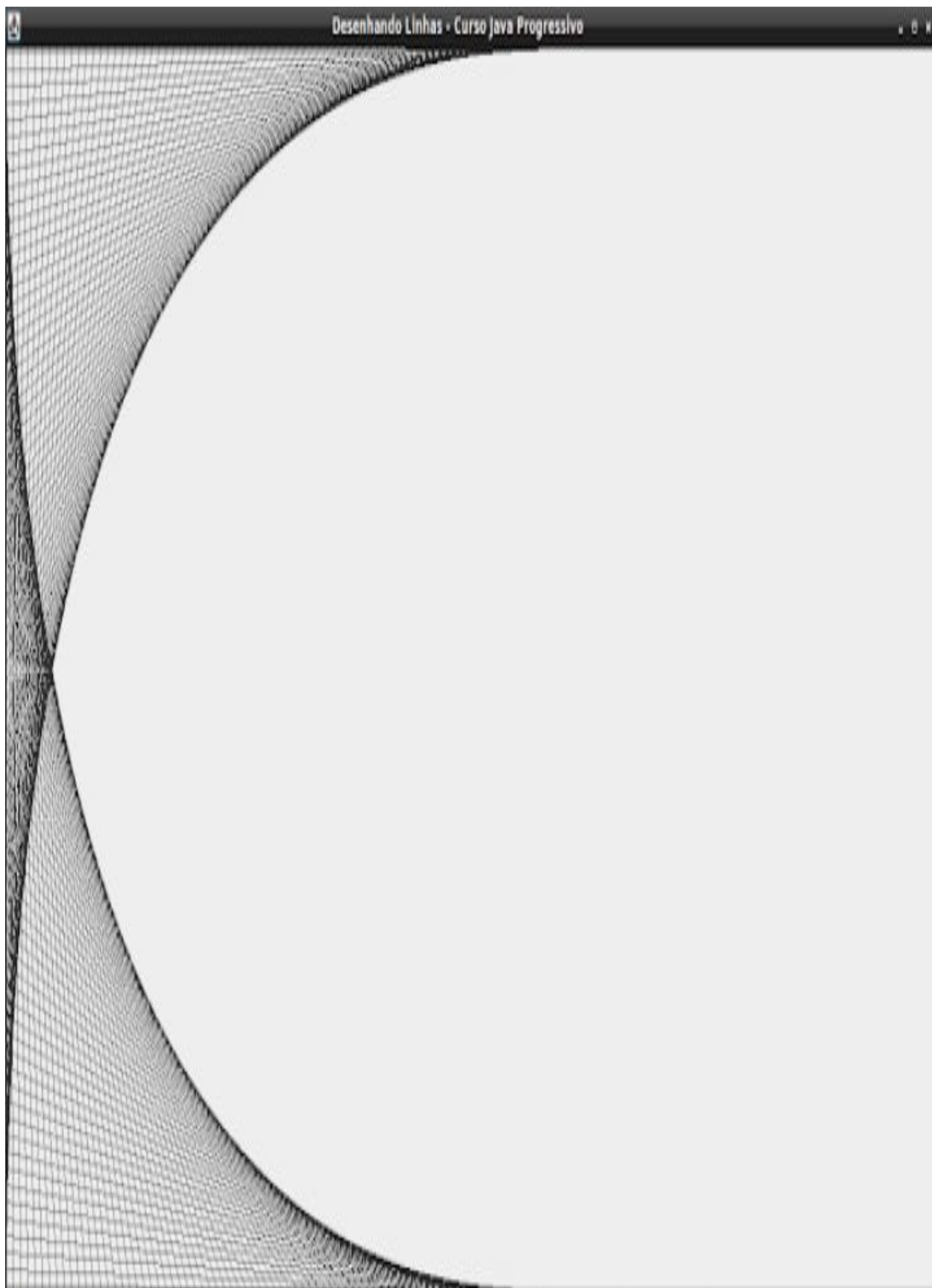
```
for(pixel=0 ; pixel <= getHeight() ; pixel += 10){
```

```
    g.drawLine(0, pixel, pixel, getHeight());
```

```
}
```

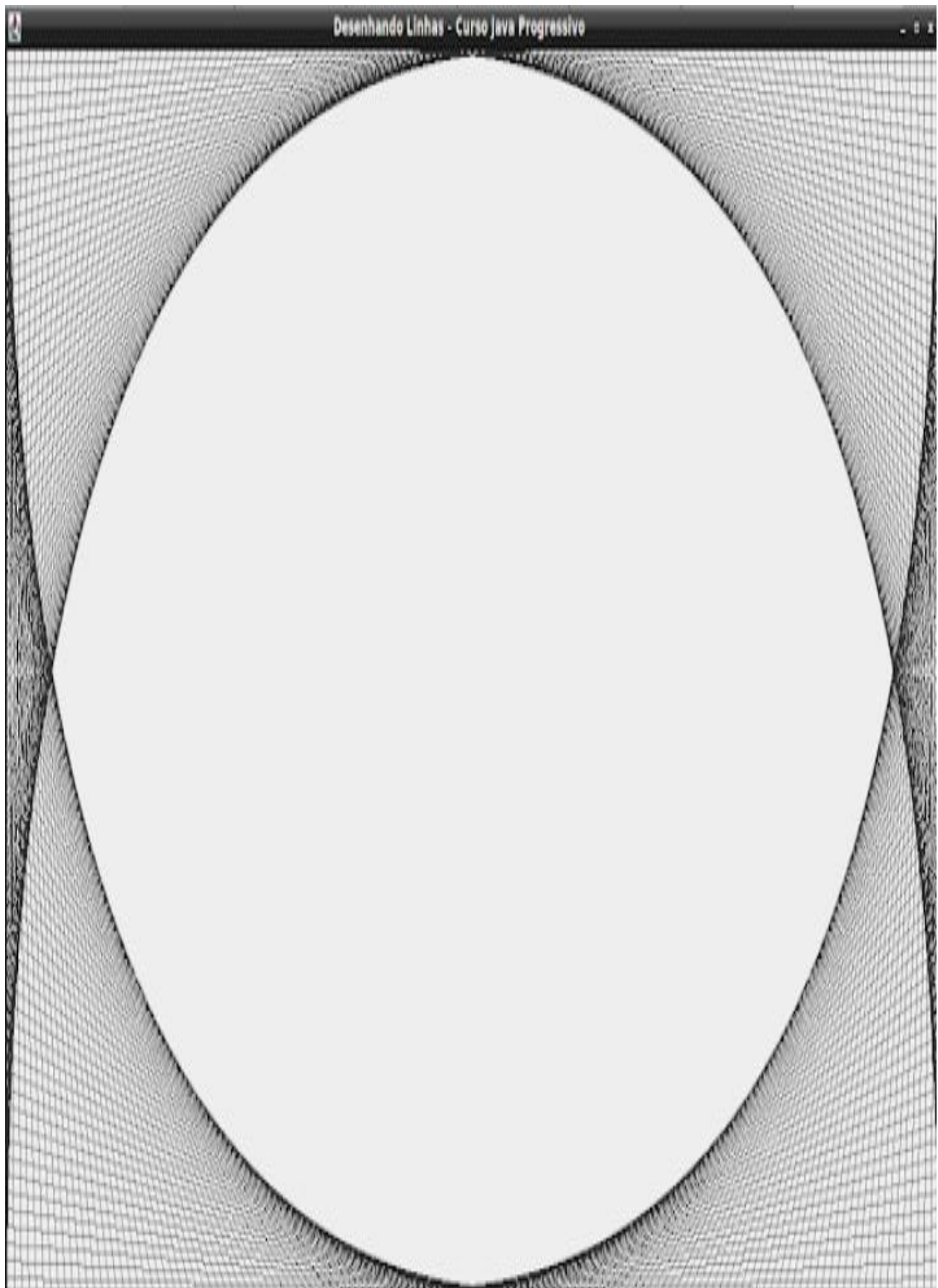
```
for(pixel=getHeight() ; pixel >=0 ; pixel -= 10){ g.drawLine(0, pixel,  
getHeight() - pixel, 0); } }
```

Que resulta no belo desenho:



Exercício:

Com base no que foi criado e ensinado nesse artigo, programe um aplicativo em Java que gere o seguinte desenho, usando apenas linhas através do método `drawLine`:



Tente até a morte. Após algumas horas quebrando a cabeça sem conseguir gerar o desenho, olhe o código a seguir.

Mas lembre-se: só se aprende tentando.

```
public void paintComponent( Graphics g ){
```

```
super.paintComponent( g );
```

```
int pixel=0;
```

```
for(pixel=0 ; pixel <= getHeight() ; pixel += 10){
```

```
    g.drawLine(0, pixel, pixel, getHeight());
```

```
}
```

```
for(pixel=getHeight() ; pixel >=0 ; pixel -= 10){
```

```
    g.drawLine(0, pixel, getHeight() - pixel, 0);
```

```
}
```

```
for(pixel=0 ; pixel <= getWidth() ; pixel +=10){
```

```
    g.drawLine(getWidth(), pixel, getWidth() - pixel, getHeight());
```

```
}
```

```
for(pixel=getHeight() ; pixel >=0 ; pixel -= 10){  
    g.drawLine(getWidth(), pixel, getWidth() - (getHeight() - pixel), 0);  
}  
}
```

Como desenhar retângulos e quadrados

Agora que você já aprendeu o básico de JFrame e JPanel, bem como desenhar linhas em Java (usando os importantes métodos `getWidth()` e `getHeight()`) e fez belos desenhos somente usando as linhas.

Agora vamos prosseguir em nosso curso de programação gráfica e desenhar retângulos e quadrados.

Para tal, usaremos o método `drawRect`, de `draw` (desenhar) e `Rect` de retângulo.

O método drawRect: como desenhar retângulos e quadrados em Java

Assim como método drawLine, o método drawRect pertence ao objeto da classe Graphics, no nosso caso ele se chamará 'g'.

Também como o o drawLine, o drawRect recebe 4 argumentos, mas não são todos coordenadas.

Os dois primeiros números são as coordenadas da origem (distância horizontal e distância vertical).

Os dois últimos são a largura e a altura do retângulo em relação ao ponto de origem.

Ou seja, para desenhar um retângulo que tem origem em (0,0) e largura 50 e altura 100, fazemos:

```
g.drawRect(0,0,50,100);
```



Desenhando Retângulos - Curso Java Progressivo



Para desenhar um retângulo que tem sua origem no centro do frame com largura 50 e altura 100 (em relação a origem), fazemos:

```
g.drawRect( getWidth()/2 , getHeight()/2, 50, 100);
```



Desenhando Retângulos - Curso Java Progressivo



Note que é a largura que é 50, e a altura é 100.

50 e 100 NÃO SÃO AS COORDENADAS DE UM PONTO! 50 é a largura do retângulo, a partir da origem, e 100 é a altura do retângulo, em relação a origem.

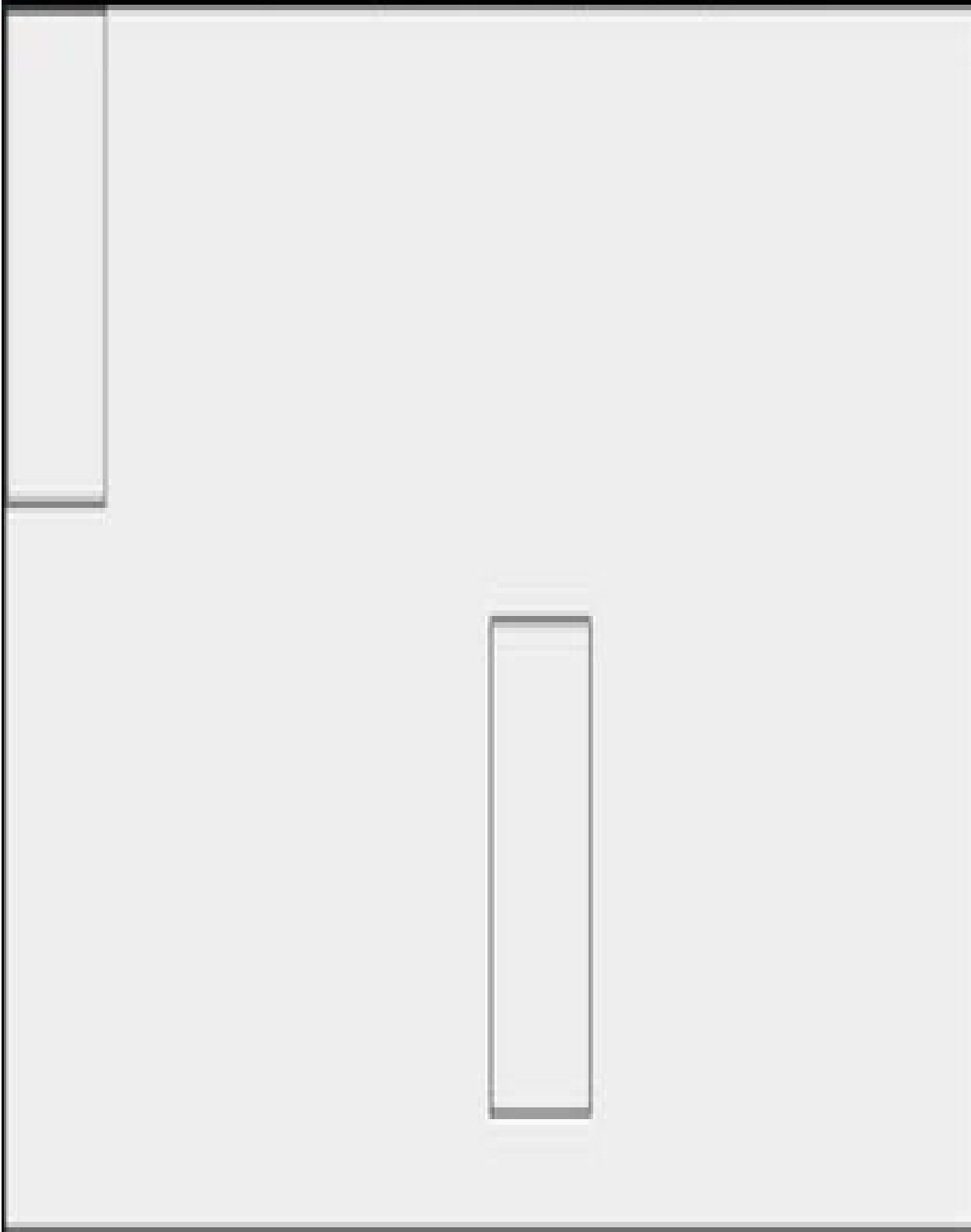
Veja quando colocamos os dois códigos juntos:

```
g.drawRec(0,0,50,100);
```

```
g.drawRect( getWidth()/2 , getHeight()/2, 50, 100);
```



Desenhando Retângulos - Curso Java Progressivo



Desenhando um Cubo

Se você não se lembra bem, um cubo nada mais é que 6 quadrados. E se realmente não é bom em geometria, um quadrado é um retângulo de lados iguais.

Trazendo para nosso conceito de programação gráfica em Java, quadrado é desenhado pelo método `drawRect` com largura igual a altura.

Vamos iniciar nosso cubo então.

Primeiro, vamos desenhar dois quadrados, de lado 100. Criamos um, e o segundo criamos de modo que sua origem esteja dentro do primeiro quadrado.

A origem do primeiro será em:

`(t.getWidth()/10 , getHeight()/10)`

Vamos colocar a origem do segundo no centro do primeiro quadrado.

As coordenadas do centro do primeiro quadrado são:

Eixo x: `larguraOrigem + 50`

Eixo y: `alturaOrigem + 50`

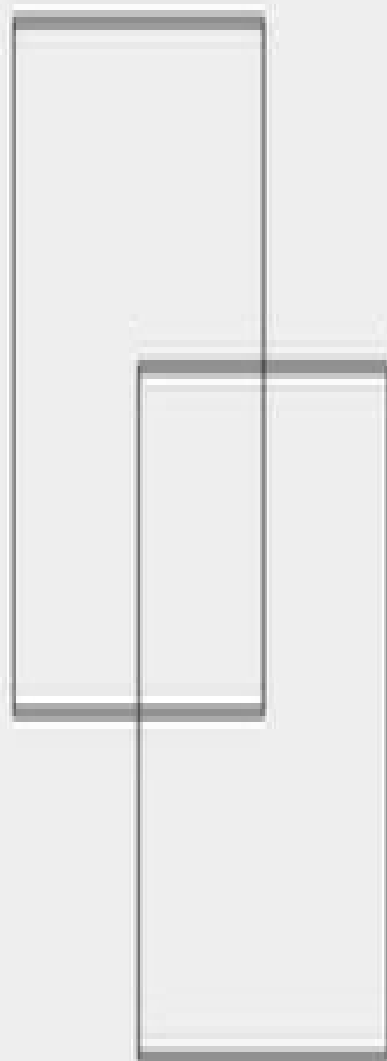
Então, desenhamos o segundo quadrado da seguinte maneira:

```
g.drawRect( (getWidth()/10) + 50, (getHeight()/10) + 50, 100, 100);
```

Temos a seguinte figura:



Desenhando Retângulos - Curso Java Progressivo



A próxima parte é criar 4 linhas, que ligarão os vértices.

