

LINGUAGEM ORIENTADA A OBJETOS

Jesimar da Silva Arantes

0

Deseja ouvir este material?

Áudio disponível no material digital.

Ver anotações

CONHECENDO A DISCIPLINA

Caro aluno, seja bem-vindo à disciplina de Linguagem Orientada a Objetos. A cada dia observamos o surgimento de novas linguagens de programação baseadas nos mais diversos paradigmas de desenvolvimento. Apesar disso, uma linguagem se destaca há um certo tempo, o Java, que utiliza a orientação a objetos como filosofia de desenvolvimento.

Atualmente, no ano de 2020, das dez linguagens de programação mais utilizadas (Java, C, Python, C++, C#, Visual Basic, JavaScript, PHP, SQL e R) no mundo, oito delas suportam, de alguma forma, o desenvolvimento orientado a objetos – as exceções são C e SQL.

As linguagens Java e C lideram o mercado de desenvolvimento de softwares há mais de 20 anos, segundo o respeitado site da Tiobe (2020). Durante esses últimos 20 anos, na grande maioria das vezes a liderança de mercado foi ocupada pela linguagem Java, algumas poucas vezes por C. Esses fatos mostram a importância de uma disciplina que aborde os conteúdos de desenvolvimento orientado a objetos com Java.

Com o desenvolvimento orientado, temos a capacidade de atuar em diversas áreas, como o desenvolvimento de aplicações para desktop, para web e para celular. Pode-se utilizar a orientação a objetos para modelar e resolver problemas em diversas áreas, como: inteligência artificial, redes neurais artificiais, computação evolutiva, computação gráfica, visão computacional, otimização, sistemas distribuídos, redes e compiladores.

Dessa forma, é muito importante a compreensão dos conceitos que estão por trás da orientação a objetos, como *classe*, *objeto*, *atributos*, *métodos*, *interface*, *herança* e *polimorfismo*.

Este livro está organizado em quatro unidades divididas da seguinte forma:

- A Unidade 1 apresenta os conceitos básicos do desenvolvimento orientado a objetos, trazendo exemplos práticos das ferramentas Alice e Greenfoot.
- A Unidade 2 explora as estruturas de controle como estruturas de decisão e repetição, além de explorar os construtores de classes, sobrecarga e sobrescrita de métodos, herança e polimorfismo.
- A Unidade 3 define os conceitos de classes abstratas, exceções e interfaces, trazendo sempre exemplos práticos, além de ensinar a construir aplicações com interface gráfica.
- A Unidade 4, por fim, mostra exemplos práticos de aplicações orientadas a objetos que envolvem o uso de arrays, strings, banco de dados e threads.

O domínio desta disciplina é uma das principais formas de entrada no mercado de desenvolvimento de software. A programação orientada a objetos é a principal filosofia de desenvolvimento de código existente no mercado atualmente.

Dedique-se a seu estudo e você rapidamente perceberá a grande importância do tema e o motivo pelo qual esse paradigma tornou-se tão popular nos últimos 40 anos e lidera o setor de desenvolvimento nos últimos 20 anos.

CONCEITOS BÁSICOS DE ORIENTAÇÃO A OBJETOS EM PROGRAMAÇÃO

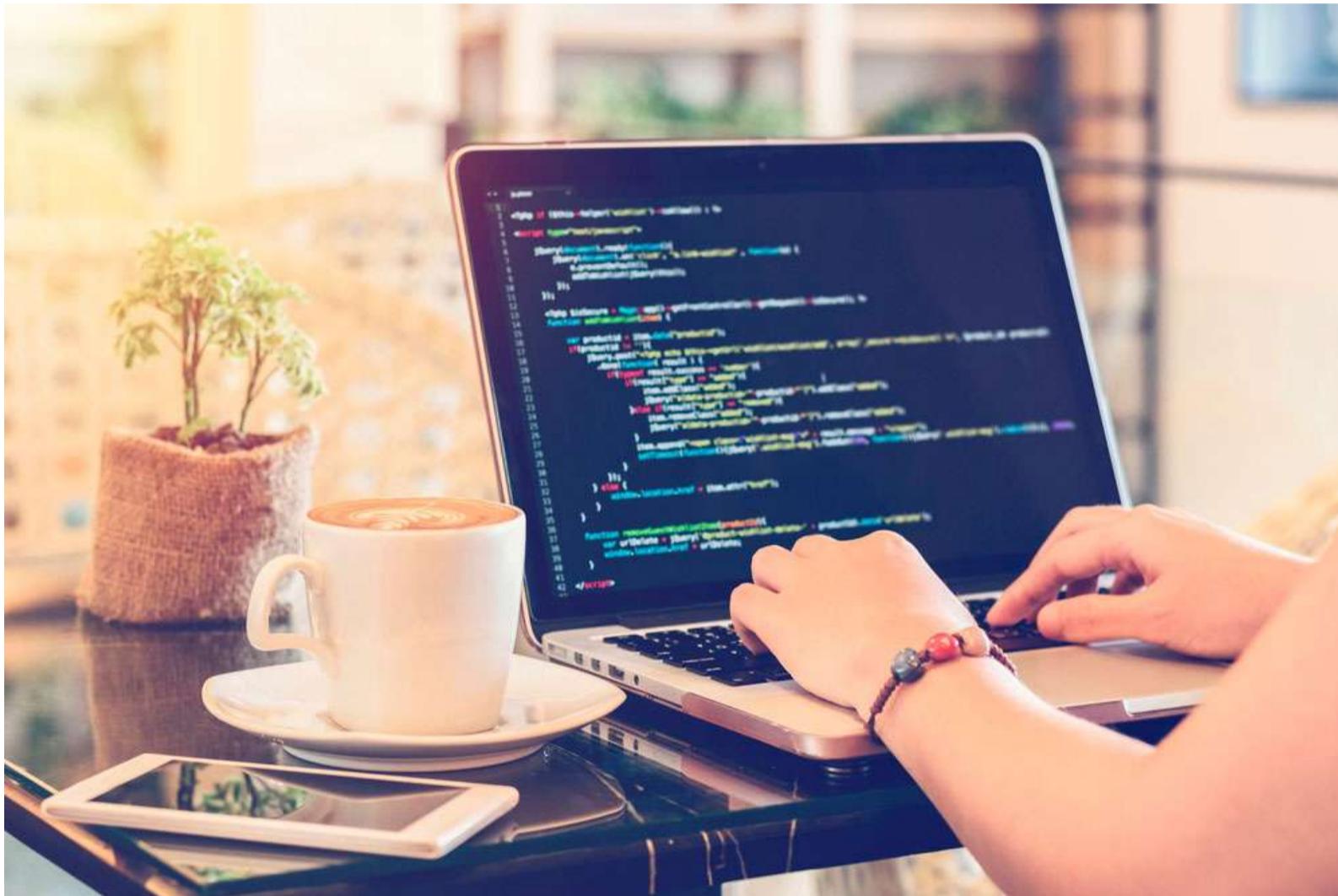
0

Jesimar da Silva Arantes

[Ver anotações](#)

O QUE É A PROGRAMAÇÃO ORIENTADA A OBJETOS?

Um paradigma de programação que aproxima a manipulação das estruturas de um programa ao manuseio das coisas do mundo real.



Fonte: Shutterstock.

Deseja ouvir este material?

Áudio disponível no material digital.

CONVITE AO ESTUDO

Prezado aluno, bem-vindo à primeira unidade do livro de Linguagem Orientada a Objetos. Nesta unidade, os conceitos e fundamentos da orientação a objetos (OO) serão apresentados. Vamos estabelecer as ideias que estão por trás das classes e dos objetos. Os conceitos de atributos e métodos de uma classe também serão elucidados, permitindo que você, aluno, seja capaz de analisar um problema do mundo real e consiga fazer as primeiras modelagens dentro do universo da OO. Você terá, também, o primeiro contato com o ambiente de desenvolvimento para fazer os primeiros programas. Ao final da unidade você

poderá desenvolver a primeira aplicação em Java com classes, objetos, atributos e métodos. Você aprenderá, ainda, a desenvolver as primeiras aplicações com as ferramentas Alice e Greenfoot, que são fantásticas e alcançam resultados bem legais com pouco tempo de uso.

0

[Ver anotações](#)

o

Ver anotações

Através de exemplos de aplicações práticas e atuais, você vai aprender a pensar a orientação a objetos. O livro apresenta uma série de ilustrações que o auxiliam a compreender o papel de cada elemento dentro da OO. A reflexão é incentivada ao longo do texto de forma que você se sinta convidado a participar também da modelagem do problema. A capacidade de abstração será treinada de forma que consigamos efetuar uma boa modelagem de diversas aplicações do mundo real.

Esta unidade vai ajudá-lo a ter uma visão geral da OO e da construção de aplicações em Java. Ela está dividida nas seguintes seções:

- A Seção 1.1 traz os conceitos básicos do paradigma orientado a objetos.
- A Seção 1.2 apresenta uma introdução a aplicações OO construídas com utilização do software Alice.
- A Seção 1.3 mostra uma visão de game em programação orientada a objetos com utilização do software Greenfoot.

Por ser uma unidade bastante prática, você, a partir de agora, passará a colocar a mão na massa com a instalação dos softwares aqui apresentados e o desenvolvimento das atividades propostas. A prática é a principal forma de consolidar os conhecimentos em qualquer linguagem de programação – e isso não é diferente com o Java. Vamos lá?

PRATICAR PARA APRENDER

Caro estudante, começamos aqui a primeira seção dos estudos sobre desenvolvimento de softwares orientado a objetos. Quem nunca se perguntou como podemos construir um robô inteligente? Talvez, uma questão ainda mais importante para os profissionais da computação seja “de que forma podemos modelar um robô que execute comportamentos inteligentes, como mover-se autonomamente, dentro de um simulador em um computador?”.

Pois bem, a ideia-chave aqui é como fazer essa modelagem do robô. E, nesse sentido, a orientação a objetos (OO) fornece uma forma poderosa de organizar o código para solucionar essa questão. Nesta seção você terá a oportunidade de aprender os conceitos e as ideias por trás da OO, como classes, objetos, atributos e métodos. Esses conceitos auxiliarão de forma direta na modelagem não só de robôs, mas também de diversos problemas presentes no mercado de trabalho.

o

Ver anotações

Para contextualizar sua aprendizagem, imagine que você foi aprovado como estagiário em uma startup que atua no ramo de alta tecnologia. Essa startup presta serviços para diversas empresas diferentes. Recentemente, seu chefe fechou um projeto com um grande portal de e-commerce (portal de venda pela internet). Uma vez que esse portal efetua as vendas de diversos produtos distintos, e uma vez que o volume de vendas dele é muito grande, é necessário um sistema automatizado para auxiliar no transporte dos produtos comprados. O seu chefe, inicialmente, propôs desenvolver um robô móvel inteligente para efetuar o transporte autonomamente dos produtos vendidos até uma determinada área destinada ao setor de carregamento da mercadoria.

Diante desse grande desafio, o seu chefe escalou você para trabalhar com uma equipe multidisciplinar a fim de resolver esse problema. A equipe com que você trabalha é composta, principalmente, por: engenheiros, cientistas da computação, profissionais de marketing e equipe de vendas. O seu chefe designou a equipe para desenvolver as estruturas de dados do sistema robótico e a demonstração da aplicação, dando apoio ao marketing e à equipe de vendas, enquanto o produto físico (hardware do robô) está em desenvolvimento pelos engenheiros. Essa demonstração da aplicação será feita por um simulador para esse robô móvel inteligente que demonstre as funcionalidades deste. Foi solicitado a você desenvolver esse simulador em Java com a utilização das boas práticas da OO. Será que a orientação a objetos o ajudará na construção desse simulador? Ou será que a programação procedural é mais adequada para a construção dele?

Agora que você foi apresentado ao desafio, como você vai modelar o seu robô? Que estruturas de dados e abstrações você vai criar? Quais comportamentos e ações o seu robô precisa executar a fim de ajudar a equipe de vendas? Que ferramentas poderão ser utilizadas para o desenvolvimento do simulador? Como você vai organizar, documentar e divulgar o código? Este livro vai auxiliá-lo a responder perguntas como essas.

Muito bem, agora que você já foi apresentado à situação-problema a ser resolvida, estude esta seção e compreenda como a OO pode ajudá-lo a modelar classes, construir objetos, definir atributos e métodos. Esses elementos são a base de OO e fornecem a estrutura fundamental para a construção de um simulador como o que foi solicitado a você. Vamos compreender a OO para poder resolver esse desafio?

Ótimo estudo!

CONCEITOS DA PROGRAMAÇÃO ORIENTADA A OBJETOS (POO)

A programação orientada a objetos (POO) é um paradigma de programação que utiliza abstrações para representar modelos baseados no mundo real. Uma forma de entender a orientação a objetos (OO) é pensar nela como uma evolução da programação procedural.

A programação procedural é uma forma de desenvolvimento de código que se caracteriza por utilizar, principalmente, funções e procedimentos como núcleo de organização estrutural.

Atualmente, existem diversos paradigmas de programação. Um paradigma nada mais é do que uma filosofia que guia todo o desenvolvimento de código. Os paradigmas de programação podem ser divididos em duas vertentes principais, que são: imperativo e declarativo.

Imperativo: a programação imperativa declara comandos dizendo exatamente como o código deve executar algo.

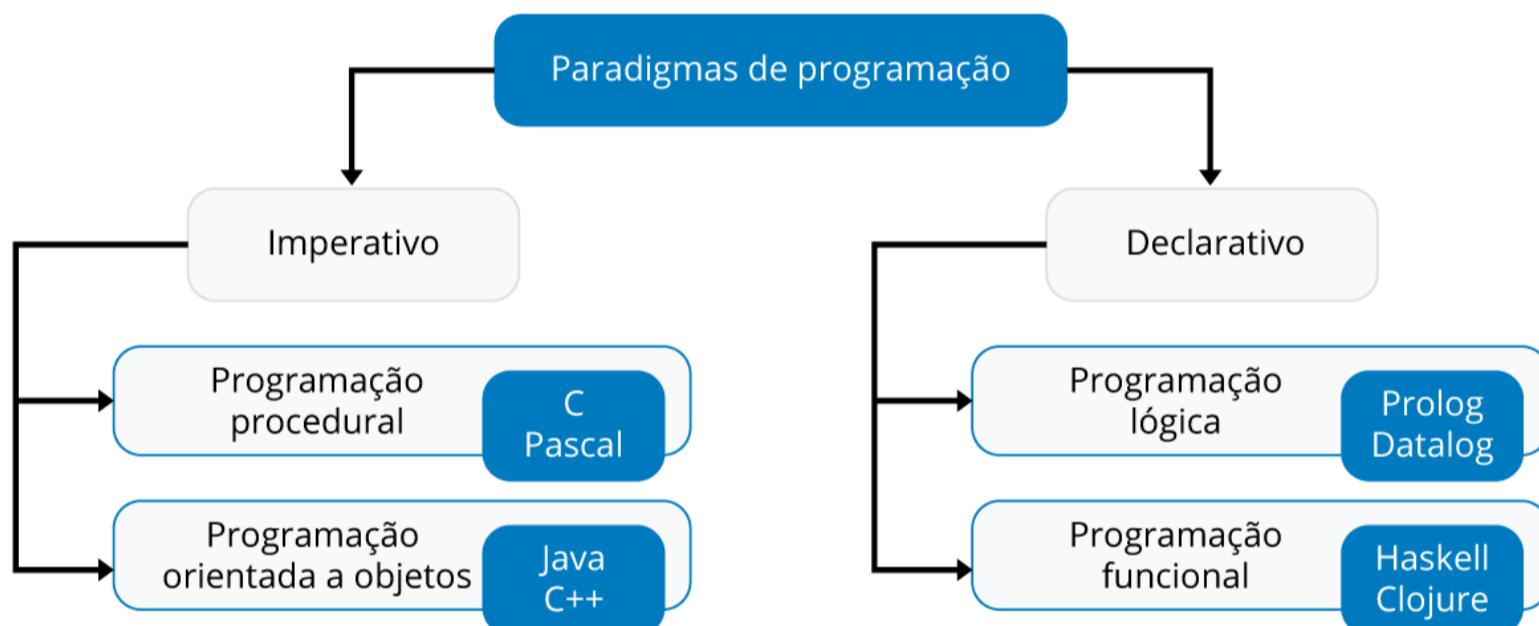
Declarativo: por sua vez, a programação declarativa descreve o que deve ser feito, mas não como deve ser feito.

A Figura 1.1 mostra um organograma que esquematiza os principais paradigmas de programação.

Podemos subdividir ainda mais o paradigma imperativo em: programação procedural e programação orientada a objetos.

Dois exemplos de linguagem orientadas a objetos são Java e C++. De forma semelhante, quando programamos em C ou Pascal estamos seguindo a estrutura de programação procedural. O paradigma declarativo pode ser subdividido em programações lógica e funcional, formas de programação que são objetos de estudo em cursos específicos, os quais fogem ao escopo deste material.

Figura 1.1 | Organograma dos principais paradigmas de programação



Fonte: elaborada pelo autor.

Embora este livro focalize os conceitos e modelos de desenvolvimento OO na linguagem de Java, os exemplos aqui apresentados podem ser facilmente adaptados para outras linguagens OO, como C++ e Python.

OS QUATRO PILARES DA ORIENTAÇÃO A OBJETOS: CLASSE, OBJETO, ATRIBUTO E MÉTODO

CLASSE E OBJETO

Um dos elementos mais básicos da OO é a ideia de organizar o código em classes que representam algo do mundo real ou imaginário. Um outro elemento básico dentro da OO é a ideia de objeto, que nada mais é do que a manifestação concreta, também chamado de instância, da classe.

A fim de entender melhor o conceito de classe, vamos pensar na entidade *robô*. Caso você tenha apenas imaginado um robô abstrato, sem forma física, que é uma máquina capaz de tomar decisões por si mesma, então você agiu corretamente. Se você imaginou um robô físico, como um robô na cor azul que anda sobre esteira, com dois braços e uma antena, então você não pensou no conceito de robô, mas sim em uma manifestação física possível de robô.

o

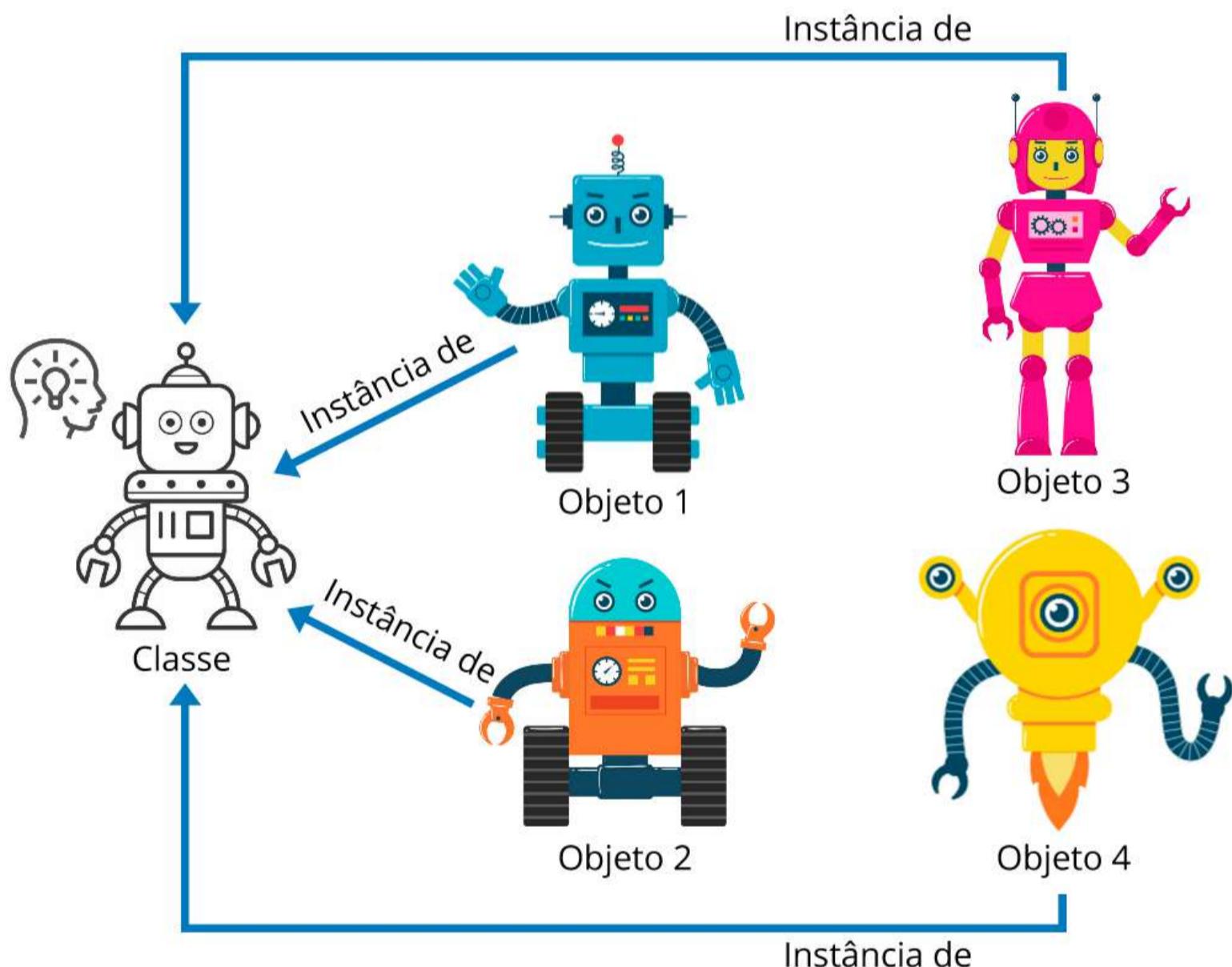
[Ver anotações](#)

Lembre-se: a ideia de robô é abstrata, assim como são as classes em Java.

Uma boa forma de compreender uma classe é pensar nela como o projeto ou a modelagem de algo. Caso tenha imaginado algo concreto, então você pensou em um objeto, que é uma manifestação possível da ideia de robô, semelhantemente à ideia de objeto em Java.

Ainda com base nesse exemplo, considere a Figura 1.2, que ilustra o relacionamento entre a classe robô, que é apenas uma ideia ou projeto, e os objetos robôs, que são manifestações da ideia de robô.

Figura 1.2 | Relacionamento entre classe e objeto



Fonte: elaborada pelo autor.

Repare que temos quatro objetos do tipo robô:

- O primeiro (objeto 1), azul, com dois braços, possui tração utilizando esteira e uma antena na cabeça.
- O segundo (objeto 2), alaranjado, tem dois braços, possui esteira e não possui antena.

- O terceiro (objeto 3), cor-de-rosa, tem dois braços, possui tração utilizando pernas e possui antena.
 - O quarto (objeto 4), amarelo, tem dois braços, possui tração com utilização de foguete e não possui antena.
-

Podemos perceber que todos os objetos modelam de forma diferente a entidade robô – em computação, dizemos que esses objetos são instâncias da classe robô.

ATRIBUTO

Embora os conceitos classe e objeto já tenham sido estabelecidos anteriormente, eles ainda estão muito abstratos, pois não definimos do que é composta uma classe nem como funcionam os objetos.

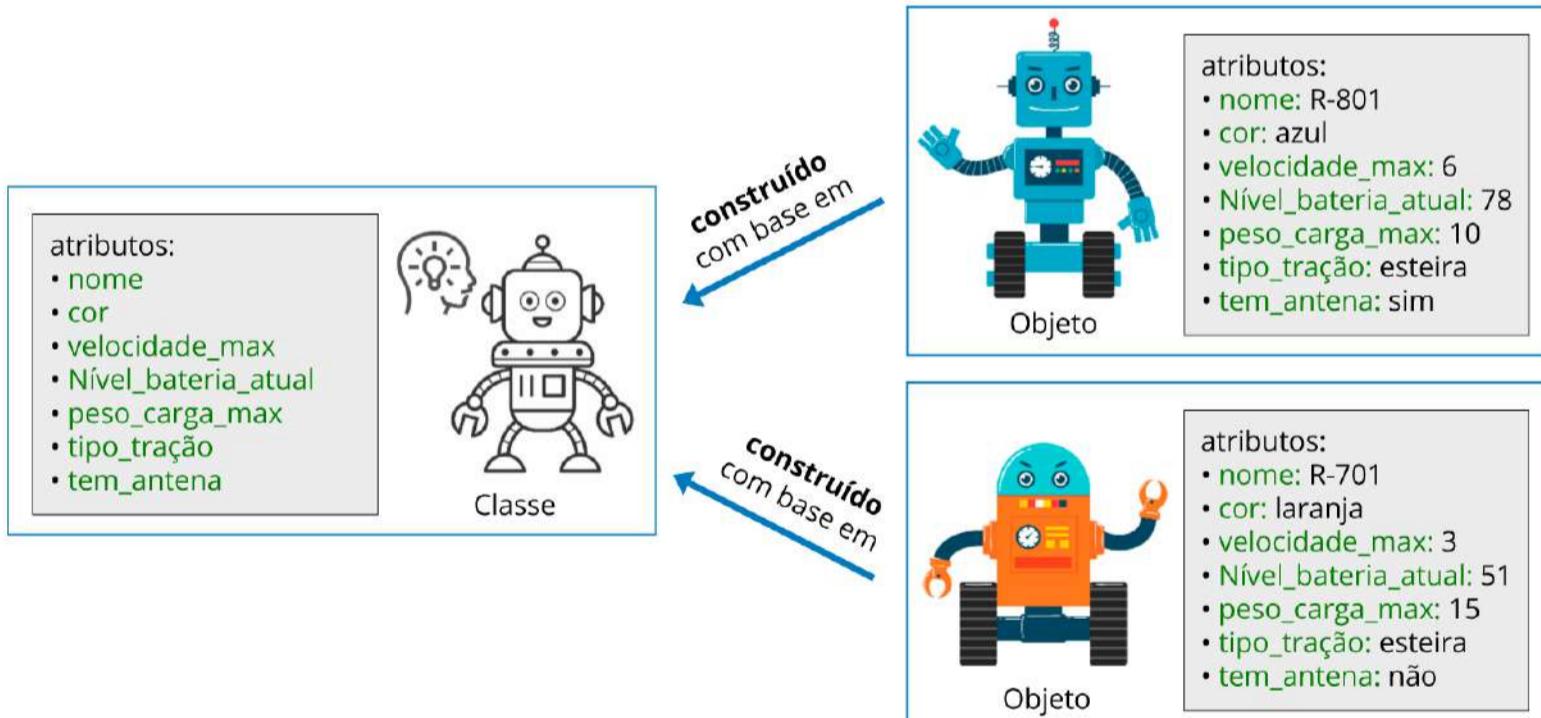
Os principais elementos que compõem uma classe são os atributos e os métodos.

Um atributo é um elemento que representa as características intrínsecas da classe.

Vamos, então, retornar à nossa ideia de robô: suponhamos que, para caracterizarmos alguns robôs, os seguintes atributos possam ser criados: nome, cor, velocidade máxima, nível atual da bateria, peso da carga máxima suportada, tipo de tração, e presença ou não de antena.

Analise a Figura 1.3, em que são acrescentados, dentro da estrutura de classe e objeto, os atributos mencionados.

Figura 1.3 | Relacionamento entre classe, objeto e atributo



Fonte: elaborada pelo autor.

Essa figura mostra que:

- O primeiro robô (acima) foi construído com base na classe robô e foram-lhe atribuídos o nome R-801, a cor azul, a velocidade de 6 metros por segundo, o nível atual da bateria em 78%, o peso máximo de carga suportado de 10 kg, o tipo de tração (esteira) e a presença antena.

- O segundo robô (abaixo) foi construído também com base na classe robô, ao qual foram atribuídos o nome R-701, a cor laranja, a velocidade de 3 metros por segundo, o nível da bateria em 51%, o peso da carga de 15 kg, o tipo de tração (esteira) e a não presença de antena.

0

[Ver anotações](#)

REFLITA

Quais são os valores para os atributos *nome*, *cor*, *velocidade máxima*, *nível atual da bateria*, *peso da carga máxima suportado*, *tipo de tração* e *presença ou não de antena* que o leitor atribuiria aos robôs da Figura 1.2 (objeto 3 e objeto 4)? Dê valores para esses atributos conforme os apresentados pela Figura 1.3. Reflita também sobre quais tipos de dados (inteiro, real, literal ou lógico) são utilizados em cada um dos atributos mencionados.

0

[Ver anotações](#)

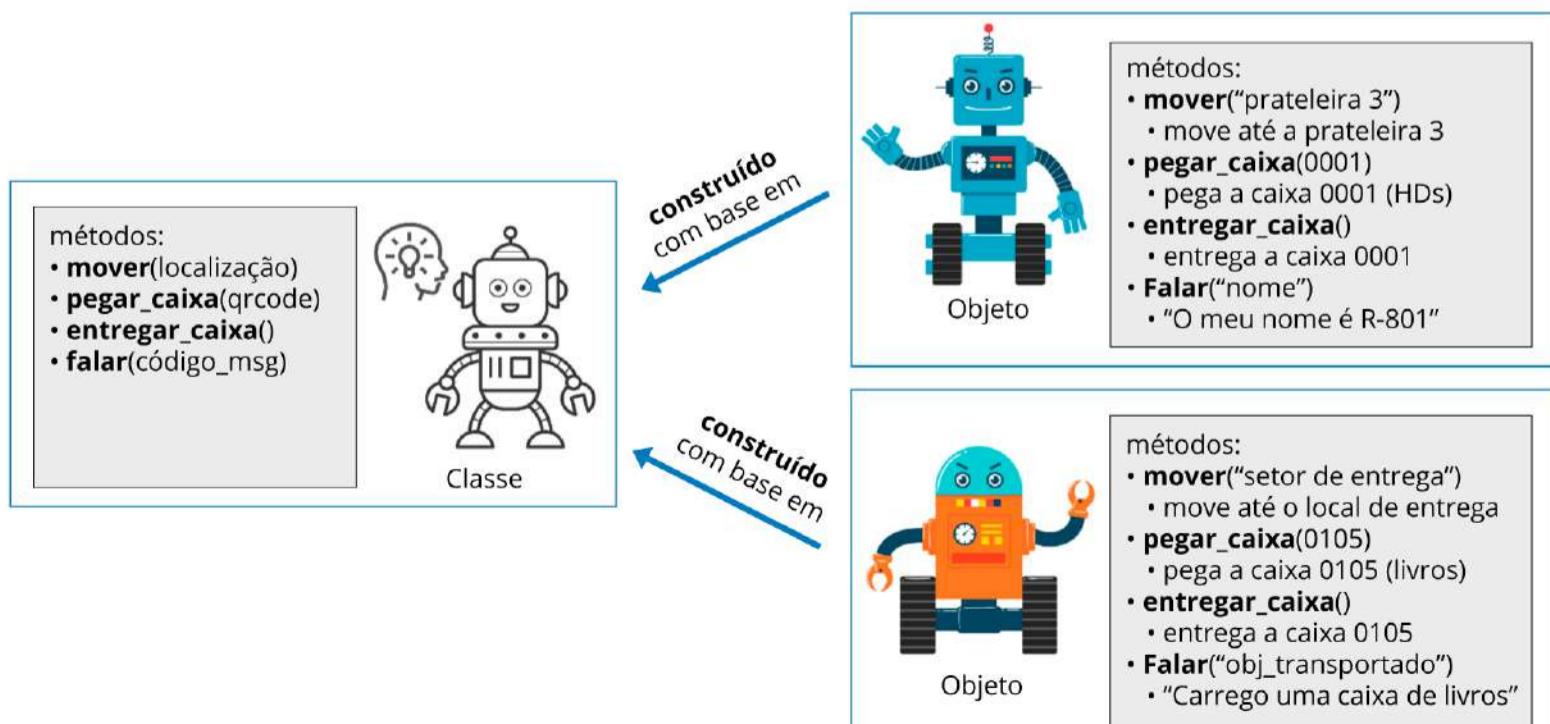
A ideia de classe, até onde foi apresentada aqui, é semelhante à ideia de registro (*struct*) na linguagem C, uma vez que esses atributos da classe são semelhantes aos campos (variáveis) de um registro. Os diversos campos de um registro são acessados por um único ponteiro, assim como os atributos da classe são acessados por um único objeto.

MÉTODO

Um outro elemento importante de uma classe são os métodos. Um método dá ao objeto da classe a capacidade de executar algum tipo de ação, comportamento ou processamento. Um robô é capaz de executar uma série de ações – o nosso robô modelado é capaz de se mover, pegar caixas, entregar as caixas e até mesmo falar.

A Figura 1.4 mostra como o conceito de método está relacionado aos conceitos de classe e objeto.

Figura 1.4 | Relacionamento entre classe, objeto e método



Vamos imaginar que o método *mover o robô R-801* foi chamado para a localização *prateleira 3*. Dessa forma, o robô executa a ação de se mover até o local indicado. De forma semelhante, imagine que o método *falar do robô R-801* foi invocado com o argumento *nome*: nesse caso, a ação executada é dizer o próprio nome, razão pela qual o robô falará “O meu nome é R-801”.

o

[Ver anotações](#)

Imagine agora que o método *falar do robô R-701* foi chamado a ser executado com o argumento *objeto transportado*, dessa maneira, o robô dirá “Carrego uma caixa de livros”. Até esse ponto, peço que você abstraia o funcionamento interno dos métodos, assumindo, assim, que eles são comportamentos de altíssimo nível. Ao longo do livro vamos incorporar mais elementos aos métodos, tornando-os capazes de realizar operações semelhantes às imaginadas aqui.

o

Ver anotações

ASSIMILE

A criação de métodos em linguagens orientadas a objetos é semelhante à criação de funções em linguagens procedurais. Uma função nas linguagens procedurais apenas processa entradas em saídas. Por sua vez, um método pode ser entendido como uma função particular que dita o comportamento de todos os objetos daquela classe, normalmente interagindo com os seus atributos, modificando-os, utilizando-os ou apenas retornando o valor de um deles.

CRIAÇÃO DE APLICAÇÕES EM JAVA

Até aqui foram apresentados os quatro pilares da orientação a objetos: classe, objeto, atributo e método. Vamos agora ver na prática como construir nossa primeira aplicação em Java.

A fim de desenvolver os códigos, você vai precisar de um ambiente no qual possa inseri-los, como um bloco de notas ou, ainda, um ambiente de desenvolvimento integrado (IDE – do inglês *integrated development environment*).

Um IDE é um editor de códigos robusto que auxilia muito o programador durante todo o desenvolvimento, permitindo-lhe codificar, executar e debugar o código.

Existem diversos IDEs no mercado, entre os quais podemos destacar: o NetBeans, o Eclipse e o IntelliJ IDEA. Os três ambientes citados são multiplataformas, ou seja, podem ser executados em sistemas operacionais Windows, Linux e macOS. Graças ao avanço da internet e, em especial, da web, existem atualmente diversos sites em que você pode escrever o seu programa em Java. Alguns exemplos de sites que permitem a programação em Java são: jdoodle.com e onlinegdb.com.

Para esta primeira seção do livro sugerimos a utilização do ambiente de desenvolvimento on-line Jdoodle, por ter utilização simples, prática e direta. Dessa forma, abra o site www.jdoodle.com/online-java-compiler/. Depois disso, você estará em uma tela semelhante à tela da Figura 1.5, mostrada a seguir.

o

Ver anotações

Figura 1.5 | Tela inicial do ambiente de programação jdoodle.com configurado para Java



Fonte: captura de tela do jdoodle.com elaborada pelo autor.

Esse ambiente de desenvolvimento mostrado na Figura 1.5 possui alguns elementos principais, que são:

- i. A área destinada ao desenvolvimento de código.
- ii. A área de inserção de argumentos via linha de comando.
- iii. A área destinada à entrada padrão de dados.
- iv. A versão do compilador utilizado.
- v. Um botão para iniciar a execução do código.
- vi. A área em que é impressa a saída do código após executado.

Observe, na Figura 1.5, o primeiro código em Java. Vamos destacar esse código a seguir para discutirmos cada um dos elementos principais dele.

Código 1.1 | Programa em Java que imprime uma mensagem de boas-vindas

```
1 public class MinhaPrimeiraClasse {  
2     public static void main(String args[]) {  
3         System.out.println("Bem-vindo, a aula de POO com Java");  
4     }  
5 }
```

Fonte: elaborado pelo autor.

O Código 1.1 define uma classe pública chamada `MinhaPrimeiraClasse` (linha 1). As palavras reservadas `public` e `class` foram utilizadas para fazer essa definição da classe. O modificador de acesso `public` será estudado nas seções seguintes. Em seguida, é aberta uma estrutura de blocos, de forma semelhante à linguagem C,

o

Ver anotações

utilizando as chaves “{” e “}”. Dentro desse bloco, há uma função pública que não retorna nenhum valor (void) chamada main (linha 2). Em Java, quando há um método estático, às vezes, chamamo-lo de função, o que ficará mais claro nas seções seguintes. Essa função main recebe um vetor de strings como argumento. Em seguida, há mais um bloco de chaves que define o escopo da função main. Por fim, temos uma linha de código que invoca uma função (método estático), chamada System.out.println, para imprimir uma mensagem na tela (linha 3). A mensagem de boas-vindas é passada como argumento dessa função entre aspas por ser do tipo literal.

Ao executar esse código, simplesmente é impresso na saída padrão a mensagem “Bem-vindo à aula de POO com Java”.

O código a seguir, em Java, mostra como é o formato da função main que é o ponto de entrada de toda aplicação Java, ou seja, é por meio dela que a sua aplicação iniciará a execução.

```
1 public static void main(String[] args) {  
2     //conjunto de comandos da função principal.  
3 }
```

O código a seguir, em C, mostra como é o formato da função main que é o ponto de entrada de toda aplicação C.

```
1 int main(int argc, char *argv[]){  
2     //conjunto de comandos da função principal.  
3     return 0;  
4 }
```

É importante reparar que existem algumas diferenças entre essas funções. A função main em C retorna um inteiro; por sua vez, a função main em Java não retorna nenhum tipo de valor (void). Em C, existem dois argumentos principais que são *argc* e *argv*. A linguagem Java sintetiza esses dois argumentos em apenas um, chamado *args*. A função principal da linguagem C pode também ser desenvolvida pela omissão dos argumentos *argc* e *argv* com a seguinte simples escrita: int main(). A linguagem Java, por sua vez, não permite que se omita o argumento *args*.

EXEMPLIFICANDO

Vamos agora modelar uma classe chamada Robô com os atributos especificados na Figura 1.3. Dessa maneira, analise o Código 1.2 apresentado.

Código 1.2 | Programa em Java que modela a classe robô e seus atributos

```
1 public class Robo {  
2     String nome;  
3     String cor;  
4     float velocidadeMax;  
5     int nivelBateriaAtual;  
6     float pesoCargaMax;  
7     String tipoTracao;  
8     boolean temAntena;  
9     public static void main(String[] args) {  
10         Robo obj1 = new Robo();  
11         obj1.nome = "R-801";  
12         obj1.cor = "azul";  
13         obj1.velocidadeMax = 6;  
14         obj1.nivelBateriaAtual = 78;  
15         obj1.pesoCargaMax = 10;  
16         obj1.tipoTracao = "esteira";  
17         obj1.temAntena = true;  
18         System.out.println("Meu nome: " + obj1.nome);  
19         System.out.println("Cor do Robô: " + obj1.cor);  
20         System.out.println("Vel Max: " + obj1.velocidadeMax);  
21         System.out.println("Bat: " + obj1.nivelBateriaAtual);  
22         System.out.println("Carga Max: " + obj1.pesoCargaMax);  
23         System.out.println("Tipo Tração: " + obj1.tipoTracao);  
24         System.out.println("Tem Antena: " + obj1.temAntena);  
25     }  
26 }
```

Fonte: elaborado pelo autor.

Implemente o Código 1.2 e analise a saída que será impressa.

o

Ver anotações

O atributo *nível da bateria*, na linha 5, possui o tipo de dados inteiro (int). O tipo *int* em Java é semelhante ao tipo *int* na linguagem C. Os atributos *velocidade máxima* e *peso da carga máxima* são do tipo float, igual ao tipo float em C. O atributo *tem antena* é do tipo lógico ou boolean. Na linguagem C não existe esse tipo de dado nativamente. Por fim, os atributos *nome*, *cor* e *tipo de tração* foram definidos como tipo String, o qual será mais bem explicado posteriormente no livro. Por ora, vale lembrar que é um tipo literal ou o equivalente a um vetor de caracteres em linguagem C. Na linha 10, há a construção do robô chamado obj1, construído a partir da classe Robo. A construção de um objeto utiliza a palavra reservada new, que será mais bem discutida nas seções seguintes. Dizemos que obj1 é uma instância de Robo.

Repare que todos os atributos podem ser acessados com a utilização do ponto, como em obj1.nome (linha 11). O acesso aos atributos é semelhante à forma como a linguagem C acessa os campos (variáveis) de um registro.

LEMBRE-SE

Você estudou a forma básica de impressão de uma mensagem na tela utilizando o System.out.println, que imprime e pula de linha. Você também pode utilizar o comando System.out.print para fazer a impressão sem pular linhas. Existe ainda a opção System.out.printf que recebe dois argumentos: o primeiro, um literal, para formatação da impressão, e, outro, uma lista com os objetos a serem impressos. O System.out.printf é semelhante ao comando printf em C. Um operador muito comum utilizado na concatenação de literais (Strings) em Java é o operador '+'. Dessa forma, ao fazer "olá" + "mundo", estaremos concatenando o valor de "olá" com o conteúdo de "mundo" e então retornamos o literal "olá mundo".

Uma melhoria que pode ser feita no Código 1.2 é criar um método que imprime o status da classe robô em vez de deixar isso a cargo da função main. Esse tipo de modificação deixa na classe robô a responsabilidade de imprimir o seu status. Essa melhoria é mostrada no Código 1.3.

Código 1.3 | Programa em Java que modela a classe robô e seus atributos e um método

0

[Ver anotações](#)

```
1 public class Robo {  
2     String nome;  
3     String cor;  
4     float velocidadeMax;  
5     int nivelBateriaAtual;  
6     float pesoCargaMax;  
7     String tipoTracao;  
8     boolean temAntena;  
9     public void printStatus(){  
10        System.out.println("-----");  
11        System.out.println("Meu nome: " + nome);  
12        System.out.println("Cor do Robô: " + cor);  
13        System.out.println("Velocidade Max: " + velocidadeMax);  
14        System.out.println("Bateria: " + nivelBateriaAtual);  
15        System.out.println("Carga Max: " + pesoCargaMax);  
16        System.out.println("Tipo de Tração: " + tipoTracao);  
17        System.out.println("Tem Antena: " + temAntena);  
18        System.out.println("-----");  
19    }  
20    public static void main(String[] args) {  
21        Robo objeto1 = new Robo();  
22        objeto1.nome = "R-801";  
23        objeto1.cor = "azul";  
24        objeto1.velocidadeMax = 6;  
25        objeto1.nivelBateriaAtual = 78;  
26        objeto1.pesoCargaMax = 10;  
27        objeto1.tipoTracao = "esteira";  
28        objeto1.temAntena = true;  
29        objeto1.printStatus();  
30        Robo objeto2 = new Robo();  
31        objeto2.nome = "R-701";  
32        objeto2.cor = "laranja";  
33        objeto2.velocidadeMax = 3;  
34        objeto2.nivelBateriaAtual = 51;  
35        objeto2.pesoCargaMax = 15;  
36        objeto2.tipoTracao = "esteira";  
37        objeto2.temAntena = false;  
38        objeto2.printStatus();  
39    }  
40 }
```

Ver anotações

Repare que agora criamos dois objetos, os chamados objeto1 (linha 21) e objeto2 (linha 30), que representam os robôs T-801 e T-701, respectivamente.

Um outro ponto a ser destacado aqui é a forma pela qual chamamos o método printStatus, nas linhas 29 e 38, também utilizando o ponto como se fosse um atributo da classe robô.

o

Ver anotações

DICA

As plataformas de hospedagem de código github e gitlab são dois grandes repositórios que milhões de desenvolvedores no mundo utilizam para fazer a divulgação de seus códigos. Além de auxiliar na divulgação dos códigos, esses ambientes possibilitam fazer o gerenciamento do controle de versão utilizando a ferramenta git. Pesquise mais sobre: Github (2020), Gitlab (2020) e Git (2020). Este livro divulga os todos seus códigos no Github, que pode ser acessado em Arantes (2020).

Caro estudante, nesta seção você estudou os conceitos de classe, objeto, atributos e métodos. Além disso, aprendeu a construir a sua primeira aplicação Java. Em seguida, aprendeu a declarar uma classe que modela a entidade robô. Por isso, criou dois objetos da classe, que são os robôs R-801 e R-701. Foi mostrada também a estrutura básica dos atributos e métodos da classe robô. Nas seções seguintes, vamos nos aprofundar cada vez mais nesse universo do desenvolvimento orientado a objetos.

REFERÊNCIAS

ARANTES, J. da S. **Desenvolvimento orientado a objetos**. Github: on-line, [2020]. Disponível em: <https://bit.ly/3eiUMcF> . Acesso em: 20 abr. 2020.

BHATNAGAR, A. The complete History of Java Programming Language. **Geeks for geeks**, 2019. Disponível em: <https://bit.ly/2BXvfJ9> . Acesso em: 30 abr. 2020.

CAELUM (org.). **Java e orientação a objetos**. Apostila do Curso FJ-11. Disponível em: <https://bit.ly/3ijX34d> . Acesso em: 17 abr. 2020.

DEITEL, P. J; DEITEL, H. M. **Java: como programar**. 10. ed. São Paulo: Pearson Education, 2016.

GEEKSFORGEEKS. **History of Java**. [2020]. 1 cartaz, color. Disponível em: <https://bit.ly/31YNBEc> . Acesso em: 30 abr. 2020.

GIR. Página inicial. [2020]. Disponível em: <https://bit.ly/38IXzLq> . Acesso em: 27 maio 2020.

GITHUB. Página inicial. [2020]. Disponível em: <https://bit.ly/2ZSr1ud> . Acesso em: 27 maio 2020.

GITLAB. Página inicial. [2020]. Disponível em: <https://bit.ly/2ZNOMNG> . Acesso em: 27 maio 2020.

JAVA version history. In: Wikipedia: the free encyclopedia. [San Francisco, CA: Wikimedia Foundation, 2020]. Disponível em: <https://bit.ly/2W5b96q> . Acesso em: 14 abr. 2020.

JDOODLE. Disponível em: <https://bit.ly/2Zd2qky> . Acesso em: 16 abr. 2020.

TIOBE. TIOBE index for may 2020. [2020]. Disponível em: <https://bit.ly/2WjGJhd> . Acesso em: 27 maio 2020.

FOCO NO MERCADO DE TRABALHO

CONCEITOS BÁSICOS DE ORIENTAÇÃO A OBJETOS EM PROGRAMAÇÃO

Jesimar da Silva Arantes

Ver anotações

MODELANDO O ROBÔ R-ATM INTELIGENTE

Utilizando a programação orientada a objetos para organizar os códigos e implementar soluções no departamento de logística do cliente.



Fonte: Shutterstock.

Deseja ouvir este material?

Áudio disponível no material digital.

SEM MEDO DE ERRAR

A *startup* em que você faz estágio está com um grande desafio na área de robótica. O seu chefe lhe havia solicitado que modelasse um simulador em Java que demostrasse as funcionalidades do robô de transporte. Você e sua equipe se reuniram e juntos decidiram chamar o robô a ser projetado de *Robô Autônomo Transporte de Mercadoria* (R-ATM).

Os projetistas do robô especificaram, por enquanto, que o R-ATM será do tipo esteira e possuirá braços robóticos para captura das caixas. O robô terá um peso de aproximadamente 70 kg, e sua velocidade de deslocamento máxima atingirá

por volta de 5 m/s. Não se sabe ainda qual será a capacidade de carga máxima do robô, mas os especialistas acreditam que ela girará em torno de 20 kg. Essa e outras especificações serão mais bem ajustadas ao longo do desenvolvimento do projeto.

Foi especificado que o robô vai operar em uma sala plana de formato, a princípio, retangular. O robô saberá, mediante tecnologia de rádio, a própria localização em tempo real. Dessa forma, o robô terá uma localização descrita em coordenadas x, y. Com base nessas especificações iniciais, o seu chefe pede que você inicie a modelagem da classe robô enquanto os engenheiros projetam o hardware dele.

Para resolver a questão da proposta, você percebe que precisa iniciar a modelagem da classe robô e precisa criá-la com os seguintes atributos:

- *nome*
- *peso*
- *velocidadeMax*
- *pesoCargaMax*
- *tipoTracao*
- *posicaoX*
- *posicaoY*

Como o seu robô pode deslocar-se pelo ambiente, você também decide criar um método para mover o robô para uma dada posição x e y. Você decide também criar um método para imprimir o status do robô. Esse método de impressão de status auxilia a verificar o estado da classe robô e prevenir erros.

Não foram especificadas, ainda, nas reuniões com a equipe as dimensões da sala em que o robô se encontra. Você então assume que é uma sala quadrada bem grande de dimensões de 100 metros por 100 metros. A fim de simplificar as coisas, você decide colocar o robô no centro da sala, ou seja, na posição (50, 50). O Código 1.4 a seguir mostra como é uma possível codificação da situação-problema aqui apresentada.

Código 1.4 | Classe em Java que modela a entidade robô da situação-problema

```
1 public class Robo {  
2     String nome = "R-ATM";  
3     float peso = 70;  
4     float velocidadeMax = 5;  
5     float pesoCargaMax = 20;  
6     String tipoTracao = "esteira";  
7     float posicaoX = 50;  
8     float posicaoY = 50;  
9     public void move(float x, float y){  
10         posicaoX = x;  
11         posicaoY = y;  
12     }  
13     public void printStatus(){  
14         System.out.println("-----Info R-ATM-----");  
15         System.out.println("Nome do Robô: " + nome);  
16         System.out.println("Peso do Robô: " + peso);  
17         System.out.println("Velocidade Max: " + velocidadeMax);  
18         System.out.println("Carga Max: " + pesoCargaMax);  
19         System.out.println("Tipo de Tração: " + tipoTracao);  
20         System.out.println("Posição X do Robô: " + posicaoX);  
21         System.out.println("Posição Y do Robô: " + posicaoY);  
22         System.out.println("-----");  
23     }  
24     public static void main(String[] args) {  
25         Robo objRobo = new Robo();  
26         objRobo.printStatus();  
27         objRobo.move(60, 50);  
28         objRobo.printStatus();  
29         objRobo.move(65, 55);  
30         objRobo.printStatus();  
31     }  
32 }
```

0

Ver anotações

Fonte: elaborado pelo autor.

No Código 1.4, após modelar a classe robô, foi colocado o método main para testar o seu funcionamento. Tendo se executado a classe robô, o robô, que inicialmente se encontrava na posição (50, 50), em seguida move-se para a posição (60, 50) e,

por fim, move-se para a posição (65, 55).

Confira as novas posições do robô impressas na tela pelo método printStatus. Até esse ponto colocamos o método estático main, ponto de entrada da aplicação, dentro da nossa classe robô. Você acha que esse método main deve ficar nessa classe? Ou parece-lhe mais adequado que ele fique em outro local?

Responderemos melhor a essa pergunta nas seções seguintes do livro.

0

[Ver anotações](#)

INTRODUÇÃO A APLICAÇÕES ORIENTADAS A OBJETOS

Jesimar da Silva Arantes

0

AS DIVERSAS APLICAÇÕES DA LINGUAGEM ORIENTADA A OBJETOS

Utilizando a linguagem orientada a objetos você pode criar animações e jogos 3D.

[Ver anotações](#)

Fonte: Shutterstock.

Deseja ouvir este material?

Áudio disponível no material digital.

PRATICAR PARA APRENDER

Caro estudante, bem-vindo à segunda seção dos estudos sobre linguagem orientada a objetos. Quem nunca se perguntou como são feitos os jogos e animações em 3D hoje em dia? É difícil dar uma resposta única para essa indagação, pois existem diversas formas de fazê-los. Uma das formas é programar o jogo ou a animação utilizando alguma linguagem de programação, como Java. Um dos elementos que podem dificultar essa tarefa é a modelagem 3D do ambiente e dos personagens, visto que é necessário ter conhecimento da área de Computação Gráfica (CG). Existem diversas ferramentas disponíveis no mercado que permitem criar jogos e animações 3D sem ter conhecimento da área de CG.

Nesta seção estudaremos uma dessas ferramentas, o Alice. Uma das vantagens do

Alice é que ele permite fazer a criação de jogos e animações de forma simples e rápida, pois utiliza uma linguagem de programação baseada em blocos. Outra vantagem do Alice é que ele auxilia no ensino dos conceitos de Orientação a Objetos (OO), devido ao seu aspecto lúdico e visual.

Você, enquanto estagiário de uma *startup*, recebe diversos desafios todos os dias. Recentemente, o seu chefe fechou um trabalho em que deverá ser desenvolvido um robô móvel inteligente. Nessa semana, o seu chefe participou de diversas reuniões com a empresa de e-commerce que o contratou. Em uma dessas reuniões, foi acordado que a sua empresa deverá apresentar uma animação 3D demonstrando algumas funcionalidades básicas do robô móvel que será projetado. Assim, o seu chefe imediatamente pensou em você para resolver esse problema. Ele, então, pediu para você fazer uma animação utilizando alguma ferramenta 3D que simule um robô andando em uma sala retangular. As principais funcionalidades que a animação deve contemplar são:

- O robô deve se mover.
- O robô deve fazer o reconhecimento do cenário.
- O robô deve sair de um local de origem e chegar até um local contendo caixas.
- O robô deve conseguir identificar os conteúdos das caixas.

Uma vez que seu desafio foi apresentado, chegou o momento de estudar esta seção, pois ela ajudará a resolvê-lo. Você aprenderá a criar animações e jogos 3D e, para isso, verá como criar objetos de classes já existentes. Você aprenderá também como posicionar os objetos no cenário, como modificar atributos e como definir comportamentos complexos que são o resultado de diversas ações em conjunto. Vamos conhecer essa ferramenta para poder resolver esse desafio?

Bom estudo!

CONCEITO-CHAVE

INTRODUÇÃO ÀS FERRAMENTAS DE PROGRAMAÇÃO ORIENTADA A OBJETO

Uma Linguagem Orientada a Objetos (OO) permite a criação de diversas aplicações. Vamos imaginar que você queira criar uma animação em um ambiente tridimensional e que esse ambiente seja composto por uma sala com alguns objetos e um personagem. Nessa animação, você deseja que o personagem criado possa interagir nesse ambiente, executando algumas ações/comportamentos.

o

Ver anotações

Uma das formas de fazer isso é desenvolver um programa utilizando alguma linguagem de programação. Conforme discutido na primeira seção, a OO junto com Java auxilia muito no desenvolvimento de programas como esse. Por ser uma animação tridimensional complexa, é necessário ao desenvolvedor conhecer a área de computação gráfica e ter um forte domínio da OO. Imagine agora que queremos fazer essa animação 3D de forma rápida e sem ter domínio da área de Computação Gráfica. Parece difícil, não é? Não se preocupe: esta seção guiará você nesse caminho, mostrando uma forma que simplifica muito esse processo.

Conheceremos a ferramenta Alice.

O software Alice é uma ferramenta que auxilia no ensino e aprendizagem de programação (DANN *et al.*, 2012). Essa ferramenta permite que programadores novos criem animações e jogos usando ambientes 3D de forma fácil, apresentando a OO de maneira bastante lúdica. Assim, convidamos você a mergulhar no software Alice, e logo perceberemos o quanto essa ferramenta é fascinante e permite compreender melhor o mundo da OO.

Nesta seção utilizaremos o software Alice 3, em específico a versão Alice 3.5.0. Acesse o site dessa ferramenta (CARNEGIE MELLON UNIVERSITY, [s.d.]), baixe e efetue sua instalação. O processo é bastante simples e por isso não será mostrado. Caso você tenha alguma dificuldade nessa fase, acesse o site da PBworks (2018), que oferece instruções de instalação. Outro ponto a ser destacado é que após a instalação do Alice pode ser necessária a instalação do *Java Development Kit* (JDK) caso você não disponha do JDK instalado. O download do JDK pode ser feito no site de download do Java (ORACLE, 2020). O Alice é uma ferramenta gratuita e multiplataforma, ou seja, executa em sistemas operacionais Windows, Linux e MacOS X.

SAIBA MAIS

O cientista da computação Randy Pausch (1960-2008) é o criador da ferramenta Alice. Randy foi um importante professor de ciência da computação na Universidade Carnegie Mellon (CMU). Ele desenvolveu importantes trabalhos na área de realidade virtual, interação homem-computador e design. A ferramenta Alice teve esse nome em homenagem ao autor Lewis Carroll (1832-1898), que escreveu *Alice no País das Maravilhas* e *Alice no País dos Espelhos*.

EXPLORANDO O ALICE

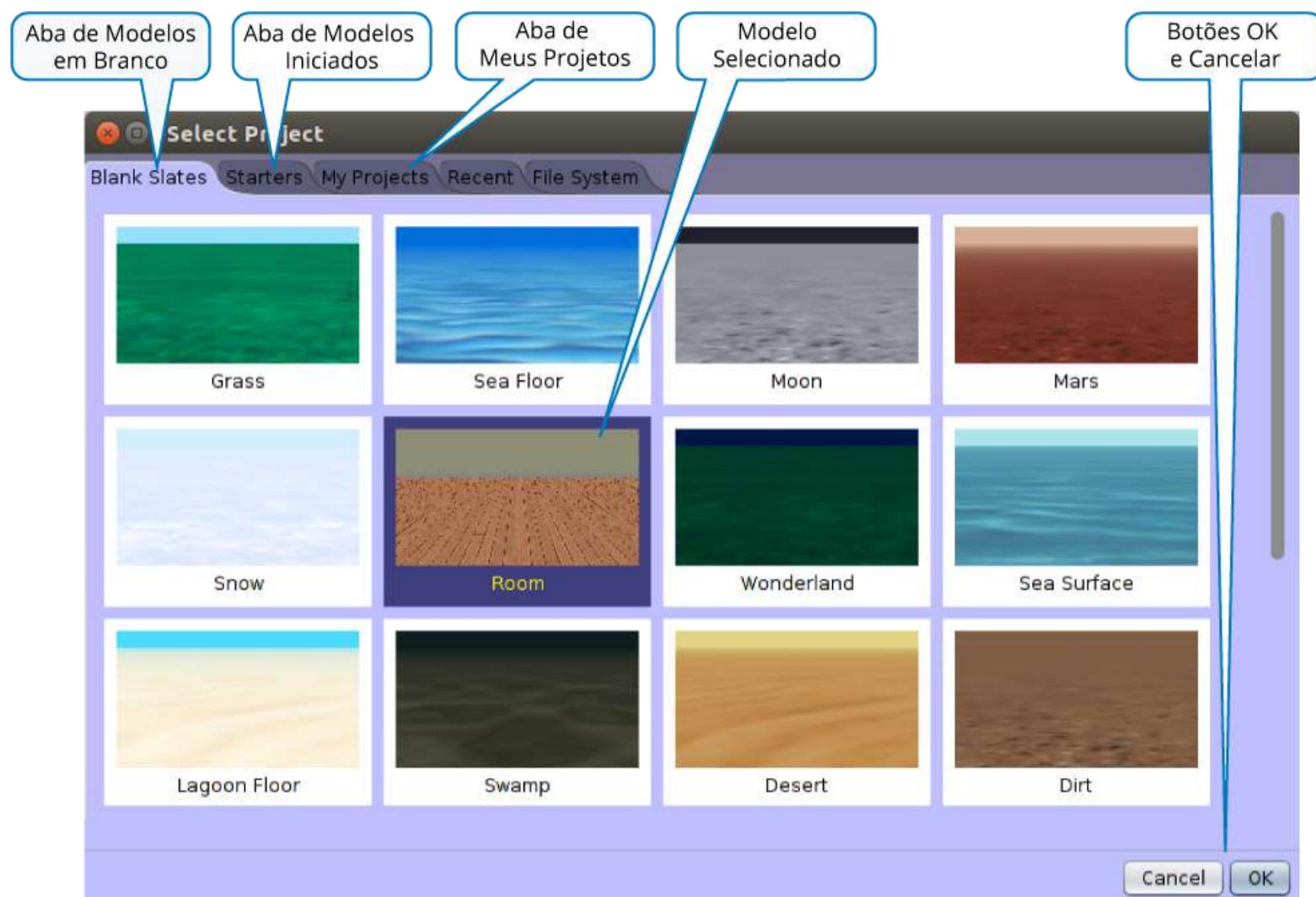
TELA DE DESENVOLVIMENTO PRINCIPAL

Ao abrir o software Alice, a primeira tela que aparece é a de seleção de projetos, conforme Figura 1.6.

0

[Ver anotações](#)

Figura 1.6 | Tela inicial de seleção de projetos do software Alice



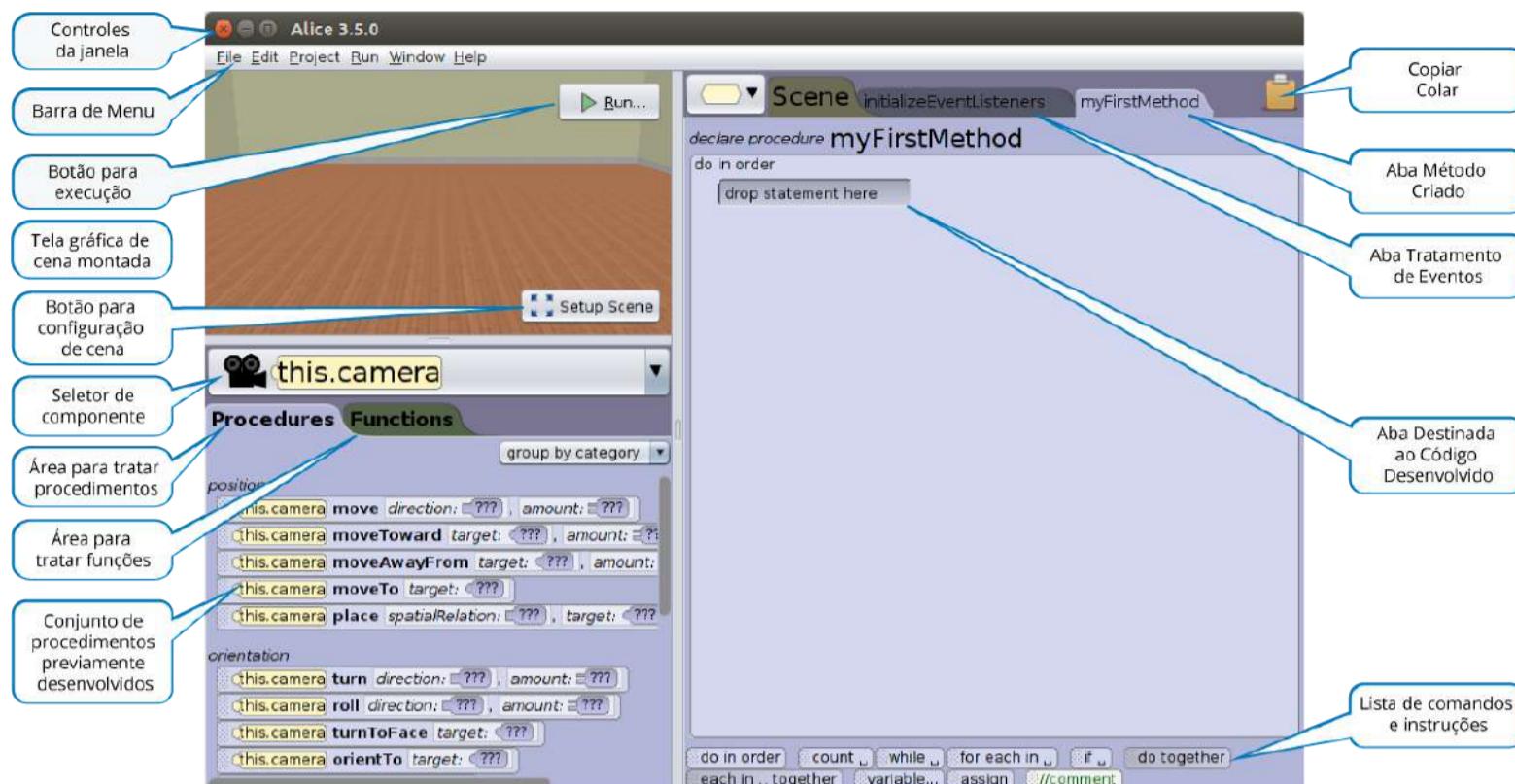
Fonte: captura de tela do software Alice elaborada pelo autor.

Nessa tela existe uma aba chamada *Blank Slates*, na qual temos disponíveis diversos modelos em branco, que são cenários de fundo mais simplificados. Na aba *Starters* existem diversos modelos iniciados, que são cenários de fundo mais complexos com diversos componentes já montados. Você pode selecionar o modelo que mais lhe agrade.

Para quem está iniciando com a ferramenta, o mais adequado é trabalhar com os modelos em branco, com menos recursos, tornando o aprendizado mais fácil. Selecione, por exemplo, o modelo *Room* e clique em *OK* para avançar.

A ferramenta Alice será então direcionada até a tela de desenvolvimento principal, também chamada tela de edição de código. Assim, pode-se ver uma tela semelhante à Figura 1.7.

Figura 1.7 | Tela de desenvolvimento principal do software Alice



Fonte: captura de tela do software Alice elaborada pelo autor.

Analise as descrições feitas na figura para conhecer os principais recursos presentes nessa janela. Alguns elementos principais disponíveis nesse ambiente são:

- A tela gráfica da cena montada.
- O botão para configuração da cena (*setup scene*).
- Os procedimentos previamente desenvolvidos.
- A área destinada ao desenvolvimento de código.

DICA

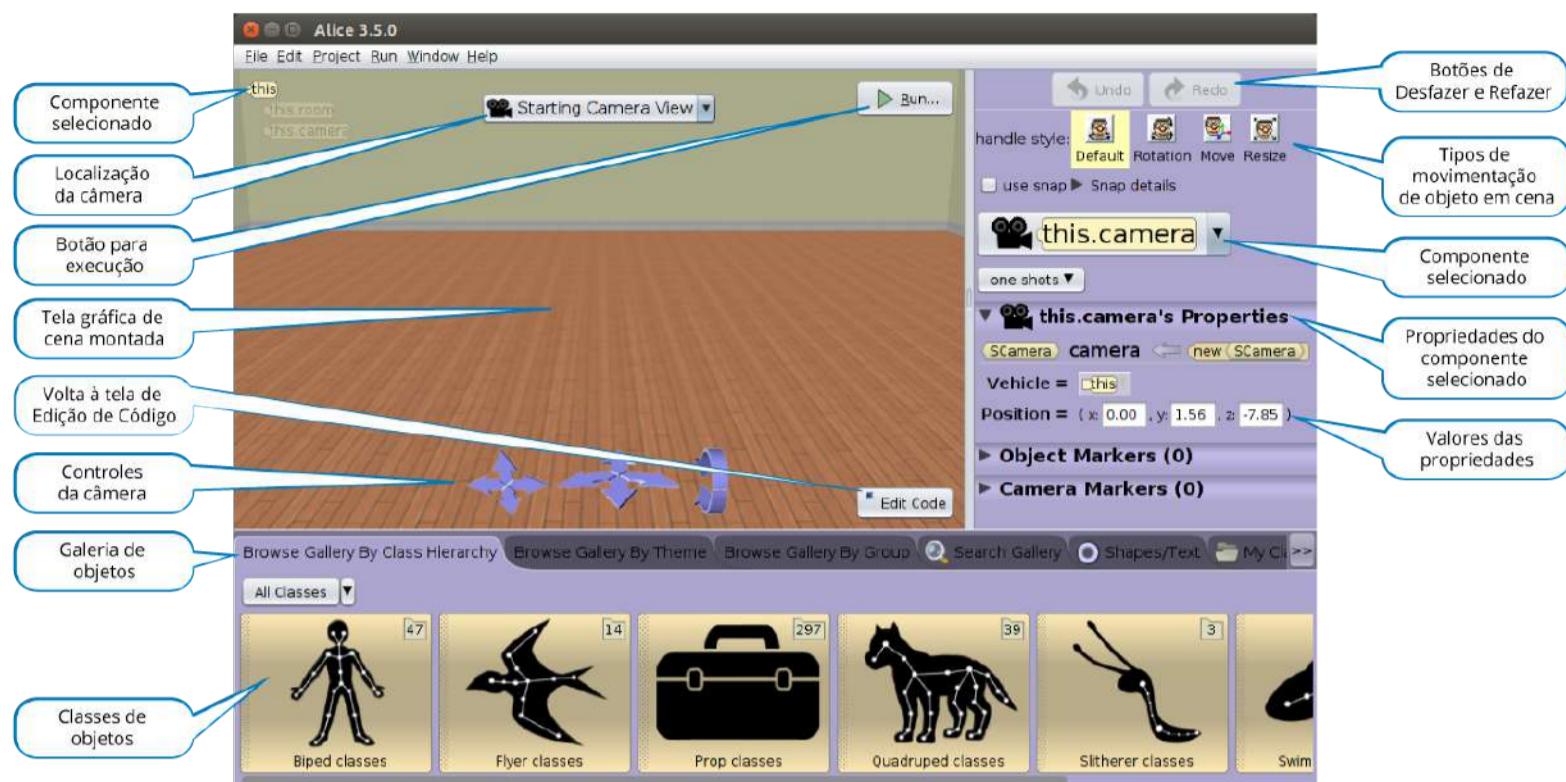
Caso você deseje, o idioma pode ser alterado para diversas línguas, inclusive o Português. Para trocar o idioma vá em menu *Window*, *Preferences*, *Locale* e então selecione o Português (Brasil).

Observação: um dos problemas de se alterar o idioma é que alguns comandos, métodos e classes utilizados na programação são também alterados (traduzidos), assim a familiaridade dos comandos da linguagem Java é reduzida.

| TELA DE E CONFIGURAÇÃO DA CENA

Uma vez conhecida a tela de desenvolvimento principal, vamos, agora, conhecer a tela de montagem e configuração da cena. Dessa maneira, clique em *setup scene*, e uma tela semelhante à Figura 1.8 será mostrada.

Figura 1.8 | Tela de montagem/criação da cena no software Alice



Fonte: captura de tela do software Alice elaborada pelo autor.

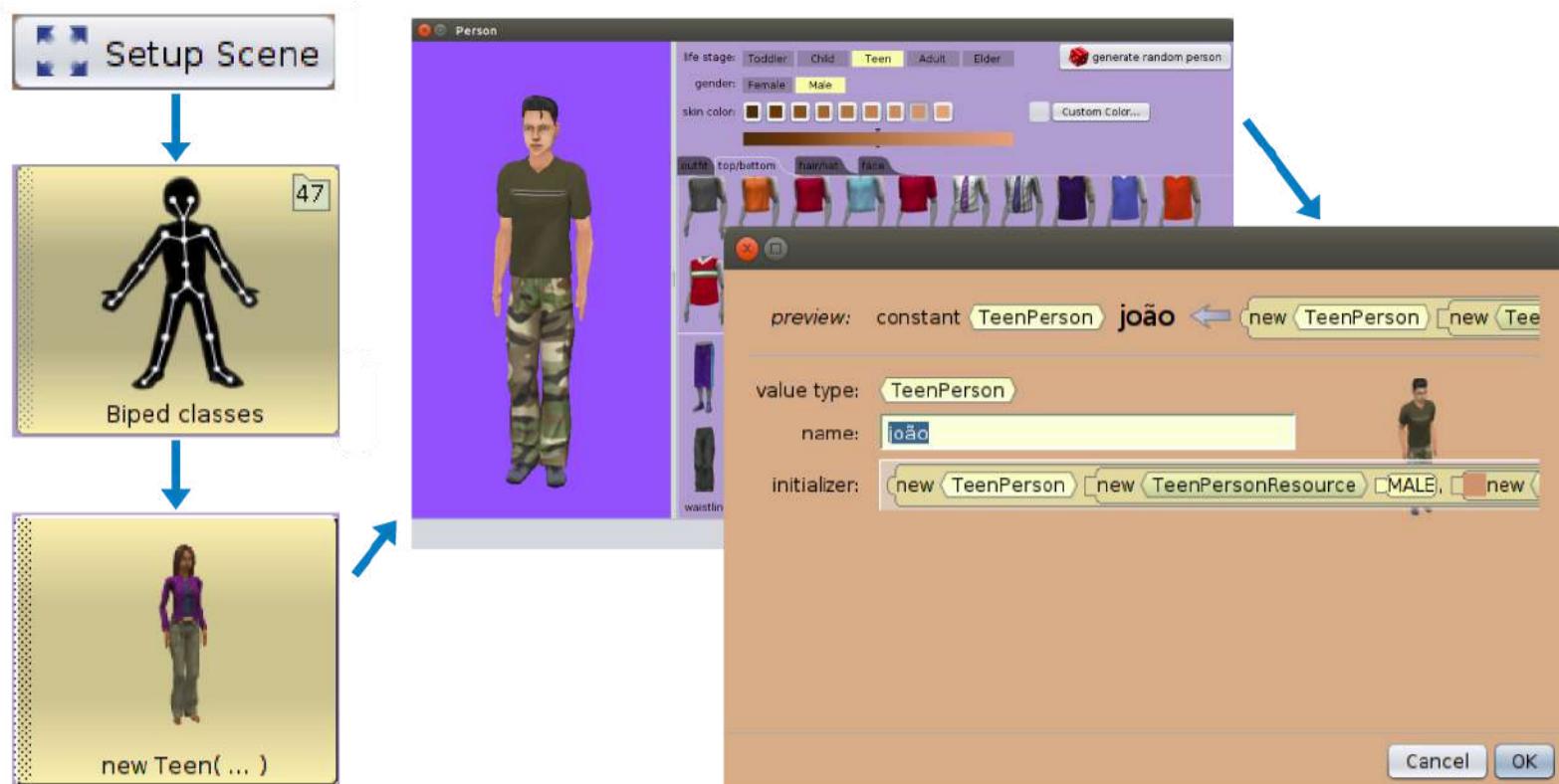
Analise as descrições feitas na figura para conhecer os principais recursos presentes nessa janela. Entre os elementos principais presentes nessa tela, temos:

- O componente que está selecionado.
- O botão para iniciar a execução.
- As classes de objetos disponíveis.
- As propriedades do componente selecionado.

ESCREVENDO UM PROGRAMA NO ALICE

Uma vez que foi dada uma visão geral do software, agora vamos montar um cenário em que possamos escrever o nosso primeiro programa com o Alice. Nesse sentido, o primeiro passo é fazer a criação de um personagem e então inseri-lo na cena. A Figura 1.9 mostra alguns passos necessários para isso.

Figura 1.9 | Passos para criação/inserção de um personagem na cena



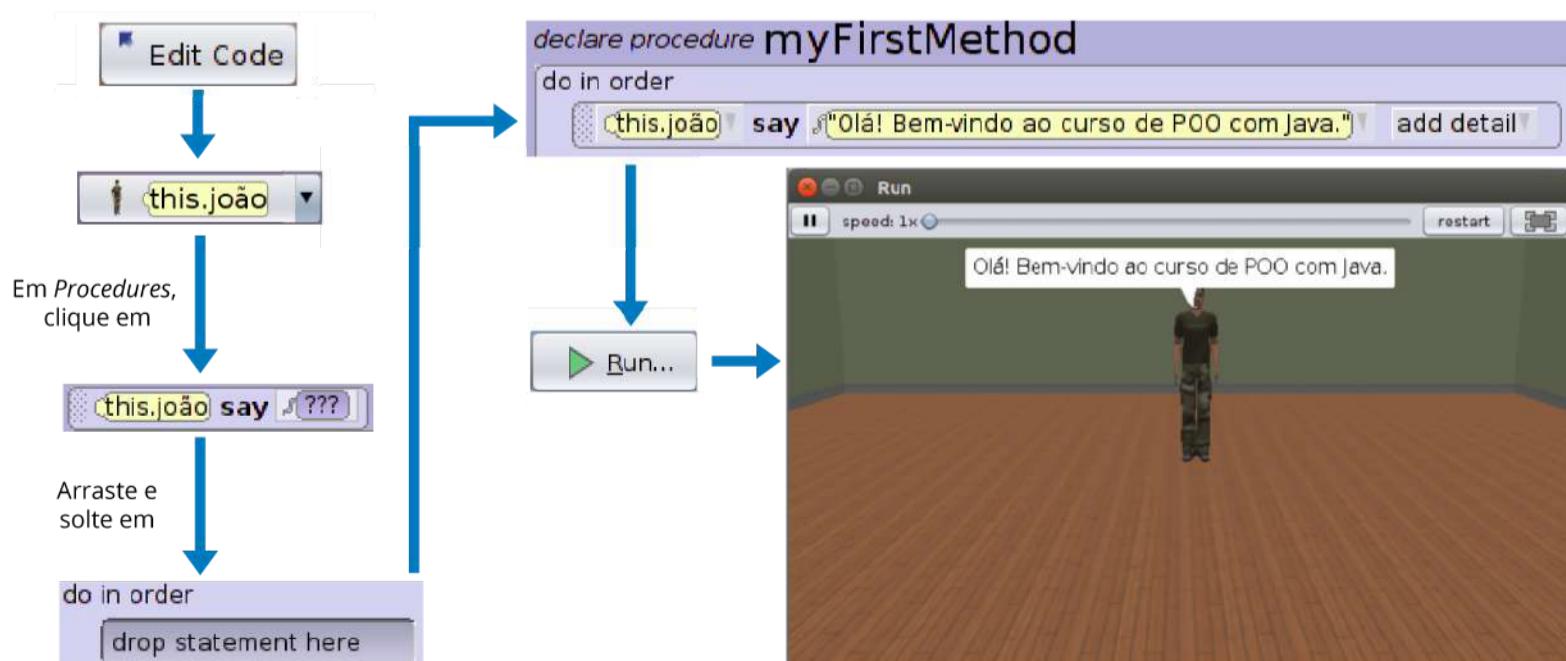
Fonte: capturas de telas do software Alice elaboradas pelo autor.