A linha que liga os vértices (cantos) superiores esquerdo:

```
g.drawLine(getWidth()/10, getHeight()/10, (getWidth()/10) + 50,  
(getHeight()/10) + 50);
```

A linha que liga os vértices inferiores esquerdo:

```
g.drawLine(getWidth()/10, (getHeight()/10) + 50, (getWidth()/10) + 100,  
(getHeight()/10) + 50 + 100);
```

A linha que liga os vértices superiores direitos:

```
g.drawLine( (getWidth()/10) + 100, getHeight()/10, (getWidth()/10) + 50 + 100,  
(getHeight()/10) + 50);
```

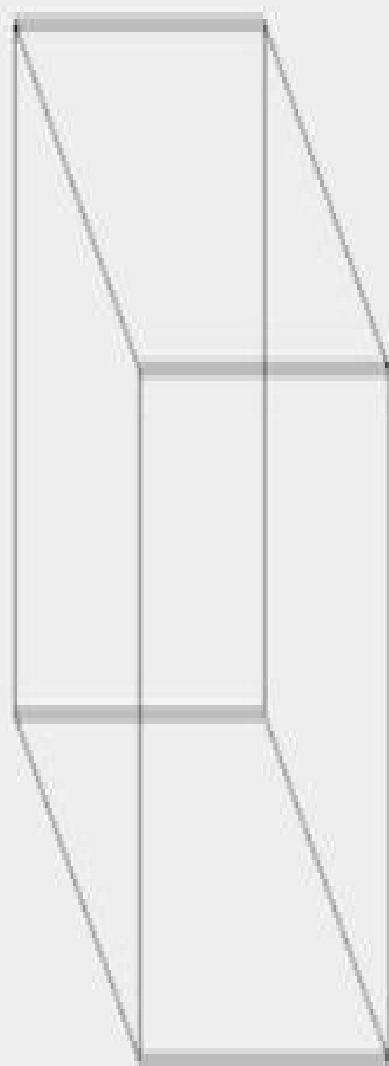
A linha que liga os vértices inferiores direito:

```
g.drawLine( (getWidth()/10) + 100, (getHeight()/10) + 100, (getWidth()/10) +  
100 + 50, (getHeight()/10) + 100 + 50);
```

Obtemos, portanto, a figura do seguinte cubo:



Desenhando Retângulos - Curso Java Progressivo



O código do nosso método paintComponent que desenha esse cubo, fica:

```
public void paintComponent( Graphics g ){
```

```
super.paintComponent( g );
```

```
g.drawRect(getWidth()/10 , getHeight()/10,100,100);
```

```
g.drawRect( (getWidth()/10) + 50, (getHeight()/10) + 50, 100, 100);
```

```
g.drawLine(getWidth()/10, getHeight()/10, (getWidth()/10) + 50,  
(getHeight()/10) + 50);
```

```
g.drawLine(getWidth()/10, (getHeight()/10) + 100, (getWidth()/10) + 50,  
(getHeight()/10) + 50 + 100);
```

```
g.drawLine( (getWidth()/10) + 100, getHeight()/10, (getWidth()/10) + 50 +  
100, (getHeight()/10) + 50);
```

```
g.drawLine( (getWidth()/10) + 100, (getHeight()/10) + 100, (getWidth()/10) +  
100 + 50, (getHeight()/10) + 100 + 50);
```

```
}
```

Exercícios sobre linhas & retângulos:

1. Desenhe o tabuleiro do jogo campo minado que fizemos em nosso curso.
2. Desenhe o tabuleiro do jogo batalha naval que fizemos em nosso curso.
3. Desenhe o tabuleiro do jogo da velha que fizemos em nosso curso.

Como colocar um menu de cores: Usando o JcolorChooser

Sabe aquelas opções de cores, transparência, tom, brilho etc, que você vê em programas de imagens por aí?

Pois é, bote no seu aplicativo Java também.

Mas calma, é bem simples. Já está tudo pronto, e o nome disso é JColorChooser.

JColorChooser - Escolher cores em Java

Objetos Graphics contém os métodos para desenhar, manipular fontes, cores e outras coisas gráficas.

É uma classe abstrata, ou seja, não pode ser instanciada, isso por conta da portabilidade, já que cada dispositivo desenha de seu próprio jeito.

Para usar a Graphics, adicionamos:

```
import java.awt.Graphics;
```

Para inserirmos componentes gráficas, como linhas, retângulos etc, usamos uma instância da Graphics:

```
public void paintComponent( Graphics g)
{
    //aqui os métodos para desenho
}
```

As cores são formadas por três níveis de cores, vermelho(R, red), verde(G, green) e azul(B, blue).

O famoso RGB.

Para usarmos as cores, adicionamos:

```
import java.awt.Color;
```

```
public Color(int r, int g, int b): inteiros variando de 0 a 255
```

```
public Color(float r, float g, float b): float de 0.0 a 1.0
```


- Mais métodos

```
public int getRed() ;
```

```
public int getGreen();
```

```
public int getBlue();
```

```
public Color getColor();
```

```
public void setColor( Color c);
```

Vamos mostrar uma aplicação que mostra o JColorChooser, uma componente GUI que exibe todo o padrão de cores no JColorChosser dialog.

Quando escolhemos uma cor, a JPanel muda de cor.

-----ColorsFrame.java

```
import java.awt.BorderLayout;
```

```
import java.awt.Color;
```

```
import java.awt.event.ActionEvent;
```

```
import java.awt.event.ActionListener;
```

```
import javax.swing.JButton;
```

```
import javax.swing.JFrame;
```

```
import javax.swing.JColorChooser;

import javax.swing.JPanel;


public class ShowColors2JFrame extends JFrame
{

    private JButton changeColorJButton;

    private Color color = Color.LIGHT_GRAY;

    private JPanel colorJPanel;

    public ShowColors2JFrame()
    {

        super( "Usando o JColorChooser" );

        colorJPanel = new JPanel();

        colorJPanel.setBackground( color );

        changeColorJButton = new JButton( "Escolher a cor" );
```

```
changeColorJButton.addActionListener(
```

```
new ActionListener()
```

```
{
```

```
public void actionPerformed((ActionEvent event)
```

```
{
```

```
    color = JColorChooser.showDialog(
```

```
        ShowColors2JFrame.this, "Escolher a color", color );
```

```
    if( color == null )
```

```
        color = Color.LIGHT_GRAY;
```

```
        colorJPanel.setBackground( color );
```

```
    }
```

```
}
```

```
);
```

```
add( colorJPanel, BorderLayout.CENTER );
```

```
add( changeColorJButton, BorderLayout.SOUTH );
```

```
setSize( 400, 130 );
```

```
        setVisible( true );  
    }  
}
```

-----Colors.java

```
import javax.swing.JFrame;
```

```
public class Colors
```

```
{
```

```
public static void main(String[] args)
```

```
{
```

```
    ColorsFrame application = new ColorsFrame();
```

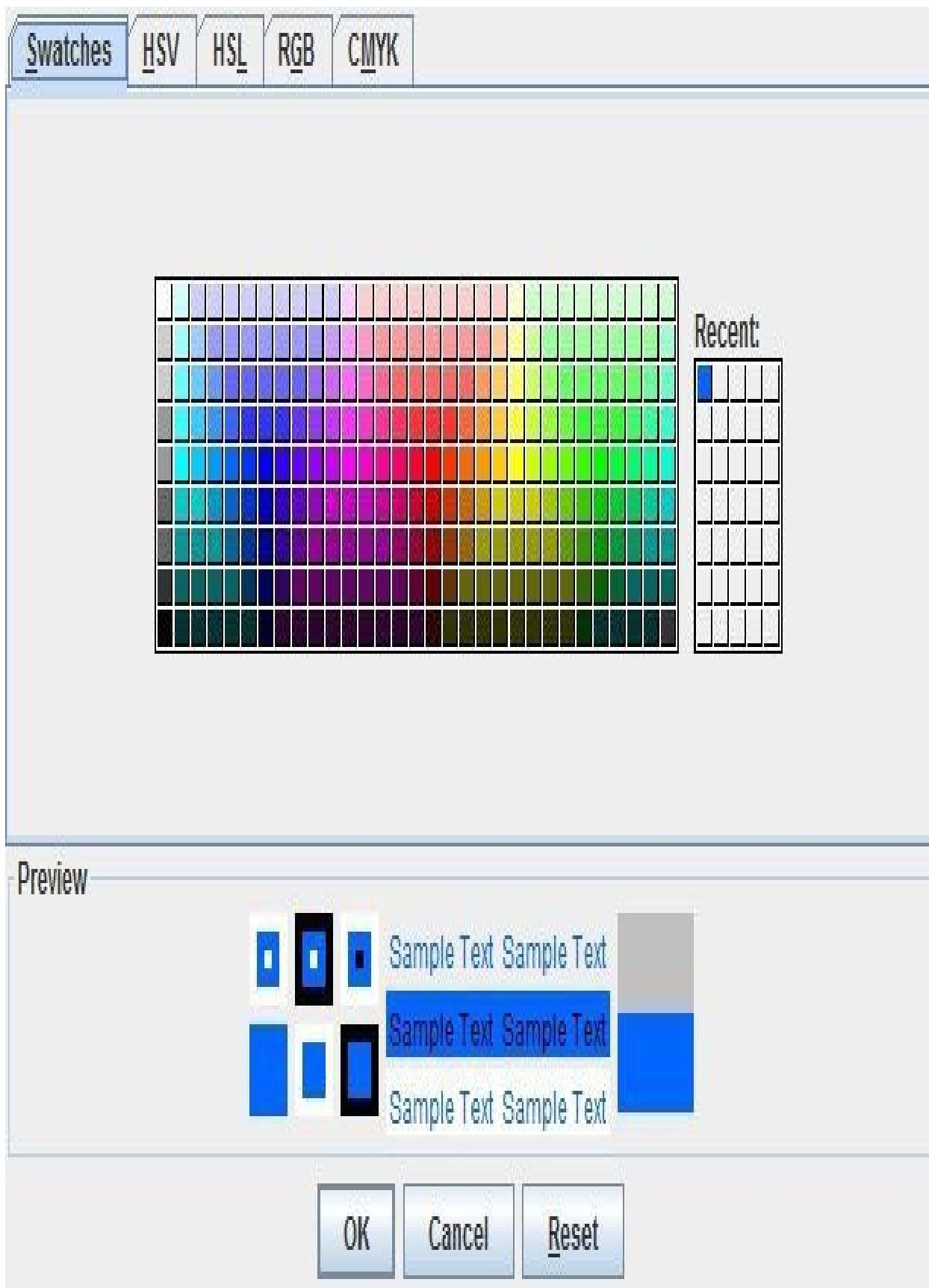
```
    application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
```

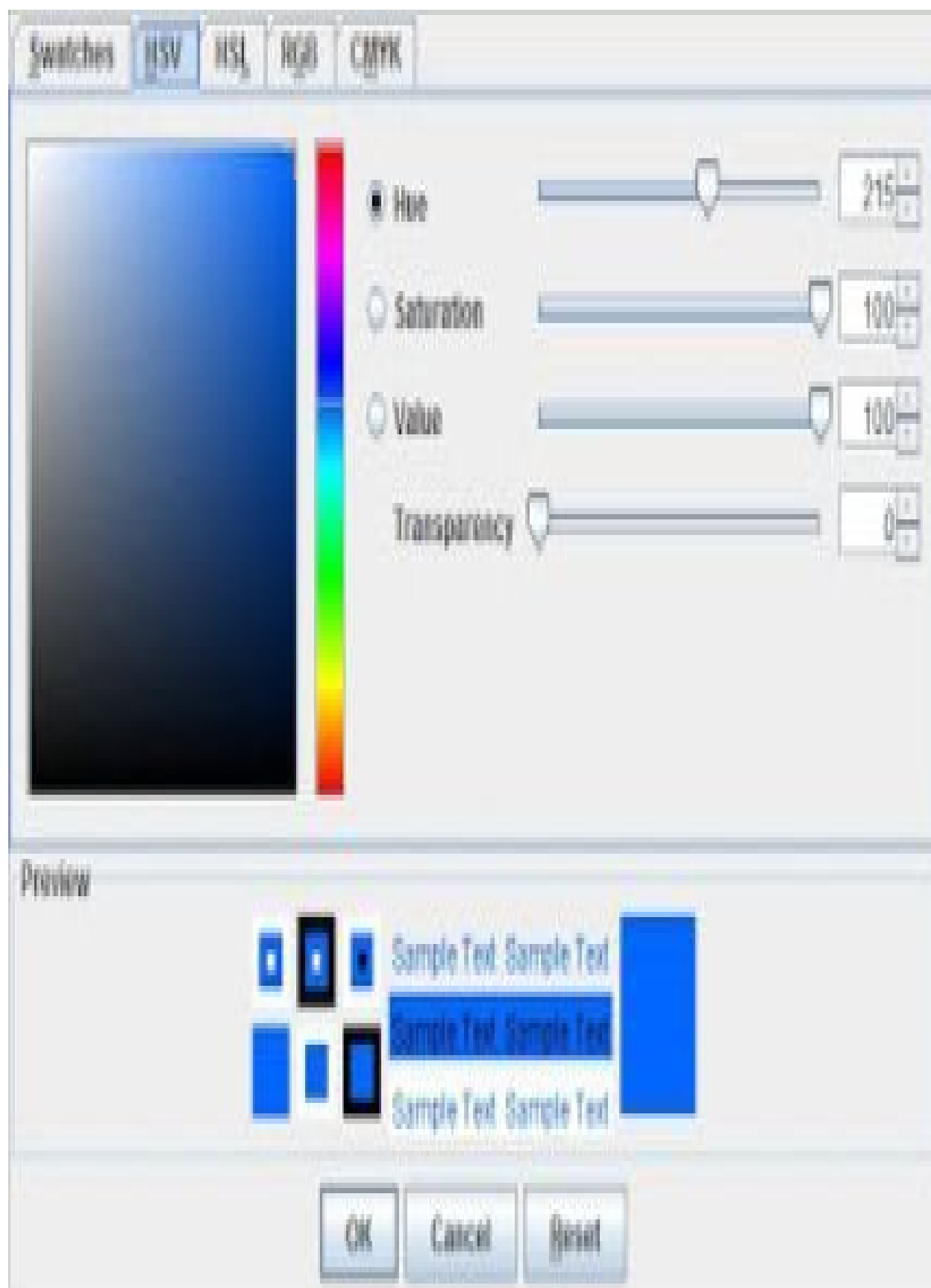
```
}
```

```
}
```

O resultado:








Swatches

HSV

HSL

RGB

CMYK



☒ Hue

215

☐ Saturation

100

☐ Lightness

50

Transparency

0

Preview







Sample Text

Sample Text







Sample Text

Sample Text



OK

Cancel

Reset


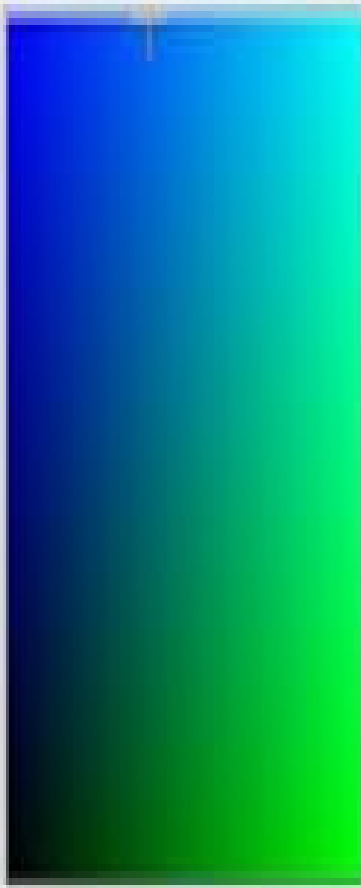
Swatches

HSV

HSB

RGB

CMYK




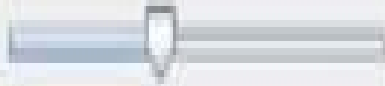
☒ Red

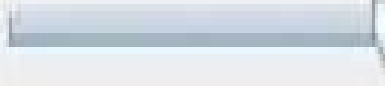
☐ Green


☐ Blue

Alpha









0

102

255

255

Color Code

0000FF

Preview





Sample Text Sample Text

Sample Text Sample Text

Sample Text Sample Text



OK

Cancel

Reset


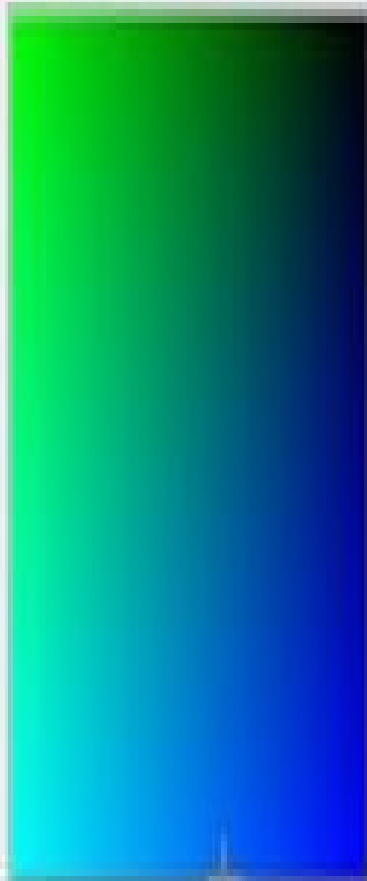
Swatches

HSV

HSB

HSL

CMYK



☒ Cyan

☐ Magenta

☐ Yellow

☐ Black

☐ Alpha

255

153

0

0

255

Preview

Sample Text

Sample Text

Sample Text

Sample Text

Sample Text

OK

Cancel

Reset



Como usar as fontes em Java

Sim, as fontes também são consideradas elementos gráficos em Java! Ou seja, também são 'pintadas' no frame (paintComponent).

Por isso, nesse artigo de nossa apostila de Java vamos aprender alguns recursos interessantes das fontes em Java.

A classe FONT: Escolhendo Fontes e efeitos em Java

A maior parte dos métodos e constantes, no que se refere as fontes, estão na classe Font.

Como:

public final static int PLAIN	public final static int PLAIN
public final static int BOLD	public final static int BOLD
public final static int ITALIC	public final static int ITALIC
public Font(String name,int style, int size)	public Font(String name,int style, int
public int getStyle()	public int getStyle()
public int getSize()	public int getSize()
public String getName()	public String getName()
public String getFamily()	public String getFamily()
public boolean isPlain()	public boolean isPlain()
public boolean isBold()	public boolean isBold()
public boolean isItalic()	public boolean isItalic()
public Font getFont()	public Font getFont()
public void setFont(Font f)	public void setFont(Font f)

Estilo de fontes: Font.PLAIN, Font.ITALIC e Font.BOLD

O Java padrão possui 5 fontes: Serif, Monospaced, SansSerif, Dialog and DialogInput

Dentro de uma classe que estende a JPanel, usamos os recursos da Font com um objeto da Graphics, como se fosse um componente para a pintura/gráficos.

Para escrevermos strings coloridas, primeiro setamos a fonte, depois usamos o método drawString(), como segue :

```
public void paintComponent( Graphics g)
```

```
{
```

```
super.paintComponent(g);
```

```
g.setFont( new Font("Serif", Font.BOLD, 15) );
```

```
g.drawString(" Serif 15, negrito", 20, 30 );
```

```
g.setFont( new Font("Monospaced", Font.ITALIC, 15) );
```

```
g.drawString("Monospaced 15, itálico", 20, 30 );
```

```
g.setFont( new Font("SansSerif", Font.PLAIN, 15) );
```

```
g.drawString("SansSerif 15, plano", 20, 30 );

g.setColor( Color.RED);

g.setFont( new Font("Serif", Font.BOLD + Font.ITALIC, 15) );

g.drawString(g.getFont().getName() + " " + g.getFont().getSize() + "
point negrito e itálico", 20, 100 );

}
```

Note que para a fonte estar em negrito e em itálico, passamos a soma dos argumentos:

`Font.BOLD + Font.ITALIC`

Prático. Bem bolado, esse Java, não?

Há ainda a classe `FontMetrics`, que tem métodos para manipulação da altura da fonte e espaços (`height`, `descent`, `ascent` e `leading`):

`leading`: distância entre a base da acentuação (^, ~, ') e o nível superior

`ascent`: distância entre a base (baseline) e o teto de uma letra normal sem acentuação

`descent`: distância entre a base (baseline) e o limite inferior, em letras que tem 'perna', como y e g

`height`: toda a altura, desde a base da 'perna' até a parte de cima do acento: g ã

Consulte a documentação do Java para saber os métodos específicos para a `FontMetrics`:

<http://docs.oracle.com/javase/7/docs/api/>

Como desenhar Polígonos e Polilinhas em Java

Aprenderemos noções básicas sobre polígonos e como desenhar estes em Java nesse tutorial, como preenchê-los e escolher o formato de borda.

Desenhando polígonos e Polilinhas

Ambos, polígonos e polilinhas, são figuras geométricas formadas por segmento de linhas conectados.

A diferença é que os polígonos são sempre fechados e as polilinhas podem ser fechadas (aí será um polígono) ou não. Embora haja diferenças, ambos são tratados como classe “Polygon”.

Alguns métodos requerem objetos do tipo "Polygon" (da java.awt).

Vamos demonstrar alguns métodos para desenhar polígonos:

public void drawPolygon(int[] xPoints, int[] yPoints, int points)

Desenha um polígono. Recebe dois vetores de inteiros, com a localização dos pontos do eixo x, os pontos do eixo y e o número total de pontos do polígono.

Caso o último ponto seja diferente do primeiro, mais um segmento de linha será criado para conectar o último com o primeiro ponto, fechando o polígono.

public void drawPolyline(int[] xPoints, int[] yPoints, int points)

Desenha linhas conectadas. Recebe dois vetores de inteiros, com a localização dos pontos do eixo x, os pontos do eixo y e o número total de pontos da polilinha.

Caso o último ponto seja diferente do primeiro a polilinha não é fechada (não é um polígono).

```
public void drawPolygon( Polygon p )
```

Desenha o objeto Polygon.

```
public void fillPolygon( int[] xPoints, int[] yPoints, int points )
```

Mesmo método Polygon, mas este preenche (pinta) o polígono.

```
public void fillPolygon( Polygon p )
```

Desenha o objeto Polygon, porém preenchido.

Construtores (constructors) e métodos da classe Polygon:

public Polygon()

Criado o objeto polígono, sem pontos.

```
public Polygon( int[] xValues, int[] yValues, int numberOfPoints )
```

Cria o objeto polígono. Recebe dois vetores de inteiros, com a localização dos pontos do eixo x, os pontos do eixo y e o número total de pontos do polígono.

```
public void addPoint( int x, int y )
```

Adiciona um par de coordenadas ao polígono.

Para criar um painel para desenhar os polígonos, crie uma classe que estenda a JPanel e nela crie a paintComponent:

```
public class PolygonsJPanel extends JPanel
```

```
{
```

```
public void paintComponent( Graphics g )
```

```
{
```

```
super.paintComponent( g );
```

```
int[] xValues = { 20, 40, 50, 30, 20, 15 };
```

```
int[] yValues = { 50, 50, 60, 80, 80, 60 };
```

```
Polygon poligono = new Polygon( xValues, yValues, 6 );
```

```
g.drawPolygon( poligono );
```

```
}
```

```
}
```


Esses métodos e classes estão presentes em:

```
import java.awt.Graphics;
```

```
import java.awt.Polygon;
```

```
import javax.swing.JPanel;
```

Na sua classe main, crie um objeto dessa sua classe que desenha o polígono e add no seu frame.

A incrível API Java 2D

A API Java 2D é um conjunto de recursos que permitem uma ampla possibilidade para se trabalhar com desenhos, gráficos e qualquer coisa relacionada com imagem 2D em Java.

API Java 2D

A API Java 2D permite mais capacidades para os gráficos/desenhos feitos em 2D, além dos detalhados neste artigo.

Obviamente, o visto na seção Gráficos e 2D foi apenas uma parte do que é possível em Java, e especificamente, com essa API.

Caso queira algo mais específico para as suas necessidades, você pode obter mais informações no site da documentação do Java, da Oracle:

<http://download.oracle.com/javase/6/docs/technotes/guides/2d/>

Para usar a Java 2D API você terá que fazer uma referência ao pacote Graphics2D, que é uma subclasse da Graphics, que foi explorada nesse artigo. O objeto usado nos métodos 'paintComponent' são uma instância da subclasse da Graphics2D que é passada a esse método e acessada pela superclasse Graphics.

Alguns desenhos possíveis de serem feitos usando o pacote java.awt.geom:

Linha em 2D : Line2D.Double

Retângulo em 2D: Rectangle2D.Double

Retângulo em 2D com bordas arredondadas: RoundRectangle2D.Double

Arcos em 2D: Arc2D.Double

Elipses em 2D: `Ellipse2D.Double`

Para usá-las, defina:

```
Graphics2D figura2D = ( Graphics2D ) g;
```

É possível criar essas formas com tipos float também.

Usando a Java 2D API é fácil criar formas que são pintadas com cores que mudam gradualmente, obtendo efeitos mais complexos de uma forma bem simples, apenas preenchendo os argumentos dos métodos de cada forma. Esse efeito de mudança é feito através da classe `GradientPaint`, exemplo:

```
figura2D.setPaint( new GradientPaint( 5, 30, Color.BLUE, 35, 100, Color.YELLOW, true ) );
```

```
figura2D.fill( new Ellipse2D.Double( 5, 30, 65, 100 ) );
```

Nesse caso, usamos o gradiente azul-amarelo pra pintar a elipse.

Até agora foi falado sobre o uso de formas pré-estabelecidas no Java.

Porém, é possível criar suas próprias formas de figuras, através da classe `GeneralPath`, da package `java.awt.geom`.

Com ela, é possível, por exemplo, criar figuras através de vetores de inteiros, que representam pontos dos eixos x e y.

Para criar uma figura:

```
Graphics2D fig = ( Graphics2D ) g;
```

```
GeneralPath fig = new GeneralPath();
```

Podemos usar os seguintes métodos:

```
fig.moveTo( int, int ) ; fig.lineTo ( int [], int [] ) ; fig.closePath();
```

Onde colocamos a posição inicial, criamos ela e fechamos a figura.

Podemos usar um objeto da GeneralPath como argumento da fig2d:

```
fig2d.translate(int , int ) ; fig2d.rotate (float ) ; fig2d.setColor ( Color)
```

Onde podemos transladar a figura, rotacionar e pintá-la.

As figuras e formas são muitas, assim como os métodos para podermos fazer os mais diversos tipos de efeitos nos desenhos, por isso não cabe aqui mostrar todos os códigos e programas prontos.

Caso deseje algo específico, vá em busca da documentação.

E se precisar de ajudar, não hesite em perguntar no campo de Comentários ou fazendo Contato.

Programação Gráfica em Java, pt III: GUI - Graphic User Interface

JLabel - Como criar rótulos com textos e imagens em frames

Agora que aprendemos os conceitos básicos de JFrame e JPanel, bem como ensinamos como fazer alguns desenhos interessantes em Java, já estamos um pouco familiarizados com a programação gráfica em Java.

Agora vamos mostrar os principais componentes do pacote Swing.

Neste artigo de nossa apostila, vamos ensinar como colocar textos (JLabel) e imagens (Icon e ImageIcon).

JLabel - Exibindo rótulos (Textos e Imagens)

Como primeiro, mais básico e simples, componente de um aplicativo do tipo GUI que iremos ensinar aqui em nossa apostila de Java, os JLabels são rótulos que podemos exibir em nossos frames.

São elementos estáticos, não sendo usado para interagir com o usuário.

Bem comum de serem vistos quando vamos instalar algum aplicativo de alguma empresa famosa, onde é possível ver um JLabel com texto e imagem, do programa/corporação.

Ou seja, um rótulo em programação Java é o mesmo que no mundo real, um texto e/ou imagem.

Pense em um rótulo de automóvel ou loja, geralmente é um texto (nome) e uma imagem (slogan).

Os JLabels são classes derivadas da JComponent e fazem parte do pacote swing.

Logo, para usar, fazemos o import:

```
import javax.swing.JLabel;
```

Vamos colocar labels em frames, para fazermos testes.

Portanto, precisamos fazer o import do JFrame também:


```
import javax.swing.JFrame;
```

```
□
```

JLabel só com String

Vamos criar rótulos.

No primeiro exemplo, vamos criar um label que só exiba um simples texto.

Para fazer isso, vamos criar uma classe chamada "Rotulo", que vai herdar a classe JFrame.

Declaramos o JLabel da seguinte maneira:

```
private JLabel texto;
```

Como podemos ver na documentação do JLabel, podemos inicializar um JLabel passando só a String de texto, só a imagem, uma string e um texto, uma string e um inteiro, uma imagem e um inteiro ou podemos inicializar a JLabel passando tudo: uma imagem, um texto e um inteiro.

Esse inteiro é o alinhamento que o JLabel vai ter no JFrame.

Como esse exemplo queremos apenas fornecer uma String, inicializamos nosso JLabel dentro do construtor da classe "Rotulo":

```
texto = new JLabel("Meu primeiro JLabel!");
```

Criado o componente JLabel, vamos adicioná-lo ao nosso JFrame, através do método add:

```
add(texto);
```

Tudo isso dentro de nosso construtor, que também fornece uma String que será o título do JFrame:

```
super("Usando rótulos em JFrame");
```

Na nossa main, criamos um objeto do tipo Rotulo, o rotulo1, tornamos ele visível, fazemos com que ele feche quando clicamos no botão close e definimos um tamanho de nosso JFrame.

Assim, o código de nosso programa fica

GUI.java

```
import javax.swing.JFrame;
```

```
public class GUI {
```

```
    public static void main(String[] args) {
```

```
        Rotulo label1 = new Rotulo();
```

```
        label1.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```
        label1.setSize(300,100);
```

```
        label1.setVisible(true);
```

```
    }
```

```
}
```

Rotulo.java

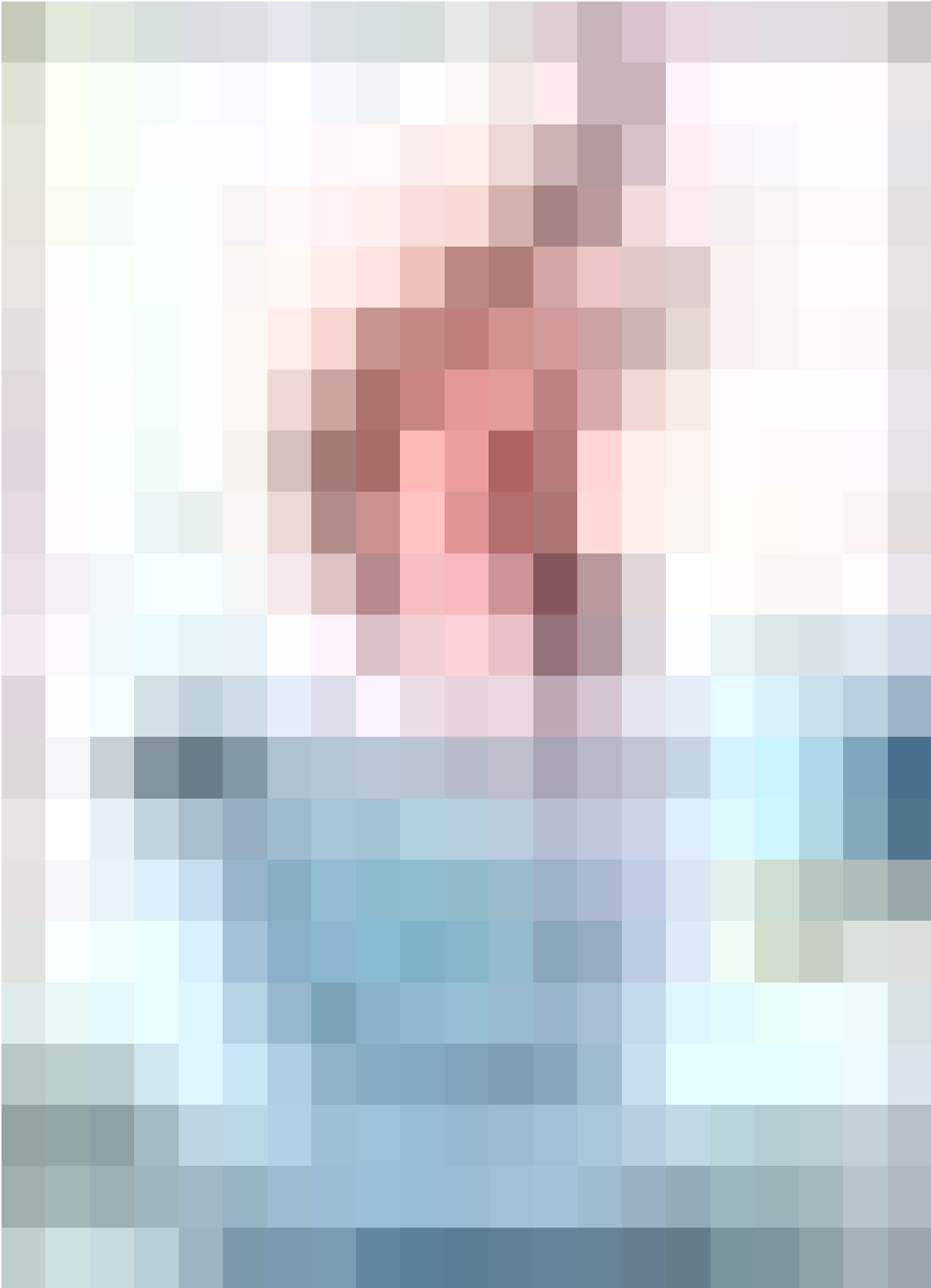
```
import javax.swing.JFrame;  
import javax.swing.JLabel;  
  
public class Rotulo extends JFrame{  
  
private JLabel texto;  
  
public Rotulo(){  
  
    super("Usando rótulos em JFrame");  
  
        texto = new JLabel("Meu primeiro JLabel!");  
        add(texto);  
    }  
}
```

JLabel só com imagem

Agora vamos aprender uma coisa bem bacana: colocar imagens de nosso computador em aplicativos gráficos em Java.

De início, vamos usar imagens simples, de rótulos, que são os ícones.

Vamos usar a seguinte imagem, que é o favicon do curso Java Progressivo:



Salve com o nome "xicara.png".

Agora uma coisa importante: coloque esta imagem dentro da pasta de seu projeto.

Ela deve ficar na pasta que tem as pastas 'bin' e 'src' de seu projeto.

Para manipular este tipo de imagem, precisamos usar as classe Icon e ImageIcon, do pacote swing:

```
import javax.swing.Icon;
```

```
import javax.swing.ImageIcon;
```

Vendo a documentação, percebemos que o JLabel recebe imagens que são objetos da classe Icon.

Vamos criar tal objeto, chamando-o de "xicara", e fazendo o polimorfismo, fazendo o objeto "xicara" ser um objeto específico da classe ImageIcon.

Como argumento, passamos a URL da imagem.

Como esta URL está na pasta do projeto, apenas colocamos o nome e extensão da imagem.

Assim, nosso objeto da classe Icon é declarado da seguinte maneira:

```
Icon xicara = new ImageIcon("xicara.png");
```

Tendo o objeto "xicara", passamos ele para o nosso JLabel "imagem":


```
imagem = new JLabel(xicara);
```

E adicionamos nosso JLabel ao nosso JFrame.

Assim, o código Java do aplicativo que exibe um rótulo composto de uma imagem de nosso HD é:

Rotulo.java

```
import javax.swing.JFrame;  
import javax.swing.JLabel;  
import javax.swing.Icon;  
import javax.swing.ImageIcon;  
  
public class Rotulo extends JFrame{  
  
private JLabel imagem;  
  
public Rotulo(){  
  
    super("Usando rótulos em JFrame");  
  
        Icon xicara = new ImageIcon("xicara.png");  
        imagem = new JLabel(xicara);  
        add(imagem);  
  
    }  
  
}
```

}

Note que não precisamos mudar em nada a main, pois ela só cria o JFrame, e para um JFrame um JLabel é um label, tendo imagem, texto ou ambos. Aliás, JLabel é um JComponent, assim como os JButtons, por exemplo.

São todos herdados da mesma superclasse, e se comportam da mesma maneira.

JLabel com imagem e texto

Agora vamos fazer os dois: criar um JLabel com String e imagem.

Porém, a coisa começa a ficar um pouco mais complicada devido ao local da String e da Imagem.

Para usar JLabel com Icon e String, precisamos usar o conceito de alinhamento.

Existem dois tipos: o horizontal e o vertical.

Todos essas posições são números inteiros, mas não precisamos nos preocupar com tais valores.

Para isso, vamos usar as constantes da SwingConstants:

```
import javax.swing.SwingConstants;
```

Na horizontal, existem as posições:

SwingConstants.LEFT, SwingConstants.CENTER e SwingConstants.RIGHT

Na vertical, existem as posições:

SwingConstants.TOP, SwingConstants.CENTER e SwingConstants.BOTTOM

O construtor do JLabel recebe um inteiro que representa o alinhamento horizontal.

Ou seja, se o Label vai ficar na esquerda, no centro ou na direita.

O outro tipo de alinhamento é o vertical, que vai definir de se o JLabel vai ficar em cima, no meio ou na parte de baixo do frame.

Para mudarmos o alinhamento vertical, usamos o método: `setVerticalAlignment`

Esse método recebe uma das três constantes: `SwingConstants.TOP`, `SwingConstants.CENTER` e `SwingConstants.BOTTOM`

Um outro conceito que devemos saber é o da posição da String e da Imagem.

Se você pensar, pode ser que queiramos a imagem em cima do texto, embaixo ou ao lado.

Para ter controle desse posicionamento, existem dois métodos:

`setHorizontalTextPosition` -> define a posição do texto, na horizontal (se vai ficar a esquerda ou direita da imagem)

`setVerticalTextPosition` -> define a posição do texto, na vertical (se o texto vai ficar abaixo ou acima da imagem)

Para definir o alinhamento horizontal no centro, podemos passar a constante já

no construtor:

```
label = new JLabel("Símbolo do Java", xicara, SwingConstants.CENTER);
```

Para que a posição do texto seja centralizada, fazemos:

```
label.setHorizontalTextPosition( SwingConstants.CENTER );
```

E finalmente, se eu quiser que o texto fique abaixo da imagem, fazemos:

```
label.setVerticalTextPosition(SwingConstants.BOTTOM);
```

Nosso código fica:

Rotulo.java

```
import javax.swing.JFrame;
```

```
import javax.swing.JLabel;
```

```
import javax.swing.Icon;
```

```
import javax.swing.ImageIcon;
```

```
import javax.swing.SwingConstants;
```

```
public class Rotulo extends JFrame{
```

```
private JLabel label;
```

```
public Rotulo(){
```

```
super("Usando rótulos em JFrame");
```

```
    Icon xicara = new ImageIcon("xicara.png");
```

```
    label=new JLabel("Símbolo do Java", xicara,  
SwingConstants.CENTER);
```

```
    label.setHorizontalTextPosition( SwingConstants.CENTER );
```

```
label.setVerticalTextPosition( SwingConstants.BOTTOM );
```

```
add(label);
```

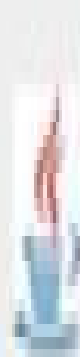
```
}
```

```
}
```

E o resultado é o esperado:



Usando rótulos em JFrame



Símbolo do java

JButton - Como criar botões em aplicativos Java

No tutorial passado de nossa apostila de Java, ensinamos como criar um JLabel para exibir textos e imagens em uma GUI (aplicação de interface gráfica do usuário) e vimos que nossos programas estão ficando cada vez mais agradáveis graficamente.