Na configuração da cena escolhemos de qual classe queremos inserir o nosso personagem; nesse exemplo, foi escolhida a classe bípede. Escolhemos também que seria um adolescente (classe Teen). Em seguida, foi aberta uma tela para edição das características físicas do nosso personagem. Selecionamos que ele seria do sexo masculino, de cor clara, cabelos pretos e curtos. Além disso, sua roupa também pode ser especificada.

Esses elementos especificados podem ser pensados como sendo os atributos da classe TeenPerson.

Vamos agora definir um primeiro comportamento para o nosso personagem. A Figura 1.10 ilustra o fluxo a ser seguido para definir um comportamento de fala.

Figura 1.10 | Passos para criação de um comportamento para o personagem criado



Fonte: capturas de telas do software Alice elaboradas pelo autor.

É necessário ir à área de edição de códigos, clicando em *Edit Code*. Em seguida, certifique-se de que `this.joão` esteja selecionado e, na aba *Procedures*, deve-se clicar, arrastar e soltar algum comportamento que você deseje. Nesse exemplo, o comportamento de falar (*say*) foi escolhido, sendo arrastado até a região de inserção de código. Uma mensagem de boas-vindas foi inserida, posteriormente, e clicamos em executar (*run*). Então, uma animação é executada em que o nosso personagem (joão) diz "Olá! Bem-vindo ao curso de POO com Java". Pronto, finalizamos a nossa primeira animação utilizando o software Alice.

o

[Ver anotações](#)

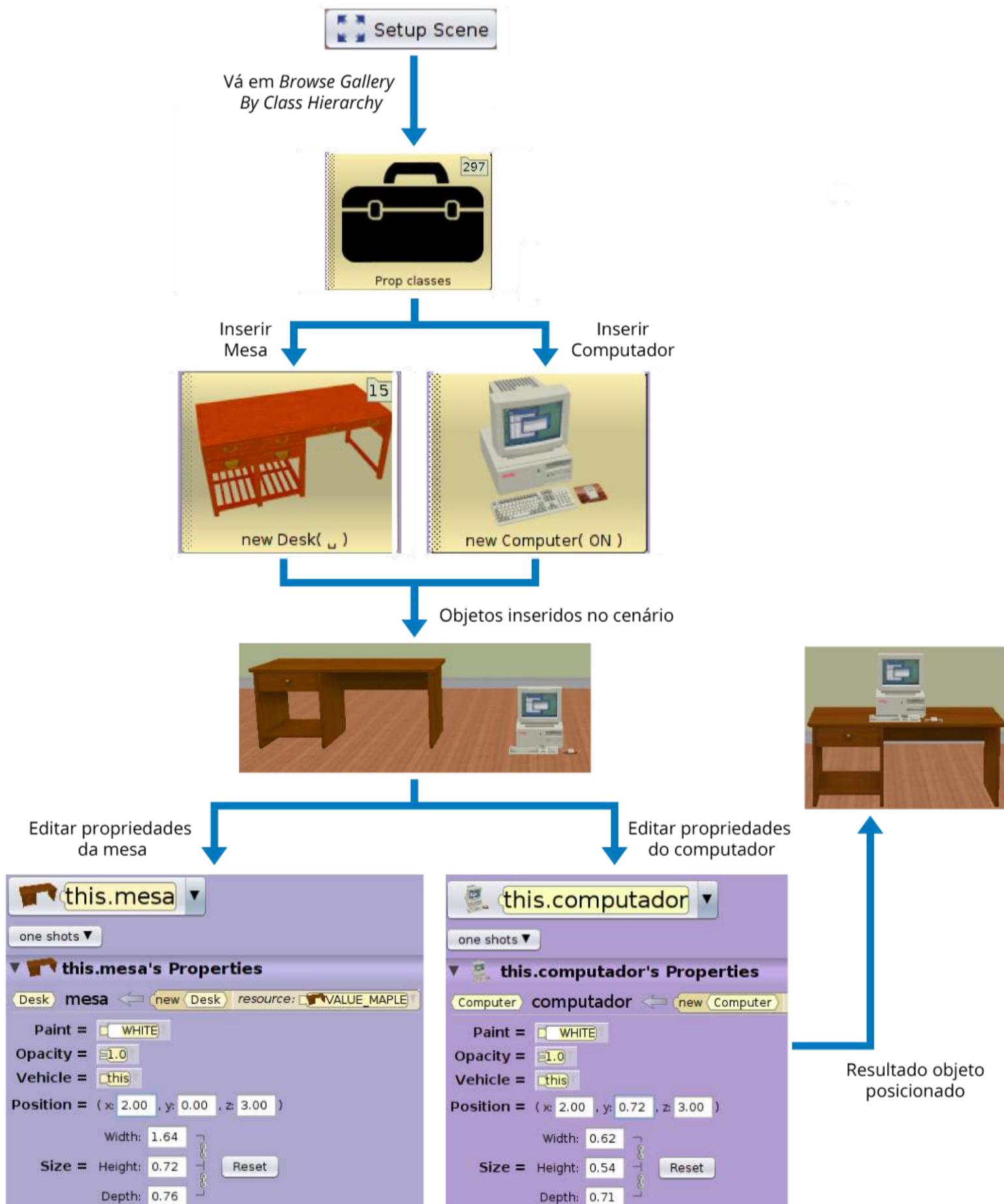
DICA

Caso você tenha interesse, pode-se exibir o código Java que foi automaticamente gerado pela ferramenta. Para isso, é preciso ir ao menu Window, Preferences e então marcar o campo Java Code On The Side (hidden). Esse recurso é interessante para que você se familiarize com a linguagem Java.

ADICIONANDO MAIS OBJETOS EM UM CENÁRIO

A construção de um cenário envolve uma série de itens (objetos) que podem ser adicionados. Cada objeto tem alguns atributos, como dimensões e posições que ocupam na cena. Vamos adicionar mais dois objetos na sala em que o personagem joão se encontra, para compreender as propriedades de dimensão e posição. A Figura 1.11 ilustra os procedimentos efetuados para inserção de uma mesa e um computador. Repare que inicialmente o computador está no chão.

Figura 1.11 | Passos para adicionar e posicionar objetos na cena



Fonte: capturas de telas do software Alice elaboradas pelo autor.

Agora vamos supor que queremos colocar o computador sobre a mesa. O primeiro passo é determinar qual posição espacial a mesa ocupará. A mesa tem o atributo *position* que contém as coordenadas (x, y, z). Dessa maneira, decidimos posicionar o objeto na coordenada (2.00, 0.00, 3.00). É importante destacar também que a mesa tem alguns atributos que dão sua dimensão, que são: *width* (largura), *height* (altura) e *depth* (profundidade). Os valores *default* são: 1.64, 0.72 e 0.76, respectivamente. Tendo esses valores em mente, agora fica fácil posicionar o computador sobre a mesa. Basta clicar nas propriedades do computador e então definir a posição como sendo $x = 2.00$, $y = 0.72$, $z = 3.00$.

Repare que colocamos o computador na mesma posição x e z que a mesa e na posição y colocamos o valor como sendo a altura da mesa.

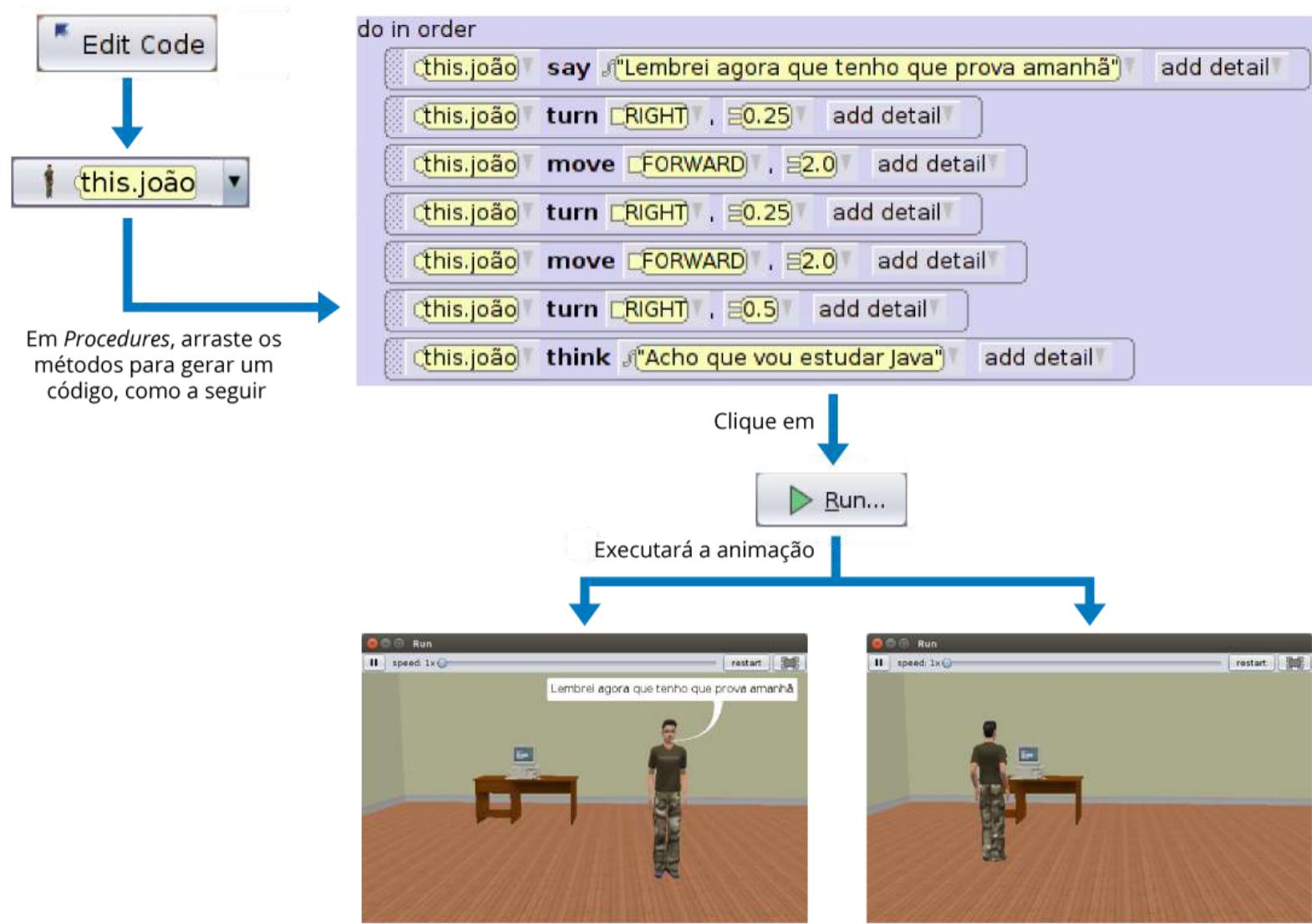
ANIMANDO UMA CENA

Agora que você já conhece a estrutura principal do software, vamos fazer uma animação que explora um conjunto de comportamentos. Imaginemos que nosso personagem joão acabou de lembrar que tem uma prova amanhã. Assim, você gostaria que ele saísse da sua posição atual e fosse até o computador para poder começar a estudar. Considere que ele se encontra na posição 0, 0, 0. Uma possível solução para isso está ilustrada na Figura 1.13, a seguir. Essa animação foi construída utilizando os métodos *say* (falar), *turn* (girar), *move* (mover), *turn*, *move*, *turn* e *think* (pensar), nesta ordem.

Figura 1.13 | Animação encadeada por diversos comportamentos diferentes

o

Ver anotações



Ainda com base no exemplo descrito, todos os métodos utilizados recebem argumentos/parâmetros de entrada.

- Os **métodos *say*** e ***think*** recebem um literal como entrada; esses argumentos serão utilizados pelos balões de fala e pensamento durante a animação.
- Os **métodos *turn*** e ***move*** recebem dois argumentos como entrada:
 - O primeiro argumento é a direção do giro (RIGHT) ou do movimento (FORWARD).
 - O segundo argumento indica um valor de intensidade usado no giro ou no movimento.

Uma dica é sempre executar o código desenvolvido, assim haverá uma maior compreensão do comportamento implementado.

Você deve ter reparado que o personagem joão, ao se deslocar, é arrastado no cenário como se estivesse de patins. Assim, pergunta-se: é possível definir movimentos para as pernas do joão de forma que ele pareça caminhar no cenário? A resposta é: sim, é possível. Fica o desafio para você, aluno: criar um comportamento no joão de forma que ele caminhe de fato pelo cenário.

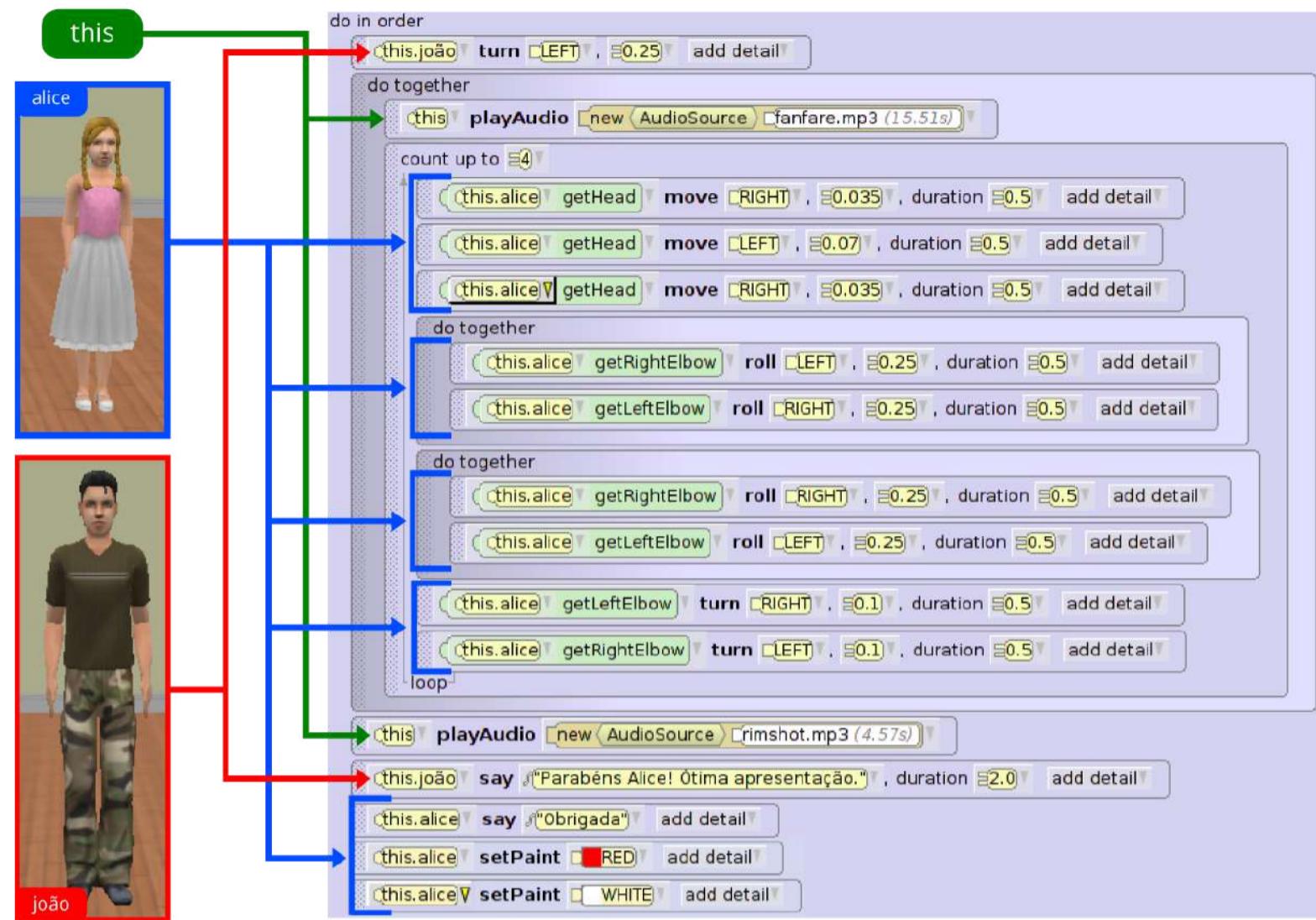
LEMBRE-SE

Ao construir um projeto na ferramenta Alice lembre-se sempre de salvá-lo. A extensão utilizada para salvar um projeto é .a3p (Alice 3 Project). Para fazer isso, basta ir ao menu *File*, clicar em *Save* e colocar um nome para o projeto.

| ESTRUTURAS DE CONTROLE UTILIZANDO O ALICE

Vamos mostrar como criar algumas estruturas de controle utilizando o Alice. Para isso, imagine que desejamos criar uma apresentação de dança, e uma nova personagem chamada alice foi criada e inserida na mesma sala que joão. Após isso, você deseja que ela faça uma simples apresentação de dança para o joão. O código mostrado na Figura 1.14 ilustra uma forma de criar um simples passo de dança feito pela alice, cena em que joão é o espectador.

Figura 1.14 | Animação da apresentação de dança da alice para joão



0

Ver anotações

Fonte: capturas de tela do software Alice elaboradas pelo autor.

Convidamos você a implementar esse código (Figura 1.14) e ver o efeito de animação que ele produzirá. O código mostrado destacou sobre quais objetos os métodos são chamados. O leitor atento deve ter reparado que algumas ações foram executadas pelo joão, outras pela alice e outras ainda pelo *this*. A ação executada pelo *this* foi a ação de tocar áudio. Como o áudio (música tocada) vem do ambiente, e não da alice ou do joão, fica fácil entender por que não chamamos essas ações a partir deles. O áudio, neste caso específico, é um elemento que deve vir do cenário em si, então foi colocado para o *this* chamar essa ação. Compreenderemos melhor esse comando nas seções seguintes deste livro.

REFLITA

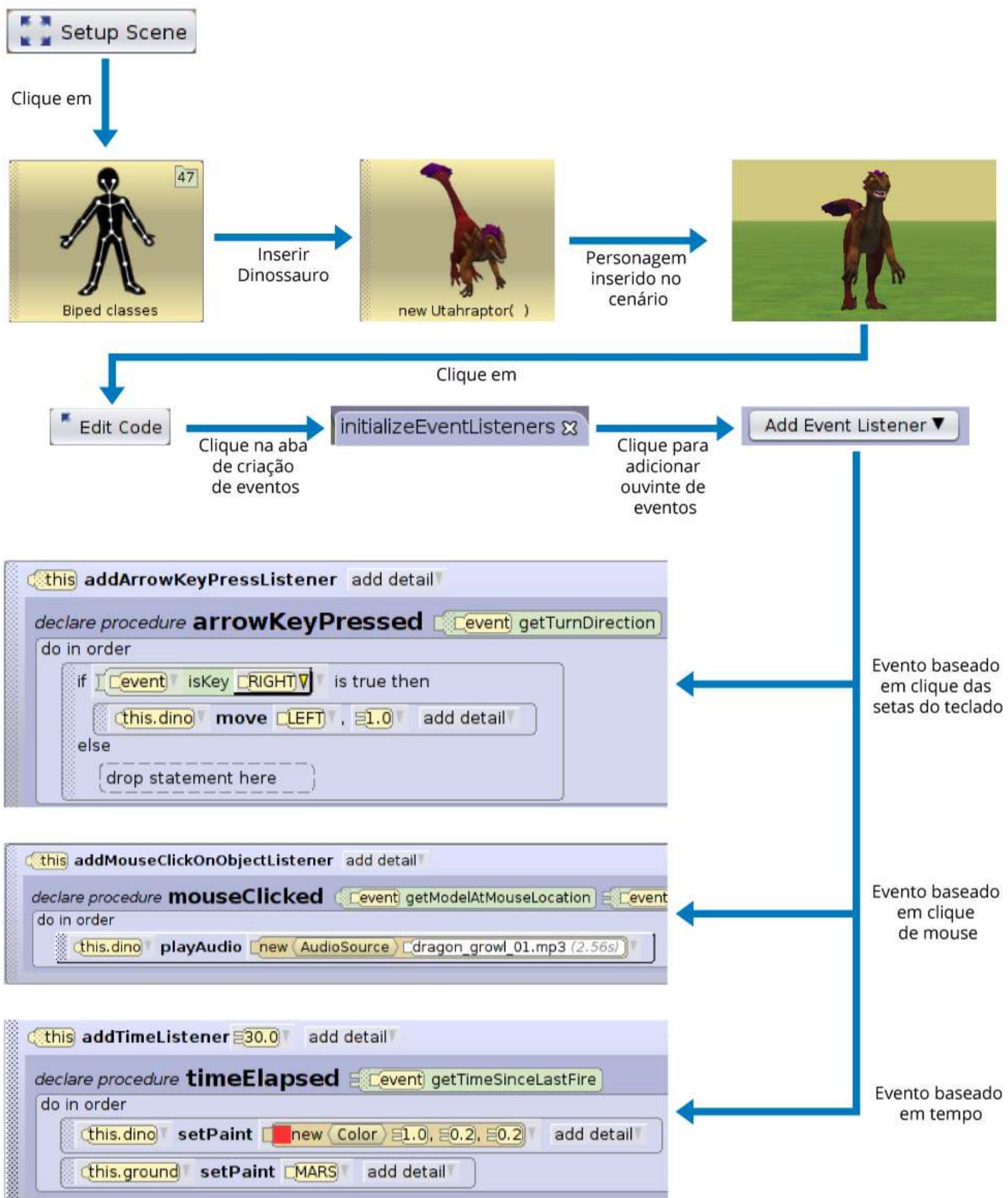
O código mostrado na Figura 1.14 apresenta alguns comandos como *count up to* e *do together*. O que esses códigos fazem? Na personagem alice a função *getHead* (pegue a cabeça) foi utilizada. O que você acha que ela faz? Ao aplicar sobre a alice a ação de mover, com o *getHead* especificado, qual parte do corpo da alice você acha que vai se mover? De forma semelhante, tente compreender o que faz as funções *getLeftElbow* e *getRightElbow*.

0

Ver anotações

No início desta seção foi dito que a ferramenta Alice permite a criação de jogos; no entanto, até agora vimos apenas a criação de animações. Um jogo, de forma geral, necessita de algum tipo de interação por meio de eventos de teclado ou mouse. Esses eventos precisam mudar algum comportamento na cena para permitir a interação entre o jogador e o personagem. A seguir, podemos ver um exemplo de jogo simples que permite ao jogador interagir com o personagem. A Figura 1.15 esquematiza alguns dos passos usados para criar esse ambiente de interação.

Figura 1.15 | Passos necessários para criação de interação entre usuário e personagem



Fonte: capturas de telas do software Alice elaboradas pelo autor.

Nesse exemplo da Figura 1.15, um personagem novo foi adicionado. É um dinossauro, ao qual chamamos dino. Na parte de edição de código, é necessário ir até a aba de criação de eventos (*initializeEventListeners*). Nessa tela, temos um botão para adicionar ouvinte de eventos. Nesse exemplo, foram adicionados três tipos de eventos distintos, que são:

- Evento para ouvir entrada de informação baseada em cliques das setas do teclado do computador, chamado *addArrowKeyPressListener*.
- Evento para ouvir entrada de informação baseada em cliques de mouse sobre um objeto, chamado *addMouseClickedOnObjectListener*.

- Evento para ouvir informação baseada em tempo, chamado *addTimeListener* (essa informação não é uma entrada do usuário).

As ideias por trás da orientação a eventos serão mais bem discutidas na Unidade 3 deste livro. Por hora, podemos pensar nesses eventos como entradas do usuário (jogador) que podem alterar o fluxo de controle do código de forma dinâmica.

Assim, ao pressionar a seta do teclado para a direita, o evento chamado *event.isKey(RIGHT)* é disparado, então o personagem dino é movido para a direita. Repare que no código está especificado que o dino moverá para a sua esquerda (*/left*), que para nós é correspondente à direita (o dinossauro está de frente para a tela, logo os sentidos de direita e esquerda estão invertidos).

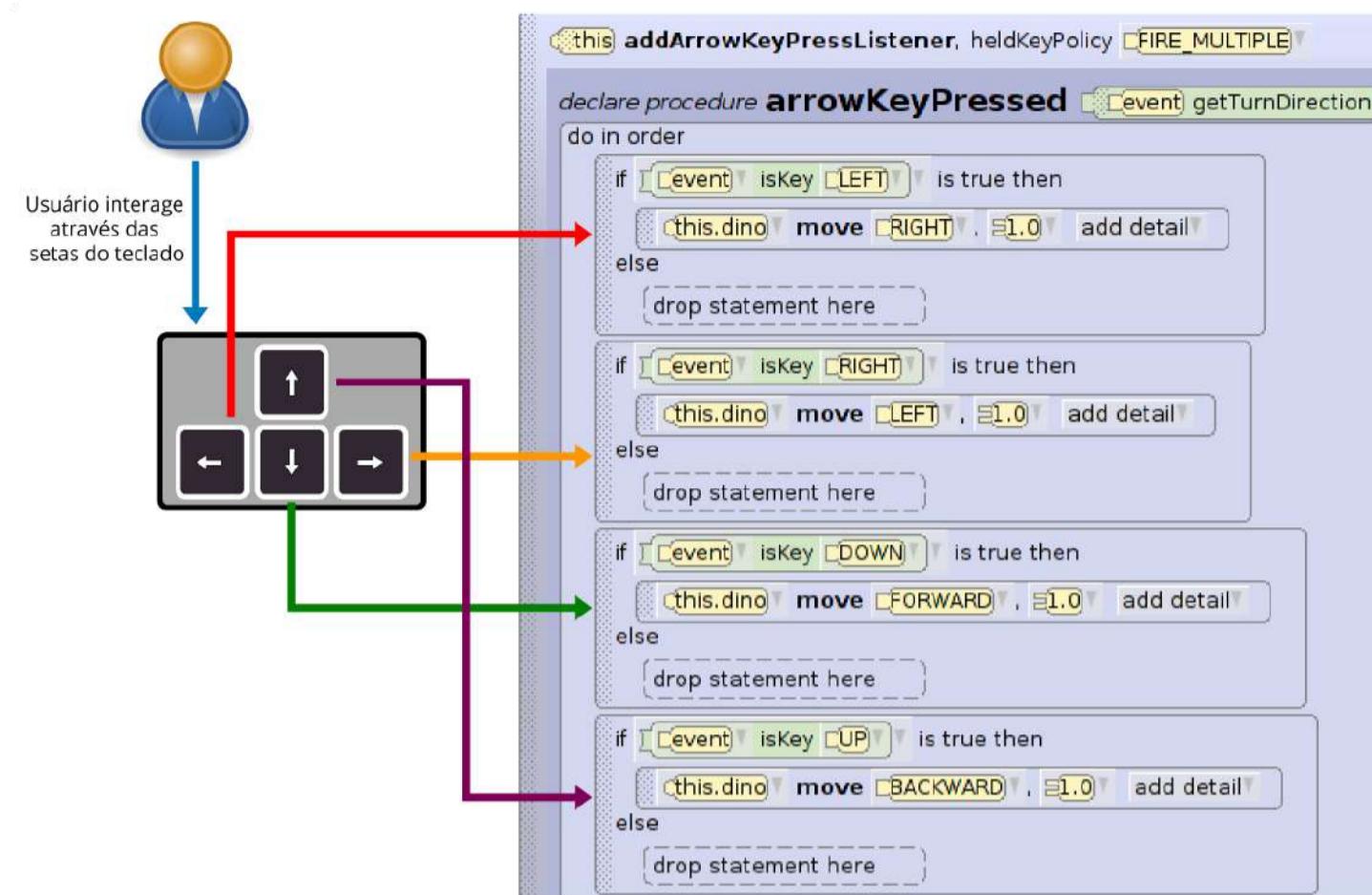
0

Ver anotações

EXEMPLIFICANDO

Considere o seguinte trecho de código, mostrado na Figura 1.16, que efetua o controle mais completo das ações de deslocamento do nosso personagem. Nesse sentido, diferentes ações de movimentação foram inseridas, de forma a responder ao clique para a esquerda, para a direita, para baixo e para cima. Ao inserir esses comandos no nosso personagem ele fica mais dinâmico, melhorando a jogabilidade. Caso incorporemos ainda mais comportamentos, o jogo vai se tornando mais complexo e, de forma geral, mais atrativo.

Figura 1.16 | Controle completo do personagem pelas setas do teclado



Fonte: captura de tela do software Alice elaborada pelo autor.

Caso você queira se aprofundar ainda mais na ferramenta Alice, recomendamos a referência Dann *et al.* (2014), um bom material de apoio ao estudo. Outra fonte de informações é o site com vários guias do software, disponíveis em Carnegie Mellon University (2014).

REFLITA

Agora que você já sabe fazer algumas animações e jogos com a ferramenta Alice e viu a enorme facilidade de utilizá-la, tente imaginar como você faria as mesmas animações e jogos sem essa ferramenta. Pense na grande quantidade de funções que você deveria implementar e que já estão prontas na ferramenta Alice.

Nesta seção você conheceu um pouco da ferramenta Alice, que auxilia no desenvolvimento de animações e jogos 3D e no ensino e aprendizagem de programação com OO. Você também viu como inserir objetos, modificar atributos e definir comportamentos para os nossos objetos, criando, assim, uma animação completa. Os projetos aqui desenvolvidos estão disponíveis no GitHub (ARANTES, 2020). A partir de agora será possível aprofundar mais na programação.

REFERÊNCIAS

ARANTES, J. S. **Livro:** Linguagem Orientada a Objetos. Github, 2020. Disponível em: <https://bit.ly/3eiUMcF> . Acesso em: 20 abr. 2020.

CARNEGIE MELLON UNIVERSITY. **Alice.** CMU, [s.d.]. Disponível em: <https://bit.ly/2Zf7k0p> . Acesso em: 26 abr. 2020.

CARNEGIE MELLON UNIVERSITY. **How to Guide for Alice 3.** CMU, 2014. Disponível em: <https://bit.ly/2O9nPVC> . Acesso em: 5 maio 2020.

DANN, W. *et al.* **ALICE 3:** how to guide. Pittsburgh, Pennsylvania: Carnegie Mellon, 2014.

DANN, W. *et al.* Mediated transfer: Alice 3 to java. *In:* ACM TECHNICAL SYMPOSIUM ON COMPUTER SCIENCE EDUCATION, 43., 2012, Raleigh, North Carolina.

Proceedings [...]. Raleigh, SIGCSE, 2012, p. 141-146.

INTRODUÇÃO à lógica de programação orientada a objetos utilizando a ferramenta Alice. Youtube, 15 jul. 2019. 8 vídeos (2h27min). Publicado pelo canal Danilo Filitto. Disponível em: <https://bit.ly/3aWv10l> . Acesso em: 2 maio 2020.

ORACLE. **Java SE Downloads.** Oracle, 2020. Disponível em: <https://bit.ly/3iKLytv> . Acesso em: 25 abr. 2020.

PBWORKS. **How to download and install Alice 3.** PBworks, 2018. Disponível em: <https://bit.ly/2Cq54e8> . Acesso em: 25 abr. 2020.

FOCO NO MERCADO DE TRABALHO

INTRODUÇÃO A APLICAÇÕES ORIENTADAS A OBJETOS

Jesimar da Silva Arantes

Ver anotações

APRESENTANDO AS FUNCIONALIDADES DO ROBÔ R-ATM MEDIANTE ANIMAÇÃO 3D

Com a ferramenta Alice você poderá dar "vida" ao seu robô inteligente.



Fonte: Shutterstock.

Deseja ouvir este material?

Áudio disponível no material digital.

SEM MEDO DE ERRAR

A *startup* em que você trabalha está com um grande desafio: fazer uma animação 3D para divulgação de um produto. Essa animação deve demonstrar algumas funcionalidades básicas do robô móvel que será projetado. Você foi incumbido de resolver esse problema. Para isso, a animação deve simular um robô que se move em uma sala retangular. As principais funcionalidades que a animação deve contemplar são:

- O robô deve se mover.
- O robô deve fazer o reconhecimento do cenário.

- O robô deve sair de um local de origem e chegar até um local contendo caixas.

- O robô deve conseguir identificar os conteúdos das caixas.

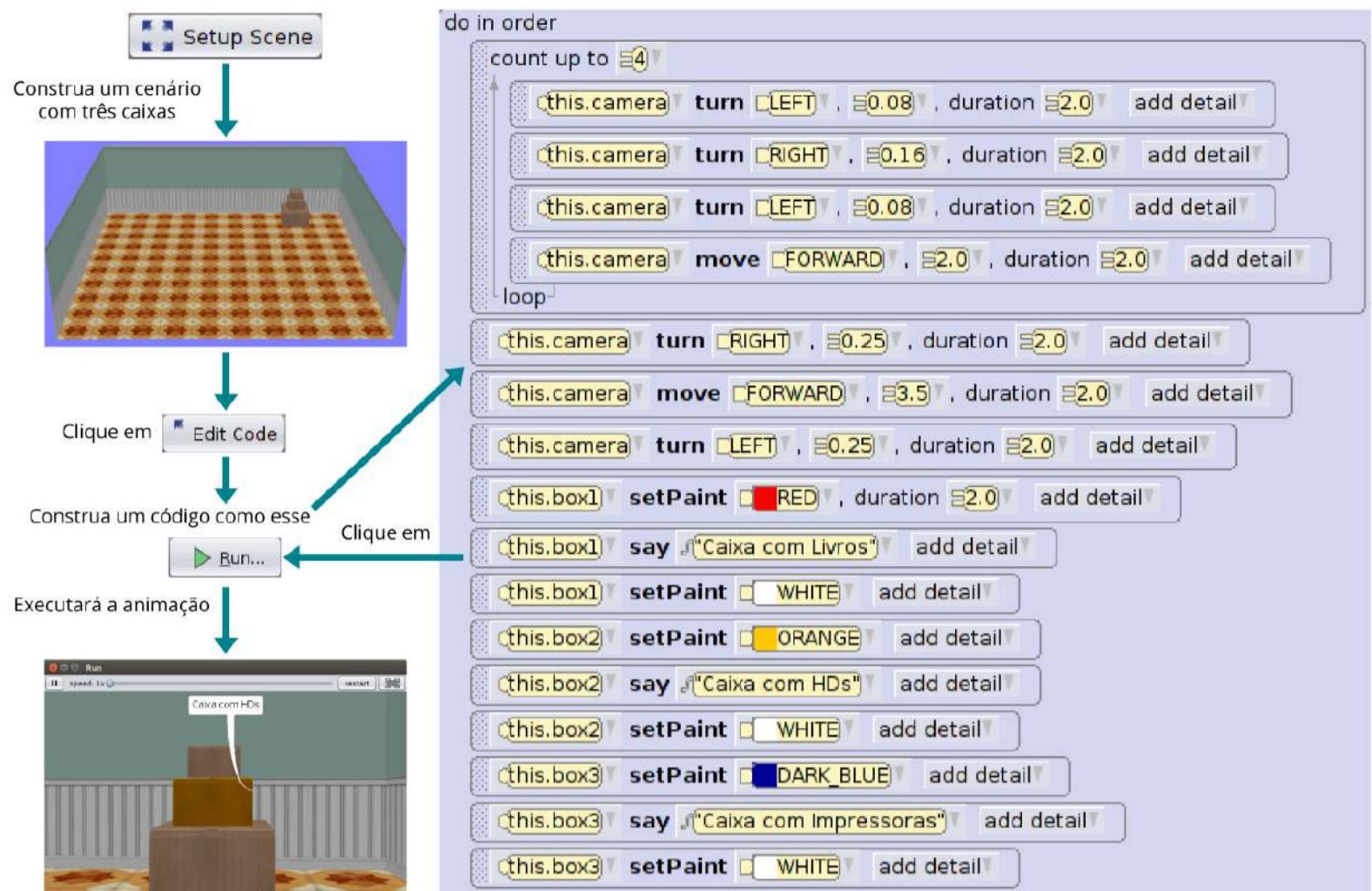
Seu chefe especificou também que essa sala deve conter três caixas empilhadas.

Em cada uma das caixas deverá haver algum tipo diferente de conteúdo, como livros, HDs e impressoras. Além disso, a animação deverá ser em primeira pessoa, ou seja, quando o robô se desloca, a visualização do cenário deve acompanhar o seu movimento.

Dado esse desafio, algumas perguntas surgem: qual tipo de modelo de cenário dentro do Alice se aproxima do que seu chefe deseja? Será que a ferramenta Alice dispõe de caixas já modeladas semelhantes às que você quer utilizar? Como você fará para criar a animação em primeira pessoa? Como você fará para dar a impressão de que o robô está fazendo o reconhecimento da sala?

Como forma de resolver o desafio proposto pelo seu chefe, você decide utilizar a ferramenta Alice. Você percebe que o primeiro passo é a criação das três caixas. Na ferramenta Alice existe uma classe chamada *Box* que está dentro de *Prop classes*. Então, você decide utilizá-la, pois ela atende ao que é necessário. Você decide colocar a primeira caixa na posição (-3, 0, 4). A segunda caixa é colocada sobre a primeira. E a terceira caixa é colocada sobre a segunda. Você, então, procura algum modelo de robô 3D na ferramenta Alice, mas não encontra nenhum que seja similar ao que se pretende construir. Inicialmente, você pensa que será um problema fazer a animação sem o robô, mas então lembra de que o seu chefe quer que a animação seja em primeira pessoa. Assim, chega à conclusão de que, se você movimentar a câmera no cenário, dará a impressão de que o robô se deslocou e não será necessário ter um modelo de robô. Tendo isso em mente, você decide fazer um código como mostrado na Figura 1.17.

Figura 1.17 | Solução da situação-problema



Fonte: capturas de telas do software Alice elaborada pelo autor.

Analizando a solução mostrada, podemos reparar que o único objeto que se move na cena é a câmera. Isso, conforme dissemos, fornece o efeito de que o robô está se deslocando na cena. Ao utilizar o método *turn* associado à câmera, o efeito é que o robô está fazendo o reconhecimento do cenário. Assim que o robô chega nas caixas, ele verifica cada uma delas, olhando o seu conteúdo. A forma que você utilizou para demostrar o reconhecimento do conteúdo das caixas é alterando as cores das caixas e colocando uma mensagem associada a cada uma delas. Então você decide mostrar a animação ao seu chefe. Ele gostou bastante da animação, que é apresentada para a empresa de e-commerce. A empresa adorou a demonstração das funcionalidades básicas do robô e está bastante animada com o andamento do projeto.

PESQUISE MAIS

Existem diversas videoaulas na internet que ensinam a construir aplicações utilizando a ferramenta Alice, no entanto a maioria está em inglês. O professor Danilo Filitto tem um canal no YouTube em que divulga algumas videoaulas em português explicando como criar aplicações utilizando o Alice. Esses vídeos fornecem uma visão geral sobre a construção de aplicações utilizando a ferramenta.

INTRODUÇÃO à lógica de programação orientada a objetos utilizando a ferramenta Alice. YouTube, 15 jul. 2019. 8 vídeos (2h27min). Publicado pelo canal Danilo Filitto.

0

Ver anotações

UMA VISÃO DE GAME EM PROGRAMAÇÃO ORIENTADA A OBJETOS

0

Jesimar da Silva Arantes

[Ver anotações](#)

A LINGUAGEM ORIENTADA A OBJETOS NA CRIAÇÃO DE JOGOS 2D

Com a ferramenta Greentoot, podemos criar jogos com diversos personagens controlados por inteligência artificial simples.



Fonte: Shutterstock.

Deseja ouvir este material?

Áudio disponível no material digital.

PRATICAR PARA APRENDER

Prezado aluno, bem-vindo à terceira seção dos estudos sobre linguagem orientada a objetos. Você já se perguntou como eram feitos os jogos de videogame de 8 e 16 bits? Já quis aprender a desenvolver o seu próprio jogo de plataforma ou aventura no estilo 8 e 16 bits? Antigamente, a tarefa de desenvolver um jogo era complexa e levava vários meses – ou anos – para que uma equipe a executasse. Parte dessa dificuldade estava relacionada à inexistência de ferramentas que auxiliassem na programação. Atualmente, existem diversas ferramentas que auxiliam nesse processo, abstraindo a complexidade da implementação.

Nesta seção estudaremos uma dessas ferramentas: o Greenfoot. Uma das vantagens do Greenfoot é que ele permite fazer a criação de jogos e animações bidimensionais de forma bastante simples rapidamente. O Greenfoot tem sido utilizado no ensino de programação em diversas universidades.

Você, durante seu estágio em uma *startup*, recebe de seu chefe diversos desafios. Recentemente, ele fechou um trabalho de desenvolvimento de um robô móvel inteligente com uma empresa de *e-commerce*. Nessa semana, seu chefe desafiou você a desenvolver um jogo/simulador 2D em que o robô R-ATM deve se deslocar em um cenário que simule uma sala/galpão contendo diversas caixas com mercadorias. Ele gostaria de mostrar esse jogo para a empresa de *e-commerce* como um produto derivado que demonstra as funcionalidades do robô que a sua *startup* desenvolverá. As principais funcionalidades que o jogo/simulação deve contemplar são:

- O robô deve se mover.
- O robô deve evitar regiões da sala em que existem máquinas trabalhando.
- O robô não pode se deslocar sobre as caixas presentes no cenário.
- O robô deve dispor de duas velocidades de deslocamento.
- A sala deve ter um formato retangular.
- A visualização do robô deve ser em terceira pessoa.

Agora que o seu desafio foi apresentado, chegou o momento de estudar esta seção, pois ela será de grande ajuda para resolvê-lo. Você aprenderá a criar jogos e animações 2D, e para isso você verá como criar atores/personagens e definir cenários, além de compreender como posicionar os atores no cenário, definir comportamentos, adicionar recursos de áudio e interagir com um personagem. Vamos conhecer essa ferramenta para poder resolver esse desafio?

Bom estudo!

CONCEITO-CHAVE

INTRODUÇÃO

A orientação a objetos (OO) facilita muito a criação de diversas aplicações, como animações e jogos. Na seção anterior, estudamos a ferramenta Alice, que usa uma linguagem de blocos construída sobre a linguagem Java, e focamos, em geral, a criação de animações tridimensionais. Nesta seção, focaremos o desenvolvimento

de jogos bidimensionais. Dessa maneira, vamos imaginar que você queira criar um jogo e que nesse jogo existem diversos personagens controlados por alguma inteligência artificial simples (*bots*). Nesse jogo também há um personagem que você pode controlar, além dos objetos que compõem o cenário. Imagine, agora, que queremos desenvolver esse jogo não em uma linguagem de blocos, mas utilizando uma linguagem de programação tradicional, como o Java. Uma das formas de fazer isso é programando manualmente todas as telas gráficas do jogo 2D, utilizando alguma biblioteca gráfica do Java. Outra forma é utilizando alguma ferramenta que abstraia os detalhes de interface, e você se preocupa em desenvolver apenas as regras do jogo. Pois bem, é exatamente isso que o Greenfoot faz. Conheceremos agora essa ferramenta que auxilia de forma imensa no processo de criação de jogos e animações.

DICA

Neste vamos nos referir ao Greenfoot como uma ferramenta para criação de jogos e simulações. No entanto, em diversos lugares o Greenfoot é chamado de Ambiente de Desenvolvimento Integrado (IDE) para a linguagem Java. As duas formas de se referir ao Greenfoot estão corretas.

EXPLORANDO O GREENFOOT

O software Greenfoot é uma ferramenta que auxilia no desenvolvimento de jogos, simulações e outros programas gráficos. Por ser simples e apresentar um visual gráfico agradável, essa ferramenta permite que programadores novos criem jogos mesmo tendo poucos conhecimentos da linguagem. Dessa forma, convidamos você a se aprofundar no software Greenfoot: você logo perceberá o quanto ele é fantástico e ajuda na compreensão da OO.

Nesta seção utilizaremos o software Greenfoot, versão Greenfoot 3.6.1. Você pode acessá-la em Greenfoot ([s.d.]a), baixar e efetuar sua instalação. O processo de instalação é bastante simples e por isso não será mostrado. Outro ponto a ser destacado é que, após a instalação do Greenfoot, pode ser necessária a instalação do Java Development Kit (JDK), caso você tenha o JDK instalado. O download do JDK pode ser feito no site da Oracle (2020). O Greenfoot é uma ferramenta gratuita e multiplataforma, ou seja, executa em sistemas operacionais Windows, Linux e MacOS X.

TELA DE DESENVOLVIMENTO

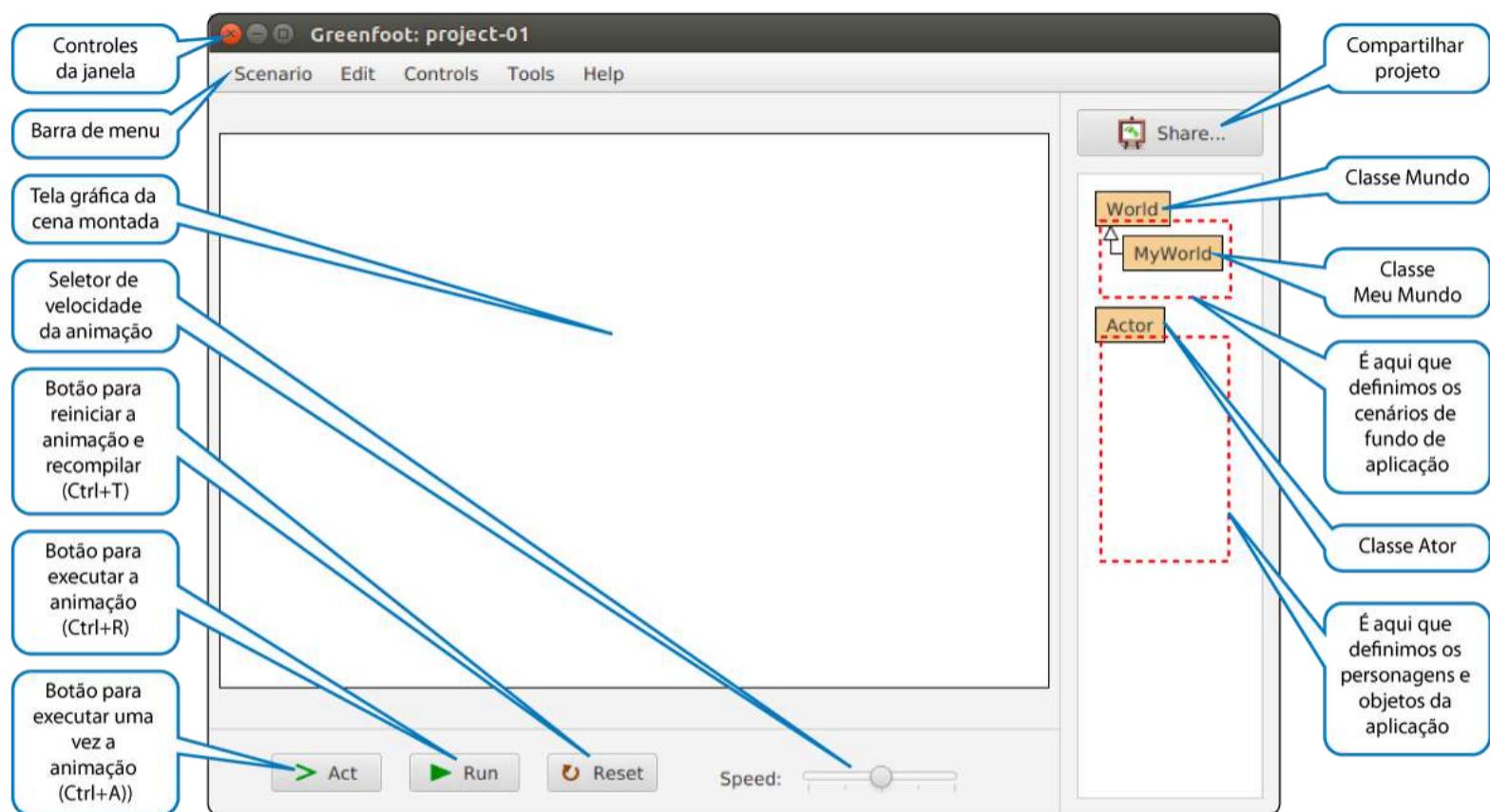
Ao abrir o software Greenfoot, a tela de trabalho principal será mostrada. Em seguida, crie um cenário: vá em *Scenario*, *New Java Scenario* e atribua um nome a ele. Assim, a sua tela deverá ser semelhante à Figura 1.18. Analise as descrições feitas na figura para conhecer os principais recursos presentes nessa janela.

Alguns elementos principais disponíveis nesse ambiente são: a tela gráfica da cena montada, o botão para executar a animação, o botão para reiniciar a animação, a classe *World* e a classe *Actor*.

o

[Ver anotações](#)

Figura 1.18 | Tela de trabalho principal do software Greenfoot



Fonte: captura de tela do software Greenfoot elaborada pelo autor.

DICA

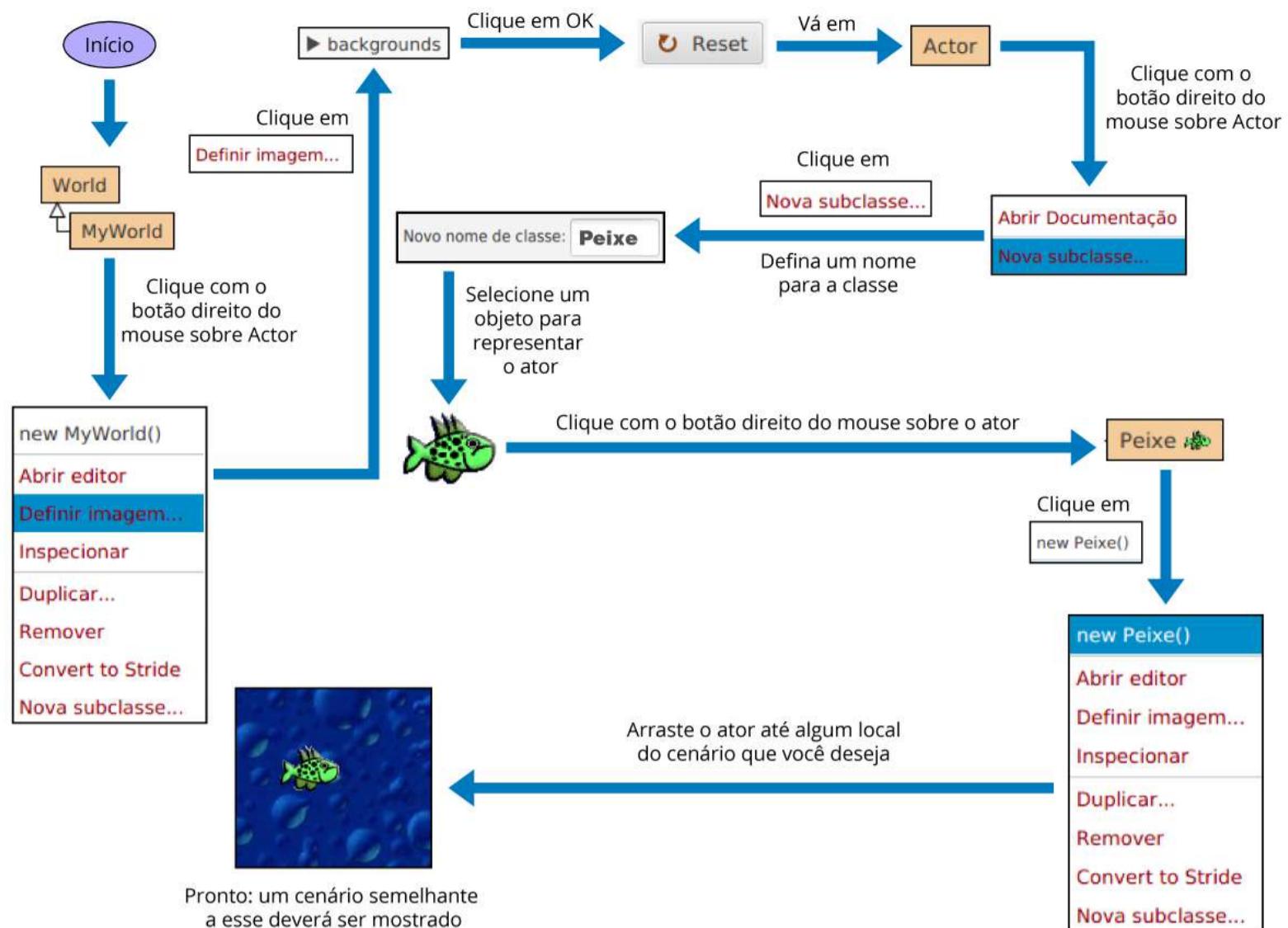
Caso você deseje, o idioma pode ser alterado para diversas línguas, inclusive o Português. Para trocar o idioma vá em menu *Tools, Preferences, Interface, Language selection* e então selecione Português.

ESCREVENDO UM PROGRAMA NO GREENFOOT

Uma vez que foi dada uma visão geral do software, vamos agora montar um cenário em que possamos escrever o nosso primeiro programa com o Greenfoot.

O primeiro passo é fazer a criação de um cenário de fundo, e então inserir um personagem sobre ele. A Figura 1.19 mostra alguns passos necessários para isso, analise-a.

Figura 1.19 | Passos para criação/inserção de um cenário de fundo e um personagem



Fonte: capturas de telas do software Greenfoot elaboradas pelo autor.

Inicialmente, foi feita a criação do mundo, sendo definida uma imagem de fundo. Em seguida, criou-se um ator (personagem). Por fim, é necessário arrastar o ator até a posição do cenário em que queremos colocá-lo. Pronto: esses são os passos básicos para a criação de um cenário e para colocarmos um personagem sobre ele.

Alguns pontos aqui precisam ser esclarecidos. O mundo (*World*) é uma classe, o meu mundo (*MyWorld*) também é uma classe.

Em específico, a classe *MyWorld* é filha de *World*, ou seja, *MyWorld* é subclasse de *World*.

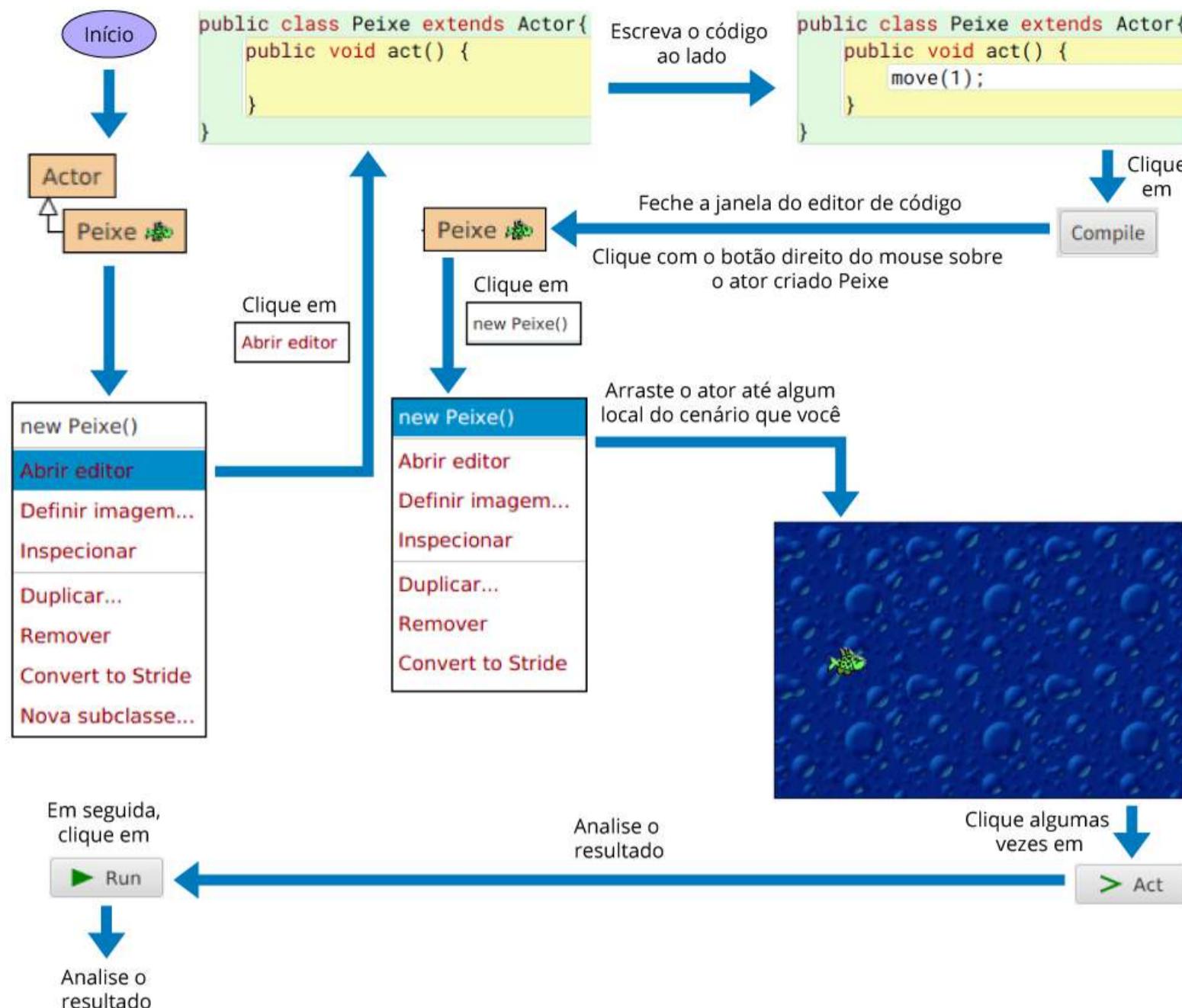
Ainda nesse exemplo, ator (*Actor*) é uma classe, o Peixe também é uma classe.

Neste caso, Peixe é subclasse de Actor.

Por hora, vamos dizer que uma subclasse tem todas as características da classe mãe (sua superclasse). Mais detalhes sobre subclasses e superclasses serão discutidos quando falarmos de herança. Aqui, é importante destacar que os dois principais componentes utilizados pelo Greenfoot para fazer um jogo ou animação são o cenário de fundo e os atores.

Suponhamos que agora desejamos adicionar um comportamento ao peixe criado; você gostaria de fazer o peixe se mover simplesmente para frente. A Figura 1.20 ilustra a sequência de passos necessários para isso, analise-a.

Figura 1.20 | Passos para criação de comportamento e execução da aplicação



Fonte: capturas de telas do software Greenfoot elaboradas pelo autor.

Convidamos você a executar os passos descritos em seu computador. O principal ponto a ser destacado aqui é a janela do editor de códigos: nela você passará grande parte do tempo para projetar um jogo. Agora que você já conhece todo o fluxo para criar uma ação/comportamento, os próximos exemplos destacarão mais os trechos de código criados dentro desse editor. Você poderá acessar e ver o resultado dessa simples implementação em Greenfoot (2020a).

Ainda com relação ao exemplo anterior, percebemos dentro da classe Peixe que ela herda (*extends*) as características de *Actor*. Dentro dessa classe existe um método que inicialmente estava vazio, chamado *act()*.

É importante ter em mente que esse método é chamado uma vez quando se clica no botão Act (ou botão Executar uma Vez, no software em português).

Por sua vez, esse método é chamado vezes seguidas quando se clica no botão *Run* (ou botão Executar, no software em português). Para criarmos o comportamento de mover-se para frente foi feita uma chamada do método *move*, passando 1 como argumento. Esse método basicamente desloca o objeto peixe 1 pixel para frente. O método *move* recebe um número inteiro como argumento. Não são aceitos números em ponto flutuante, e para entender o porquê pense em termos de pixels.

LEMBRE-SE

O site do Greenfoot dispõe de uma galeria utilizada na divulgação de jogos e animações. A fim de se inspirar na criação do seu próprio jogo, visualize as aplicações lá disponibilizadas (GREENFOOT, [s.d.]b). Você também pode baixar alguns dos jogos disponíveis e olhar o seu código-fonte para entender como alguns recursos são utilizados. A seguir estão listados alguns jogos interessantes desenvolvidos pela comunidade e que podem servir de inspiração.

Quadro 1.1 | Jogos desenvolvidos pela comunidade Greenfoot

Nome do jogo	Endereço do jogo
Basic 3x3x3 Cube	https://www.greenfoot.org/scenarios/25500
33 Seconds	https://www.greenfoot.org/scenarios/24284
Roadkill	https://www.greenfoot.org/scenarios/25372
World War 3	https://www.greenfoot.org/scenarios/25222
ChessWorld	https://www.greenfoot.org/scenarios/2899
Memory Tester	https://www.greenfoot.org/scenarios/10837
Dawn of War2	https://www.greenfoot.org/scenarios/25812

Fonte: elaborado pelo autor.

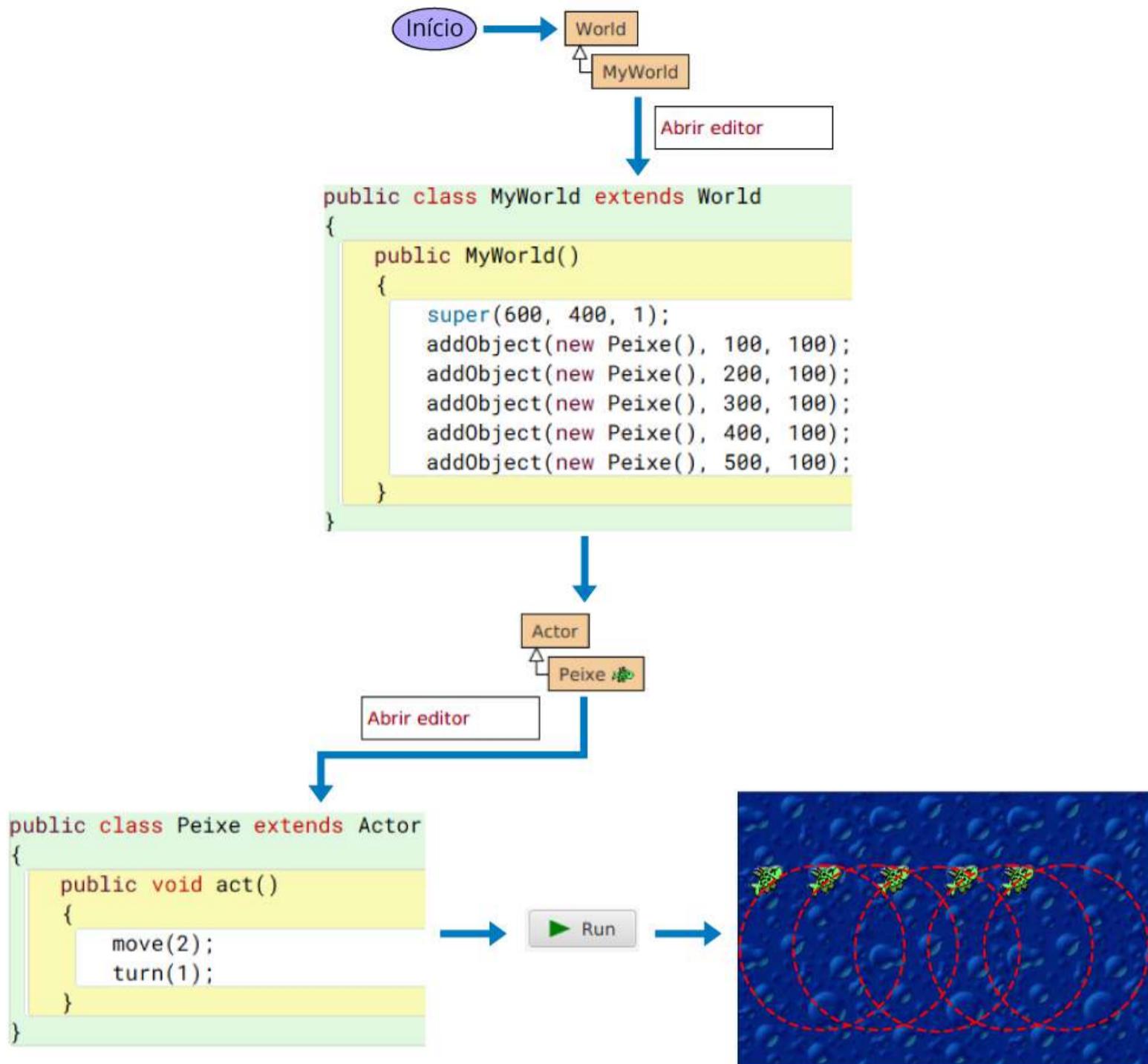
INCORPORANDO ELEMENTOS ATRAVÉS DE UMA CLASSE

Veremos agora como construir uma aplicação com diversos peixes (diversos objetos) movendo-se em círculos. No exemplo anterior, para criação/inserção do objeto tivemos que clicar sobre o ator e arrastá-lo até o cenário. Agora

mostraremos como fazer isso incorporando-o diretamente na classe mundo.

Analise a Figura 1.21 para entender como foi feito esse processo.

Figura 1.21 | Passos para inserção automática de atores no mundo e movimentação



Na classe meu mundo foi inserido um total de cinco peixes, utilizando uma chamada ao método addObject. Esse método recebe como parâmetros o objeto que será inserido (nesse caso, um novo peixe) e as coordenadas (x, y) do objeto. Assim, repare que cada um dos peixes foi colocado em uma posição diferente. Para dar um comportamento de nadar em círculos foram invocados dois métodos:

- Um para deslocar 2 pixels para frente, chamado *move(2)*.
- Um para virar um grau, chamado *turn(1)*.

Recomendamos variar os valores desses parâmetros para melhor compreendê-los, testando inclusive valores negativos (mas devem ser números inteiros). Você poderá acessar e ver o resultado dessa simples implementação em Greenfoot (2020f).

O Greenfoot já apresenta algumas classes implementadas, e cada classe tem diversos métodos. É importantíssimo que o leitor tenha o hábito de ler a documentação das classes e métodos já disponíveis para compreender como utilizá-las. A documentação das bibliotecas do Greenfoot pode ser acessada em Greenfoot ([s.d.]b).

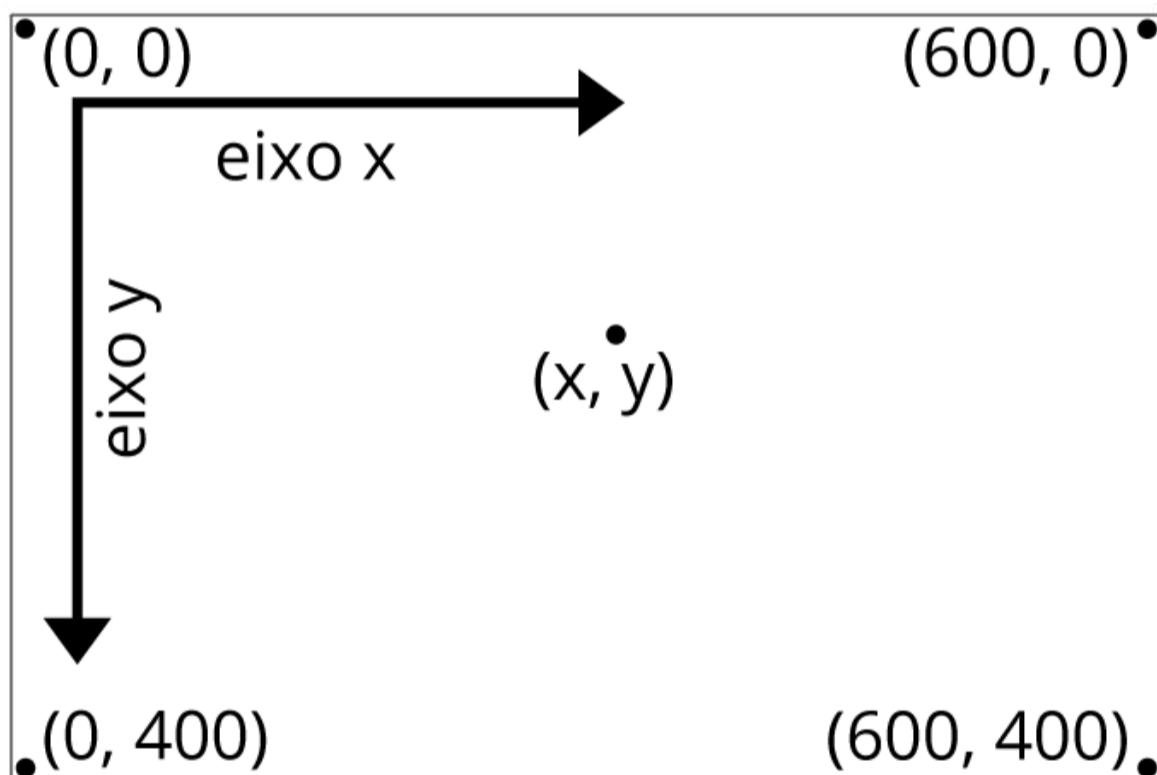
o

Ver anotações

ASSIMILE

A ferramenta Greenfoot utiliza um sistema de coordenadas semelhantes às coordenadas cartesianas para inserção dos objetos no cenário. Os cenários, em geral, apresentam 600×400 pixels, no entanto, essas dimensões podem ser alteradas. É importante ter em mente que a coordenada $(0, 0)$ está na parte superior esquerda. O eixo x cresce no sentido esquerda para direita. O eixo y cresce no sentido de cima para baixo (inverso do sistema cartesiano tradicional). Analise a Figura 1.22.

Figura 1.22 | Sistema de coordenadas no software Greenfoot



Fonte: elaborada pelo autor.

Agora veremos como incorporar diferentes atores no nosso cenário, cada um com diferentes comportamentos. Mostraremos também como carregar imagens próprias para serem utilizadas pelos nossos personagens e cenário. Além disso, veremos como inserir áudio tanto no cenário quanto em um personagem. Assim, o primeiro passo para isso é salvar o projeto. Quando se salva um projeto utilizando o Greenfoot na pasta do projeto, são criadas automaticamente três pastas nomeadas de *images*, *sounds* e *doc*.

- Na pasta *images* você pode adicionar qualquer imagem que deseje, e ela estará automaticamente disponível para ser utilizada pelos atores e cenários de fundo. As extensões de imagens suportadas são: png, jpeg e gif.
- Na pasta *sounds* devem ficar armazenados todos os áudios utilizados no projeto. As extensões de áudios suportadas são midi, wav, mp3, au e aiff.
- Na pasta *doc* são armazenadas automaticamente um conjunto de arquivos que estão relacionados com a documentação do projeto.

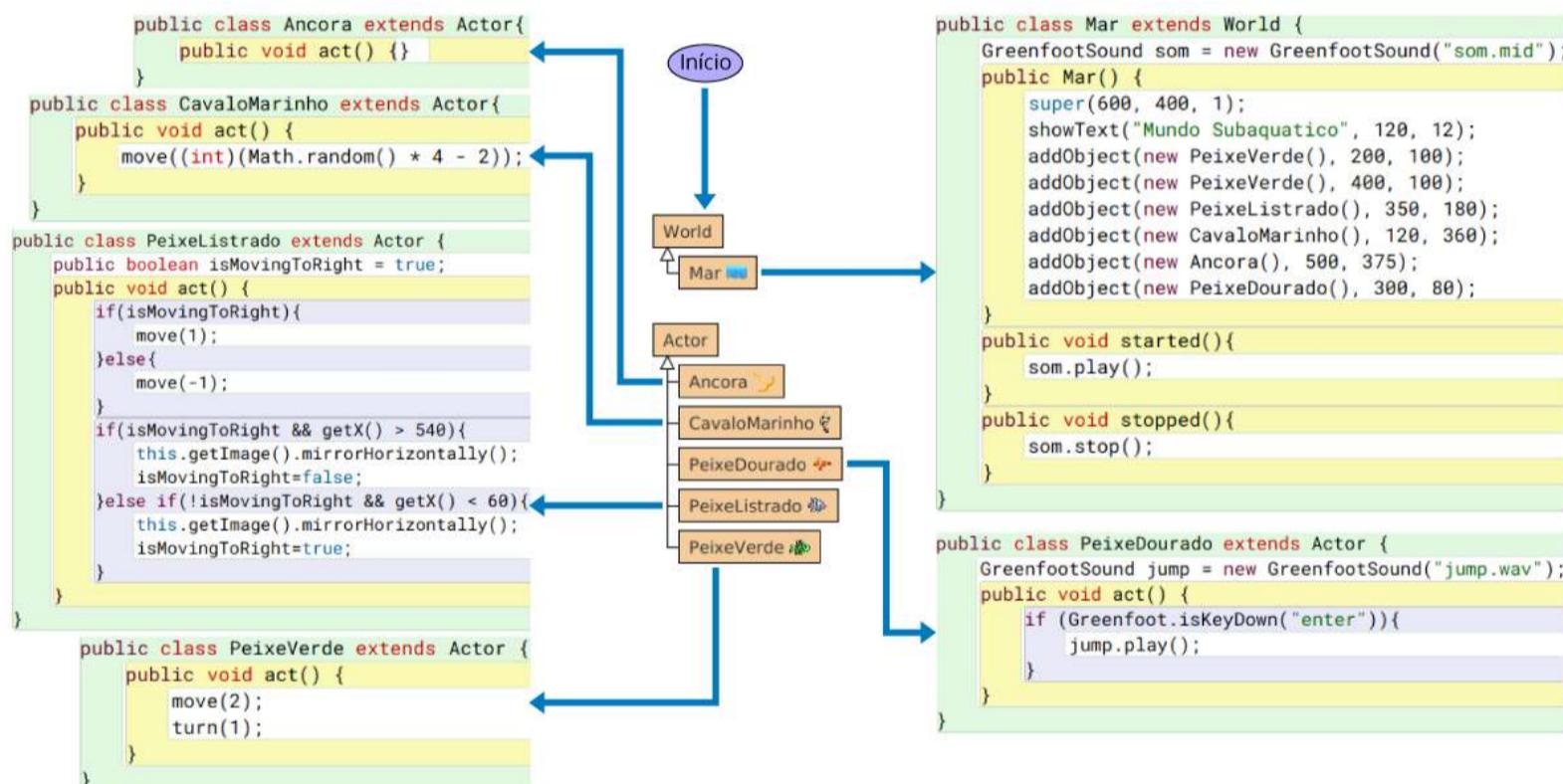
DICA

Para que seu jogo ou animação fique com o aspecto bem legal, você pode querer utilizar alguma imagem ou áudio criado por alguém da comunidade. É sempre bom ter certeza que as imagens e áudios que você utiliza estejam com uma licença livre, para não ter problemas com questões de direitos autorais. Os sites OpenGameArt ([s.d.]) e Freesound (2020) dispõem de muitas imagens e áudios que podem ser utilizados livremente. Mesmo assim, é bom dar os devidos créditos ao autor da imagem ou áudio.

Ver anotações

Continuando o exemplo anterior, analise a Figura 1.23.

Figura 1.23 | Criação de diferentes atores, objetos e inserção de áudio



Fonte: capturas de telas do software Greenfoot elaboradas pelo autor.

Perceba que criamos um cenário chamado Mar, além de alguns atores chamados Ancora, CavaloMarinho, PeixeDourado, PeixeListrado e PeixeVerde.

- A **Ancora** criada não apresenta nenhuma ação associada, sendo então apenas um objeto no cenário.
- O **CavaloMarinho** apresenta um pequeno movimento aleatório em torno de sua posição inicial.
- O **PeixeListrado** tem um movimento de ida e volta.
- O **PeixeVerde** tem o mesmo comportamento definido anteriormente, de nadar em círculos.

- O **PeixeDourado** apresenta apenas um comportamento, que é tocar um áudio quando a tecla *enter* for pressionada pela função *isKeyDown*, presente na classe Greenfoot. Para tocar o áudio é necessário existir um arquivo chamado “jump.wav” na pasta *sounds*.

Por fim, a classe Mar representa o cenário de fundo e nela inserimos todos os objetos utilizando o método *addObject* já explicado.

Outro método também foi utilizado, o **showText**. Este método mostra uma mensagem de texto na tela nas coordenadas passadas como argumento. Ainda na classe Mar temos dois métodos, o *started()* e o *stopped()*.

- O método *started()* é chamado pelo sistema Greenfoot quando a execução é iniciada – repare que dentro desse método colocamos uma chamada para tocar uma música: isso tocará uma música de fundo (para tanto, é necessário existir um arquivo chamado “som.mid” na pasta *sounds*).
- O método *stopped()* é chamado pelo sistema quando a execução é interrompida/pausada. Repare que dentro desse método colocamos uma chamada para interromper a música.

Você poderá acessar essa implementação em Greenfoot (2020g).

REFLITA

No código mostrado na Figura 1.23, vemos que especificamente a classe PeixeListrado apresenta uma variável lógica usada para controle, chamada *isMovingToRight*. Reflita sobre o que acontece quando o seu valor lógico é verdadeiro e o que ocorre quando seu valor é falso. Ainda nessa classe existe um método que pega a posição do peixe na coordenada x, chamado *getX()*. Tente entender o porquê das comparações entre a posição x do peixe com o valor 540 e com o valor 60. Reflita também sobre qual a ação executada pelo método *mirrorHorizontally()* chamado a partir de *getImage()*.