Neste tutorial vamos falar sobre um dos elementos mais importantes e usados em aplicativos GUI: os botões! Que são criados através do componente JButton.

JButton - O que são e para que servem os botões

Explicar o que é um botão é uma tarefa praticamente redundante e óbvia, afinal sempre estamos contato com botões nos mais diversos tipos aplicativos gráficos.

Botões nada mais são que uma região gráfica (geralmente um retângulo) que ao ser clicado (ou selecionado pelo teclado) disparam um evento. É como se tivéssemos que dar um enter quando estamos no terminal de comando.

Aliás, tudo é possível de ser feito através do terminal de comando e programação de baixo nível (de fato, é assim que as coisas acontecem). Os componentes gráficos vem para substituir esses comandos e linguagem mais técnica, para que o usuário leigo possa fazer coisas em sua máquina de maneira simples, sem precisar ser 'expert' em computação.

Em breve falaremos em detalhes sobre esses eventos que ocorrem quando apertamos um botão, e as ações que são disparadas quando fazemos isso, pois iremos escolher que ações vão ocorrer quando um botão for clicado, por exemplo.

JButton - Como usar botões em Java

Se parar para pensar, todo botão possui: um texto, ou uma imagem ou os dois.

Afinal, não faz sentido ter um botão que você não sabe o que é ou para que serve.

É exatamente por esse motivo que ensinamos a usar JLabel no tutorial passado de nosso curso de Java, portanto iremos partir do pressuposto que você estudou o artigo e já sabe de todos os paranauês dos labels (vamos usar a parte das imagens, inclusive a figura da xicara.png).

Inicialmente vamos criar uma classe chamada "Botao", que vai estender a JFrame e onde vamos criar nossos JButtons, assim como fizemos com JLabels.

Como o JButton é um componente da JComponent, temos que importar sua classe do pacote swing:

```
import javax.swing.JButton;
```

Para usar o botão, apenas instanciamos um objeto da classe JButton.

Vamos criar o botão 'ok':

```
private JButton ok;
```

Vamos inicializar esse botão com uma string, que será o texto exibido no botão:

```
ok = new JButton("OK");
```

Depois apenas usamos o método add para adicionar o objeto 'ok'.

Nossa classe principal e a Botao ficam:

GUI.java

```
import javax.swing.JFrame;
```

```
public class GUI {
```

```
public static void main(String[] args) {
```

```
    Botao botao1 = new Botao();
```

```
    botao1.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```
    botao1.setSize(350,80);
```

```
    botao1.setVisible(true);
```

```
}
```

```
}
```

Botao.java

```
import javax.swing.JFrame;
import javax.swing.JButton;

public class Botao extends JFrame{
    private JButton ok;

    public Botao(){

        super("Criando botões");

        ok = new JButton("OK");
        add(ok);
    }

}
```

Rodando o código você verá uma coisa peculiar: o botão ocupa a tela inteira.

Mas orgulhe-se, criou seu primeiro botão :)

Layout e organização de uma aplicação GUI

Você já parou para perguntar o que acontece quando usamos o método add, da JFrame ?

É sabido que serve para adicionar componentes gráficas. Mas como? Onde elas são adicionadas?

Se eu adicionar um JLabel, um JButton e um campo de texto, como vai ficar a disposição desses elementos?

Se eu quiser colocar uma componente em um lado, outro na outra, reduzir o tamanho de um botão e uma série de outras coisas?

Pois é, a disposição desses elementos em seu aplicativo gráfico é o que chamamos de layout, é uma espécie de organização dos elementos, para deixá-los bonitos e intuitivos de serem usados.

Existem diversas classes e métodos que servem para nos auxiliar na organização dos componentes que iremos adicionar em um JFrame, existindo até mesmo recursos no NetBeans que permitem que vocês coloquem botões, listas, imagens e tudo que possível em um JFrame apenas usando o mouse e definindo a posição de todos os componentes, de uma maneira bem fácil e simples.

Mais à frente iremos nos dedicar aos estudos dos layouts e containers, e por hora vamos usar um manager que vai organizar os elementos de um frame à medida

que formos adicionando-os, é o `FlowLayout`, que vai adicionando os `JComponents` ao lado do outro, e quando não couberam mais, ele bota na linha debaixo.

Para fazer uso desse layout manager, importe:

```
import java.awt.FlowLayout;
```

E no construtor, use o método:

```
setLayout( new FlowLayout() );
```

Vamos criar outro botão, chamado "cancelar", e fazer uso do `FlowLayout`:

Botao.java

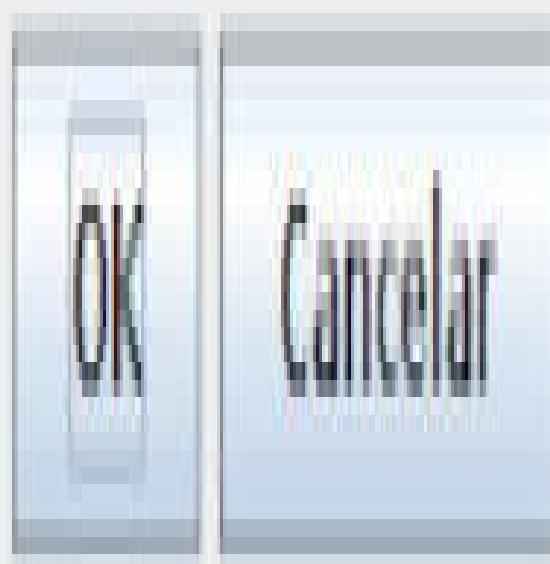
```
import javax.swing.JFrame;  
import javax.swing.JButton;  
import java.awt.FlowLayout;  
  
public class Botao extends JFrame{  
  
private JButton ok,  
cancelar;  
  
public Botao(){  
  
super("Criando botões");  
setLayout( new FlowLayout() );  
  
ok = new JButton("OK");  
cancelar = new JButton("Cancelar");  
add(ok);  
add(cancelar);  
  
}
```

}

Veja como agora os botões possuem tamanho bem mais interessante e estão dispostos um ao lado do outro, como é costume vermos por aí:



Criando botões - Java Progressivo



Imagens em JButton

Como havíamos dito, há outra maneira de inicializar um JButton, que é colocando objetos da classe Icon, como fizemos com JLabel.

Vamos criar o objeto 'xicara', com a mesma figura que usamos no artigo passado:

```
Icon xicara = new ImageIcon("xicara.png");
```

E vamos inicializar o botão 'ok' com uma String e a Imagem:

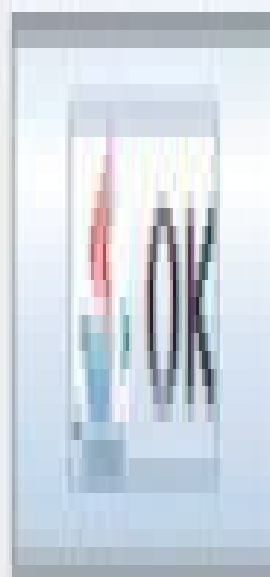
```
ok = new JButton("OK", xicara);
```

E pronto. Se você rodar seu programa, verá que agora existe a pequena imagem de uma xícara ao lado do 'OK' do botão.

Veja o resultado:



Criando botões - Java Progressivo



Simples, não?

O que um JButton faz - Eventos e Ações

Ok, confesso que a primeira vez que criei um botão meus olhos ficaram cheios de lágrimas.

Depois de muito tempo estudando C e Java só pelo terminal de comando, ver um botão desses criados pelos nossos códigos é bem interessante e animador.

Mas o que fazem esses botões? Experimentou clicar?

Como deve saber, usamos botões para confirmar um comando.

Após apertar um botão, podemos fechar o aplicativo, ir para outra janela do frame, limpar um formulário, enviar os dados de um formulário que acabamos de preencher, damos início a um jogo ao apertar um botão, e infinitas outras coisas são feitas com botões.

Então, o que os nosso botões vão fazer?

Simples: o que você programar eles para fazerem.

Se nossos botões não fizeram nada, é porque simplesmente não os programamos para fazer nada, afinal eles não iriam adivinhar o que queríamos que acontecesse quando apertássemos.

No próximo tutorial de nossa apostila Java Progressivo iremos estudos este tipo

de evento: de apertar um botão. Iremos definir através de programação a ação que um programa deve tomar após ter acontecido esse evento (o clique).

Lá, continuaremos a usar os JButtons para ilustrar importantes conceitos na programação de interfaces gráficas para o usuário (GUI), que são os Events, Event Handling, Action Listeners etc.

Ou seja, continuaremos a criar e ensinar mais coisas sobre os JButtons em Java, porém vamos fazer com que eles se comportem como botões de verdade, que disparem algum processo, uma ação, sempre que ocorrer o evento do click.

Nos vemos lá!

Tratando evento e ações em GUI - Event Handling, ActionListener, ActionEvent e actionPerformed

No artigo passado de nossa apostila de Java, ensinamos como criar botões em Java através da classe JButton, e vimos que nossos aplicativos gráficos estão começando a ter um aspecto mais agradável e estão se tornando cada vez mais profissionais.

Neste tutorial de Java vamos ensinar como tratar eventos e ações, e a partir de agora nossas ações (como clicar, apertar alguma tecla etc) na GUI irão dar disparar qualquer funcionalidade que queiramos.

GUI - Controlando eventos e ações

Se você parar para reparar, um aplicativo gráfico é como se fosse como um cardápio de um restaurante. Você chega, faz seus pedidos e suas ordens são atendidas.

Cada pessoa vai fazer pedidos diferentes, tanto dos pratos como do número de coisas ordenadas.

Da mesma maneira é uma aplicação gráfica em Java.

Um programa nada mais é que uma série de opções, onde cada usuário vai fazer o que quiser, clicando em botões, apertando teclas do teclado, rolando a barra de informações, marcando, selecionando, escrevendo, minimizando, fechando e uma infinidade de possibilidades.

Cada vez que o usuário faz uma destas coisas, dizemos que foi realizado um evento.

Ou seja, um click, o mouse passou em alguma região e algo mudou, ele escreveu algo, deu enter etc etc.

O que a GUI (Graphic User Interface) faz é nada mais que tratar estes eventos.

Se ele apertar isso, acontece aquele, Se digitar isso, aquilo abre.

Se clicar aqui, aquilo vai fechar. Se apertar enter, vai pra próxima janela etc etc.

Ou seja, um aplicativo gráfico é uma maneira do usuário realizar pedidos e comandos de uma maneira bem mais simples e intuitiva.

O usuário realiza o evento, e uma ação ocorre.

E é isso que vamos aprender neste tutorial de Java de nossa apostila. Vamos aprender a identificar os eventos e realizar as ações que queiramos que aconteça.

□

Como tratar eventos - A interface ActionListener e o método actionPerformed

Existe uma classe em Java que será a responsável pelo tratamento de eventos.

Ou seja, é nela que vamos identificar o evento que ocorreu e é nela que vamos definir que ações nossos aplicativos devem executar quando tal evento ocorrer.

A classe ActionListener é uma interface, ou seja, é classe composta apenas de métodos abstratos.

E isso é até óbvio, pois a classe não tem como saber que tipos de eventos vamos tratar em um aplicativo, muito menos vai saber que tipo ação queremos que nosso software tome quando um evento ocorrer.

Listener pode ser traduzido como 'ouvinte'.

E isso faz sentido, pois essa interface é que vai ficar 'esperando' algo ocorrer. É como se ela ficasse em um loop infinito, o tempo todo testando:

"Ocorreu algo agora? E agora? Foi um evento? É um evento? E agora? E agora?..."

Ou seja, a interface fica na 'escuta', na espreita até o usuário fazer alguma interação com o aplicativo e um evento ocorrer.

A interface ActionListener possui somente um método. Então temos apenas um método obrigatório para implementar, que é o método actionPerformed, e ele é o responsável por tomar alguma ação caso algum evento ocorra.

Assim, a interface fica na espera de algum evento, e caso ocorra ele é imediatamente passado para o método `actionPerformed`. É dentro deste método que iremos definir as ações que ocorrem.

Para fazer uso dessas funcionalidades, devemos importar:

```
import java.awt.event.ActionEvent;
```

```
import java.awt.event.ActionListener;
```

Event Handling - Criando um tratador de eventos

Agora que já temos noção do que é a interface `ActionListener` e seu método `actionPerformed`, vamos aprender como, de fato, tratar um evento.

O tratamento é feito por um objeto da classe `ActionListener`. Há várias maneiras de se criar esse objeto na prática, vamos aprender mais no próximo tutorial, mas neste exemplo vamos fazer de uma maneira bem didática.

Como já estudamos `JButton`, vamos fazer um aplicativo que irá mostrar uma caixa de diálogo quando clicamos no botão ok ou no botão cancelar.

Para iniciar, vamos criar nossa classe que irá implementar a `ActionListener`, vamos chamar de `"ButtonHandler"`, (handle é manusear).

Assim, o esqueleto de nosso tratador é:

```
import java.awt.event.ActionEvent;  
import java.awt.event.ActionListener;  
  
public class ButtonHandler implements ActionListener {  
  
public void actionPerformed(ActionEvent evento) {
```

```
}  
  
}
```

Note que o método `actionPerformed` recebe um argumento 'evento' do tipo `ActionEvent`.

Este argumento 'evento' vai armazenar a natureza do evento, ele sabe especifica e exatamente que componente ocorreu o evento.

Tal componente está armazenado no método `getSource()` deste objeto.

No nosso exemplo, vamos criar dois botões chamados "ok" e "cancela".

Esses botões foram declarados na classe "Botao.java", que usamos no exemplo do artigo passado.

Assim, quando formos criar o tratador (handler) pros botões, precisamos passar estes dois botões para nossa classe "ButtonHandler", e esta classe recebe eles por meio do construtor.

Para saber que botão foi clicado, basta fazer testes condicionais para saber o que está guardado em:

```
evento.getSource()
```


Dependendo do botão que foi clicado, iremos exibir uma mensagem.

Uma dizendo que o "OK" foi clicado, e o outro exibe a mensagem dizendo que o botão "CANCELAR" foi pressionado.

As mensagens serão exibidas através das Dialog Boxes(Caixas de diálogo).

Assim, nossa classe "ButtonHandler" que irá tratar os eventos dos botões é:

ButtonHandler.java

```
import java.awt.event.ActionEvent;  
import java.awt.event.ActionListener;  
  
import javax.swing.JButton;  
import javax.swing.JOptionPane;  
  
public class ButtonHandler implements ActionListener {  
  
private JButton ok, cancela;  
  
public ButtonHandler(JButton ok, JButton cancela){  
  
    this.ok = ok;  
  
    this.cancela = cancela;  
  
    }
```

```
public void actionPerformed(ActionEvent evento) {
```

```
    if(evento.getSource() == ok)
```

```
        JOptionPane.showMessageDialog(null, "O botão OK foi  
        clicado");
```

```
    if(evento.getSource() == cancela)
```

```
        JOptionPane.showMessageDialog(null, "O botão CANCELA foi  
        clicado");
```

```
    }
```

```
}
```

O método addActionListener - Adicionando um tratador de eventos aos componentes

Pronto, nosso tratador de eventos dos botões, ou handler, foi construído, que é nossa classe "ButtonHandler". Vamos criar um objeto dessa classe, e chamar de "handler":

```
ButtonHandler handler = new JButton(ok,cancela);
```

Pronto, agora temos um tratador de botões do tipo 'ok' e 'cancela', que é o objeto handler.

Porém, podem existir vários componentes, e para cada um deles há a possibilidade de existir um tratador de evento diferente.

Clicar em um botão é diferente de escrever um texto numa caixa de texto. Numa você digita, e na outra componente você clica.

Assim, para cada componente que vamos criar temos que definir que tratador de eventos aquele objeto vai usar.

Isso é definido passando um objeto do tipo ActionListener para o método addActionListener, existente nos componentes.

Como queremos tratar apenas os botões 'ok' e 'cancela', e o tratador deles é o mesmo, fazemos:

```
ok.addActionListener(handler);  
cancela.addActionListener(handler);
```

O código de nossa classe "Botao", que é um JFrame com dois JButtons fica:

Botao.java

```
import java.awt.FlowLayout;  
import javax.swing.JFrame;  
import javax.swing.JButton;  
  
public class Botao extends JFrame{  
  
private JButton ok = new JButton("OK");  
  
private JButton cancela = new JButton("Cancela");  
  
private ButtonHandler handler;  
  
public Botao(){  
  
    super("Criando botões");  
  
        setLayout(new FlowLayout());  
  
        handler=new ButtonHandler(ok, cancela);  
  
}
```

```
        ok.addActionListener(handler);  
        add(ok);  
  
        cancela.addActionListener(handler);  
        add(cancela);  
    }  
  
}
```

E a classe principal é:

Main.java

```
import javax.swing.JFrame;

public class Main {

    public static void main(String[] args) {

        Botao botoes = new Botao();

        botoes.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        botoes.setSize(350,70);

        botoes.setVisible(true);

    }

}
```

Se rodarmos o projeto, teremos os seguintes resultados sempre que clicarmos em "OK" ou "CANCELAR":





Tratamento de eventos - Extends e Implements, Classe Interna e Objeto anônimo - Como mudar a cor de um JFrame

No tutorial passado de nossa apostila de Java, demos introdução ao tratamento de eventos e ações em GUI. Explicamos os conceitos de ActionListener,(ActionEvent, actionPerformed e Event Handling, de um modo geral.

Lá mostramos uma maneira de criar um tratador de eventos, onde criamos uma classe que implementa a classe abstrata ActionListener.



Neste artigo iremos mostrar outras maneiras de criar esse tratador de eventos.

Vamos fazer isso através de um aplicativo que cria botões com JButton que fazem com que o fundo (background) de nosso programa mude de cor quando clicamos nesses botões.

Estendendo o JFrame e implementando a ActionListener

Se você notar bem, fizemos basicamente duas coisas no artigo passado de nossa apostila:

criamos uma classe chamada "Botao" que estende, ou seja, é um JFrame

criamos a classe "ButtonHandler", que implementa a interface "ActionListener"

Essas duas classes que tratam os eventos e ações, e exibem o resultado.

Temos uma desvantagem nesse método que é que precisamos passar para a classe "ButtonHandler" os JButtons que criamos em nosso JFrame.

Isso foi feito através da composição, pois passamos objetos de uma classe para outra.

Tal ação foi necessária pois o tratador de eventos, ButtonHandler, tenha noção da existência desses botões que estão em outra classe, para que possa tomar uma atitude sempre que esses botões forem clicados.

Agora imagine uma aplicação real, com dezenas de botões, menus, campos de texto, campos pra selecionar etc etc etc de opções em GUI (Graphic User Interface), seria extremamente incômodo ficar passando essas informações sobre os elementos, através de composição.

Por isso, vamos usar um método especial, que não vai mais ser necessário a criação da classe "ButtonHandler", somente usaremos nossa "Main" e "Botao".

O segredo da coisa é: fazer com essa classe "Botao" seja um JFrame e ao mesmo tempo implemente a interface ActionListener.

Isso é feito da seguinte maneira:

public class Botao extends JFrame implements ActionListener

Porém, como sabemos, devemos obrigatoriamente implementar os métodos abstratos de uma interface.

Felizmente, a interface `ActionListener` só tem um método, o `actionPerformed`.

Então, basicamente só temos que colocar esse método na classe "Botao", que será o responsável por tratar os eventos. Como os botões estão na mesma classe, não é necessário passar objetos de uma classe pra outra.

□

Alterando a cor do background (fundo) de um JFrame e o método repaint()

Para mostrar um exemplo deste método de implementação de um tratamento de eventos, vamos mostrar um aplicativo que vai exibir três botões: "verde", "azul" e "amarelo", e ao clicarmos nele a cor de fundo da janela (o background do JFrame) vai mudar de acordo com o botão que criamos.

Quando criamos os botões no exemplo do artigo passado, tínhamos que criar um objeto do tipo "ButtonHandler", pois esta era a classe que tinha o método actionPerformed, e passávamos esse objeto como argumento para o método addActionListener de cada JButton que criávamos.

Mas agora não temos mais essa classe, pois a "Botao.java" implementa essa interface.

Como passamos o objeto para o método addActionListener, então?

Simples, passamos o próprio método. Isso é feito simplesmente escrevendo this.

Já para alterarmos a cor do fundo do background do JFrame, usamos o método:

```
getContentPane().setBackground(Color c)
```

Que recebe um objeto da classe "Cor" como argumento (Color.GREEN, Color.DARK, Color.WHITE etc)

Outro método importantíssimo para podermos alterar o conteúdo de um JFrame durante uma aplicação é o método `repaint()`.

Sua funcionalidade é simples: ele desenha de novo o JFrame na tela.

Isso é necessário pois ao clicarmos em um dos botões, ele vai alterar a característica do JFrame (a cor do fundo). Porém, precisamos avisar ao nosso aplicativo Java para que ele desenhe novamente a GUI para que o usuário veja que algo foi alterado.

Logo, após termos detectado que botão foi clicado usando o método `getSource()` e alterado a cor de fundo através do método `getContentPane().setBackground()`, nós usamos o método `repaint()` para redesenhar nossa JFrame.

Assim, o código de nosso programa é simplesmente:

Main.java

```
import javax.swing.JFrame;
```

```
public class Main {
```

```
public static void main(String[] args) {
```

```
    Botao botoes = new Botao();
```

```
    botoes.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```
    botoes.setSize(400,400);
```

```
    botoes.setVisible(true);
```

```
}
```

```
}
```

Botao.java

```
import java.awt.Color;
```

```
import java.awt.FlowLayout;
```

```
import java.awt.event.ActionEvent;
```

```
import java.awt.event.ActionListener;
```

```
import javax.swing.JFrame;
```

```
import javax.swing.JButton;
```

```
public class Botao extends JFrame implements ActionListener{
```

```
private JButton verde = new JButton("Verde"),
```

```
        azul = new JButton("Azul"),
```

```
        amarelo = new JButton("Amarelo");
```

```
public Botao(){
```

```
    super("Alterando a cor de Background");
```

```
setLayout(new FlowLayout());
```

```
verde.addActionListener(this);
```

```
add(verde);
```

```
azul.addActionListener(this);
```

```
add(azul);
```

```
amarelo.addActionListener(this);
```

```
add(amarelo);
```

```
}
```

```
public void actionPerformed(ActionEvent evento) {
```

```
if(evento.getSource() == verde)
```

```
this.getContentPane().setBackground(Color.GREEN);
```

```
if(evento.getSource() == azul)
```

```
this.getContentPane().setBackground(Color.BLUE);
```

```
if(evento.getSource() == amarelo)
```

```
    this.getContentPane().setBackground(Color.YELLOW);
```

```
        repaint();
```

```
    }
```

```
}
```

Notaram como ficou mais simples e enxuto nosso programa?

Classe Interna em Java

Outra maneira de tratarmos eventos de uma GUI é através das classes internas.

Classe interna é um conceito bem útil e interessante que o Java nos fornece.

Como o próprio nome pode sugerir, classe interna é uma classe que é declarada DENTRO de outra classe! Por exemplo:

```
public class Botao Externa{
```

```
private class Interna{
```

```
    }
```

```
}
```

Vamos usar a classe interna para tratar nossos eventos, logo, esta classe deve implementar a interface `ActionListener`. E como uma interface é uma classe com todos os métodos abstratos, precisamos implementar sempre todos os métodos, que no caso desta classe é apenas o nosso conhecido método `actionPerformed()`.

Vamos simplesmente declarar uma classe interna, chamada `ButtonHandler`, que vai implementar a `ActionListener` e tratar nossos eventos, que são os cliques em nos botões de cores, que simplesmente mudam a cor de nosso `JFrame`.

Uma importante vantagem da classe interna, é que, por ela estar dentro de outra, ela vai ter acesso aos métodos e variáveis desta classe externa. Assim, nossa classe `ButtonHandler` tratadora de eventos, vai ter acesso aos métodos, `JButtons`, variáveis, `JFrame` etc, da classe externa.

Veja como fica o código de nosso mesmo programa, agora usando a técnica da classe interna:

Main.java

```
import javax.swing.JFrame;
```

```
public class Main {
```

```
    public static void main(String[] args) {
```

```
        Botao botoes = new Botao();
```

```
        botoes.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```
        botoes.setSize(400,400);
```

```
        botoes.setVisible(true);
```

```
    }
```

```
}
```

Botao.java

```
import java.awt.Color;

import java.awt.FlowLayout;

import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;


import javax.swing.JFrame;

import javax.swing.JButton;


public class Botao extends JFrame{


private JButton verde = new JButton("Verde"),
           azul = new JButton("Azul"),
           amarelo = new JButton("Amarelo");


private ButtonHandler handler = new ButtonHandler();


public Botao(){
```

```
super("Alterando a cor de Background");
```

```
    setLayout(new FlowLayout());
```

```
    verde.addActionListener(handler);
```

```
    add(verde);
```

```
    azul.addActionListener(handler);
```

```
    add(azul);
```

```
    amarelo.addActionListener(handler);
```

```
    add(amarelo);
```

```
}
```

```
private class ButtonHandler implements ActionListener{
```

```
public void actionPerformed(ActionEvent evento) {
```

```
    if(evento.getSource() == verde)
```

```
        getContentPane().setBackground(Color.GREEN);
```

```
if(evento.getSource() == azul)
```

```
    getContentPane().setBackground(Color.BLUE);
```

```
if(evento.getSource() == amarelo)
```

```
    getContentPane().setBackground(Color.YELLOW);
```

```
        repaint();
```

```
    }
```

```
}
```

```
}
```

Objeto anônimo

Por fim, a apostila Java Progressivo vai ensinar uma última maneira na qual você pode usar para trabalhar com tratamento de eventos, que é a do Objeto anônimo (ou classe).

Mas antes, para entender o que é isso, vamos relembrar um pouco alguns conceitos de Orientação a Objetos em Java.

Para que possamos usar o método `addActionListener` em nossos `JComponents`, como `JButton`, precisamos passar um objeto como argumento. Na verdade, precisamos passar uma referência, um endereço de memória para que o método vá lá e utilize daquele objeto para tratar seus eventos.

Geralmente declaramos nossos objetos assim: `NomeDaClasse nomeDoObjeto;`

E para atribuir um endereço de memória à ele, fazemos: `nomeDoObjetos = new NomeDaClasse();`

É essa palavra especial `new` que irá pegar um espaço na memória para o objeto.

Ou seja, o `new` retorna uma referência, que é tudo que nosso método `addActionListener` precisa.

Assim, nem sempre é necessário declarar um objeto, dar um nome a ele, e usar o `new`.

Nos exemplos passados, quando passamos o objeto handler, podemos substituir isso por "new ButtonHandler()" que teremos exatamente o mesmo efeito, pois ambos casos passar uma referência de um objeto da classe ButtonHandler() para o método addActionListener.

Isso se chama objeto anônimo, pois ele não tem nome, simplesmente usamos o new.

Então, em vez de passar o nome de um objeto, vamos passar somente a referência:

```
new ActionListener()
```

Mas vamos fazer outra coisa interessante e essencial: vamos implementar a classe ActionListener e seu método actionPerformed() ali mesmo, na hora de passar o argumento.

Faremos algo do tipo:

```
verde.addActionListener( new ActionListener() { implementação} );
```

Para ficar mais legível, geralmente se faz:

```
verde.addActionListener( new ActionListener() {  
    implementação  
}  
);
```

Veja como fica nosso programa, usando artifícios de Objeto/Classe Anônimo(a):

Main.java

```
import javax.swing.JFrame;
```

```
public class Main {
```

```
    public static void main(String[] args) {
```

```
        Botao botoes = new Botao();
```

```
        botoes.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```
        botoes.setSize(400,400);
```

```
        botoes.setVisible(true);
```

```
    }
```

```
}
```

Botao.java

```
import java.awt.Color;  
import java.awt.FlowLayout;  
import java.awt.event.ActionEvent;  
import java.awt.event.ActionListener;  
  
import javax.swing.JFrame;  
import javax.swing.JButton;  
  
public class Botao extends JFrame{  
  
private JButton verde = new JButton("Verde"),  
        azul = new JButton("Azul"),  
        amarelo = new JButton("Amarelo");  
  
public Botao(){  
  
    super("Alterando a cor de Background");
```

```
setLayout(new FlowLayout());
```

```
verde.addActionListener( new ActionListener() {
```

```
public void actionPerformed(ActionEvent evento) {
```

```
    getContentPane().setBackground(Color.GREEN);
```

```
    repaint();
```

```
    }
```

```
}
```

```
);
```

```
add(verde);
```

```
azul.addActionListener(new ActionListener() {
```

```
public void actionPerformed(ActionEvent evento) {
```

```
    getContentPane().setBackground(Color.BLUE);
```

```
    repaint();
```

```
    }
```

```
}
```

```
);
```

```
add(azul);
```

```
amarelo.addActionListener(new ActionListener() {  
  
    public void actionPerformed(ActionEvent evento) {  
  
        getContentPane().setBackground(Color.YELLOW);  
        repaint();  
    }  
}  
);  
add(amarelo);  
}  
  
}
```

Afinal, qual a melhor maneira de tratar eventos em Java?

Vimos diversas maneiras de se tratar eventos em Java.

Primeiro, no tutorial passado de nossa apostila, fizemos o tratamento de eventos criando uma classe externa (como fazemos normalmente) e passando para ela, através de composição, os JButtons como objetos.

Já neste tutorial, fizemos a própria classe que cria e cuida dos Botões ser JFrame e ActionListener, e criamos o método actionPerformed na própria classe.

Em seguida, fizemos uma classe interna e por fim usamos a técnica da classe anônima.

E aí, qual técnica usar? Como programar para tratar eventos?

Bom, antes de tudo, é importante frisar que o bom é ter opções.

Como mostramos, todas quatro maneiras funcionam da mesma forma, o que vai mudar é a eficiência, organização e tempo que vai levar para cada implementação.

O último método, sobre objeto anônimo, deve ser usado somente quando a implementação do tratamento de eventos for coisa simples, como um ou outro botão que dispara um evento único.

Isso porque temos que implementar tudo em cada componente, e se tiver vários botões por exemplo, seu código ficaria enorme e extremamente repetitivo.

Portanto, uma dica é usar objeto anônimo em tratamento simples e pequenos de

eventos.

As duas maneiras iniciais deste tutorial são as mais usadas, principalmente pela organização.

Se você tiver uma classe não muito grande, como a nossa que simplesmente trata de 3 botões, pode ser uma boa fazer essa class implementar a interface ActionListener diretamente, pois apenas temos que adicionar um método.

Porém, se sua classe for grande e complexa, é bom criar outra classe, uma classe interna para tratar os eventos. Fica mais organizada e terá a vantagem desta classe interna ter acesso aos elementos da classe externa.

O interessante é que você pode criar diversas classes para tratar os mais diversos tipos de evento.

No momento apenas trabalhamos com botões, mas em breve iremos trabalhar com lists, checkbox, menus, campos de texto, e aí vai ser necessário criar tratamentos de eventos diferentes para cada um destes tipos de JComponents, e ter uma classe para tratar cada componente é realmente uma boa estratégia.

Já a primeira maneira, que mostramos no tutorial passado de nossa apostila, é realmente a mais trabalhosa, pois temos que ficar enviando e recebendo objetos entre classes (Composição), além de ter que criar outra classe. Este método é realmente pouco usado, e fizemos apenas por questões didáticas.

JTextField e JPasswordField - Como usar caixas de texto e de senha em Java

Login em Java

Usuário: java

Senha:

Entrar

Limpar

Dando continuidade a nossa seção de Interfaces Gráficas do Usuário (GUI), vamos ensinar o que é, para que serve e como usar as caixas de textos e de senha, as `JTextField` e `JPasswordField`, que nos possibilitarão a comunicação entre nosso aplicativo Java e a entrada do usuário pelo teclado.

Ao final deste tutorial de Java, iremos criar um exemplo mostrando uma aplicação simples de Login e Senha.

JTextField - Caixas de Texto

Qual a primeira coisa que você faz após ligar seu computador?

Qual a primeira ação, interação que você tem com sua máquina?

Provavelmente é preencher o login e senha para logar em seu sistema operacional.

Deve fazer o mesmo para acessar seu e-mail ou alguma rede social.

Ao fazer uma compra na internet, você precisa preencher todos os seus dados, como nome, data de nascimento, número do cartão de crédito etc.

Não sei se reparou, mas a todo instante estamos tendo contato com caixas de texto.

Ou seja, escrevemos algo e o que digitamos é enviado para algum lugar, é usado em algum algoritmo para testar, validar, armazenar ou checar o que digitamos.

Tudo isso é feito com caixas de texto, que é o foco deste nosso tutorial de nossa apostila de Java.

Ou seja, nem precisamos entrar em detalhes o quão importante estas caixas são, portanto devemos aprender a criar tal recurso em programação Java.

Isso é feito através da componente JTextField.

JTextField - Como criar e usar uma caixa de texto em Java

Assim como outros JComponents, precisamos primeiramente importar a classe responsável por cada componente. No caso das caixas de texto, devemos importar:

```
import javax.swing.JTextField;
```

Vamos declarar isso em uma classe chamada "CaixaDeTexto", que irá estender um JFrame.

Inicialmente, vamos usar 3 componentes. Duas caixas de texto: "fixo" e "caixa" ; e uma JButton, o "exibe".

Vale ressaltar que sempre que inicializamos uma caixa de texto, ela terá automaticamente uma string, que é exibida na caixa de texto.

Devemos aprender, inicialmente, três importantes métodos de um JTextField:

setTexto(String str) - Faz a caixa de texto armazenar a string "str"

getTexto() - Retorna a string de um JTextField

setEditable(boolean b) - Há duas opções de caixas de texto, as editáveis (onde é possível escrevermos algo) e as não editáveis (caixas que possuem um texto fixo, que não é possível alterar, como as conhecidas "Termos e Regulamentos" que sempre aceitamos sem ler :)

Exemplo sobre JTextField

Crie um programa que exiba duas caixas de texto, uma editável e outra não editável, e dois botões, um de "Exibe" que mostra o texto digitado em uma caixa de diálogo e um botão de "Limpar", que apaga o conteúdo da caixa de texto editável.

Vamos inicializar a caixa de texto não editável com a string "Não é possível alterar essa caixa de texto".

Já a JTextField editável, vamos iniciar com o número 20, que representa o número de caracteres que a string dessa caixa de texto armazena.

Os dois botões são inicializados com seus nomes "Exibir" e "Limpar".

Agora vamos tratar os eventos dos botões (não precisamos tratar as JTextField's, pois ao passo que digitamos algo na caixa de texto, a sua string já é automaticamente mudada e atualizada).

Vamos usar objetos anônimos, visto que são JButtons simples, onde cada um faz uma coisa específica:

O botão "Exibir" simplesmente exibe uma JOptionPane com a string da caixa de texto, que está armazenada em: `caixa.getText()`

E o botão "Limpar" simplesmente coloca uma string vazia na caixa de texto. Isso é feito pelo método:

```
caixa.setText("");
```

E pronto! Veja o código e teste:

Main.java

```
import javax.swing.JFrame;

public class Main {

    public static void main(String[] args) {

        CaixaDeTexto texfield = new CaixaDeTexto();

        texfield.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        texfield.setSize(310,110);

        texfield.setVisible(true);

    }

}
```

CaixaDeTexto.java

```
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JTextField;
import java.awt.FlowLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JOptionPane;

public class CaixaDeTexto extends JFrame{

    private JTextField fixa, caixa;

    private JButton exhibe, limpa;

    public CaixaDeTexto(){

        super("JTextField - Caixas de texto");
```



```
setLayout(new FlowLayout());
```

```
fixa = new JTextField("Não é possível alterar essa caixa de texto");
```

```
fixa.setEditable(false);
```

```
add(fixa);
```

```
caixa=new JTextField(20);
```

```
add(caixa);
```

```
exibe = new JButton("Exibir");
```

```
exibe.addActionListener(new ActionListener() {
```

```
public void actionPerformed(ActionEvent evento){
```

```
if(evento.getSource() == exib
```

```
caixa.getText());          JOptionPane.showMessageDialog(null,
```

```
    }
```

```
}
```

```
);
```

```
add(exibe);
```

```
        limpa = new JButton("Limpar");

        limpa.addActionListener(new ActionListener() {

public void actionPerformed(ActionEvent evento){

        if(evento.getSource() == limpa)

                caixa.setText("");

        }

        });

        add(limpa);

    }

}
```

JPasswordField - Como criar caixas de senha

Se você aprendeu bem como se usar o JTextField, já entendeu como funciona a JPasswordField, pois a única diferença é que este JComponent exibe asteriscos no lugar dos caracteres que o usuário escreve.

Você não vê a string digitada, mas ela está lá, presente no JPasswordField.

Embora não possamos ver, podemos trabalhar normalmente com a string.

Por exemplo, podemos comparar duas strings: a senha original e a que o usuário digitou, se estiverem iguais, sucesso. Se não, não entra ;)

Exemplo de uso do JPasswordField

Crie um programa em Java que pede o nome de usuário e senha para o usuário.

Caso estejam certos, uma mensagem de sucesso é exibida, caso contrário uma mensagem de erro é mostrada na caixa de diálogo.

Implemente também um botão para limpar ambos campos.

A única dificuldade que alguém pode ter neste exercício é o da comparação de strings, que é feita através do método `X.equals(Y)`, que retorna `TRUE` caso `X` seja igual a `Y`, e `FALSE` caso contrário.