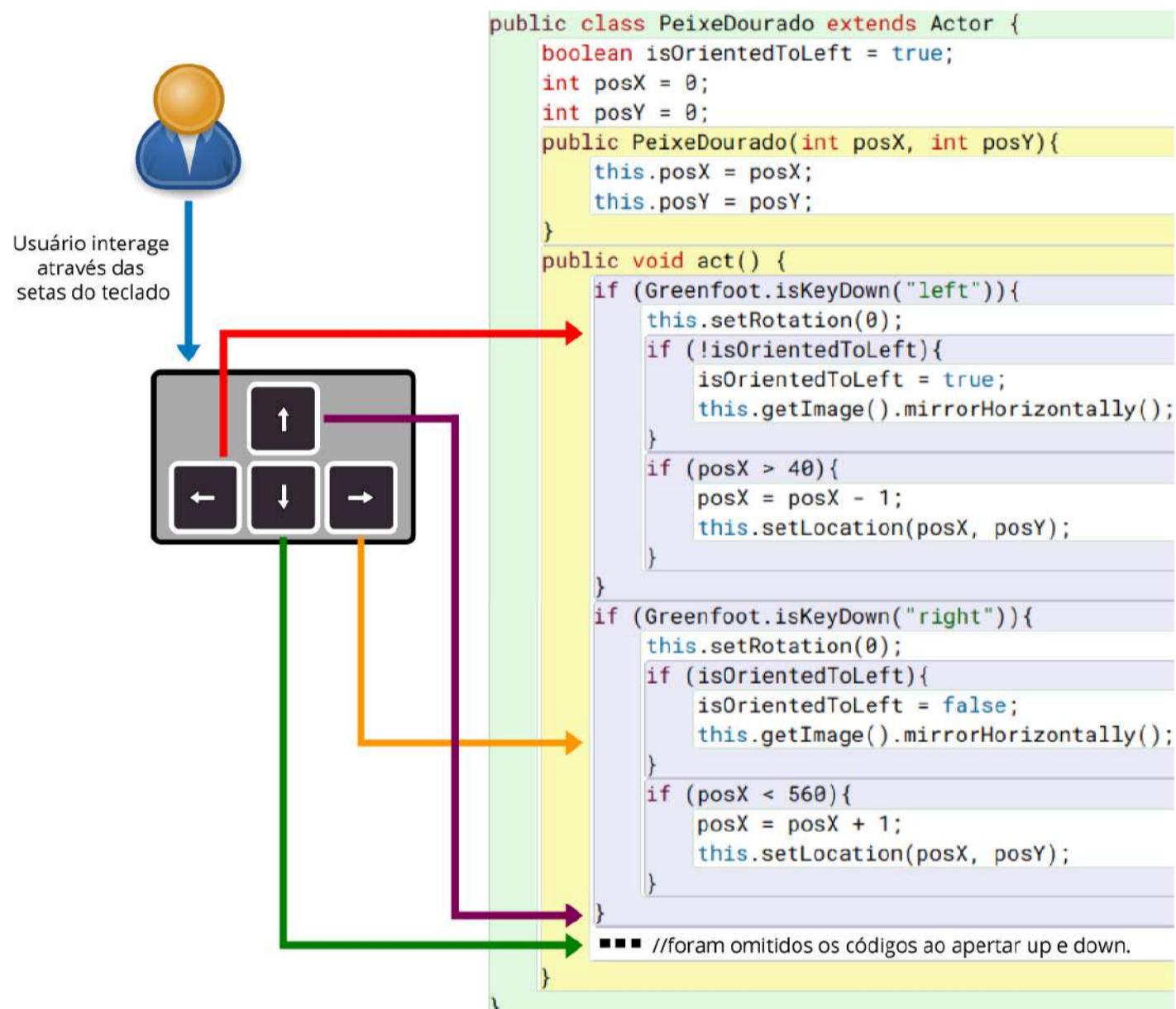
IMPLEMENTANDO UM MECANISMO DE INTERAÇÃO

Você deve ter reparado que até aqui falamos da criação de jogos, mas foi feita apenas uma pequena animação. Para termos um jogo propriamente dito, precisamos de alguma interação. A seguir aprenderemos a implementar um mecanismo de interação entre o jogador e o personagem.

EXEMPLIFICANDO

Considere o seguinte trecho de código, mostrado na Figura 1.24, que efetua o controle das ações de deslocamento do ator/personagem PeixeDourado. Diferentes ações de movimentação foram inseridas de forma a responder as teclas esquerda, direita, baixo e cima. Ao inserir esses comandos no nosso personagem ele fica mais dinâmico, melhorando a jogabilidade.

Figura 1.24 | Especificação das ações executadas pelo personagem



Fonte: captura de tela do software Greenfoot elaborada pelo autor.

A classe `PeixeDourado` tem alguns atributos que auxiliam no controle do nosso personagem. Esses atributos são a posição (x, y) e a orientação do personagem. De forma geral, o que esse código faz é deslocar o personagem para a esquerda ao pressionar do teclado a seta para esquerda. O código garante que a imagem do peixe esteja na mesma orientação que foi pressionada a tecla. Por sua vez, ao pressionar a tecla para a direita, a imagem é espelhada horizontalmente e o peixe se movimentará para a direita, desde que não esteja próximo à borda do cenário. Os códigos de movimentação para baixo e para cima foram omitidos, mas são semelhantes aos anteriores. Você poderá acessar essa implementação em Greenfoot (2020).

Nesta seção você conheceu um pouco da ferramenta Greenfoot, que auxilia no desenvolvimento de jogos e animações 2D. Vimos como inserir objetos, definir comportamentos e executar comandos dados pelo jogador. Com essa pequena base você já terá capacidade de criar um jogo do zero. Alguns comandos e sintaxes aqui utilizados ficarão mais claros ao longo deste livro, assim, não se preocupe caso você não os tenha compreendido totalmente. Lembre-se: todos os projetos aqui desenvolvidos estão disponíveis no GitHub e podem ser acessados (ARANTES, 2020). Agora que você já tem uma visão geral do que pode ser desenvolvido com a linguagem Java, falaremos mais da sua sintaxe.

PESQUISE MAIS

Existem diversas videoaulas na internet que ensinam a construir aplicações utilizando a ferramenta Greenfoot. Existe um canal no YouTube, chamado *Channel Greenfoot*, que apresenta dezenas de vídeos explicando como criar aplicações utilizando o software. Esses vídeos dão uma ótima visão sobre a construção de jogos utilizando a ferramenta.

CHANNEL GREENFOOT. Página inicial. Youtube, [s.d.]. Disponível em: <https://www.youtube.com/user/18km>. Acesso em: 9 maio 2020.

REFERÊNCIAS

ARANTES, J. S. **Livro**: Linguagem Orientada a Objetos. Github, 2020. Disponível em: <https://bit.ly/3eiUMcF> . Acesso em: 9 maio 2020.

CHANNEL GREENFOOT. Página inicial. Youtube, [s.d.]. Disponível em: <https://bit.ly/2Ov8E9f> . Acesso em: 9 maio 2020.

FREESOUND. Página inicial. Freesound, 2020. Disponível em: <https://bit.ly/308vOrD> . Acesso em: 8 maio 2020.

GREENFOOT. **33 Seconds**. Greenfoot scenarios, 26 jun. 2019. Disponível em: <https://bit.ly/30cV7Zp> . Acesso em: 10 maio 2020.

GREENFOOT. **Account**. Greenfoot, [s.d.]c. Disponível em: <https://bit.ly/38To6Wk> . Acesso em: 10 maio 2020.

GREENFOOT. **Animação Simples – Peixe**. Greenfoot scenarios, 10 maio 2020a. Disponível em: <https://bit.ly/38TXgx8> . Acesso em: 10 maio 2020.

GREENFOOT. **Animação Simples – Peixes**. Greenfoot scenarios, 10 maio 2020f. Disponível em: <https://bit.ly/32oae5j> . Acesso em: 10 maio 2020.

GREENFOOT. **Basic 3x3 Cube.** Greenfoot scenarios, 16 mar. 2020b. Disponível em: <https://bit.ly/2Czs36m> . Acesso em: 10 maio 2020.

GREENFOOT. **ChessWorld.** Greenfoot scenarios, 7 maio 2011. Disponível em: <https://bit.ly/3j0fUII> . Acesso em: 10 maio 2020.

GREENFOOT. **Dawn of War 2.** Greenfoot scenarios, 10 maio 2020e. Disponível em: <https://bit.ly/2ZrXkRz> . Acesso em: 10 maio 2020.

GREENFOOT. **Memory Tester.** Greenfoot scenarios, 15 fev. 2014. Disponível em: <https://bit.ly/2CBhlMA> . Acesso em: 10 maio 2020.

GREENFOOT. **Mundo Subaquático - v1.** Greenfoot scenarios, 14 maio 2020g. Disponível em: <https://bit.ly/2OrbVXq> . Acesso em: 14 maio 2020.

GREENFOOT. **Mundo Subaquático - v2.** Greenfoot scenarios, 14 maio 2020h. Disponível em: <https://bit.ly/2ZvsvvD> . Acesso em: 14 maio 2020.

GREENFOOT. **Package greenfoot.** Greenfoot Javadoc, [s.d.]b. Disponível em: <https://bit.ly/2OoVBX5> . Acesso em: 10 maio 2020.

GREENFOOT. Página inicial. [S.d.]a. Disponível em: <https://bit.ly/2OIAx3E> . Acesso em: 7 maio 2020.

GREENFOOT. **Roadkill.** Greenfoot scenarios, 27 fev. 2020c. Disponível em: <https://bit.ly/2ZqWCnK> . Acesso em: 10 maio 2020.

GREENFOOT. **Scenarios.** Greenfoot, [s.d.]b. Disponível em: <https://bit.ly/38TucWL> . Acesso em: 10 maio 2020.

GREENFOOT. **Simulação RATM.** Greenfoot scenarios, 14 maio 2020i. Disponível em: <https://bit.ly/305su0m> . Acesso em: 14 maio 2020.

GREENFOOT. **World War 3.** Greenfoot scenarios, 21 jan. 2020d. Disponível em: <https://bit.ly/3fGO6XH> . Acesso em: 10 maio 2020.

OPENGAMEART. Página inicial. [S.d.]. Disponível em: <https://bit.ly/2Wfz0R2> . Acesso em: 8 maio 2020.

ORACLE. **Java SE Downloads.** Oracle, 2020. Disponível em: <https://bit.ly/3iKLytv> . Acesso em: 5 maio 2020.

FOCO NO MERCADO DE TRABALHO

UMA VISÃO DE GAME EM PROGRAMAÇÃO ORIENTADA A OBJETOS

Jesimar da Silva Arantes

Ver anotações

UTILIZANDO JOGOS DE SIMULAÇÃO PARA DEMONSTRAR A EFICIÊNCIA DO ROBÔ R-ATM

Para criar jogos 2D, é necessário definir cenários, utilizar atores/personagens, posicionar os atores no cenário, definir comportamentos, adicionar recursos de áudio e criar interações com o personagem.



Fonte: Shutterstock.

Deseja ouvir este material?

Áudio disponível no material digital.

SEM MEDO DE ERRAR

A *startup* em que trabalha passou a você um grande desafio: fazer um jogo/simulador 2D para demonstrar algumas funcionalidades do robô que sua empresa está desenvolvendo. As principais funcionalidades que o jogo/simulação deve contemplar são:

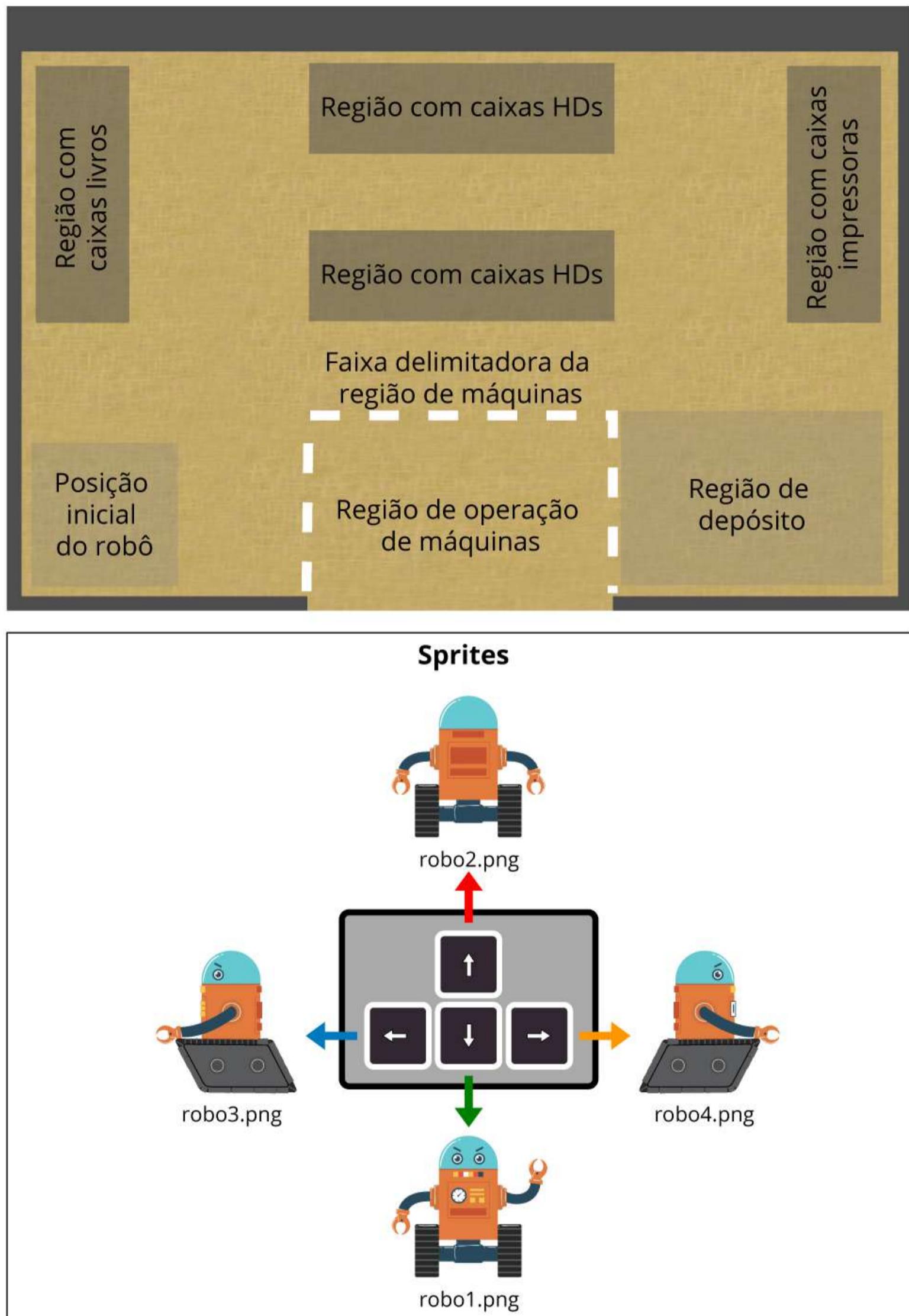
- O robô deve se mover.
- O robô deve evitar regiões da sala em que existem máquinas trabalhando.

- O robô não pode se deslocar sobre as caixas presentes no cenário.
- O robô deve apresentar duas velocidades de deslocamento.
- A sala deve ter um formato retangular.
- A visualização do robô deve ser em terceira pessoa.

Dado esse desafio, algumas perguntas podem surgir: qual ferramenta utilizar? Será que o Greenfoot consegue modelar esse desafio? Como definir as regiões proibidas ao deslocamento? Como fazer o robô não se movimentar sobre as caixas? Qual imagem de background utilizar? Será que há alguma imagem pronta? Será que existe alguma imagem de robô já modelado ou terei que criar as minhas imagens? Como você fará para criar o jogo em terceira pessoa?

Após analisar bem o desafio proposto, você chega à conclusão de que a ferramenta Greenfoot consegue resolver bem esse problema. Então, o primeiro passo que deve ser feito é a definição de como deve ser a sala/galpão. Assim, você monta uma sala, conforme a Figura 1.25, em que você especifica onde deverão ficar cada um dos itens do jogo. O próximo passo é baixar algumas imagens de caixas, pois o Greenfoot não dispõe de nenhuma do seu agrado. Você decide também desenvolver as próprias imagens de robô, pois o Greenfoot não tem nenhuma do seu agrado. Assim, você desenvolve quatro imagens do robô, conforme Figura 1.25, cada uma associada ao deslocamento para um dos lados (baixo, frente, esquerda e direita). A este tipo de imagens utilizadas em jogos damos o nome de *sprite*.

Figura 1.25 | Projeto da sala/galpão onde o robô operará e sprites do robô

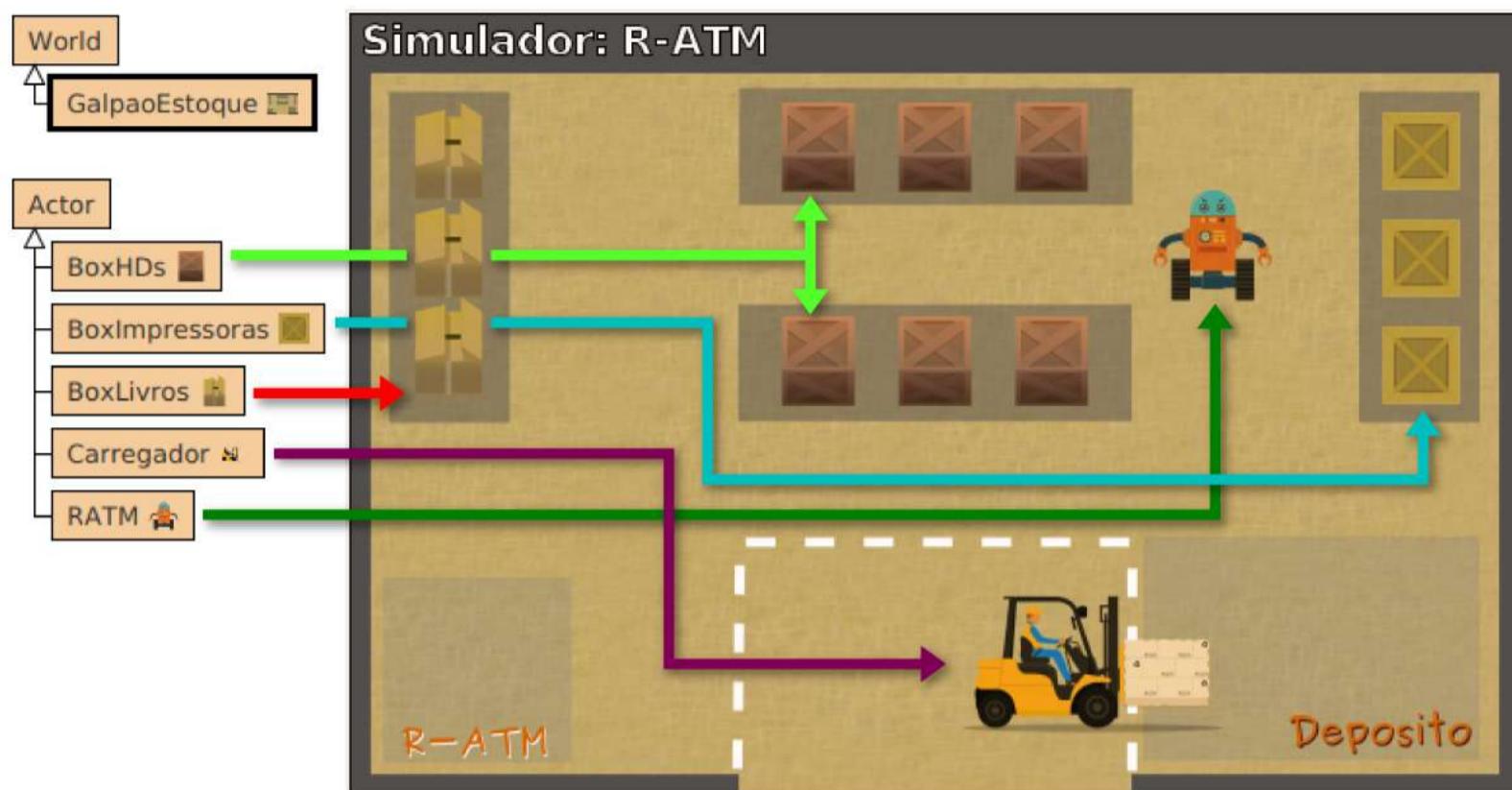


Após resolvida a questão das imagens a serem utilizadas, chega-se a uma solução semelhante à apresentada na Figura 1.26. Essa imagem mostra o cenário e os atores criados, bem como as suas posições.

0

Ver anotações

Figura 1.26 | Cenário da situação-problema com os atores



Fonte: captura de tela do software Greenfoot elaborada pelo autor.

A Figura 1.27 mostra alguns trechos de códigos implementados para resolver o problema de deslocamento do robô por algumas regiões do cenário. Alguns trechos foram omitidos. Essa figura mostra também o código utilizado para fazer a mudança de velocidade do robô, acionada pela tecla espaço. Você pode ver, jogar e acessar o código desta situação-problema Greenfoot (2020i).

Figura 1.27 | Código para controle das regiões de deslocamento do robô e velocidade

```
public class RATM extends Actor {
    String imgRoboFrente = "robo1.png";
    String imgRoboTraz = "robo2.png";
    String imgRoboEsquerda = "robo3.png";
    String imgRoboDireita = "robo4.png";
    int velocidade = 1;
    int posX = 0;
    int posY = 0;
    public void act() {
        if (Greenfoot.isKeyDown("down")){
            this.setImage(imgRoboFrente);
            if (avaliaPosicao(posX, posY + velocidade)){
                posY = posY + velocidade;
                this.setLocation(posX, posY);
            }
        }
        if (Greenfoot.isKeyDown("up")){
            this.setImage(imgRoboTraz);
            if (avaliaPosicao(posX, posY - velocidade)){
                posY = posY - velocidade;
                this.setLocation(posX, posY);
            }
        }
        ■■■ foram omitidos os códigos ao apertar left e right.
        if (Greenfoot.isKeyDown("space")){
            if (velocidade == 1){
                velocidade = 2;
            } else if (velocidade == 2){
                velocidade = 1;
            }
        }
    }
}

private boolean avaliaPosicao(int posX, int posY){
    if (posX < 40 || posX > 560 || posY < 60 || posY > 360){
        return false;//delimitação da fronteira da sala/galpão
    }
    if ((posX >= 170 && posX <= 430) && (posY >= 240 && posY <= 400)){
        return false;//delimitação da região de operação de máquinas
    }
    if ((posX >= 0 && posX <= 100) && (posY >= 0 && posY <= 200)){
        return false;//delimitação da região de caixas com livros
    }
    if ((posX >= 500 && posX <= 600) && (posY >= 0 && posY <= 200)){
        return false;//delimitação da região de caixas com impressoras
    }
    if ((posX >= 170 && posX <= 430) && (posY >= 0 && posY <= 90)){
        return false;//delimitação da região de caixas com HDs acima
    }
    if ((posX >= 170 && posX <= 430) && (posY >= 120 && posY <= 200)){
        return false;//delimitação da região de caixas com HDs abaixo
    }
    return true;
}
```

Fonte: capturas de telas do software Greenfoot elaboradas pelo autor.

Após terminado o jogo/simulação, você decide mostrá-lo ao seu chefe. Ele gostou muito, e o apresenta para a empresa de *e-commerce*. A empresa também gostou bastante, e está muito animada com o andamento do projeto.

0

[Ver anotações](#)

CONSTRUTORES E SOBRECARGA

Jesimar da Silva Arantes

0

[Ver anotações](#)

ATRIBUTOS E MÉTODOS ESTÁTICOS, CONSTRUTORES, SOBRECARGA E SOBREPOSIÇÃO DE MÉTODOS

Esses conceitos auxiliam de forma direta, na modelagem de problemas do mundo real, trazendo uma grande flexibilidade na manipulação de classes e objetos.



Fonte: Shutterstock.

Deseja ouvir este material?

Áudio disponível no material digital.

CONVITE AO ESTUDO

Prezado estudante, bem-vindo à segunda unidade do livro *Linguagem Orientada a Objetos*. Nesta unidade, estudaremos as principais estruturas do desenvolvimento Orientado a Objetos (OO), bem como estabeleceremos as ideias por trás de atributos e métodos estáticos, construtores, sobrecarga e sobreposição de métodos, assim, após a leitura, você será capaz de identificar quando declarar atributo e método de forma estática. Além disso, mostraremos como fazer a criação de diversos construtores em uma única classe e as formas como esses construtores são invocados, como se criar a sobrecarga e a sobreposição de

métodos e uma comparação entre esses conceitos. As estruturas de controle, como tomada de decisão e repetição, também serão estudadas, assim como as teorias por trás da herança, polimorfismo e encapsulamento. Entenderemos, por fim, o quanto esses conceitos auxiliam, de forma direta, na reutilização de código e o tornam mais organizado.

o

[Ver anotações](#)

o

Ver anotações

Após a leitura desta unidade, espera-se que o leitor seja capaz de criar os seus primeiros programas Java dentro de um Ambiente de Desenvolvimento Integrado (IDE) poderoso, sendo que um bom IDE provê um local de trabalho em que os aspectos de produtividade são acelerados. Espera-se, também, que o leitor comece a se familiarizar com as plataformas de compartilhamento de código, como o GitHub, e este livro apresenta uma série de exemplos práticos que auxiliam na compreensão da teoria. A capacidade de reflexão, modelagem e abstração será treinada ao longo do estudo.

Esta unidade o ajudará na compreensão da OO dentro da linguagem Java a partir das seguintes seções: a Seção 2.1 apresentará os conceitos por trás dos atributos e métodos estáticos, métodos construtores, sobrecarga e sobreposição de métodos. A Seção 2.2 trará as estruturas de controle como estruturas de decisão e repetição, bem como apresentará os operadores aritméticos, lógicos e relacionais. A Seção 2.3, por fim, mostrará as ideias relacionadas à reutilização de código, como encapsulamento, herança e polimorfismo em programação orientada a objetos.

Sendo esta uma unidade prática, convido você a implementar as atividades aqui propostas. Lembre-se: a prática leva à perfeição.

Bons estudos!

PRATICAR PARA APRENDER

Caro estudante, bem-vindo à primeira seção da segunda unidade dos estudos sobre linguagem orientada a objetos. Qual desenvolvedor nunca se perguntou como construir uma aplicação que tenha grande flexibilidade na invocação de atributos, construtores e métodos? Pois bem, a ideia-chave aqui é a modelagem a partir dos conceitos por trás da Orientação a Objetos (OO). Nesta seção, você terá a oportunidade de aprender os conceitos e as ideias sobre atributos e métodos estáticos, construtores, sobrecarga e sobreposição de métodos. Esses conceitos, por sua vez, o auxiliarão, de forma direta, na modelagem de problemas do mundo real, trazendo uma grande flexibilidade na manipulação de classes e objetos.

Como forma de contextualizar sua aprendizagem, tenha em mente a *startup* pela qual você foi contratado. O seu chefe deseja dar continuidade à modelagem do robô que você iniciou na primeira seção da primeira unidade; ele acredita que a modelagem desenvolvida pode ser melhorada ao se incorporar novos recursos que embasem, cada vez mais, o simulador de robô que você deve construir. Assim,

após o seu chefe revisar todo o código que você desenvolveu, ele pediu a você que criasse quatro novos construtores para a classe robô, bem como definisse alguns atributos como constantes, argumentando que isso impedirá eventuais erros no código.

0

[Ver anotações](#)

o

Ver anotações

Quanto ao método *move* que você criou, ele gostou muito, por isso, quer que você faça a sua sobrecarga, criando novas opções para que seu robô se desloque no cenário. O seu chefe percebeu que você colocou o ponto de entrada da aplicação (método *main*) dentro da classe robô, no entanto, ele não gostou disso e quer que você o coloque em outra classe. Além disso, ele também percebeu que você está programando a sua aplicação na nuvem (internet), utilizando compilador web, porém, sabendo que a sua aplicação crescerá muito ao longo do tempo, pediu para que instalasse algum IDE em seu computador local, justificando que o IDE instalado localmente lhe dará um suporte melhor durante o seu desenvolvimento.

Diante do desafio que lhe foi apresentado, como você criará esses construtores para o seu robô? Como fará para tornar alguns de seus atributos constantes? Como irá fazer a sobrecarga do método *move*? O que é um construtor? O que é sobrecarga de método? Como você colocará a aplicação principal em outra classe? Qual IDE você irá instalar?

Esta seção o auxiliará nas respostas de tais perguntas.

Muito bem, agora que você foi apresentado à sua nova situação-problema, estude esta seção e compreenda como a OO pode auxiliá-lo na modelagem de diferentes construtores, constantes e sobrecarga de métodos. Esses conceitos serão fundamentais na OO e trarão uma maior sofisticação à construção do simulador de robô que você deve fazer. E aí, vamos juntos compreender esses conceitos para resolver esse desafio?

Bom estudo!

CONCEITO-CHAVE

Uma Linguagem Orientada a Objetos (OO) utiliza diversas abstrações para representar modelos do mundo real ou imaginário. Na unidade anterior, aprendemos como se dá a construção de aplicações básicas em Java a partir do site *jdoode*, bem como a de animações tridimensionais a partir da ferramenta Alice e de jogos bidimensionais com a ferramenta *Greenfoot*. As ferramentas Alice e *Greenfoot*, na verdade, são mais do que ferramentas, são Ambientes de Desenvolvimento Integrado (IDE).

O ambiente de programação online *jdoode* atende bem às necessidades do programador quanto à construção de um programa ou sistema simples e pequeno. Imagine, agora, que desejamos desenvolver um sistema grande com

diversas classes e com muitas linhas de código. Apesar de ser possível desenvolvê-lo por meio do jdoodle, ele não é tão adequado, sendo necessária a utilização de um IDE mais poderoso.

0

[Ver anotações](#)

IDE é um ambiente que auxilia muito a vida do programador, e um bom ambiente de desenvolvimento colabora com diversos recursos, como:

- **Autocompletar:** todos os possíveis comandos são apresentados conforme você digita.
- **Realce de sintaxe:** o código fica colorido, ressaltando as variáveis, os métodos, a classe, as palavras reservadas, entre outros.
- **Indicativos de erros:** as linhas com erros léxicos e sintáticos são destacadas.
- **Depuração de código (*debug*):** auxilia na verificação do correto funcionamento de algum programa, reduzindo defeitos.
- **Visualização da documentação (*javadoc*) dos métodos e de classes:** podemos consultar, de forma rápida, a descrição de como se utilizar o programa.
- **Refatoração de código:** podemos alterar nomes de variáveis, métodos e classes de forma rápida e automatizada.
- **Geração de executável:** um arquivo .jar é criado e pode ser executado em qualquer computador que tenha a *Java Virtual Machine* (JVM) instalada.
- **Testes de software:** podemos criar testes automatizados de classes e métodos de forma a fazer a verificação e validação dos mesmos.

DICA

Pesquise mais sobre os IDEs Eclipse, IntelliJ IDEA e NetBeans e selecione e instale o IDE que mais se adequa ao seu perfil. Todos são igualmente bons.

ATRIBUTOS E MÉTODOS ESTÁTICOS

Imagine que você gostaria de modelar uma classe chamada *Matematica* para resolver problemas relacionados a essa área do conhecimento. Nessa classe, você gostaria que houvesse as seguintes constantes: número pi (PI), número de Euler (E) e número de ouro (PHI). A constante PI é famosa na matemática e representa a razão entre o comprimento da circunferência e o seu diâmetro; já a constante E representa a base dos logaritmos naturais; e a constante PHI representa uma proporção encontrada em diversas aplicações na natureza. Além disso, você também gostaria que houvesse dois métodos: um para a soma e outro para a multiplicação de inteiros, logo, decide implementar o Código 2.1.

Código 2.1 | Versão inicial do programa em Java que modela a classe *Matematica*

```
1  public class Matematica {  
2      double PI = 3.1415926535;  
3      double E = 2.7182818284;  
4      double PHI = 1.6180339887;  
5      int soma(int a, int b){  
6          return a + b;  
7      }  
8      int mult(int a, int b){  
9          return a * b;  
10     }  
11     public static void main(String[] args) {  
12         Matematica mat = new Matematica();  
13         System.out.println("Numero Pi: " + mat.PI);  
14         System.out.println("Numero E: " + mat.E);  
15         System.out.println("Numero Phi: " + mat.PHI);  
16         System.out.println("Soma: " + mat.soma(5, 3));  
17         System.out.println("Mult: " + mat.mult(5, 3));  
18     }  
19 }
```

0

Ver anotações

Fonte: elaborado pelo autor.

As linhas 2, 3 e 4, do Código 2.1, possuem a declaração das constantes matemáticas mencionadas acima; já as linhas 5 a 10 possuem a declaração dos métodos de soma e de multiplicação; por fim, nas linhas de 11 a 18, temos a função *main*, em que fizemos a utilização da classe *Matematica*. Você deve ter reparado que tivemos que criar um objeto da classe *Matematica* para a invocação das constantes (PI, E e PHI) e dos métodos *soma* e *mult*.

LEMBRE-SE

Os códigos aqui apresentados devem ser implementados em algum IDE que você escolheu e salvou utilizando-se a extensão .java; já o nome do arquivo deve ser o mesmo nome da classe.

A forma como modelamos a classe *Matematica* não ficou boa, pois, toda vez que tivermos de invocar uma constante ou método dessa classe, teremos de criar um objeto. Repare que essas constantes e os métodos devem ter o mesmo comportamento, independentemente da instância de que fazem parte (eles não manipulam atributos específicos da classe). A melhor forma, neste caso, de implementação das constantes e dos métodos é declará-los como estáticos, evitando-se a construção de um objeto para a invocação de constantes e métodos. Considere o Código 2.2 abaixo.

o

[Ver anotações](#)

```

1  public class Matematica {
2      static final double PI = 3.1415926535;
3      static final double E = 2.7182818284;
4      static final double PHI = 1.6180339887;
5      static int soma(int a, int b){
6          return a + b;
7      }
8      static int mult(int a, int b){
9          return a * b;
10     }
11     public static void main(String[] args) {
12         System.out.println("Número Pi: " + Matematica.PI);
13         System.out.println("Número E: " + Matematica.E);
14         System.out.println("Número Phi: " + Matematica.PHI);
15         System.out.println("Soma: " + Matematica.soma(5, 3));
16         System.out.println("Mult: " + Matematica.mult(5, 3));
17     }
18 }
```

0

Ver anotações

Fonte: elaborado pelo autor.

Nas linhas 2 a 4 foi acrescentada a palavra reservada *static*, que modifica o comportamento das constantes PI, E e PHI. Devido a essa modificação, as constantes são, agora, acessadas sem que se tenha que declarar um objeto da classe *Matematica*, escrevendo apenas *Matematica.PI*, *Matematica.E* e *Matematica.PHI*, respectivamente. Repare também que, nas linhas 2 a 4, foi acrescentada a palavra reservada *final*, que transforma, de fato, os atributos PI, E e PHI em constantes. Essa modificação é importante, uma vez que tais valores nunca deveriam ser alterados, evitando-se possíveis tentativas de mudanças. A palavra reservada *final* em Java tem algumas semelhanças com a palavra reservada *const* em C.

ATENÇÃO

A tentativa de atribuição de um valor a um atributo constante gera um erro de compilação. Lembre-se: um valor constante nunca pode ser modificado. Assim, o seguinte código gera um erro de compilação.

```
final int ID = 1000;  
ID = 1001; //Erro. Atribuição não permitida.
```

0

[Ver anotações](#)

o

Ver anotações

Ainda em relação ao Código 2.2, as linhas 5 a 10 redefiniram os métodos *soma* e *mult* como estáticos, tornando-os, assim, métodos da classe. Dessa maneira, a forma de acesso a esses métodos foi alterada e não é mais necessária a criação de objetos. Nas linhas 15 e 16, a chamada é feita com o nome da própria classe, como *Matematica.soma(...)* e *Matematica.mult(...)*; os métodos estáticos são também designados de métodos da classe ou, simplesmente, funções, e a principal diferença entre métodos e funções está relacionada a quem faz a invocação: se invocado por um objeto (instância da classe), trata-se de método, se invocado pelo nome da Classe, então, é uma função (método estático). Talvez, o exemplo mais importante de classe composta por apenas métodos estáticos seja a classe *Math* (veremos mais exemplos dessa classe adiante).

DICAS

A linguagem Java possui uma série de convenções de nomes de variáveis, constantes, métodos, classes, objetos e pacotes. Lembre-se: a linguagem Java é *case-sensitive*, ou seja, os nomes utilizados são sensíveis a letras maiúsculas e minúsculas. As variáveis são convencionalmente nomeadas com letras minúsculas (exceto quando começar com _ ou \$), e caso sejam compostas por mais uma palavra, esta deverá começar com letra maiúscula, notação chamada de *lowerCamelCase*. Já as constantes são nomeadas com todas as letras maiúsculas, mas em caso de mais de uma palavra, estas deverão ser separadas por *underline*, notação designada por SCREAMING_SNAKE_CASE.

Os métodos seguem o mesmo padrão de nomeação de variáveis (*lowerCamelCase*); já as classes são nomeadas com a primeira letra de cada palavra maiúscula e as demais minúsculas, notação chamada de *UpperCamelCase*. Os objetos, por sua vez, seguem o mesmo padrão de nomeação de variáveis (*lowerCamelCase*); já os pacotes devem possuir letras minúsculas, não sendo recomendada a utilização de caracteres especiais, mas, se necessário, fazer o uso do *underline*. Por exemplo, os seguintes nomes são adequados:

Variáveis: i, _x, \$y, maximo, nomeRobo, idadePessoa, areaQuadradoMaior.

Constantes: PI, E, PHI, INDEX, ID_PESSOA, VALOR_CONST_ARRASTO.

Métodos: soma, mult, convertString, calculaQuadrado, calculaMaiorValor.

Classes: Robo, Animal, SimulacaoRobo, CadastraPessoaBancoDados.

Objetos: meuRobo, bob, simRobo, cadastrapessoBD.

Pacotes: src, util, view, window, resources, operadores, src_imgs.

MÉTODOS CONSTRUTORES

O próximo assunto que iremos tratar diz respeito aos métodos construtores. Toda classe em Java possui um método construtor padrão que possui o mesmo nome da classe e não recebe nenhum argumento. O código 2.1, apesar de não ter sido escrito a classe *Matematica*, possui um construtor padrão da seguinte forma:

```
public Matematica(){
}
```

Esse construtor sempre existirá de forma explícita (como mostrado acima) ou implícita (como no Código 2.1). É importante reparar que a declaração de um construtor deve sempre possuir o mesmo nome da classe; que não existe nenhum tipo de retorno, nem mesmo a palavra *void* pode ser especificada; e que alguns outros modificadores de acesso são possíveis, além do *public*, mas veremos isso nas seções seguintes.

A fim de que entenda melhor os métodos construtores, vamos criar uma nova classe em Java que modela a entidade *Triangulo*. O Código 2.3 nos mostra como podemos modelar essa classe, para isso, nas linhas 2 a 4 foram definidos três atributos que representam os lados do triângulo. Da linha 5 até a 9, temos o primeiro construtor, que recebe uma medida de lado de um triângulo e atribui esse valor a todos os lados. Como esse construtor possui parâmetro, o chamamos de construtor não padrão. Dessa forma, repare que esse construtor será chamado quando o nosso triângulo for isósceles, pois os três lados serão iguais. Da linha 10 até a 14, temos um segundo construtor, que recebe três argumentos, representando os lados a, b e c. As linhas 15 até 19 não serão explicadas por enquanto, apenas assuma que elas auxiliam na impressão da informação na tela. Por fim, temos o método *main*, que é o ponto de entrada da aplicação. Nele, temos a construção de dois objetos do tipo triângulo, em que o primeiro é do tipo isósceles (com lados iguais a 5) e o segundo do tipo escaleno (com lados iguais a 3, 25 e 26). Neste exemplo, criamos dois construtores, porém podemos ter quantos construtores quisermos em nossa aplicação.

Código 2.3 | Modelagem da classe *Triangulo* com dois construtores

0

```
1  public class Triangulo {  
2      double ladoA;  
3      double ladoB;  
4      double ladoC;  
5      public Triangulo(double lado) {  
6          this.ladoA = lado;  
7          this.ladoB = lado;  
8          this.ladoC = lado;  
9      }  
10     public Triangulo(double a, double b, double c) {  
11         this.ladoA = a;  
12         this.ladoB = b;  
13         this.ladoC = c;  
14     }  
15     @Override  
16     public String toString() {  
17         return String.format("a: %.2f\nb: %.2f\nc: %.2f",  
18                         this.ladoA, this.ladoB, this.ladoC);  
19     }  
20     public static void main(String[] args) {  
21         Triangulo trianIsosceles = new Triangulo(5);  
22         Triangulo trianEscaleno = new Triangulo(3, 25, 26);  
23         System.out.println(trianIsosceles);  
24         System.out.println(trianEscaleno);  
25     }  
26 }
```

Ver anotações

Fonte: elaborado pelo autor.

ASSIMILE

Os construtores são sempre invocados utilizando-se a palavra reservada *new* seguida do nome da classe. O construtor, como o nome já sugere, constrói a classe inicializando-a de forma implícita ou explícita aos atributos dela, bem como aloca a quantidade de memória necessária para utilização do objeto. Caso seja especificado algum tipo de processamento, este pode ser feito dentro do construtor, apesar de não muito indicado. O retorno do operador *new* é uma referência para o objeto criado.

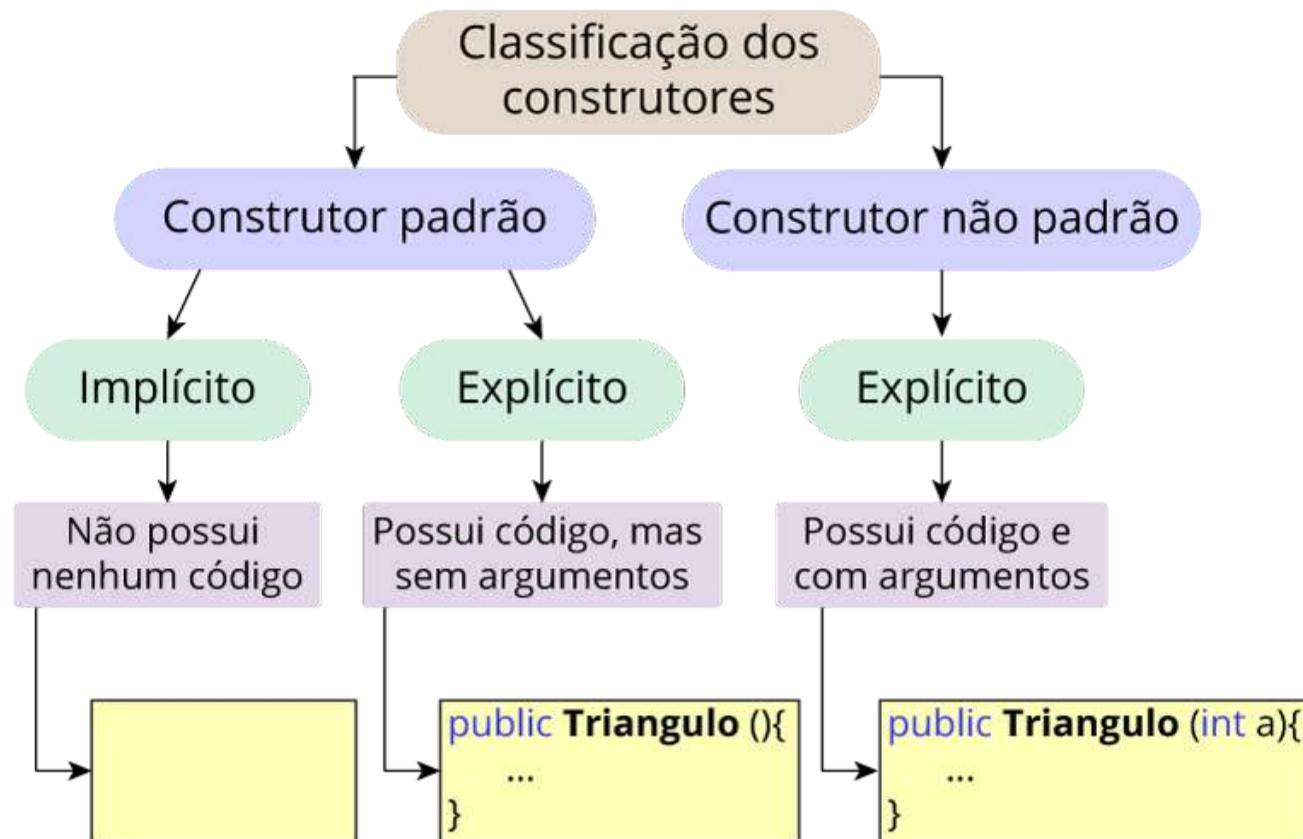
A Figura 2.1 sintetiza algumas das características dos métodos construtores.

Conforme dito, os construtores são classificados em dois tipos: **padrão** e **não padrão**, podendo, também, ser classificados como **implícitos** e **explícitos**. Os construtores explícitos devem ser codificados, de fato; já os construtores não padrão têm que possuir algum tipo de argumento em sua assinatura.

o

[Ver anotações](#)

Figura 2.1 | Organograma dos tipos de construtores presentes na linguagem Java



0

Ver anotações

Fonte: elaborada pelo autor.

ATENÇÃO

Uma observação importante sobre construtores é que, quando um construtor não padrão é declarado, automaticamente, o construtor padrão implícito deixa de existir. Assim, caso seja necessária a existência do construtor padrão, é imprescindível a sua declaração explícita.

SOBRECARGA E SOBREPOSIÇÃO DE MÉTODOS

Outro assunto muito importante dentro da orientação a objetos é a sobrecarga e sobreposição de métodos.

A sobrecarga caracteriza-se por haver mais de um método em uma mesma classe com o mesmo nome, no entanto, com diferentes tipos de dados ou diferentes quantidades de parâmetros.

Analise o Código 2.4, em que o método *mult* (responsável pela multiplicação) foi sobrecarregado. Nas linhas 2 a 4, temos o método *mult* com dois argumentos (*a*, *b*) do tipo inteiro como entrada e o resultado, um número inteiro, como saída. Nas linhas 5 a 7, temos o método *mult* com também dois argumentos (*a*, *b*), no

entanto, do tipo *double*, e o resultado como um número do tipo *double*. Por fim, nas linhas 8 a 10, temos o método *mult* com três argumentos (*a*, *b*, *c*) do tipo *double*, sendo a saída um número do tipo *double*.

```

1  public class Matematica {
2      static int mult(int a, int b){
3          return a * b;
4      }
5      static double mult(double a, double b){
6          return a * b;
7      }
8      static double mult(double a, double b, double c){
9          return a * b * c;
10     }
11     public static void main(String[] args) {
12         System.out.println("Mult: " + Matematica.mult(5, 3));
13         System.out.println("Mult: " + Matematica.mult(3.2, 4.1));
14         System.out.println("Mult: " + Matematica.mult(1.4, 2));
15         System.out.println("Mult: " + Matematica.mult(2, 3.5));
16         System.out.println("Mult: " + Matematica.mult(4.5, 5.2, 2));
17     }
18 }
```

Fonte: elaborado pelo autor.

Ainda no Código 2.4, nas linhas 11 a 17, temos o método principal da aplicação. Na linha 12, foi chamado o método *mult* com dois argumentos do tipo inteiro (5 e 3), dessa forma, qual dos três métodos *mult* você acha que será executado? Bem, a resposta é o primeiro (das linhas 2 a 4), porque os argumentos são inteiros e o primeiro método recebe argumentos do tipo inteiro. Na linha 13, foi invocado o método *mult* com dois argumentos do tipo *double* (3.2 e 4.1), frente a isso, qual dos métodos *mult* será executado? A resposta é o segundo (das linhas 5 a 7), pois, como os argumentos são do tipo *double*, então, a única assinatura de método que casa com esse padrão é a segunda. As linhas 14 e 15 também executam chamadas para o segundo método *mult*, e nesse caso, como apenas um dos argumentos é do tipo *double* e o outro é do tipo inteiro, então, nenhum método definido acima casa exatamente com esse padrão, porém, como o tipo *int* é um caso particular do tipo *double*, logo, o segundo *mult* será executado. Por fim, na linha 16, temos uma invocação do método *mult* com três argumentos do tipo *double*. Nesse caso, o único método *mult* que casa com esse padrão é o terceiro (das linhas 8 a 10).

Uma pergunta que você pode fazer relacionada à sobrecarga de métodos é: em qual momento o compilador irá decidir qual método executar? Bem, a resposta é que o compilador fará tal decisão em tempo de compilação. Um leitor atento deve ter reparado que o que fizemos com os construtores no código 2.3 foi um tipo de sobrecarga, só que não de métodos, mas sim de construtores. Assim, o construtor *Triangulo(double lado)* e *Triangulo(double a, double b, double c)* foram sobreescarregados, pois o que muda de um para o outro é a quantidade de parâmetros.

o

[Ver anotações](#)

REFLITA

No texto acima, mostramos uma série de exemplos de atributos estáticos e métodos estáticos, bem como de sobrecarga de métodos. Acesse o conteúdo Class Math, do site Oracle, cujo link está no tópico Referências, e reflita sobre como foram declaradas as constantes; o porquê de todos os métodos dessa classe serem estáticos; da maioria dos métodos dessa classe possuir sobrecarga; do método *abs* (que calcula o valor absoluto) possuir quatro assinaturas, que são: *static double abs(double a)*, *static float abs(float a)*, *static int abs(int a)* e *static long abs(long a)*; do método *cos* (que calcula o cosseno) possuir apenas uma assinatura, que é: *static double cos(double a)*; de **não** existirem as assinaturas *static int cos(int a)*, *static long cos(long a)* e *static float cos(float a)*; e, por fim, repare na padronização dos nomes de constantes e métodos presentes nessa classe. Repare que eles seguem a mesma padronização de nomes sugerida neste livro.

0

Ver anotações

Vamos, agora, estudar a sobreposição de métodos.

A sobreposição também chamada de sobrescrita de métodos, se caracteriza por possuir mais uma classe, em que a primeira herda a segunda.

Neste momento, vamos abstrair alguns detalhes da herança, que será melhor abordada na seção 3 desta unidade. Analise o Código 2.5 abaixo, em que temos duas classes.

Código 2.5 | Modelagem da classe *Triangulo* com sobreposição do método *calcPerimetro*

```
1 public class Geom2D {  
2     double perimetro;  
3     double calcPerimetro(){  
4         return 0;  
5     }  
6 }  
7  
8 public class Triangulo extends Geom2D {  
9     ...  
10    @Override  
11    double calcPerimetro() {  
12        super.perimetro = this.ladoA + this.ladoB + this.ladoC;  
13        return super.perimetro;  
14    }  
15    public static void main(String[] args) {  
16        Triangulo trian = new Triangulo(3, 25, 26);  
17        System.out.println("Perímetro: " + trian.calcPerimetro());  
18    }  
19 }
```

0

Ver anotações

Fonte: elaborado pelo autor.

A primeira classe, nomeada de *Geom2D*, está descrita nas linhas 1 a 6. Aqui, é importante destacarmos que esse código deve estar em um arquivo chamado *Geom2D.java* e que a segunda classe nomeada de *Triangulo* está descrita nas linhas 8 a 19. Essa classe também deve estar em um arquivo separado, chamado *Triangulo.java*. A primeira classe cria um atributo chamado perímetro e um método que calcula o perímetro (*calcPerimetro*). Esse método, inicialmente, retorna zero, mas essa não é a melhor forma de criação deste método, veremos depois como transformá-lo em um método abstrato. A segunda classe herdou a classe *Geom2D* (utilizando a palavra reservada *extends*) e, em seguida, algumas linhas de códigos foram omitidas (podemos ver essas linhas no Código 2.3). Das linhas 10 a 14, temos a sobreposição do método *calcPerimetro*. O método sobreescrito possui a mesma assinatura (*double calcPerimetro()*) do método da sua superclasse (*Geom2D*). Repare a utilização do recurso de anotações do Java em que temos a palavra *@Override* escrita acima do método sobreposto.

Nas linhas 15 a 18, temos o método principal, que, basicamente, constrói um triângulo e chama o método *calcPerimetro()*. A pergunta que devemos fazer aqui é: qual método será executado: o método *calcPerimetro* em Geom2D ou *calcPerimetro* em *Triangulo*? Bem, a resposta para essa pergunta é o método da classe *Triangulo*. A sobrecarga ficará mais clara quando explicarmos com mais detalhes a herança.

o

Ver anotações

EXEMPLIFICANDO

Você deve ter observado no Código 2.5, na linha 12, a utilização das palavras reservadas *this* e *super*; mas o que elas fazem? A palavra *this* é utilizada para fazer referência a elementos da própria classe em que estamos. Por exemplo: os atributos ladoA, ladoB e ladoC foram declarados na classe *Triangulo*, assim, ao nos referirmos a eles dentro de *Triangulo*, podemos utilizar o *this*, como em *this.ladoA*. A referência utilizando-se o *this* é opcional; em muitos casos, ele ajuda a tornar mais legível o software. Por sua vez, a palavra *super* é utilizada para se fazer referência a elementos da superclasse. Por exemplo: o atributo *perimetro* não foi declarado em *Triangulo*, mas sim em *Geom2D*, logo, para nos referirmos ao perímetro, podemos utilizar o *super*, como em *super.perimetro*. A referência utilizando-se o *super* é opcional; em muitos casos, ele ajuda a tornar mais legível o código. Às vezes, o *this* é utilizado para desambiguir a variável a qual estamos nos referindo. Considere o seguinte exemplo:

```
public class Ponto2D {
    double x;
    double y;
    public Ponto2D(double x, double y) {
        this.y = x; //retira a ambiguidade de x = x;
        this.x = y; //retira a ambiguidade de y = y;
    }
}
```

Fonte: elaborado pelo autor.

No exemplo acima, o construtor Ponto2D recebe dois parâmetros x e y.

Queremos atribuir os valores desses argumentos aos atributos x e y da classe, dessa forma, precisamos utilizar o *this* para evitarmos ambiguidade, senão, teremos uma linha de código da seguinte forma x=x, não deixando claro a qual x nos referimos: se ao x do argumento ou ao x da classe.

Vamos, agora, fazer um breve comparativo entre a sobrecarga e a sobreposição de métodos. O Quadro 2.1 sintetiza algumas características presentes na sobrecarga e na sobreposição.

Quadro 2.1 | Comparativo entre sobrecarga e sobreposição de métodos

Característica	Sobrecarga	Sobreposição
Argumentos	Devem ser trocados	Não devem ser trocados
Tipo de Retorno	Pode ser trocado	Não pode ser trocado
Tipo de Acesso	Pode ser trocado	Pode ser trocado por um modificador menos restritivo
Tipo de Exceção	Pode ser trocado	Pode ser trocado por uma exceção menos restritiva
Classe	Ocorre em uma classe	Ocorre entre duas classes

Característica	Sobrecarga	Sobreposição
Herança	Não envolve herança	Envolve herança
Invocação	Ocorre em tempo de compilação	Ocorre em tempo de execução

Fonte: elaborado pelo autor.

0

Ver anotações

ATENÇÃO

Você deve ter reparado que os códigos aqui desenvolvidos estão em português, mas, mesmo assim, evitamos a utilização de acentos e cê-cedilha. Por exemplo: escrevemos *Matematica* e *Triangulo* sem acentuação. A linguagem Java utiliza a codificação unicode, sendo assim, ela dá suporte a nomes de classes, métodos e variáveis com acentos e cê-cedilha, porém, evitamos o uso desses símbolos pois a utilização desses códigos em diferentes arquiteturas de computador e sistemas operacionais pode gerar problemas.

Considere os seguintes exemplos abaixo.

```
int numeroDeInterações = 1000; //os acentos são aceitos em Java  
int numeroDeInteracoes = 1000; //prefira assim, pois evita erros  
double π = 3.141592; //letras gregas são aceitas em Java  
double pi = 3.141592; //prefira assim, pois evita erros
```

O ideal é utilizar os símbolos presentes na tabela ASCII, a não ser que se tenha a certeza de que o código nunca será executado em uma arquitetura diferente, como, por exemplo, Raspberry Pi, BeagleBone Pi ou Intel Edison.

Figura 2.2 | Síntese sobre atributos

Para visualizar o objeto, acesse seu material digital.

Fonte: elaborada pelo autor.

PESQUISE MAIS

Foram apresentadas algumas regras para nomeação de variáveis, constantes, métodos, classe, objetos e pacotes. A utilização de bons nomes padronizados auxilia diretamente na legibilidade e no entendimento dos programas. Pesquise mais sobre convenção de nomes nos seguintes materiais:

GEEKSFORGEEKS. **Java Naming Conventions.**

HOWTODOINJAVA. **Java Naming Conventions.**

ORACLE. **9 - Naming conventions.**

Caro estudante, nesta seção, você estudou os conteúdos relacionados a atributos e métodos estáticos, a construtores, à sobrecarga e à sobreposição de métodos, bem como analisou diversos exemplos sobre como esses conceitos são implementados na linguagem Java. Nas seções seguintes, estudaremos melhor como funcionam as estruturas de controle e os operadores relacionais e lógicos, avançando ainda mais no entendimento da linguagem Java.

REFERÊNCIAS

ARANTES, J. da S. **Github**. 2020. Disponível em: <https://bit.ly/3eiUMcF>. Acesso em: 20 mai. 2020.

CAELUM. **Java e orientação a objetos**: curso FJ-11. [S.I.]: Caleum, [s.d.]. Disponível em: <https://bit.ly/3ijX34d>. Acesso em: 17 mai. 2020.

CURSO de Java #01 - História do Java - Gustavo Guanabara. [S.I.: s.n.], 2015. 1 vídeo (36 min). Publicado pelo canal Curso em Vídeo. Disponível em: <https://bit.ly/2YvIJDZ>. Acesso em: 22 jul. 2020.

DEITEL, P. J.; DEITEL, H. M. **Java**: como programar. 10. ed. São Paulo: Pearson Education, 2016.

ECLIPSE FOUNDATION. **The 2020 Jakarta Developer Survey Results are Now Available**. 2020. Disponível em: <https://bit.ly/2EknQVi>. Acesso em: 20 jul. 2020.

GEEKSFORGEEKS. **Java Naming Conventions**. [s.d.]. Disponível em: <https://bit.ly/2El7pls>. Acesso em: 20 jul. 2020.

HOWTODOINJAVA. **Java Naming Conventions**. 14 jun. 2020. Disponível em: <https://bit.ly/2ExL9uw>. Acesso em: 20 jul. 2020.

INTELLIJ IDEA. 2020. Disponível em: <https://bit.ly/3gwC0A0>. Acesso em: 20 jul. 2020.

NETBEANS. [s.d.]. Disponível em: <https://bit.ly/2FTW56z>. Acesso em: 20 jul. 2020.

ORACLE. **9 - Naming Conventions**. [s.d.]. Disponível em: <https://bit.ly/2Qj3wWH>. Acesso em: 20 jul. 2020.

ORACLE. Class math. [s.d.] Disponível em: <https://bit.ly/32iL725>. Acesso em: 21 jul. 2020.

SIERRA, K.; BATES, B. **Use a cabeça! Java**. 2. ed. Rio de Janeiro: Alta Books, 2005.

FOCO NO MERCADO DE TRABALHO

CONSTRUTORES E SOBRECARGA

Jesimar da Silva Arantes

0

Ver anotações

MELHORANDO A MODELAGEM DO ROBÔ E INCORPORANDO DE NOVOS RECURSOS

Criação de construtores para classe robô, definição de atributos em constantes, sobrecarga de método e criação de uma nova classe para a função *main*.



Fonte: Shutterstock.

Deseja ouvir este material?

Áudio disponível no material digital.

SEM MEDO DE ERRAR

A *startup* em que você trabalha lhe passou a tarefa de melhorar a modelagem da classe robô incorporando a ela novos recursos. Assim, o seu chefe, após revisar o seu código, listou as seguintes tarefas:

- Criar quatro novos construtores para a classe robô.

- Definir alguns atributos como constantes, de forma a prevenirem eventuais erros.
- Fazer a sobrecarga do método *move*, dando maior flexibilidade na movimentação.
- Retirar o ponto de entrada da aplicação (método *main*) de dentro da classe robô.
- Instalar algum IDE no seu computador de forma a melhorar o seu desenvolvimento.

Após olhar com calma todas as sugestões de seu chefe, você percebeu que a primeira coisa que devia fazer era pesquisar qual o IDE que melhor lhe atenderia. Para isso, você decidiu instalar os principais (Eclipse, Netbeans e IntelliJ IDEA), e após alguns testes, escolheu um.

Em seguida, você decidiu revisar o Código 1.4 para iniciar as modificações solicitadas. Inicialmente, você analisou com calma todos os atributos que seu robô tinha, como nome, peso, velocidadeMax, pesoCargaMax, tipoTracao, posicaoX e posicaoY, e percebeu que os atributos nome, peso, velocidade máxima, peso da carga máxima e tipo de tração não se alteravam para um mesmo robô, dessa forma, decidiu declará-los como constantes, utilizando a palavra reservada *final*.

Posteriormente, você analisou a classe Robô e viu que ela tinha apenas o construtor padrão (implícito), então, criou quatro construtores com as seguintes assinaturas: *public Robo()*, *public Robo(String nome)*, *public Robo(String nome, float peso)* e *public Robo(String nome, float peso, float posX, float posY)*.

Após isso, você se deparou com o problema de sobrestrar o método *move*. Após pensar bastante, percebeu que esse problema parecia bem difícil no contexto, logo, criou apenas a seguinte assinatura: *public void move(float pos)*, em que o robô se moveria apenas para frente ou para trás, conforme o valor passado como argumento.

Por fim, você decidiu retirar da classe Robo a função *main*. Para isso, criou uma nova classe chamada *App.java*, em que colocou a função *main*. Após esses passos, o seu programa ficou pronto, com todas as alterações do seu chefe, e parecido com o Código 2.7.

Código 2.7 | Nova modelagem das classes App e Robo

```
1  public class App {  
2      public static void main(String[] args) {  
3          Robo objRobo = new Robo();  
4          objRobo.move(60, 50);  
5          objRobo.move(55);  
6      }  
7  }  
8  public class Robo {  
9      float posicaoX;  
10     float posicaoY;  
11     final String nome;  
12     final float peso;  
13     final float velocidadeMax = 5;  
14     final float pesoCargaMax = 20;  
15     final String tipoTracao = "esteira";  
16     ...  
17     public Robo(String nome, float peso){  
18         this.nome = nome;  
19         this.peso = peso;  
20         this.posicaoX = 50;  
21         this.posicaoY = 50;  
22     }  
23     public Robo(String nome, float peso, float posX, float posY){  
24         this.nome = nome;  
25         this.peso = peso;  
26         this.posicaoX = posX;  
27         this.posicaoY = posY;  
28     }  
29     public void move(float pos){  
30         this.posicaoY = pos;  
31     }  
32     public void move(float posX, float posY){  
33         this.posicaoX = posX;  
34         this.posicaoY = posY;  
35     }  
36 }
```

0

Ver anotações

ATENÇÃO

Diversas partes foram omitidas, e o código completo pode ser acessado no GitHub do autor.

0

[Ver anotações](#)

ESTRUTURAS DE DECISÃO, CONTROLE E REPETIÇÃO

Jesimar da Silva Arantes

0

[Ver anotações](#)

ESTRUTURAS DE CONTROLE, TOMADAS DE DECISÃO E LAÇOS DE REPETIÇÃO

Utilização de estruturas básicas, mas que são de extrema importância, pois rompem o fluxo linear do código tornando-o mais dinâmico.



Fonte: Shutterstock.

Deseja ouvir este material?

Áudio disponível no material digital.

PRATICAR PARA APRENDER

Caro estudante, bem-vindo à segunda seção da segunda unidade de estudos sobre Linguagem Orientada a Objetos.

Comecemos com uma questão: qual desenvolvedor nunca utilizou, em seus códigos, estruturas de decisão, de repetição, operadores aritméticos, relacionais e lógicos? A maioria dos desenvolvedores, se não todos, já utilizou, mas qual a melhor estrutura de tomada de decisão a ser utilizada? Qual a melhor estrutura de repetição a ser utilizada? Nesta seção, você terá a oportunidade de aprender a

sintaxe básica por trás dos comandos if, if-else, operador ternário, switch, for, while e do-while, bem como conhecerá todos os tipos primitivos de dados presentes no Java.

Como forma de contextualizar sua aprendizagem, lembre-se de que você é contratado de uma *startup* e que seu chefe quer apresentar um simulador completo de robô em Java capaz de transportar caixas em uma sala. Ele está muito satisfeito com o que você está desenvolvendo, mas sabe que ainda há muito trabalho a se fazer. Assim, ele lhe pediu a criação de uma classe que modele a sala em que o robô irá operar e uma classe que modele a entidade caixa. Percebendo que o robô que você modelou não possui nenhum atributo que defina a sua orientação, solicitou, também, um atributo em que a orientação do robô seja especificada e que a manipulação desse atributo utilize alguma estrutura de decisão. A fim de testar a aplicação, ele lhe pediu um robô que se deslocasse em forma de um quadrado, utilizando laços de repetição, mas percebeu que o projeto do simulador do robô estava apenas na sua máquina e lhe pediu para que começasse a utilizar o GitHub, mantendo o código sempre seguro e versionado.

0

Ver anotações

Diante do desafio que lhe foi apresentado, como você irá criar essa sala que representa o mundo do seu robô? Como você irá criar as caixas que comporão o cenário? Como você irá criar esse atributo orientação? Qual estrutura de decisão você irá utilizar para definir a orientação. Como você irá trabalhar com essas diversas classes e como você irá conectá-las? Esta seção irá ajudá-lo a ter um maior domínio das tarefas requisitadas pelo seu chefe, e agora que você foi apresentado à sua nova situação-problema, estude esta seção e compreenda algumas das estruturas básicas da linguagem Java. Essas estruturas, apesar de básicas, são de extrema importância para a construção de qualquer aplicação de pequeno e grande porte.

E aí, vamos juntos compreender esses conceitos e resolver esse desafio?

Bom estudo!

CONCEITO-CHAVE

A linguagem Java, assim como a maioria das linguagens de programação, possui estruturas de controle, como tomadas de decisão e laços de repetição. Outro aspecto inerente à maioria das linguagens de programação diz respeito aos operadores aritméticos, relacionais e lógicos. A linguagem Java é fortemente

tipada, ou seja, os tipos das variáveis são importantes para a manipulação de dados e devem ser declarados no início de criação da variável. Mas calma, todos esses aspectos mencionados serão abordados com mais detalhes nesta seção.

Antes de darmos continuidade, gostaríamos de ressaltar dois pontos importantes:

- A maioria dos exemplos apresentados nesta seção traz apenas fragmentos de código. Dessa maneira, é importante que o aluno implemente o restante da aplicação, por exemplo, criando classe, método *main* e até assinatura do método para o fragmento de código aqui disponível.
- Realize testes de mesa de forma manual para compreender as ideias propostas, bem como testes de mesa de forma automática, utilizando-se o site Java Tutor.

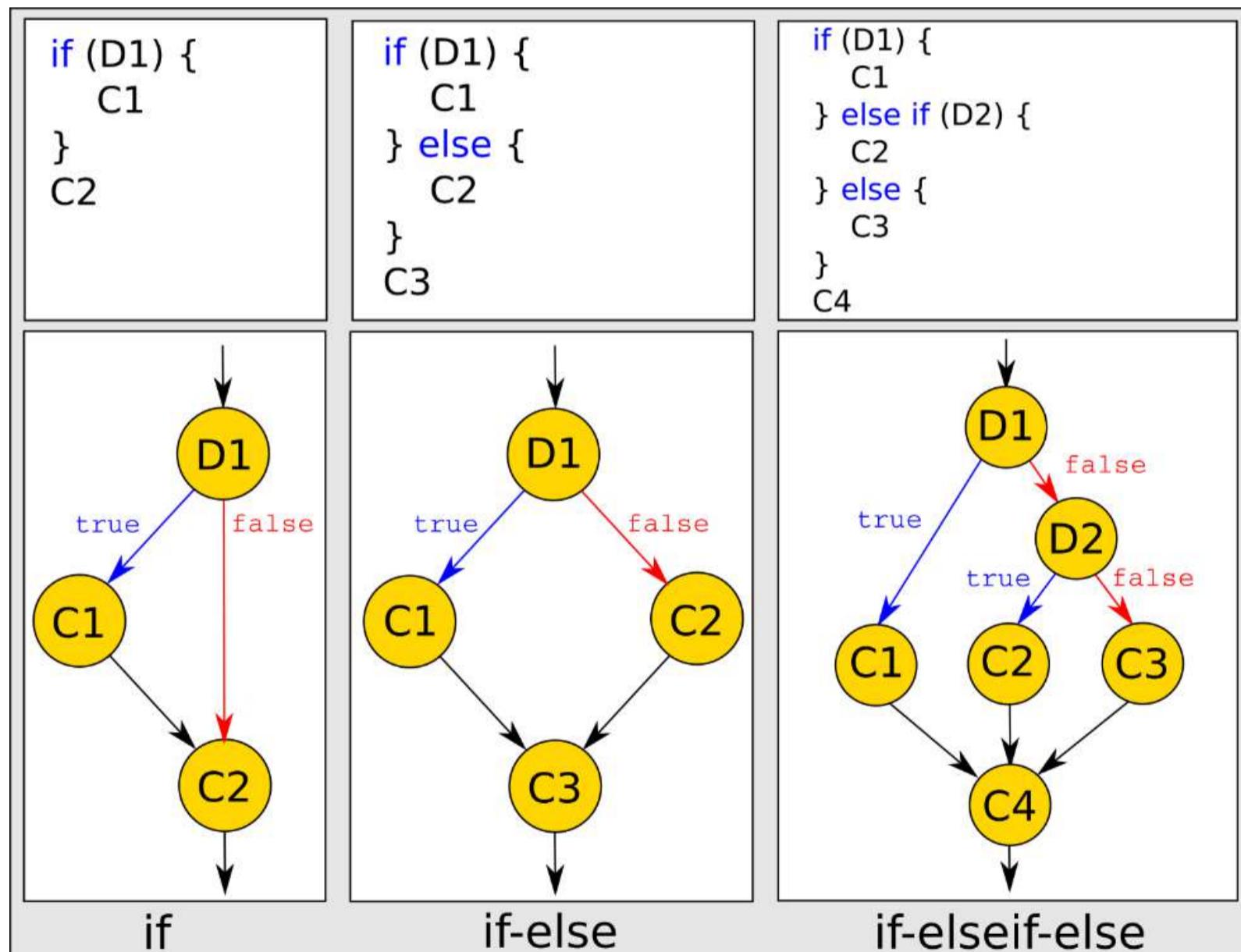
ESTRUTURAS DE DECISÃO

Antes de explicarmos a estrutura de decisão, é bom lembrarmos que a linguagem Java, assim como a maioria das linguagens de programação, executa o seu código de forma sequencial e de cima para baixo, linha a linha. Às vezes, é necessário romper com essa execução sequencial, e queremos executar um trecho do código somente se alguma coisa ocorrer. Essas tomadas de decisão rompem com o fluxo linear do código, criando bifurcações que ajudam a tornar o código mais dinâmico.

COMANDO IF E IF-ELSE

A principal estrutura de decisão é o comando *ife* *if-else*. Esses comandos também podem ser concatenados sucessivamente por outros *ifs*, como em *if-elseif-else*, etc. A Figura 2.3 mostra, de forma gráfica, como ocorre o fluxo de execução em cada um desses comandos mencionados.

Figura 2.3 | Diferentes fluxos de execução para os comandos de decisão



Fonte: elaborada pelo autor.

Na Figura 2.3, podemos perceber que esses três comandos rompem com o fluxo sequencial do código se uma determinada condição (D1 ou D2) for satisfeita.

Assim, se o valor de D1 for verdadeiro, o comando C1 será executado; se o valor de

D1 for falso, um outro comando será executado. Ainda, na Figura 2.3, C1, C2, C3 e C4 indicam comandos a serem executados, já D1 e D2 indicam tomadas de decisão a serem avaliadas.

As estruturas dos comandos de decisão em nível de código podem ser sintetizadas conforme o Quadro 2.2 a seguir.

0

[Ver anotações](#)

Quadro 2.2 | Síntese dos comandos de decisão *if*, *if-else* e *if-elseif-else*

Comando: <i>if</i>	Comando: <i>if-else</i>	Comando: <i>if-elseif-else</i>
<pre>if (ExpLógicaA) { SeqDeComandosA; }</pre>	<pre>if (ExpLógicaA) { SeqDeComandosA; } else { SeqDeComandosB; }</pre>	<pre>if (ExpLógicaA) { SeqDeComandosA; } else if (ExpLógicaB){ SeqDeComandosB; } else { SeqDeComandosC; }</pre>

0

Ver anotações

Fonte: elaborado pelo autor.

Podemos perceber que a sintaxe é a mesma da linguagem de programação C. Vamos analisar por partes esse comando. Inicialmente, na primeira coluna do Quadro 2.2, temos a palavra reservada *if*, logo, em seguida, devemos colocar uma expressão que possua um valor lógico (verdadeiro ou falso) dentro de parênteses. Após isso, a abertura do bloco de código utilizando-se as chaves é necessária se a quantidade de comandos a seguir for maior que um. No Quadro 2.2, a sequência SeqDeComandosA só será executada se a expressão ExpLógicaA for verdadeira. Por sua vez, na segunda coluna, o comando SeqDeComandosB só será executado se a expressão ExpLógicaA for falsa. Ainda no Quadro 2.2, na terceira coluna, temos um exemplo de como é feita a concatenação de dois comandos *if*.

OPERADOR TERNÁRIO

Uma forma mais simplificada de se construir comandos do tipo *if-else* dá-se por meio do operador ternário em Java. O nome operador ternário deve-se ao fato de que o comando é quebrado em três partes separadas pelos símbolos de interrogação (?) e de dois pontos (:). A utilização desse operador é igual ao

comando da linguagem C. Considere o seguinte trecho de código em Java no Quadro 2.3, em que foi utilizado, na esquerda, *if-else* e, na direita, o operador ternário.

0

[Ver anotações](#)

Quadro 2.3 | Comparação entre *if-else* e operador ternário em Java

Comando: if-else	Comando: operador-ternário
<pre>if (ExpLógicaA) { ComandoA; } else { ComandoB; }</pre>	<pre>ExpLógicaA ? ComandoA : ComandoB;</pre>

Fonte: elaborado pelo autor.

No Quadro 2.3, os dois códigos são equivalentes. No operador ternário, inicialmente, temos uma expressão lógica que será avaliada (ExpLógicaA), caso ela seja verdadeira, o ComandoA será executado, caso ela seja falsa, o ComandoB será executado. Uma forma muito simples e útil de utilização do operador ternário se dá para o cálculo do máximo e mínimo de dois valores. Analise o Quadro 2.4 a seguir.

Quadro 2.4 | Exemplo de aplicação do operador ternário em Java

exemplo máximo	exemplo mínimo
<pre>double max = x > y ? x : y; System.out.println("max: " + max);</pre>	<pre>double min = x < y ? x : y; System.out.println("min: " + min);</pre>

Fonte: elaborado pelo autor.

No exemplo acima, a variável max irá armazenar o valor máximo entre as variáveis x e y (considere que x e y foram declarados anteriormente), pois, caso o x seja maior que y, então, x será retornado, caso o x não seja maior que y, então, y será retornado. O exemplo ao lado calcula o mínimo e seu funcionamento é semelhante. Implemente o código completo para calcular o máximo e o mínimo e analise o resultado.

EXEMPLIFICANDO

O operador ternário pode também ser combinado com ele mesmo, criando expressões ainda mais complexas, como nos trechos de códigos mostrados a seguir:

O operador ternário pode também ser combinado com ele mesmo, criando expressões ainda mais complexas, como nos trechos de códigos mostrados a seguir:

0

[Ver anotações](#)

```
String compXY = x>y ? "maior" : x<y ? "menor" :"igual";
String diaSem = dia==0 ? "dom" : dia==1 ? "seg" :
                dia==2 ? "ter" : dia==3 ? "qua" :
                dia==4 ? "qui" : dia==5 ? "sex" :"sab";
```

Fonte: elaborado pelo autor.

0

Ver anotações

Acima, no primeiro exemplo, temos a comparação entre duas variáveis: x e y. Se x for maior que y, retornará “maior”, se x for menor que y, retornará “menor”, se forem iguais, retornarão “iguais”.

No segundo exemplo mostrado acima, temos a comparação entre a variável dia com os números inteiros de 0 a 5. Caso o dia seja 0, então, retornará o valor “dom”, indicando domingo; caso o dia seja 1, então, retornará o valor "seg", representando segunda-feira; e esse raciocínio se repetirá até a sexta-feira, e caso o dia não seja nenhum dos anteriores, então, o valor "sab", indicando o sábado, será retornado.

Caro aluno, acesse o conteúdo no Java Tutor Comparação XY e no Java Tutor DiaDaSemana, presentes no tópico Referências, e execute o código acompanhando a sua execução, linha a linha, para entendê-lo melhor.

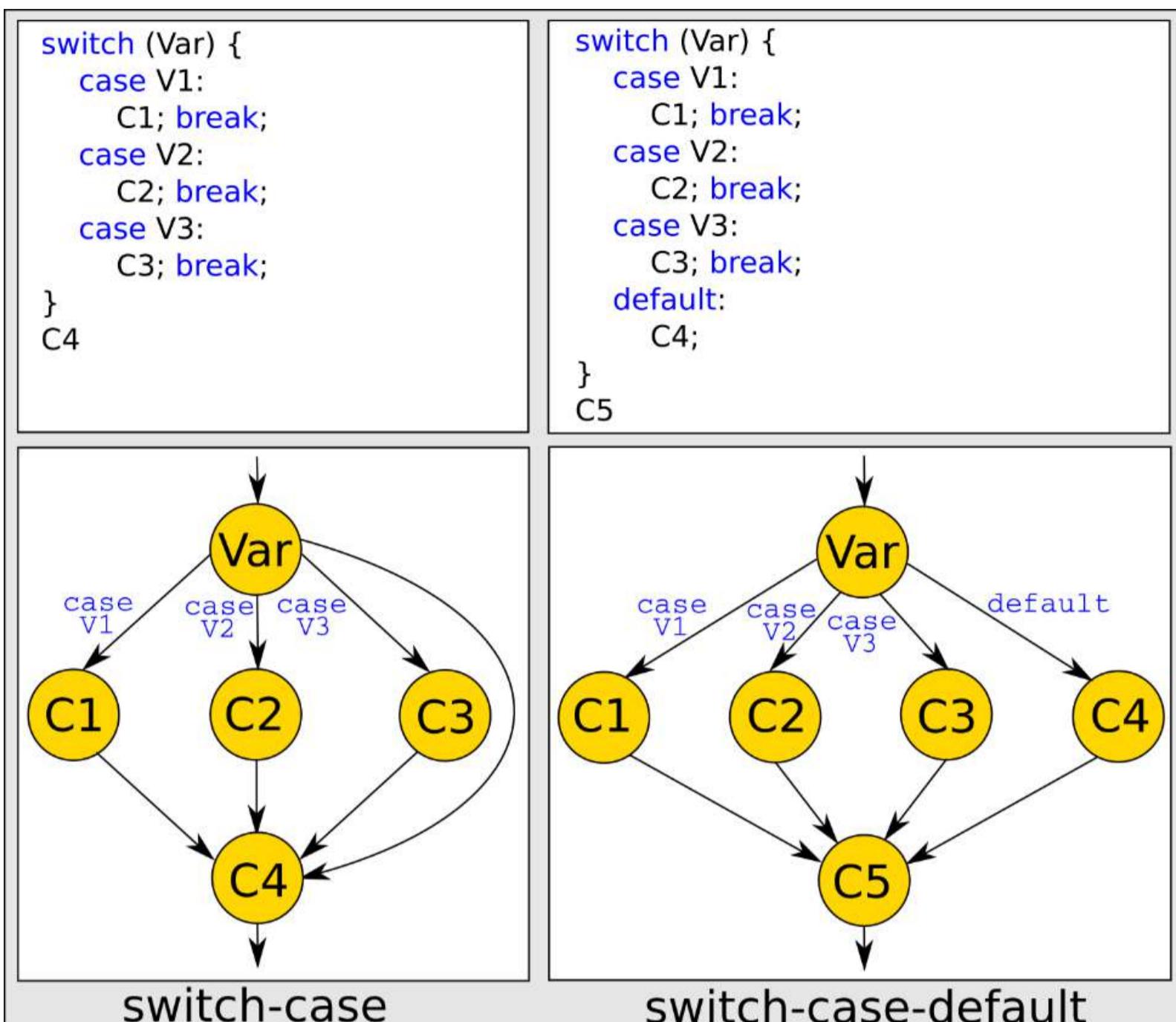
COMANDO SWITCH

Outra estrutura de decisão é o comando switch. Esse comando provê o que chamamos de estrutura de seleção múltipla baseada em alguma variável de controle. A Figura 2.4 mostra, de forma gráfica, como ocorre o fluxo de execução no comando *switch-case* e *switch-case-default*.