E agora nosso botão "Limpar" coloca uma string vazia tanto no usuário como na senha.

Vamos usar dois JLabels, que estarão escritos "Usuário" e "Senha", ao lado das caixas de texto e de senha.

Como você é programador Java, certamente olhando para o código você é capaz de descobrir qual o usuário e senha para entrar no sistema :)

Main.java

```
import javax.swing.JFrame;

public class Main {

    public static void main(String[] args) {

        CaixaDeTexto texfield = new CaixaDeTexto();

        texfield.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        texfield.setSize(310,110);

        texfield.setVisible(true);

    }

}
```

CaixaDeTexto.java

```
import javax.swing.JButton;  
import javax.swing.JFrame;  
import javax.swing.JPasswordField;  
import javax.swing.JTextField;  
import javax.swing.JLabel;  
import java.awt.FlowLayout;  
import java.awt.event.ActionEvent;  
import java.awt.event.ActionListener;  
import javax.swing.JOptionPane;  
  
public class CaixaDeTexto extends JFrame{  
  
private JTextField usuario;  
  
private JPasswordField senha;  
  
private JButton login, limpa;
```

```
private JLabel user, pass;
```

```
public CaixaDeTexto(){
```

```
    super("Login em Java");
```

```
        setLayout(new FlowLayout());
```

```
        user = new JLabel("Usuário: ");
```

```
        add(user);
```

```
        usuario = new JTextField(15);
```

```
        add(usuario);
```

```
        pass = new JLabel("Senha: ");
```

```
        add(pass);
```

```
        senha = new JPasswordField(15);
```

```
        add(senha);
```

```
        login = new JButton("Entrar");
```

```
login.addActionListener(new ActionListener() {  
  
    public void actionPerformed(ActionEvent evento){  
  
        if(evento.getSource() == login)  
  
            if(usuario.getText().equals("Java") && senha.getText().equals("progressivo"))  
  
                JOptionPane.showMessageDialog(null, "Parabéns,  
                você entrou na Matrix!");  
  
            else  
  
                JOptionPane.showMessageDialog(null, "Senha  
                errada! Estude Java!");  
  
            }  
        }  
    };  
    add(login);  
  
    limpa = new JButton("Limpar");  
    limpa.addActionListener(new ActionListener() {
```



```
public void actionPerformed(ActionEvent evento){
```

```
    if(evento.getSource() == limpa){
```

```
        usuario.setText("");
```

```
        senha.setText("");
```

```
    }
```

```
    }
```

```
    }
```

```
    );
```

```
    add(limpa);
```

```
    }
```

```
}
```

Exercício de Java

Crie um programa que peça o peso do usuário (em kg) e sua altura (em metros) através de caixas de texto do tipo `TextField` e exiba o resultado do IMC da pessoa quando ela clicar em "Calcular">

A fórmula do IMC é: $\text{Peso}/(\text{altura} \times \text{altura})$

Implemente um botão de limpar.

JCheckBox - Como Usar Botões de Checagem (CheckBox ou Caixa de Seleção) (Tutorial de Java)

Neste Tutorial de Java sobre GUI, vamos saber o que é, para que serve e como usar o JCheckBox, um componente muitíssimo importante e usado em aplicações gráficas, que serve para 'checar' ou marcar uma opção.



Que Linguagens Devo Aprender?

 Java

 PHP

 C#

 C

JCheckBox em Java - O Que É e Para Que Serve

O JCheckBox é mais um elemento que o Java nos oferece para usarmos em aplicações gráficas (GUI), e nada mais é que aquela caixinha que vemos em sites e aplicativos, para "marcarmos" ou "checarmos" algo.

Sem dúvidas você já usou ela pra marcar a opção "Li e Aceito os Termos" (embora nunca tenha lido, né? :), para poder se cadastrar em algum serviço de email, rede social ou para aceitar a instalação de algum software.

Ela é dita de duplo estado: marcada ou não marcada.

A lógica dela, em Java, é que se ela estiver marcada, é como se retornasse o valor lógico true, e se não estiver marcada, retorna um false.

O JCheckBox, é uma espécie de botão (como JButton), porém é um botão que mantém seu estado.

Ou seja, uma vez que você clica nesse elemento, uma mudança ocorre nele e esta mudança permanece (ele fica selecionado ou não, de maneira permanente).

Diferente dos JButtons, que você clica, algo muda temporariamente e ele volta ao seu estado padrão (não pressionado).

Tutorial de JCheckBox - Como Usar Caixas De Seleção em Java

Sem mais delongas, vamos ver como inserir esses elementos em nossos aplicativos Java.

Inicialmente, vamos fazer um aplicativo que exiba a figura inicial deste tutorial: quatro JCheckBox, um com a opção "Java", outro "PHP", outro "C#" e uma caixa de seleção escrito "C".

Inicialmente, crie seu projeto Java.

Vamos criar uma classe chamada MyCheckBox.java, que vai conter o código Java necessário para a exibição dessas caixas de seleção. Esta classe deve estender a JFrame

Agora, importe a classe JCheckBox, do pacote javax.swing, a JFrame e a do layout que vamos usar:

```
import java.awt.FlowLayout;  
  
import javax.swing.JCheckBox;  
  
import javax.swing.JFrame;
```

Como queremos quatro caixas de seleção, vamos declarar quatro objetos da classe JCheckBox, e chamá-los de "java", "php", "csharp" e "c":

```
private JCheckBox java, php, csharp, c;
```

Dentro do construtor padrão, vamos definir o título de nosso aplicativo: "Que Linguagens Deseja Aprender?", definimos o que o programa deve fazer quando clicarem em fechar, dar um tamanho para ele (setSize) e vamos usar o FlowLayout (setLayout(new FlowLayout());).

Agora vamos inicializar nossas caixas de seleção JCheckBox, apenas passando os nomes que queremos que apareça nelas:

(o mesmo para os outros JCheckBox), e em seguida adicionamos cada um deles no layout:

```
add(java);
```

O código desta classe fica:

MyCheckBox.java

```
import java.awt.FlowLayout;
import javax.swing.JCheckBox;
import javax.swing.JFrame;

public class MyCheckBox extends JFrame {
private JCheckBox java, php, csharp, c;
public MyCheckBox(){

    super("Que Linguagens Deseja Aprender ?");
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setSize(400,100);
    setLayout( new FlowLayout() );

    //Criando os JCheckBox

    java = new JCheckBox("Java");
    php = new JCheckBox("PHP");
    csharp = new JCheckBox("C#");
    c = new JCheckBox("C");
```



```
//Adicionando os JCheckBox no Layout
```

```
add(java);
```

```
add/php);
```

```
add(csharp);
```

```
add(c);
```

```
setVisible(true);
```

```
}
```

```
}
```

Criando uma classe Main que tenha o método main para criar um objeto da classe MyCheckBox:

Main.java

```
public class Main {  
  
public static void main(String[] args) {  
    MyCheckBox myCheckBox = new MyCheckBox();  
}  
  
}
```

Rodando nosso projeto, o resultado é a figura lá de cima, deste tutorial.

Os JCheckBox aparecem, você pode clicar, selecionar e deselecionar.

Eventos em JCheckBox - Handler e Listener

Ok, os JCheckBox já apareceram e estão funcionando.

Mas o que eles fazem quando são clicados? Ué, faz o que você os programou para fazerem (no caso, nada).

Precisamos tratar os eventos e ações, e isso é possível em JCheckBox através da classe ItemListener e seu método itemStateChanged.

Diferente de antes, vamos usar Listener e Handler próprios par ao JCheckBox.

Importe:

```
import java.awt.event.ItemEvent;
```

```
import java.awt.event.ItemListener;
```

Agora vamos criar um JLabel, o "linguagens" que vai mostrar uma String com as linguagens que o usuário selecionar nas caixas de seleção.

Para fazer o tratamento de eventos, vamos criar uma classe interna chamada CheckBoxHandler, e ela deve implementar a ItemListener, e dentro uma string "texto" que vai servir para armazenar os itens selecionados.

Essa classe que implementa a ItemListener deve implementar o método "public

```
void itemStateChanged( ItemEvent event ) { } "
```

Dentro deste método `itemStateChanged` é que está o segredo de nosso aplicativo, pois iremos usar testes condicionais IF para testar se cada caixa de checagem está checada ou não.

Para saber isso (se a checkbox está marcada ou não), usaremos o método `isSelected()` da classe `JCheckBox`, que retorna `true` se a caixa estiver marcada e `false` se não estiver.

E caso esteja selecionado o `JCheckBox`, vamos adicionar o nome da linguagem de programação selecionada na string "texto", que ao final será exibida no `JLabel`.

Agora, vamos criar um objeto da classe `CheckBoxHandler`, o "tratador" e adicionar esse handler em cada um dos `JCheckBox`, através do método `addItemListener`.

Vejam como ficou o código da classe `MyCheckBox.java`:

MyCheckBox.java

```
import java.awt.FlowLayout;  
import java.awt.event.ItemEvent;  
import java.awt.event.ItemListener;  
import javax.swing.JCheckBox;  
import javax.swing.JFrame;  
import javax.swing.JLabel;  
  
public class MyCheckBox extends JFrame {  
  
private JCheckBox java, php, csharp, c;  
  
private JLabel linguagens = new JLabel("Linguagens selecionadas: ");  
  
public MyCheckBox(){  
  
    super("Que Linguagens Deseja Aprender ?");  
  
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
  
    setSize(400,100);  
  
    setLayout( new FlowLayout() );
```

```
//Criando os JCheckBox
```

```
java = new JCheckBox("Java");
```

```
php = new JCheckBox("PHP");
```

```
csharp = new JCheckBox("C#");
```

```
c = new JCheckBox("C");
```

```
//Criando e adicionando o handler
```

```
CheckBoxHandler tratador = new CheckBoxHandler();
```

```
java.addItemListener(tratador);
```

```
php.addItemListener(tratador);
```

```
csharp.addItemListener(tratador);
```

```
c.addItemListener(tratador);
```

```
//Adicionando os JCheckBox no Layout
```

```
add(java);
```

```
add(php);
```

```
add(csharp);
```

```
add(c);
```

```
add(linguagens);
```

```
setVisible(true);
```

```
}
```

```
private class CheckBoxHandler implements ItemListener{
```

```
private String texto = "";
```

```
@Override
```

```
public void itemStateChanged(ItemEvent evento) {
```

```
    texto = "";
```

```
    if(java.isSelected())
```

```
        texto += "Java ";
```

```
    if/php.isSelected())
```

```
        texto += "PHP ";
```

```
    if(csharp.isSelected())
```

```
        texto += "C# ";
```

```
    if(c.isSelected())
```

```
        texto += "C ";
```

```
linguagens.setText("Linguagens selecionadas: "+texto);
```

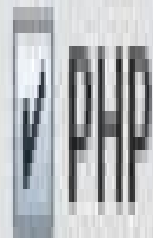
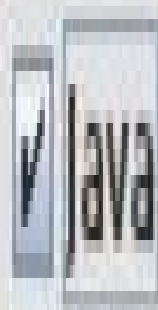
```
}
```

```
}
```

```
}
```




Que Linguagens Deseja Aprender?



Linguagens selecionadas: Java PHP C# C

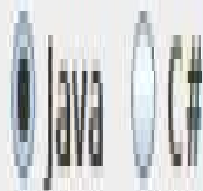
JRadioButton - Botão de Rádio ou de Opção (Tutorial de Java GUI)

Neste Tutorial de GUI em Java, vamos falar dos Radio Buttons, também conhecidos por botão de opção ou botão de rádio, que são usados para fazer escolhas únicas:



Radio Buttons - JavaProgressivo.net

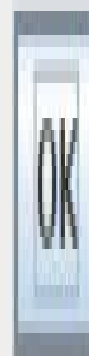
Qual a melhor linguagem de programação?



Mensagem



Parabéns, você é o cara





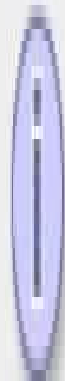
Radio Buttons - JavaProgressivo.net

Qual a melhor linguagem de programação?

☐ Java ☒ C#



Mensagem



Sabe de nada, inocente!

OK

JRadioButton em Java - O Que É e Para Que Serve

Certamente você já se deparou com algum Radio Button em algum aplicativo ou site, pois são muito comuns e usados no mundo da informática.

São aquelas "bolinhas" que você clica para marcar e fazer uma escolha.

Mas essa escolha é única, dentre todas as opções, você só pode marcar uma e é isso que difere o JRadioButton do JCheckBox, é que nas caixas de checagem você pode marcar várias opções ao mesmo tempo.

Logo, você deve usar os JRadioButton's quando quer oferecer uma diversidade de opções, mas somente uma delas deve ser marcada.

Por exemplo:

"Você quer programar em Java?"

Sim

Não

"Qual sua faixa de idade?"

Menos 12 anos

Entre 12 e 18 anos

Acima de 18 anos

No primeiro exemplo, ou você quer programar em Java, ou não. Não há duas respostas.

No segundo exemplo, idem. Ou tem menos de 12 anos, ou tem entre 12 e 18 ou tem mais, não há uma opção e outra ao mesmo tempo.

Essa é a ideia, a lógica por trás dos Radio Buttons, ou botões de opção: use quando a resposta for somente uma.

Como Usar Radio Button em Java - JRadioButton

Agora que já sabemos o que é um Radio Button, para que serve e em que situações são usados, vamos aprender como programar eles em Java. Vamos fazer o exemplo das duas imagens iniciais deste artigo.

Para tal, vamos iniciar importando a classe JRadioButton:

```
import javax.swing.JRadioButton;
```

Vamos criar dois botões de rádio, um para a opção Java e outro para C# (csharp), além de um JLabel para exibir o texto "Qual a melhor linguagem de Programação?"

Para inicializar nossos Botões de Opção, devemos fornecer duas coisas para o construtor JRadioButton, que é uma string com o texto que vai aparecer nas opções (no caso, "Java" e "C#") e o estado inicial do botão.

Note que o botão pode estar marcado ou desmarcado.

Se quiser que seu aplicativo Java inicie um um Radio Button marcado coloque true, ou false para desmarcado.

Vamos inicializar os nossos desmarcados:

Depois simplesmente adicionamos o label e os radio buttons (na ordem do

FlowLayout).

RadioButton Handler - Eventos com os Botões de Rádio

Como sempre, elementos gráficos em Java sem um handler, não são muito úteis.

O bacana é que certos eventos e ações ocorram quando formos interagir com esses elementos.

Ou seja, vamos criar nosso handler, para que as coisas aconteçam quando selecionarmos "Java" ou "C#" no nosso aplicativo.

Vamos criar nossa classe privada interna chamada "RadioButtonHandler" que deve, obrigatoriamente, implementar a interface "ItemListener".

Nessa interface, devemos implementar, obrigatoriamente, o método `itemStateChanged`.

Esse método que vai "ficar de olho" se algo foi clicado, marcado ou desmarcado.

Já que ele vai ficar atento se algo ocorre...o que pode ocorrer?

Ora, um radio button ser marcado ou desmarcado.

O método que retorna `true` se ele for marcado e `false` se for desmarcado é o `"isSelected()"`, presente em todo objeto do tipo `JRadioButton`.

Colocamos esses métodos dentro de testes condicionais IF.

Caso "java.isSelected()" seja verdadeiro (seja selecionado), exibimos um JOptionPane com a mensagem de parabéns. Caso "csharp.isSelected()" que retorne verdadeiro, mandamos a mensagem de "Sabe de nada, inocente".

Mas claro, para que tudo ocorra, devemos criar um handler (objeto do tipo RadioButtonHandler) e fazer com que os objetos "java" e "csharp" tenham seus eventos tratados por um handler dessa classe.

Isso é feito através do método "addItemListener" da classe JRadioButton.

Código Fonte do Tutorial de Java

Vamos ver agora tudo funcionando.

Crie seu projeto Java, e as duas classes "Main.java" e "RadioButton.java"

Main.java

```
import javax.swing.JFrame;
```

```
public class Main {
```

```
    public static void main(String[] args) {
```

```
        JButton radioButtonFrame = new JButton();
```

```
        radioButtonFrame.setDefaultCloseOperation(  
JFrame.EXIT_ON_CLOSE );
```

```
        radioButtonFrame.setSize( 350, 100 );
```

```
        radioButtonFrame.setVisible( true );
```

```
    }
```

```
}
```

RadioButton.java

```
import javax.swing.JFrame;  
import javax.swing.JRadioButton;  
import java.awt.FlowLayout;  
import java.awt.event.ItemEvent;  
import java.awt.event.ItemListener;  
import javax.swing.JLabel;  
import javax.swing.JOptionPane;  
  
public class RadioButton extends JFrame {  
  
private JRadioButton java,  
        csharp;  
  
private JLabel myLabel;  
  
private RadioButtonHandler handler;
```

```
public RadioButton(){
```

```
    super("Radio Buttons - JavaProgressivo.net");
```

```
        setLayout( new FlowLayout() );
```

```
        handler = new RadioButtonHandler();
```

```
        myLabel = new JLabel("Qual a melhor linguagem de programação?");
```

```
        java = new JRadioButton("Java", false);
```

```
        csharp = new JRadioButton("C#", false);
```

```
        add(myLabel);
```

```
        add(java);
```

```
        add(csharp);
```

```
        java.addItemListener(handler);
```

```
        csharp.addItemListener(handler);
```

```
    }
```

```
private class RadioButtonHandler implements ItemListener{
```

```
    @Override
```

```
public void itemStateChanged(ItemEvent event) {
```

```
    if(java.isSelected())
```

```
        JOptionPane.showMessageDialog(null,"Parabéns, você é o  
cara");
```

```
    if(csharp.isSelected())
```

```
        JOptionPane.showMessageDialog(null,"Sabe de nada,  
inocente!");
```

```
    }
```

```
}
```

```
}
```


A Classe `ButtonGroup` do Java

Ok, é fácil entender que somente uma das opções dos botões de rádio devem ser escolhida.

Mas se tivermos grupos diferentes de botões?

Por exemplo, se quisermos saber que linguagem o usuário quer estudar (que gera somente uma resposta) e também queremos saber em que estado brasileiro ele mora (usando radio buttons também) ?

Obviamente queremos ter alguns radio buttons para a primeira pergunta e outros botões para a outra pergunta. Mas vimos que o Java só deixa a gente selecionar uma opção como resposta, o que fazer?

A resposta é: agrupar os botões usando o `ButtonGroup`.

Importe essa classe:

```
import javax.swing.ButtonGroup;
```

E declare objetos que serão os grupos.

Vamos fazer duas perguntas no próximo exemplo, a primeira é "Que linguagem você deseja aprender" e a segunda é "Tem certeza?".

Vamos criar então mais dois JRadioButtons, o "sim" e o "nao", bem como outro JLabel para o outro texto da pergunta.

Vamos criar dois grupos, o "grupo1" e o "grupo2", que são objetos da classe ButtonGroup.

Esses objetos possuem o método "add()" para adicionamos os JRadioButtons separadamente.