Figura 2.4 | Diferentes fluxos de execução para o comando de decisão switch

0

Ver anotações



Fonte: elaborada pelo autor.

A Figura 2.4 mostra que esses dois comandos rompem com o fluxo sequencial do código se uma determinada variável (Var) satisfizer algum dos casos base. Assim, se Var satisfizer o valor V1, então, o comando C1 será executado, se Var satisfizer o valor de V2, então, o comando C2 será executado, e assim sucessivamente. Nessa figura, C1, C2, C3, C4 e C5 indicam comandos a serem executados. No exemplo *switch-case* existe um fluxo unindo diretamente Var a C4, que será executado somente se nenhum dos casos for satisfeito. Já no exemplo *switch-case-default*, existe ao menos um comando dentro do *switch* que deverá ser executado; nesse exemplo, o comando C4 será executado se não for atendido nenhum dos casos anteriores, sendo assim, um comando padrão (*default*).

Você deve ter reparado certa semelhança entre a combinação de vários *if-else* encadeados e o *switch-case*. De forma geral, quando se tem três ou mais *if-else* seguidos e existe a possibilidade de se fazer a troca para um *switch-case*, essa troca é aconselhável.

A grande maioria dos programadores acha mais elegante e legível um código com switch-case do que com múltiplos *if-else*, além disso, o *switch-case* possui um desempenho ligeiramente melhor.

As estruturas básicas dos códigos dos comandos de decisão do tipo switch estão sintetizadas conforme o Quadro 2.5 a seguir.

Quadro 2.5 | Comparação entre switch-case e switch-case-default presentes no Java

Comando: switch-case	Comando: switch-case-default
<pre>switch (VariávelDeControle) { case Valor1: Comando1; break; case Valor2: Comando2; break; ... case ValorN: ComandoN; break; }</pre>	<pre>switch (VariávelDeControle) { case Valor1: Comando1; break; case Valor2: Comando2; break; ... case ValorN: ComandoN; break; default: ComandoParaCasoPadrão; }</pre>

Fonte: elaborado pelo autor.

As variáveis de controle utilizadas dentro do *switch* podem ser dos seguintes tipos: *byte*, *short*, *int*, *char* e *String*. Nesse ponto, gostaríamos de desafiar o aluno a pensar e criar exemplos de aplicações que utilizam a estrutura de seleção múltipla com esses diversos tipos de dados mencionados.

REFLITA

Dentro do comando *switch*, no Quadro 2.5, foram utilizados diversos comandos *break*. O comando *break*, basicamente, interrompe o fluxo de execução dentro do comando *switch*. Dessa maneira, reflita sobre o que

ocorre ao se esquecer de colocar o comando *break* em algum dos *cases* ou em todos os *cases*. Dica: faça a implementação do comando *switch* e não coloque o *break*, em seguida, reflita sobre o que ocorrerá.

OPERADORES

OPERADORES ARITMÉTICOS

Outro assunto importante a ser tratado diz respeito aos operadores aritméticos. Os principais operadores aritméticos presentes na linguagem Java são: soma (+), subtração (-), multiplicação (*), divisão (/) e resto da divisão (%). O funcionamento desses operadores é igual na linguagem C, e todos eles necessitam de dois operandos do tipo numérico e retornam um dado do tipo numérico. O Quadro 2.6 sintetiza algumas informações sobre os operadores aritméticos.

Quadro 2.6 | Síntese dos operadores aritméticos em Java

Nome Operador	Símbolo	Exemplo em Java	Exemplo Numérico
Soma	+	$x + y$	$5 + 3 (= 8)$
Subtração	-	$x - y$	$5 - 3 (= 2)$
Multiplicação	*	$x * y$	$5 * 3 (= 15)$
Divisão	/	x / y	$5 / 3 (= 1)$
Resto da divisão	%	$x \% y$	$5 \% 3 (= 2)$

Fonte: elaborado pelo autor.

Em relação ao Quadro 2.6, duas considerações são importantes: a primeira é sobre o operador da divisão: repare que no exemplo numérico, $5 / 3$ deu resultado 1 e não 1.666..., mas por que isso ocorreu? A divisão de dois números inteiros é um valor inteiro, caso os valores numéricos fossem $5.0 / 3$ ou $5 / 3.0$ ou, ainda, $5.0 / 3.0$, aí sim a resposta seria o valor 1.666..., pois o numerador ou o denominador é número em ponto flutuante. Outra consideração importante refere-se ao operador resto da divisão. Nesse operador, tanto o numerador quanto o denominador e o resultado devem ser números inteiros.

As regras de precedência de operadores em Java funcionam da seguinte forma:

- A multiplicação, a divisão e o resto são avaliados primeiro e possuem o mesmo nível de precedência.
- A soma e a subtração são avaliadas em seguida e possuem o mesmo nível de precedência.
- Caso exista mais de um operador de mesmo nível, então, as expressões serão agrupadas da esquerda para a direita.
- Os parênteses podem ser empregados para se alterar a ordem de precedência dos operadores aritméticos.

■ OPERADORES RELACIONAIS

Agora, quando aos operadores relacionais, os principais presentes na linguagem Java são: igualdade (`==`), diferente (`!=`), maior que (`>`), maior ou igual a (`>=`), menor que (`<`) e menor ou igual a (`<=`). O funcionamento desses operadores é igual na linguagem C. Todos eles necessitam de dois operandos de qualquer tipo e retornam um dado do tipo lógico (booleano). O Quadro 2.7 sintetiza algumas informações sobre os operadores relacionais.

Quadro 2.7 | Síntese dos operadores relacionais em Java

Nome Operador	Símbolo	Exemplo em Java	Exemplo Numérico
Igualdade	<code>==</code>	<code>x == y</code>	<code>5 == 3</code> (falso)
Diferente	<code>!=</code>	<code>x != y</code>	<code>5 != 3</code> (verdade)
Maior que	<code>></code>	<code>x > y</code>	<code>5 > 3</code> (verdade)
Maior ou igual a	<code>>=</code>	<code>x >= y</code>	<code>5 >= 3</code> (verdade)
Menor que	<code><</code>	<code>x < y</code>	<code>5 < 3</code> (falso)
Menor ou igual a	<code><=</code>	<code>x <= y</code>	<code>5 <= 3</code> (falso)

Fonte: elaborado pelo autor.

As regras de precedência de operadores relacionais em Java são da seguinte forma:

- Os operadores (`>`, `>=`, `<`, `<=`) são avaliados primeiro e possuem o mesmo nível de precedência.
- Os operadores (`==`, `!=`) são avaliados em seguida e possuem o mesmo nível de precedência.
- Caso exista mais de um operador de mesmo nível, então, as expressões serão agrupadas da esquerda para a direita.
- Os parênteses podem ser empregados para se alterar a ordem de precedência dos operadores relacionais.

■ OPERADORES DE ATRIBUIÇÃO

Assim como a linguagem C, a linguagem Java também possui operadores de atribuição compostos, e os principais operadores de atribuição são: `+=`, `-=`, `*=`, `/=`, `%=`. O Quadro 2.8 sintetiza algumas informações sobre os operadores de atribuição compostos.

Quadro 2.8 | Síntese dos operadores de atribuição compostos em Java

0

Nome do Operador	Símbolo	Exemplo Encurtado	Significado Exemplo
Atribuição de adição	$+=$	$x += 3$	$x = x + 3$
Atribuição de subtração	$-=$	$x -= 3$	$x = x - 3$
Atribuição de multiplicação	$*=$	$x *= 3$	$x = x * 3$
Atribuição de divisão	$/=$	$x /= 3$	$x = x / 3$
Atribuição de resto	$\%=$	$x \%= 3$	$x = x \% 3$

Ver anotações

Fonte: elaborado pelo autor.

OPERADOR DE INCREMENTO E DECREMENTO

Outro operador muito importante em Java é o operador de incremento e decremento que está sintetizado no Quadro 2.9. Esse é um operador unário, pois tem somente um operando.

Quadro 2.9 | Síntese dos operadores de incremento e decremento em Java

Nome Operador	Símbolo	Exemplo em Java	Explicação do Exemplo
Pré-incremento	$++$	$++x$	Incrementa x em 1 e, então, utiliza x na expressão atual.
Pós-incremento	$++$	$x++$	Utiliza x na expressão atual e, então, incrementa x em 1.
Pré-decremento	$--$	$--x$	Decrementa x em 1 e, então, utiliza x na expressão atual.
Pós-decremento	$--$	$x--$	Utiliza x na expressão atual e, então, decrementa x em 1.

Fonte: elaborado pelo autor.

OPERADORES LÓGICOS

Os principais operadores lógicos presentes em Java são: E lógico (`&&`), Ou Lógico (`||`), negação lógica (`!`). Os operadores E lógico e Ou lógico são operadores binários, já o operador negação lógico é um operador binário.

O Quadro 2.10 sintetiza os operadores lógicos presentes em Java.

0

[Ver anotações](#)

Quadro 2.10 | Síntese dos operadores lógicos em Java

Nome Operador	Símbolo	Exemplo em Java	Exemplo com Valores
E Lógico	&&	exp1 && exp2	true && false (falso)
Ou Lógico		exp1 exp2	true false (verdade)
Negação Lógico	!	!exp	!true (falso)

Fonte: elaborado pelo autor.

Os operadores lógicos em Java são avaliados conforme as tabelas verdade da lógica matemática. Java faz uma avaliação dos valores lógicos chamada **curto-circuito**, em que a avaliação é interrompida assim que o valor global da expressão puder ser inferido. Por exemplo, caso o operador seja o E lógico e a primeira expressão seja falsa, então, o valor falso será retornado, pois, falso E, qualquer coisa é falsa; caso o operador seja o Ou lógico e a primeira expressão seja verdadeira, então, o valor verdadeiro será retornado, pois, verdadeiro Ou, qualquer coisa é verdadeira.

As regras de precedência de operadores lógicos em Java são da seguinte forma:

- O operador de negação lógica (!) é avaliado primeiro. A associatividade é feita da direita para a esquerda.
- O operador E lógico (&&) é avaliado em seguida. A associatividade é feita da esquerda para a direita.
- O operador Ou lógico (||) é avaliado por último. A associatividade é feita da esquerda para a direita.
- Os parênteses podem ser empregados para se alterar a ordem de precedência dos operadores lógicos.

TIPOS DE DADOS

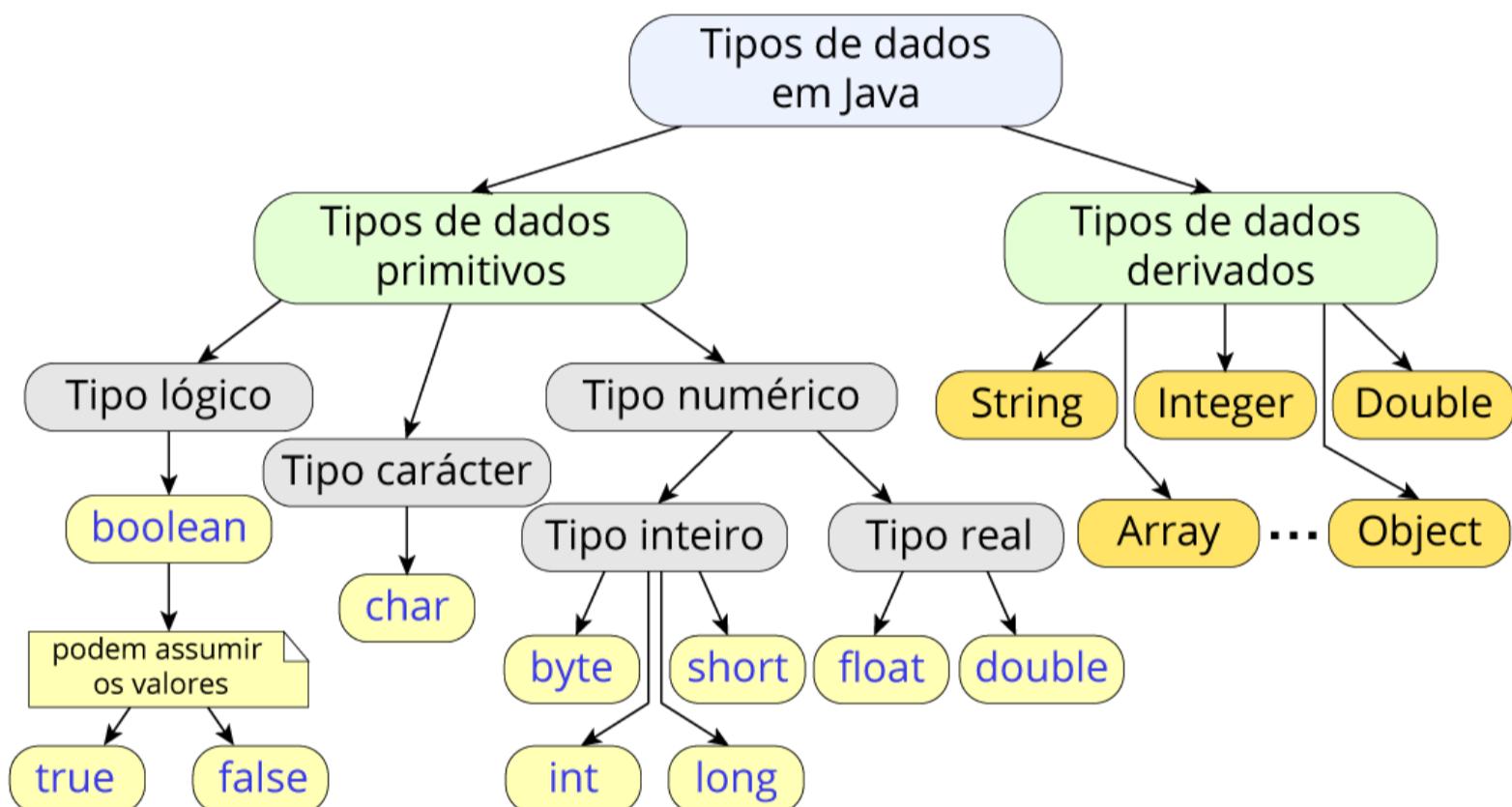
Outro ponto importante a ser abordado: os tipos de dados suportados. Java suporta dados do tipo **primitivo** e **derivado**.

Os tipos primitivos podem ser do tipo lógico, tipo carácter ou tipo numérico, que pode ser separado em dois grupos: o tipo inteiro e o tipo real (ou ponto flutuante). Eles podem ser classificados em: *boolean*, *char*, *byte*, *short*, *int*, *long*, *float* e *double*.

Os tipos de dados derivados podem ser de qualquer classe, como, por exemplo, *String*, *Integer*, *Double*, *Array* e *Object*.

A Figura 2.5 mostra a organização dos tipos de dados em Java.

Figura 2.5 | Organização dos tipos de dados em Java



Fonte: elaborada pelo autor.

ATENÇÃO

É importante lembrar que, em Java, os tipos primitivos são passados para métodos por meio da passagem por valor ou cópia; já os tipos derivados são passados por meio da passagem por referência. Em Java, não se costuma falar em ponteiros, tal como na linguagem C, no entanto, vale lembrar que todas as variáveis do tipo derivado são, na verdade, referências (ponteiros) para uma instância da classe.

O Quadro 2.11 apresenta uma síntese com algumas informações dos tipos primitivos em Java. Neste quadro, foram destacados a quantidade de bits e bytes utilizada na representação dos tipos primitivos e os intervalos de valores dos

dados.

Quadro 2.11 | Síntese dos tipos primitivos em Java

Nome do Tipo	Tamanho em bits	Tamanho em bytes	Intervalo de Valores
boolean	1	-	true ou false
char	16	2	0 até 65535 ou '\u0000' até '\uffff'
byte	8	1	-128 até 127 ou -2^7 até 2^7-1
short	16	2	-32.768 até 32.767 ou -2^{15} até $2^{15}-1$
int	32	4	-2.147.483.648 até 2.147.483.647 ou -2^{31} até $2^{31}-1$
long	64	8	-2^{63} até $2^{63}-1$
float	32	4	1.4e-45 até 3.4e+38
double	64	8	4.9e-324 até 1.8e+308

Ver anotações

Fonte: elaborado pelo autor.

DICA

Ter o hábito de consultar as classes da biblioteca padrão do Java é uma boa prática de programação. Essa consulta ajuda você a ter domínio de quais métodos construtores e constantes estão disponíveis para utilização e qual a sua assinatura. Acima, falamos dos tipos primitivos de dados presentes em Java, e você sabia que Java possui um tipo derivado equivalente a cada um dos tipos primitivos de dados? Esses tipos derivados são: Boolean, Character, Byte, Short, Integer, Long, Float e Double. Tendo isso em mente que tal dar uma olhada nessas classes?

Observação: uma forma de consultá-las é acessando cada uma diretamente pela IDE. Não é necessário ler toda a classe, o importante é ter uma visão geral e consultá-la sempre que necessário.

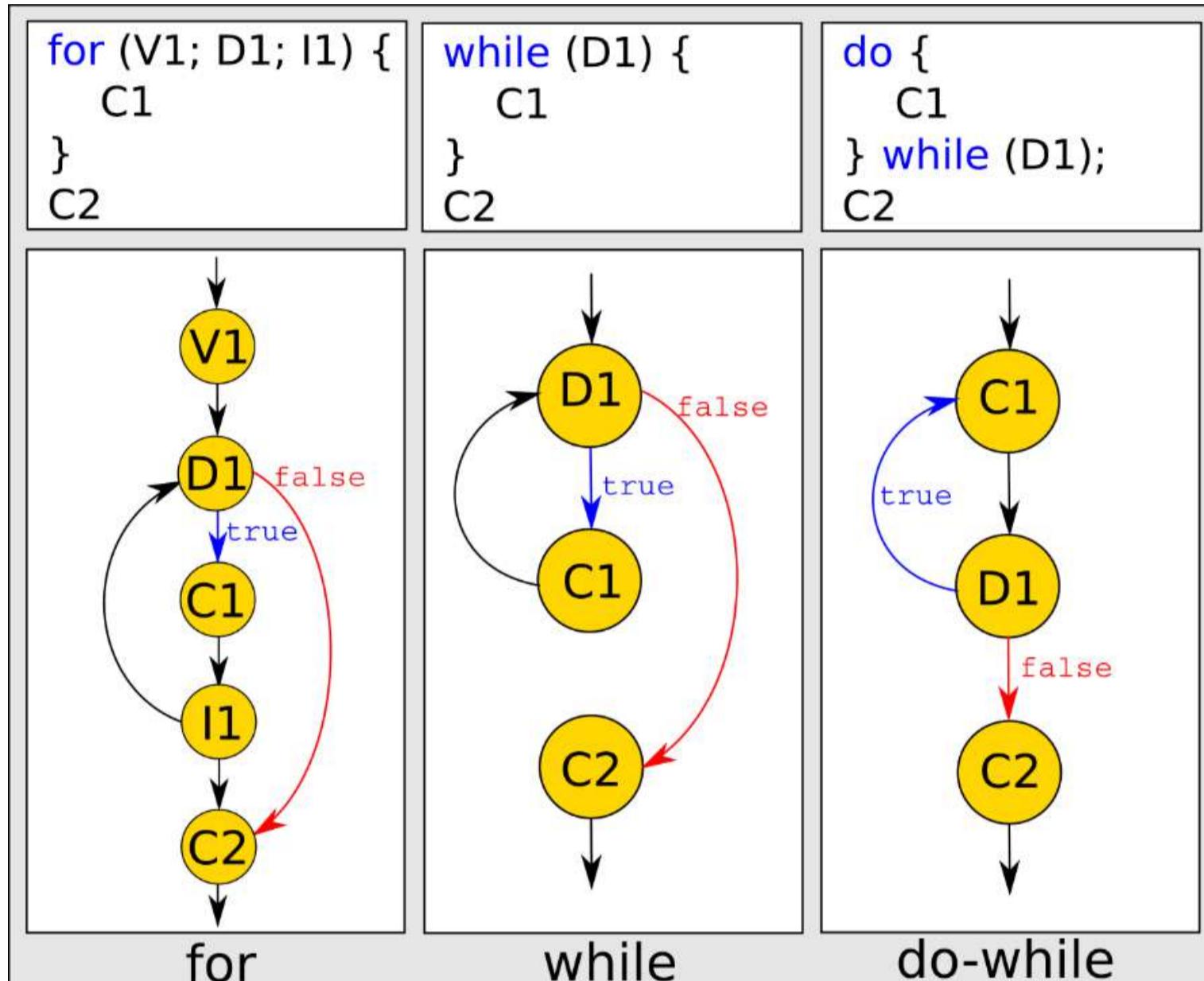
ESTRUTURAS DE REPETIÇÃO

Assim como as estruturas de decisão, um outro assunto de extrema importância são as estruturas de repetição. A linguagem Java possui três estruturas de repetição principais, que são: *for*, *while* e *do-while*. A Figura 2.6 mostra de forma gráfica como ocorre o fluxo de execução em cada um desses comandos.

o

Ver anotações

Figura 2.6 | Diferentes fluxos de execução para os comandos de repetição



Fonte: elaborada pelo autor.

Na Figura 2.6, podemos perceber que esses três comandos criam laços que se repetem enquanto uma determinada condição (D1) é satisfeita, e assim que essa condição se torna falsa, o laço de repetição é interrompido. Ainda na Figura 2.6, V1 indica a inicialização de uma variável, C1 e C2 indicam comandos a serem executados e I1 indica o incremento de uma variável.

COMANDOS *FOR*, *WHILE* E *DO-WHILE*

As estruturas dos comandos de repetição em nível de código podem ser sintetizadas conforme o Quadro 2.12 a seguir.

Quadro 2.12 | Síntese dos comandos de repetição *for*, *while* e *do-while*

Comando: for	Comando: while	Comando: do-while
<pre>for (Inicializ; ExpLóg; Inc){ SeqDeComandosA; }</pre>	<pre>while (ExpLógicaIni){ SeqDeComandosA; }</pre>	<pre>do { SeqDeComandosA; } while (ExpLógicaFim);</pre>

Fonte: elaborado pelo autor.

O comando *for* possui quatro partes principais, que são: inicialização de alguma variável de controle (indicada por *Inicializ*), avaliação de alguma expressão lógica (indicada por *ExpLóg*), incremento de alguma variável (indicado por *Inc*) e, por fim, a sequência de comandos que gostaríamos de executar (indicada por *SeqDeComandosA*). O comando *while* possui duas partes principais, que são: avaliação de uma expressão lógica no início e, em seguida, a sequência de comandos a serem executados. O comando *do-while* possui duas partes principais, que são: a sequência de comandos que serão executados e, somente no final, a avaliação da expressão lógica. Assim, a principal diferença do comando *while* para o *do-while* é que, no segundo, temos a garantia de que a sequência de comandos será executada pelo menos uma vez.

ASSIMILE

As estruturas de repetição estudadas acima são chamadas de estruturas iterativas. A linguagem Java suporta também estruturas recursivas que garantem também a repetição de alguma tarefa. A recursão em Java é feita assim como na linguagem C. Lembre-se de que todo código recursivo pode ser transformado em um código iterativo. O Quadro 2.13 apresenta dois códigos para o cálculo do fatorial de um número inteiro n.

Quadro 2.13 | Exemplos de códigos iterativo e recursivo para cálculo do fatorial

Exemplo fatorial iterativo	Exemplo fatorial recursivo
<pre>public static long fat(int n){ long factorial = 1; for (int i=1; i <= n; i++) { factorial *= i; } return factorial; }</pre>	<pre>public static long fat(int n){ if (n == 0 n == 1) { return 1; } return n * fat(n - 1); }</pre>

Fonte: elaborado pelo autor.

Os códigos acima foram declarados como *static*, pois o fatorial não depende da classe em si. O resultado é um número inteiro do tipo *long* para se garantir o retorno do fatorial com 64 bits de precisão. Com esse tipo de dado, os valores de fatoriais de 0 a 20 serão calculados corretamente, pois cabem dentro da representação *long*. Para valores de n maiores que 20, ocorrem *overflow* e, assim, os resultados começam a não corresponder ao fatorial.

Faça a implementação dessas funções e execute diversos testes; avalie também valores de n maiores que 20 para entender mais sobre o *overflow*. Dica: veja o sinal do número retornado.

Caro aluno, acesse o link presente na seção Referências com o tema JavaTutor (Iterativo e Fatorial) e execute o código acompanhando a sua execução, linha a linha, para entendê-lo melhor.

0

Ver anotações

O programador pode querer interromper todo o fluxo de um laço de repetição a qualquer momento ou, em vez disso, desejar interromper parte de um trecho de código dentro de um loop e ir diretamente para a estrutura de decisão; para tanto, a palavra reservada *break* e *continue* deverão ser utilizadas, respectivamente.

Esse comando *break* e *continue* funcionam de forma igual, tanto na linguagem Java quanto na linguagem C.

0

[Ver anotações](#)

Caro estudante, nesta seção, você estudou os conteúdos relacionados às estruturas de decisão e de repetição, aos comandos *if*, *if-else*, ao operador ternário e *switch*, aos operadores aritméticos, relacionais e lógicos e, por fim, aos comandos *for*, *while* e *do-while*. Foi mostrado diversos exemplos sobre esses conceitos e como esses comandos são implementados em Java. Na seção seguinte, estudaremos melhor como funcionam os modificadores de acesso, o encapsulamento, a herança e o polimorfismo, avançando ainda mais no entendimento da linguagem Java.

Ver anotações

REFERÊNCIAS

ARANTES, J. da S. **Github**. 2020. Disponível em: <https://bit.ly/3eiUMcF>. Acesso em: 20 mai. 2020.

CAELUM. **Java e orientação a objetos**: curso FJ-11. [S.I.]: Caleum, [s.d.]. Disponível em: <https://bit.ly/3ijX34d>. Acesso em: 17 mai. 2020.

CURSO de Java #01 - História do Java - Gustavo Guanabara. [S.I.: s.n.], 2015. 1 vídeo (36 min). Publicado pelo canal Curso em Vídeo. Disponível em: <https://bit.ly/2YvIJDZ>. Acesso em: 22 jul. 2020.

DEITEL, P. J.; DEITEL, H. M. **Java: como programar**. 10. ed. São Paulo: Pearson Education, 2016.

GITHUB. 2020. Disponível em: <https://bit.ly/2ZSr1ud>. Acesso em: 21 jul. 2020.

LOIANE GRONER. **Curso de Java 18**: comandos break e continue. 2015. Disponível em: <https://bit.ly/32lp8Yo>. Acesso em: 20 jul. 2020.

ORACLE. **Class Boolean**. [s.d.] Disponível em: <https://bit.ly/31rt76v>. Acesso em: 20 jul. 2020.

ORACLE. **Class Byte**. [s.d.] Disponível em: <https://bit.ly/3jfJDwu>. Acesso em: 20 jul. 2020.

ORACLE. **Class Character**. [s.d.] Disponível em: <https://bit.ly/2EfveIb>. Acesso em: 20 jul. 2020.

ORACLE. **Class Double**. [s.d.]. Disponível em: <https://bit.ly/3jaYV5u>. Acesso em: 20 jul. 2020.

ORACLE. **Class Float**. [s.d.]. Disponível em: <https://bit.ly/3gweZgm>. Acesso em: 20 jul. 2020.

ORACLE. **Class Integer**. [s.d.]. Disponível em: <https://bit.ly/3Ihp4BC>. Acesso em: 20 jul. 2020.

ORACLE. **Class Long**. [s.d.]. Disponível em: <https://bit.ly/34sFJw1>. Acesso em: 20 jul. 2020.

ORACLE. **Class Short**. [s.d.]. Disponível em: <https://bit.ly/2EufpGO>. Acesso em: 20 jul. 2020.

PYTHON TUTOR. **Java tutor - visualize Java code execution to learn Java online**. [s.d.]. Disponível em: <https://bit.ly/32s8S81>. Acesso em: 20 jul. 2020.

SIERRA, K.; BATES, B. **Use a cabeça! Java**. 2. ed. Rio de Janeiro: Alta Books, 2005.

0

[Ver anotações](#)

FOCO NO MERCADO DE TRABALHO

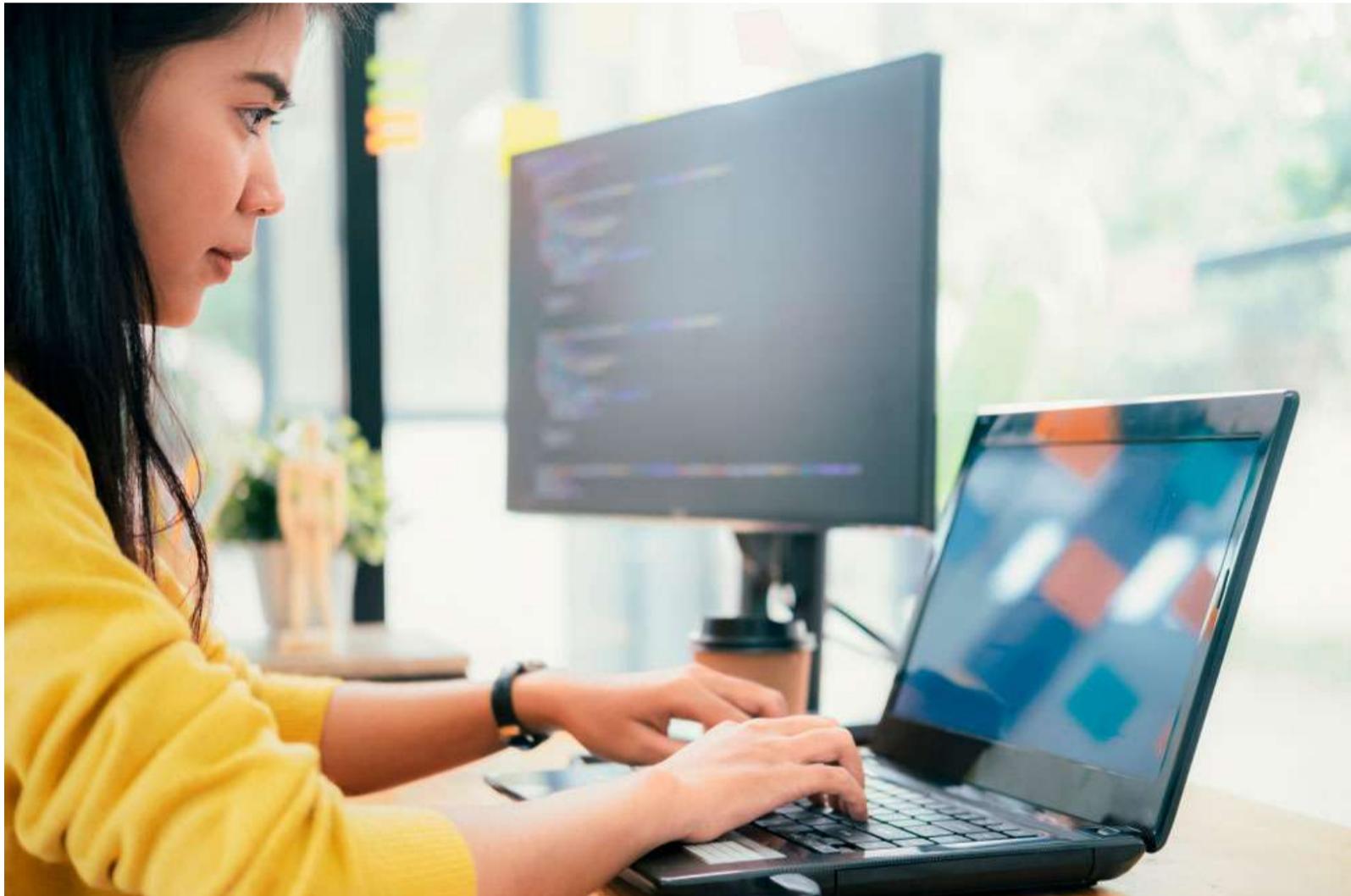
ESTRUTURAS DE DECISÃO, CONTROLE E REPETIÇÃO

Jesimar da Silva Arantes

Ver anotações

MODELANDO A SALA, A CAIXA E A ORIENTAÇÃO DO ROBÔ

Criação de classe para modelar a sala em que o robô vai operar, a entidade sala e o atributo orientação para fazer o robô deslocar.



Fonte: Shutterstock.

Deseja ouvir este material?

Áudio disponível no material digital.

SEM MEDO DE ERRAR

A *startup* em que você trabalha lhe passou algumas tarefas, que foram especificadas assim:

- Criar uma classe que modele a sala em que o robô irá operar.
- Criar uma classe que modele a entidade caixa.
- Definir um atributo que modele a orientação do robô.

- Manipular o atributo orientação utilizando alguma estrutura de decisão.
- Fazer o robô se deslocar em forma de um quadrado utilizando laços de repetição.
- Submeter o projeto no GitHub.

Como forma de iniciar o trabalho, você decidiu revisar o Código 2.7 para adicionar os recursos que seu chefe deseja. Inicialmente, você optou por criar um atributo chamado orientação dentro da classe Robo. Você teve a ideia de declará-lo do tipo inteiro e, inicialmente, quis que o robô tivesse somente quatro orientações: FRENTE, ATRAS, ESQUERDA e DIREITA.

O próximo passo foi modelar a classe caixa. Dessa forma, você pensou nos seguintes atributos: nome do item da caixa, quantidade de itens da caixa, posição no eixo x, posição no eixo y, peso, comprimento, largura e altura. Nesse momento, você decidiu criar essa classe apenas com esses atributos, sem nenhum método.

Em seguida, você optou por criar uma classe que representasse a sala, então, chamou essa classe de Mundo2D. Após isso, pensou sobre os atributos que essa classe deveria possuir e decidiu que ela teria apenas os atributos dimensão x e dimensão y.

Por fim, você modificou o programa principal na classe App e colocou dois laços do tipo *for* aninhados, ou seja, um laço dentro do outro. O laço mais externo controlava a direção do movimento e o laço mais interno controlava quanto de movimento seria executado em cada direção. Por fim, para selecionar a direção seguida pelo robô, você utilizou o comando *switch-case*.

Após esses passos, o seu programa ficou pronto, com todas as alterações do seu chefe e parecido com o Código 2.9. Repare que algumas partes foram omitidas; o código completo pode ser acessado no GitHub do autor.

```
1 public class Robo {  
2     ...  
3     int orientacao;  
4     static final int FRENTE = 0;  
5     static final int ATRAS = 1;  
6     static final int ESQUERDA = 2;  
7     static final int DIREITA = 3;  
8     ...  
9     public void setOrientacao(char tecla){  
10        if (tecla == 'w') {  
11            this.orientacao = FRENTE;  
12        }else if (tecla == 's') {  
13            this.orientacao = ATRAS;  
14        }else if (tecla == 'a') {  
15            this.orientacao = ESQUERDA;  
16        }else if (tecla == 'd') {  
17            this.orientacao = DIREITA;  
18        }  
19    }  
20 }  
21 public class Caixa {  
22     String nomeItem;  
23     int qtdItem;  
24     int posX;  
25     int posY;  
26     float peso;  
27     final float comprimento;  
28     final float largura;  
29     final float altura;  
30     public Caixa(String nomeItem, int qtdItem, int posX, int posY,  
31                     float peso, float comprimento, float largura,  
32                     float altura) {  
33         this.nomeItem = nomeItem;  
34         this.qtdItem = qtdItem;  
35         this.posX = posX;  
36         this.posY = posY;  
37         this.peso = peso;  
38         this.comprimento = comprimento;  
39         this.largura = largura;  
40         this.altura = altura;
```

0

Ver anotações

```
41     }
42 }
43 public class Mundo2D {
44     final int DIM_X;
45     final int DIM_Y;
46     public Mundo2D(int dimX, int dimY) {
47         this.DIM_X = dimX;
48         this.DIM_Y = dimY;
49     }
50 }
51 public class App {
52     public static void main(String[] args) {
53         Robo robo = new Robo();
54         for (int d = 0; d < 4; d++) {
55             for (int j = 1; j <= 10; j++) {
56                 robo.printPos();
57                 switch (d) {
58                     case 0: //move ao longo do eixo x para frente
59                         robo.move(50 + j * 4, 50);
60                         break;
61                     case 1: //move ao longo do eixo y para cima
62                         robo.move(90, 50 + j * 4);
63                         break;
64                     case 2: //move ao longo do eixo x para trás
65                         robo.move(90 - j * 4, 90);
66                         break;
67                     case 3: //move ao longo do eixo y para baixo
68                         robo.move(50, 90 - j * 4);
69                         break;
70                 }
71             }
72         }
73         robo.printPos();
74     }
75 }
```

Fonte: elaborado pelo autor.

Por fim, você decidiu criar um projeto no GitHub com o seguinte nome SimuladorRobo, em que o código-fonte desenvolvido ficará disponível. Para tanto, a fim de submeter o projeto no GitHub, você precisará utilizar o git, que já deve estar instalado em seu computador. Existem duas formas principais de se fazer envio, uma delas se dá por meio de alguma ferramenta gráfica e a outra por meio de linha de comando. Caso você decida fazer isso via linha de comando, você provavelmente precisará dos seguintes comandos na primeira vez: *git clone*, *git init*, *git add*, *git status*, *git commit*, *git remote add origin* e *git push -u origin master*. Após criado o projeto e submetido, as próximas vezes serão mais simples e, em geral, você utilizará os seguintes comandos: *git add*, *git status*, *git commit* e *git push*.

0

[Ver anotações](#)

REUTILIZAÇÃO DE CLASSES

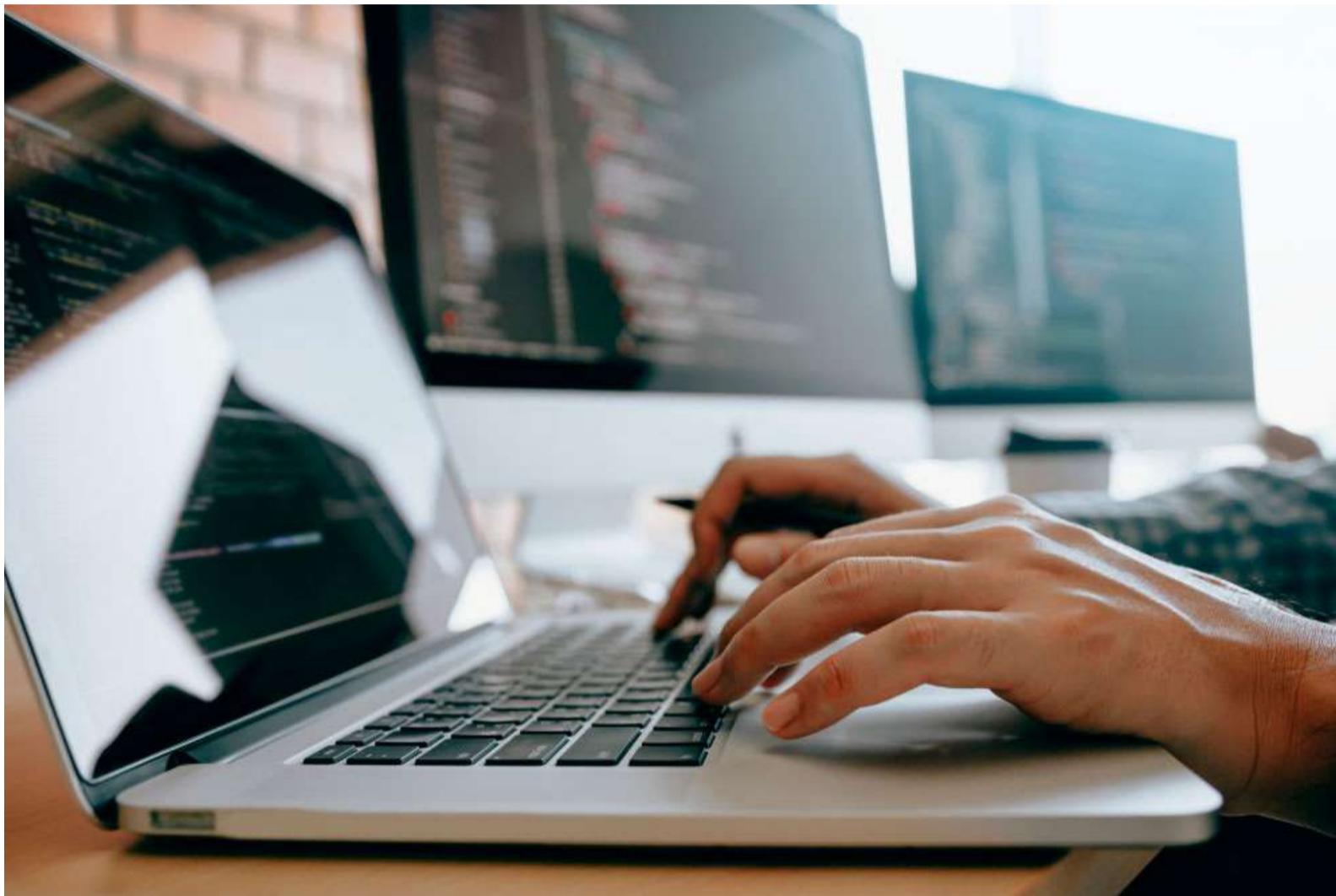
Jesimar da Silva Arantes

0

[Ver anotações](#)

ESTRUTURA ORGANIZACIONAL DE PACOTE

A linguagem Java utiliza pacotes para fazer a organização das classes e recursos.



Fonte: Shutterstock.

Deseja ouvir este material?

Áudio disponível no material digital.

PRATICAR PARA APRENDER

Prezado aluno, bem-vindo a mais uma seção do livro *Linguagem Orientada a Objetos*. Nesta seção, alguns dos conceitos fundamentais da Orientação a Objetos (OO) serão apresentados. Qual programador não gosta de desenvolver códigos reutilizáveis, bem organizados e que possam ser facilmente utilizados por outros programadores? A maioria dos desenvolvedores, se não todos, gosta desses aspectos em seus sistemas, mas, afinal, como construir softwares reutilizáveis, como organizar melhor a aplicação Java e como incorporar bibliotecas desenvolvidas por outros desenvolvedores? Nesta seção, você terá a oportunidade de responder a essas e outras perguntas.

o

Ver anotações

De forma a contextualizar sua aprendizagem, lembre-se de que você trabalha em uma *startup* e que seu chefe quer desenvolver um simulador completo de robô em Java que seja capaz de transportar caixas em uma sala. Para tanto, mais uma vez, o seu chefe se reuniu com você e pediu para que organizasse melhor o seu projeto, utilizando a estrutura de pacotes que o Java fornece. Ele analisou as classes que você implementou e percebeu que, praticamente, nenhum modificador de acesso foi utilizado. Dessa maneira, ele pediu a você que colocasse modificadores de acessos adequados aos atributos e métodos, e ao analisar as classes, percebeu que os atributos estavam sendo acessados diretamente, logo, solicitou que criasse métodos *getters* e *setters* onde fossem necessários, de forma a encapsular melhor os atributos. Ele também analisou as suas classes e percebeu que você não utilizou herança em nenhum lugar, solicitando a você que começasse a utilizar mais esse conceito em sua aplicação, bem como começasse a sobrescrever os métodos *toString* e *equals* e continuasse a utilizar GitHub e a estudar alguns clientes GUI para o GitHub, de forma a melhorar a utilização desse gerenciador de repositório.

Diante do desafio que lhe foi apresentado, como você irá organizar o seu código? O que são pacotes? O que são modificadores de acesso? Como você definirá os modificadores para os seus atributos? O que são métodos *getters* e *setters* e como criá-los? O que é herança e como você irá utilizá-la? O que são clientes GUI para GitHub?

Acalme-se. Esta seção irá ajudá-lo a ter um maior domínio dos conteúdos mencionados.

Agora, uma vez que foi apresentado à sua nova situação-problema, estude esta seção e compreenda os três grandes pilares da orientação a objetos, que são: encasulamento, herança e polimorfismo. Esses conceitos somados aos conceitos de classe, objeto, método e atributo englobam as ideias centrais da OO. E aí, vamos juntos compreender esses conceitos e resolver esse desafio?

Bom estudo!

CONCEITO-CHAVE

As linguagens Orientadas a Objetos (OO), como Java, possuem três importantes pilares, que são: o encapsulamento, a herança e o polimorfismo, e outro conceito presente na linguagem Java diz respeito aos modificadores de acesso. Esse são alguns dos conceitos abordados em mais detalhes nesta seção, porém, antes de

darmos continuidade, gostaríamos de ressaltar um ponto importante. Alguns dos exemplos apresentados nesta seção são apenas fragmentos de código. Dessa maneira, é importante que o aluno implemente o restante da aplicação, por exemplo, criando a classe e o método *main*.

o

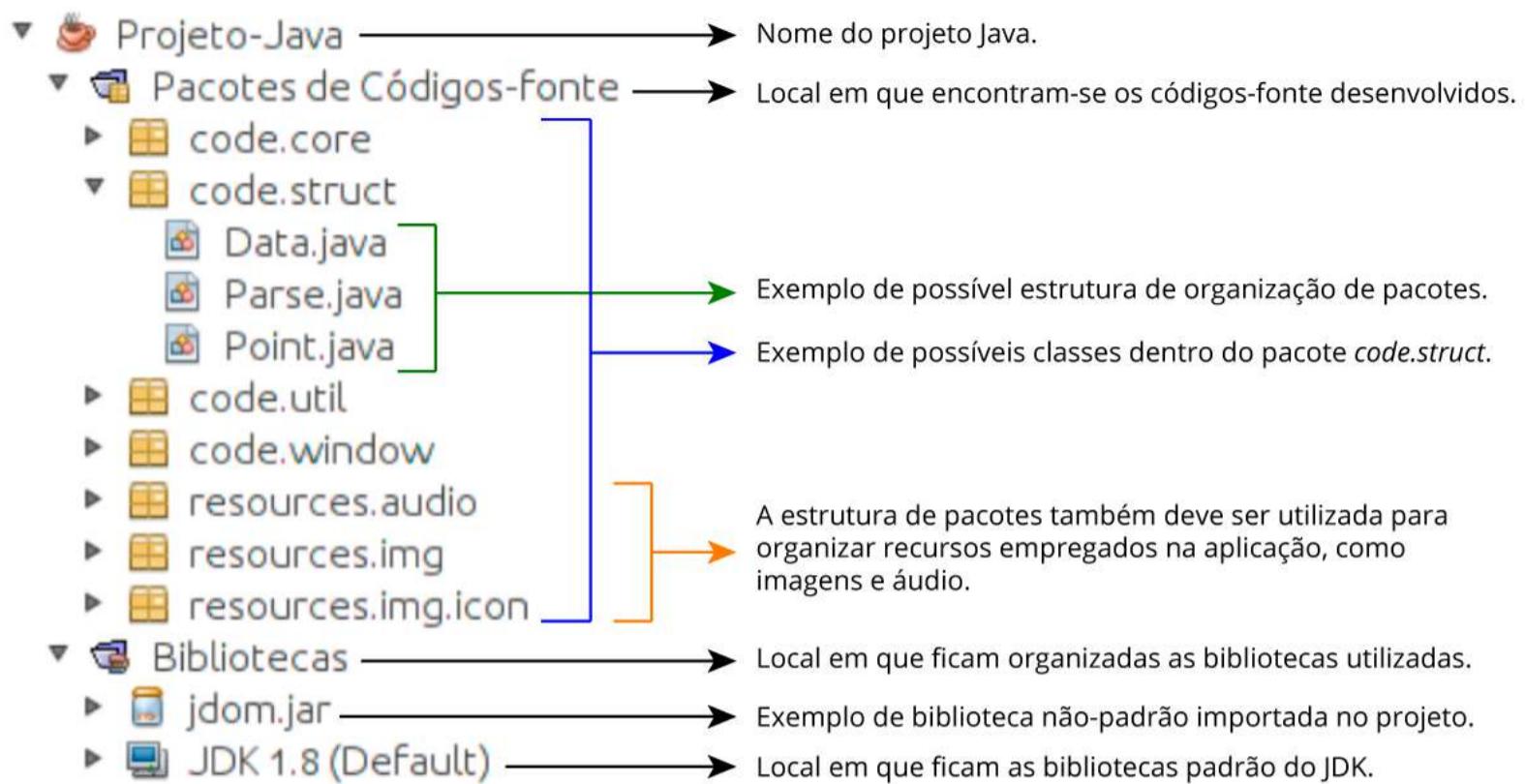
ORGANIZAÇÃO DE PACOTES

Antes de avançarmos nesta seção, você deve ter em mente que a linguagem Java tem a capacidade de desenvolver aplicações que podem crescer bastante. Até este momento, desenvolvemos aplicações com poucas classes, mas imagine que você está desenvolvendo uma aplicação que está crescendo e que conta com 10 classes, logo, começa a ficar incomodado com a organização. No entanto, você ainda não faz nada e ela continua a crescer, agora, ela tem 20 classes. Nesse ponto, você pensa que não dá mais para continuar a desenvolver sem antes organizar as classes em estruturas que simplifiquem o processo.

A fim de entendermos como é desenvolvido um grande projeto em Java, em primeiro lugar, vamos apresentar uma visão geral de como é feita a organização de um projeto qualquer. Sendo assim, a Figura 2.7 mostra a organização da estrutura de um projeto qualquer desenvolvido em Java.

Ver anotações

Figura 2.7 | Organização da estrutura de um projeto qualquer na linguagem Java



Ver anotações

Fonte: elaborada pelo autor.

De acordo com a Figura 2.7, o projeto é subdividido em algumas partes.

Inicialmente, temos o nome do projeto e, em seguida, temos o local em que ficam armazenados os códigos-fonte desenvolvidos pelo usuário. Conforme o código desenvolvido cresce, faz-se necessário criar uma estrutura para organizar as classes. Chamamos essa estrutura organizacional de pacote. Nesse exemplo, temos os seguintes pacotes: *core*, *struct*, *util* e *Windows* dentro do pacote *code*, os pacotes *img* e *audio* dentro do pacote *resources* e, por fim, temos o pacote *icon* dentro do pacote *img*. O ideal é que cada pacote contenha classes relacionadas ao nome do pacote. Nesse exemplo, dentro do pacote *code.struct* temos três classes, que são: Data, Parse e Point. Em seguida, temos o local em que ficam armazenadas as bibliotecas, e nesse local, existem dois tipos de bibliotecas, que são as bibliotecas padrão e as bibliotecas não padrão do Java. No exemplo acima, a biblioteca não padrão, chamada *jdom.jar*, foi importada para utilização e o conjunto de bibliotecas padrão do JDK também foi importado, uma vez que essas bibliotecas padrão são sempre importadas em qualquer projeto Java.

A linguagem Java utiliza pacotes para fazer a organização das classes e recursos.

Uma forma de se pensar em um pacote é pensar em um diretório no sistema de arquivos do Sistema Operacional (SO). O diretório, de forma geral, serve para agrupar um determinado conjunto de arquivos, assim como um pacote em Java

serve para agrupar um determinado conjunto de classes. Dessa maneira, os pacotes são utilizados também para se fazer o encapsulamento de um grupo de classes.

0

[Ver anotações](#)

LEMBRE-SE

O separador utilizado na organização de um pacote é o ponto (.). Conforme dito acima, no sistema de arquivos do computador, um pacote corresponde a um diretório, ou seja, cada vez que você utiliza um ponto para separar um nome de pacote, um novo diretório é criado no computador. Por exemplo, o pacote *code.struct* possui duas pastas, uma chamada *code* e outra *struct* (que fica dentro da pasta *code*). Uma dica para entender melhor o que foi explicado é fazer navegação por meio do sistema de arquivos do seu computador pelas pastas (pacotes) e, enquanto isso, criar pacotes no Java (por meio da IDE) para ver como ficaram organizados os diretórios do projeto.

Ainda com relação à Figura 2.7, vamos considerar a classe Data que está dentro do pacote *code.struct*. Em Java, é necessário informar, no início da declaração da classe, em qual pacote ela se encontra. Para isso, a palavra reservada *package* é utilizada, seguida pelo nome do pacote. Analise o trecho de código abaixo da classe Data.

Código 2.10 | Trecho do código da classe Data

```
1 package code.struct; // definição da localização do pacote da classe
2 public class Data { // declaração da classe Data
3     ...
4 }
```

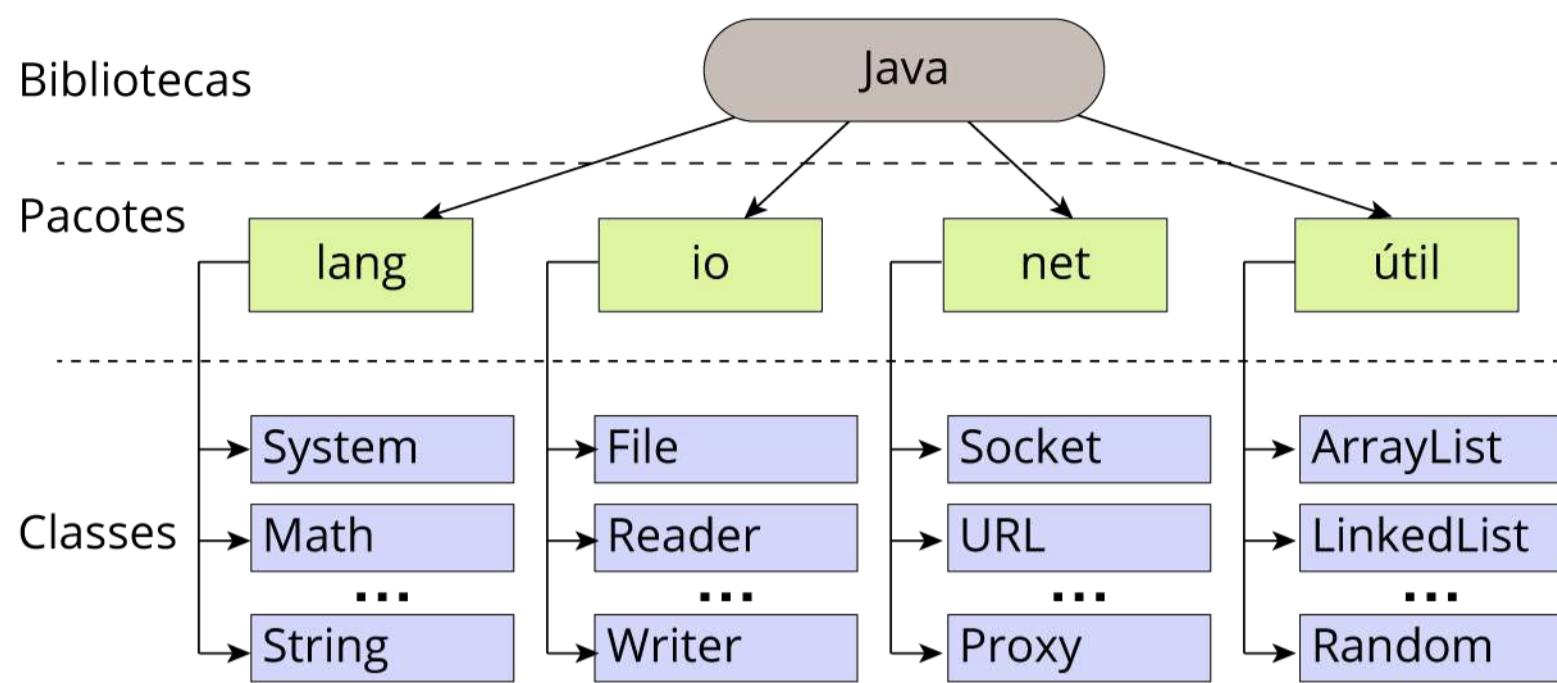
Fonte: elaborado pelo autor.

No texto acima, destacamos como é feita a organização de pacotes definidos pelo usuário durante o desenvolvimento do seu projeto. A linguagem Java é extremamente rica em bibliotecas nativas ou, ainda, *Application Programming Interfaces* (APIs) disponíveis para utilização. Dessa maneira, é importante destacarmos também como estão organizados os pacotes nativos da linguagem Java. O pacote nativo mais importante do Java é o *java.lang*, que agrupa as classes fundamentais para o projeto da linguagem Java, sendo elas: Object, Class, System, Math, String, Double, Float, Integer, etc. (ORACLE, 2020). Alguns outros pacotes importantes do Java são: *java.io*, *java.net* e *java.util*.

Na Figura 2.8 vemos os principais pacotes nativos da linguagem Java.

Figura 2.8 | Organização dos principais pacotes nativos da linguagem Java

0



Ver anotações

Fonte: elaborada pelo autor.

IMPORTAÇÃO DE PACOTES

Na linguagem Java, toda vez que se for utilizar alguma classe que esteja em outro pacote diferente do pacote atual, sua importação deverá ser feita; para isso, será utilizada a palavra reservada *import* seguida pelo nome do pacote e, por fim, pelo nome da classe. O comando *import* do Java possui certa semelhança com o comando *include* da linguagem C. Considere o Quadro 2.14 a seguir com alguns exemplos de importação.

Quadro 2.14 | Exemplos de importações de classes de alguns pacotes nativos do Java

Exemplo de Código	Descrição sobre as Importações
<code>import java.io.File;</code>	Importação da classe File que está dentro pacote <i>java.io</i> .
<code>import java.net.Socket;</code>	Importação da classe Socket do pacote <i>java.net</i> .
<code>import java.util.Random;</code>	Importação da classe Random do pacote <i>java.util</i> .
<code>import java.lang.System;</code>	Importação da classe System do pacote <i>java.lang</i> . OBS: esta importação é a única opcional, pois é feita de forma automática.
<code>import java.io.*;</code>	Importação de todas as classes do pacote <i>java.io</i> .

Fonte: elaborado pelo autor.

Conforme ressaltado no Quadro 2.14, a única importação que não é necessária é a importação do pacote `java.lang`, pois já é feita de forma automática pelo compilador do Java. Nesse quadro, percebemos também que podemos utilizar o caractere curinga asterisco (*) (como em `import java.io.*;`) na importação de todas as classes de um determinado pacote.

o

[Ver anotações](#)

DICA

Você pode ficar pensando que programar em Java é difícil, pois é preciso saber fazer a importação de pacotes (e muitos pacotes possuem caminhos muito longos) e a indicação correta do nome do pacote em que uma determinada classe está inserida. No entanto, você pode ficar tranquilo, pois a maioria dos IDEs para Java disponíveis hoje em dia faz a busca e a indicação da importação necessária de forma automática, baseadas apenas no nome da Classe. Sendo assim, não é necessário saber os comandos *import* e *package* de memória.

0

Ver anotações

Existe um tipo especial de importação que é a **importação estática**. Por meio dessa importação, podemos fazer a utilização direta de métodos e atributos estáticos sem utilizarmos o nome da classe.

EXEMPLIFICANDO

A título de exemplo, vamos considerar o Código 2.11, que utiliza a importação tradicional e a importação estática.

Código 2.11 | Implementação com importação estática e tradicional

```
1 //pacote em que a classe TesteImportacao está.  
2 package code.unidade2.secao3;  
3  
4 //importação tradicional de classes  
5 import java.util.Random;  
6  
7 //importação estática de classes  
8 import static java.lang.System.out;  
9 import static java.lang.Math.*;  
10  
11 public class TesteImportacao {  
12     public static void main(String[] args) {  
13         out.println("Número(PI) = " + PI);  
14         out.println("Número(E) = " + E);  
15         out.println("Raiz(25) = " + sqrt(25));  
16         out.println("Pot(2^10) = " + pow(2, 10));  
17         out.println("Log(100) = " + log10(100));  
18         Random rnd = new Random();  
19         out.println("Rnd = " + rnd.nextDouble());  
20     }  
21 }
```

0

[Ver anotações](#)

Fonte: elaborado pelo autor.

o

Ver anotações

Na linha 2 do código acima, temos a especificação do pacote, em que se encontra a presente classe chamada de TestelImportacao; em seguida, na linha 5, temos a importação da classe Random, presente no pacote *java.util*; na linha 8, temos a importação estática do atributo estático *out*, que está localizado na classe System do pacote *java.lang*; na linha 9, temos a importação estática de todos os métodos e atributos estáticos presentes na classe Math do pacote *java.lang*; nas linhas 13 a 19 temos a utilização dos métodos e atributos estáticos importados — repare que a utilização das constantes PI e E foi feita de forma direta. Repare também que as chamadas aos métodos estáticos *sqrt*, que calcula a raiz quadrada, *pow*, que calcula a potência, e *log10*, que calcula o logaritmo na base 10, foram feitas de forma direta, sem a utilização da classe Math ponto (.) nome do método. Por fim, nas linhas 18 e 19, temos a declaração de um objeto (rnd) da classe Random seguida pela geração de um número aleatório entre 0 e 1, obtido por meio do método *nextDouble*.

É um bom的习惯 de programação estudar as bibliotecas nativas já disponíveis para uso, assim como procurar bibliotecas de terceiros que possam resolver o problema no qual estamos lidando. Vale lembrar que alguns desenvolvedores preferem desenvolver a sua própria solução e, assim, evitar a utilização de bibliotecas de terceiros. Essa segunda alternativa não é ruim, mas pode levar um maior tempo no desenvolvimento. Uma dica de onde consultar e baixar bibliotecas de terceiros é o MVN Repository

EXEMPLIFICANDO

De forma a entender como pesquisar e incorporar uma biblioteca feita por outra pessoa em seu projeto, acesse o site MVN Repository e, em seguida, busque a seguinte biblioteca: *apache commons math*. Então, procure a versão mais recente da biblioteca e baixe o arquivo .jar disponível. Após isso, o arquivo deverá se chamar algo como *commons-math3-3.6.1.jar*, e para que possa incluir/importar esse arquivo .jar, um projeto novo na sua IDE de preferência deverá ser criado (**observação:** os passos para isso não foram mostrados pois, em cada IDE, os passos são ligeiramente diferentes). Após esses passos, desenvolva um código similar ao Código 2.12 abaixo.

Código 2.12 | Implementação que imprime todos os primos de 0 a 100

```
1 import org.apache.commons.math3.primes.Primes;
2 public class TesteBibliotecaTerceiros {
3     public static void main(String[] args) {
4         for (int i = 0; i < 100; i++) {
5             if (Primes.isPrime(i)) {
6                 System.out.println("i = " + i);
7             }
8         }
9     }
10 }
```

0

Ver anotações

Fonte: elaborado pelo autor.

O código acima utiliza uma biblioteca de matemática desenvolvida pela comunidade para o cálculo de todos os números primos de 0 a 100. Esse foi apenas um exemplo de utilização de biblioteca feita por outras pessoas em que há diversas classes que lidam com problemas relacionados à área de matemática.

MODIFICADORES DE ACESSO

Outro assunto muito importante a ser abordado refere-se aos modificadores de acesso. Por meio deles, podemos restringir ou permitir que um atributo, método, construtor e/ou classe seja acessado ou não. Em Java, temos quatro modificadores de acesso, que são: público, privado, protegido e *default*. O Quadro 2.15 sintetiza esses modificadores.

Quadro 2.15 | Síntese dos modificadores de acesso presentes em Java

Modificador	Palavra-Reservada	Descrição da Visibilidade
Público	<i>public</i>	Aplicável à própria classe e a qualquer outra.
Protegido	<i>protected</i>	Aplicável à própria classe, outras classes dentro do próprio pacote e dentro de subclasses em outros pacotes.

Modificador	Palavra-Reservada	Descrição da Visibilidade
<i>Default</i>	-	Aplicável à própria classe e a outras classes dentro do mesmo pacote.
Privado	<i>private</i>	Aplicável à própria classe somente.

Fonte: elaborado pelo autor.

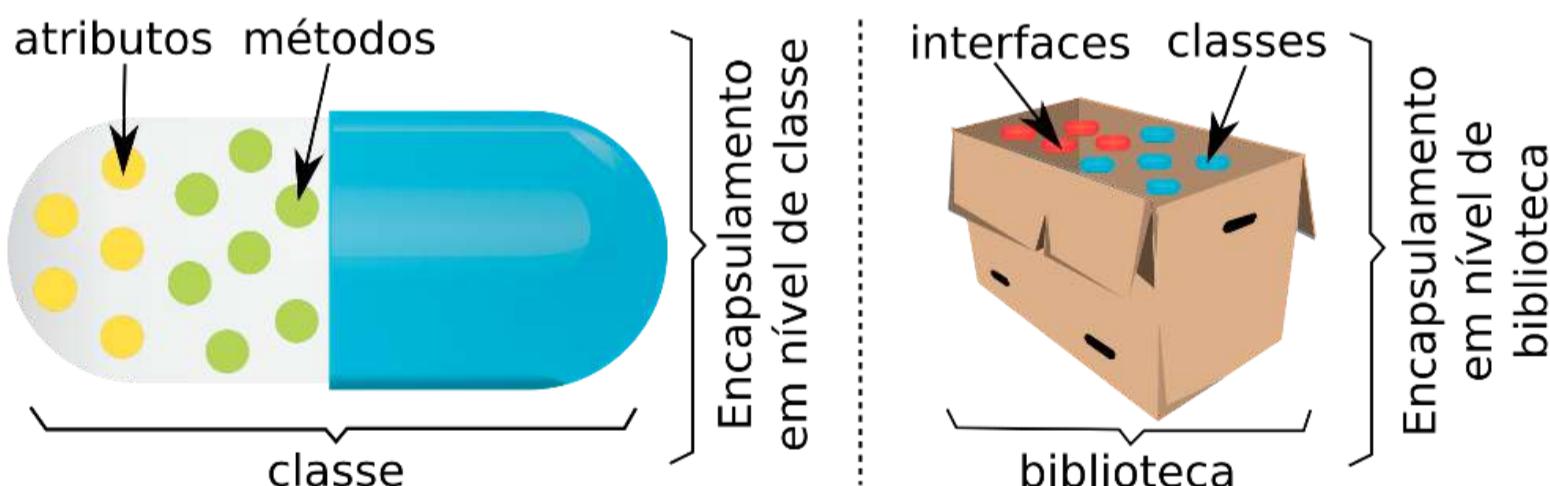
Com base na descrição da visibilidade de cada um dos modificadores, podemos perceber que o modificador privado é o mais restritivo (seguido por *default*) e protegido; por fim, temos o público, que é o menos restritivo.

ENCAPSULAMENTO DE CÓDIGO

Acima, vimos como modificar a visibilidade dos códigos desenvolvidos, a fim de se ocultar/proteger informações que não devem ser acessadas por outras pessoas ou desenvolvedores. Dessa maneira, conseguimos encapsular um código mantendo acessível apenas o que consideramos importante, evitando-se “quebras” ou erros na aplicação. Ao restringir o acesso a algumas informações que visivelmente não são úteis a aplicações externas, simplificamos a utilização das classes desenvolvidas.

A principal ideia do encapsulamento de código é esconder os detalhes de implementação das classes e bibliotecas. De forma geral, assumimos como regra que um objeto nunca deve manipular diretamente os atributos de outro objeto. Em muitos casos, a manipulação direta pode ocasionar erros e gerar muitas inconsistências no programa desenvolvido. Dessa forma, devemos bloquear os atributos utilizando os modificadores de acesso *private*, *default* ou, ainda, *protected*, porém, após o bloqueio dos atributos, como acessá-los e modificá-los? Uma prática comum é a criação de métodos que chamamos de *getters* e *setters* ou, ainda, *gets* e *sets*. Os métodos *gets* servem para pegar alguma informação, já os métodos *sets* servem para definir alguma informação. A Figura 2.9 mostra um esquema didático que representa o encapsulamento de código em nível de classe e em nível de biblioteca/pacote.

Figura 2.9 | Esquema representando o encapsulamento de código



Fonte: elaborada pelo autor.

A manipulação de uma determinada classe, geralmente, é feita por meio de métodos públicos. Quando necessário, o acesso a constantes pode ser feito de forma direta, ou seja, constantes podem ser declaradas como públicas,

principalmente as constantes do tipo *static*. Acima, foi falado sobre métodos do tipo *getters* e *setters*, e para entendermos melhor como declará-los, considere a classe chamada Pessoa, mostrada no Código 2.13.

0

[Ver anotações](#)

Código 2.13 | Classe pessoa com o atributo idade e seus métodos *get* e *set*.

```
1 public class Pessoa {  
2     private int idade;  
3     public int getIdade() {  
4         return idade;  
5     }  
6     public void setIdade(int idade) {  
7         this.idade = idade;  
8     }  
9 }
```

0

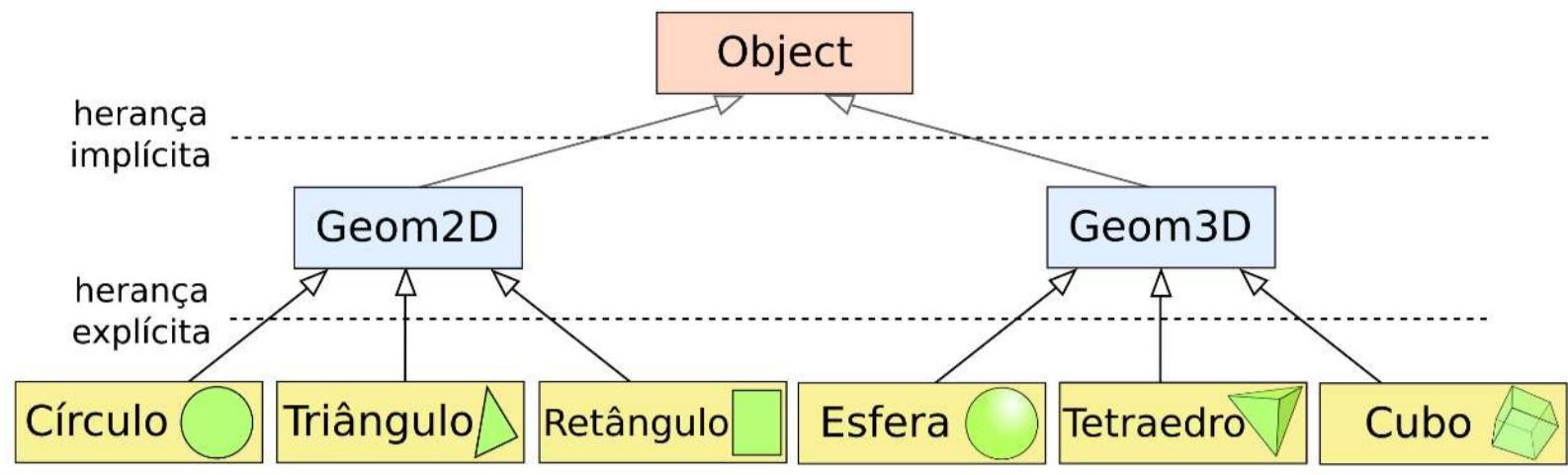
Ver anotações

Fonte: elaborado pelo autor.

Em relação ao Código 2.13, queremos destacar que o atributo *idade* (linha 2) foi declarado como privado, assim, as chamadas para se obter o atributo *idade* é feita utilizando-se o método público *getIdade* (definido nas linhas 3 a 5). Caso seja necessário fazer alguma alteração na *idade*, podemos utilizar o método público *setIdade* (definido nas linhas 6 a 8). Posteriormente, veremos que podemos/devemos tratar alguns possíveis erros na manipulação de dados. Por exemplo, no método *setIdade*, poderíamos impedir a atribuição de uma *idade* negativa, visto que isso não faz sentido na aplicação.

■ OPERAÇÃO DE HERANÇA

Outro assunto de extrema importância em Java é a **herança**. A ideia de herança foi brevemente abordada na seção 1 da unidade 2, quando falamos de sobreposição de métodos. A operação de herança envolve duas classes, em que uma das classes é chamada de **subclasse** e a outra é chamada de **superclasse**. Nessa operação, a subclasse herda todas as características definidas na superclasse. De forma geral, a subclasse personaliza a superclasse ao adicionar novos recursos e aspectos próprios. A linguagem Java suporta apenas herança simples, em que pode haver apenas uma superclasse. Linguagens como C++ e Python suportam herança múltipla, em que pode haver mais de uma superclasse. A fim de que possa compreender melhor o conceito de herança, vamos considerar o seguinte exemplo mostrado na Figura 2.10.



Fonte: elaborada pelo autor.

Na Figura 2.10, as classes Círculo, Triângulo e Retângulo herdam as características da classe Geom2D. Por sua vez, as classes Esfera, Tetraedro e Cubo herdam as características da classe Geom3D. Por fim, as Classes Geom2D e Geom3D herdam, implicitamente (automaticamente), as características da classe Object.

O código 2.14 nos mostra a sintaxe básica usada na implementação de uma subclasse e uma superclasse qualquer.

Código 2.14 | Sintaxe básica para definição de subclasses e superclasses

```
1 class SuperClasse {  
2     ... //código associado a superclasse  
3 }  
4 class SubClasse extends SuperClasse { //operação de herança  
5     ... //código associado a subclasse  
6 }
```

Fonte: elaborado pelo autor.

Em Java, todas as classes são automaticamente descendentes da classe Object. Essa herança é feita de forma implícita e, dessa forma, todos os métodos disponíveis em Object estão disponíveis em todas as outras classes implementadas do Java. Alguns exemplos de métodos disponíveis são: *toString*, *equals*, *clone* e *hashCode*.

O Código 2.15 mostra a implementação do esquema apresentado na Figura 2.10.

Código 2.15 | Implementação das superclasses e subclasses do exemplo da Figura 2.10

```
1  public class Geom2D { //herda implicitamente a classe Object
2      protected double perimetro;
3      protected double area;
4      public double calcPerimetro(){
5          return 0; //não existe perímetro em objeto abstrato
6      }
7      public double calcArea(){
8          return 0; //não existe área em objeto abstrato
9      }
10 }
11
12 public class Geom3D { //herda implicitamente a classe Object
13     protected double area;
14     protected double volume;
15     public double calcArea(){
16         return 0; //não existe área em objeto abstrato
17     }
18     public double calcVolume(){
19         return 0; //não existe volume em objeto abstrato
20     }
21 }
22
23 public class Retangulo extends Geom2D {
24     private final double base;
25     private final double altura;
26     public Retangulo(double lado) {
27         this.base = lado;
28         this.altura = lado;
29     }
30     public Retangulo(double base, double altura) {
31         this.base = base;
32         this.altura = altura;
33     }
34     @Override
35     public double calcPerimetro() {
36         super.perimetro = 2 * this.base + 2 * this.altura;
37         return super.perimetro;
38     }
39     @Override
40     public double calcArea() {
```

0

Ver anotações

```
41         super.area = this.base * this.altura;
42
43     }
44
45     @Override
46
47     public String toString() {
48
49         return String.format("Rect:{\n peri: %.2f\n area: %.2f\n}",
50                             this.calcPerimetro(), this.calcArea());
51
52     }
53
54     @Override
55
56     public boolean equals(Object obj) {
57
58         if (obj instanceof Retangulo) {
59
60             Retangulo r = (Retangulo)obj;
61
62             return (base == r.base) && (altura == r.altura);
63
64         }else{
65
66             return false;
67
68         }
69     }
70
71     public static void main(String[] args) {
72
73         Retangulo rect1 = new Retangulo(4, 25);
74
75         Retangulo rect2 = new Retangulo(4, 26);
76
77         System.out.println(rect1); // invoca método toString
78
79         System.out.println(rect2); // invoca método toString
80
81         if (rect1.equals(rect2)) { // invoca método equals
82
83             System.out.println("Figuras Geométricas Iguais");
84
85         } else {
86
87             System.out.println("Figuras Geométricas Diferentes");
88
89         }
90
91     }
92
93     public class Cubo extends Geom3D {
94
95         private final double lado;
96
97         public Cubo(double lado) {
98
99             this.lado = lado;
100
101        }
102
103        @Override
104
105        public double calcArea() {
106
107            super.area = 6 * this.lado * this.lado;
108
109            return super.area;
110
111        }
112
113        @Override
```

```
82     public double calcVolume() {  
83         super.volume = this.lado * this.lado * this.lado;  
84         return super.volume;  
85     }  
86     @Override  
87     public String toString() {  
88         return String.format("Cubo:{\n area: %.2f\n vol: %.2f\n}",  
89             this.calcArea(), this.calcVolume());  
90     }  
91     @Override  
92     public boolean equals(Object obj) {  
93         if (obj instanceof Cubo){  
94             Cubo cubo = (Cubo)obj;  
95             return this.lado == cubo.lado;  
96         }else{  
97             return false;  
98         }  
99     }  
100    public static void main(String[] args) {  
101        Cubo cubo1 = new Cubo(1);  
102        Cubo cubo2 = new Cubo(1);  
103        System.out.println(cubo1); // invoca método toString  
104        System.out.println(cubo2); // invoca método toString  
105        if (cubo1.equals(cubo2)) { // invoca método equals  
106            System.out.println("Figuras Geométricas Iguais");  
107        } else {  
108            System.out.println("Figuras Geométricas Diferentes");  
109        }  
110    }  
111 }
```

0

Ver anotações

Fonte: elaborado pelo autor.