Ou seja, no "grupo1" colocamos os botões "Java" e "C#", já no "grupo2" colocamos os botões "sim" e "nao".

Simples!

Agora vamos adicionar nosso objeto handler nesses botões "sim" e "nao", e dentro da classe RadioButtonHandler vamos fazer com que as mensagens sejam exibidas somente se o usuário clicar na linguagem que deseja aprender e no "Sim".

Código Fonte do Tutorial de Java

```
import javax.swing.JFrame;

public class Main {

    public static void main(String[] args) {

        RadioButton radioButtonFrame = new RadioButton();

        radioButtonFrame.setDefaultCloseOperation(
JFrame.EXIT_ON_CLOSE );

        radioButtonFrame.setSize( 450, 100 );

        radioButtonFrame.setVisible( true );

    }

}
```

RadioButton.java

```
import javax.swing.JFrame;  
import javax.swing.JRadioButton;  
import javax.swing.ButtonGroup;  
import java.awt.FlowLayout;  
import java.awt.event.ItemEvent;  
import java.awt.event.ItemListener;  
import javax.swing.JLabel;  
import javax.swing.JOptionPane;  
  
public class RadioButton extends JFrame {  
  
private JRadioButton java, csharp,  
                sim, nao;  
  
private JLabel myLabel, myLabel2;  
  
private ButtonGroup grupo1, grupo2;
```

```
private RadioButtonHandler handler;
```

```
public RadioButton(){
```

```
    super("Radio Buttons - JavaProgressivo.net");
```

```
        setLayout( new FlowLayout() );
```

```
        handler = new RadioButtonHandler();
```

```
        myLabel = new JLabel("Qual a melhor linguagem de programação?");
```

```
        myLabel2 = new JLabel("\nVocê tem certeza disso?");
```

```
        java = new JRadioButton("Java", false);
```

```
        csharp = new JRadioButton("C#", false);
```

```
        sim = new JRadioButton("Sim", false);
```

```
        nao = new JRadioButton("Não", false);
```

```
        add(myLabel);
```

```
        add(java);
```

```
        add(csharp);
```

```
        add(myLabel2);
```

```
        add(sim);
```

```
add(nao);
```

```
grupo1 = new ButtonGroup();
```

```
grupo1.add(java);
```

```
grupo1.add(csharp);
```

```
grupo2 = new ButtonGroup();
```

```
grupo2.add(sim);
```

```
grupo2.add(nao);
```

```
java.addItemListener(handler);
```

```
csharp.addItemListener(handler);
```

```
sim.addItemListener(handler);
```

```
nao.addItemListener(handler);
```

```
}
```

```
private class RadioButtonHandler implements ItemListener{
```

```
@Override
```

```
public void itemStateChanged(ItemEvent event) {
```

```
if(java.isSelected() && sim.isSelected())
```

```
    JOptionPane.showMessageDialog(null,"Parabéns, você é o  
cara");
```

```
if(csharp.isSelected() && sim.isSelected())
```

```
    JOptionPane.showMessageDialog(null,"Sabe de nada,  
inocente!");
```

```
    }
```

```
}
```

```
}
```

Arquivos (Files): Escrevendo (writing) , lendo (reading) , anexando (appending) e manipulando

Arquivos (Files) em Java

Neste tutorial de Java de nosso curso, vamos apresentar um conceito bem diferente de se trabalhar com dados, e sem dúvidas um dos mais importantes em programação Java.

Não é por menos que iremos dedicar uma seção inteira de nosso site para ensinar como usar arquivos (files), em Java.

O que são Arquivos em Java

Até o momento, em nossa Apostila Java Progressivo, vínhamos armazenando as informações em variáveis.

Inicialmente aprendemos a usar os vários tipos de variáveis, depois aprendemos a colocar tudo em Arrays, e por fim quando aprendemos ArrayList vimos a infinidade de possibilidade que o Java nos dá para organizar os elementos, inserir, tirar, achar, alterar e fazermos o que quisermos com objetos em um Array.

Porém, existe um problema nesses tipos de variáveis, nessas maneiras de armazenar e organizar informações: elas são temporárias.

Isso mesmo, quando seu programa termina, tudo o que você criou ou que foi computado pelo seu aplicativo Java irá se perder.

Imagine só, sempre que você desligar seu computador, perder todos os arquivos que você trabalhou/usou, todas as fotos, textos etc.

Sem condições, não é?

E é exatamente por isso que está faltando um 'algo a mais' em nosso aprendizado de Java!

No dia-a-dia, os programas que usamos, armazenam as informações de maneira permanente.

Você abre o aplicativo, usa, escolhe o que salva, o que não salva, fecha o programa e no outro dia quando for usar de novo, vai estar tudo lá.

É aí que entram os arquivos: eles são maneiras de armazenarmos informações de maneira permanente.

São os arquivos mesmo, que conhecemos, como um arquivo de texto (.txt, .doc), uma música (.mp3), um vídeo (.avi) ou uma foto que salvamos (.jpg).

Vamos aprender como criar arquivos, abrir um arquivo existente no seu computador, como ler suas informações, alterar e até mesmo apagar arquivos.

Para que estudar Arquivos em Java

Vamos voltar ao nosso clássico exemplo da empresa, daquela que você foi contratado para criar um sistema com os dados de cada funcionário da empresa.

Cada empregado é um objeto (da classe "Funcionário", por exemplo), com os dados de nome, endereço, telefone, salário, setor etc.

Você criou um sistema em que eles forneceram tais informações. Ok, e onde estas informações vão ficar?

Em ArrayList, no computador-servidor da empresa? Ok.

E se houver um problema? Faltar energia ou o servidor quebrar? Vai perder os dados?

Vai pedir para eles preencherem tudo de novo sempre que isso ocorrer?

E o registro de mudanças, contratações, demissões, promoção etc?

Onde vai guardar as informações sobre tudo isso.

Não há para onde correr: você deve armazenar essas informações de maneira permanente (além de ter que fazer um backup periodicamente em outros locais).

Sabe quando liga seu computador, com seu login e senha?

Como o computador sabe que aqueles dados estão certos? Ora, ele deve ter essa informação de login e da senha armazenadas em algum lugar em sua memória. Daí você entra com os dados, e seu sistema operacional vai lá checar se está ok.

E quando você vai jogar algum jogo no seu computador? Já reparou que sempre inicia exatamente de onde você parou? E isso acontece no computador, vídeo-game ou até naquele joguinho Java do seu celular.

E aí, o jogo adivinhou o que você fez no passado?

Óbvio que não, ele simplesmente foi armazenando tudo o que você fez em arquivos de seu computador.

E podemos ficar horas aqui explicando a importância dos arquivos, mas certamente já deu pra entender o mundo de novas coisas que podemos fazer com os arquivos.

O fato é que nossos programas irão se 'comunicar' com os computadores, abrir arquivos, alterar dados de um pendrive, se auto-duplicar, guardar senhas, apagar arquivos, fazer back-up e praticamente tudo o que você quiser.

A coisa está começando a ficar interessante, não?

Entrada, Saída e Tipos de dados - Fluxo (stream) e o pacote Java.io

Agora que já explicamos o que são e a importância dos arquivos, nesse tutorial vamos entrar em mais detalhes sobre como usar os arquivos em nossos programas Java.

Para isso, vamos entender um pouco mais sobre os tipos de dados que vamos trabalhar, por onde entram, saem, fluxo (stream) e o pacote java.io que vai nos permitir fazer uso dos arquivos.

Tipos de dados dos arquivos em Java

Nesta altura do campeonato de nossa apostila Java Progressivo, você já deve estar careca de saber que, em níveis mais baixos, tudo em programação e computador se resume aos números binários (1's e 0's), que são formas de se representar alguns processos físicos (voltagem, corrente etc) em eletrônica.

E se tudo, à rigor é 0 e 1, o Java também vê os arquivos como números binários.

No caso específico do Java, ele trabalha com bytes.

Assim, teremos sempre a opção de se trabalhar diretamente com os números binários, em arquivos.

Porém, para os programadores humanos (alguns parecem que não são :), é algo extremamente enfadonho, para não dizer fora de cogitação, se trabalhar com uma sequência de números 0's e 1's.

Por exemplo, cada caractere, em unicode, é representado por 2 bytes em Java, ou 8 bits.

Assim, uma outra maneira de lidarmos com arquivos, que a linguagem Java nos oferece, é através de caracteres, e é dessa maneira que iremos focar em nosso curso.

Fluxo (stream) - Entrada(in) e Saída(out) de dados

Em Java, os dados de um arquivo estão organizados em série, ou seja, um atrás do outro.

Ao abrirmos um arquivo para leitura, por exemplo, começamos a leitura (acesso aos dados) a partir dos bytes iniciais, e vamos tendo acesso byte por byte.

O término desse fluxo sequencial de bytes é dado por uma espécie de sinalização, um marcador que avisa que o arquivo chegou ao final.

O uso dos arquivos em Java é bem mais comum do que imaginamos até o momento. É tanto que ele sempre criamos um aplicativo, o Java cria três objetos para se trabalhar com o fluxo de dados.

Eles são o System.in (para receber dados, in é dentro em inglês), o System.out (saída de dados, out é fora em inglês) e o System.err (para erros).

E sim, já trabalhamos como o System.in quando trabalhamos com a classe Scanner para receber dados e com System.out com as funções printf para exibir saídas de texto na tela.

Essas são as entradas e saídas padrão do Java.

Nesta seção sobre arquivos vamos usar as saídas e entradas, mas não as padrões (teclado e tela), e sim arquivos. Ou seja, vamos receber as entradas de um arquivo (e não mais pelo teclado) e vamos direcionar as saídas de nosso aplicativo para os arquivos (e não mais para a tela do computador).

O pacote java.io

No estudo e manipulação de arquivos em Java, será necessário o uso de um pacote especial, o java.io (io de in out), pois não iremos mais trabalhar com as entrada e saída padrão.

Esse package oferece um acervo gigantesco de possibilidades, classes e funções para trabalharmos com arquivos de uma maneira bem, mas bem mais simples do que em muitas outras linguagens.

Este pacote possui funções que nos permitem checar se um dado endereço (URL) é um arquivo, um diretório, se existe ou não, deletar, ver o tamanho, localização, ver quando foi criado ou modificado, e diversas outras possibilidades, como é possível ver na documentação oficial:

<http://docs.oracle.com/javase/7/docs/api/java/io/package-summary.html>

Em nosso curso iremos focar bastante a classe InputStream e a FileInputStream (para ler) e OutputStream e FileOutputStream (para escrever) trabalhando-se com bytes, além de classes específicas, como as classes Reader, InputStreamReader e BufferedReader para trabalharmos com chars (caracteres) e Strings.

Como ler caracteres, Strings e Bytes de um arquivo em Java

Neste tutorial de nossa apostila Java Progressivo, iremos aprender as diferentes maneiras de se ler dados de um arquivo na linguagem Java.

Iniciaremos mostrando como ler caractere por caractere de um arquivo.

Em seguida, veremos como ler Strings (ler linhas inteiras).

E por fim, veremos como ler quaisquer tipos de bytes de um arquivo, seja seu conteúdo de texto, vídeo, música ou binário.

Como ler caracteres de um arquivo em Java

Inicialmente, crie um arquivo chamado "file.txt", e o bote dentro do projeto de Java, junto das pastas src e bin. Escreve apenas a letra 'a' nesse arquivo.

Para abrir este arquivo, iremos usar a classe `FileInputStream` que irá receber uma string que representa o local onde está armazenado o arquivo em seu computador.

Como ele está na pasta do próprio projeto, apenas colocamos o nome do arquivo: `file.txt`

Mas você pode colocar o endereço completo:
`/usr/JavaProgressivo/Arquivos/file.txt`

Vamos chamar de "entrada" o objeto responsável pela abertura do arquivo.

Como queremos trabalhar com caracteres, a maneira que este objeto vai ler o arquivo é uma maneira especial, pois deve ler bytes que representem caracteres.

Vamos usar a classe `InputStreamReader` para 'tratar' esse objeto que abriu o arquivo, essa classe vai fazer com que a leitura do arquivo seja uma leitura específica, uma leitura de caracteres.

Vamos chamar esse objeto de 'entradaFormatada'.

E, finalmente, para ler o primeiro caractere do arquivo (não importa quantos bytes tenham sido lidos, pois a `InputStreamReader` já tratou de ler e formatar

essa entrada para caractere) iremos usar o método read(), que retorna um inteiro que representa o caractere.

Assim, nosso código que lê esse primeiro caractere fica:

```
import java.io.*;
```

```
public class Arquivos {
```

```
public static void main(String[] args) throws IOException{
```

```
FileInputStream entrada = new FileInputStream("file.txt");
```

```
InputStreamReader entradaFormatada = new  
InputStreamReader(entrada);
```

```
int c = entradaFormatada.read();
```

```
System.out.println(c);
```

```
}
```

```
}
```

O resultado deve ser 97, que é a representação de 'a'.

Para podermos exibir na tela o caractere 'a', basta fazermos o cast para char:

```
System.out.print( (char) c);
```

Lendo mais de um caractere - Fim de arquivo

Ok, aprendemos a ler um caractere de um arquivo.

Mas, e para ler outro? Usar o método de novo? Sim, funciona.

E se seu arquivo for um texto enorme, de milhares de caracteres, usar milhares de vezes o `read()` ?

Óbvio que não.

Podemos usar um looping, como através do laço `while`, que deve ler todos os caracteres e só parar quando terminar o arquivo.

E como que vamos saber quando termina o arquivo?

Fácil: o maravilhoso método `read()` retorna o valor `-1` quando isso ocorre.

E ele faz outra coisa interessante: após ele ler um caractere, ele imediatamente aponta para o próximo.

Assim, escreva em seu arquivo 'file.txt' o seguinte texto (ou qualquer outro):
Apostila Java Progressivo

Para exibir todo o conteúdo de texto de um arquivo, usaremos o seguinte código:

```
import java.io.*;
```

```
public class Arquivos {  
  
public static void main(String[] args) throws IOException{  
  
FileInputStream entrada = new FileInputStream("file.txt");  
  
InputStreamReader entradaFormatada = new  
InputStreamReader(entrada);  
  
    int c = entradaFormatada.read();  
  
    while( c!=-1){  
  
        System.out.print( (char) c);  
  
        c = entradaFormatada.read();  
  
    }  
  
}
```

PS: Quando você cria o código de leitura de arquivos, automaticamente a IDE insere o 'throws IOException', para tratar eventuais erros e exceções.

Como ler Strings de um arquivo em Java

Muitas vezes se trabalhar com caractere por caractere (como é feito na linguagem C), é algo um pouco chato e trabalhoso.

Por exemplo, no código passado tivemos que criar um laço while para ler uma frase.

Visando facilitar a vida do programador Java, existe uma classe chamada `BufferedReader` que recebe como argumento um objeto do tipo `InputStreamReader` e agrupa os caracteres até formar uma linha.

Ou seja, ele recebe os caracteres (entrada formatada) e vai agrupando eles até encontrar o símbolo de new line '\n', e retorna essa String (sem o \n).

Vamos chamar esse objeto da classe `BufferedReader` de 'entradaString' e vamos ler a primeira e única linha de nosso arquivo 'file.txt':

Também iremos criar a variável 'linha', do tipo String para receber as linhas.

Linhas essas que serão lidas e retornadas através do método `readLine()`.

Note como as coisas são análogas ao caso do caractere.

Lá, quando chegava ao final do arquivo retornava o valor -1, como aqui estamos trabalhando com Strings, ao encontrar o final do arquivo, o método `readLine()` retorna o nosso velho e conhecido null.

Agora escreva outra linha: "C Progressivo", "HTML Progressivo", "Programação Progressiva" etc.

O código Java para exibir todas essas linhas será:

```
import java.io.*;
```

```
public class Arquivos {
```

```
public static void main(String[] args) throws IOException{
```

```
FileInputStream entrada = new FileInputStream("file.txt");
```

```
InputStreamReader entradaFormatada = new  
InputStreamReader(entrada);
```

```
BufferedReader entradaString = new BufferedReader(entradaFormatada);
```

```
String linha = entradaString.readLine();
```

```
while(linha != null){
```

```
System.out.println(linha);
```

```
        linha = entradaString.readLine();
```

```
    }
```

```
}
```

```
}
```

Como ler bytes de um arquivo em Java

Para ler byte por byte de um arquivo, iremos usar duas classes: a `InputStream` e a `FileInputStream` (esta é filha da primeira).

Vamos passar para esta segunda classe o endereço (URL) de nosso arquivo de texto.

Como vamos colocar ele no diretório do projeto (junto as pastas `src` e `bin`), fica:

```
InputStream entrada = new FileInputStream("file.txt");
```

Isso já abre e posiciona o leitor de bytes apontando para o primeiro byte do arquivo.

Vamos usar um inteiro para receber byte por byte, o `"umByte"`.

Note que, embora ele leia o byte (8 bits), o Java já transforma esta informação em um inteiro (bacana, esse Java, não?)

Para ler um byte, vamos fazer uso do método `read()`, que lê e retorna o byte do arquivo.

Escreva no `'file.txt'` simplesmente o caractere `'a'`.

Nosso programa que lê 1 byte fica:

```
import java.io.*;
```

```
public class Arquivos {  
  
    public static void main(String[] args) throws IOException{  
  
        InputStream entrada = new FileInputStream("file.txt");  
  
        int umByte = entrada.read();  
  
        System.out.print(umByte);  
  
        }  
  
    }
```

Note que o resultado é 97, que é o inteiro que representa o caractere 'a'.

Para ver o caractere 'a' aparecer na tela, basta colocar um cast para que o Java automaticamente transforme o inteiro em caractere. Para isso, basta fazer:

```
System.out.print((char)umByte)
```

Vamos agora escrever mais coisas em nosso arquivo.

Substitua o 'a' por "Apostila Java Progressivo". Ou seja, agora temos vários, vários bytes para ler.

Para criarmos um programa que leia e exiba todos os bytes, vamos usar um laço que vai rodar até a classe FileInputStream encontrar o final do arquivo, e quando isso ocorrer o método read() retorna o valor, que em inteiro, é -1.

E como o método read(), antes de retornar o byte lido, pula (aponta) para o próximo byte, para lermos todos os bytes de um arquivo de texto, nosso código deve ser:

```
import java.io.*;
```

```
public class Arquivos {
```

```
public static void main(String[] args) throws IOException{
```

```
InputStream entrada = new FileInputStream("file.txt");
```

```
    int umByte = entrada.read();
```

```
while(umByte != -1){
```

```
    System.out.print((char)umByte);
```

```
        umByte = entrada.read();
```

```
    }
```

```
}
```

```
}
```

Embora tenhamos usado a `FileInputStream` para ler caracteres, ela pode ser usada para se trabalhar com quaisquer tipos de dados. É tanto que não usamos o `InputStreamReader` para tratar nossos bytes, como fizemos quando fomos ler caracteres e Strings.