Acima, nas linhas 1 a 10, temos a definição da classe Geom2D. Inicialmente, temos, nessa classe, dois atributos principais, que são o perímetro e a área. Como sabemos que toda figura geométrica 2D deve saber o seu perímetro, assim, definimos o método calcPerímetro. Da mesma forma, toda figura 2D deve saber a sua área, assim, definimos o método calcArea. Por enquanto, colocamos esses

métodos apenas retornando um valor 0, mas, na próxima seção, mostraremos como alterar essa classe para se tornar uma classe abstrata, e então, teremos apenas a assinatura do método melhorando a legibilidade da implementação.

De forma semelhante à classe Geom2D, temos a classe Geom3D, que está definida nas linhas 12 a 21. Nessa classe, temos os atributos área e volume, que são atributos comuns de figuras geométricas 3D.

Já nas linhas 23 a 68, temos a definição da classe Retangulo. Repare que essa classe herda todas as características de Geom2D por meio da palavra-reservada *extends*. A classe Retangulo possui dois atributos, que são base e altura. Em seguida, foram definidos dois construtores, e os métodos calcPerimetro e calcArea foram sobrescritos/sobrepostos. A ideia de fazer a sobreposição de métodos vem do fato de que somente a classe Retangulo sabe como calcular o seu perímetro e a sua área. O método *toString* (lembre-se: esse método pertence à classe Object) foi sobreescrito nas linhas 44 a 48 e auxilia no momento de se fazer a impressão de informações da classe na tela; em seguida, o método *equals* (lembre-se: esse método também pertence à classe Object) foi sobreescrito nas linhas 49 a 57, a fim de auxiliar na comparação entre dois objetos; de forma geral, desejamos que dois objetos do mesmo tipo e que contêm o mesmo conteúdo sejam considerados iguais. Na linha 51, por sua vez, foi utilizada a palavra reservada *instanceof*, que podemos ler da seguinte forma: "é uma instância de" ou "é um". O comando *instanceof* é um operador que compara o tipo de uma variável a uma classe. No código acima, o tipo da variável *obj* é comparado com a classe Retangulo, então, na linha 52, foi feita uma operação de *casting*, de forma a converter o tipo Object (tipo menos específico) em tipo Retangulo (tipo mais específico). Para fecharmos o método *equals*, repare que foi considerado que dois retângulos são iguais quando possuem mesma base e mesma altura. Por fim, nas linhas 58 a 69, foi criado um método *main* para se testar a classe Retangulo. Aqui, temos dois pontos importantes a serem destacados:

- Nas linhas 61 e 62, temos uma chamada implícita ao método *toString* (esse método é sempre chamado quando se pede para imprimir informações da classe).
- Na linha 63, temos uma chamada ao método *equals*, em que são comparados dois objetos do tipo Retangulo.

REFLITA

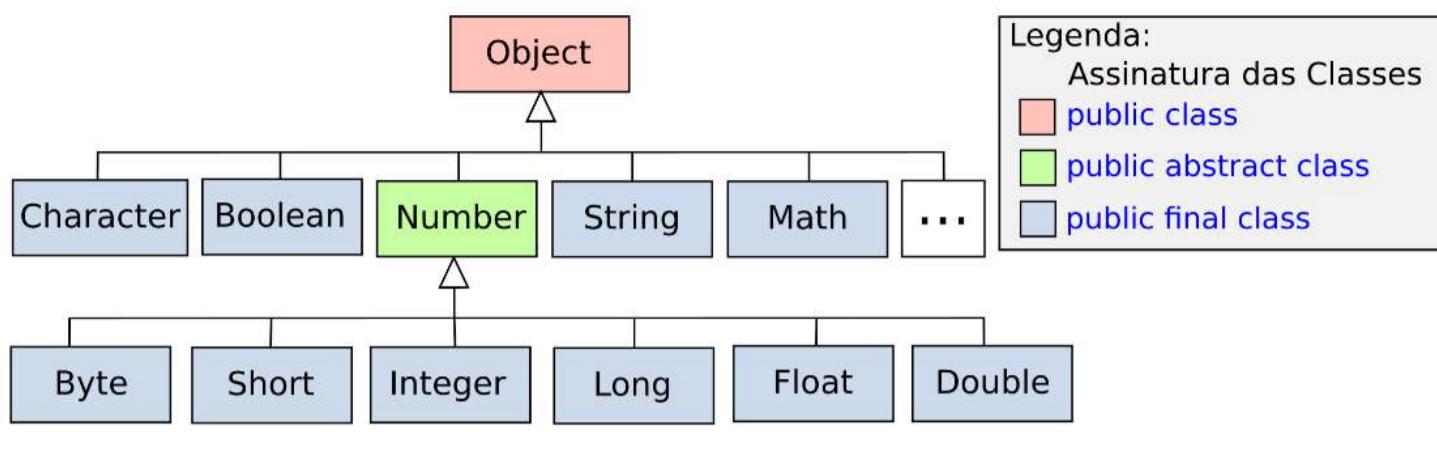
Gostaria de propor a você uma reflexão sobre a importância da sobrescrita dos métodos *toString* e *equals*. Talvez, ainda mais importante, seja descobrir, em nossa aplicação, quando devemos sobrescrever esses métodos. Gostaria de destacar que a implementação do método *equals* pode ser feita de diversas formas e que a escolha de uma forma ou outra é uma questão que deve ser respondida durante o projeto do software. Por exemplo, imagine uma classe que modele a entidade Pessoa: como definiremos que dois objetos do tipo Pessoa são iguais? Um primeiro jeito, não muito bom, pode ser utilizando-se o atributo nome da pessoa, porém todos sabemos que existem pessoas homônimas, dessa maneira, uma outra forma é utilizando-se o atributo CPF, que sabemos ser único.

Ainda com relação ao Código 2.15, nas linhas 71 a 101, temos a definição da classe Cubo, que herda todas as características de Geom3D. A explicação do funcionamento dessa classe é semelhante à classe Retangulo, logo, ela não será explicada. O restante das classes mostradas na Figura 2.10 não será exibido, mas gostaríamos de propor ao aluno que as implemente, pois, assim, terá a oportunidade de fixar o conteúdo.

ASSIMILE

A operação de herança também pode ser estudada e entendida a partir de algumas classes nativas do Java, como mostrado na Figura 2.11. Algumas classes foram declaradas como *final*, de forma a impedir que outras classes tentem herdá-las. Por exemplo, pelo fato de a classe String ser declarada como *final*, não se pode criar uma classe que herde String. Ainda na Figura 2.11, percebemos que a classe Number foi declarada como *abstract*, isso implica a não construção de um objeto do tipo Number, ou seja, ela é uma classe abstrata. Estudaremos mais sobre classes abstratas na unidade seguinte.

Figura 2.11 | Esquema representando a herança em classes nativas do Java



Fonte: elaborada pelo autor.

POLIFORMISMO

Assim como o encapsulamento e a herança, o polimorfismo é um dos pilares da Orientação a Objetos (OO). No polimorfismo, o tipo de variável de referência pode ser de um tipo diferente do tipo do objeto, mas, para isso, é necessário que a variável de referência seja uma superclasse do tipo do objeto. A ideia do polimorfismo está no fato de que uma referência pode assumir diversas formas diferentes. O polimorfismo caracteriza-se também por fazer com que duas ou mais classes tenham métodos com o mesmo nome, de forma que uma função possa utilizar um objeto de qualquer uma das classes polimórficas. Diante do que foi exposto, as seguintes linhas de código são possíveis e demonstram comportamentos polimórficos:

Código 2.16 | Demonstração de comportamentos polimórficos

```
1 //Variável de Referência → Tipo do Objeto
2 Geom2D geom1 = new Circulo(1);
3 Geom2D geom2 = new Triangulo(3, 4, 5);
4 Geom2D geom3 = new Retangulo(4, 10);
```

Fonte: elaborado pelo autor.

Nos exemplos acima, as variáveis de referência são geom1, geom2 e geom3, que são do tipo Geom2D. Por sua vez, os tipos de objetos são, respectivamente, Circulo, Triangulo e Retangulo. Aqui, é importante reparar que Geom2D é uma superclasse dos tipos de objetos, e de forma a tornar mais clara a ideia do polimorfismo, considere o exemplo de Código 2.17 mostrado a seguir.

Código 2.17 | Exemplo de modelagem usando-se polimorfismo

```
1 public class TestePolimorfismo {  
2     public static void main(String[] args) {  
3         Geom2D objCirculo = new Circulo(2.9);  
4         objCirculo.calcPerimetro();  
5         Geom2D objTriangulo = new Triangulo(6);  
6         objTriangulo.calcPerimetro();  
7         Geom2D maxPeri = maxPerimetro(objCirculo, objTriangulo);  
8         System.out.println("Maior Perímetro: " + maxPeri);  
9         if (maxPeri instanceof Circulo){  
10             Circulo max = (Circulo)maxPeri;  
11             System.out.println("Max Circulo: " + max);  
12         } else if (maxPeri instanceof Triangulo){  
13             Triangulo max = (Triangulo)maxPeri;  
14             System.out.println("Max Triangulo: " + max);  
15         }  
16     }  
17     public static Geom2D maxPerimetro(Geom2D geom1, Geom2D geom2){  
18         if (geom1.getPerimetro() > geom2.getPerimetro()) {  
19             return geom1;  
20         } else {  
21             return geom2;  
22         }  
23     }  
24 }
```

0

Ver anotações

Fonte: elaborado pelo autor.

No Código 2.17, em específico nas linhas 3 e 5, temos a declaração de dois objetos polimórficos. Na linha 7, temos a chamada a um método que calcula e retorna o objeto com maior perímetro entre duas figuras geométricas 2D. Por fim, das linhas 9 a 15, temos um processamento feito baseado no tipo de objeto retornado; aqui, o tipo de objeto mais genérico é convertido para um tipo de objeto mais específico, e essa é uma das vantagens do polimorfismo: a capacidade que um objeto possui de assumir diversas formas diferentes.

Caro estudante, nesta seção, você estudou os conteúdos relacionados à organização de pacotes, à importação de pacotes, a modificadores de acesso, a encapsulamento, à herança e a polimorfismo, bem como foi mostrado diversos exemplos sobre a implementação desses conceitos em Java. Na seção seguinte, estudaremos o tratamento de exceção e o uso de classes abstratas, o que nos possibilitará avançar ainda mais no entendimento da linguagem Java.

0

Ver anotações

REFERÊNCIAS

ARANTES, J. da S. **Github**. 2020. Disponível em: <https://bit.ly/3eiUMcF>. Acesso em: 20 mai. 2020.

ASLAM, A. Access Modifiers in Java. **SlideShare**, publicado em 28 fev. 2017.

Disponível em: <https://bit.ly/2QpeVV3>. Acesso em: 22 jul. 2020.

CAELUM. **Java e orientação a objetos**: curso FJ-11. [S.I.]: Caleum, [s.d.]. Disponível em: <https://bit.ly/3ijX34d>. Acesso em: 17 mai. 2020.

CURSO de Java #01 - História do Java - Gustavo Guanabara. [S.I.: s.n.], 2015. 1 vídeo (36 min). Publicado pelo canal Curso em Vídeo. Disponível em: <https://bit.ly/2YvIJDZ>. Acesso em: 22 jul. 2020.

CURSO POO Teoria #12a - Conceito Polimorfismo (Parte 1). [S.I.: s.n.]. 1 vídeo (28:42 min). Publicado pelo canal Curso em Vídeo. Disponível em: <https://bit.ly/2YBc0wW>. Acesso em: 21 jul. 2020.

CURSO POO Teoria #13a - Conceito Polimorfismo (Parte 2). [S.I.: s.n.]. 1 vídeo (20:14 min). Publicado pelo canal Curso em Vídeo. Disponível em: <https://bit.ly/3b0hF4H>. Acesso em: 21 jul. 2020.

DEITEL, P. J.; DEITEL, H. M. **Java: como programar**. 10. ed. São Paulo: Pearson Education, 2016.

GITAHEAD. **GitAhead**. 2019. Disponível em: <https://bit.ly/3aVZihd>. Acesso em: 21 jul. 2020.

GITHUB. **GitHub**. 2020. Disponível em: <https://bit.ly/2ZSr1ud>. Acesso em: 21 jul. 2020.

GITKRAKEN. **GitKraken Git GUI**. 2020. Disponível em: <https://bit.ly/3huTDRI>. Acesso em: 21 jul. 2020.

MVN REPOSITORY. **What's new in maven**. 2020. Disponível em: <https://bit.ly/34zGEdT>. Acesso em: 21 jul. 2020.

ORACLE. **Package java.lang**. 2018. Disponível em: <https://bit.ly/31uQPi3>. Acesso em: 21 jul. 2020.

SOURCETREE. **Sourcetree**. 2020. Disponível em: <https://bit.ly/3jlhELx>. Acesso em: 21 jul. 2020.

TORTOISEGIT. **Tortoise Git**. 2020. Windows Shell Interface to Git. Disponível em: <https://bit.ly/2YCpvMT>. Acesso em: 21 jul. 2020.

FOCO NO MERCADO DE TRABALHO

REUTILIZAÇÃO DE CLASSES

Jesimar da Silva Arantes

0

Ver anotações

ORGANIZANDO O PROJETO, UTILIZANDO A ESTRUTURA DE PACOTES DO JAVA

Criação de modificadores de acessos adequados aos atributos e métodos, e de métodos *getters* e *setters* para encapsular os atributos, utilização de herança na aplicação, além de sobrescrever os métodos *toString* e *equal*.



Fonte: Shutterstock

Deseja ouvir este material?

Áudio disponível no material digital.

SEM MEDO DE ERRAR

A empresa em que você trabalha lhe passou, novamente, mais algumas tarefas para desenvolver. As tarefas listadas abaixo foram especificadas para você:

- Organizar melhor o seu projeto utilizando a estrutura de pacotes.
- Criar modificadores de acessos que sejam adequados aos atributos e métodos.

- Criar, de forma adequada, métodos *getters* e *setters* para os atributos.
- Utilizar o conceito de herança na sua aplicação.
- Sobrescrever os métodos *toString* e *equals*.
- Continuar a utilizar o GitHub e estudar alguns clientes GUI para o GitHub.

0

[Ver anotações](#)

De forma a iniciar o trabalho, você decidiu revisar o Código 2.9 para adicionar os recursos mencionados. Inicialmente, você criou dois pacotes: *simulator.code* (aqui, ficarão as classes Robo, Caixa e Mundo2D) e *simulador.main* (em que ficará a classe App). Após isso, criou os devidos modificadores de acesso para os atributos, bem como os métodos *getters* e *setters* para os atributos, de forma a garantir um melhor encapsulamento, e uma classe chamada Caixaldeia, que será uma superclasse de Caixa.

Por fim, então, você decidiu sobrescrever os métodos *toString* (na classe Caixa e Robo) e *equals* (na classe Robo), finalizando o seu programa com todas as alterações sugeridas pelo seu chefe.

O seu programa ficou parecido com o Código 2.18. Repare que algumas poucas partes foram omitidas. O código completo pode ser acessado no GitHub do autor.

Código 2.18 | Modelagem das classes Robo, Caixaldeia, Caixa e Mundo2D

o

Ver anotações

0

[Ver anotações](#)

```
1 public class Robo {  
2     private float posicaoX;  
3     private float posicaoY;  
4     private int orientacao;  
5     private final String nome;  
6     private final float peso;  
7     private final float velocidadeMax = 5;  
8     private final float pesoCargaMax = 20;  
9     private final String tipoTracao = "esteira";  
10    public static final int FRENTA = 0;  
11    public static final int ATRAS = 1;  
12    public static final int ESQUERDA = 2;  
13    public static final int DIREITA = 3;  
14    ...  
15    public float getPosicaoX() {  
16        return posicaoX;  
17    }  
18    public float getPosicaoY() {  
19        return posicaoY;  
20    }  
21    public int getOrientacao() {  
22        return orientacao;  
23    }  
24    @Override  
25    public String toString() {  
26        return "Robo{" + "posicaoX=" + posicaoX + ", posicaoY="  
27                + posicaoY + ", orientacao=" + orientacao + '}';  
28    }  
29    @Override  
30    public boolean equals(Object obj) {  
31        if (obj instanceof Robo){  
32            Robo robo = (Robo)obj;  
33            return this.nome.equals(robo.nome);  
34        } else {  
35            return false;  
36        }  
37    }  
38}  
39 public class CaixaIdeia {  
40     protected int posX;
```

```
41     protected int posY;
42     protected float peso;
43     protected final float comprimento;
44     protected final float largura;
45     protected final float altura;
46     public CaixaIdeia(int posX, int posY, float peso,
47                         float comprimento, float largura, float altura) {
48         this.posX = posX;
49         this.posY = posY;
50         this.peso = peso;
51         this.comprimento = comprimento;
52         this.largura = largura;
53         this.altura = altura;
54     }
55     ... //métodos gets e sets ocultos para posX, posY e peso.
56 }
57 public class Caixa extends CaixaIdeia {
58     public String nomeItem;
59     public int qtdItem;
60     public Caixa(String nomeItem, int qtdItem, int posX,
61                  int posY, float peso, float comprimento,
62                  float largura, float altura) {
63         super(posX, posY, peso, comprimento, largura, altura);
64         this.nomeItem = nomeItem;
65         this.qtdItem = qtdItem;
66     }
67     @Override
68     public String toString() {
69         return "Caixa{" + "nomeItem=" + nomeItem +
70                 ", qtdItem=" + qtdItem + '}';
71     }
72 }
73 public class Mundo2D {
74     public final int DIM_X;
75     public final int DIM_Y;
76     ...
77 }
```

0

Ver anotações

Após todas essas modificações, você continuou a utilizar o GitHub, conforme sugerido. Inicialmente, você estava utilizando o repositório de código via linha comando, porém, seguindo a sugestão de utilizar alguma ferramenta para facilitar o processo de versionamento, entrou no site do git e viu alguns dos principais clientes GUI (clientes com interface gráfica) para acessar o GitHub, que são: Sourcetree, disponível para Windows e MacOS; TortoiseGit, disponível apenas para Windows; GitKraken, sistema multiplataforma; e GitAhead, que também é um sistema multiplataforma. Caso tenha interesse, escolha e instale uma dessas ferramentas. Como dica, há diversos vídeos tutoriais na internet ensinando-nos como utilizar essas ferramentas.

0

[Ver anotações](#)

TRATAMENTO DE EXCEÇÕES E USO DE CLASSES ABSTRATAS

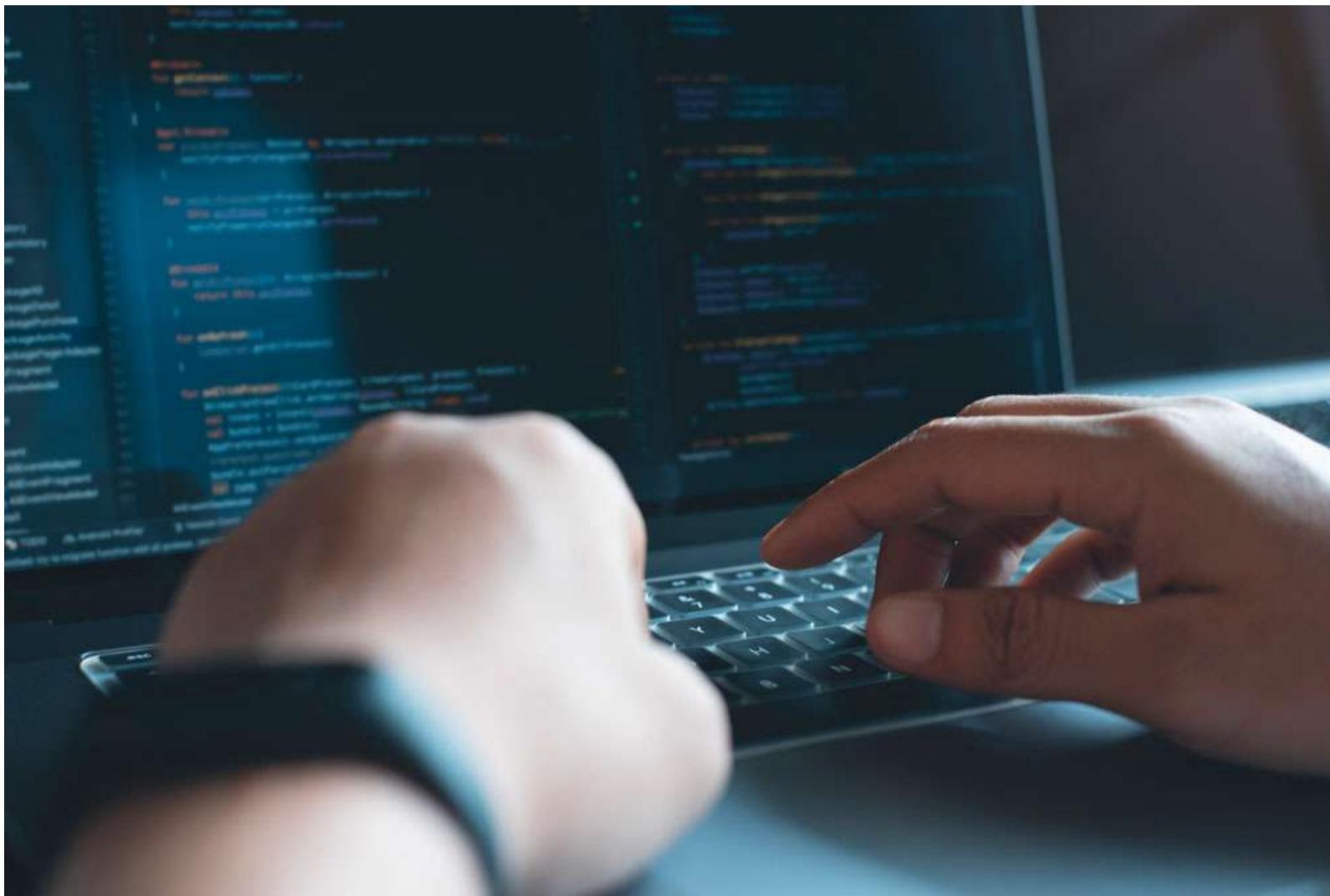
Jesimar da Silva Arantes

0

[Ver anotações](#)

CONSTRUÇÃO DE APLICAÇÕES COM SUPORTE A EVENTUAIS FALHAS

O tratamento de exceções é extremamente útil para a criação de um software mais robusto a falhas, pois pode dar soluções mais elegantes a uma série de problemas.



Fonte: Shutterstock.

Deseja ouvir este material?

Áudio disponível no material digital.

CONVITE AO ESTUDO

Prezado estudante, bem-vindo à unidade do livro *Linguagem Orientada a Objetos*. Nesta unidade, vamos estudar alguns importantes tópicos dentro do desenvolvimento orientado a objetos (OO), bem como estabelecer as ideias por trás das exceções, asserções, classes abstratas, da implementação de interfaces e do desenvolvimento de interfaces gráficas.

Após a leitura desta unidade, espera-se que o leitor seja capaz de fazer os primeiros tratamentos de exceção, tenha aprendido a lançar exceções, criar suas próprias exceções, declarar assertivas, criar classes abstratas, definir métodos

abstratos, criar interfaces, implementar interfaces e desenvolver aplicações com interfaces gráficas.

0

[Ver anotações](#)

Ao longo do capítulo, será incentivada a utilização da plataforma de compartilhamento de código GitHub. O livro apresentará diversos exemplos práticos que o auxiliarão, de forma direta, na compreensão da teoria, e as capacidades de reflexão, modelagem e abstração também serão treinadas ao longo deste livro.

Esta unidade, a fim de ajudá-lo na compreensão da OO na linguagem Java, encontra-se dividida nas seguintes seções: a Seção 3.1, em que serão abordados os conceitos por trás do tratamento de exceção, assertivas e classes abstratas; a Seção 3.2, em que trataremos do conceito de interfaces e de exemplos de como implementá-las; e, por fim, a Seção 3.3, em que serão mostrados os passos necessários para a criação de aplicações gráficas na linguagem Java.

Pelo fato desta unidade ser prática, convido você a implementar as atividades aqui propostas. Lembre-se de que a prática leva à perfeição.

Bons estudos!

PRATICAR PARA APRENDER

Caro aluno, bem-vindo à primeira seção da terceira unidade dos estudos sobre linguagem orientada a objetos.

Qual desenvolvedor nunca se perguntou como construir uma aplicação que dê um maior suporte a eventuais falhas? Acredito que todo desenvolvedor já ficou chateado por ver a sua aplicação travar e fechar. No geral, não desejamos ver as nossas aplicações travarem e fecharem simplesmente por termos digitado um valor literal em vez de um valor numérico. Pois bem, nesta seção, estudaremos o tratamento de exceções, que dará soluções mais elegantes a uma série de problemas. Saber realizar o tratamento de exceção é extremamente útil quando se deseja criar um software mais robusto a falhas.

Nesta seção, estudaremos, também, como fazer a entrada de dados por meio de passagem de valores pela linha de comando, a leitura de dados pelo teclado e a compilação e execução de códigos e Java pelo terminal de comandos. Além disso, trabalharemos com as assertivas e criaremos classes abstratas que auxiliam na reusabilidade de código e facilitam a manutenção de aplicações.

Como forma a contextualizar sua aprendizagem, tenha em mente a *startup* em que você foi contratado. Mais uma vez, o seu chefe deseja dar continuidade à modelagem do robô que você iniciou na primeira seção da primeira unidade. Ele

acredita que a modelagem desenvolvida pode ser melhorada ao serem incorporados novos recursos que embasarão, cada vez mais, o simulador de robô que você deve construir. Dessa maneira, ele pediu a você que fizesse uma leitura do teclado do computador para movimentar o robô, e percebendo que não fez um tratamento de exceção em seu código, solicitou que estudasse esse assunto e melhorasse a sua implementação, deixando-a mais tolerante a falhas. Além disso, ele também lhe pediu para criar uma classe robô abstrata que modelasse a ideia da entidade robô, pois viu uma classe que você havia criado, chamada *Caixaldeia*, e percebeu que ela devia ser do tipo abstrata e não concreta.

Diante do desafio que lhe foi apresentado, como você fará a leitura de dados do teclado e o tratamento de exceção que lhe foi pedido? O que é tratamento de exceção? Como você irá criar essas classes abstratas? O que é uma classe abstrata e uma classe concreta? Esta seção irá auxiliá-lo na resposta de tais perguntas.

Muito bem, agora que você foi apresentado à sua nova situação-problema, estude esta seção e compreenda como a linguagem Java trata exceções e como ela faz a modelagem de classes abstratas. Esses conceitos são fundamentais na OO e trarão uma maior sofisticação na construção do simulador de robô que você deve fazer. Vamos juntos compreender esses conceitos para poder resolver esse desafio?

Bom estudo!

CONCEITO-CHAVE

A linguagem Java possui suporte a alguns recursos muito importantes, que são tratamento de exceção, assertivas e classes abstratas. Esses são alguns dos conceitos que serão abordados nesta seção, porém, gostaríamos, em primeiro lugar, de explicar um pouco sobre como interagir com uma aplicação.

ENTRADA DE DADOS OU LEITURA DE VALORES EM JAVA

Até este momento, você ainda não aprendeu nenhuma forma de entrada de dados ou leitura de valores em Java. Existem duas formas principais para se interagir com uma aplicação; a primeira delas se dá passando-se argumentos no momento da execução do programa, já a segunda se dá por meio da leitura de valores em tempo real. Estudaremos, a seguir, essas duas formas.

PASSAGEM DE ARGUMENTOS VIA LINHA DE COMANDO

o

Ver anotações

A forma de interação por meio da passagem de argumentos via linha de comando é similar à linguagem C. A linguagem Java possui um ponto de entrada da aplicação com a seguinte assinatura: *public static void main(String[] args)*. O argumento args é, na verdade, um vetor de Strings, ou seja, pode passar quantos argumentos quiser, e a aplicação Java poderá utilizá-los. Analise o Código 3.1 a seguir para entender como isso funciona.

o

Ver anotações

Código 3.1 | Implementação básica para passagem de argumentos via linha de comando

```
1 public class ArgsLinhaDeComandoBasico {  
2     public static void main(String[] args) {  
3         System.out.printf("qtd de argumentos = %d%n", args.length);  
4         for (int i = 0; i < args.length; i++) {  
5             System.out.printf("\targs[%d] = %s%n", i, args[i]);  
6         }  
7     }  
8 }
```

Fonte: elaborado pelo autor.

O Código 3.1 apresenta cinco pequenas novidades, que são:

- A utilização do parâmetro args, que recebe valores no momento da execução do código.
- A utilização do atributo *length* do vetor args, que retorna à quantidade de argumentos passados.
- A utilização do acesso às posições do vetor args por meio da notação de colchetes, similar à manipulação de vetores na linguagem C.
- A utilização de alguns especificadores de formato, como o %d, %s e %n. De forma geral, o %n tem o mesmo efeito que o \n, ou seja, pula uma linha.
- A utilização de algumas sequências de escape, como o \t, que é similar à linguagem C.

Analise o Quadro 3.1 a seguir para aprender/relembrar as principais sequências de escape presentes na linguagem Java.

Sequência de Escape	Descrição	Exemplo de utilização
\n	Insere nova linha.	<code>System.out.print("Introdução\nna\nProgramação\ncom\nJava");</code>
\t	Insere tabulação na horizontal.	<code>System.out.print("Col A\tCol B\tCol C\tCol D");</code>
\	Insere barra invertida.	<code>System.out.print("C:\\\\Windows\\\\system32");</code>
\"	Insere aspa dupla.	<code>System.out.print("Nome do livro \"Dom Quixote\" de Miguel de Cervantes");</code>
\r	Realiza retorno do carro.	<code>System.out.print("Texto Não Mostrado\rEsse Texto Aparece\\n");</code>

o

Ver anotações

Fonte: elaborado pelo autor.

Analise o Quadro 3.2 a seguir para aprender/relembrar os principais especificadores de formato presentes na linguagem Java.

Quadro 3.2 | Síntese de especificadores de formato na linguagem Java

Especificador	Descrição	Exemplo de Utilização	Saída
%d	Valor inteiro em decimal com sinal (pode ser usado para <i>byte</i> , <i>short</i> , <i>int</i> e <i>long</i>).	<code>System.out.print("%d", 127);</code>	127
%o	Valor inteiro em octal com sinal.	<code>System.out.print("%o", 127);</code>	177
%x	Valor inteiro em hexadecimal com sinal (minúsculo).	<code>System.out.printf("%x", 127);</code>	7f
%X	Valor inteiro em hexadecimal com sinal (maiúsculo).	<code>System.out.printf("%X", 127);</code>	7F

Especificador	Descrição	Exemplo de Utilização	Saída
%f	Valor real (<i>float</i> ou <i>double</i>).	System.out.printf("%f", 3.141592);	3,141592
%e	Valor real (notação exponencial) (minúsculo).	System.out.printf("%e", 3.14);	3,14e+00
%E	Valor real (notação exponencial) (maiúsculo).	System.out.printf("%E", 3.14);	3,14E+00
%X	Valor inteiro em hexadecimal com sinal (maiúsculo).	System.out.printf("%x", 10);	7F
%b	Valor lógico (<i>boolean</i>) (minúsculo).	System.out.printf("%b", 3 > 2);	true
%B	Valor lógico (<i>boolean</i>) (maiúsculo).	System.out.printf("%B", 2 > 3);	FALSE
%c	Caractere (normal).	System.out.printf("%c", 'v');	v
%C	Caractere (maiúsculo).	System.out.printf("%C", 'v');	V
%s	String (normal).	System.out.printf("%s", "String");	String
%S	String (maiúscula).	System.out.printf("%S", "String");	STRING
%%	Imprime símbolo porcentagem.	System.out.printf("%d%%", 68);	68%
%n	Insere uma nova linha portável.	System.out.printf("L1%nL2");	L1 L2

o

Ver anotações

DICA

Uma dica muito importante para fixar e compreender as sequências de escape e os especificadores de formato é criar exemplos simples em que estes são utilizados. Gostaríamos de destacar que os especificadores de formato mostrados no Quadro 3.2 são utilizados principalmente no comando *System.out.printf*. Para complementar o seu estudo, assista ao vídeo *Curso de Java 63: printf*.

0

[Ver anotações](#)**LEMBRE-SE**

Ao utilizar os comandos *System.out.println()* ou *System.out.print()*, utilize o \n para pular linhas; já para os comandos *System.out.printf()*, *System.out.format()*, *String.format()* ou para a classe *Formatter*, utilize o %n para pular linhas. O %n é um pulador de linhas portável entre diferentes sistemas operacionais. Em muitos casos, o \n funcionará dentro do comando *System.out.printf()*, porém é bom evitá-lo, a fim de que não tenha problemas quando utilizar seu código em outras plataformas.

Explicamos, brevemente, o funcionamento do Código 3.1, porém não mencionamos como executá-lo; logo, a fim de simplificarmos as coisas, recomendamos a criação de apenas um arquivo, chamado *ArgsLinhaDeComandoBasico.java*. (Como dica, não coloque esse arquivo dentro de nenhum pacote, a fim de simplificar a compilação.) Em seguida, abra um terminal de comandos no diretório em que está salvo esse arquivo e compile o código utilizando o compilador do Java chamado javac.

```
$ javac ArgsLinhaDeComandoBasico.java
```

ou

```
$ javac NomeDoPrograma.java
```

Após esse comando mostrado acima, será gerado um arquivo com o nome *ArgsLinhaDeComandoBasico.class*. Para executar esse arquivo, digite o seguinte comando no terminal (repare que os argumentos chamados arg1 arg2 e arg2 são parâmetros que são passados ao programa principal).

```
$ java ArgsLinhaDeComandoBasico arg1 arg2 arg3
```

ou

```
$ java NomeDoProgramma lista_de_argumentos_separados_por_espaço
```

0

[Ver anotações](#)

Dessa maneira, o Código 3.1 produzirá a seguinte saída.

```
quantidade de argumentos = 3
```

```
args[0] = arg1
```

```
args[1] = arg2
```

```
args[2] = arg3
```

0

[Ver anotações](#)

Pronto, agora você já sabe como criar um simples programa em Java que é compilado e executado por meio do terminal, bem como passar argumentos via linha de comando para interagir com a aplicação.

ASSIMILE

Ao observarmos o Quadro 3.2, percebemos que a saída do comando "System.out.printf("%e", 3.141592);" foi o valor 3,141592, utilizando-se a vírgula como separador decimal e não o ponto. Na verdade, a saída pode utilizar qualquer uma das formas, tanto o **ponto** (utilizado, por exemplo, nos EUA) quanto a **vírgula** (utilizada, por exemplo, no Brasil). De forma a compreender como alterar o formatador de impressão utilizado, implemente o Código 3.2 mostrado a seguir e alterne a execução do código comentando e descomentando as linhas 5 e 6.

Código 3.2 | Implementação básica que define a localização para formatar a saída usada da impressão

```
1 import java.util.Locale;
2 public class ExemploLocalizacao {
3     public static void main(String[] args) {
4         //alterne a execução com essas duas linhas
5         Locale.setDefault(new Locale("pt", "BR"));
6         //Locale.setDefault(new Locale("en", "US"));
7         double pi = 3.141592;
8         System.out.printf("Valor Pi: %f%n", pi);
9     }
10 }
```

Fonte: elaborado pelo autor.

Ao utilizar a linha 5 e comentar a linha 6, a localização do Brasil é especificada, assim, os números são impressos utilizando-se o separador vírgula. Ao utilizar a linha 6 e comentar a linha 5, a localização dos EUA é especificada, assim, os números são impressos utilizando-se o separador ponto.

0

[Ver anotações](#)

Agora que vimos o básico de passagem de argumentos via linha de comando, podemos construir uma aplicação mais complexa que utiliza esse recurso. Dessa forma, crie um novo projeto com o nome **AppCmdMath**. Dentro desse projeto, crie uma única classe com o nome *ArgsLinhaDeComandoMath.java*. O Código 3.3 mostra o conteúdo dessa classe, veja:

Código 3.3 | Aplicação matemática que utiliza argumentos via linha de comando

o

[Ver anotações](#)

```
1 import java.util.Locale;
2 public class ArgsLinhaDeComandoMath {
3     public static void main(String[] args) {
4         Locale locale = new Locale("en", "US");
5         if (args.length == 0) {
6             System.err.println("Precisa de Argumentos");
7             System.exit(0);
8         }
9         String resp = "ERROR";
10        args[0] = args[0].toLowerCase();
11        if (args[0].equals("--help")) {
12            resp = "Programa: Cmd Math via args\n" +
13                "Funções:\n\tsqrt x\n\tpow x y\n\tlog10 x\n" +
14                "Constantes:\n\tPI\n\tE\n\tPHI\n";
15        } else if (args[0].equals("--author")) {
16            resp = "Autor: Jesimar da Silva Arantes";
17        } else if (args[0].equals("--version")) {
18            resp = "Versão: 1.0";
19        } else if (args[0].equals("sqrt")) {
20            resp = Math.sqrt(Double.parseDouble(args[1])) + "";
21        } else if (args[0].equals("pow")) {
22            double x = Double.parseDouble(args[1]);
23            double y = Double.parseDouble(args[2]);
24            resp = Math.pow(x, y) + "";
25        } else if (args[0].equals("log10")) {
26            resp = Math.log10(Double.parseDouble(args[1])) + "";
27        } else if (args[0].equals("pi")) {
28            resp = Math.PI + "";
29        } else if (args[0].equals("e")) {
30            resp = Math.E + "";
31        } else if (args[0].equals("phi")) {
32            resp = 1.618033988749895 + "";
33        }
34        System.out.println(resp);
35    }
36 }
```

0

Ver anotações

Fonte: elaborado pelo autor.

Ao executar esse código, os seguintes argumentos são válidos:

- **--help**: imprimirá na tela o nome do programa seguido pela forma de uso com algumas funções e constantes matemáticas.
- **--author**: imprimirá na tela o autor desse código.
- **--version**: imprimirá na tela a versão do código.
- **sqrt x**: calcula a raiz quadrada de x e imprime na tela o resultado, sendo x um número real.
- **pow x y**: calcula x elevado a y e imprime na tela o resultado, sendo x e y números reais.
- **log10 x**: calcula o logaritmo de x na base 10 e imprime na tela o resultado, sendo x um número real.
- **PI**: imprime o número pi na tela.
- **E**: imprime o número de Euler (E) na tela.
- **PHI**: imprime o número de ouro (PHI) na tela.

Algumas considerações sobre o Código 3.3 são necessárias: na linha 4, foi especificado que desejamos utilizar o sistema de medidas dos EUA, assim, a impressão dos valores numéricos decimais será dada utilizando-se ponto e não vírgula. Nas linhas 5 a 8, o programa verifica se foi passado argumentos via linha de comando, e caso não haja argumentos, o programa imprimirá uma mensagem na saída de erro invocada com o *System.err*. Até então, os nossos programas apenas utilizavam a saída padrão que é invocada com o *System.out*. Na linha 10, foi convertido o primeiro argumento para minúsculo, dessa maneira, dará na mesma digitar --help ou --HELP ou --HeLp. A próxima novidade está nas linhas 20, 22, 23 e 26, que contêm a chamada para o método *Double.parseDouble(args[1])*. O método *parseDouble* da classe *Double* é um método estático que recebe uma *String* como argumento e a converte para o tipo primitivo *double*.

No Código 3.1, foi sugerido compilar o arquivo .java, gerando um arquivo .class, e executar o .class via linha de comando. Agora, faremos um procedimento um pouco diferente: geraremos um arquivo .jar e o executaremos por meio da linha de comando, passando os argumentos necessários. Dessa maneira, procure como gerar um arquivo .jar utilizando o seu IDE. Em geral, é só apertar um botão na interface do IDE (no Netbeans tem um botão chamado "Limpar e Construir

Projeto") ou, então, apertar *Shift+F11*. Após isso, um arquivo .jar será gerado dentro de um diretório chamado *dist*. Abra um terminal de comandos nesse diretório *dist* e, então, mande executar o código da seguinte forma:

```
$ java -jar AppCmdMath.jar --help
```

ou

```
$ java -jar NomeDoProjeto lista_de_argumentos_separados_por_espaço
```

Após esse comando, a seguinte saída será impressa na tela:

Programa: Cmd Math via args

Funções:

sqrt x

pow x y

log10 x

Constantes:

PI

E

PHI

Experimente fazer a execução do programa acima conforme sugerido nas linhas abaixo. Em cada uma das execuções, tente entender o resultado que será gerado.

Execução 1: \$ java -jar AppCmdMath.jar --author

Execução 2: \$ java -jar AppCmdMath.jar --version

Execução 3: \$ java -jar AppCmdMath.jar sqrt 25

Execução 4: \$ java -jar AppCmdMath.jar pow 2 4

Execução 5: \$ java -jar AppCmdMath.jar log10 100

Execução 6: \$ java -jar AppCmdMath.jar PI

Execução 7: \$ java -jar AppCmdMath.jar E

Execução 8: \$ java -jar AppCmdMath.jar PHI

No texto acima, foram mencionados os arquivos .class e .jar. Um arquivo .class é um arquivo binário gerado pelo compilador do Java. Um conjunto de arquivos .class pode ser agrupado para formar um arquivo .jar. Um Java ARchive (JAR) é, basicamente, um arquivo compactado usado para distribuir um conjunto de classes do Java (.class e .java), podendo-se ter também dentro do JAR arquivos de imagens, xml, json, txt, entre outros. Além disso, um arquivo .jar pode ser também pensado como um executável da aplicação, uma vez que, a partir dele, executamos o programa java.

DICA

O texto acima referiu-se a um diretório chamado *dist*, em que é gerado o arquivo .jar contendo o executável do projeto. O Quadro 3.3 sintetiza a organização geral em termos de estrutura de diretórios de um projeto qualquer feito na linguagem Java. Analise a descrição feita nesse quadro e navegue também pelos seus projetos, em seu computador, para conhecer melhor essa estrutura que os IDEs criam automaticamente.

Quadro 3.3 | Organização geral das pastas (diretório) em um projeto Java

Diretório	Descrição do Conteúdo
<i>src/</i>	Código-fonte da aplicação.
<i>test/</i>	Código de teste unitário (não utilizado neste livro).
<i>lib/</i>	Dependências do projeto.
<i>dist/</i>	Arquivos de distribuição como .jar e suas dependências.
<i>build/</i>	Arquivos gerados pelo processo de compilação.

Fonte: elaborado pelo autor.

| LEITURA DE DADOS EM TEMPO REAL

Acima, vimos como fazer a entrada de valores para a aplicação com argumentos via linha de comando. Uma outra forma de interagirmos com uma aplicação Java se dá por meio da leitura de dados em tempo real via classe *Scanner*, que fornece métodos de leitura com sintaxes diferentes, mas com funcionamento similar à função *scanf* presente na linguagem C. Analise o Código 3.4 mostrado a seguir.

Código 3.4 | Exemplo de leitura de dados em Java utilizando-se a classe *Scanner*

```
1 import java.util.Scanner;
2 public class ExemploLeituraDados {
3     public static void main(String[] args) {
4         Scanner scan = new Scanner(System.in);
5         System.out.print("Digite um valor inteiro (int): ");
6         int entradaInt = scan.nextInt();
7         System.out.print("Digite um valor real (double): ");
8         double entradaDouble = scan.nextDouble();
9         System.out.print("Digite um valor lógico (boolean): ");
10        boolean entradaBoolean = scan.nextBoolean();
11        System.out.print("Digite uma string (uma palavra): ");
12        String entradaPalavra = scan.next();
13        scan.nextLine(); // comando para esvaziar o buffer do teclado
14        System.out.print("Digite uma string (várias palavras): ");
15        String entradaString = scan.nextLine();
16        System.out.println("Saída dos valores lidos: ");
17        System.out.printf("\tValorInteiro: %d%n", entradaInt);
18        System.out.printf("\tValorReal: %f%n", entradaDouble);
19        System.out.printf("\tValorLógico: %b%n", entradaBoolean);
20        System.out.printf("\tValorPalavra: %s%n", entradaPalavra);
21        System.out.printf("\tValorFrase: %s%n", entradaString);
22    }
23 }
```

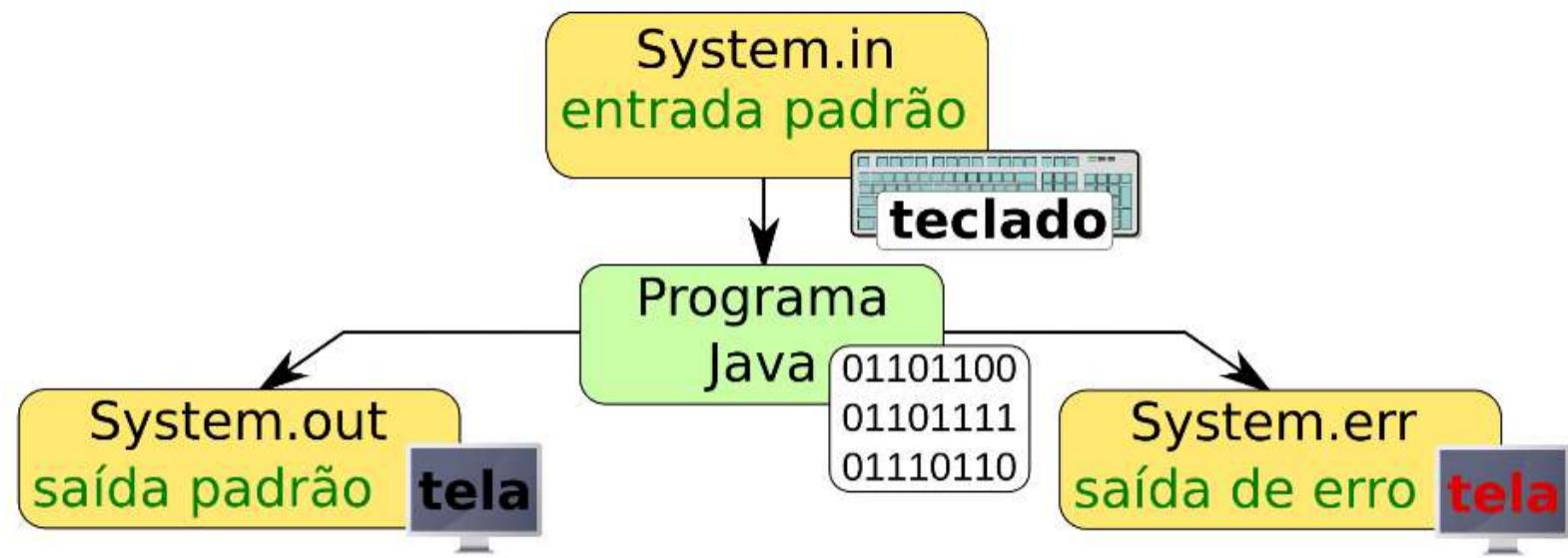
0

Ver anotações

Fonte: elaborado pelo autor.

No Código 3.4, na linha 1, temos a importação da classe *Scanner* que será utilizada para fazer a leitura dos dados. Na linha 4, temos a criação de um objeto do tipo *Scanner* que fará a leitura de dados da entrada padrão (*System.in*). É importante ressaltarmos que a entrada padrão utiliza o dispositivo periférico teclado. A Figura 3.1 nos mostra o fluxo de entrada e saída básico de uma aplicação Java qualquer; já a saída padrão (*System.out*) utiliza a tela do computador (monitor) e a saída de erro (*System.err*) também (veremos mais sobre essa saída ainda nesta unidade).

Figura 3.1 | Fluxo de entrada e saída em qualquer aplicação na linguagem Java



Fonte: elaborada pelo autor.

o

Ver anotações

Ainda no Código 3.4, o método *nextInt()* na linha 6 lê um valor do tipo inteiro (*int*) da entrada; já o método *nextDouble()* na linha 8 lê um valor *double* da entrada. A depender de como estiver configurado o seu Java, você terá de digitar o valor *double* utilizando o separador ponto ou o separador vírgula (dica: pode-se alterar essa forma de entrada de dados utilizando a classe Locale de forma semelhante ao Código 3.2). O método *nextBoolean()* na linha 10 lê um valor lógico da entrada (*true* ou *false*); o método *next()* na linha 12 lê uma String da entrada, mas apenas uma palavra pode ser lida (ao se digitar um espaço, a leitura com o *next()* é interrompida); e na linha 13, tivemos que inserir uma leitura com *nextLine()* para esvaziar o *buffer* do teclado. O problema de *buffer* do teclado está associado, neste caso, à leitura de um \n que não foi consumido na leitura anterior e se dá toda vez que se está lendo algum tipo de dado, como byte, short, int, long, float, double ou string (com o comando *next()*), e, em seguida, é solicitada uma leitura de uma linha completa utilizando-se o *nextLine()*. Já o método *nextLine()* na linha 15 lê uma String da entrada, podendo esta ser uma frase inteira com separador de espaço (sendo que a leitura com *nextLine()* é interrompida com o primeiro *enter* digitado.); e, por fim, nas linhas 17 a 21, temos a impressão na tela dos valores lidos do teclado.

DICA

Caro aluno, implemente o Código 3.4 acima e faça diversos testes com diferentes valores de entrada. Para cada um desses testes, analise os valores de saída, faça testes que falhem ou gerem erros e veja a saída impressa.

TRATAMENTO DE EXCEÇÕES

Um importante recurso presente na linguagem Java é a sua capacidade de lidar com exceções, que são comportamentos fora do padrão e que, em geral, ocorrem poucas vezes.

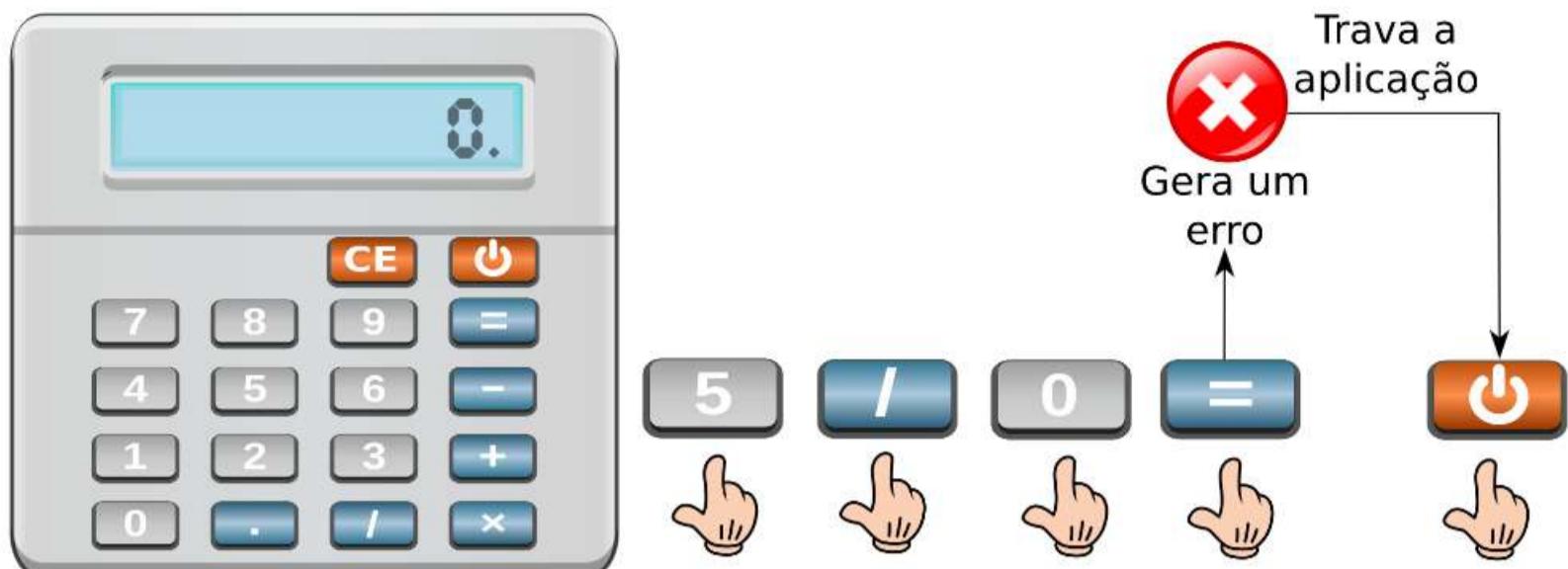
De forma a entender melhor a ideia de tratamento de exceção, vamos considerar o seguinte exemplo: imagine que você programou uma calculadora capaz de executar as seguintes operações básicas: soma, subtração, multiplicação e divisão. Agora, imagine que você quer que essa calculadora seja utilizada por diversas pessoas e, então, pede a um amigo para testá-la. O seu amigo, inicialmente, faz algumas operações básicas, como soma, subtração, multiplicação e divisão, e tudo funciona perfeitamente. Em seguida, ele decide verificar se a sua

calculadora/implementação é realmente robusta; para tanto, divide 5 por 0 e aperta a tecla igual. Após isso, a sua aplicação trava e a única forma que encontra de fazê-la voltar é apertando o botão para desligar e ligar novamente. Esses passos descritos acima podem ser acompanhados na Figura 3.2.

0

[Ver anotações](#)

Figura 3.2 | Esquema didático que representa o lançamento de exceção sem tratamento

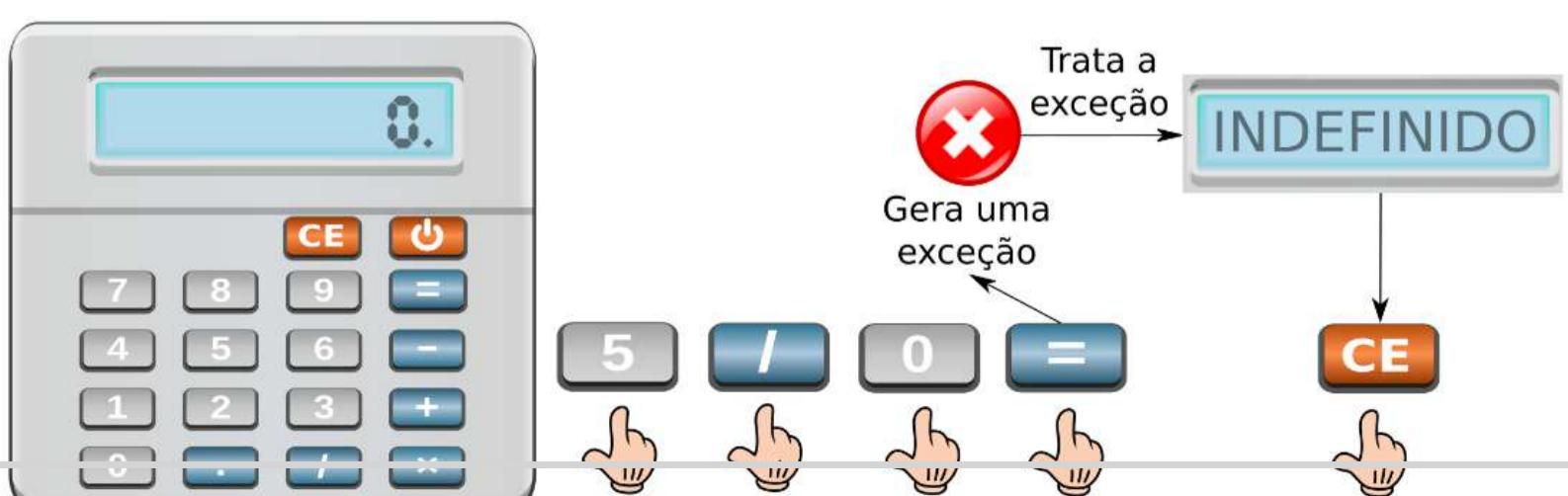


Fonte: elaborada pelo autor.

O seu amigo, então, diz que a calculadora precisa de alguns ajustes. Você, acuado, fala para ele que não, que a calculadora funciona perfeitamente e que o usuário é quem **não** deve fazer uma divisão por zero, afinal, todos sabemos que não existe divisão por zero. Bem, nesse exemplo, o programador está errado; de forma geral, não podemos assumir que o usuário não informará uma operação que não está definida. Você, então, deve tratar isso a nível de código, deixando-o “mais” tolerante a falhas. Em geral, construir um código totalmente livre de falhas é quase impossível, mas você deve buscar minimizar ao máximo esses equívocos.

A Figura 3.3 nos mostra como seria um possível tratamento de exceção em alto nível. Após o usuário digitar 5 dividido por 0 e apertar a tecla igual, uma exceção é gerada e tratada em código; além disso, uma mensagem é exibida no visor da calculadora dizendo que o valor é indefinido. Após isso, para continuar a usá-la, o usuário deverá apertar o botão para limpar o conteúdo, e é essa a essência básica do tratamento de exceção em que damos um tratamento adequado (nesse caso, imprimir uma mensagem dizendo “INDEFINIDO”) a alguma eventual falha que, se não tratada, pode vir a travar a aplicação.

Figura 3.3 | Esquema didático que representa o tratamento de exceção



Fonte: elaborada pelo autor.

0

[Ver anotações](#)

Antes de explicarmos como tratar uma exceção, gostaríamos de propor ao leitor a criação de algumas aplicações simples em Java que lancem alguns tipos de exceção. Cada uma das aplicações deve possuir apenas a função principal *main* e uma das linhas de código do Quadro 3.4 a seguir, assim, será uma aplicação por linha do quadro. Ao executar a primeira aplicação, a divisão do valor inteiro 5 por 0 será feita e, então, uma exceção será gerada. O leitor verá que será impresso algo do tipo “*Exception in thread "main" java.lang.ArithmetricException: / by zero*”, que indica o tipo de exceção lançada, que foi, nesse caso, do tipo *ArithmetricException*, devido à divisão (/) por zero. Após essa mensagem, será mostrado o que chamamos de rastro de pilha ou *stack trace*, que indica o caminho percorrido pelo erro, passando por diversos métodos chamados até o método *main*, que iniciou a execução da aplicação.

Quadro 3.4 | Trechos de código que lançam diversas exceções

	Linhas de Código	Tipo de Exceção Lançada
1	<code>int divPor0 = 5/0;</code>	<code>ArithmetricException</code>
2	<code>int valorStr = Integer.parseInt("A");</code>	<code>NumberFormatException</code>
3	<code>int refNull = Integer.parseInt(null);</code>	<code>NullPointerException</code>
4	<code>String indiceNegativo = args[-1];</code>	<code>ArrayIndexOutOfBoundsException</code>
5	<code>System.out.printf("%d", "5");</code>	<code>IllegalFormatConversionException</code>
6	<code>Scanner sc = new Scanner(System.in);</code> <code>int vFloat = sc.nextInt(); //ler 3.5</code>	<code>InputMismatchException</code>

Fonte: elaborado pelo autor.

O exemplo 2 do Quadro 3.4 lança uma exceção do tipo *NumberFormatException* pois um valor String "A" não pode ser convertido para inteiro. O exemplo 3 do Quadro 3.4 lança uma exceção do tipo *NullPointerException* pois uma referência nula (utilizando-se a palavra-reservada *null*) não pode ser passada para o método *parseInt()*. No Quadro 3.4, todos os erros foram forçados a fim de que o leitor conhecesse algumas das possíveis exceções que podem ser lançadas. Na maioria

das aplicações desenvolvidas, os erros não são tão evidentes quanto esses, mas podem ocorrer, eventualmente, devido à grande complexidade e grande dinamicidade das aplicações.

A seguir, serão estudadas as estruturas de código em Java que são necessárias para se fazer uma aplicação que trata exceções.

- O primeiro comando utilizado é o **try-catch**, em que, na parte do *try* (tentar), estão as linhas de código que são suscetíveis a algum tipo de exceção ou erro; já na parte do *catch* (pegar), há as linhas de código que serão executadas caso alguma exceção ocorra na parte do *try*.
- O segundo comando mostrado é o **try-catch-finally**, em que as partes do *try* e *catch* são iguais ao anterior, e ainda há uma parte chamada *finally* (finalmente), que contém as linhas de código que devem ser executadas independentemente de ocorrer exceção ou não.
- O terceiro comando mostrado é o **try-multicatch-finally**, que nada mais é do que o comando anterior, só que com diversas exceções diferentes que podem ser tratadas.

O Quadro 3.5 apresenta uma síntese dos principais comandos capazes de lidar com o tratamento de exceção.

Quadro 3.5 | Síntese dos comandos: *try-catch*, *try-catch-finally* e *try-multicatch-finally*

cmd: try-catch	cmd: try-catch-finally	cmd: try-multicatch-finally
<pre>try { //tente executar } catch (ClassEx e) { //trate a exceção }</pre>	<pre>try { //tente executar } catch (ClassEx e) { //trate a exceção } finally { //execute sempre }</pre>	<pre>try { //tente executar } catch (ClassEx e) { //trate a exceção } catch (ClassEx e) { //trate a exceção } ... //múltiplos catchs finally { //execute sempre }</pre>

0

Ver anotações

Fonte: elaborado pelo autor.

O Código 3.5 contém uma possível forma de realizar o tratamento de exceção para a divisão por zero de números inteiros.

Código 3.5 | Implementação da operação de divisão que trata a divisão por zero

```

1 public static int div(int a, int b) {
2     try {
3         return a / b;
4     } catch (ArithmeticException ex) {
5         System.err.println("A divisão por zero é indefinida");
6         return 0;
7     }
8 }
```

0

[Ver anotações](#)

Fonte: elaborado pelo autor.

Os métodos em Java podem também lançar exceções, pois, às vezes, desejamos que o tratamento seja feito não no método corrente, mas no método que o chamou, e assim, utilizamos a palavra reservada *throw* para lançar uma exceção.

Analise o Código 3.6, que contém o lançamento de uma exceção, pois não existe o fatorial de um número inteiro negativo. Dessa maneira, a função que chama o método fatorial é que deve tratar números negativos e não o fatorial em si.

Código 3.6 | Implementação do fatorial que lança uma exceção caso n seja negativo

```

1 public static long fatorial(int n) {
2     if (n < 0) {
3         throw new IllegalArgumentException("O n deve ser >= 0");
4     }
5     long fat = 1;
6     for (int i = 1; i <= n; i++) {
7         fat *= i;
8     }
9     return fat;
10 }
```

Fonte: elaborado pelo autor.

A Figura 3.4 nos mostra como é feita a organização das classes do Java em termos da hierarquia de exceções. Assim, no topo da hierarquia, temos a classe *Throwable* (lançável), que modela qualquer objeto que possa lançar uma exceção ou erro. Em seguida, temos, basicamente, dois tipos de problemas que podem ocorrer na linguagem Java: as exceções (classe *Exception* ou suas subclasses) e os erros (classe *Error* ou suas subclasses). Os erros são lançados pela JVM e costumam ser

bem raros de ocorrer. Caso ocorra algum tipo de erro em sua aplicação, não tente tratá-lo; a recomendação é que você simplesmente deixe-o ocorrer e execute novamente a sua aplicação. Em relação às exceções, de forma geral, as subdividimos em checadas ou não checadas. As exceções que herdam

º

RuntimeException são não checadas e ocorrem, essencialmente, em tempo de execução. As exceções que herdam **Error** também são não checadas; já as exceções que herdam **Exception** (exceto a **RuntimeException**) são checadas e podem ocorrer em tempo de compilação. As exceções do tipo checadas obrigam o programador a tratá-las com um bloco *try-catch* ou lançar a exceção para o método chamador utilizando-se o comando *throws*.

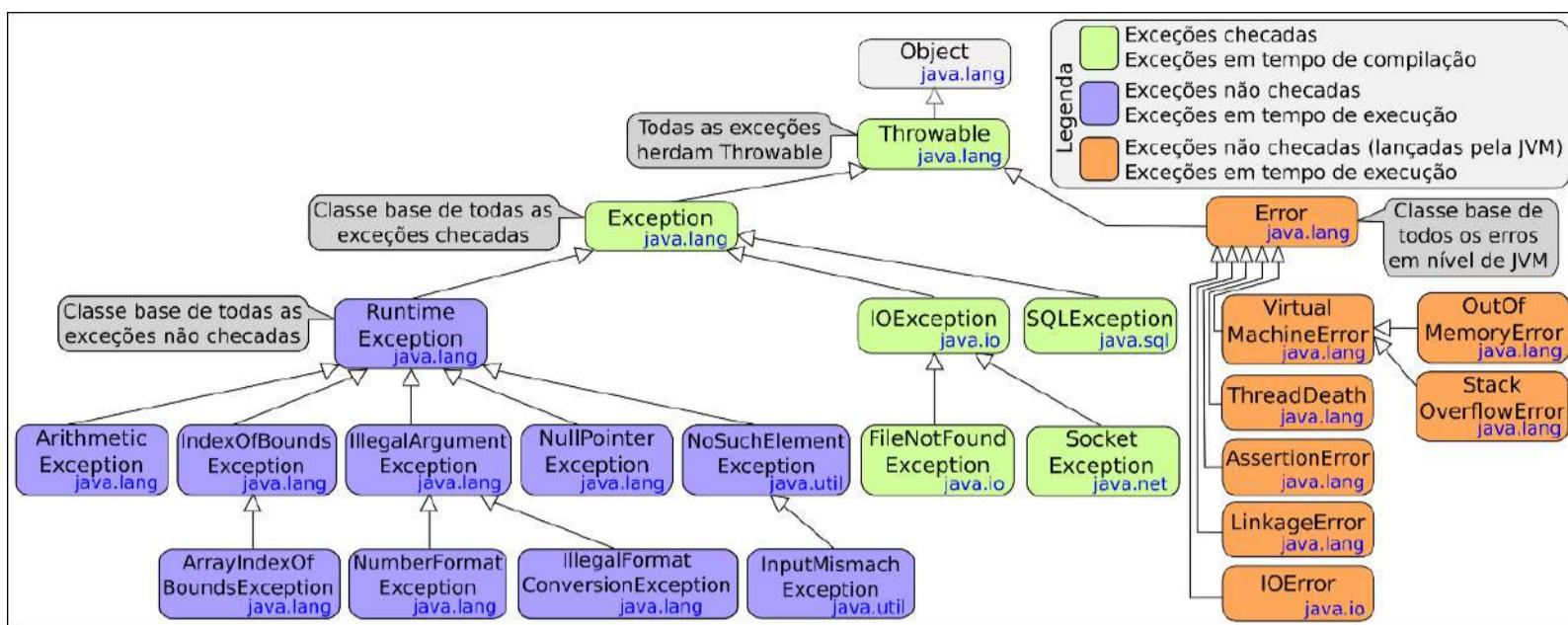
Ver anotações

Existem milhares de classes nativas do Java que auxiliam no tratamento de exceção, no entanto, na Figura 3.4, veremos apenas as principais, assim como, a seguir, veremos como criar as nossas próprias classes que lançam e tratam exceções.

Figura 3.4 | Organização das classes na hierarquia de exceções em Java

0

[Ver anotações](#)



Fonte: elaborada pelo autor.

A seguir, serão brevemente descritas algumas das exceções mostradas na Figura 3.4.

- **ArithmetricException:** lançada quando uma condição aritmética excepcional ocorre, como, por exemplo, uma divisão por zero de números inteiros.
- **IndexOutOfBoundsException:** lançada para indicar que um índice de algum tipo, como um vetor, uma string ou uma matriz, está fora do intervalo.
- **ArrayIndexOutOfBoundsException:** lançada para indicar que um vetor foi acessado com um índice ilegal, como valor negativo ou maior ao tamanho do vetor.
- **IllegalArgumentException:** lançada para indicar que um método recebeu um argumento ilegal ou inapropriado.
- **NumberFormatException:** lançada para indicar que a aplicação tentou converter um valor em algum tipo numérico, mas o valor não possui o formato apropriado.
- **NullPointerException:** lançada quando uma aplicação tenta usar um objeto *null* quando uma instância de objeto é necessária.

A linguagem Java permite-nos criar nossas próprias classes que lançam exceções. Uma boa prática é utilizarmos as classes de exceções que já existem na plataforma, no entanto, nem sempre a linguagem possui uma exceção já programada que representa aquilo que desejamos, assim, devemos criar a nossa própria classe.

Frente a isso, criaremos uma classe que modelará exceções não checadas e que será lançada toda vez que entrarmos com um valor negativo. Como queremos que essa exceção seja não checada, estenderemos a classe `RuntimeException` e, em seguida, três construtores bem simples serão criados. Faz-se importante ressaltar que esse exemplo tem um propósito didático.

o

[Ver anotações](#)

```

1  public class ValorNegativoException extends RuntimeException {
2      public ValorNegativoException() {
3      }
4      public ValorNegativoException(String message) {
5          super(message);
6      }
7      public ValorNegativoException(String msg, Throwable cause) {
8          super(msg, cause);
9      }
10 }

```

0

[Ver anotações](#)

Fonte: elaborado pelo autor.

Agora, para utilizarmos a exceção criada acima, basta modificarmos, por exemplo, a linha 3 do Código 3.6 para lançarmos uma exceção do tipo *ValorNegativoException* em vez de lançarmos a exceção do tipo *IllegalArgumentException*. Gostaríamos de ressaltar a importância de se fazer as implementações e os testes mencionados acima e, só assim, prosseguir na leitura.

■ ASSERTIVAS

A linguagem Java possui um outro recurso chamado asserção ou assertiva. Esse recurso auxilia no desenvolvimento de aplicações e as mantém limpas para a etapa de produção na linha de desenvolvimento de software. Imagine que você precisa criar uma aplicação que receberá um valor inteiro que representará a idade de uma pessoa; todos nós concordamos que não existe idade negativa, mas um valor de idade negativa pode ser erroneamente informado no sistema, podendo acarretar algum tipo de erro no sistema como um todo. Frente a isso, a assertiva é um tipo de tratamento de exceção em nível de desenvolvimento de código, que, quando pronto, tem as assertivas automaticamente removidas.

O Quadro 3.6 apresenta a sintaxe básica das assertivas em Java. Nele, percebemos que as assertivas utilizam a palavra reservada *assert*. Podemos perceber também que existem dois tipos de comandos *assert*: no primeiro, nenhuma mensagem é especificada, já no segundo comando, há uma mensagem especificada. A ideia de

especificar uma mensagem é imprimi-la quando um erro é gerado. Caso a assertiva (*ExprLógica*) seja falsa, um erro do tipo *AssertionError* será lançado e a aplicação encerrada.

0

Quadro 3.6 | Síntese da sintaxe utilizada no comando *assert*

	cmd: assert (sem msg)	cmd: assert (com mensagem)
Sintaxe	<code>assert (ExprLógica);</code>	<code>assert (ExprLógica) : "MsgDeAviso";</code>
Exemplo de Exceção Lançada	Exception in thread "main" <code>java.lang.AssertionError</code>	Exception in thread "main" <code>java.lang.AssertionError: MsgDeAviso</code>

Ver anotações

Fonte: elaborado pelo autor.

O Código 3.8 possui um exemplo básico de assertiva em que valores negativos de idade geram mensagens de alertas e interrompem a execução da aplicação. Uma questão muito importante sobre as asserções é que nunca devemos tratá-las; o ideal é deixarmos as exceções encerrarem a aplicação, logo, o programador deve fazer o devido tratamento para que esse tipo de exceção nunca ocorra. Lembre-se: as asserções não vão para o programa final, elas são usadas apenas durante o desenvolvimento e devem guiar o programador na construção de aplicações que não permitem que tais erros ocorram.

Código 3.8 | Implementação básica para mostrar o uso de assertivas em Java

```
1 public class ExemploAssercao {  
2     public static void main(String[] args) {  
3         //faça testes com as idades 20, 12, -1, 84, -20  
4         int idade = 20;  
5         assert (idade >= 0) : "Aviso: não existe idade negativa";  
6         System.out.printf("Idade: %d%n", idade);  
7     }  
8 }
```

Fonte: elaborado pelo autor.

Como mencionado anteriormente, as assertivas são automaticamente removidas do código quando vai para a produção. Assim, para a execução do programa com a assertiva, a forma de compilação é ligeiramente diferente. A fim de que consiga executar o código acima, o argumento `-enableassertions` (também aceita-se `-ea`) faz-se necessário para habilitar asserções.

A seguir, temos, na primeira linha, a compilação do código `ExemploAssercao.java` (igual ao anterior) e, em seguida, a forma como devemos executar o código.

```
$ javac ExemploAssercao.java
```

```
$ java -enableassertions ExemploAssercao
```

O Código 3.9 mostra um método desenvolvido em produção que verifica os argumentos recebidos de uma função que calcula o Índice de Massa Corporal (IMC). Os argumentos de massa e altura devem satisfazer a todos os requisitos, caso contrário, a aplicação encerra a sua execução.

o

Ver anotações

DICA

Caro aluno, construa o método *main* que invoca *calcularIMC()*, faça diversos testes, passando valores para falhar, e perceba o funcionamento das asserções.

0

[Ver anotações](#)

Código 3.9 | Método que calcula o IMC com uso de assertivas em Java

```
1 public float calcularIMC(float massa, float altura){  
2     assert (massa >= 0) : "Aviso: não existe massa negativa";  
3     assert (altura >= 0) : "Aviso: não existe altura negativa";  
4     assert (massa != 0) : "Aviso: não existe massa igual a 0";  
5     assert (altura != 0) : "Aviso: não existe altura igual a 0";  
6     return massa / (altura * altura);  
7 }
```

Fonte: elaborado pelo autor.

REFLITA

As assertivas geram erros do tipo *AssertionError*, que é uma subclasse de *Error* (consulte novamente a Figura 3.4); dessa maneira, reflita: por que **não** devemos tratar as exceções lançadas pelas assertivas? Por que a maneira de execução do código com as assertivas mudou? Por que o argumento *-enableassertions* não é o argumento *default* de compilação? Por fim, gostaria de propor também ao leitor uma reflexão sobre as diferenças entre as exceções e asserções em Java.

I USO DE CLASSES ABSTRATAS

A linguagem Java possui dois tipos de classes, que são: classes concretas e classes abstratas. As **classes concretas** são as classes que estudamos até aqui. A partir dessas classes, podemos criar instâncias e fazer a manipulação delas. Já as **classes abstratas** são apenas uma abstração de alguma entidade, assim, não podemos fazer a sua instanciação. As classes abstratas servirão de modelo para a criação de subclasses (concretas ou abstratas), e a declaração de uma classe abstrata utiliza a palavra reservada *abstract*.

As classes abstratas podem possuir métodos abstratos, ou seja, métodos que não possuem nenhuma implementação, assim, nesses métodos, são declaradas ~~apenas as suas assinaturas, e a classe que herdar uma classe abstrata é que~~

deverá sobrescrever as assinaturas dos métodos abstratos. A declaração de um método abstrato utiliza a palavra reservada *abstract*. Os métodos do tipo estático não podem ser do tipo abstrato, pois não podem ser sobreescritos. Uma classe abstrata não pode ser declarada como final, pois uma classe final não pode ser herdada.

o

[Ver anotações](#)

Vamos olhar novamente a Figura 2.9 e o Código 2.12 da unidade 2. Perceba que, nesses exemplos, as classes Geom2D e Geom3D foram declaradas como classes concretas. As entidades figuras geométricas 2D e 3D são entidades abstratas e não deviam ser declaradas daquela forma; já as entidades Círculo, Triângulo, Retângulo, Esfera, Tetraedro e Cubo são concretas e foram declaradas de forma adequada ao nosso propósito. Dessa maneira, é necessário fazer uma reimplementação das classes Geom2D e Geom3D de forma mais consistente com o seu significado.

0

[Ver anotações](#)

EXEMPLIFICANDO

De forma a exemplificar a criação de classes abstratas, analise o Código 3.10 que apresenta uma possível reimplementação das classes Geom2D e Geom3D de forma abstrata.

Código 3.10 | Implementação de Geom2D e Geom3D de forma abstrata

```
1 //declaração da classe abstrata
2 public abstract class Geom2D {
3     protected double perimetro;
4     protected double area;
5     //métodos abstratos calcPerimetro e calcArea
6     public abstract double calcPerimetro();
7     public abstract double calcArea();
8     public double getPerimetro() {
9         return perimetro;
10    }
11    public double getArea() {
12        return area;
13    }
14 }
15
16 //declaração da classe abstrata
17 public abstract class Geom3D {
18     protected double area;
19     protected double volume;
20     //métodos abstratos calcArea e calcVolume
21     public abstract double calcArea();
22     public abstract double calcVolume();
23     public double getArea() {
24         return area;
25    }
26     public double getVolume() {
27         return volume;
28    }
29 }
```

0

Ver anotações

Fonte: elaborado pelo autor.

No Código 3.10, na linha 2, foi feita a declaração da classe Geom2D como abstrata; nas linhas 6 e 7, foram declarados os métodos *calcPerimetro()* e *calcArea()* como abstratos. Repare que colocamos apenas as assinaturas desses métodos, pois as classes que herdarem de Geom2D é que deverão implementar, de fato, esses métodos. O raciocínio é o mesmo para a classe Geom3D. Na implementação das classes concretas, não é necessário alterar nada nas classes Círculo, Triângulo, Retângulo, Esfera, Tetraedro e Cubo. Tente fazer essa adaptação de código e realize alguns testes para entender o seu funcionamento.

o

Ver anotações

Caro estudante, nesta seção, você estudou os conteúdos relacionados a argumentos via linha de comando, à leitura de dados por meio da classe *Scanner*, ao tratamento de exceção, ao lançamento de exceção, à criação de exceção, a assertivas e a classes abstratas, bem como analisou diversos exemplos sobre como esses conceitos são implementados na linguagem Java. Nas seções seguintes, estudaremos melhor como funcionam as interfaces em Java e avançaremos ainda mais no entendimento dessa linguagem.

REFERÊNCIAS

ARANTES, J. da S. **Livro-POO-Java**. 2020. Disponível em: <https://bit.ly/3eiUMcF>. Acesso em: 29 jul. 2020.

CAELUM. **Java e orientação a objetos**: apostila do curso FJ-11. [s.d.]. Disponível em: <https://bit.ly/2ZpTqrZ>. Acesso em: 29 jul. 2020.

DEITEL, P. J.; DEITEL, H. M. **Java**: como programar. 10. ed. São Paulo: Pearson Education, 2016.

LOIANE GRONER. **Curso de Java 63: printf**. 2016. Disponível em: <https://bit.ly/3mf5BIx> . Acesso em: 29 jul. 2020.