Fechando arquivos - O método close()

Embora a JVM (Java Virtual Machine) tenha um fantástico Garbage Collector, que vai eliminando variáveis que não usamos e fechando arquivos que não mais acessaremos, nem sempre ela consegue prever o que vai ser usado ou não em um programa.

Por isso, independente da complexidade de seus aplicativos Java é importante você fechar os arquivos que abriu (seja pra ler ou escrever).

Isso é feito através do método `close()`, presentes nas classes que usamos para tratar os arquivos.

Assim, faça:

```
entrada.close();
```

Java: Class File (a classe File) - Obtendo informações de arquivos e diretórios

Objetos dessa classe irão servir para objetos de outras classes da java.io package.

Usamos ela para obter informações SOBRE o arquivo ou diretório/pasta, e não sobre seu conteúdo.

Ao contrário de outras classe e do que o nome possa sugerir, essa classe não abre, ou permite o processamento de conteúdo de um arquivo. A sua função está relacionada com o caminho / diretório / endereço dos arquivos ou diretório que desejarmos manipular.

Ela provém, diversos métodos para checagem de existência de arquivos ou diretórios, se é um ou outro, o tamanho do arquivo, a última vez que foi alterado, se é possível escrever ou lê-lo, etc. Adiante explicarei mais detalhadamente alguns dos métodos da classe File.

Dentre os argumentos passados para o constructor de um File object, destacamos:

path -> é o caminho do arquivo ou diretório

absolute path -> todos os diretórios, começando pelo do root, até o arquivo ou diretório de interesse

relative path -> começa no diretório onde a aplicação é executada, que é parente do diretório atual

URL -> endereço na web, Uniform Resource Locators

URI -> endereço no computador, Uniform Resource Identifier. Como o

'/home/ProgramacaoProgressiva/' ou 'C:/windows/forever alone' caso ainda esteja no caminho das trevas

Alguns métodos da classe File:

boolean canRead() -> retorna true se for possível ler o arquivo, falso o contrário

boolean canWrite() -> retorna true se for possível escrever no arquivo, falso o contrário

boolean exists() -> retorna true se o diretório ou arquivo se o objeto File existe, falso o contrário

boolean isFile() -> retorna true se o argumento passado ao construtor da File é um arquivo, falso o contrário

boolean isDirectory() -> retorna true se o argumento passado ao construtor da File é um diretório, falso o contrário

boolean isAbsolute() -> retorna true para caso o argumento seja de um caminho absoluto, falso o contrário

String getAbsolutePath() -> retorna uma String com o caminho absoluto do diretório ou arquivo

String getName() -> retorna uma String com o nome do arquivo ou do diretório

String getPath() -> retorna uma String com o caminho do arquivo ou diretório

String getParent() -> retorna uma String com o caminho do diretório pai (acima;anterior) ao do arquivo ou diretório atual

long length() -> retorna um tamanho, em bytes, do arquivo ou inexistente, caso seja diretório

long lastModified() -> retorna o tempo em que o arquivo ou diretório foi modificado pela última vez; varia de acordo com o sistema

String[] list() -> retorna um array de Strings com o conteúdo do diretório, ou null se for arquivo

Para efeito de fixar conhecimento, um código-fonte vale mais que mil tutoriais (inventei agora, ficou legal?):

FileStudy.java

```
import java.util.Scanner;
```

```
import java.io.File;
```

```
public class FileStudy
```

```
{
```

```
public static void main(String[] args)
```

```
{
```

```
Scanner entrada = new Scanner( System.in );
```

```
System.out.print("Entre com um endereço válido de um arquivo: ");
```

```
String caminho = entrada.nextLine();
```

```
    metodos(caminho);  
}
```

```
public static void metodos(String caminho)  
{
```

```
File arquivo = new File(caminho);
```

```
if( arquivo.exists() )  
  
    {
```

```
System.out.println("O caminho especificado existe !\nVamos aos testes:\n");
```

```
if(arquivo.isAbsolute())
```

```
System.out.println("É um caminho absoluto");
```

```
else
```

```
System.out.println("Não é um caminho absoluto");
```

```
if(arquivo.isFile())
```

```
System.out.printf("É um arquivo de tamanho %s bytes\n"
```

```
    + "Útima vez modificado %s\n"
```

```
    + "Cujo caminho é %s\n"
```

```
    + "De caminho absoluto %s\n"
```

```
    + "E está no diretório pai %s\n",
```

```
        arquivo.length(), arquivo.lastModified(), arquivo.getPath(),  
arquivo.getAbsolutePath(), arquivo.getParent() );
```

```
else
```

```
    {
```

```
System.out.println("É um diretório cujo conteúdo tem os seguintes arquivos: ");
```

```
String[] arquivos = arquivo.list();
```

```
for( String file : arquivos)
```

```
System.out.println( file );
```

```
    }
```

```
}
```

else

```
System.out.println("Endereço errado");
```

```
}
```

```
}
```

Diferente do comum, não irei comentar nada aqui, pois ao rodar o programa verá o que cada método faz.

Enfim, rode e estude. Está tudo prontinho e mastigado em suas mãos.

Java: Class Formatter (a classe Formatter) - Escrevendo em arquivos

Já vimos como obter informações sobre arquivos e diretórios, através da classe `File`, agora vamos realmente criar um arquivo que nos permita manipular informações de texto na forma sequencial.

- Criando um arquivo de texto

Para criar uma saída formatada, vamos usar um objeto da classe Formatter.

Além de abrir o arquivo para escrita, vamos testar antes para ver se é possível escrever nele, se temos permissões e se o arquivo existe.

Isso é importante para deixar nossas aplicações bem robustas e a prova de erros.

Não vá, simplesmente, abrindo os arquivos. Antes, teste se eles existem, se o usuário tem permissão de abrir, para checar se é possível mesmo escrever nele.

CriandoArquivoTexto.java

```
public class CriandoArquivoTexto
```

```
{
```

```
public static void main(String[] args)
```

```
{
```

```
    EscreverMetodos teste = new EscreverMetodos();
```

```
    teste.abrir();
```

```
    teste.escrever();
```

```
    teste.fechar();
```

```
}
```

```
}
```

EscreverMetodos.java

```
import java.util.Formatter;
```

```
import java.util.NoSuchElementException;
```

```
import java.util.FormatterClosedException;
```

```
import java.lang.SecurityException;
```

```
import java.io.FileNotFoundException;
```

```
public class EscreverMetodos
```

```
{
```

```
private Formatter arquivo;
```

```
public void abrir()
```

```
{
```

```
try
```

```
{  
    arquivo = new Formatter("ProgramacaoProgressiva.txt");  
}
```

```
catch( SecurityException semPermissao)
```

```
{
```

```
System.err.println(" Sem permissao para escrever no arquivo ");
```

```
System.exit(1); //exit(0) é sucesso, outro número significa que terminou com  
problemas
```

```
}
```

```
catch( FileNotFoundException arquivoInexistente )
```

```
{
```

```
System.err.println(" Arquivo inexistente ou arquivo não pode ser criado");
```

```
System.exit(1);
```

```
}
```

```
}
```

```
public void escrever()
```

```
{
```

```
try
```

```
{
```

```
    arquivo.format("Escrita no arquivo realizada com sucesso");
```

```
}
```

```
catch(FormatterClosedException formatoDesconhecido)
```

```
{
```

```
    System.err.println("Erro ao escrever");
```

```
return;
```

```
}
```

```
catch(NoSuchElementException excecaoElemento)
```

```
{
```

```
    System.err.println("Entrada invalida. Por exemplo, era pra ser uma string, mas
```

```
foi um inteiro");
```

```
}
```

```
}
```

```
public void fechar()
```

```
{
```

```
    arquivo.close();
```

```
}
```

```
}
```

Após compilar e rodar, verá que foi criado um arquivo chamado ProgramacaoProgressiva.txt com a string "Escrita no arquivo realizada com sucesso".

Usei a saída formatada 'arquivo.format()' para você saber que pode passar informações formatadas exatamente da mesma maneira que você fazia com printf.

Note que é sempre importante fecharmos um arquivo que não estamos mais usando, liberando recursos da máquina.

Java: Class Scanner (a classe Scanner) - Lendo e Recebendo dados de arquivos

Agora que já escrevemos dados em um arquivo, vamos aprender como ler dados de um arquivo existente em disco.

Receber dados é 'input', então podemos fazer usando a Scanner. A diferença é que, em vez de receber a entrada pelo teclado, vamos receber os dados de um arquivo.

Assim como na Class Formatter, coloquei algumas exceções para o programa ficar mais robusto, estável e para que você tenha o conhecimento de alguns dos métodos de tratamento de erros. As explicações estão nas strings do código-fonte.

Crie um arquivo chamado "teste.txt" e escreva algo nele, de preferência algo com espaços em branco, tabs e quebras de linhas para você ver o que acontece.

Eu coloquei:

linha 1

linha 2

linha 3

Vamo aos códigos fonte:

Lendo.java

public class Lendo

{

public static void main(String[] args)

{

LendoMetodos teste = new LendoMetodos();

teste.abrir();

teste.ler();

teste.fechar();

}

}

LendoMetodos.java

import java.io.File;

import java.util.Scanner;

import java.lang.IllegalStateException;

import java.util.NoSuchElementException;

import java.io.FileNotFoundException;

```
public class LendoMetodos
```

```
{
```

```
private Scanner entrada;
```

```
public void abrir()
```

```
{
```

```
try
```

```
{
```

```
    entrada = new Scanner( new File("teste.txt") );
```

```
}
```

```
catch(FileNotFoundException arquivoNaoEncontrado)
```

```
{
```

```
    System.err.println("Nao foi possível abrir o arquivo! Não encontrado!");
```

```
    System.exit(1); //saída de erro
```

```
}
```

```
}
```



```
public void ler()
```

```
{
```

```
try
```

```
{
```

```
while( entrada.hasNext() )
```

```
System.out.printf("%s - %d\n",entrada.next(), entrada.nextInt());
```

```
}
```

```
catch(NoSuchElementException entradaDiferente)
```

```
{
```

```
System.err.println("Entrada diferente da esperada");
```

```
    entrada.close();
```

```
System.exit(1);
```

```
}
```

```
catch(IllegalStateException erroLeitura)
{

System.err.println("Erro de leitura. Scanner foi fechada antes da input");

System.exit(1);

}
}
```

public void fechar()

```
{
    entrada.close();
}
}
```

O looping: while(entrada.hasNext())

mostra linha por linha do arquivo, até o final. Assim, o processamento e formatação é feito tendo em mente esse raciocínio, por linha.

A formatação é você quem escolhe, usando entrada.next(), entrada.nextInt(), entrada.nextDouble() etc.

Mais uma vez, ao término do uso do arquivo, fechá-lo: `entrada.close()`

Mercado de Trabalho

Neste artigo de nossa apostila Java Progressivo, iremos sair um pouco do conteúdo técnico de programação e passar um pouco de nossa experiência sobre o mercado de trabalho brasileiro para quem deseja ganhar a vida como programador.

Como se tornar um programador Java profissional

Entre dúvidas por comentários, no fórum e por e-mail, sobre classes, objetos, herança, polimorfismo e outros assuntos dessa sopinha de letras que é a Orientação a Objetos em Java, surge sempre um tema um pouco diferente: trabalho.

Isso mesmo, programação é algo tão apaixonante que uma das primeiras coisas que as pessoas pensam é em trabalhar, viver e ganhar dinheiro com isso. Nada mais natural, todos tem contas para pagar e sonhos a serem realizados que só são possíveis com dinheiro.

A importância da programação já foi abordada em diversos artigos de nossa apostila, principalmente nos mais básicos, mas pra resumir: olhe em torno de você. Veja seu computador, netbook, tablet, celular, som, tv digital, carros modernos, GPS, relógios etc etc.

Repare tudo que for digital.

Reparou?

É programação. Só funciona porque existem programadores que criaram.

Resumindo: é algo que tá fazendo tanto parte da vida das pessoas que não digo que é importante, é essencial. Se tiver dúvida, pergunte para algum parente adolescente do que seria a vida dele sem o Facebook (feito com programação, obviamente).

Existe mercado, e muito, para isso. Mas melhor que isso é: a demanda de mais programadores é simplesmente fora do normal. A primeira coisa que uma pessoa faz ao conseguir dinheiro é investir em coisas digitais, que usam tecnologia. Enfim: estão usando e gastando cada vez mais com tecnologia.

Então ter a programação como profissão é investir no futuro, o negócio só vai esquentar com o tempo...

Se interessou? Vamos dar duas dicas sobre como ser um programador profissional...

1. Estude. Estude mais, e de novo. Não pare.

Vamos ser logo sinceros de cara: você não vai assistir umas aulas, ler uns tutoriais, se tornar programador e nunca mais estudar na vida.

Primeiro você vai estudar muito, mas muito mesmo no começo. No início vai parecer impossível, mas a medida que for estudando, as coisas vão fazendo sentido.

Lembra do ICQ ? Do MSN? Do Super Mario World? Do Orkut? Do seu celular tijolão?

Enfim, as coisas mudam, evoluem, se transformam muito rápido nesse ramo.

Daqui uns anos você vai falar em Facebook e vão rir de você, por ser velho.

Você vai sempre ter que estudar.

A coisa mais importante para se tornar um programador profissional é estudar, saber o máximo possível, ter lido o máximo possível, programado o máximo possível.

Não tem segredo, é trabalho duro.

Estude por nossos cursos, faça os exercícios, compre livros e os revire de ponta a ponta (durma com eles na cabeceira), baixe apostilas, veja vídeo-aulas, acesse fóruns, pergunte e responda dúvidas, e o principal: arregace as mangas e crie códigos, na mão. Vire madrugadas, pense numa melhor solução, numa melhor sacada para seu código até no banho.

Pronto. Isso é o principal, se esforçar o máximo possível.

Os maiores gênios desse ramo não acordaram um dia, tiveram uma ideia na sorte e ficaram ricos e famosos. Não, eles estudaram pra cara...mba.

2. Prove que estudou

Já ouviu alguém falar que chegou num hospital, disse que era médico e foi contratado?

E de alguém que chegou numa empresa e disse 'eu posso administrar' e virou gerente?

Não importa o ramo profissional, é necessário provar que estudou, que se qualificou.

Sim, é possível frequentar faculdades e cursos e ser um baita de um incompetente.

No geral, basta decorar umas coisas na noite anterior a prova, que você conseguirá se formar em boa parte dos cursos do Brasil.

Mas se você seguiu a dica 1 que demos, você vai passar longe desse grupo.

O fato é que: seja em Havard, ITA, IME, Federal, Particular, Curso Técnico ou Certificação, vão te cobrar documentos. É necessário provar que estudou em algum lugar.

Infelizmente só a palavra não conta muito.

Tem gente até que compra uns livros altamente 'bizurados' sobre certificação em Java, decora umas coisas e se tornam certificados oficialmente, e pouco sabem. Sim, dá pra conseguir certificados sem fazer o passo 1, mas vamos adiantando: não adianta de nada ter um diploma, se na hora que você sentar na cadeira pra programar for um baita de um incompetente.

Vale a pena fazer uma faculdade, um curso? Claro que vale, mas aproveite, aproveite seu professor, o material, tente sugar o máximo de conhecimento possível.

Você pode até levar estudando pouco, mas nunca vai ser bom assim.

E vamos ser sincero novamente: o mais indicado é o roteiro básico de prestar um vestibular, ir pra faculdade, estagiar pra pegar experiência e trabalhar.

É o mais requisitado e normal de acontecer.

Mas nem todo mundo pode. Tem gente que não estudou pra passar numa universidade pública, tem gente que não tem dinheiro pra pagar uma faculdade particular, algumas pessoas trabalham e não tem tempo, temos muitos leitores adolescentes, vários de nossos estudantes moram em cidades que não oferecem cursos de programação.

E aí José, o que fazer?

Curso de Java Online com Certificado da Brava Cursos

Essas pessoas vêm até nós e pedem um certificado do Java Progressivo, e infelizmente isso necessitaria de uma maneira de avaliar o conhecimento, estrutura pedagógica e profissionais do ramo.

E como sabem, nosso projeto é totalmente independente, assim não temos condições (no momento) de ofertar uma certificação.

Mas não vamos deixar os leitores na mão, principalmente quem quer estudar por vontade e determinação própria. Vamos dar uma dica: Brava Cursos.

É uma empresa totalmente voltada, especializada, com experiência e respeito no ramo de cursos online com certificado.

Eles oferecem um curso com:

Vídeo aulas

Materiais próprios

Tutores para tirar dúvidas

Material em áudio

Exercícios

Avaliações

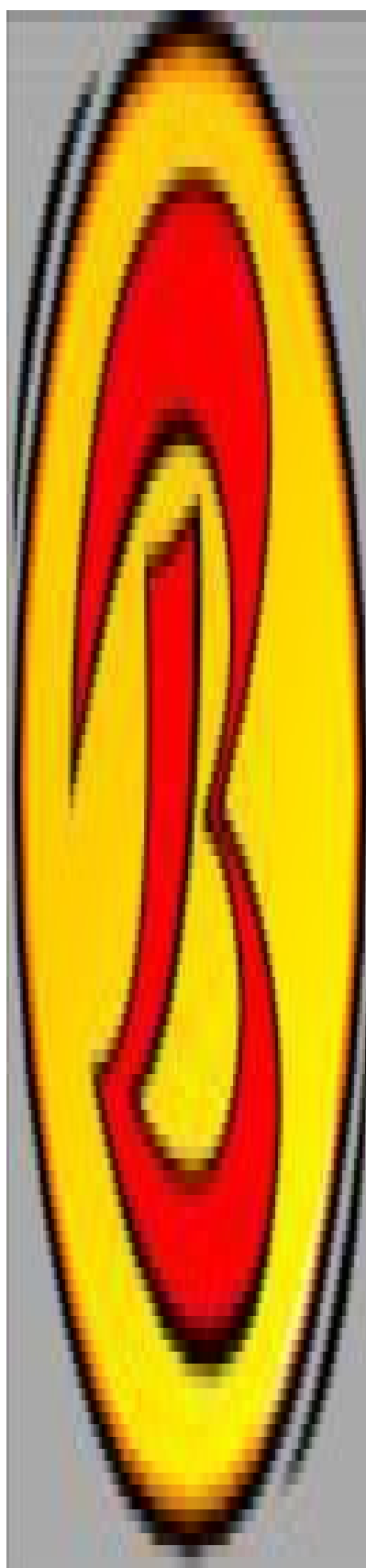
Certificado válido em todo território nacional (ABED - Associação Brasileira de Ensino à Distância)

Valor único e baixíssimo!!!

Ao fazer o curso, a Brava Cursos repassa uma porcentagem para o Projeto Java Progressivo, e nos ajuda a crescer e cada vez mais oferecer material gratuito e de qualidade

Gostou? Clique abaixo para tirar seu certificado:

[Obter Certificado do Curso Online de Java da Brava Cursos](#)



Brava Cursos

ENSINO A DISTÂNCIA



Writer

to

PUB

Created with Writer2ePub

by Luca Calcinai