LOIANE GRONER. **Curso de Java básico gratuito com certificado e fórum**. 2016. Disponível em: <https://bit.ly/2VUtDGT>. Acesso em: 29 jul. 2020.

ORACLE. **ArrayListOutOfBoundsException**. [s.d.]. Disponível em: <https://bit.ly/33ilAXf>. Acesso em: 29 jul. 2020.

ORACLE. **Class ArithmeticException**. [s.d.]. Disponível em: <https://bit.ly/2DTTRDC>. Acesso em: 29 jul. 2020.

ORACLE. **IllegalArgumentException**. [s.d.]. Disponível em: <https://bit.ly/3iqZfgs>.

Acesso em: 29 jul. 2020.

ORACLE. **IndexOutOfBoundsException**. [s.d.]. Disponível

em: <https://bit.ly/2RjhsAO>. Acesso em: 29 jul. 2020.

ORACLE. **NullPointerException**. [s.d.]. Disponível em: <https://bit.ly/2FnYt5v>. Acesso

em: 12 jul. 2020.

ORACLE. **NumberFormatException**. [s.d.]. Disponível em: <https://bit.ly/2DPDUOE>.

Acesso em: 12 jul. 2020.

FOCO NO MERCADO DE TRABALHO

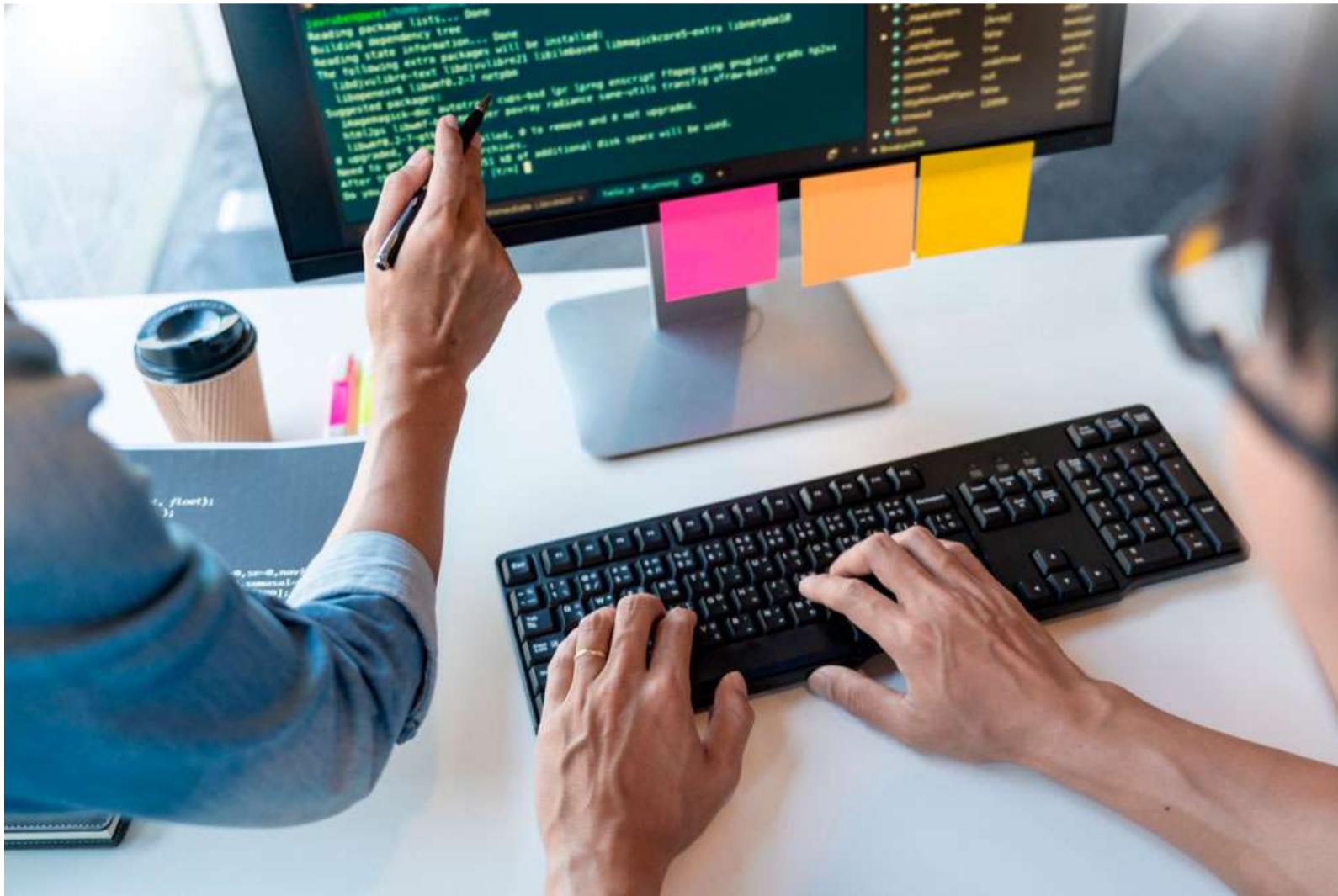
TRATAMENTO DE EXCEÇÕES E USO DE CLASSES ABSTRATAS

Jesimar da Silva Arantes

Ver anotações

MAIOR SOFISTICAÇÃO NA CONSTRUÇÃO DO SIMULADOR DE ROBÔ

Leitura de dados do teclado para movimentação do robô, tratamento de exceção para deixar a aplicação mais tolerante a falhas e criação de uma classe robô abstrata para modelar a ideia da entidade robô.



Fonte: Shutterstock.

Deseja ouvir este material?

Áudio disponível no material digital.

SEM MEDO DE ERRAR

A *startup* em que você faz estágio lhe passou a tarefa de melhorar a modelagem do simulador de robô que está produzindo. Dessa maneira, o seu chefe, após revisar o seu código, listou as seguintes tarefas para desenvolver:

- Fazer a leitura das posições que o robô irá deslocar por meio do teclado.
- Criar um tratamento de exceção em pontos possivelmente críticos.

- Criar uma classe robô abstrata que irá modelar a ideia da entidade robô.
- Transformar a classe *Caixaldeia* em uma classe abstrata.

o

Ver anotações

Após olhar com calma todas as sugestões de seu chefe, você, então, partiu do Código 2.14 que tinha e percebeu que a primeira coisa que devia fazer era alterar a classe App. Nessa classe, você criou um objeto da classe Scanner, que receberá os comandos por meio do teclado, sendo estas as possíveis teclas a serem pressionadas: 'w', 'a', 's' e 'd'. (Caso a tecla zero (0) seja pressionada, então, será encerrada a aplicação, mas caso alguma outra tecla seja pressionada, uma exceção será lançada e uma nova tecla será requisitada.) Em seguida, você criou a classe *Roboldeia*, na qual constam os seguintes atributos: posicaoX, posicaoY, orientacao, nome e peso, bem como a definição de que todo robô deve sobrescrever os métodos move, moveX e moveY. Após isso, a classe *Robo* foi criada como uma subclasse de *Roboldeia*. Entre as linhas 35 e 80, foram destacados alguns aspectos modificados do código anterior. Nessa versão do código, você colocou os métodos move, moveX e moveY para lançarem exceções do tipo *IllegalArgumentException* caso o argumento seja *NaN (Not a Number)* ou infinito. O método *setOrientacao*, por sua vez, também lança uma exceção caso não seja pressionada nenhuma das teclas 'w', 'a', 's' e 'd'.

Por fim, você apenas modificou a assinatura da classe *Caixaldeia*, conforme mostrado na linha 81, para transformá-la em uma classe abstrata. Após esses passos, o seu programa ficou pronto, apresentando todas as alterações do seu chefe, e parecido com o Código 3.11. Repare que diversas partes foram omitidas, mas o código completo pode ser acessado no GitHub do autor.

Código 3.11 | Nova modelagem das classes App, Roboldeia, Robo e Caixaldeia

```
1 public class App {  
2     public static void main(String[] args) {  
3         Robo robo = new Robo();  
4         String tecla = "";  
5         Scanner scan = new Scanner(System.in);  
6         boolean ok = false;  
7         do {  
8             try {  
9                 tecla = scan.next();  
10                robo.setOrientacao(tecla.charAt(0));  
11                robo.printPos();  
12            } catch (IllegalArgumentException ex){  
13                if (tecla.charAt(0) == '0') {  
14                    ok = true;  
15                } else {  
16                    System.out.println("Valor errado");  
17                    scan.nextLine();  
18                    ok = false;  
19                }  
20            }  
21        } while(!ok);  
22        robo.printPos();  
23    }  
24}  
25 public abstract class RoboIdeia {  
26     protected float posicaoX;  
27     protected float posicaoY;  
28     protected int orientacao;  
29     protected String nome;  
30     protected float peso;  
31     public abstract void move(float posX, float posY);  
32     public abstract void moveX(float dist);  
33     public abstract void moveY(float dist);  
34}  
35 public class Robo extends RoboIdeia {  
36     ...  
37     @Override  
38     public void move(float posX, float posY) {  
39         if (Float.isNaN(posX) || Float.isNaN(posY) ||  
40             Float.isInfinite(posX) || Float.isInfinite(posY)) {
```

0

Ver anotações

```
41         throw new IllegalArgumentException("Args não válidos");
42     }
43     super.posicaoX = posX;
44     super.posicaoY = posY;
45 }
46
47 @Override
48 public void moveX(float dist) {
49     if (Float.isNaN(dist) || Float.isInfinite(dist)) {
50         throw new IllegalArgumentException("Arg não válido");
51     }
52     super.posicaoX += dist;
53 }
54
55 @Override
56 public void moveY(float dist) {
57     if (Float.isNaN(dist) || Float.isInfinite(dist)) {
58         throw new IllegalArgumentException("Arg não válido");
59     }
60     super.posicaoY += dist;
61 }
62
63 public void setOrientacao(char tecla) {
64     if (tecla == 'w') {
65         super.orientacao = FRENTE;
66         moveY(5);
67     } else if (tecla == 's') {
68         super.orientacao = ATRAS;
69         moveY(-5);
70     } else if (tecla == 'a') {
71         super.orientacao = ESQUERDA;
72         moveX(-5);
73     } else if (tecla == 'd') {
74         super.orientacao = DIREITA;
75         moveX(5);
76     } else {
77         throw new IllegalArgumentException("Arg não válido");
78     }
79 }
80 }
81 public abstract class CaixaIdeia {
```

82

...

83 }

Fonte: elaborado pelo autor.

0

[Ver anotações](#)

DEFINIÇÃO E USO DE INTERFACES

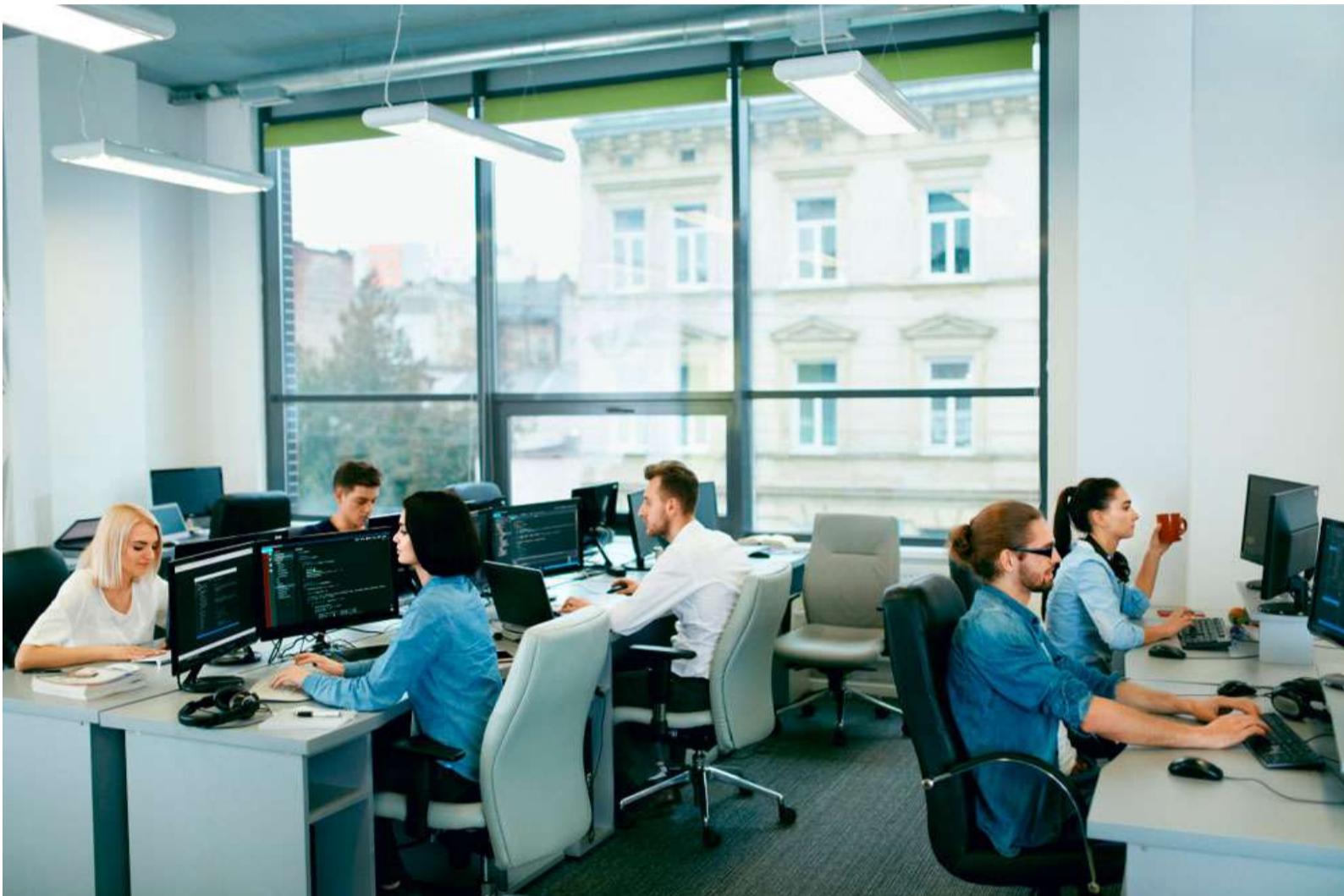
Jesimar da Silva Arantes

0

[Ver anotações](#)

INTERFACES JAVA

O conceito de interfaces em Java é o de sanar a limitação da linguagem em não permitir heranças múltiplas, assim uma interface estabelece um contrato a ser seguido ao desenvolver as classes que implementam as interfaces.



Fonte: Shutterstock.

Deseja ouvir este material?

Áudio disponível no material digital.

PRATICAR PARA APRENDER

Caro estudante, bem-vindo à segunda seção da terceira unidade de estudos sobre Linguagem Orientada a Objetos. Uma vez que estudamos a respeito de herança, às vezes podemos nos perguntar se há a possibilidade de herdarmos mais de uma classe. A linguagem Java não nos permite fazer a herança de mais de uma classe, no entanto, o Java dá suporte a interfaces que são como contratos, que devem ser desenvolvidos pelas classes que os implementam. Ao unir conteúdo de herança, classe abstrata e interface, a linguagem Java consegue oferecer uma grande reusabilidade de código e fechar o assunto de polimorfismo. Nesta seção, você

terá a oportunidade de aprender a sintaxe básica por trás do comando *for each*, que é muito utilizado para se iterar sobre coleções, como vetores e listas, além de como se faz para passar para métodos uma quantidade variada de argumentos e como trabalhar com enumeração em Java.

0

[Ver anotações](#)

De forma a contextualizar sua aprendizagem, imagine que você foi contratado em uma *startup*. A ideia é que você desenvolva um simulador completo de robô que seja capaz de transportar caixas em uma sala. O seu chefe está muito satisfeito com o que você está desenvolvendo, porém sabe que ainda há muito trabalho a se fazer. Após revisar o seu código, percebeu que ele não estava devidamente documentado, então pediu a você que começasse a documentá-lo utilizando a estrutura do *javadoc*. Além disso, ele também notou que em seu código havia algumas constantes declaradas como *static* e *final*, mas que ele acreditava que ficariam melhores utilizando-se enumeradores. O seu chefe também identificou que o seu código não possuía nenhuma descrição de utilização para quando fosse executado via linha de comando, logo, pediu a você que imprimisse algumas informações, caso requisitadas durante a execução pelo terminal. Por fim, ele deseja que o robô possa receber, por meio da execução por linha de comando, uma lista de comandos ou movimentações possíveis, bem como executar esses movimentos automaticamente; para isso, ele disse a você que é possível utilizar o comando *for each* e a ideia de argumentos variáveis (*varargs*) do Java.

o

Ver anotações

Diante desse desafio que lhe foi dado, como documentar o seu código? O que é *javadoc*? O que é enumeração em Java? Como utilizar o comando *for each*? O que são argumentos variáveis?

Esta seção o ajudará a ter um maior domínio dessas tarefas requisitadas pelo seu chefe.

Agora que você foi apresentado à sua nova situação-problema, estude esta seção e compreenda esses recursos presentes na linguagem Java. Esses recursos são de extrema importância para a construção de qualquer aplicação de pequeno e grande porte. E aí, vamos juntos compreender esses conceitos e, então, resolver esse desafio?

Bom estudo!

CONCEITO-CHAVE

A linguagem Java possui suporte para um recurso muito importante, que é a criação de interfaces. Porém, antes de iniciarmos esse assunto, gostaríamos de explicar um pouco a importância da documentação de código, a utilização do comando *for each* e como passar uma quantidade variável de parâmetros para um método, bem como apresentar um tipo especial de classe, que é a enumeração

DOCUMENTAÇÃO DE CÓDIGO

Independentemente da linguagem de programação utilizada, construir um código legível e bem documentado é extremamente importante, pois ele necessitará de atualizações e poderá ser lido por outros programadores. A forma mais simples de documentar um código se dá por meio de comentários.

o

Existem dois tipos de comentários em Java:

- O primeiro trata-se do comentário de uma única linha, em que se utiliza o comando `//` para inserção do comentário;
- Já o segundo refere-se aos comentários de múltiplas linhas, em que se utiliza o comando `/* conteúdo */` para inserção do comentário.

Ver anotações

No entanto, essas duas formas são relativamente pobres e adequadas apenas para pequenos projetos. Dessa maneira, outra forma de se criar a documentação é utilizar o que chamamos de *javadoc*. Para se criar uma documentação utilizando-se o *javadoc*, é preciso colocar a documentação dentro da seguinte estrutura `/**
 * documentação em javadoc */`. De forma a ilustrar como se criar uma documentação em *javadoc*, analise o Código 3.12 mostrado a seguir. Repare que as linhas 15 a 23 contêm um método totalmente artificial, criado apenas para ilustrar a ideia da documentação.

Código 3.12 | Documentação utilizando *tags* do *javadoc* em um método qualquer

```

1  /**
2   * Colocar a descrição do método aqui. Este é um exemplo de método
3   * com dois parâmetros, com retorno e que pode lançar exceções.
4   * @param arg1 colocar a descrição do parâmetro arg1.
5   * @param arg2 colocar a descrição do parâmetro arg2.
6   * @return colocar a descrição do valor que é retornado.
7   * @throws SQLException
8   * @throws IOException
9   * @see java.lang.System
10  * @see #exemploMetodoLancaMultiExcecoes
11  * @since 1.0
12  * @version 1.0
13  * @author Jesimar da Silva Arantes
14 */
15 public double exemploMetodoCompleto(int arg1, float arg2)
16     throws SQLException, IOException{
17     if (arg1 < 0) {
18         throw new SQLException()
19     } else if (arg2 < 0) {
20         throw new IOException();
21     }
22     return 0.0;
23 }

```

0

Ver anotações

Fonte: elaborado pelo autor.

No Código 3.12, podemos perceber que toda a documentação foi colocada dentro da estrutura **`/** documentação */`**. Essa estrutura caracteriza uma documentação mais robusta e não apenas um simples comentário. Inicialmente, devemos colocar uma breve descrição do que o método faz, em seguida, podemos acrescentar algumas *tags* de forma a enriquecer a nossa documentação. Nesse exemplo, foram utilizadas *tags* como `@param`, `@return`, `@throws`, `@see`, `@since`, `@version` e `@author`. Analise o Quadro 3.7 para entender para que serve cada uma das *tags* comentadas. É importante mencionarmos também que esse tipo de *tag* pode ser utilizado na classe, no método, no construtor, nos atributos e nas interfaces.

Quadro 3.7 | Algumas das *tags* disponíveis para construção de documentação em Javadoc

Tags	Descrição
@author	Nome do autor do código.
@param	Descreve o parâmetro recebido pelo método ou construtor.
@return	Descreve o que será retornado pelo método.
@throws	Descreve a exceção ou o erro lançado pelo método ou construtor.
@see	Documenta o código criando um link com outra classe ou método.
@since	Documenta a partir de que versão o método foi adicionado.
@version	Documenta a versão da classe ou do método.
@deprecated	Marca o método como descontinuado ou obsoleto.

Fonte: elaborado pelo autor.

Até agora, essa documentação é parecida com a documentação baseada em comentários, no entanto, contém alguns marcadores. O interessante é que existe uma ferramenta no Java que permite que você gere um código HTML contendo toda a documentação do seu projeto Java. A maioria dos IDEs gera esse HTML apertando-se apenas um botão. Pesquise como fazer isso com a sua IDE de preferência. No Netbeans, basta você ir no menu **Executar** e, em seguida, ir em **Gerar Javadoc (Nome do Projeto)**. Após esses passos, na pasta *dist* será criada uma subpasta chamada *javadoc*, que conterá diversos arquivos; abra o arquivo *index.html* em algum navegador de internet e pronto. A partir disso, você poderá acessar toda a documentação do seu código com uma excelente organização estrutural. A Figura 3.5 nos mostra o resultado gerado pela ferramenta *javadoc*.

Figura 3.5 | Documentação gerada utilizando-se o *javadoc* do Java

The screenshot shows a Javadoc-generated HTML page with the following structure:

- Constructor Summary** (highlighted in orange)
- Constructors** (highlighted in orange)
- Constructor and Description**
- ExemploGerarJavadoc()**
Este é um exemplo de construtor da classe.
- Method Summary**
- All Methods** (highlighted in orange)
- Static Methods**
- Instance Methods**
- Concrete Methods**
- Deprecated Methods**
- Modifier and Type**
- Method and Description**
- void exemploMetodoDepreciado()**
Deprecated.
desde a versão 1.0
- void exemploMetodo()**
Este é um exemplo de método sem parâmetros e sem retorno.
- void exemploMetodoComParametros(char valor1, int valor2, float valor3)**
Este é um exemplo de método com parâmetros e sem retorno.

0

Ver anotações

Fonte: captura de tela do *javadoc* elaborada pelo autor.

Uma forma de aprender como documentar os seus projetos é observar como são feitas as documentações nas classes nativas do Java. Dessa maneira, entre nas classes Math e System, por exemplo, e observe como elas foram documentadas.

DICA

Caro aluno, pegue alguns dos códigos em Java que você desenvolveu; em seguida, escreva uma documentação utilizando o que você aprendeu sobre *javadoc*; gere a documentação em formato HTML; analise-a em um navegador de internet; faça diferentes alterações na documentação e analise a saída.

COMANDO *FOR EACH*

Na Seção 2.2 da Unidade 2, foram vistos alguns laços de repetição, como o *for*, *while* e *do-while*. A linguagem Java suporta outro tipo de comando que controla a repetição, chamado *for each* (para cada). O comando *for each* é utilizado, geralmente, para se fazer a iteração em coleções como vetores, matrizes, listas, entre outros.

Analise o Quadro 3.8 a seguir.

Quadro 3.8 | Síntese dos comandos de repetição *for* e *for each*

	Comando: for	Comando: for each
Sintaxe	<pre>for (Inicializ; ExpLóg; Inc) { SeqDeComandos; }</pre>	<pre>for (Tipo elem : ColeçãoIteravel) { SeqDeComandos; }</pre>
Exemplo	<pre>int soma = 0; for (int i = 1; i < 6; i++) { soma += i; }</pre>	<pre>int colecao[] = {1, 2, 3, 4, 5}; int soma = 0; for (int elem : colecao) { soma += elem; }</pre>

Fonte: elaborado pelo autor.

O Quadro 3.8 apresenta, inicialmente, a forma geral da sintaxe do comando *for each* em contraste com o comando *for* (já estudado); em seguida, vemos um simples exemplo que soma os valores de 1 a 5 utilizando-se o comando *for* e o comando *for each*. No exemplo do Quadro 3.8, cada elemento da coleção (vetor) é acessado pela variável *elem*, que é acumulada na variável *soma*.

A seguir, no Código 3.13, podemos ver duas aplicações diferentes utilizando-se o comando *for each*. Implemente esses métodos e analise a saída em cada um deles para compreendê-los melhor.

Código 3.13 | Exemplos de utilização do comando *for each* em Java

```

1 public static void forEachChar(){
2     String livro = "Linguagem Orientada a Objetos";
3     for (char letra : livro.toCharArray()){
4         System.out.println("character: " + letra);
5     }
6 }
```

```
1 public static void forEachString(){
2     String livro[] = {"Linguagem", "Orientada", "a", "Objetos"};
3     for (String nome : livro){
4         System.out.println("nome: " + nome);
5     }
6 }
```

0

[Ver anotações](#)

Fonte: elaborado pelo autor.

Agora, vamos imaginar que temos a intenção de criar um método em Java que some dois números e retorne o resultado. No entanto, precisamos criar também um método que some três números e retorne a soma, bem como criar um método que some quatro números e retorne a soma. O Código 3.14 nos apresenta esses três métodos descritos.

Código 3.14 | Métodos em Java que efetuam a soma de 2, 3 e 4 argumentos inteiros

```
1 public static int soma(int a, int b){
2     return a + b;
3 }
4 public static int soma(int a, int b, int c){
5     return a + b + c;
6 }
7 public static int soma(int a, int b, int c, int d){
8     return a + b + c + d;
9 }
```

Fonte: elaborado pelo autor.

ARGUMENTOS VARIÁVEIS (VARARGS)

Imagine, agora, que queremos criar um método que some os números inteiros que forem passados como argumentos. Se seguirmos a lógica do código 3.14, não conseguiremos nunca, pois teremos de criar infinitos métodos, acrescentando, em cada um deles, um novo argumento. A linguagem Java, por sua vez, permite-nos fazer isso de forma bastante simples, utilizando a ideia de argumentos variáveis ou *varargs*. O Quadro 3.9 nos mostra a sintaxe básica em dois exemplos de métodos genéricos que utilizam *varargs*. No primeiro exemplo, o método recebe apenas um

argumento do tipo *varargs*, que é caracterizado por três pontos (...) depois do tipo de argumento. Já no segundo exemplo, o método recebe um argumento normal seguido de um argumento do tipo *varargs*.

0

Quadro 3.9 | Sintaxe de métodos que receberem argumentos variáveis (*varargs*)

Exemplos de métodos com *varargs*

```
1 modif_acesso tipo_retorno nomeMetodo(TipoArg... args){  
2     SeqDeComandos;  
3 }
```

```
1 modif_acesso tipo_retorno nomeMetodo(TipoArg arg, TipoArg... args){  
2     SeqDeComandos;  
3 }
```

Fonte: elaborado pelo autor.

De forma a compreender melhor como funcionam os argumentos variáveis, considere o Código 3.15, que nos mostra como podemos criar um método que soma quaisquer quantidades de argumentos passados.

Código 3.15 | Método capaz de somar todos os valores especificados como argumentos

Ver anotações

```
1 public class ExemploVarargs {  
2     public static void main(String[] args) {  
3         System.out.println(soma());  
4         System.out.println(soma(6));  
5         System.out.println(soma(1, 6));  
6         System.out.println(soma(3, 5, 4));  
7         System.out.println(soma(2, 7, 4, 8));  
8         System.out.println(soma(5, 6, 9, 1, 8));  
9     }  
10    public static int soma(int... args){  
11        int valorDaSoma = 0;  
12        for (int valor : args){  
13            valorDaSoma += valor;  
14        }  
15        return valorDaSoma;  
16    }  
17 }
```

0

Ver anotações

Fonte: elaborado pelo autor.

Nas linhas 10 a 16 do Código 3.15, temos a definição do método *soma* que recebe uma quantidade variável de argumentos. Nesse código, foi utilizado o comando *for each* para se fazer a iteração entre cada um dos argumentos recebidos como parâmetros e acumular na variável *valorDaSoma*. Já o método *main*, nas linhas 2 a 9, é responsável pelas chamadas do método *soma* com diferentes quantidades de argumentos.

Aluno, frente a isso, implemente o código acima e analise a saída da execução. O link para visualização da execução do site Java Tutor está no tópico Referências.

Algumas regras devem ser seguidas ao se utilizar *varargs*. Apenas o último argumento passado pode ser do tipo *varargs*. Um parâmetro *varargs* pode receber **zero, muitos ou um array** de parâmetros, e os argumentos variáveis podem ser utilizados tanto para métodos quanto para construtores.

REFLITA

Reflita sobre o porquê de apenas o último argumento passado para um método ou construtor poder ser do tipo *varargs*. Quais seriam as dificuldades em se permitir que o primeiro argumento seja do tipo *varargs*?

Reflita sobre por que não se pode ter dois argumentos do tipo *varargs* em um mesmo método ou construtor. Quais seriam as dificuldades em se permitir dois argumentos *varargs*?

ENUMERAÇÃO

A linguagem Java dá suporte a um tipo especial de classe chamada enumeração ou, simplesmente, **Enum**. A sua declaração utiliza a palavra-reservada *enum*. Uma classe do tipo Enum é utilizada para se fazer a organização de um conjunto de constantes, que, em geral, são *static* e *final*. Assim, as constantes são declaradas com letras maiúsculas e são separadas simplesmente por vírgula. Analise o Código 3.16 mostrado a seguir.

Código 3.16 | Exemplos de declarações de enumerações em Java

```
1 public enum DiaDaSemana {  
2     DOM, SEG, TER, QUA, QUI, SEX, SAB  
3 }
```

```
1 public enum Mes {  
2     JAN, FEV, MAR, ABR, MAI, JUN, JUL, AGO, SET, OUT, NOV, DEZ  
3 }
```

```
1 public enum NivelDeDificuldade {  
2     FACIL, MEDIO, DIFICIL  
3 }
```

Fonte: elaborado pelo autor.

No Código 3.16 mostrado acima, a ideia é que cada um dos tipos *enum* seja codificado em um arquivo *.java* separado. No entanto, isso não é uma regra, pois se pode ter tipos *enum* dentro de classes. Vamos analisar a enumeração

DiaDaSemana: repare que temos, ao todo, sete constantes, que representam os dias da semana, de domingo a sábado. Em muitos códigos, a definição desse tipo de constante auxilia na legibilidade, ao contrário de números de 0 a 6 para a representação dos dias da semana. Analise, agora, a enumeração

NivelDeDificuldade mostrado acima, que define três constantes para o nível fácil, médio e difícil. Nesse caso, também poderíamos ter utilizado alguma outra representação, porém a grande maioria das demais representações perde em legibilidade para a enumeração. A ideia é utilizar enumeração sempre que houver valores que não podem ser trocados, como dias da semana, nomes dos meses, nomes de planetas, cartas do baralho, cores, entre outros.

EXEMPLIFICANDO

Analise o Código 3.17 a seguir que nos mostra como criar um objeto do tipo enumeração. Ao observar esse código, perceba que na linha 3 a criação de uma enumeração é bem simples. Inicialmente, temos o tipo do *enum* (*DiaDaSemana*) seguido do nome da variável (*dia*) e, então, a constante que será atribuída (*DiaDaSemana.TER*). Da linha 4 a 20, temos um *switch-case*, que, baseado no valor de entrada, imprime o dia da semana equivalente.

Diante disso, implemente esse código e faça algumas alterações na variável *dia* e analise a saída impressa; repare o quanto se ganha em termos de legibilidade de código ao se utilizar a enumeração.

Código 3.17 | Demonstração de uma simples aplicação com enumeração

o

Ver anotações

```

1  public class Main {
2
3      public static void main(String[] args) {
4          DiaDaSemana dia = DiaDaSemana.TER;
5
6          switch (dia) {
7
8              case DOM:
9                  System.out.println("Domingo"); break;
10             case SEG:
11                 System.out.println("Segunda"); break;
12             case TER:
13                 System.out.println("Terça"); break;
14             case QUA:
15                 System.out.println("Quarta"); break;
16             case QUI:
17                 System.out.println("Quinta"); break;
18             case SEX:
19                 System.out.println("Sexta"); break;
20             case SAB:
21                 System.out.println("Sábado"); break;
22         }
23     }
24 }
```

0

Ver anotações

Fonte: elaborado pelo autor.

INTERFACES

Nas seções anteriores, estudamos a herança e as classes abstratas. A linguagem Java permite que uma classe herde de apenas uma classe, ou seja, não permite heranças múltiplas, e para sanar essa “limitação”, o Java dá suporte ao conceito de interfaces. Uma interface estabelece um contrato que deve ser seguido ao desenvolver as classes que implementam as interfaces.

A linguagem Java possui algumas importantes interfaces implementadas. A seguir, apresentaremos uma breve descrição de três delas, que são: Comparable, Runnable e Serializable.

- **Comparable:** é utilizada para impor uma ordem nos objetos de uma determinada classe que a implementa. Para definir essa ordem, o método *compareTo* deve ser implementado, e nele serão estabelecidos os critérios da comparação. Um vetor ou uma lista de objetos que implementa essa interface

pode ser ordenada automaticamente por métodos de ordenação disponíveis no Java.

- **Runnable:** é utilizada para especificar alguma tarefa a ser realizada. Para se definir essa tarefa, o método *run* deve ser implementado, e nele será estabelecida a tarefa a ser executada. De forma geral, os objetos dessa classe são executados em paralelo, utilizando-se várias *threads*.
- **Serializable:** é utilizada para identificar classes em que os objetos podem ser gravados (também chamados de serializados) ou lidos (também chamados de desserializados) de algum dispositivo de armazenamento, como HD. A interface Serializable não possui métodos ou campos e serve apenas para identificar a semântica de ser serializável.

ASSIMILE

O Código 3.18 nos mostra alguns trechos das interfaces Comparable, Runnable e Serializable. É importante destacarmos aqui que a declaração de uma interface utiliza a palavra-reservada *interface*. De forma geral, as interfaces possuem apenas as assinaturas dos métodos e podem também conter atributos. A classe que implementar a interface é que deverá sobrescrever os métodos. A interface Comparable é genérica, pois recebe o tipo da classe ao utilizar o operador *<>*. A ideia de classes e interfaces genéricas não será explorada neste livro, assim, esses detalhes de implementação deverão ser abstraídos pelo aluno. A interface Comparable possui apenas um método, que é o *compareTo*, e a interface Runnable possui apenas um método, chamado *run()*. Repare que a interface Serializable não possui nada em seu corpo, dessa forma, essa interface foi utilizada como um recurso apenas semântico, já o polimorfismo auxilia nos métodos que serão serializáveis.

Código 3.18 | Trechos das interfaces Comparable, Runnable e Serializable

```
1 public interface Comparable<T extends Object> {  
2     public int compareTo(T o);  
3 }
```

```
1 public interface Runnable {  
2     public void run();  
3 }
```

```
1 public interface Serializable {  
2 }
```

Fonte: elaborado pelo autor.

O Código 3.19 ilustra um exemplo em que temos uma classe chamada Livro que implementa a interface *Comparable*. Uma classe implementa uma interface utilizando, para isso, a palavra-reservada *implements*. Aqui, é importante destacarmos que foi colocado entre os símbolos < e > o nome da classe. Esse procedimento poderia ter sido omitido se quiséssemos, mas ele simplifica o processo, evitando-se um *casting* (conversão de objetos).

0

Ver anotações

Código 3.19 | Exemplo de classe que implementa a interface Comparable do Java

```
1  public class Livro implements Comparable<Livro> {
2      public String nome;
3      public double custo;
4      public Livro(String nome, double custo) {
5          this.nome = nome;
6          this.custo = custo;
7      }
8      @Override
9      public int compareTo(Livro livro) {
10         if (Math.abs(custo - livro.custo) < 0.001) {
11             return 0;
12         } else if (custo > livro.custo) {
13             return 1;
14         } else {
15             return -1;
16         }
17     }
18 }
19 public class Main {
20     public static void main(String[] args) {
21         Livro l1 = new Livro("Alice no País das Maravilhas", 38);
22         Livro l2 = new Livro("O Senhor dos Anéis", 44);
23         Livro l3 = new Livro("Harry Potter", 38);
24         int compL1L2 = l1.compareTo(l2);
25         int compL1L3 = l1.compareTo(l3);
26         int compL2L3 = l2.compareTo(l3);
27         System.out.println("Comparação l1 e l2: " + compL1L2);
28         System.out.println("Comparação l1 e l3: " + compL1L3);
29         System.out.println("Comparação l2 e l3: " + compL2L3);
30     }
31 }
```

0

Ver anotações

Fonte: elaborado pelo autor.

O ponto principal a ser destacado no Código 3.19 é a sobrescrita do método *compareTo* (mostrado nas linhas 8 a 17), que recebe um objeto do tipo *Livro* como argumento. Esse método compara o preço de dois livros, e se a diferença de preço for menor que um determinado valor limite (0.001), então, o resultado 0 será

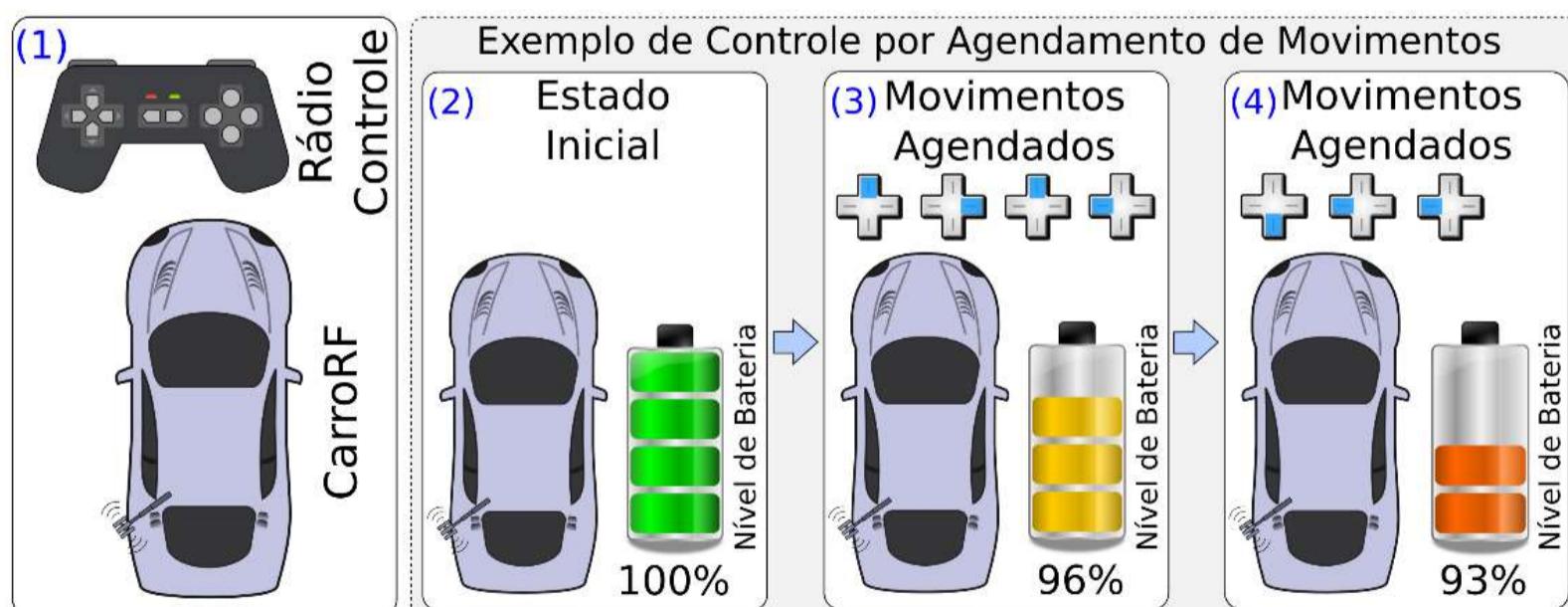
retornado, indicando que os dois livros têm o mesmo preço. Essa comparação foi feita dessa forma pois a comparação de valores de ponto flutuante tem algumas limitações no computador. Se o preço do livro corrente (atual) for maior do que o preço do livro passado como argumento, então, o resultado 1 será retornado, caso contrário, se o preço do livro corrente for menor do que o preço do livro passado como argumento, o resultado -1 será retornado. Nas linhas 19 a 31, temos a criação de uma classe principal para testar o código. Repare que três objetos do tipo livro foram criados e, em seguida, três testes foram realizados. Caro aluno, reflita sobre qual será a saída do código acima e o que significam esses valores.

É imprescindível que você comprehenda que, ao implementar a interface *Comparable* e sobrescrever o método *compareTo*, você poderá fazer comparações diretas entre dois livros. Nesse exemplo, as comparações foram baseadas no preço, mas poderiam ter sido baseadas em outros critérios, como, por exemplo, o nome do livro. Frente a isso, implemente o Código 3.19 e tente adaptá-lo para fazer comparações baseadas no nome, em que o nome que vem antes, de acordo com o alfabeto, deve ser menor do que os nomes que vêm depois.

Agora, considere este novo exemplo em que temos um carrinho de brinquedo movido à bateria e que é controlado por radiofrequência (RF). Vamos imaginar que, para que esse carrinho se move, devemos agendar um conjunto de movimentos, e que após cada um dos movimentos executados, 1% da bateria foi gasto.

Inicialmente, o carrinho está com sua bateria totalmente carregada (100%). Dito isso, observe a Figura 3.6.

Figura 3.6 | Exemplo de aplicação que controla um carro rádio controlado



Fonte: elaborada pelo autor.

A Figura 3.6 nos mostra, no quadro (1), como seria esse controle e o carro controlado por RF. Em seguida, no quadro (2), percebemos o estado inicial do carro com 100% da bateria. Posteriormente, no quadro (3), temos um exemplo em que quatro movimentos foram agendados, que foram: cima, direita, cima e esquerda. Após a execução desses quatro movimentos, a bateria caiu para 96%. Lembre-se, a bateria cai 1% a cada movimento executado, conforme as regras descritas acima. Então, no quadro (4), temos mais três movimentos agendados, que foram: baixo, esquerda e esquerda, e após a execução desses três movimentos, o nível de bateria caiu para 93%. Por meio desse exemplo lúdico, podemos explorar uma série de conceitos em Java. Dessa maneira, analise o Código 3.20 que ilustra esse exemplo descrito.

o

Ver anotações

Código 3.20 | Exemplo de classe que implementa a interface Runnable do Java

0

[Ver anotações](#)

```
1 import java.util.Arrays;
2 public class CarroRF implements Runnable {
3     private int nivelBateria;
4     private int numMovimentos;
5     public CarroRF() {
6         this.nivelBateria = 100;
7         this.numMovimentos = 0;
8     }
9     public void agendarMovimentos(Direcao... direcoesMovimentos) {
10        String msg = String.format("Lista movimentos agendados:%s",
11                               Arrays.toString(direcoesMovimentos));
12        System.out.println(msg);
13        numMovimentos += direcoesMovimentos.length;
14    }
15    @Override
16    public void run() {
17        System.out.println("Executando movimentos ...");
18        //execução da movimentação do carro (abstrair)
19        nivelBateria -= numMovimentos;
20        numMovimentos = 0;
21    }
22    @Override
23    public String toString() {
24        return String.format("Nível da Bateria: %d%n" +
25                               "Movimentos a Executar: %d",
26                               nivelBateria, numMovimentos);
27    }
28 }
29
30 public enum Direcao {
31     CIMA, BAIXO, ESQUERDA, DIREITA;
32 }
33
34 import static code.unidade3.secao2.ex3.Direcao.*;
35 public class Main {
36     public static void main(String[] args) {
37         CarroRF carro = new CarroRF();
38         carro.agendarMovimentos(CIMA, DIREITA, CIMA, ESQUERDA);
39         System.out.println(carro);
40         carro.run();
```

```
41     System.out.println(carro);
42     carro.agendarMovimentos(BAIXO, ESQUERDA, ESQUERDA);
43     System.out.println(carro);
44     carro.run();
45     System.out.println(carro);
46 }
47 }
```

0

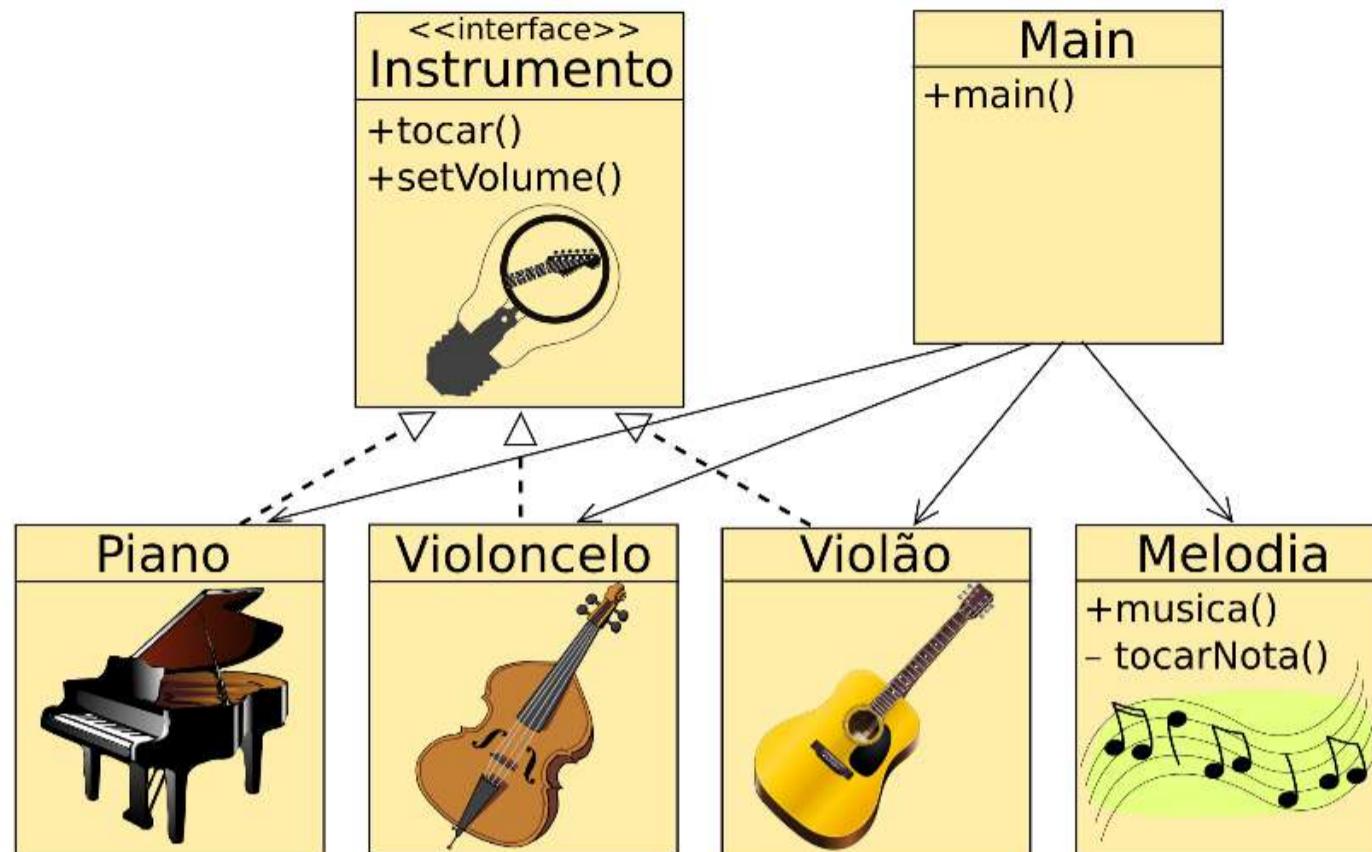
[Ver anotações](#)

Fonte: elaborado pelo autor.

No Código 3.20, a primeira classe apresentada chama-se CarroRF, que modela o carro controlado por radiofrequência e implementa a interface Runnable (executável). Nas linhas 3 a 9, temos dois atributos que modelam o nível da bateria e o número de movimentos agendados. Nas linhas 9 a 14, temos o método agendarMovimentos, que recebe uma *varargs* de direções possíveis. Nas linhas 15 a 21, temos o método *run*, que deve ser sobreescrito devido à implementação da interface Runnable. Como esse exemplo é apenas didático, não foi feita nenhuma movimentação do carro, contudo, se fosse implementado o método, este seria feito ali. Dessa maneira, foi dado destaque apenas ao nível corrente da bateria e ao número de movimentos agendados. Nas linhas 30 a 32, um tipo de enumeração foi criado, contendo as quatro direções em que o carro pode ser movimentado. Por fim, nas linhas 34 a 47, temos uma classe para testar a aplicação. Frente a isso, implemente esse código, faça algumas alterações nas direções movidas do carro e analise com calma a saída impressa.

Vamos, agora, ilustrar um exemplo em que criamos a nossa própria interface. Assim, considere que estamos criando uma aplicação para tocar música, que poderá ser executada por um conjunto de instrumentos diferentes. Nesse exemplo, os instrumentos suportados são: piano, violoncelo e violão. A música tocada se baseará em alguma melodia que será passada para cada um dos instrumentos. Analise a Figura 3.7, em que temos quatro classes e uma interface.

Figura 3.7 | Exemplo de aplicação que toca música a partir de diferentes instrumentos



Fonte: elaborada pelo autor.

Analise, agora, o Código 3.21, que foi construído a partir da descrição acima e da Figura 3.7.

Código 3.21 | Exemplo que toca música e criação da interface Instrumento

```
1  public interface Instrumento {  
2      public void tocar(Melodia melodia);  
3      public void setVolume(int volume);  
4  }  
5  
6  import javax.sound.midi.MidiChannel;  
7  import javax.sound.midi.MidiSystem;  
8  import javax.sound.midi.MidiUnavailableException;  
9  import javax.sound.midi.Synthesizer;  
10 public class Piano implements Instrumento {  
11     private int volume;  
12     private final Synthesizer sintetizador;  
13     public Piano() throws MidiUnavailableException {  
14         sintetizador = MidiSystem.getSynthesizer();  
15     }  
16     @Override  
17     public void tocar(Melodia melodia) {  
18         try {  
19             sintetizador.open();  
20             MidiChannel canal = sintetizador.getChannels()[0];  
21             canal.programChange(1); //código ID do piano  
22             melodia.musica(canal, volume);  
23             sintetizador.close();  
24         } catch (MidiUnavailableException |  
25                 InterruptedException ex) {  
26             System.out.println(ex);  
27             sintetizador.close();  
28         }  
29     }  
30     @Override  
31     public void setVolume(int volume) {  
32         this.volume = volume;  
33     }  
34 }  
35  
36 public class Violoncelo implements Instrumento {  
37     //o código foi omitido, pois é similar a classe Piano  
38     //acesse https://github.com/jesimar/Livro-POO-Java  
39 }  
40
```

Ver anotações

0

```
41 public class Violao implements Instrumento {
42     //o código foi omitido, pois é similar a classe Piano
43     //acesse https://github.com/jesimar/Livro-POO-Java
44 }
45
46 import javax.sound.midi.MidiChannel;
47
48 public class Melodia {
49     private final static int DO = 60;
50     private final static int RE = 62;
51     private final static int MI = 64;
52     private final static int FA = 65;
53     private final static int SOL = 67;
54     private final static int SILENCIO = 0;
55     private final int UM_TEMPO;
56     private final int DOIS_TEMPOS;
57     public Melodia(double velocidade) {
58         UM_TEMPO = (int)(250/velocidade);
59         DOIS_TEMPOS = (int)(500/velocidade);
60     }
61     public void musica(MidiChannel canal, int volume)
62         throws InterruptedException{
63         tocarNota(canal, DO, UM_TEMPO, volume);
64         tocarNota(canal, RE, UM_TEMPO, volume);
65         tocarNota(canal, MI, UM_TEMPO, volume);
66         tocarNota(canal, FA, DOIS_TEMPOS, volume);
67         tocarNota(canal, FA, UM_TEMPO, volume);
68         tocarNota(canal, FA, UM_TEMPO, volume);
69         tocarNota(canal, SILENCIO, UM_TEMPO, volume);
70     }
71     private static void tocarNota(MidiChannel canal, int nota,
72         int tempo, int volume) throws InterruptedException {
73         canal.noteOn(nota, volume);
74         Thread.sleep(tempo);
75         canal.noteOff(nota, volume);
76     }
77
78 import javax.sound.midi.MidiUnavailableException;
79
80 public class Main {
81     public static void main(String[] args)
82         throws MidiUnavailableException {
```

Ver anotações

0

```
82     System.out.println("Tocando melodia piano vel. normal");
83     Melodia melodiaNormal = new Melodia(1.00);
84     Piano piano = new Piano();
85     piano.setVolume(75);
86     piano.tocar(melodiaNormal);
87     System.out.println("Tocando melodia celo vel. rápida");
88     Melodia melodiaRapida = new Melodia(1.25);
89     Violoncelo violoncelo = new Violoncelo();
90     violoncelo.setVolume(100);
91     violoncelo.tocar(melodiaRapida);
92     System.out.println("Tocando melodia violão vel. lenta");
93     Melodia melodiaLenta = new Melodia(0.75);
94     Violao violao = new Violao();
95     violao.setVolume(50);
96     violao.tocar(melodiaLenta);
97 }
98 }
```

0

Ver anotações

Fonte: elaborado pelo autor.

No Código 3.21, nas linhas 1 a 4, temos a interface Instrumento, que possui a assinatura de dois métodos, que são: tocar e setVolume. Nas linhas 6 a 34, temos a classe Piano, que implementa a interface Instrumento criada. Você deve ter reparado que um conjunto de classes do pacote *javax.sound.midi* foi utilizado. Não será explicado em detalhes como funcionam as classes desse pacote, mas, por hora, você apenas precisa saber que elas manipulam dados para tocar notas musicais em um determinado canal de saída. Assim, para que esse código funcione, o computador que executá-lo deverá ter caixas de som para saída do áudio. Nas linhas 36 a 44, temos as classes Violoncelo e Violao, que foram omitidas por serem semelhantes à classe Piano. Nas linhas 46 a 76, temos a classe Melodia, que define a música a ser tocada fazendo uso das notas musicais. Por fim, nas linhas 78 a 98, temos a classe responsável por organizar a execução da aplicação. Diante disso, analise com calma as linhas desse código e tente compreender a sua lógica.

Ao desenvolvermos um código, às vezes, ficamos confusos sobre qual recurso utilizarmos: classe abstrata ou interface, mas, quanto a isso, fique tranquilo, isso é natural, às vezes, podemos utilizar um ou outro, e ambos funcionarão, desde que utilizados da forma correta. O Quadro 3.10 sintetiza de forma bastante objetiva algumas das características entre as classes abstratas e as interfaces em Java. Analise com cautela esse quadro, mas não se preocupe em memorizar todas as informações, em geral, isso não é necessário.

Quadro 3.10 | Síntese de algumas características de classes abstratas e interfaces

Características	Classe Abstrata	Interface
Pode possuir construtores	Sim	Não
Pode possuir métodos concretos	Sim	Não
Pode possuir métodos abstratos	Sim	Sim
Pode possuir métodos final	Sim	Não
Pode possuir métodos não-final	Sim	Sim
Pode possuir métodos estáticos	Sim	Sim
Pode possuir atributos final	Sim	Sim

Características	Classe Abstrata	Interface
Pode possuir atributos não-final	Sim	Não
Pode possuir atributos estáticos	Sim	Sim
Pode possuir atributos não-estáticos	Sim	Não

Fonte: elaborado pelo autor.

O Quadro 3.11 a seguir apresenta uma síntese da comparação entre as classes abstratas e as interfaces. Caro aluno, analise este quadro.

Quadro 3.11 | Comparação entre as classes abstratas e interfaces em Java

Características	Classe Abstrata	Interface
Palavra-reservada utilizada.	Abstract	interface
Exemplo de declaração.	<code>public abstract class X</code>	<code>public interface X</code>
Exemplo de utilização.	<code>public class Y extends X</code>	<code>public class Y implements X</code>
Relação de herança.	Uma classe abstrata pode herdar apenas uma classe (abstrata ou não).	Uma interface pode herdar de múltiplas interfaces.
Relação de implementação.	Uma classe pode herdar apenas uma classe abstrata.	Uma classe pode implementar múltiplas interfaces.
Modificador de acesso dos métodos e atributos.	Pode ser público, <i>default</i> , protegido ou privado.	É público por padrão.
Modificador dos atributos.	Pode ser final, não final, estático ou não estático.	É final por padrão. É estático por padrão.
Instanciação.	Não permitida.	Não permitida.

0

Ver anotações

Fonte: elaborado pelo autor.

A linguagem Java ganha em aspectos de reusabilidade de código ao dar suporte à herança, classe abstrata e interface, e o polimorfismo também pode ser utilizado com interfaces. Dessa maneira, pode-se criar um objeto do tipo da interface com a instanciação do objeto do tipo da classe que implementa a interface.

SAIBA MAIS

Até agora, nada foi dito a respeito da identação de código. É importante que os códigos desenvolvidos sejam indentados corretamente, uma vez que colaboram muito na identificação de erros.

Dica: muitos Ambientes de Desenvolvimento Integrados (IDEs) têm recursos de autoidentação. No Netbeans, basta ir no menu **Código-Fonte** e, em seguida na opção **Formatar** (ou, então, no atalho *Alt+Shift+F*). No Eclipse, o atalho é *Ctrl+Shift+F*; já no IntelliJ IDEA, o atalho é *Ctrl+Alt+I*.

o

PESQUISE MAIS

O texto apresentou brevemente o conteúdo de enumeração em Java. Este assunto, apesar de simples, é muito importante. Dessa maneira, assista ao vídeo sugerido abaixo para entender melhor como aplicar na prática esse conteúdo.

LOIANE GRONER. **Curso de Java 53:** enumeradores (Enum). 2016.

LOIANE GRONER. **Curso de Java 54: enumeradores como classe (construtor e métodos).** 2016.

LOIANE GRONER. **Curso de Java 55:** Enum: métodos value e valueOf. 2016.

Ver anotações

Caro estudante, nesta seção você estudou os conteúdos relacionados à documentação com *javadoc*, ao comando *for each*, ao método com argumentos variáveis, à enumeração, a interfaces e à comparação entre classes abstratas e interfaces, bem como foram mostrados diversos exemplos desses conceitos em Java. Aprenderemos, na próxima seção, como construir aplicações que utilizam interfaces gráficas, avançando ainda mais no entendimento da linguagem Java.

REFERÊNCIAS

CURSO EM VÍDEO. **Curso de Java para iniciantes** – grátis, completo e com certificado. 2019. Disponível em: <https://bit.ly/35ykFEQ>. Acesso em: 20 ago. 2020.

ORACLE. **Interface Comparable<T>**. [s.d.]. Disponível em: <https://bit.ly/35Acisd>. Acesso em: 20 ago. 2020.

ORACLE. **Interface Runnable**. [s.d.]. Disponível em: <https://bit.ly/3mjUajh>. Acesso em: 20 ago. 2020.

ORACLE. **Interface Serializable**. [s.d.]. Disponível em: <https://bit.ly/2ZArCl5>. Acesso em: 20 ago. 2020.

PYTHON TUTOR. **Java Tutor - visualize Java code execution to learn Java online**. [s.d.]. Disponível em: <https://bit.ly/2FryN84>. Acesso em: 24 jul. 2020.

FOCO NO MERCADO DE TRABALHO

DEFINIÇÃO E USO DE INTERFACES

Jesimar da Silva Arantes

[Ver anotações](#)

IMPLEMENTAÇÃO DE NOVOS RECURSOS NO SIMULADOR DE ROBÔ

Documentação do código utilizando *javadoc*, declaração de constantes utilizando enumeração, descrição de utilização do código executado pelo terminal e execução de uma lista de comandos automaticamente com a utilização de *for each* e *varargs*.



Fonte: Shutterstock.

Deseja ouvir este material?

Áudio disponível no material digital.

SEM MEDO DE ERRAR

A *startup* em que você trabalha lhe passou diversas tarefas:

- Documentar o código utilizando *javadoc*.
- Declarar algumas constantes utilizando enumeração.
- Colocar uma descrição de utilização quando o código for executado pelo terminal.

- Executar uma lista de comandos automaticamente utilizando *for each* e *varargs*.

Como forma de iniciar o trabalho, você decidiu revisar o Código 3.11 para adicionar os recursos que lhe foram designados. Observando o seu código, você percebeu que um tipo Enum podia ser criado para agrupar as constantes FRENTE, ATRAS, ESQUERDA e DIREITA, e notou que, após isso, seu código ficou melhor estruturado. No Código 3.22, mostrado abaixo, nas linhas 1 a 3, temos a definição dessa enumeração, e nas linhas 6 a 22, temos o código que foi adaptado para receber a nova estrutura de dados.

0

[Ver anotações](#)

Em seguida, você decidiu imprimir algumas informações que dão uma descrição da aplicação caso especificado durante a execução por meio do terminal. Dessa maneira, você criou as opções: `--author`, `--version`, `--help`, `--commands` e `-move`, que podem ser vistas nas linhas 38 a 56 do Código 3.22.

Após isso, você decidiu trabalhar na opção `-move`, que modela as movimentações agendadas, usando uma lista de comandos passados pelo terminal. O método que manipula os movimentos agendados pode ser visto nas linhas 23 a 30. Repare que esse método utilizou a ideia de argumentos variados (*varargs*) e o comando *for each*, conforme sugerido.

O Código 3.22 destacou apenas as principais alterações feitas. O código completo pode ser acessado no GitHub do autor. Lá, você encontrará a documentação do código utilizando *javadoc*.

Caro aluno, faça as devidas implementações e execute o código para que possa entender melhor o seu funcionamento.

Código 3.22 | Modelagem das classes Orientacao, Robo e App

```
1 public enum Orientacao {  
2     FRENTE, ATRAS, ESQUERDA, DIREITA;  
3 }  
4  
5 public class Robo extends RoboIdeia {  
6     public void setOrientacao(char tecla) {  
7         if (tecla == 'w') {  
8             super.orientacao = Orientacao.FRENTE;  
9             moveY(5);  
10        } else if (tecla == 's') {  
11            super.orientacao = Orientacao.ATRAS;  
12            moveY(-5);  
13        } else if (tecla == 'a') {  
14            super.orientacao = Orientacao.ESQUERDA;  
15            moveX(-5);  
16        } else if (tecla == 'd') {  
17            super.orientacao = Orientacao.DIREITA;  
18            moveX(5);  
19        } else {  
20            throw new IllegalArgumentException("Arg não válido");  
21        }  
22    }  
23    public void movimentosAgendados(String... moves){  
24        for (String tecla : moves){  
25            if (!tecla.equals("--move")) {  
26                setOrientacao(tecla.charAt(0));  
27                printPos();  
28            }  
29        }  
30    }  
31    //foi omitido o código do restante da classe Robo.  
32 }  
33  
34 public class App {  
35     public static void main(String[] args) {  
36         if (args.length != 0){  
37             args[0] = args[0].toLowerCase();  
38             if (args[0].equals("--author")) {  
39                 System.out.println("Autor: Jesimar S. Arantes");  
40             } else if (args[0].equals("version")) {  
41                 System.out.println("Versão: 1.0");  
42             }  
43         }  
44     }  
45 }
```

0

Ver anotações

```
41         System.out.println("Versão: 1.0.0");
42     } else if (args[0].equals("--help")) {
43         System.out.println("Exec. Movimentos Agendados:");
44         System.out.println("\t--move w a w w s d d a");
45     } else if (args[0].equals("--commands")) {
46         System.out.println("Comandos suportados:");
47         System.out.println("\tw: move para cima");
48         System.out.println("\ta: move para esquerda");
49         System.out.println("\ts: move para baixo");
50         System.out.println("\td: move para direita");
51         System.out.println("\t0: sair da aplicação");
52     } else if (args[0].equals("--move")) {
53         Robo robo = new Robo();
54         System.out.println("Movimentos agendados: ");
55         robo.movimentosAgendados(args);
56     } else {
57         System.out.println("Argumento não válido");
58     }
59     System.exit(0);
60 }
61 //foi omitido o código do restante da aplicação.
62 }
63 }
```

Fonte: elaborado pelo autor.

DESENVOLVIMENTO DE INTERFACES GRÁFICAS NA LINGUAGEM JAVA

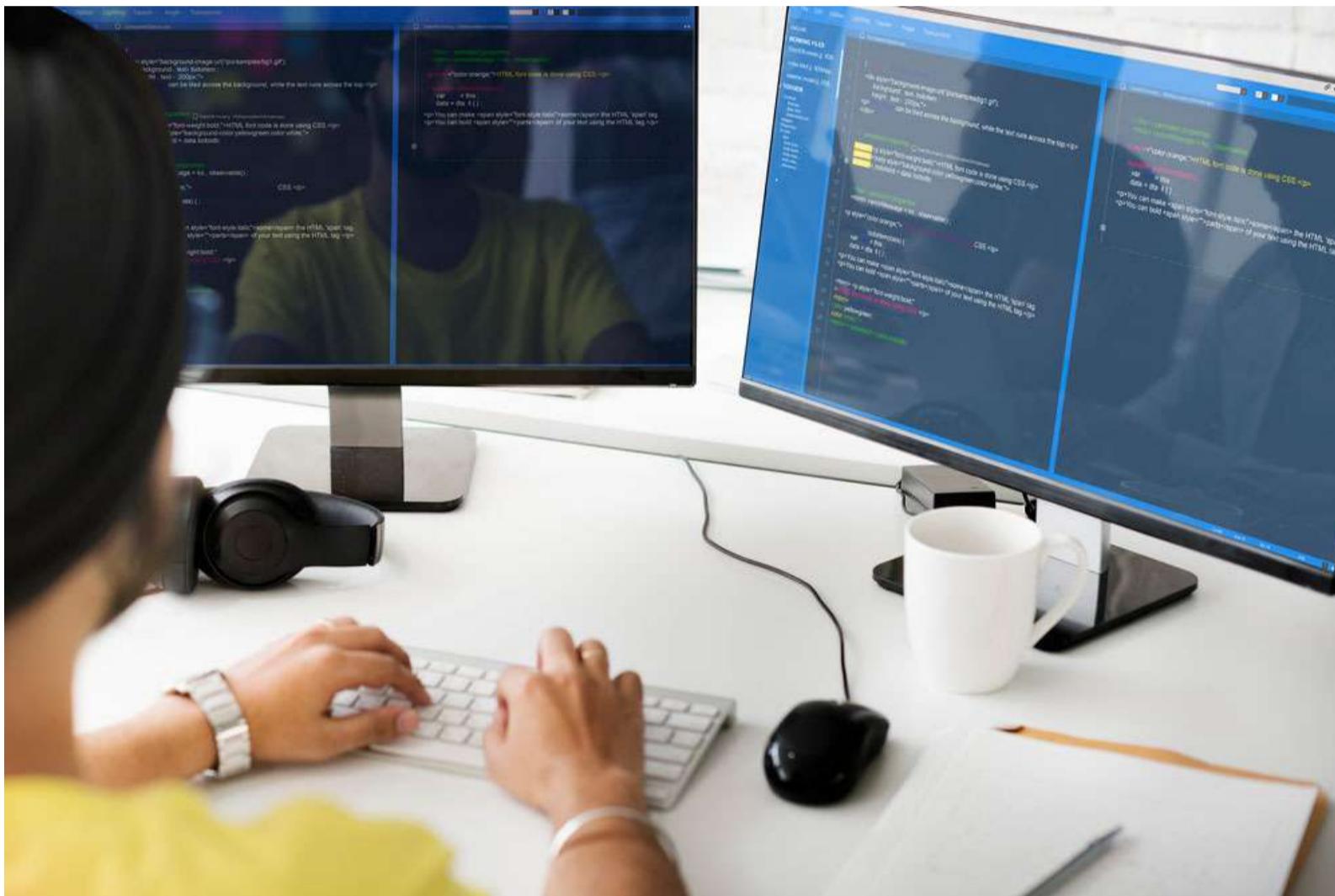
0

Jesimar da Silva Arantes

[Ver anotações](#)

CONSTRUÇÃO DE APLICAÇÕES GRÁFICAS

O JavaFX é uma biblioteca muito boa que a linguagem Java conta para a construção de aplicações gráficas.



Fonte: Shutterstock.

Deseja ouvir este material?

Áudio disponível no material digital.

PRATICAR PARA APRENDER

Caro estudante, bem-vindo à terceira seção da terceira unidade dos estudos sobre Linguagem Orientada a Objetos. Qual desenvolvedor nunca quis construir interfaces gráficas para suas aplicações? A maioria dos desenvolvedores de hoje em dia já desejou isso. Às vezes, construir uma aplicação que é executada apenas no terminal não basta diante do que se pretende fazer. Os jogos são bons exemplos disso, pois é muito complicado e pouco atrativo jogar jogos via linha de

comando. A linguagem Java conta com uma biblioteca muito boa para a construção de aplicações gráficas, o JavaFX, e nesta seção você terá a oportunidade de aprender a construir aplicações com essa biblioteca.

Como forma de contextualizar a sua aprendizagem, lembre-se de que você está desenvolvendo um simulador completo de um robô. O seu gestor revisou mais uma vez o seu código e lhe passou mais uma lista de tarefas que deve ser desenvolvida. Ele percebeu que, no seu código, não existe nenhuma interface gráfica que mostre o seu robô andando pela sala, logo, pediu a você que desenvolva a GUI para o robô por meio da biblioteca JavaFX.

0

[Ver anotações](#)

Diante desse desafio que lhe foi dado, como você criará essa interface gráfica? O que é GUI? O que é a biblioteca JavaFX? Como criar uma aplicação com JavaFX? Esta seção irá guiá-lo até as respostas para essas questões levantadas.

Muito bem, agora que você foi apresentado à sua nova situação-problema, estude o conteúdo desta seção e compreenda como construir aplicações gráficas com JavaFX. Ao aprender o conteúdo, um novo leque de oportunidades em termos de construção de aplicações de pequeno e grande porte se abrirá para você. E aí, vamos juntos compreender como utilizar essa biblioteca e, então, resolver esse desafio?

Bom estudo!

CONCEITO-CHAVE

A linguagem Java possui um suporte muito bom para a criação de Interfaces Gráficas de Usuário ou, em inglês, *Graphical User Interface* (GUI), e existem várias formas de se fazer a criação de interfaces gráficas em Java. Ao longo da evolução dessa linguagem, diversas bibliotecas gráficas foram criadas, como: *Abstract Window Toolkit* (AWT), *Swing*, *Standard Widget Toolkit* (SWT), *Apache Pivot*, *SwingX*, *JGoodies*, *QtJambi* e *JavaFX*. Este livro irá lhe mostrar como criar aplicações gráficas utilizando o JavaFX; a partir da versão do Java 8, a API do JavaFX é distribuída nativamente, ou seja, basta se ter a versão 8 ou superior que o JavaFX funcionará naturalmente, sem a necessidade de qualquer instalação.

DICA

Muitos alunos gostam de aprender mais de uma tecnologia/API, e existem muitos materiais na internet, como vídeos e tutoriais, a respeito das APIs Swing e AWT, que são boas bibliotecas, mas que serão descontinuadas na versão do Java 9. Diante disso, o mais aconselhável é estudar tecnologias mais novas, como o JavaFX.

■ JAVAFX

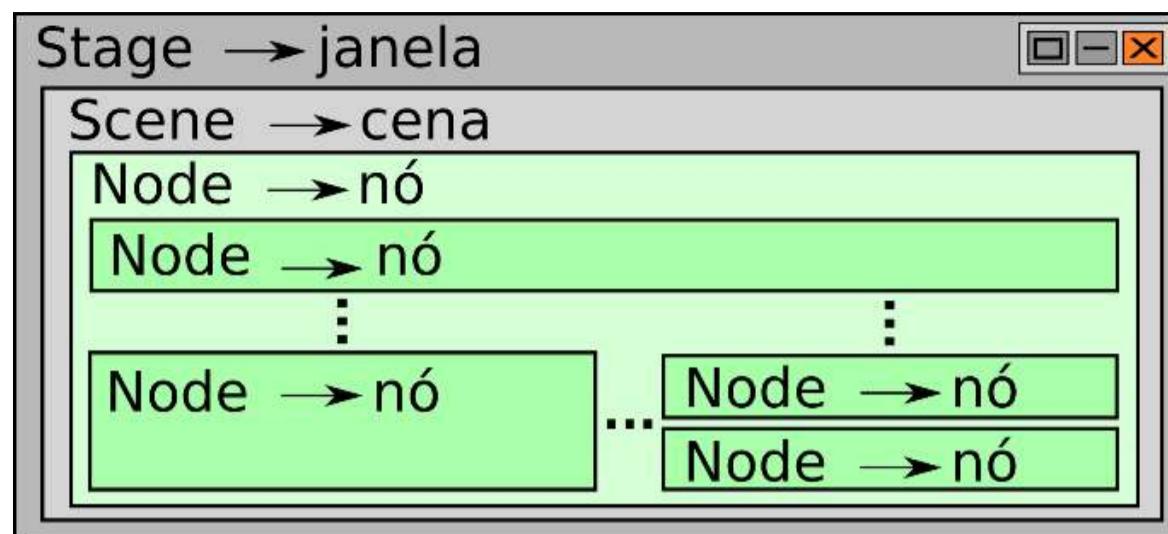
Antes de iniciarmos a codificação de aplicações com interfaces gráficas, é importante que você compreenda a lógica de como organizar uma tela utilizando JavaFX. Dessa maneira, observe a Figura 3.8, em que temos, inicialmente, uma janela (classe *Stage*). Essa janela é a parte mais externa da interface gráfica, em que ficam os botões de maximizar, minimizar, fechar e o título da janela. Dentro

dessa janela, temos uma cena (classe *Scene*) em que são definidas as dimensões da tela e colocamos os componentes que desejamos criar na interface gráfica. Por fim, no interior da cena, temos os nós (que herdam da classe *Node*), que nada mais são do que todos os componentes que desejamos colocar na tela. Dentro de alguns nós, podemos ter outros nós. Veremos mais sobre isso adiante.

o

[Ver anotações](#)

Figura 3.8 | Organização estrutural de uma aplicação que utiliza JavaFX



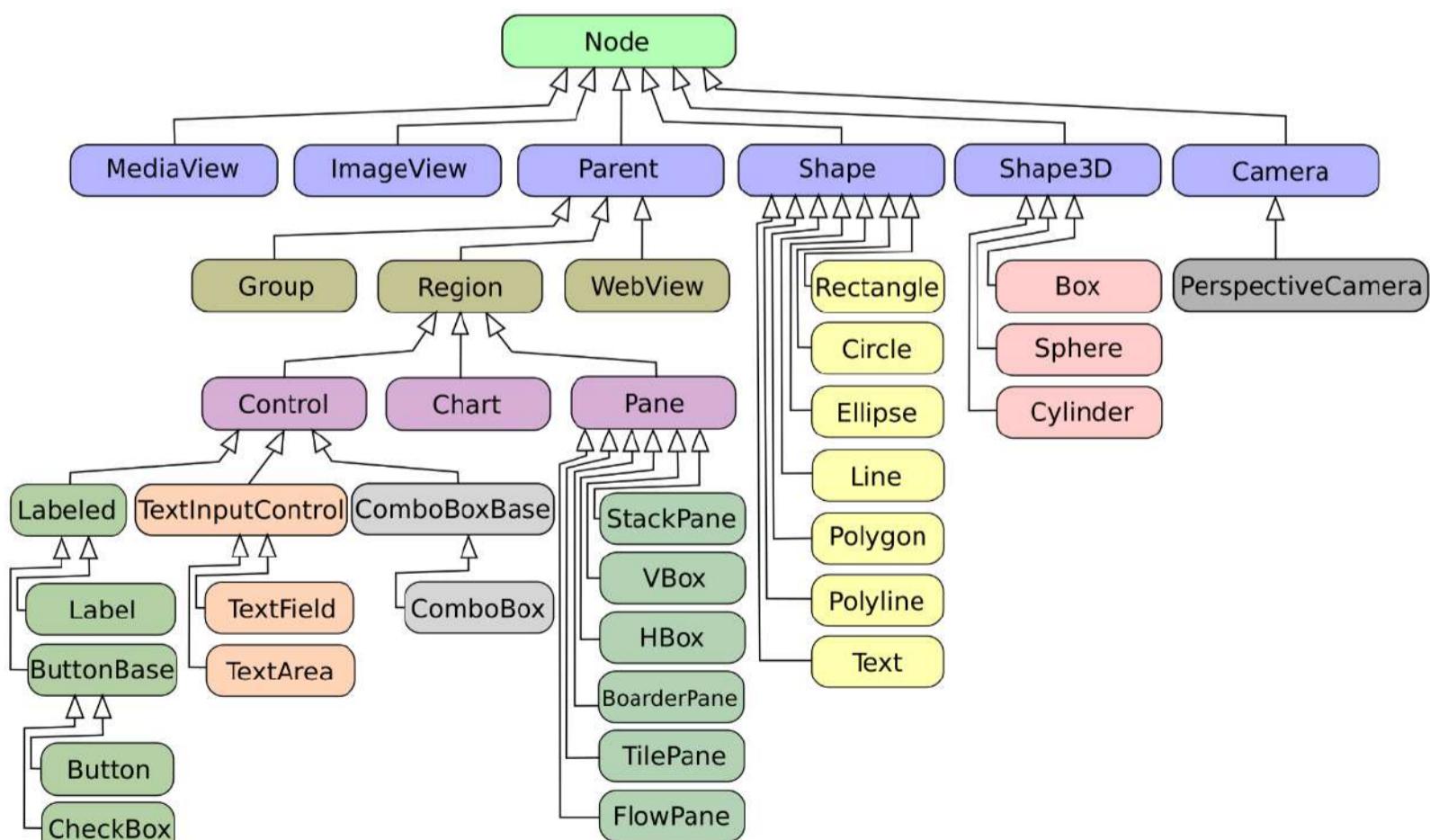
0

Ver anotações

Fonte: elaborada pelo autor.

Outro ponto muito importante é entender a organização hierárquica das classes que descendem a classe *Node*. A Figura 3.9 nos mostra uma pequena parte das classes que descendem de *Node*. Como exemplos de classe que herdam de *Node*, podemos destacar *ImageView*, *Parent* e *Shape*. A classe *Shape* possui diversas subclasses, como *Rectangle*, *Circle*, *Ellipse*, etc., e a classe *Parent* também possui diversas subclasses, como *Group*, *Region* e *WebView*. Neste momento, esperamos que você tenha apenas uma noção dessa organização; quando os exemplos do livro começarem a aparecer, não hesite em consultar esta figura novamente para compreender a aplicação.

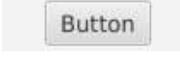
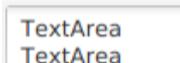
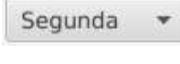
Figura 3.9 | Organização hierárquica das classes descendentes da classe Node



Fonte: elaborada pelo autor.

Antes de começarmos a implementação, apresentaremos, brevemente, alguns dos componentes que estão disponíveis no JavaFX. O Quadro 3.12 apresenta uma síntese de alguns componentes que estão no pacote *javafx.scene.control*.

Quadro 3.12 | Síntese de alguns componentes presentes no pacote *javafx.scene.control*

Nome do Componente	Classe do Componente	Descrição do Componente	Aspecto Visual do Componente
Botão	Button	Simples botão de controle.	
Rótulo	Label	Campo que permite exibir um texto não editável.	
Campo de Texto	TextField	Campo que permite a inserção de texto em uma única linha.	
Área de Texto	TextArea	Campo que permite a inserção de texto com múltiplas linhas.	
Botão de Opção	RadioButton	Seleção de opções mutuamente excludentes.	
Caixa de Seleção	CheckBox	Seleção de opções não mutuamente excludentes.	
Caixa de Combinação	ComboBox	Seleção de opções mutuamente excludentes em forma de lista.	

0

Ver anotações

Fonte: elaborado pelo autor.

A seguir, encontraremos exemplos dos quatro primeiros componentes, que são: botão (Button), rótulo (Label), campo de texto (TextField) e área de texto (TextArea). Veremos, então, que a utilização desses componentes é bem simples. Os

componentes botão de opção (`RadioButton`), caixa de seleção (`CheckBox`) e caixa de combinação (`ComboBox`) não serão mostrados neste livro, porém são bem simples e você pode encontrar exemplos de como utilizá-los na internet.

GERENCIAMENTO DE LAYOUT

A fim de construirmos uma tela gráfica, precisamos definir onde ficará cada um dos componentes na tela. Assim, será necessário especificar as coordenadas (x, y) de todos os componentes colocados na tela. No entanto, isso costuma dar muito trabalho, logo, é uma boa estratégia utilizar algum gerenciador de *layout*, que organiza de forma automática os componentes na tela conforme alguma estratégia. A biblioteca JavaFX possui diversas classes que realizam o gerenciamento de *layout*, mas neste livro, utilizaremos:

- **StackPane**: utiliza a estratégia de fazer o alinhamento baseado no empilhamento dos componentes.
- **VBox**: utiliza a estratégia de fazer o alinhamento dos componentes verticalmente.
- **HBox**: utiliza a estratégia de fazer o alinhamento dos componentes horizontalmente.

CRIAÇÃO DE INTERFACES GRÁFICAS (GUI)

Vamos, agora, criar o nosso primeiro código que faz uso de interfaces gráficas (GUI).

Alguns IDEs, como o Netbeans, geram automaticamente a aplicação base (*hello world*). Dessa forma, se você estiver no Netbeans, execute os passos descritos na Figura 3.10. Após executá-los, o Código 3.23 será gerado (foram feitas poucas modificações nos nomes das variáveis).

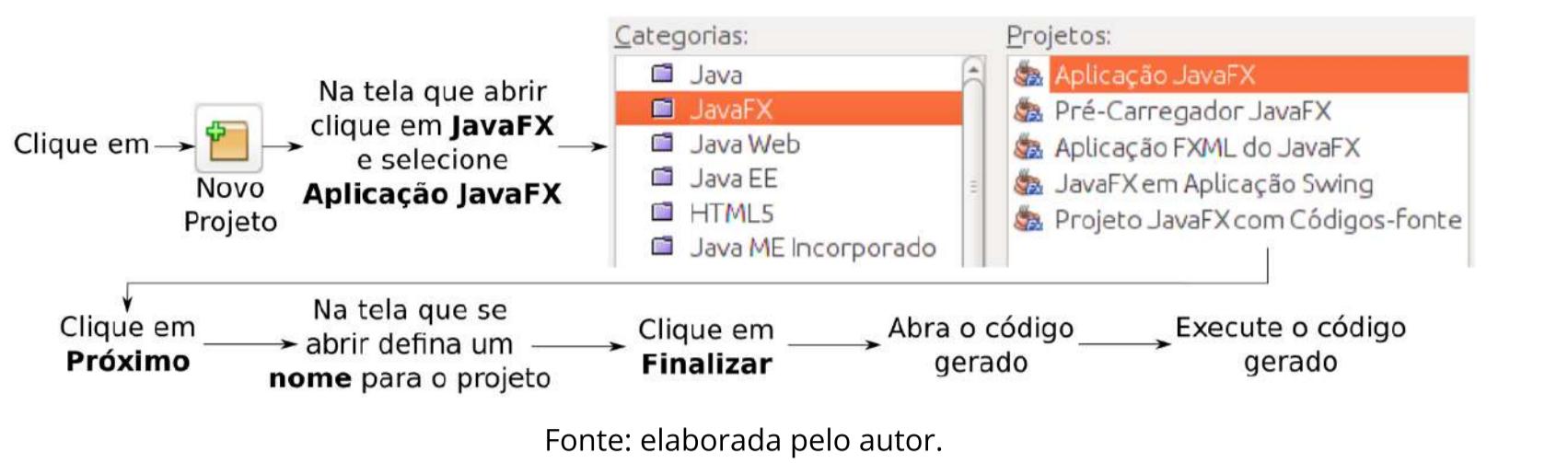
DICA

Caso você esteja trabalhando em outro IDE, veja se ele dá suporte à criação de aplicação de exemplo em JavaFX. Se o seu IDE não tem essa opção disponível, tudo bem! Basta digitar o código abaixo.

Lembre-se de que a única restrição é utilizar o Java 8 (JDK 1.8) ou superior.

Nota: se você estiver utilizando a versão do OpenJDK, é possível que tenha algumas problemas e que seja preciso configurar manualmente o JavaFX.

Figura 3.10 | Passo a passo de como criar a primeira aplicação com JavaFX



0

Ver anotações

Analise o Código 3.23 que foi gerado a partir dos passos da Figura 3.10.

Código 3.23 | Exemplos de aplicação *hello world* utilizando JavaFX

0

Ver anotações

```
1 import javafx.application.Application;
2 import javafx.event.ActionEvent;
3 import javafx.event.EventHandler;
4 import javafx.scene.Scene;
5 import javafx.scene.control.Button;
6 import javafx.scene.layout.StackPane;
7 import javafx.stage.Stage;
8 public class AppHelloWorld extends Application {
9     public static void main(String[] args) {
10         launch(args);
11     }
12     @Override
13     public void start(Stage janela) {
14         Button btn = new Button();
15         btn.setText("Imprime 'Hello World'");
16         btn.setOnAction(new EventHandler<ActionEvent>() {
17             @Override
18             public void handle(ActionEvent event) {
19                 System.out.println("Hello World!");
20             }
21         });
22         StackPane gerenciadorLayout = new StackPane();
23         gerenciadorLayout.getChildren().add(btn);
24         Scene cena = new Scene(gerenciadorLayout, 300, 100);
25         janela.setTitle("Aplicação Hello World!");
26         janela.setScene(cena);
27         janela.show();
28     }
29 }
```

Fonte: elaborado pelo autor.

No Código 3.23, nas linhas de 1 a 7, temos um conjunto de importações de classes da biblioteca do JavaFX. Essas importações carregam as classes que permitem a criação da aplicação gráfica, abstraindo, assim, os detalhes de implementação de diversos componentes, como o botão (Button), o gerenciador de *layout* (*StackPane*), a cena (*Scene*), entre outros. Em seguida, na linha 8, temos a criação

o

Ver anotações

da classe *AppHelloWorld*, que herda da classe *Application*. Já nas linhas 9 a 11, temos o método *main*, que é responsável por lançar a aplicação por meio da invocação do método *launch*, que é um método estático disponível na classe *Application*. Ao herdar da classe *Application*, você deve, obrigatoriamente, implementar o método abstrato *start* (nas linhas 12 a 28), que é chamado automaticamente pelo JavaFX, além disso, é o local onde definimos os elementos presentes na interface gráfica e invocamos a nossa lógica de negócios. Nas linhas 14 e 15, criamos um botão e, então, definimos um texto para ele. Já nas linhas 16 a 21, criamos um evento que executará uma ação toda vez que clicarmos no botão. A ação executada, neste caso, será imprimir a mensagem “Hello World!”. Na linha 22, por sua vez, foi criado um objeto do tipo *StackPane*, que nada mais é do que um gerenciador de *layout*. Já na linha 23, o botão foi adicionado ao gerenciador de *layout*, e na linha 24, uma cena com o gerenciador de *layout* utilizado foi criada e as dimensões da janela foram especificadas. Nas linhas 25 e 26, foi definido um título para a janela da aplicação e, então, a cena foi colocada dentro da janela. Por fim, na linha 27, pedimos para que a janela gráfica seja mostrada no monitor.

Diante disso, reveja a Figura 3.8 e tente compreender como ocorreu a lógica de adicionarmos um componente sobre o outro. Por exemplo: a cena foi adicionada sobre a janela, o gerenciador de *layout* foi adicionado na cena e, por fim, o botão foi adicionado sobre o gerenciador de *layout*.

Após analisar o Código 3.23, mande executá-lo. Clique no botão “Imprime ‘Hello World’” e veja a saída impressa no terminal. Ao executar a aplicação, uma tela semelhante à mostrada na Figura 3.11 deverá aparecer. Em seguida, faça alguns testes redimensionando as dimensões da janela e, por fim, clique nos botões maximizar, minimizar e fechar; perceba que essas funcionalidades testadas são propriedades inerentes à janela criada com o JavaFX, mesmo você não tendo implementado o código para realizar isso.

Pronto, você criou o seu primeiro código que utiliza interfaces gráficas em Java.

Figura 3.11 | Interface gráfica da aplicação *hello world*



Fonte: elaborada pelo autor.

Neste próximo exemplo, vamos adaptar um pouco esse código de *Hello World* para transformá-lo em uma aplicação que calcula o Fibonacci de um número lido da tela e exibe o resultado. Dessa maneira, analise o Código 3.24 mostrado a seguir.

Código 3.24 | Exemplos de aplicação que calcula o Fibonacci e exibe o resultado

Ver anotações

```
1 import javafx.application.Application;
2 import javafx.event.ActionEvent;
3 import javafx.event.EventHandler;
4 import javafx.geometry.Insets;
5 import javafx.geometry.Pos;
6 import javafx.scene.Scene;
7 import javafx.scene.control.Button;
8 import javafx.scene.control.Label;
9 import javafx.scene.control.TextArea;
10 import javafx.scene.control.TextField;
11 import javafx.scene.layout.VBox;
12 import javafx.scene.paint.Color;
13 import javafx.scene.text.Font;
14 import javafx.stage.Stage;
15 public class AppFibonacci extends Application {
16     public static void main(String[] args) {
17         launch(args);
18     }
19     @Override
20     public void start(Stage janela) {
21         Label lbl = new Label("Calcula Fibonacci");
22         lbl.setTextFill(Color.DARKGREEN);
23         lbl.setFont(Font.font("Serif", 25));
24         TextField numField = new TextField();
25         ImageView imgBtn = new ImageView(new Image(getClass()
26             .getResourceAsStream("recursos/icones/mission.png")));
27         Button btn = new Button("Calcular", imgBtn);
28         TextArea textArea = new TextArea();
29         btn.setOnAction(new EventHandler<ActionEvent>() {
30             @Override
31             public void handle(ActionEvent event) {
32                 try {
33                     int num=Integer.parseInt(numField.getText());
34                     if (num >= 1 && num <= 42) {
35                         String resultado = String.format(
36                             "Fibonacci(%d) = %d\n",
37                             num, fibonacci(num));
38                         textArea.appendText(resultado);
39                     } else if (num < 1) {
40                         String msg = String.format(
41                             "O número deve ser maior ou igual a 1 e menor ou igual a 42."
42                         );
43                         textArea.appendText(msg);
44                     }
45                 } catch (Exception e) {
46                     e.printStackTrace();
47                 }
48             }
49         });
50         VBox vBox = new VBox(lbl, numField, btn, textArea);
51         Scene scene = new Scene(vBox, 300, 200);
52         janela.setScene(scene);
53         janela.show();
54     }
55 }
```

```

41                     "O número deve ser >= 1\n");
42                     textArea.appendText(msg);
43             } else if (num > 42) {
44                     String msg = String.format(
45                         "Número muito grande\n");
46                     textArea.appendText(msg);
47             }
48         } catch (Exception ex) {
49                     String msg = String.format(
50                         "Digite um número inteiro\n");
51                     textArea.appendText(msg);
52             }
53         }
54     });
55     VBox vbox = new VBox(lbl, numField, btn, textArea);
56     vbox.setPadding(new Insets(10, 10, 10, 10));
57     vbox.setSpacing(10);
58     vbox.setAlignment(Pos.TOP_CENTER);
59     vbox.setStyle("-fx-background-color: #BBDDFF;");
60     Scene cena = new Scene(vbox, 400, 300);
61     janela.setTitle("Aplicação Calcula Fibonacci");
62     janela.setScene(cena);
63     janela.show();
64 }
65 public long fibonacci(int n) {
66     if (n == 1 || n == 2) {
67         return 1;
68     } else {
69         return fibonacci(n - 1) + fibonacci(n - 2);
70     }
71 }
72 }
```

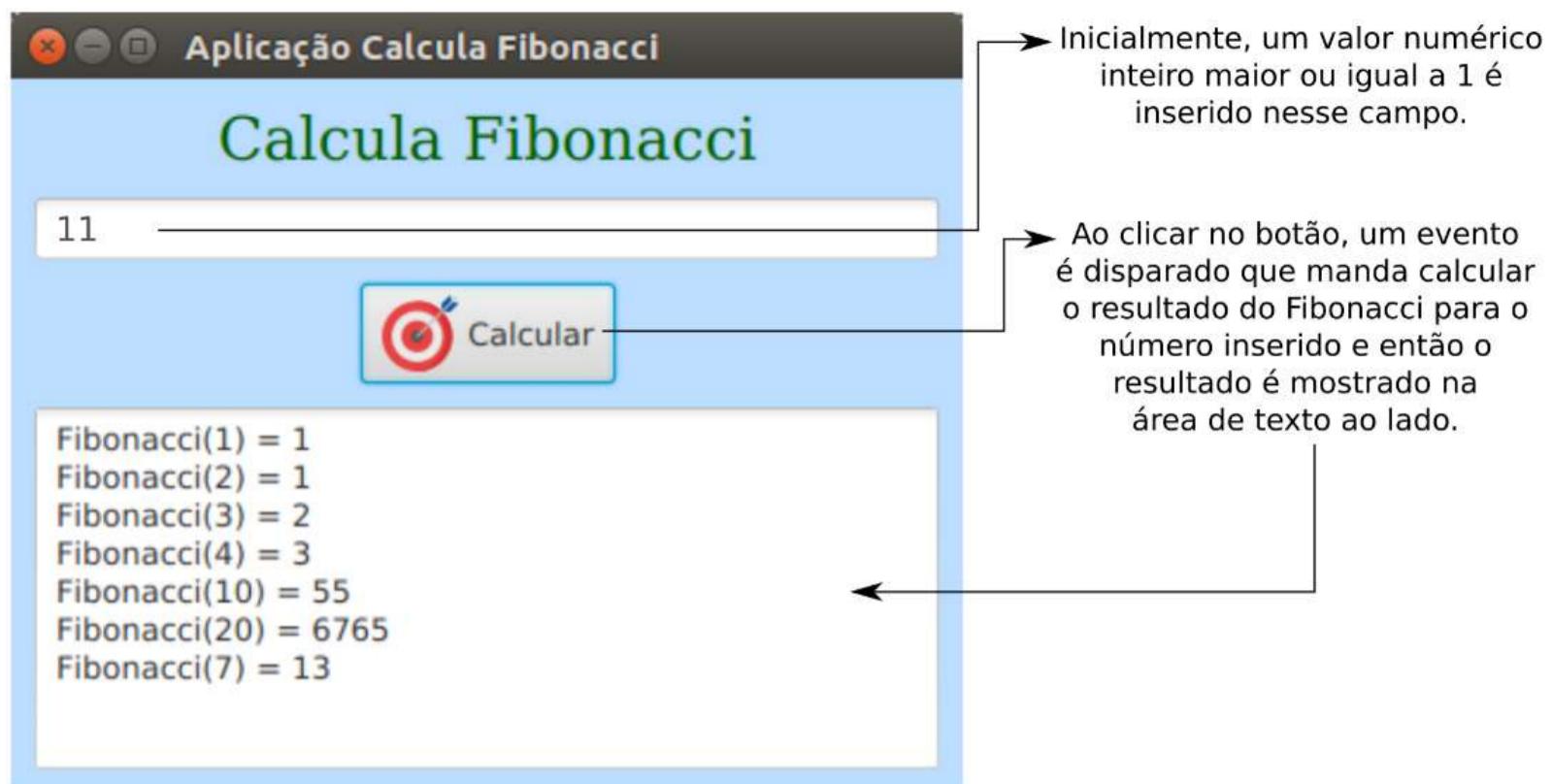
Fonte: elaborado pelo autor.

No Código 3.24, nas linhas 1 a 14, temos as importações das classes do JavaFX que permitem a utilização dos componentes gráficos; já nas linhas 16 a 18, temos o método *main*; na linha 21, temos a declaração de um rótulo (label) que irá aparecer no topo da janela; nas linhas 22 e 23, definimos a cor desse rótulo e a fonte utilizada seguida do tamanho da fonte; na linha 24, criamos um campo de texto

que irá receber o número digitado para calcularmos o Fibonacci; nas linhas 25 e 26, carregamos uma imagem de um ícone para deixar a aplicação com aspecto mais agradável (repare que, para o carregamento da imagem, foi preciso passar o caminho relativo do local em que ela se encontra, assim, nesse exemplo, a imagem está na pasta “recursos” e na subpasta “ícones”); na linha 27, criamos um botão com um ícone carregado; na linha 28, criamos uma área de texto em que será exibido o resultado do cálculo; já nas linhas 29 a 54, definimos um evento que será disparado caso o usuário clique no botão. A lógica desse trecho é basicamente pegar o número digitado no campo de texto, convertê-lo em um número inteiro, calcular o Fibonacci e exibir o resultado na área de texto. Repare que foi criado um tratamento de exceção caso um número não inteiro seja digitado pelo usuário, bem como só será calculado o Fibonacci para números maiores ou iguais a um (abaixo desse valor, a operação não será definida) e menor ou igual a 42 (acima desse valor, ocorrerá overflow). Na linha 55, foi criado um gerenciador de *layout* que organiza os componentes na vertical; nas linhas 56 e 57, por sua vez, foi dado um espaçamento entre as bordas e os componentes; nas linhas 58 e 59, foram definidos o tipo de alinhamento e a cor de fundo; nas linhas 60 a 63, foi criada a cena e adicionada na janela; por fim, nas linhas 65 a 71, temos o método que calcula o Fibonacci de um número inteiro.

Implemente o Código 3.24 e mande executá-lo. Faça a inserção de diversos valores no campo de texto, clique no botão calcular e analise o resultado. Sugerimos que faça diversas modificações nos parâmetros dos métodos acima, como *setTextFill*, *setFont*, *setPadding*, *setSpacing*, *setAlignment* e *setStyle*, bem como execute a aplicação para compreender, de fato, como esses parâmetros influenciam no aspecto gráfico.

Figura 3.12 | Interface gráfica da aplicação que calcula Fibonacci



Ver anotações

Fonte: elaborada pelo autor.

DICA

No Código 3.24 acima, foi utilizado um ícone para deixar a interface mais intuitiva e agradável. Atualmente, existem diversos sites na internet que disponibilizam ícones de forma gratuita. Alguns bons sites para baixar ícones são: Flaticon, Iconfinder e Icon-icons.

0

[Ver anotações](#)

Imagine, agora, que queremos construir uma aplicação que desenhe figuras geométricas 2D. Dessa maneira, o nosso próximo exemplo nos mostrará como desenhar retângulos, círculos, elipses, linhas, polígonos e polilinhas a partir do JavaFX. Para isso, as seguintes classes serão utilizadas: Rectangle, Circle, Ellipse, Line, Polygon e Polyline.

Analise o Código 3.25, que apresenta o trecho do método *start*, que é responsável por criar as figuras geométricas mencionadas. Alguns trechos do código foram omitidos por serem similares ao código 3.24, como a importação das bibliotecas, a criação da classe que herda Application e a criação do método *main*.

Código 3.25 | Trecho de código que mostra figuras geométricas 2D

```
1 public void start(Stage janela) {  
2     Rectangle retangulo = new Rectangle(100, 100);  
3     retangulo.setTranslateX(10);  
4     retangulo.setTranslateY(10);  
5     retangulo.setFill(Color.RED);  
6     Circle circulo = new Circle(50);  
7     circulo.setTranslateX(170);  
8     circulo.setTranslateY(60);  
9     circulo.setFill(Color.GREEN);  
10    Circle circunferencia = new Circle(50);  
11    circunferencia.setTranslateX(280);  
12    circunferencia.setTranslateY(60);  
13    circunferencia.setStroke(Color.BLUE);  
14    circunferencia.setStrokeWidth(3.0);  
15    circunferencia.setFill(Color.WHITE);  
16    Ellipse elipse = new Ellipse(50, 25);  
17    elipse.setTranslateX(390);  
18    elipse.setTranslateY(60);  
19    elipse.setFill(Color.YELLOW);  
20    Line linha = new Line(10.0f, 10.0f, 100.0f, 100.0f);  
21    linha.setTranslateX(450);  
22    linha.setTranslateY(10);  
23    linha.setStroke(Color.GREENYELLOW);  
24    linha.setStrokeWidth(3.0);  
25    Polygon poligono = new Polygon();  
26    poligono.getPoints().addAll(new Double[]{  
27        0.0, 00.0,  
28        100.0, 0.0,  
29        50.0, 100.0  
30    });  
31    poligono.setTranslateX(550);  
32    poligono.setTranslateY(10);  
33    poligono.setFill(Color.SKYBLUE);  
34    Polyline polilinha = new Polyline();  
35    polilinha.getPoints().addAll(new Double[]{  
36        0.0, 0.0,  
37        100.0, 0.0,  
38        0.0, 100.0,  
39        100.0, 100.0  
40    });  
});
```

```
41     polilinha.setTranslateX(670);
42     polilinha.setTranslateY(10);
43     polilinha.setStroke(Color.BLACK);
44     polilinha.setStrokeWidth(3.0);
45     Group grupo = new Group();
46     grupo.getChildren().addAll(retangulo, circulo,
47         circunferencia, elipse, linha, poligono, polilinha);
48     Scene cena = new Scene(grupo, 800, 120);
49     janela.setTitle("Aplicação Figuras Geométricas 2D");
50     janela.setScene(cena);
51     janela.show();
52 }
```

0

Ver anotações

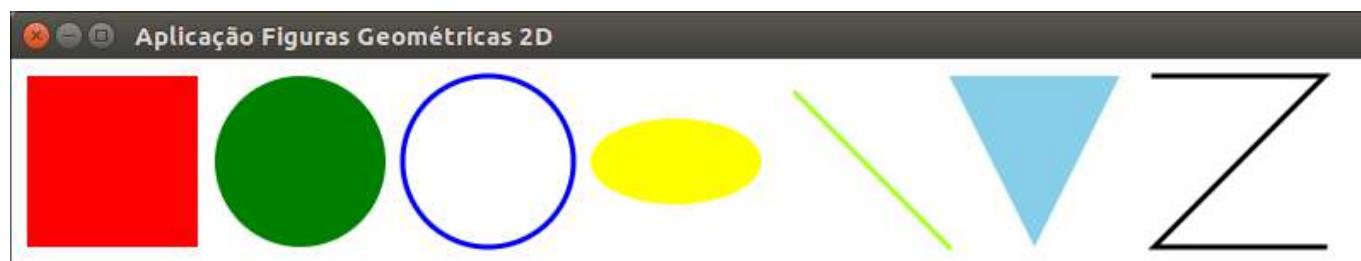
Fonte: elaborado pelo autor.

No Código 3.25, na linha 2, criamos o retângulo que especifica a largura e a altura do objeto (a unidade de medida desses atributos é *pixels*); em seguida, nas linhas 3 a 5, definimos a localização na coordenada (x, y) e a cor do retângulo; já na linha 6, criamos um círculo passando o raio; já as linhas 7 a 9 são semelhantes ao retângulo e não serão comentadas; na linha 10, criamos um objeto chamado circunferência, e a novidade em relação ao círculo está nas linhas 13 e 14, em que definimos a cor da borda e a grossura da linha; na linha 16, criamos uma elipse e passamos o raio nos eixos x e y; na linha 20, criamos um objeto do tipo linha, em que são passadas as coordenadas (x, y) de início e (x, y) de fim; já nas linhas 25 a 30, criamos um polígono e adicionamos as coordenadas de seus vértices; nas linhas 34 a 40, criamos uma polilinha e adicionamos as coordenadas de seus vértices; por fim, nas linhas 45 a 47, criamos um nó do tipo grupo e adicionamos todas as figuras geométricas 2D.

Implemente o Código 3.25 e mande executá-lo. Sugerimos que você faça diversas modificações nos parâmetros dos métodos, como cor, localização (x, y), dimensões das figuras, entre outras, bem como execute a aplicação para compreender, de fato, como esses parâmetros influenciam no aspecto gráfico.

A Figura 3.13 nos mostra a interface gráfica gerada pelo código 3.25.

Figura 3.13 | Interface gráfica da aplicação que manipula figuras geométricas 2D



Fonte: elaborada pelo autor.

0

Ver anotações

ASSIMILE

As telas gráficas construídas com JavaFX utilizam um sistema de coordenadas semelhantes às coordenadas cartesianas para inserção dos objetos/nós no cenário. O cenário da Figura 3.13 possui 800×120 pixels, e faz-se importante ter em mente que a coordenada $(0, 0)$ está na parte superior esquerda. O eixo x cresce no sentido da esquerda para direita, já o eixo y cresce no sentido de cima para baixo (inverso do sistema cartesiano tradicional). O sistema de coordenadas utilizado pelo JavaFX é igual ao utilizado pela ferramenta Greenfoot, dessa maneira, analise a Figura 1.22, presente na Unidade 1, para compreender melhor esses dados.

Imagine, agora, que queremos construir uma aplicação que carregue imagens da internet e as exiba na tela. A biblioteca JavaFX nos ajudará a fazer isso de forma fácil. Para tanto, analise o Código 3.26 que apresenta os principais trechos da aplicação responsáveis por isso. Alguns trechos do código foram omitidos por serem similares ao código 3.24, como a importação das bibliotecas, a criação da classe que herda Application e a criação do método *main*.

Código 3.26 | Trecho de código que manipula imagens

```

1  private final String IMG_URL1 = "https://bit.ly/2Pxqady";
2  private final String IMG_URL2 = "https://bit.ly/2EZlriY";
3  private final String IMG_URL3 = "https://bit.ly/3fylvmj";
4  private final String IMG_URL4 = "https://bit.ly/3ia00gk";
5
6  @Override
7
8  public void start(Stage janela) {
9
10    Text texto = new Text("Aplicação Exibe Imagens");
11
12    texto.setTranslateX(50);
13
14    texto.setTranslateY(20);
15
16    Image img1 = new Image(IMG_URL1);
17
18    ImageView view1 = new ImageView(img1);
19
20    view1.setTranslateX(100);
21
22    view1.setTranslateY(30);
23
24    Image img2 = new Image(IMG_URL2);
25
26    ImageView view2 = new ImageView(img2);
27
28    view2.setTranslateX(100);
29
30    view2.setTranslateY(90);
31
32    Image img3 = new Image(IMG_URL3);
33
34    ImageView view3 = new ImageView(img3);
35
36    view3.setTranslateX(100);
37
38    view3.setTranslateY(160);
39
40    Image img4 = new Image(IMG_URL4);
41
42    ImageView view4 = new ImageView(img4);
43
44    view4.setTranslateX(100);
45
46    view4.setTranslateY(220);
47
48    Group grupo = new Group();
49
50    grupo.getChildren().addAll(texto, view1, view2, view3, view4);
51
52    Scene cena = new Scene(grupo, 260, 300);
53
54    janela.setTitle("Aplicação Manipula Imagens");
55
56    janela.setScene(cena);
57
58    janela.show();
59
60  }

```

0

Ver anotações

Fonte: elaborado pelo autor.

No Código 3.26, nas linhas 1 a 4, temos as URLs (*Uniform Resource Locator*) das imagens; já na linha 7, criamos um objeto de texto para ser colocado no topo da tela (semelhante a um *label*); na linha 10, criamos uma imagem a partir da URL; na

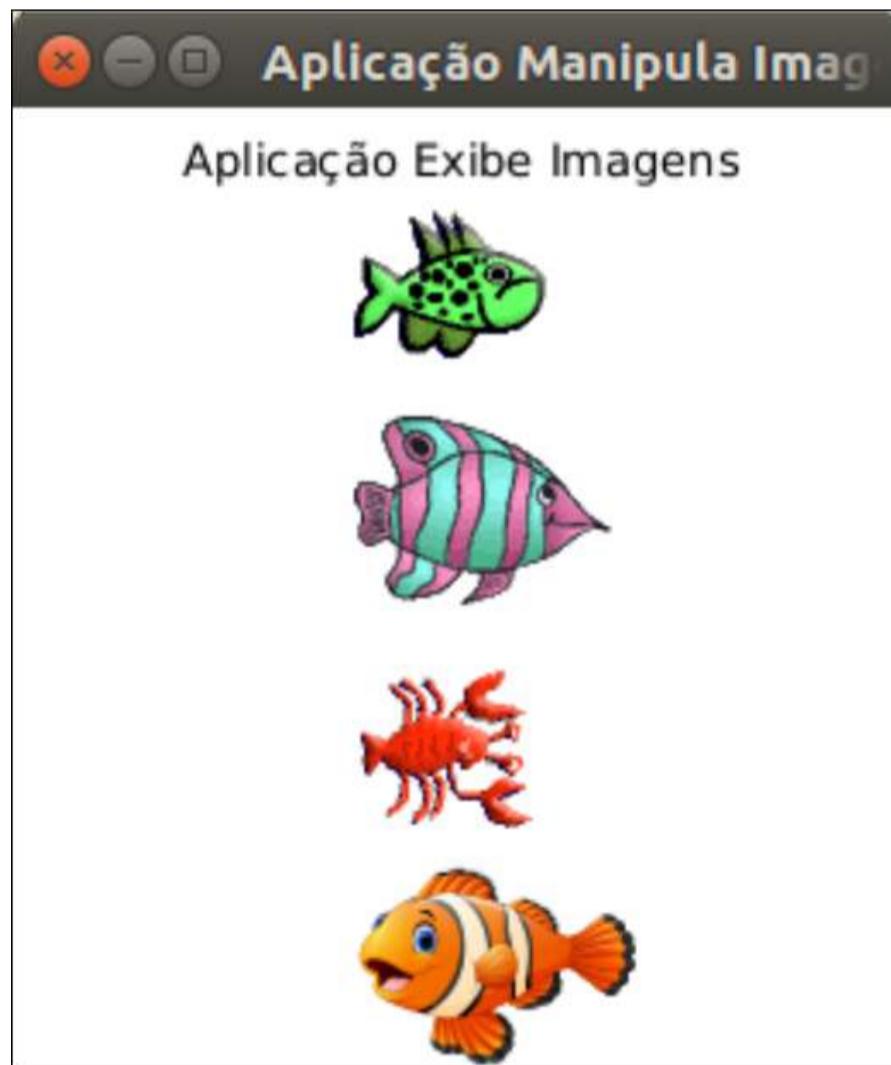
linha 11, criamos um objeto do tipo *ImageView*, que é a imagem visualizável na tela, pois *ImageView* é um tipo de nó (herda da classe *Node*); por fim, as demais linhas do código são semelhantes às linhas anteriores já comentadas.

DICA

No Código 3.26, as URLs foram encurtadas para ficarem mais limpas no material, mas essa não é uma boa prática, pois cria uma camada a mais e desnecessária de processamento. Nesse tipo de aplicação, o ideal é que se tenha as imagens armazenadas localmente no computador, pois, assim, evita-se o tempo de carregamento da imagem no site, além de permitir que sua aplicação seja executada mesmo sem acesso à internet.

Implemente o Código 3.26 e o execute. A Figura 3.14 nos apresenta a interface gráfica gerada por esse código.

Figura 3.14 | Interface gráfica da aplicação que manipula imagens



Fonte: elaborada pelo autor.

CRIAÇÃO DE GRÁFICOS

A biblioteca JavaFX dá um bom suporte também à criação de gráficos. Vamos criar, agora, uma aplicação que plota dois gráficos na tela. O primeiro gráfico plotado é um gráfico de barras, já o segundo é um gráfico de pizza. Analise o Código 3.27 que nos apresenta um trecho da aplicação que exibe os gráficos mencionados.

Código 3.27 | Trecho de código que exibe gráficos utilizando-se JavaFX

0

[Ver anotações](#)

```
1 public void start(Stage janela) {  
2     int dimx = 1000;  
3     int dimy = 440;  
4     BarChart bar = new BarChart<>(new CategoryAxis(), new  
5         NumberAxis());  
6     bar.setTitle("Relação de Vendas de Carros Por Ano");  
7     XYChart.Series carro1 = new XYChart.Series();  
8     carro1.setName("Chevrolet Onix");  
9     carro1.getData().add(new XYChart.Data("2017", 188654));  
10    carro1.getData().add(new XYChart.Data("2018", 210466));  
11    carro1.getData().add(new XYChart.Data("2019", 241214));  
12    carro1.getData().add(new XYChart.Data("2020", 17463));  
13    XYChart.Series carro2 = new XYChart.Series();  
14    carro2.setName("Hyundai HB20");  
15    carro2.getData().add(new XYChart.Data("2017", 105539));  
16    carro2.getData().add(new XYChart.Data("2018", 105518));  
17    carro2.getData().add(new XYChart.Data("2019", 101590));  
18    carro2.getData().add(new XYChart.Data("2020", 6555));  
19    XYChart.Series carro3 = new XYChart.Series();  
20    carro3.setName("Ford Ka");  
21    carro3.getData().add(new XYChart.Data("2017", 94893));  
22    carro3.getData().add(new XYChart.Data("2018", 104450));  
23    carro3.getData().add(new XYChart.Data("2019", 104331));  
24    carro3.getData().add(new XYChart.Data("2020", 7334));  
25    bar.getData().addAll(carro1, carro2, carro3);  
26    bar.setPrefSize(dimx/2 - 20, dimy - 20);  
27    PieChart graficoPizza = new PieChart();  
28    graficoPizza.setTitle("Relação Consumo Energia por Dia");  
29    graficoPizza.getData().addAll(  
30        new PieChart.Data("Domingo", 18),  
31        new PieChart.Data("Segunda", 38),  
32        new PieChart.Data("Terça", 34),  
33        new PieChart.Data("Quarta", 35),  
34        new PieChart.Data("Quinta", 31),  
35        new PieChart.Data("Sexta", 36),  
36        new PieChart.Data("Sábado", 27));  
37    graficoPizza.setPrefSize(dimx/2 - 20, dimy - 20);  
38    HBox hbox = new HBox(bar, graficoPizza);  
39    Scene cena = new Scene(hbox, dimx, dimy);  
janela.setTitle("Aplicação Exibe Gráficos");
```

0

Ver anotações

```

40     janela.setScene(cena);
41     janela.show();
42 }

```

0

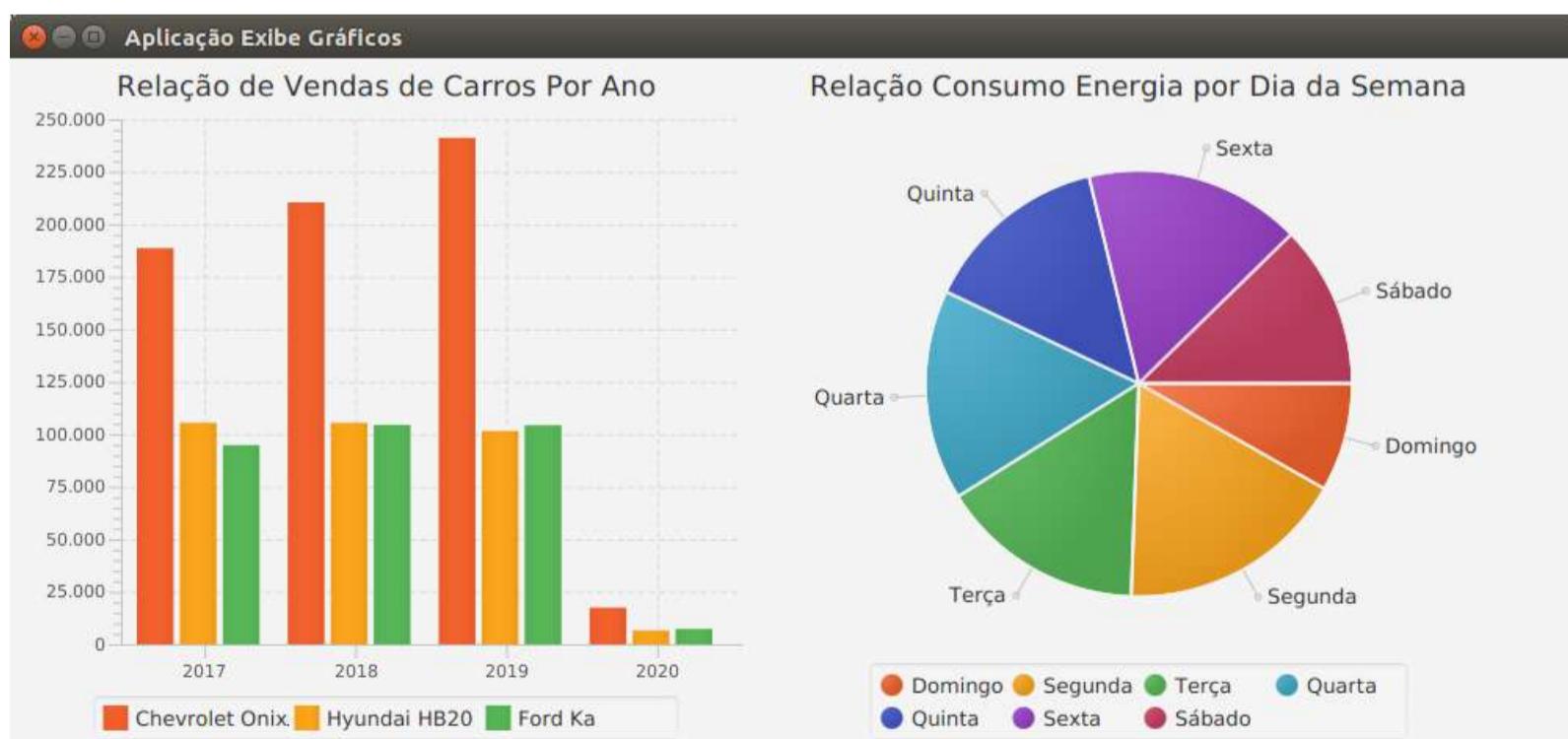
Fonte: elaborado pelo autor.

[Ver anotações](#)

Nas linhas 2 e 3, do Código 3.27, estão as dimensões x e y da janela; nas linhas 4 e 5, criamos um objeto que armazenará os dados do gráfico de barras e definimos um título para ele; nas linhas 6 e 7, criamos um objeto que armazenará as séries de dados e definimos o seu nome (neste caso, dados do carro Chevrolet Onix); já nas linhas 8 a 11, foram adicionados os dados desse carro, sendo que o primeiro valor é o eixo x e o segundo é o eixo y; nas linhas 13 a 23, mais dois tipos de objetos com os dados dos carros foram criados; na linha 24, foi adicionado ao gráfico de barras os dados relativos aos carros criados; nas linhas 26 e 27, foi criado um gráfico de pizza e seu título foi definido; nas linhas 28 a 35, os dados foram adicionados ao gráfico de pizza; por fim, na linha 37, um gerenciador de layout que organiza os nós na horizontal foi criado.

Implemente o Código 3.27 e o execute; após isso, uma tela semelhante à Figura 3.15 deverá ser mostrada. Leitor, faça alterações nos parâmetros dos métodos, execute a aplicação e analise as mudanças ocorridas. Esse tipo de teste o ajudará a compreender a influência de cada um dos parâmetros na aplicação.

Figura 3.15 | Interface gráfica da aplicação que exibe gráficos



Fonte: elaborada pelo autor.

INTERAÇÃO POR MEIO DO TECLADO

Vamos considerar, agora, que queremos construir uma aplicação em que controlamos um robô em um cenário livre de obstáculos. Esse robô poderá se mover para cima, para baixo, para a esquerda e para a direita; o controle dele será feito por meio das setas do teclado do computador; ele poderá se movimentar em duas velocidades diferentes; e esse recurso será acionado ao pressionarmos a tecla espaço. Nessa aplicação, também desejamos que o robô fique restrito às dimensões da janela, ou seja, ele não pode sair da janela gráfica. Dessa maneira, analise o Código 3.28 mostrado abaixo.

Código 3.28 | Aplicação que faz um robô se movimentar no cenário

```
1 import javafx.application.Application;
2 import javafx.scene.Group;
3 import javafx.scene.Scene;
4 import javafx.scene.image.Image;
5 import javafx.scene.image.ImageView;
6 import javafx.scene.input.KeyCode;
7 import javafx.stage.Stage;
8
9 public class AppMovRobo extends Application {
10
11     private final int DIM_X = 300;
12
13     private final int DIM_Y = 220;
14
15     private final int altura = 60;
16
17     private final int largura = 64;
18
19     private int posX = DIM_X/2 - largura/2;
20
21     private int posY = DIM_Y/2 - altura/2;
22
23     private int velocidade = 1;
24
25     private final Image imgRoboFrente = new Image(getClass()
26
27             .getResourceAsStream("recursos/robo1.png"));
28
29     private final Image imgRoboCostas = new Image(getClass()
30
31             .getResourceAsStream("recursos/robo2.png"));
32
33     private final Image imgRoboEsq = new Image(getClass()
34
35             .getResourceAsStream("recursos/robo3.png"));
36
37     private final Image imgRoboDir = new Image(getClass()
38
39             .getResourceAsStream("recursos/robo4.png"));
40
41     private final ImageView imgRobo = new ImageView(imgRoboFrente);
42
43     public static void main(String[] args) {
44
45         launch(args);
46
47     }
48
49     @Override
50
51     public void start(Stage janela) {
52
53         imgRobo.setTranslateX(posX);
54
55         imgRobo.setTranslateY(posY);
56
57         Group grupo = new Group();
58
59         grupo.getChildren().addAll(imgRobo);
60
61         Scene cena = new Scene(grupo, DIM_X, DIM_Y);
62
63         janela.setTitle("Aplicação Movimentar Robô");
64
65         janela.setScene(cena);
66
67         janela.show();
68
69         cena.setOnKeyPressed((evt) -> {
70
71             if (evt.getCode() == KeyCode.UP) {
72
73                 imgRobo.setImage(imgRoboCostas);
74
75             }
76
77         });
78
79     }
80
81 }
```

```
41         posY = posY - velocidade;
42
43         if (posY < 0) {
44
45             posY = 0;
46
47         }
48
49         imgRobo.setTranslateX(posX);
50
51         imgRobo.setTranslateY(posY);
52
53     }
54
55     imgRobo.setTranslateX(posX);
56
57     imgRobo.setTranslateY(posY);
58
59 }
60
61     if (evt.getCode() == KeyCode.LEFT) {
62
63         imgRobo.setImage(imgRoboEsq);
64
65         posX = posX - velocidade;
66
67         if (posX < 0) {
68
69             posX = 0;
70
71         }
72
73         imgRobo.setTranslateX(posX);
74
75         imgRobo.setTranslateY(posY);
76
77     }
78
79     if (evt.getCode() == KeyCode.RIGHT) {
80
81         imgRobo.setImage(imgRoboDir);
82
83         posX = posX + velocidade;
84
85         if (posX + largura > DIM_X) {
86
87             posX = DIM_X - largura;
88
89         }
90
91         imgRobo.setTranslateX(posX);
92
93         imgRobo.setTranslateY(posY);
94
95     }
96
97     if (evt.getCode() == KeyCode.SPACE) {
98
99         if (velocidade == 1) {
100
101             velocidade = 4;
102
103         } else {
104
105             velocidade = 1;
106
107         }
108
109     }
110
111 }
```

```
82     });
83 }
84 }
```

Fonte: elaborado pelo autor.

0

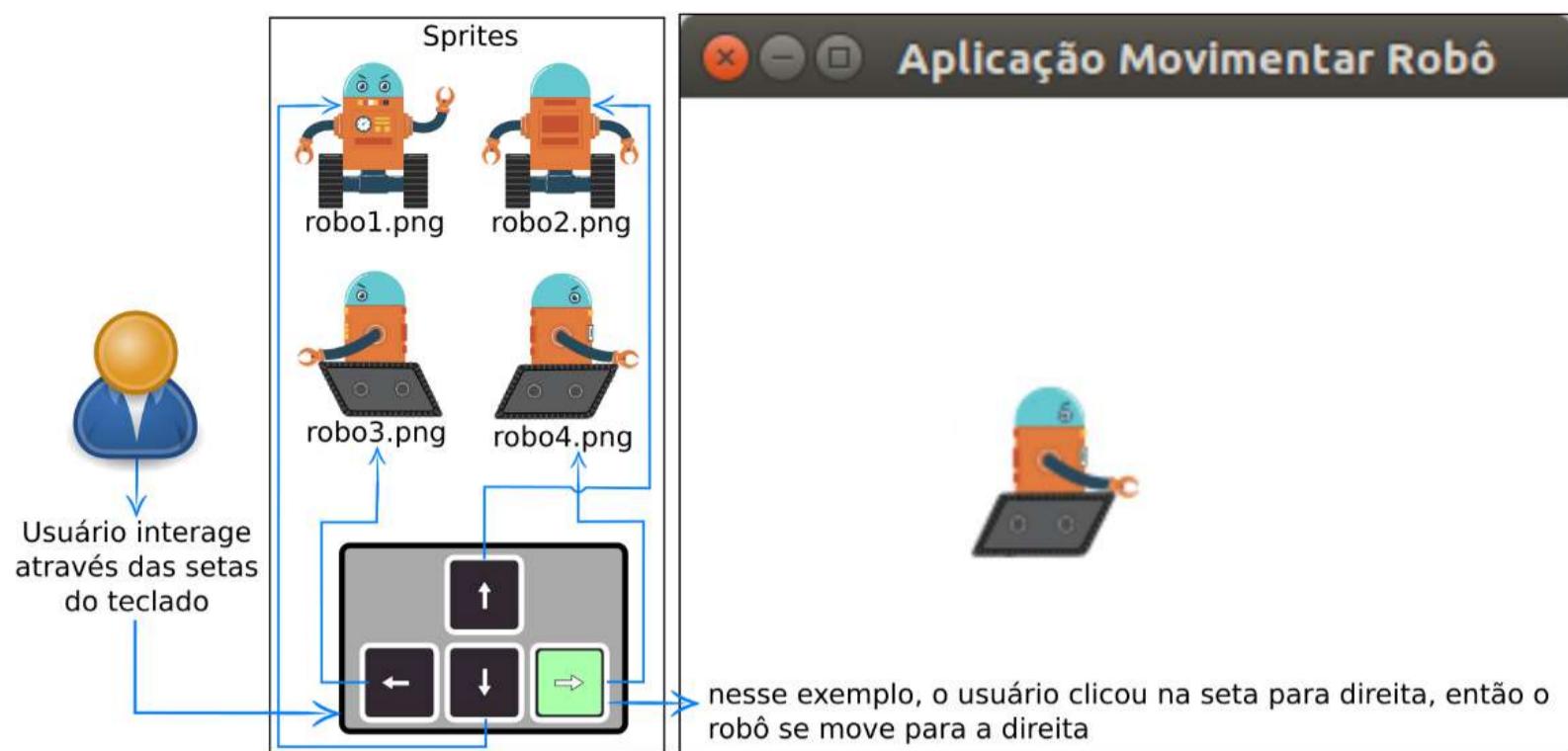
Ver anotações

No Código 3.28, nas linhas 9 a 15, temos a declaração de diversos atributos que são intuitivos, logo, não serão explicados; já nas linhas 16 a 23, foram carregadas quatro imagens diferentes do robô, cada uma com uma determinada orientação. As imagens estão especificadas a partir do caminho relativo e estão disponíveis no GitHub do autor. Na linha 24, temos um objeto do tipo *ImageView* que é responsável por exibir o robô na tela. A lógica principal da aplicação está nas linhas 38 a 82, que definem os eventos disparados quando clicadas as setas do teclado e a tecla espaço. Nas linhas 39 a 47, temos o controle do robô caso pressionada a tecla seta para cima; por fim, o restante do código intuitivo é semelhante ao código comentado, portanto, não será explicado.

EXEMPLIFICANDO

O Código 3.28 exemplifica como se criar uma aplicação com GUI em que seja possível interagir com o personagem por meio de comandos do teclado. Implemente o Código 3.28 e execute-o; após isso, uma tela semelhante à Figura 3.16 deverá ser mostrada. Utilize as setas do teclado para controlar o robô; aperte também a tecla espaço e perceba que o robô ficará mais rápido ou mais lento à medida que você pressioná-la. Aplicações como essas, que se assemelham a um jogo de computador, ficam mais ricas/divertidas quanto mais comandos forem inseridos. Caso desejar, utilize essa aplicação como base para realizar a implementação de um jogo de seu interesse.

Figura 3.16 | Interface gráfica da aplicação que movimenta o robô



Ver anotações
0

Fonte: elaborada pelo autor.

REFLITA

Agora que você aprendeu a criar diversas aplicações com JavaFX, reflita sobre como os códigos 3.24, 25, 26, 27 e 28 seguem a estrutura mostrada na Figura 3.8. Tente desenhar uma imagem semelhante à Figura 3.8 para cada um dos códigos. Você irá notar que o que muda de uma figura para outra é a estrutura de nós; o restante da aplicação permanece fixa. Esperamos que essa reflexão e esse exercício o ajudem a criar aplicações com dezenas e até centenas de nós.

Gostaríamos de ressaltar que o foco desta seção é a criação de aplicações básicas e simples envolvendo interfaces gráficas, logo, a separação entre os componentes da interface e da lógica de negócio não foi feita. No entanto, é importante que o leitor aprenda a desenvolver códigos modularizados separando a lógica de negócio da interface gráfica, principalmente quando as aplicações crescem.

Caro estudante, nesta seção você estudou os conteúdos relacionados à criação de interfaces gráficas, criou botões, rótulos, campos de texto, áreas de texto, desenhou figuras, inseriu imagens, criou gráficos e tratou eventos de clique. Foram mostrados a você diversos exemplos desses conceitos em JavaFX, e todos os códigos aqui mostrados podem ser acessados no GitHub do autor.

Na próxima seção, vamos construir aplicações que utilizam vetores e matrizes, bem como realizar diversas manipulações envolvendo strings; dessa maneira, avançaremos ainda mais no entendimento da linguagem Java.

REFERÊNCIAS

ARANTES, J. da S. **Livro-POO-Java**. 2020. Disponível em: <https://bit.ly/3eiUMcF>. Acesso em: 29 jul. 2020.

CAELUM. **Java e orientação a objetos**: apostila do curso FJ-11. [s.d.]. Disponível em: <https://bit.ly/2ZpTqrZ>. Acesso em: 28 ago. 2020.

CURSO EM VÍDEO. **Curso de Java #05 - Introdução ao Swing e JavaFX**. 2015. Disponível em: <https://bit.ly/3iD60Mg>. Acesso em: 28 ago. 2020.

DEITEL, P. J.; DEITEL, H. M. **Java**: como programar. 10. ed. São Paulo: Pearson Education, 2016.

DESCOMPILA. **JavaFX para Iniciantes**. 2018. Disponível em: <https://bit.ly/32uwdqO>. Acesso em: 28 ago. 2020.

ECLIPSE. **SWT: The Standard Widget Toolkit**. 2020. Disponível em: <https://bit.ly/2FFPiNy>. Acesso em: 28 ago. 2020.

GITHUB. **SwingLabs SwingX Project**. 2020. Disponível em: <https://bit.ly/2FvaVR6>. Acesso em: 28 ago. 2020.

JGOODIES. **Professional Java Desktop**. [s.d.]. Disponível em: <https://bit.ly/3hxzsC5>. Acesso em: 28 ago. 2020.

OPEN JFX. **JavaFX**. [s.d.]. Disponível em: <https://bit.ly/2E3dusR>. Acesso em: 28 ago. 2020.

ORACLE. **Abstract Window Toolkit**. 2020. Disponível em: <https://bit.ly/3c0xWae>. Acesso em: 28 ago. 2020.

ORACLE. **Trail: creating a GUI with JFC/Swing**. [s.d.]. Disponível em: <https://bit.ly/33A7vo2>. Acesso em: 28 ago. 2020.

PIVOT. **Apache Pivot**. [s.d.]. Disponível em: <https://bit.ly/2ZF12XW>. Acesso em: 28 ago. 2020.

QTJAMBI. **Qt Jambi Reference Documentation**. 2009. Disponível em: <https://bit.ly/3hyx1iC>. Acesso em: 28 ago. 2020.

SIQUEIRA, W. A. **Visualizando gráficos de funções matemáticas com JavaFX**. 2018. Disponível em: <https://bit.ly/3c2BrNB>. Acesso em: 28 ago. 2020.

FOCO NO MERCADO DE TRABALHO

DESENVOLVIMENTO DE INTERFACES GRÁFICAS NA LINGUAGEM JAVA

Jesimar da Silva Arantes

Ver anotações

IMPLEMENTAÇÃO DE UMA INTERFACE GRÁFICA PARA O ROBÔ

Desenvolvimento da GUI para o robô por meio da biblioteca JavaFX.



Fonte: Shutterstock

Deseja ouvir este material?

Áudio disponível no material digital.

SEM MEDO DE ERRAR

A *startup* em que você trabalha lhe passou uma grande tarefa. O seu gestor lhe pediu para implementar uma interface gráfica para o robô utilizando a biblioteca JavaFX.

Como forma de iniciar o trabalho, você decidiu partir do Código 3.22 para adicionar os recursos da interface gráfica. Após quebrar a cabeça, você conseguiu terminar o desenvolvimento. O Código 3.29 destacou apenas as principais alterações feitas.

mas o código completo pode ser acessado no GitHub do autor. Caro aluno, faça as devidas implementações e execute o código para que possa entender melhor o seu funcionamento.

Código 3.29 | Modelagem da interface gráfica da aplicação simulador de robótica

0

[Ver anotações](#)

```
1 //código de importação omitidos
2
3 public class AppGUI extends Application {
4
5     private final String IMG_FUNDO = "recursos/galpao.png";
6
7     private final String IMG_FRENT = "recursos/robo1.png";
8
9     private final String IMG_COSTAS = "recursos/robo2.png";
10
11    private final String IMG_ESQ = "recursos/robo3.png";
12
13    private final String IMG_DIR = "recursos/robo4.png";
14
15    private final String IMG_BOX_LVR = "recursos/box.png";
16
17    private final String IMG_BOX_HD = "recursos/box2.png";
18
19    private final String IMG_BOX_PRT = "recursos/box3.png";
20
21    private final Image imgFundo = new Image(getClass()
22
23        .getResourceAsStream(IMG_FUNDO));
24
25    private final Image imgRoboFrete = new Image(getClass()
26
27        .getResourceAsStream(IMG_FRENT));
28
29    private final Image imgRoboCostas = new Image(getClass()
30
31        .getResourceAsStream(IMG_COSTAS));
32
33    private final Image imgRoboEsq = new Image(getClass()
34
35        .getResourceAsStream(IMG_ESQ));
36
37    private final Image imgRoboDir = new Image(getClass()
38
39        .getResourceAsStream(IMG_DIR));
40
41    private final Image imgBoxLvr = new Image(getClass()
42
43        .getResourceAsStream(IMG_BOX_LVR));
44
45    private final Image imgBoxHd = new Image(getClass()
46
47        .getResourceAsStream(IMG_BOX_HD));
48
49    private final Image imgBoxPrt = new Image(getClass()
50
51        .getResourceAsStream(IMG_BOX_PRT));
52
53    private final ImageView viewFundo = new ImageView(imgFundo);
54
55    private final ImageView viewRobo = new ImageView(imgRoboFrete);
56
57    private final ImageView viewBoxLvr = new ImageView(imgBoxLvr);
58
59    private final ImageView viewBoxHd = new ImageView(imgBoxHd);
60
61    private final ImageView viewBoxPrt = new ImageView(imgBoxPrt);
62
63    private final Mundo2D mundo = new Mundo2D(600, 400);
64
65    private final Robo robo = new Robo(32, 300);
66
67    private final Caixa caixaLvr = new Caixa(
68
69        "Caixa de Livros", 15, 25, 100, 30, 0.4f, 0.4f, 0.3f);
70
71    private final Caixa caixaHd = new Caixa(
72
73        "Caixa de HDs", 20, 280, 50, 40, 0.5f, 0.5f, 0.3f);
74
75    private final Caixa caixaPrt = new Caixa(
76
77        "Caixa Impressoras", 8, 525, 100, 40, 0.6f, 0.6f, 0.4f);
78
79
80    public static void main(String[] args) {
```

0

Ver anotações

```
41     launch(args);
42 }
43 @Override
44 public void start(Stage janela) {
45     viewFundo.setTranslateX(0);
46     viewFundo.setTranslateY(0);
47     viewBoxLvr.setTranslateX(caixaLvr.getPosX());
48     viewBoxLvr.setTranslateY(caixaLvr.getPosY());
49     viewBoxHd.setTranslateX(caixaHd.getPosX());
50     viewBoxHd.setTranslateY(caixaHd.getPosY());
51     viewBoxPrt.setTranslateX(caixaPrt.getPosX());
52     viewBoxPrt.setTranslateY(caixaPrt.getPosY());
53     viewRobo.setTranslateX(robo.getPosicaoX());
54     viewRobo.setTranslateY(robo.getPosicaoY());
55     Group grupo = new Group();
56     grupo.getChildren().addAll(viewFundo, viewBoxLvr,
57         viewBoxHd, viewBoxPrt, viewRobo);
58     Scene cena = new Scene(grupo, mundo.DIM_X, mundo.DIM_Y);
59     janela.setTitle("Simulador de Robótica");
60     janela.setScene(cena);
61     janela.show();
62     cena.setOnKeyPressed((evt) -> {
63         if (evt.getCode() == KeyCode.UP) {
64             viewRobo.setImage(imgRoboCostas);
65             robo.setPosicaoY(
66                 robo.getPosicaoY()-(int)robo.getVelocidade());
67             if (robo.getPosicaoY() < 0) {
68                 robo.setPosicaoY(0);
69             }
70             viewRobo.setTranslateX(robo.getPosicaoX());
71             viewRobo.setTranslateY(robo.getPosicaoY());
72         }
73         //código de movimentação DOWN, LEFT e RIGHT omitidos
74         if (evt.getCode() == KeyCode.SPACE) {
75             if (robo.getVelocidade() == 1) {
76                 robo.setVelocidade(5);
77             } else {
78                 robo.setVelocidade(1);
79             }
80         }
81     });
}
```

```
82     }
83 }
```

Fonte: elaborado pelo autor.

0

Ao executar o Código 3.29, uma tela semelhante à Figura 3.17 deve ser mostrada. Repare que utilizamos o JavaFX e conseguimos reproduzir um simulador bem parecido com o construído utilizando a ferramenta Greenfoot mostrada na Seção 3 da Unidade 1.

[Ver anotações](#)

Figura 3.17 | Interface gráfica do simulador de robótica



Fonte: elaborada pelo autor.

ARRAYS E STRINGS EM JAVA

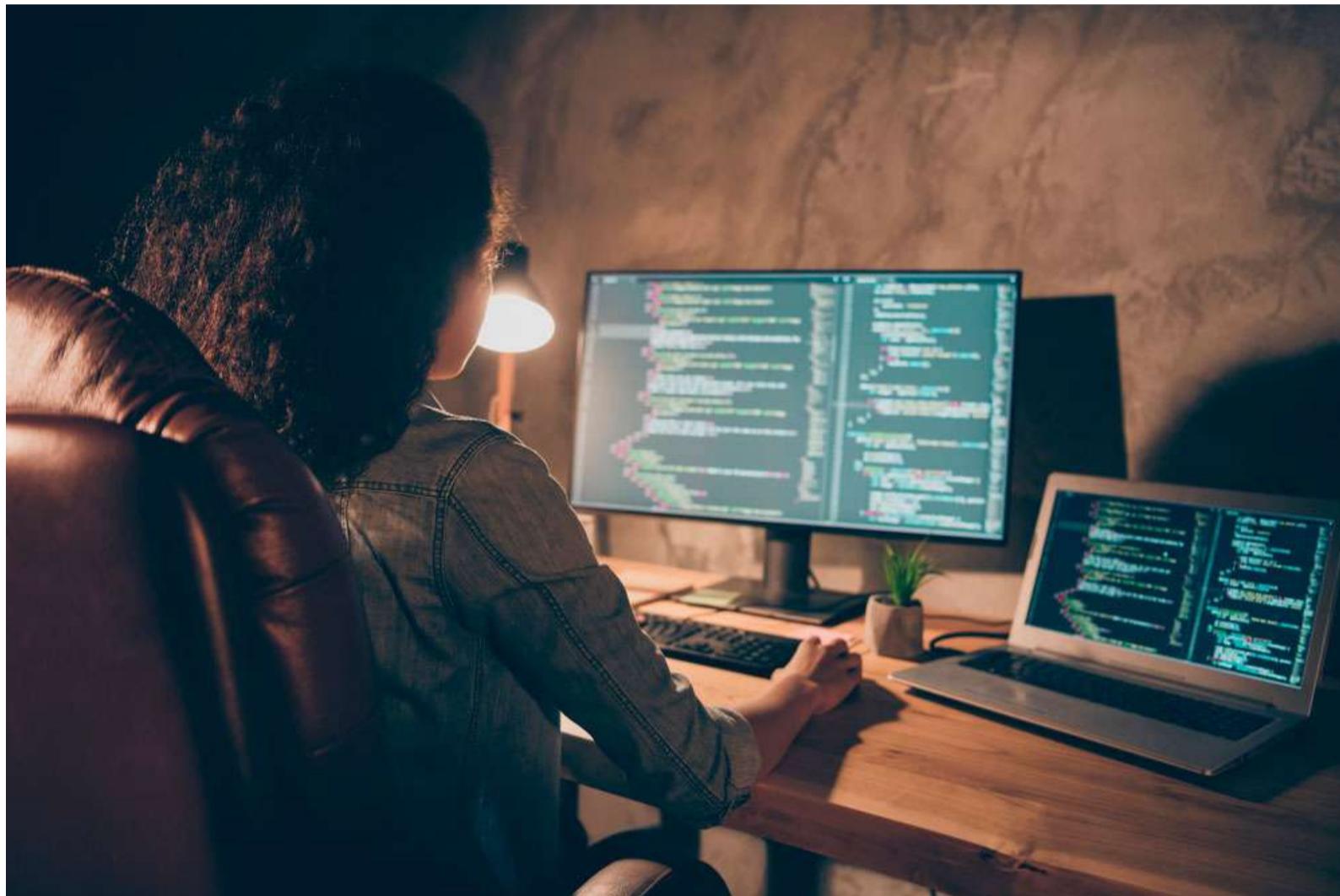
Jesimar da Silva Arantes

0

[Ver anotações](#)

APLICAÇÕES COM ARRAYS UNIDIMENSIONAIS E BIDIMENSIONAIS

A maioria dos desenvolvedores já se deparou com problemas que exigiram um grande volume de variáveis do mesmo tipo, tornando necessário o uso de estruturas de dados como vetores e matrizes para serem resolvidos.



Fonte: Shutterstock.

Deseja ouvir este material?

Áudio disponível no material digital.

CONVITE AO ESTUDO

Prezado estudante, bem-vindo à quarta e última unidade do livro Linguagem Orientada a Objetos. Nesta unidade, vamos estudar alguns assuntos importantes dentro do desenvolvimento orientado a objetos (OO), estabelecer as ideias por meio do estudo dos *arrays* (vetores) unidimensionais e multidimensionais, aprender mais sobre a classe *String*, trabalhar com leituras de arquivos, aprender como interagir com um banco de dados e ver brevemente como funcionam as *threads* em Java.

Após a leitura desta unidade, espera-se que você, aluno, seja capaz de criar e manipular vetores unidimensionais e multidimensionais, manipular a classe String, criar aplicações utilizando banco de dados relacional e criar e manipular as *threads* para realizar processamento em paralelo.

0

Ao longo da unidade, é recomendado que o aluno desenvolva e teste todos os códigos apresentados, pois isso o levará a ter uma maior fixação do conteúdo, bem como utilize alguma plataforma de compartilhamento de código, como o GitHub. O livro apresentará diversos exemplos práticos que o auxiliarão de forma direta na compreensão da teoria, e as capacidades de reflexão, modelagem e abstração, por sua vez, serão treinadas ao longo deste livro.

Ver anotações

O conteúdo desta unidade facilitará a compreensão da Linguagem Orientada a Objetos focando aplicações sobre a linguagem Java; para isso, a unidade foi dividida nas seguintes seções: a Seção 4.1 apresentará os conceitos por trás dos *arrays* e *strings* em Java; a Seção 4.2 trará aplicações relacionadas ao uso de banco de dados; e a Seção 4.3 mostrará os passos necessários para a criação de aplicações envolvendo *threads*.

Pelo fato desta unidade ser prática, convido você a implementar as atividades aqui propostas. Lembre-se de que a prática leva à perfeição.

Bons estudos!

PRATICAR PARA APRENDER

Caro aluno, bem-vindo à primeira seção da quarta e última unidade de estudos sobre Linguagem Orientada a Objetos. Qual desenvolvedor nunca se deparou com problemas que exigiram um grande volume de variáveis do mesmo tipo? Acredito que a maioria dos desenvolvedores já se deparou com problemas que exigiram estruturas de dados como vetores e matrizes para serem resolvidos. Pois bem, nesta seção, estudaremos os *arrays* unidimensional e multidimensional e como manipular literais (*strings*) em Java. Para isso, três classes serão estudadas: String, StringBuilder e StringBuffer.

De forma a contextualizar sua aprendizagem, lembre-se de que você está trabalhando em um simulador de robô. O seu patrão, por sua vez, reviu o seu código e percebeu que, na sala em que o robô operará, existem poucas caixas, logo, pediu a você que criasse um *array* unidimensional para colocar as caixas de livros e impressoras. Como ele sabe que a sala possuirá mais caixas de HDs,

solicitou a criação de um *array* multidimensional de caixas de HDs. Além disso, ele percebeu que o robô estava se movendo livremente pelo cenário (sala), mesmo sobre as caixas, e lhe pediu para restringir a sua região de navegação, argumentando que isso tornará a simulação mais realista.

o

Diante desse cenário que lhe foi apresentado, como você criará esse *array* unidimensional? O que é um *array* unidimensional? Como você criará um *array* multidimensional? E o que é um *array* multidimensional? (Esta seção o auxiliará na resposta de tais perguntas.)

Ver anotações

Muito bem, agora que você foi apresentado à sua nova situação-problema, estude esta seção e compreenda como a linguagem Java trata os *arrays* de dados; esses conceitos são fundamentais em quaisquer linguagens de programação.

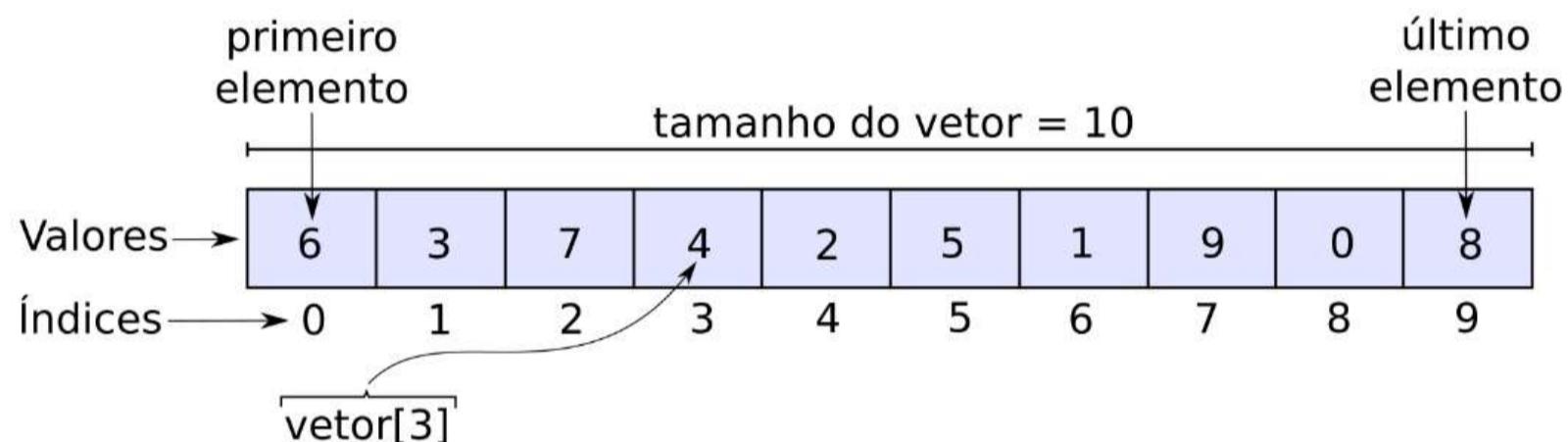
E aí, vamos juntos compreender esses conceitos e resolver esse desafio?

Bom estudo!

ARRAYS UNIDIMENSIONAIS: VETORES

A linguagem Java permite a criação de *arrays* unidimensionais também chamados de vetores. Um vetor agrupa um conjunto de dados de um mesmo tipo organizados em linha ou coluna, e por trabalhar com dados do mesmo tipo, falamos que o vetor é uma estrutura de dados homogênea. A criação e a manipulação de vetores em Java têm algumas similaridades com a linguagem C, mas parte da alocação de vetores em Java é mais simples que na linguagem C, pois não é necessária a utilização de comandos como o *malloc* (alocação de memória). Analise a Figura 4.1 que nos mostra a representação de um *array* unidimensional.

Figura 4.1 | Representação de um vetor unidimensional em forma de linha



Fonte: elaborada pelo autor.

Na Figura 4.1, podemos ver um vetor em forma de linha, e abaixo dos valores, temos os índices do vetor. Os índices são sempre valores inteiros que vão de zero até o tamanho do vetor menos um. Nesse exemplo, o nosso vetor tem tamanho 10, assim, os índices vão de 0 a 9. Os índices são utilizados para acessar as posições e os valores armazenados nos vetores. A tentativa de acessar um valor de índice negativo, maior ou igual ao tamanho do vetor lança uma exceção do tipo *ArrayIndexOutOfBoundsException*. O Quadro 4.1 agrupa um conjunto de comandos que fazem a criação e manipulação de um vetor unidimensional.

Quadro 4.1 | Síntese dos principais comandos para criação e manipulação de um vetor

Operação sobre Vetores	Exemplo de Código
Declaração de um vetor.	<code>tipo nomeVetor[];</code>
Alocação de espaço de um vetor.	<code>nomeVetor = new tipo[tamanho];</code>

Operação sobre Vetores	Exemplo de Código
Declaração e alocação de um vetor.	<code>tipo nomeVetor[] = new tipo[tamanho];</code>
Acesso de uma posição do vetor.	<code>nomeVetor[indice]</code>
Atribuição de um valor a uma posição.	<code>nomeVetor[indice] = valor;</code>
Acesso ao tamanho de um vetor.	<code>nomeVetor.length</code>

Fonte: elaborado pelo autor.

Imagine que queremos elaborar uma simples aplicação que cria um vetor de inteiros com tamanho 5, preencher esses valores e, por fim, imprimir os valores preenchidos. O Código 4.1 nos mostra essa primeira aplicação utilizando vetores.

Código 4.1 | Trecho de código que cria, preenche e imprime os elementos de um vetor

```

1 int vet[] = new int[5];
2 vet[0] = 6;
3 vet[1] = 3;
4 vet[2] = 7;
5 vet[3] = 4;
6 vet[4] = 2;
7 for (int i = 0; i < vet.length; i++) {
8     System.out.println("Array[" + i + "]: " + vet[i]);
9 }
```

Fonte: elaborado pelo autor.

Na linha 1 do Código 4.1, temos a declaração do nosso *array*, nomeado de *vet*, que contém cinco posições; já nas linhas 2 a 6, fizemos a inicialização dos valores do vetor; por fim, nas linhas 7 a 9, temos um laço que percorre todas as posições do vetor e o imprime.

Por meio de uma análise do exemplo acima, podemos perceber que a inicialização do vetor ocupou muitas linhas de código, logo, outra forma de fazermos isso é inicializar os valores na criação do vetor. Observe o Quadro 4.2 em que são mostrados alguns exemplos de como isso pode ser feito para diferentes tipos de dados. Neste quadro, podemos perceber que existem duas formas de se declarar

o vetor: na primeira delas, o colchete é colocado depois do nome da variável; já na segunda, o colchete é colocado antes do nome da variável; ambas as formas funcionam em Java.

Quadro 4.2 | Duas formas distintas de se fazer a criação e a inicialização de um vetor

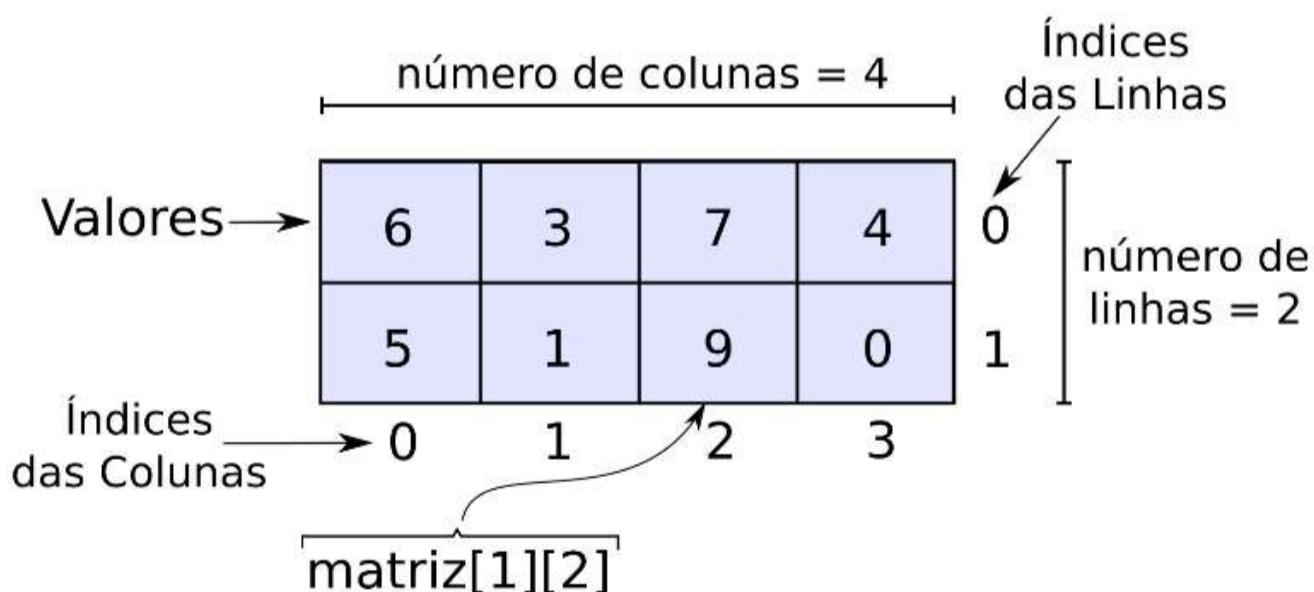
Tipo do Vetor	Exemplos usando a forma 1	Exemplos usando a forma 2
Inteiro	<code>int vet[] = {1, 5, 3, 4};</code>	<code>int[] vet = {1, 5, 3, 4};</code>
Real	<code>double vet[] = {2.0, 5.3, 3.8};</code>	<code>double[] vet = {2.0, 5.3, 3.8};</code>
Caractere	<code>char vet[] = {'a', 'e', 'i'};</code>	<code>char[] vet = {'a', 'e', 'i'};</code>

Fonte: elaborado pelo autor.

■ ARRAYS MULTIDIMENSIONAIS: MATRIZES

Os *arrays* multidimensionais, também chamados de matrizes, também são estruturas de dados muito importantes. Uma matriz agrupa um conjunto de dados de um mesmo tipo organizados em forma de linhas e colunas. Da mesma forma que os vetores, a criação e manipulação de matrizes em Java têm algumas similaridades com a linguagem C. Analise a Figura 4.2 que nos mostra a representação de um *array* bidimensional.

Figura 4.2 | Representação de uma matriz bidimensional



Fonte: elaborada pelo autor.

Na Figura 4.2, podemos ver uma matriz bidimensional em forma de linhas e colunas cujas células da matriz são acessadas pelos seus índices, que são sempre valores inteiros. O Quadro 4.3 agrupa um conjunto de comandos que fazem a criação e manipulação de uma matriz. Analise.

Quadro 4.3 | Síntese dos principais comandos para criação e manipulação de uma matriz

Operação sobre Matrizes	Exemplo de Código
Declaração de uma matriz.	<i>tipo</i> nomeMat[][];
Alocação de espaço de uma matriz.	nomeMat = new <i>tipo</i> [tamX][tamY];
Declaração e alocação de uma matriz.	<i>tipo</i> nomeMat[][] = new <i>tipo</i> [tamX][tamY];
Acesso de uma posição da matriz.	nomeMat[indiceX][indiceY]
Atribuição de um valor a uma posição.	nomeMat[indiceX][indiceY] = valor;
Acesso ao número de linhas.	nomeMat.length
Acesso ao número de colunas da i-ésima linha.	nomeMat[i].length

0

Ver anotações

Fonte: elaborado pelo autor.

Imagine, agora, que queremos elaborar uma simples aplicação que cria uma matriz de *doubles* com três linhas e duas colunas já preenchidas na declaração, bem como imprimir os valores preenchidos. O Código 4.2 nos mostra essa aplicação utilizando matrizes.

Código 4.2 | Trecho de código que cria, preenche e imprime os elementos de uma matriz

```

1 double mat[][] = {{1.5, 5.2}, {3.6, 4.9}, {2.4, 8.1}};
2 for (int i = 0; i < mat.length; i++) {
3     for (int j = 0; j < mat[i].length; j++) {
4         System.out.println("M[" + i + "][" + j + "]: " + mat[i][j]);
5     }
6 }
```

Fonte: elaborado pelo autor.

No Código 4.2, na linha 1, temos a declaração e o preenchimento do nosso *array* bidimensional, chamado mat, que contém três linhas e duas colunas, totalizando seis elementos. Na linha 2, por sua vez, iniciamos o primeiro laço que percorrerá as linhas da matriz; já na linha 3, iniciamos o segundo laço que percorrerá as colunas

da matriz; por fim, na linha 4, fizemos a impressão dos elementos da matriz. Por meio do exemplo acima, percebemos que podemos fazer a inicialização dos elementos da matriz também durante a sua declaração.

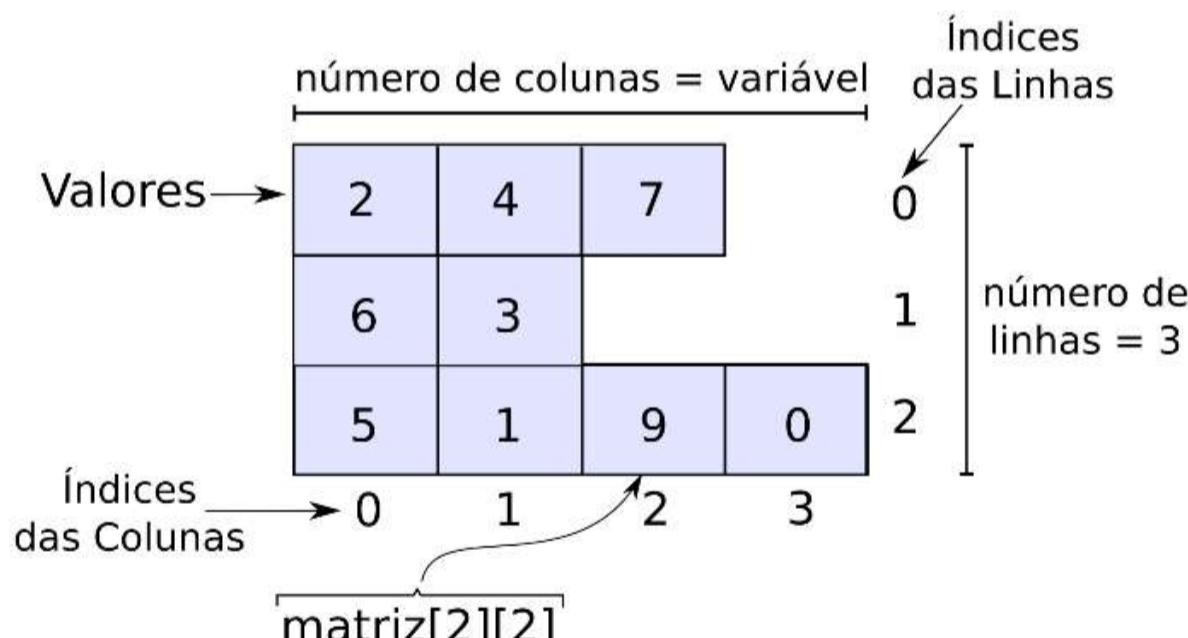
0

ASSIMILE

Em algumas aplicações, desejamos construir matrizes bidimensionais cujo comprimento das linhas varie. Um exemplo de aplicação em que essa ideia é utilizada é a construção de uma matriz bidimensional em que as linhas refletem os meses do ano (de janeiro a dezembro) e as colunas refletem os dias do mês. Como sabemos, o número de dias de cada mês varia, uma vez que janeiro, março, maio, julho, agosto, outubro e dezembro têm 31 dias, já os meses de abril, junho, setembro e novembro têm 30 dias e fevereiro tem 28 (nesse exemplo, desconsideramos os anos bissextos para simplificar). Dessa maneira, nessa aplicação, a matriz teria 12 linhas (indicando os meses) e um número variável de colunas, que vai de 28 a 31 (indicando os dias).

De forma a ilustrar essa ideia, vamos imaginar que queremos construir uma matriz como a indicada na Figura 4.3.

Figura 4.3 | Representação de matriz 2D com comprimento variável das linhas



Fonte: elaborada pelo autor.

O Código 4.3 a seguir nos mostra como construir uma matriz com comprimento de linhas variado.

Implemente esse código e faça testes; mude os valores da matriz e analise o seu comportamento.

```
1 int mat[][] = {{2, 4, 7}, {6, 3}, {5, 1, 9, 0}};
2 for (int i = 0; i < mat.length; i++) {
3     for (int j = 0; j < mat[i].length; j++) {
4         System.out.println("M["+i+"]["+j+"]="+mat[i][j]);
5     }
6 }
```

0

Ver anotações

Fonte: elaborado pelo autor.

Caro leitor, caso tenha interesse, no GitHub do autor há um exemplo de código que mostra a construção da matriz com dias do mês variável. Nessa aplicação, a matriz é construída de uma forma mais dinâmica e simples, mas com o mesmo efeito do exemplo explicado acima.

EXEMPLIFICANDO

Existem diversas aplicações envolvendo vetores que necessitam que os dados estejam ordenados. A linguagem Java já possui métodos capazes de ordenar vetores de tipos básicos ou derivados, desde que implementem a classe Comparable. A seguir, no Código 4.4, mostraremos como utilizar o método estático *sort*(ordenar) da classe Arrays, presente na biblioteca java.util.Arrays, que deve ser importada.

Analise o código mostrado abaixo.

Código 4.4 | Ordenação de um vetor com o método *sort* da classe Arrays

```
1 int vetor[] = {6, 3, 7, 4, 2, 5, 1, 9, 0, 8};
2 System.out.println("Vetor antes de ordenar");
3 for (int i = 0; i < vetor.length; i++) {
4     System.out.print(vetor[i]);
5 }
6 Arrays.sort(vetor);
7 System.out.println("Vetor depois de ordenar");
8 for (int i = 0; i < vetor.length; i++) {
9     System.out.println(vetor[i]);
10 }
```

Fonte: elaborado pelo autor.

Como curiosidade, no Código 4.4, o método de ordenação *sort* utiliza, internamente, uma variante do método quicksort que utiliza dois pivôs, por isso, esse método é chamado de *dual-pivot quicksort*.

STRINGS

Ao longo do livro, utilizamos a classe **String** para declarar variáveis (objetos) do tipo literal; agora, vamos entender com mais detalhes como ela funciona.

A classe **String** possui uma importante característica, que é ser imutável. Isso quer dizer que uma vez atribuído um valor literal para a variável, existirá uma cópia dessa variável na memória do computador. O Java, por sua vez, permite que você coloque outro conteúdo nessa variável, no entanto, ela continuará na memória, só que sem nenhum objeto apontando para si. Assim, se você construir uma aplicação que modifique constantemente o valor da variável, você terá um desempenho ruim em termos de gastos de memória e de processamento.

A linguagem Java possui duas outras importantes classes que manipulam *strings*, que são: a `StringBuilder` e a `StringBuffer`. Os valores literais armazenados nessas duas classes são mutáveis, ou seja, podem ser alterados a qualquer momento sem que seja feita a cópia do literal na memória.

A classe **StringBuilder** é uma boa alternativa à classe `String` se desejar que o conteúdo da *string* seja mutável. Já a classe **StringBuffer** é uma boa alternativa às outras duas classes se desejar que o conteúdo seja mutável e necessitar de uma aplicação *thread safe*. Dizemos que uma classe é *thread safe* quando duas ou mais *threads* não podem invocar métodos dessa classe simultaneamente. Para isso, existe um mecanismo de sincronização dos métodos que garante esse aspecto de segurança em aplicações *multi-threads* (mais detalhes sobre *threads* serão estudados na Seção 4.3).

A fim de que possa entender melhor como criar valores literais utilizando essas três classes apresentadas, analise o Código 4.5 abaixo.

Código 4.5 | Criação de valores literais com as classes `String`, `StringBuilder` e `StringBuffer`

```
1 String str1 = new String("Orientação a objetos");
2 String str2 = "Orientação a objetos";
3 StringBuilder str3 = new StringBuilder("Orientação a objetos");
4 StringBuilder str4 = "Orientação a objetos"; //não é aceito (dá erro)
5 StringBuffer str5 = new StringBuffer("Orientação a objetos");
6 StringBuffer str6 = "Orientação a objetos"; //não é aceito (dá erro)
```

Fonte: elaborado pelo autor.

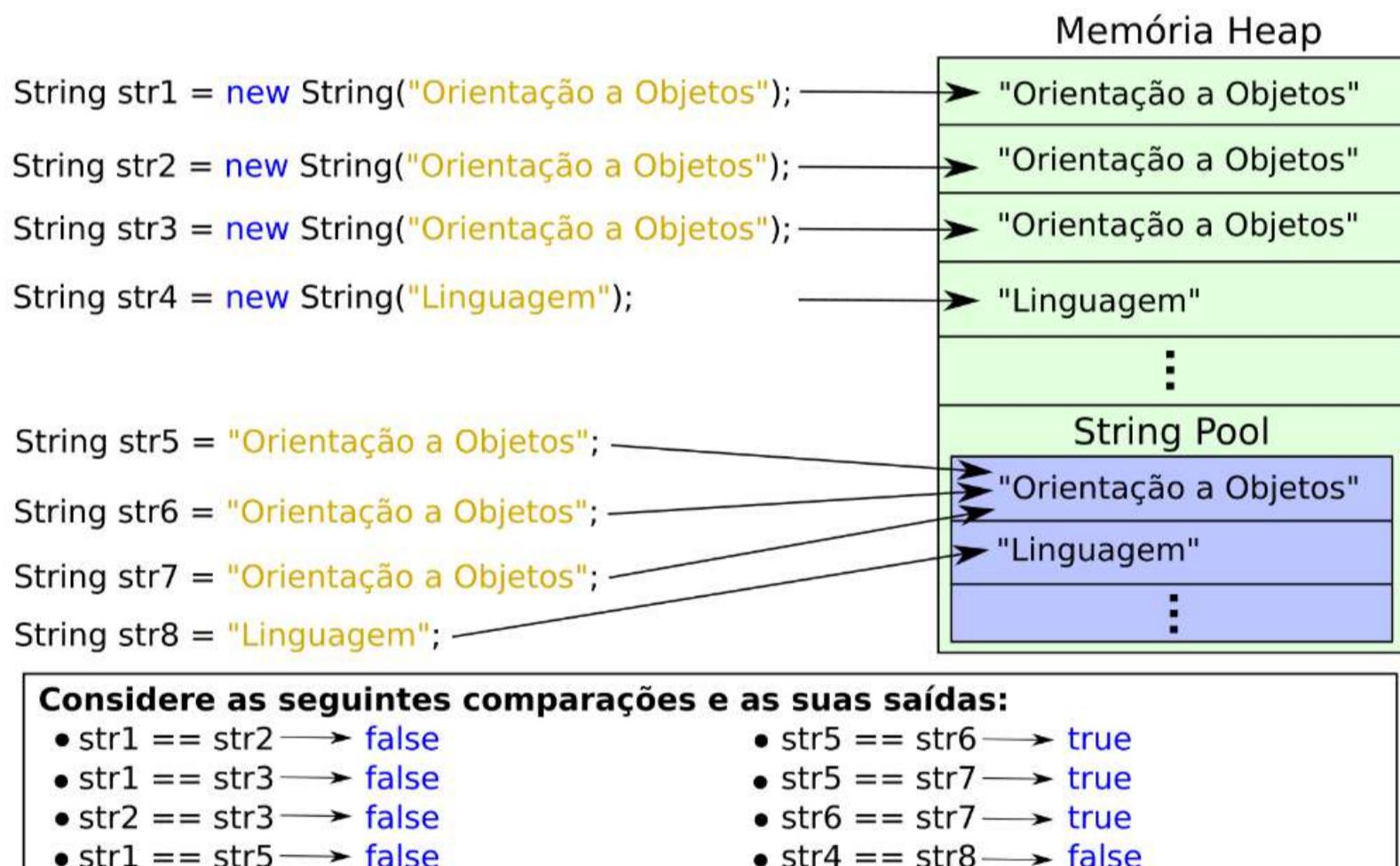
No Código 4.5, na linha 1, vemos a primeira forma de criação de uma *string*, em que o objeto `str1` aponta para um objeto do tipo `String` com a mensagem “Orientação a objetos”, que é armazenada em uma região de memória chamada Heap (pilha). Na linha 2, vemos a segunda forma de criação de *strings* feita diretamente, sem a utilização do construtor, e essa segunda forma tem um funcionamento diferente em relação à primeira, pois o objeto `str2` aponta para o literal `String` com a mensagem “Orientação a objetos”, que é armazenada em uma região especial da memória, chamada String Pool. Já na linha 3, temos a construção de um objeto do tipo `StringBuilder`, que coloca esse objeto na memória *heap*; na linha 4, destacamos que não é possível a criação de objetos `StringBuilder` com atribuição direta de valores literais; na linha 5, temos a construção de um objeto

do tipo `StringBuffer`, que aloca esse objeto na memória *heap*; por fim, na linha 6, destacamos que também não é possível a criação de objetos `StringBuffer` com atribuição direta de valores literais.

Como forma de entender melhor a ideia por trás das memórias *Heap* e *String Pool*, analise a Figura 4.4 a seguir. Repare que toda vez que criamos um objeto com o operador `new`, o objeto é colocado na *heap*, e toda vez que criamos um objeto literal diretamente (sem o `new`), este é colocado na *string pool*. A vantagem da *string pool* sobre a *heap* é que ela economiza gastos com memória.

Ainda nessa figura, analise também as saídas das comparações usando o operador de igualdade (`==`), que compara o endereço de memória dos objetos; isso explica o porquê da comparação `str1 == str2` ser falsa e o porquê da comparação `str5 == str6` ser verdadeira. Para o caso de querer comparar o conteúdo de duas *strings*, utilize o método `equals` e não o operador de igualdade (`==`).

Figura 4.4 | Forma de armazenamento de *strings* na memória *Heap* e *String Pool*



Fonte: elaborada pelo autor.

DICA

Uma dica muito relevante é observar as documentações das classes `String`, `StringBuilder` e `StringBuffer`. É muito importante ver os métodos disponíveis nessas classes; repare que a maioria deles possui a mesma assinatura.

O Quadro 4.4 nos apresenta uma síntese das principais características das classes `String`, `StringBuilder` e `StringBuffer`.

Quadro 4.4 | Síntese das características das classes String, StringBuilder e StringBuffer

Característica	String	StringBuilder	StringBuffer
Memória Usada	String Pool + Heap	Heap	Heap
Modificável	Imutável	Mutável	Mutável
Sincronizado	Não	Não	Sim
Thread Safe	Não	Não	Sim
Versão do Java	Desde Java 1.0	Desde Java 1.5	Desde Java 1.0
Desempenho Geral	Rápido	Rápido	Lento
Desempenho Concatenação	Muito Lento	Super Rápido	Rápido
Quando Usar?	Usar a classe String quando o conteúdo da <i>string</i> for fixo.	Usar StringBuilder quando o conteúdo da <i>string</i> não for fixo e não for necessário ser <i>thread safe</i> .	Usar StringBuffer quando o conteúdo da <i>string</i> não for fixo e for necessário ser <i>thread safe</i> .

0

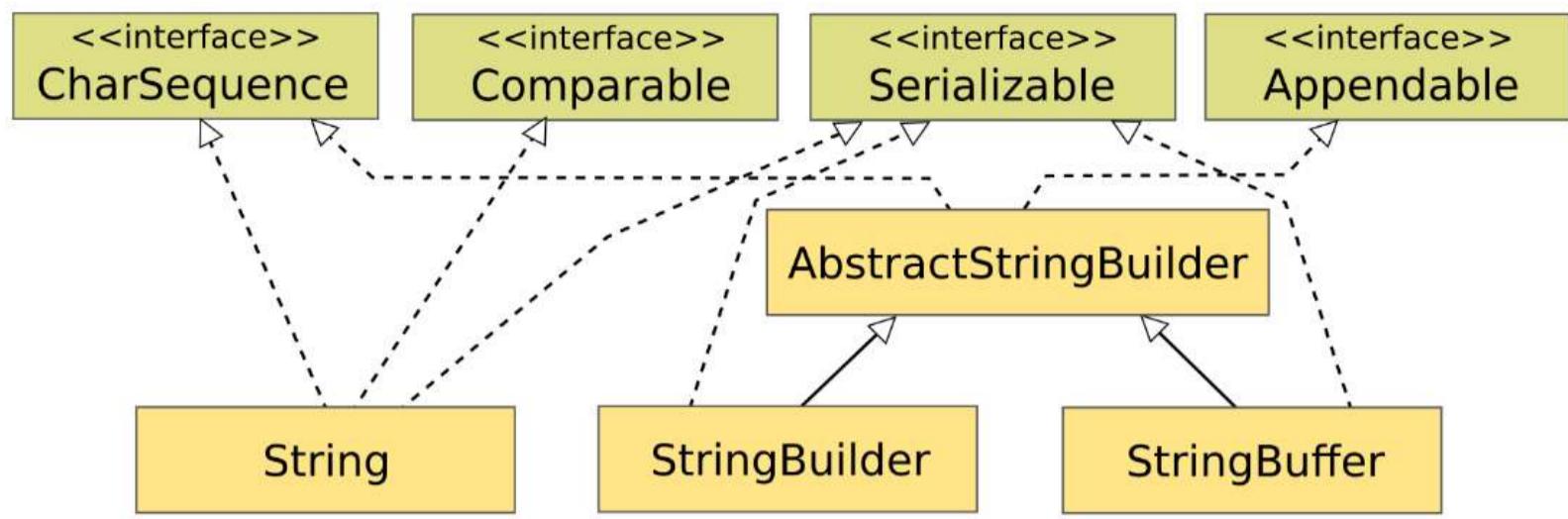
Ver anotações

Fonte: elaborado pelo autor.

Após analisar o Quadro 4.4, perceba que a classe StringBuffer é sincronizada, ou seja, é *thread safe*, isso significa que duas *threads* não podem chamar métodos da classe StringBuffer simultaneamente. Já as classes String e StringBuilder são não sincronizadas, ou seja, não são *thread safe*, isso indica que duas *threads* podem chamar métodos dessas classes simultaneamente. Por hora, pense que uma *thread* é um fluxo de execução da sua aplicação em um determinado núcleo do seu processador (na terceira seção desta unidade estudaremos mais sobre as *threads*).

A Figura 4.5 nos mostra a organização hierárquica das classes que manipulam *strings* em Java.

Figura 4.5 | Organização hierárquica das classes em Java que manipulam *strings*



0

Ver anotações

Fonte: elaborada pelo autor.

Uma recomendação é que você dê uma olhada nas documentações das classes e interfaces mostradas na Figura 4.5, principalmente nas interfaces CharSequence e Appendable. A interface CharSequence representa uma sequência de caracteres; repare que muitos métodos das classes String, StringBuilder e StringBuffer aceitam como parâmetros objetos do tipo CharSequence, e isso garante uma maior interoperabilidade com outras classes que manipulam literais que implementam a interface CharSequence. Já a interface Appendable garante às classes que a implementam a capacidade de acrescentarem novos valores literais aos seus conteúdos.

REFLITA

Conforme vimos na seção, a classe String é imutável e as classes StringBuilder e StringBuffer são mutáveis. Analise a Figura 4.5 e reflita sobre o porquê de a classe String não implementar a interface Appendable (Anexável), bem como o porquê de as classes StringBuilder e StringBuffer implementarem a interface Appendable. Dica: pense em termos que você aprendeu sobre mutabilidade.

Caro estudante, nesta seção você estudou os conteúdos relacionados a vetores, matrizes, classes String, StringBuilder e StringBuffer, bem como analisou diversos exemplos sobre como esses conceitos são implementados na linguagem Java. Lembre-se de que todos os códigos aqui mostrados podem ser acessados no GitHub do autor.

Na seção seguinte, estudaremos como criar uma aplicação que interage com um banco de dados, avançando ainda mais no entendimento da linguagem Java.

REFERÊNCIAS

ARANTES, J. da S. **Livro-POO-Java**. 2020. Disponível em: <https://bit.ly/3eiUMcE>.

Acesso em: 8 set. 2020.

CAELUM. **Java e orientação a objetos**: apostila do curso FJ-11. [s.d.]. Disponível em: <https://bit.ly/2ZpTqrZ>. Acesso em: 8 set. 2020.

DEITEL, P. J.; DEITEL, H. M. **Java**: como programar. 10. ed. São Paulo: Pearson Education, 2016.

JAVATPOINT. **Difference between StringBuffer and StringBuilder**. [s.d.].

Disponível em: <https://bit.ly/2EeUFTM>. Acesso em: 8 set. 2020.

LOIANE GRONER. **Curso de Java básico gratuito com certificado e fórum**. 2016.

Disponível em: <https://bit.ly/2VUtDGT>. Acesso em: 8 set. 2020.

LOIANE GRONER. **Curso de Java Módulo 2**: intermediário. Disponível

em: <https://bit.ly/2Fv0uNz>. Acesso em: 8 set. 2020.

ORACLE. **String**. [s.d.]. Disponível em: <https://bit.ly/2FNIHka>. Acesso em: 8 set.

2020.

ORACLE. **StringBuffer**. [s.d.]. Disponível em: <https://bit.ly/35KYBxD>. Acesso em: 8

set. 2020.

ORACLE. **StringBuilder**. [s.d.]. Disponível em: <https://bit.ly/2ROZA0x>. Acesso em: 8

set. 2020.

FOCO NO MERCADO DE TRABALHO

ARRAYS E STRINGS EM JAVA

Jesimar da Silva Arantes

0

Ver anotações

CRIAÇÃO DE ARRAYS

Criação de *arrays* unidimensional e multidimensional para colocação de caixas, e restrição da região de navegação do robô.



Fonte: Shutterstock.

Deseja ouvir este material?

Áudio disponível no material digital.

SEM MEDO DE ERRAR

A *startup* em que você trabalha lhe passou algumas tarefas:

- Criar um *array* unidimensional para colocar as caixas de livros e impressoras.
- Criar um *array* multidimensional para colocar caixas de HDs.
- Restringir a região de navegação do robô, de forma que ele não passe sobre as caixas, e a região de operação de máquinas.

Após revisar as sugestões de seu gestor, você, então, partindo do Código 3.29 que havia desenvolvido, decidiu, em primeiro lugar, alterar a classe AppGUI.

O Código 4.6 destaca as principais alterações feitas. Nessa classe, você criou dois vetores que armazenarão as imagens das caixas de livros e impressoras (linhas 3 e 4), uma matriz bidimensional que armazenará as imagens das caixas de HDs (linha 5) e, de forma semelhante, os objetos que conterão os dados das caixas, sendo dois vetores e uma matriz (linhas 6 a 8). Nas linhas 12 a 39, temos a instanciação de cada uma das caixas e de cada uma das imagens com suas respectivas localizações; por fim, nas linhas 56 a 62, temos a chamada para o método que avalia a posição do robô, permitindo que ele só se mova para regiões livres de obstáculos.

o

[Ver anotações](#)

Após esses passos, o seu programa ficou pronto, com todas as alterações requisitadas pelo seu chefe, semelhante ao Código 4.6. (Repare que diversas partes foram omitidas; o código completo pode ser acessado no GitHub do autor.)

Código 4.6 | Novas modificações na modelagem das classes AppGUI e Robo

0

[Ver anotações](#)

```
1 public class AppGUI extends Application {  
2     //código omitido  
3     private final ImageView viewBoxLvr[] = new ImageView[3];  
4     private final ImageView viewBoxPrt[] = new ImageView[3];  
5     private final ImageView viewBoxHd[][] = new ImageView[2][3];  
6     private final Caixa caixaLvr[] = new Caixa[3];  
7     private final Caixa caixaPrt[] = new Caixa[3];  
8     private final Caixa caixaHd[][] = new Caixa[2][3];  
9     @Override  
10    public void start(Stage janela) {  
11        //código omitido  
12        for (int i = 0; i < caixaLvr.length; i++) {  
13            caixaLvr[i] = new Caixa("Caixa de Livros", 15,  
14                                25, 50 + i * 50, 30, 0.40f, 0.40f, 0.30f);  
15            viewBoxLvr[i] = new ImageView(imgBoxLvr);  
16            viewBoxLvr[i].setTranslateX(caixaLvr[i].getPosX());  
17            viewBoxLvr[i].setTranslateY(caixaLvr[i].getPosY());  
18        }  
19        for (int i = 0; i < caixaPrt.length; i++) {  
20            caixaPrt[i] = new Caixa("Caixa de Impressoras", 8,  
21                                525, 50 + i * 50, 40, 0.60f, 0.60f, 0.40f);  
22            viewBoxPrt[i] = new ImageView(imgBoxPrt);  
23            viewBoxPrt[i].setTranslateX(caixaPrt[i].getPosX());  
24            viewBoxPrt[i].setTranslateY(caixaPrt[i].getPosY());  
25        }  
26        for (int i = 0; i < caixaHd.length; i++) {  
27            for (int j = 0; j < caixaHd[i].length; j++) {  
28                if (i == 0) {  
29                    caixaHd[i][j] = new Caixa("Caixa de HDs", 20,  
30                                230 + j * 50, 50, 40, 0.50f, 0.50f, 0.30f);  
31                } else {  
32                    caixaHd[i][j] = new Caixa("Caixa de HDs", 20,  
33                                230 + j * 50, 160, 40, 0.50f, 0.50f, 0.30f);  
34                }  
35                viewBoxHd[i][j] = new ImageView(imgBoxHd);  
36                viewBoxHd[i][j].setTranslateX(caixaHd[i][j].getPosX());  
37                viewBoxHd[i][j].setTranslateY(caixaHd[i][j].getPosY());  
38            }  
39        }  
40        Group grupos = new Group();
```

0

Ver anotações

```
41     for (int i = 0; i < viewBoxLvr.length; i++) {
42         grupo.getChildren().add(viewBoxLvr[i]);
43     }
44     for (int i = 0; i < viewBoxPrt.length; i++) {
45         grupo.getChildren().add(viewBoxPrt[i]);
46     }
47     for (int i = 0; i < viewBoxHd.length; i++) {
48         for (int j = 0; j < viewBoxHd[i].length; j++) {
49             grupo.getChildren().add(viewBoxHd[i][j]);
50         }
51     }
52     //código omitido
53     cena.setOnKeyPressed((evt) -> {
54         if (evt.getCode() == KeyCode.UP) {
55             viewRobo.setImage(imgRoboCostas);
56             if (robo.avaliaPos(robo.getPosicaoX(),
57                     robo.getPosicaoY() - (int)robo.getVel())) {
58                 robo.setPosicaoY(
59                     robo.getPosicaoY() - (int)robo.getVel());
60                 viewRobo.setTranslateX(robo.getPosicaoX());
61                 viewRobo.setTranslateY(robo.getPosicaoY());
62             }
63         }
64         //código omitido
65     });
66 }
67 }
68 public class Robo extends RoboIdeia {
69     //código omitido
70     public boolean avaliaPos(int posX, int posY){
71         posX = posX + largura/2;
72         posY = posY + altura/2;
73         if (posX<40 || posX>560 || posY<30 || posY>360) {
74             return false; //delimitação da fronteira da sala
75         }
76         if ((posX>=170 && posX<=430) && (posY>=240 && posY<=400)){
77             return false; //delimitação região operação máquinas
78         }
79         if ((posX>=0 && posX<=100) && (posY>=0 && posY<=200)) {
80             return false; //delimitação regiao caixas livros
81     }
```

```
82     if ((posX>=500 && posX<=600) && ( posY>=0 && posY<=200)) {  
83         return false; //delimitação região caixas impressoras  
84     }  
85     if ((posX>=170 && posX<=430) && ( posY>=0 && posY<=90)) {  
86         return false; //delimitação região caixas com HDs  
87     }  
88     if ((posX>=170 && posX<=430) && ( posY>=120 && posY<=200)){  
89         return false; //delimitação região caixas com HDs  
90     }  
91     return true;  
92 }  
93 }
```

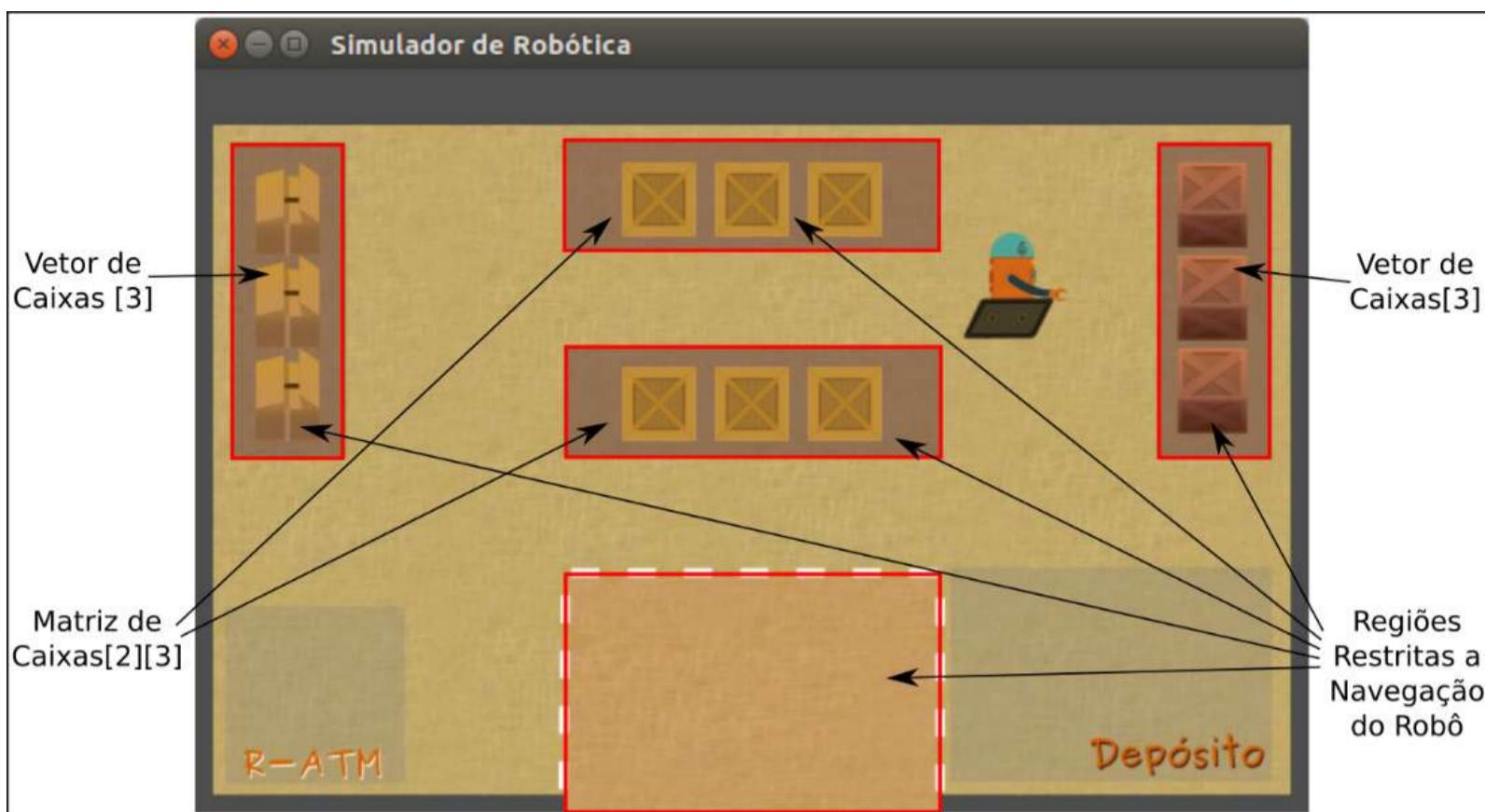
0

Ver anotações

Fonte: elaborado pelo autor.

Ao executar o Código 4.6, uma tela semelhante à Figura 4.6 deverá ser mostrada. Repare que conseguimos reproduzir um simulador bem parecido com o construído utilizando a ferramenta Greenfoot mostrada na seção 3 da unidade 1.

Figura 4.6 | Interface gráfica do simulador de robótica



Fonte: elaborada pelo autor.

BANCO DE DADOS RELACIONAL E NOSQL

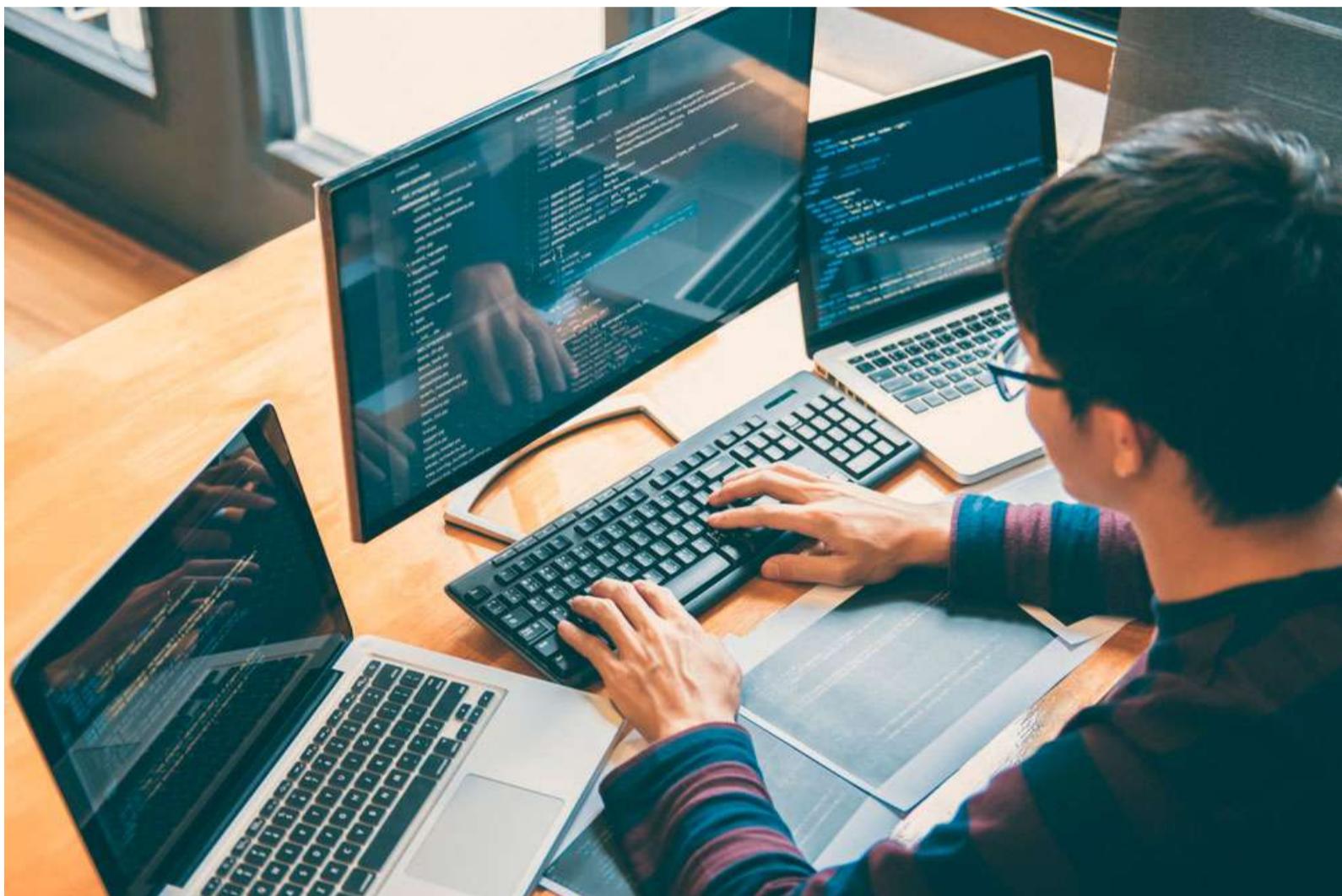
Jesimar da Silva Arantes

0

[Ver anotações](#)

APLICAÇÕES COM INTERAÇÃO COM BANCO DE DADOS

As empresas necessitam de aplicações com mecanismos de armazenamento de dados, sejam simples como leitura de texto (.txt), manipulação de arquivos CSV ou integração com banco de dados mais poderoso.



Fonte: Shutterstock.

Deseja ouvir este material?

Áudio disponível no material digital.

PRATICAR PARA APRENDER

Prezado estudante, bem-vindo à segunda seção da quarta e última unidade de estudos sobre Linguagem Orientada a Objetos. Qual empresa de *software* nunca precisou de uma aplicação que armazenasse os dados de clientes ou fornecedores em um Banco de Dados (BD). A maioria das empresas já precisou e ainda precisa, em algum nível, trabalhar com BD. Nesta seção, abordaremos de forma simplificada o assunto banco de dados, visto que é um tema complexo e extremamente abrangente, sendo necessário um curso para que todo o seu

conteúdo seja estudado. Ainda nesta seção, estudaremos como ler arquivos de texto e arquivos CSV em Java, que podem ser entendidos como mecanismos de armazenamento de dados mais simples, porém não menos importantes.

Para a contextualização de sua aprendizagem, lembre-se de que você está trabalhando em um simulador de robô e que seu gestor reviu o seu código e percebeu que você não está utilizando nenhum tipo de banco de dados para armazenar a rota seguida pelo robô.

Diante disso, foi pedido a você que criasse um banco de dados relacional e armazenasse nesse banco as informações da rota seguida pelo robô, bem como que instalasse o pacote do XAMPP, que vem com um software chamado phpMyAdmin, para administrar o BD MySQL ou MariaDB. Com isso, o seu gestor deseja que você integre a sua aplicação em Java ao BD criado e, a partir dessa aplicação, realize as primeiras *queries* (consultas) com o BD.

Diante do desafio que lhe foi apresentado, como você criará esse Banco de Dados? O que é um banco de dados relacional? O que é o XAMPP? O que é o phpMyAdmin? Como integrar o Java ao MySQL? Como criar as primeiras *queries* para o BD?

Agora que você foi apresentado à sua nova situação-problema, estude-a e aprenda um pouco sobre o banco de dados MySQL e como integrá-lo à linguagem Java.

E aí, vamos juntos compreender esses conceitos e resolver esse desafio?

Bom estudo!

CONCEITO-CHAVE

A presente seção tem por objetivo criar aplicações que interajam com o Banco de Dados (BD) utilizando-se Java. Um BD pode ser pensado como uma forma de armazenar e consultar dados, assim, um simples arquivo pode ser entendido como uma forma primitiva/simples de se manter os dados. Dessa maneira, antes de criarmos uma aplicação com um BD mais poderoso, como o MySQL, mostraremos, em primeiro lugar, como se ler um arquivo de texto (.txt) e, em seguida, manipular um arquivo CSV (*Comma-Separated Values* ou Valores Separados por Vírgula).

| LEITURA DE ARQUIVOS DE TEXTO

Suponha que desejamos construir uma aplicação que realize a leitura e a impressão de um simples arquivo de texto em Java; o Código 4.7 nos mostra como construir essa aplicação. A execução desse código pressupõe que existe um arquivo chamado “poema.txt” dentro da pasta “dados”.

0

[Ver anotações](#)

Código 4.7 | Aplicação que realiza a leitura de um arquivo de texto e imprime o resultado

```
1 import java.io.File;
2 import java.io.FileNotFoundException;
3 import java.util.Scanner;
4 public class ManipulaArquivosTexto {
5     public static void main(String[] args) {
6         try {
7             File arquivo = new File("dados/poema.txt");
8             Scanner scanner = new Scanner(arquivo);
9             while (scanner.hasNextLine()) {
10                 String linha = scanner.nextLine();
11                 System.out.printf("%s%n", linha);
12             }
13         } catch (FileNotFoundException ex) {
14             System.out.printf("Erro abertura do arquivo: %s.%n",
15                             ex.getMessage());
16         }
17     }
18 }
```

0

[Ver anotações](#)

Fonte: elaborado pelo autor.

No Código 4.7, nas linhas 1 a 3, temos a importação das classes que manipulam arquivos; já na linha 7, um objeto do tipo File foi criado em que o caminho relativo do arquivo é passado como argumento no construtor da classe; na linha 8, foi criado um objeto do tipo Scanner, responsável pela leitura do arquivo; na linha 9, um loop será iniciado enquanto existir uma próxima linha no arquivo de texto; nas linhas 10 e 11, uma linha inteira é lida e em seguida impressa na tela; por fim, as linhas 13 a 16 realizam o tratamento de exceção do tipo FileNotFoundException, que pode ser lançada em caso de o arquivo não existir.

DICA

Caro aluno, existem outras formas de se fazer a leitura de um arquivo em Java, logo, sugerimos que pesquise na internet como fazer isso utilizando a classe BufferedReader e quais as diferenças entre as classes Scanner e BufferedReader.

LEITURA DE ARQUIVOS CSV

Outro tipo de arquivo muito utilizado na área de computação é o arquivo CSV, que possui um conjunto de valores separados por vírgula ou algum outro delimitador em sua organização, como o ponto e vírgula, a tabulação (tab), o espaço, entre outros. Um arquivo CSV é um tipo de arquivo de texto especial, que pode ser aberto por qualquer editor de planilha eletrônica, como Excel, Calc, Gnumeric, entre outros.

Ver anotações

Experimente abrir um editor de texto simples e digitar o seguinte conteúdo:

```
Nome, Idade, Profissão, Cidade, Estado
Adriano, 56, Escritor, Rio de Janeiro, Rio de Janeiro
Barbara, 34, Médica, Belo Horizonte, Minas Gerais
João, 18, Estudante, São Paulo, São Paulo
Márcio, 22, Jornalista, São Carlos, São Paulo
Renata, 37, Professora, Florianópolis, Santa Catarina
```

Feito isso, experimente salvar o arquivo com o nome “dadosclientes.csv” (vamos considerar que os dados digitados representam informações sobre diversos clientes de uma empresa) e, em seguida, abra o conteúdo desse arquivo utilizando algum editor de planilha eletrônica. Após aberto, um conteúdo semelhante à Figura 4.7 será mostrado:

Figura 4.7 | Visualização de um arquivo CSV aberto com um editor de planilha eletrônica

	A	B	C	D	E
1	Nome	Idade	Profissão	Cidade	Estado
2	Adriano	56	Escritor	Rio de Janeiro	Rio de Janeiro
3	Barbara	34	Médica	Belo Horizonte	Minas Gerais
4	João	18	Estudante	São Paulo	São Paulo
5	Márcio	22	Jornalista	São Carlos	São Paulo
6	Renata	37	Professora	Florianópolis	Santa Catarina

Fonte: elaborada pelo autor.

Repare que o conteúdo foi separado em linhas e colunas. As linhas são delimitadas pelas quebras de linhas no arquivo .csv, já as colunas são delimitadas pelas vírgulas (,) colocadas no arquivo. Os arquivos CSV são muito utilizados em cursos

da área de computação ou engenharias para auxiliar na organização de diversos tipos de dados.

DICA

Como dito anteriormente, outros separadores podem ser utilizados, logo, experimente trocar a vírgula (,) pelo separador ponto e vírgula (;) ou, ainda, pelo separador tabulação; em seguida, abra o conteúdo salvo em um editor de planilha eletrônica e veja o resultado.

Imagine que, agora, desejamos criar um programa em Java que realize a leitura desse arquivo .csv. Após essa leitura, por sua vez, os dados dos clientes devem ser armazenados em uma estrutura de lista e, por fim, impressos na tela. O Código 4.8 nos mostra como tal aplicação pode ser implementada em Java.

Código 4.8 | Aplicação que realiza a leitura de um arquivo CSV e imprime o resultado

0

[Ver anotações](#)

```
1 import java.io.File;
2 import java.io.FileNotFoundException;
3 import java.util.ArrayList;
4 import java.util.List;
5 import java.util.Scanner;
6 public class ManipulaArquivosCSV {
7     public final String DELIMITADOR_PONTO_VIRGULA = ";" ;
8     public final String DELIMITADOR_VIRGULA = "," ;
9     public static void main(String[] args) {
10         ManipulaArquivosCSV csv = new ManipulaArquivosCSV();
11         List<List<String>> registroDados = csv.leitura();
12         csv.imprimeDados(registroDados);
13     }
14     public List<List<String>> leitura() {
15         List<List<String>> registroDados = new ArrayList<>();
16         try {
17             File arquivo = new File("dados/dadosclientes.csv");
18             Scanner sc = new Scanner(arquivo);
19             while (sc.hasNextLine()) {
20                 String linha = sc.nextLine();
21                 registroDados.add(getRegistroDaLinha(linha));
22             }
23         } catch (FileNotFoundException ex) {
24             System.out.printf("Erro abertura do arquivo: %s.%n",
25                               ex.getMessage());
26             System.exit(0);
27         }
28         return registroDados;
29     }
30     private List<String> getRegistroDaLinha(String linha) {
31         List<String> listValores = new ArrayList<String>();
32         try (Scanner linhaScanner = new Scanner(linha)) {
33             linhaScanner.useDelimiter(DELIMITADOR_PONTO_VIRGULA);
34             while (linhaScanner.hasNext()) {
35                 listValores.add(linhaScanner.next());
36             }
37         }
38         return listValores;
39     }
40     private void imprimeDados(List<List<String>> registroDados){
```

0

Ver anotações

```
41     for (List<String> lista : registroDados){  
42         for (String elemento : lista){  
43             System.out.printf(" |%15s ", elemento);  
44         }  
45         System.out.println(" |");  
46     }  
47 }  
48 }
```

0

[Ver anotações](#)

Fonte: elaborado pelo autor.

Nas linhas 1 a 5, do Código 4.8, temos as importações das classes utilizadas; nas linhas 7 e 8, foram declaradas duas *strings* que representam os delimitadores que podem ser utilizados; nas linhas 9 a 13, o método *main* foi criado; já nas linhas 14 a 29, um método que realiza a leitura foi criado. (Algumas linhas desse código são complexas, como a ideia de se criar uma lista de listas, porém a intenção, aqui, é dar uma visão geral de como ler esse tipo de arquivo e armazenar os dados.) Nas linhas 30 a 39, uma linha do arquivo é passada, então, retorna-se uma lista com os campos de dados. Por fim, nas linhas 40 a 47, dá-se a impressão dos dados.

A intenção desse exemplo é apresentar os arquivos CSV, que armazenam os dados em tabelas que são organizadas em linhas e colunas semelhantes a um Banco de Dados, como o MySQL, que será estudado a seguir. Assim, na Figura 4.7, na primeira linha do quadro, temos os atributos de cada coluna, que são: Nome, Idade, Profissão, Cidade e Estado; em breve, aprenderemos como construir esse mesmo exemplo em um BD como o MySQL.

BANCO DE DADOS

Até aqui, dizemos que arquivos de texto e arquivos CSVs podem ser entendidos como BD. De forma a avançarmos no estudo, vamos, agora, estudar bancos de dados classificados como relacionais, e podemos destacar o MySQL, no entanto, existem diversos outros BD que se encaixam nessa categoria, como: MariaDB, PostgreSQL, Oracle, Microsoft SQL Server, IBM Db2, SQLite, entre outros. Um BD como o MySQL funciona bem para aplicações que necessitam de dados estruturados, como os dados de clientes mostrados no exemplo anterior, que possuía campos como nome, idade, profissão, cidade e estado.

Os bancos de dados, geralmente, utilizam uma linguagem de consulta, e a principal linguagem de consulta utilizada é a SQL (*Structured Query Language* ou Linguagem de Consulta Estruturada), que é uma linguagem padrão para armazenamento, manipulação e recuperação de dados em BD, que pode ser utilizada para trabalhar com diversos BDs, como: MySQL, MariaDB, Microsoft SQL Server, Oracle, PostgreSQL, entre outros.

Existem quatro operações fundamentais sobre um BD, conhecidas como CRUD (*Create, Read, Update, Delete*), que indica as operações de criação, consulta, atualização e destruição de dados.

o

[Ver anotações](#)

MYSQL E MARIADB

Antes de passarmos para a parte prática do livro, precisamos configurar o nosso ambiente de trabalho. Dessa maneira, é necessário que você instale o MySQL em seu computador; para isso, existem diversas maneiras de proceder, e uma delas é instalar os pacotes XAMPP (Apache + MySQL + PHP + Perl) ou AMPPS (Apache + MySQL + PHP + Perl + Python). O XAMPP pode ser encontrado no site Apache Friends e possui versão de instalação para Windows, Linux e Mac OS X. Já o AMPPS pode ser encontrado no site AMPPS e também possui versão de instalação nesses três sistemas operacionais. Além disso, gostaríamos de ressaltar que esses pacotes têm trocado a versão do BD MySQL pelo MariaDB, e isso ocorreu após a empresa Oracle adquirir o MySQL, assim, caso queira, sinta-se seguro para instalar o MySQL ou, ainda, o MariaDB, que são dois pacotes de BDs que têm uma grande compatibilidade entre si. Adicionalmente, um dos softwares instalados com o XAMPP ou com o AMPPS é o phpMyAdmin.

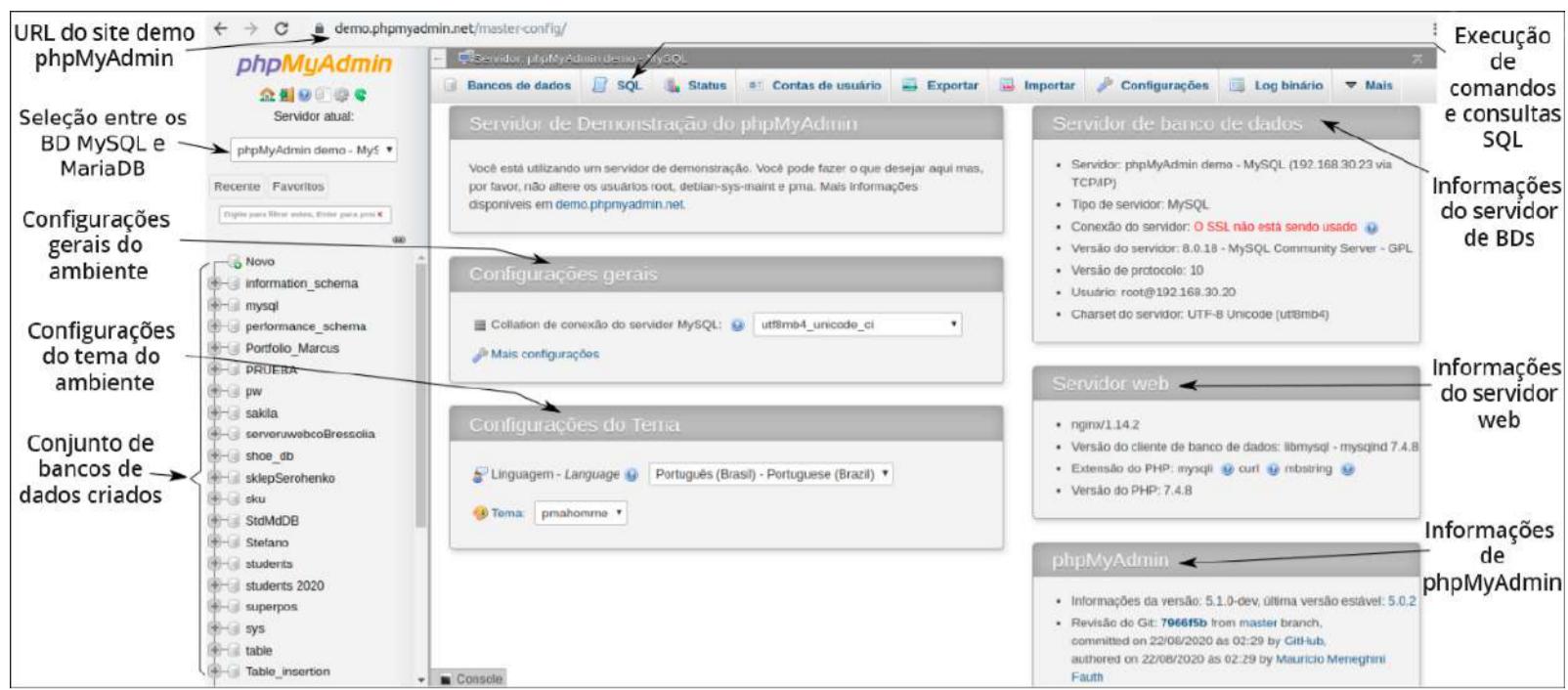
ATENÇÃO

Gostaríamos de destacar também que essas são apenas sugestões de ferramentas para o trabalho, pois existem muitas outras; assim, caso queira utilizar outras ferramentas, como MySQL Workbench ou Database Workbench, fique à vontade.

PHPMYADMIN

O phpMyAdmin é uma ferramenta de software livre, escrita em PHP, que tem por objetivo a administração do BD MySQL ou MariaDB na web. Assim, o phpMyAdmin suporta um conjunto de operações sobre esses bancos, como fazer o gerenciamento de BD, criar tabelas, colunas, estabelecer relações, etc. Essa ferramenta permite que você faça isso por meio da interface gráfica do usuário (GUI) ou, ainda, executando instruções em SQL. Uma versão de utilização de demonstração do phpMyAdmin pode ser acessada em [phpMyAdmin.net.](http://phpmyadmin.net/); sugerimos que acesse esse endereço e conheça um pouco desse ambiente. Ao acessá-lo, uma tela semelhante à Figura 4.8 deve aparecer.

Figura 4.8 | Tela do ambiente de administração de banco de dados phpMyAdmin



Fonte: captura de tela do phpMyAdmin elaborada pelo autor.

A Figura 4.8 destaca algumas das principais informações disponíveis do phpMyAdmin. Repare que é possível alterar as versões do BD utilizado, como MySQL e MariaDB (na parte esquerda da tela), e que todos os BDs criados ficam listados à esquerda.

CRIAÇÃO DO BD

Frente a isso, gostaríamos de convidá-lo a clicar no menu SQL para inserir os primeiros comandos relacionados a um BD. Sendo assim, digite o comando mostrado no Código 4.9.

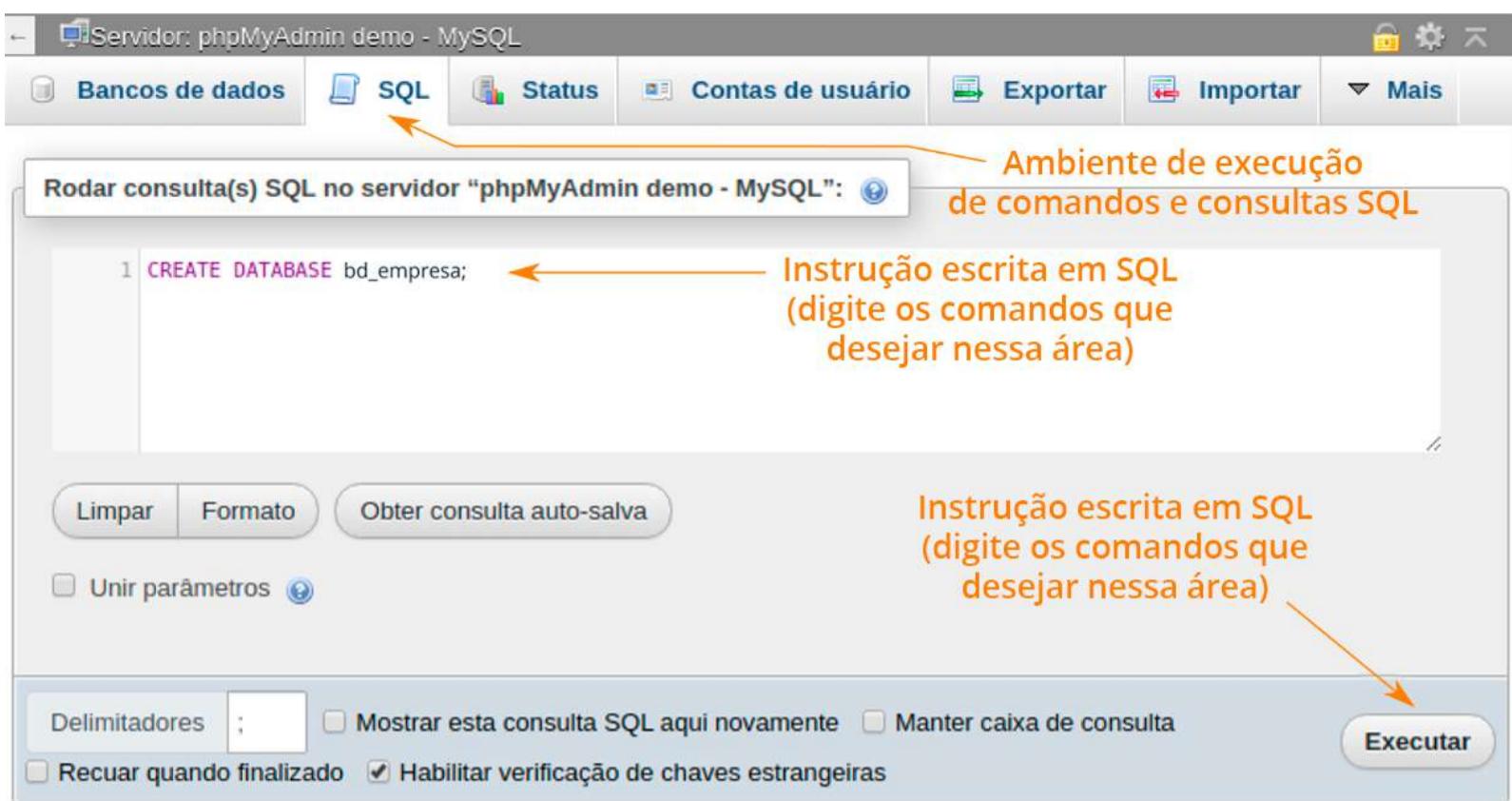
Código 4.9 | Comando para criar um banco de dados chamado bd_empresa

```
CREATE DATABASE `bd_empresa`;
```

Fonte: elaborado pelo autor.

Após digitar, clique no botão Executar; esse comando criará um Banco de Dados chamado bd_empresa. Observe a Figura 4.9 para entender melhor os passos que devem ser executados. A partir de agora, todos os comandos SQL que forem pedidos para ser digitados, serão feitos nesse campo. Ao clicar no botão Executar, o comando SQL será processado pelo sistema gerenciador de banco de dados.

Figura 4.9 | Tela do ambiente de escrita de comandos em SQL no phpMyAdmin



Fonte: captura de tela do phpMyAdmin - MySQL elaborada pelo autor.

CRIAÇÃO DA TABELA DO BD

Vamos, agora, criar uma tabela nesse banco de dados com os seguintes campos: **nome, idade, profissao, cidade, estado** e **codigo**. Para isso, digite o comando mostrado no Código 4.10.

Código 4.10 | Comando para se criar uma tabela chamada clientes no BD bd_empresa

```

1 CREATE TABLE `bd_empresa`.`clientes`(
2     `nome` VARCHAR(100) DEFAULT NULL,
3     `idade` INT(4) DEFAULT NULL,
4     `profissao` VARCHAR(100) DEFAULT NULL,
5     `cidade` VARCHAR(100) DEFAULT NULL,
6     `estado` VARCHAR(100) DEFAULT NULL,
7     `codigo` INT(10) NOT NULL AUTO_INCREMENT,
8     PRIMARY KEY(`codigo`)
9 ) ENGINE = INNODB AUTO_INCREMENT = 28 DEFAULT CHARSET = latin1;

```

Fonte: elaborado pelo autor.

Após clicar em executar, uma tabela chamada clientes será criada dentro do BD bd_empresa.

ASSIMILE

No MySQL, existem dois tipos principais de dados, que são: numéricos e *strings*.

Dados String:

- CHAR(size): representa uma *string* de comprimento fixa. O parâmetro *size* pode ir de 0 a 255.
- VARCHAR(size): representa uma *string* de comprimento variável. O parâmetro *size* pode ir de 0 a 65535.
- TEXT(size): contém uma *string* com comprimento máximo de 65535 bytes.

Dados Numéricos:

- INT(size) ou INTEGER(size): representa um número inteiro. O número fica no intervalo -2.147.483.648 a 2.147.483.647.
- DEC(size, d) ou DECIMAL(size, d): representa um número em ponto flutuante.
- BOOL ou BOOLEAN: representa valores lógicos. O número 0 (zero) é considerado falso e os valores não zero são considerados verdadeiros.

Uma descrição mais completa sobre os tipos de dados em MySQL pode ser encontrada no site W3Schools.

INSCRIÇÃO DE DADOS NA TABELA

Agora, é preciso popularmos essa tabela, ou seja, inserirmos os clientes. Neste exemplo, vamos inserir cinco clientes, com os mesmos valores da Figura 4.7, a partir do comando mostrado no Código 4.11.

Código 4.11 | Comando para inserção de fregueses dentro da tabela clientes

```

1  INSERT INTO `bd_empresa`.`clientes`(
2      `nome`, `idade`, `profissao`, `cidade`, `estado`
3  )
4  VALUES(
5      'Adriano', 56, 'Escritor', 'Rio de Janeiro', 'Rio de Janeiro'
6  ),(
7      'Barbara', 34, 'Médica', 'Belo Horizonte', 'Minas Gerais'
8  ),(
9      'João', 18, 'Estudante', 'São Paulo', 'São Paulo'
10 ),(
11     'Márcio', 22, 'Jornalista', 'São Carlos', 'São Paulo'
12 ),(
13     'Renata', 37, 'Professora', 'Florianópolis', 'Santa Catarina'
14 );

```

Ver anotações 0

Fonte: elaborado pelo autor.

Inserido o código, clique em executar e pronto! A tabela clientes contém alguns fregueses, que são: Adriano, Barbara, João, Márcio e Renata. A Figura 4.10 nos mostra como aparecerá a tabela clientes clicando-se no menu Visualizar do phpMyAdmin.

Figura 4.10 | Tela do ambiente de visualização de tabelas no phpMyAdmin

	nome	idade	profissao	cidade	estado	codigo
<input type="checkbox"/>	Adriano	56	Escritor	Rio de Janeiro	Rio de Janeiro	28
<input type="checkbox"/>	Barbara	34	Médica	Belo Horizonte	Minas Gerais	29
<input type="checkbox"/>	João	18	Estudante	São Paulo	São Paulo	30
<input type="checkbox"/>	Márcio	22	Jornalista	São Carlos	São Paulo	31
<input type="checkbox"/>	Renata	37	Professora	Florianópolis	Santa Catarina	32

Fonte: captura de tela do phpMyAdmin elaborada pelo autor.

DICA

O site w3schools possui um excelente conteúdo sobre a linguagem de consulta SQL. Como dica, utilize esse site para conhecer e compreender melhor a sintaxe dos comandos SQL, pois este livro apresentará apenas um pequeno subconjunto dos comandos SQL existentes.

0

Até aqui, tivemos uma noção geral de como se criar um BD, criar tabelas e inserir valores nela. No entanto, esse ambiente web deve ser utilizado apenas para pequenos testes, pois ele tem uma série de limitações, logo, é importante que esteja configurado em seu computador local.

[Ver anotações](#)

Como você já instalou o XAMPP em seu computador, vamos iniciar o phpMyAdmin localmente. Em cada sistema operacional, o procedimento é de um jeito diferente. No computador do autor, o seguinte comando foi executado para se iniciar o XAMPP:

```
$ sudo /opt/lampp/xampp start
```

O comando utilizado para parar a execução do XAMPP nesse computador foi:

```
$ sudo /opt/lampp/xampp stop
```

Aluno, procure iniciar a execução do XAMPP em seu computador, após isso, o servidor Apache e o MySQL serão iniciados, então, o próximo passo será acessar o ambiente do phpMyAdmin localmente. Para isso, acesse o seu navegador de internet favorito e, então, entre no seguinte endereço:

`http://localhost/phpmyadmin`

Após o acesso, uma tela semelhante à Figura 4.8 deverá aparecer (a URL será diferente, pois estará rodando localmente). Repita os procedimentos realizados nos Códigos 4.9 a 4.11 para criar um Banco de Dados, uma tabela e popular o BD localmente; feito isso, vamos construir uma aplicação em Java que seja capaz de ler o conteúdo armazenado no BD e inserir novos clientes por meio da própria aplicação Java.

Código 4.12 | Conjunto de classes em Java para interação com BD MySQL

```
1  public class Cliente {  
2      private int codigo;  
3      private String nome;  
4      private int idade;  
5      private String profissao;  
6      private String cidade;  
7      private String estado;  
8      public Cliente(int codigo, String nome, int idade,  
9                      String profissao, String cidade, String estado) {  
10         this.codigo = codigo;  
11         this.nome = nome;  
12         this.idade = idade;  
13         this.profissao = profissao;  
14         this.cidade = cidade;  
15         this.estado = estado;  
16     }  
17 }  
18  
19 import java.sql.Connection;  
20 import java.sql.DriverManager;  
21 import java.sql.SQLException;  
22 public class ConexaoBD {  
23     private Connection con = null;  
24     private final String jdbcDriver = "com.mysql.cj.jdbc.Driver";  
25     private final String prefixoBD = "jdbc:mysql://";  
26     private final String nomeHost = "localhost";  
27     private final String portaBD = "3306";  
28     private final String nomeBD = "bd_empresa";  
29     private final String usuario = "root";  
30     private final String senha = "";  
31     private String url = null;  
32     public ConexaoBD() {  
33         url = prefixoBD + nomeHost + ":" + portaBD + "/" + nomeBD;  
34     }  
35     public Connection getConexao() {  
36         try {  
37             if (con == null) {  
38                 Class.forName(jdbcDriver);  
39                 con=DriverManager.getConnection(url,usuario,senha);  
40             } else if (con.isClosed()) {  
41                 Class.forName(jdbcDriver);  
42                 con=DriverManager.getConnection(url,usuario,senha);  
43             }  
44         } catch (Exception e) {  
45             e.printStackTrace();  
46         }  
47     }  
48 }
```

```
41             con = null;
42
43         }
44     } catch (ClassNotFoundException ex) {
45
46         System.out.println(ex);
47     } catch (SQLException ex) {
48
49         System.out.println(ex);
50     }
51
52     public void fecharConexao() {
53
54         if (con != null) {
55
56             try {
57
58                 con.close();
59             } catch (SQLException ex) {
60
61                 System.out.println(ex);
62             }
63
64         }
65
66     }
67
68     import java.sql.Connection;
69     import java.sql.PreparedStatement;
70     import java.sql.ResultSet;
71     import java.sql.SQLException;
72     import java.sql.Statement;
73
74     public class Consultas {
75
76         public void listarClientes(Connection conn) {
77
78             try {
79
80                 String sql = "SELECT * FROM `clientes`";
81                 PreparedStatement pStat = conn.prepareStatement(sql);
82                 ResultSet result = pStat.executeQuery();
83
84                 System.out.println("Resultados da Consulta: ");
85
86                 System.out.printf(" |%8s |%16s |%16s |%16s |%16s |%18s |\n",
87
88                     "codigo", "nome", "idade", "profissao",
89
90                     "cidade", "estado");
91
92                 while (result.next()) {
93
94                     String nome = result.getString("nome");
95                     String idade = result.getString("idade");
96                     String profissao = result.getString("profissao");
97
98                     String cidade = result.getString("cidade");
99
100                }
```

```
82             String estado = result.getString("estado");
83             String codigo = result.getString("codigo");
84             System.out.printf(" |%8s|%16s|%16s|%16s|%16s|%18s|%n",
85                               codigo, nome, idade, profissao,
86                               cidade, estado);
87         }
88     } catch (SQLException ex) {
89         System.out.println(ex);
90     }
91 }
92 public void inserirCliente(Connection conn, Cliente cliente){
93     try {
94         Statement st = conn.createStatement();
95         String sql = "INSERT INTO `clientes` (nome, " +
96                         "idade, profissao, cidade, estado) VALUES ('" +
97                         cliente.getNome() + "', '" +
98                         cliente.getIdade() + "', '" +
99                         cliente.getProfissao() + "', '" +
100                        cliente.getCidade() + "', '" +
101                        cliente.getEstado() + "')";
102         st.executeUpdate(sql);
103         st.close();
104     } catch (SQLException ex) {
105         System.out.println(ex);
106     }
107 }
108 }
109
110 import java.sql.Connection;
111 public class App {
112     public static void main(String[] args) {
113         ConexaoBD conBD = new ConexaoBD();
114         Connection connection = conBD.getConexao();
115         Consultas consultas = new Consultas();
116         consultas.listarClientes(connection);
117         Cliente clienteRita = new Cliente(100, "Rita", 27,
118                                         "Dona de casa", "Pelotas", "Rio Grande do Sul");
119         consultas.inserirCliente(connection, clienteRita);
120         Cliente clientePedro = new Cliente(101, "Pedro", 53,
121                                         "Pedreiro", "Salvador", "Bahia");
122         consultas.inserirCliente(connection, clientePedro);
```

0

Ver anotações

```
123     consultas.listarClientes(connection);
124     conBD.fecharConexao();
125 }
126 }
```

0

Fonte: elaborado pelo autor.

Ver anotações

O funcionamento do Código 4.12 necessita da inclusão de uma biblioteca chamada *mysql-connector-java-X.Y.Z.jar*, em que X, Y e Z determinam a versão da biblioteca. Você pode baixar essa biblioteca em MySQL; para isso, nesse site, haverá uma caixa chamada Sistema Operacional (*Operating System*), então, selecione a opção Plataforma Independente (*Platform Independent*), baixe o arquivo zip e, por fim, inclua o arquivo .jar que estará dentro da pasta baixada em seu projeto do Código 4.12 e veja o resultado.

Vamos, agora, fazer uma breve explicação do Código 4.12. Não entraremos em muitos detalhes, pois os códigos que manipulam BD são relativamente intuitivos. Nas linhas 1 a 17, temos a modelagem de uma classe Cliente usada apenas para armazenar os dados dos clientes; nas linhas 19 a 21, temos algumas importações utilizadas na classe ConexaoBD; nas linhas 23 a 31, temos alguns atributos utilizados para se estabelecer a conexão com o BD (lembre-se de alterar alguns dos atributos, como nomeBD, usuário e senha, conforme as configurações feitas em seu computador); nas linhas 35 a 50, temos um método que é capaz de estabelecer uma conexão com o BD, e será por meio dessa conexão que serão feitas todas as consultas ao banco; nas linhas 51 a 59, por sua vez, temos um método que fecha a conexão com o BD; já nas linhas 62 a 66, temos algumas importações utilizadas na classe chamada Consultas; nas linhas 68 a 91, temos um método que é capaz de listar os clientes a partir de uma conexão com o BD (gostaríamos de destacar que a consulta foi definida por meio do comando SQL “SELECT * FROM `clientes`”, que retorna todos os campos da tabela clientes); nas linhas 92 a 107, temos um método que insere um novo cliente no BD; por fim, nas linhas 110 a 127, definimos o ponto de entrada da aplicação. Aqui, vale destacarmos que, inicialmente, foi listado o conteúdo armazenado no BD, em seguida, foram inseridos dois novos clientes (a Rita e o Pedro) e, por fim, listado novamente o conteúdo do BD.

Reflita sobre o funcionamento do Código 4.12, especificamente sobre as linhas que definem as consultas SQL, ou seja, as linhas 70 e as linhas 95 a 101. Tente entender que essas são as linhas-chaves para listar e inserir clientes, além disso, reflita sobre como ficaria um código em Java para editar e excluir um cliente do BD.

o

[Ver anotações](#)

Caro aluno, tente alterar o código acima para incluir novos clientes. Navegue pela ferramenta phpMyAdmin e veja que o conteúdo inserido por meio da aplicação Java está disponível também nessa ferramenta.

EXEMPLIFICANDO

Conforme mencionado anteriormente, existe uma grande quantidade de comandos possíveis sobre um banco de dados. O Quadro 4.5 nos mostra uma síntese dos principais comandos CRUD em SQL suportados pelo MySQL e MariaDB.

Quadro 4.5 | Síntese de comandos SQL suportados pelo MySQL e MariaDB

Operações	Exemplo de Comando
Criar BD	<code>CREATE DATABASE `nome_bd`;</code>
Destruir BD	<code>DROP DATABASE `nome_bd`;</code>
Criar Tabela	<code>CREATE TABLE `nome_bd`.`nome_tbl`(`col1` TIP01, `col2` TIP02, ... `colN` TIPON);</code>
Destruir Tabela	<code>DROP TABLE `nome_bd`.`nome_tbl`;</code> ou <code>DROP TABLE `nome_tbl`;</code>
Consultar Tudo	<code>SELECT * FROM `nome_bd`.`nome_tbl`;</code> ou <code>SELECT * FROM `nome_tbl`;</code>
Consultar Coluna	<code>SELECT `col1`, `col2` FROM `nome_tbl`;</code> ou <code>SELECT `col1`, `col2` FROM `nome_bd`.`nome_tbl`;</code>

Operações	Exemplo de Comando
Atualizar Campo	UPDATE `nome_tbl` SET `colX`='NovoValor' WHERE `colY`='Valor';
Atualizar Coluna	UPDATE `nome_tbl` SET `colX`='NovoValor' WHERE 1;
Inserir Nova Linha	INSERT INTO `nome_tbl` (`col1`, `col2`, ...`colN`) VALUES (valor1, valor2, ... valorN);
Deletar Linha	DELETE FROM `nome_tbl` WHERE `colX`='Valor';

Fonte: elaborado pelo autor.

Experimente testar todos os comandos listados acima para compreender, na prática, o seu funcionamento.

Além dos BDs do tipo relacional, como o MySQL e MariaDB, existem outros BDs que, atualmente, vêm ganhando bastante destaque e são classificados como NoSQL. Existem diversos BDs que se encaixam nessa categoria, e podemos destacar: mongoDB, Cassandra, CouchDB, Redis, HBase, Druid, Riak, entre outros. Diversas empresas no ramo da tecnologia, como Facebook, Amazon, Google e Microsoft, utilizam bancos de dados NoSQL para armazenar grandes volumes de dados. O mongoDB funciona bem para aplicações em que os dados são não estruturados.

O Quadro 4.6 a seguir nos apresenta uma síntese de características presentes nos bancos MySQL e mongoDB.

Quadro 4.6 | Síntese de características entre os bancos de dados MySQL e mongoDB

Características	MySQL	mongoDB
Ano de Lançamento	1995	2009
Código-Fonte	Aberto	Aberto
Plataforma	Multiplataforma	Multiplataforma

Características	MySQL	mongoDB
Empresa Mantenedora	Oracle Corporation	MongoDB Inc.
Linguagem de Consulta	SQL	NoSQL
Classificação	BD Relacional	BD Não Relacional
Escalabilidade	Verticalmente	Horizontalmente
Esquemas	Estáticos	Dinâmicos
ACID	Sim	Sim
Pontos principais	Tabela, Linha e Coluna	Coleção, Documento e Campo

Fonte: elaborado pelo autor.

Caro aluno, acesse os sites dos Bancos de Dados MySQL, MariaDB e mongoDB para conhecê-los melhor, bem como o site do phpMyAdmin, que é um famoso gerenciador de BDs na web. É extremamente importante ficar atento às novas versões e notícias das ferramentas.

Nesta seção, você estudou os conteúdos relacionados à leitura de arquivos de texto, leitura de arquivos CSV, criação de banco de dados MySQL e integração do Java com BD MySQL, bem como analisou diversos exemplos sobre como esses conceitos são implementados na linguagem Java, e todos os códigos aqui mostrados podem ser acessados no GitHub do autor. Na seção seguinte, estudaremos como criar uma aplicação que execute em paralelo por meio da utilização de *threads*, o que nos fará avançar ainda mais no entendimento da linguagem Java.

REFERÊNCIAS

ARANTES, J. da S. **Livro-POO-Java**. 2020. Disponível em: <https://bit.ly/3eiUMcE>. Acesso em: 8 set. 2020.

CURSO EM VÍDEO. **Curso de Banco de Dados MySQL**. 2018. Disponível em: <https://bit.ly/2FwY2pR>. Acesso em: 8 set. 2020.

CURSO EM VÍDEO. **Curso MySQL #02a - Instalando o MySQL com WAMP**. 2016. Disponível em: <https://bit.ly/3hIZW3J>. Acesso em: 8 set. 2020.

CURSO EM VÍDEO. **Curso MySQL #09 – PHPMyAdmin (Parte 1)**. 2016. Disponível em: <https://bit.ly/32Jku7C>. Acesso em: 8 set. 2020.

CURSO EM VÍDEO. **Curso MySQL #10 - PHPMyAdmin (Parte 2)**. 2016. Disponível em: <https://bit.ly/3cboHEf>. Acesso em: 8 set. 2020.

MARIADB. **MariaDB**. 2020. Disponível em: <https://bit.ly/2RG7kSS>. Acesso em: 8 set. 2020.

MONGODB. **Mongo DB**. 2020. Disponível em: <https://bit.ly/35PCgbA>. Acesso em: 8 set. 2020.

ORACLE. **MySQL**. 2020. Disponível em: <https://bit.ly/2ZPUrd9>. Acesso em: 8 set. 2020.

PHPMYADMIN. **PhpMyAdmin**. 2020. Disponível em: <https://bit.ly/35NKjWf>. Acesso em: 8 set. 2020.

W3SCHOOLS. **SQL Tutorial**. [s.d.]. Disponível em: <https://bit.ly/3myPyPG>. Acesso em: 8 set. 2020.

FOCO NO MERCADO DE TRABALHO

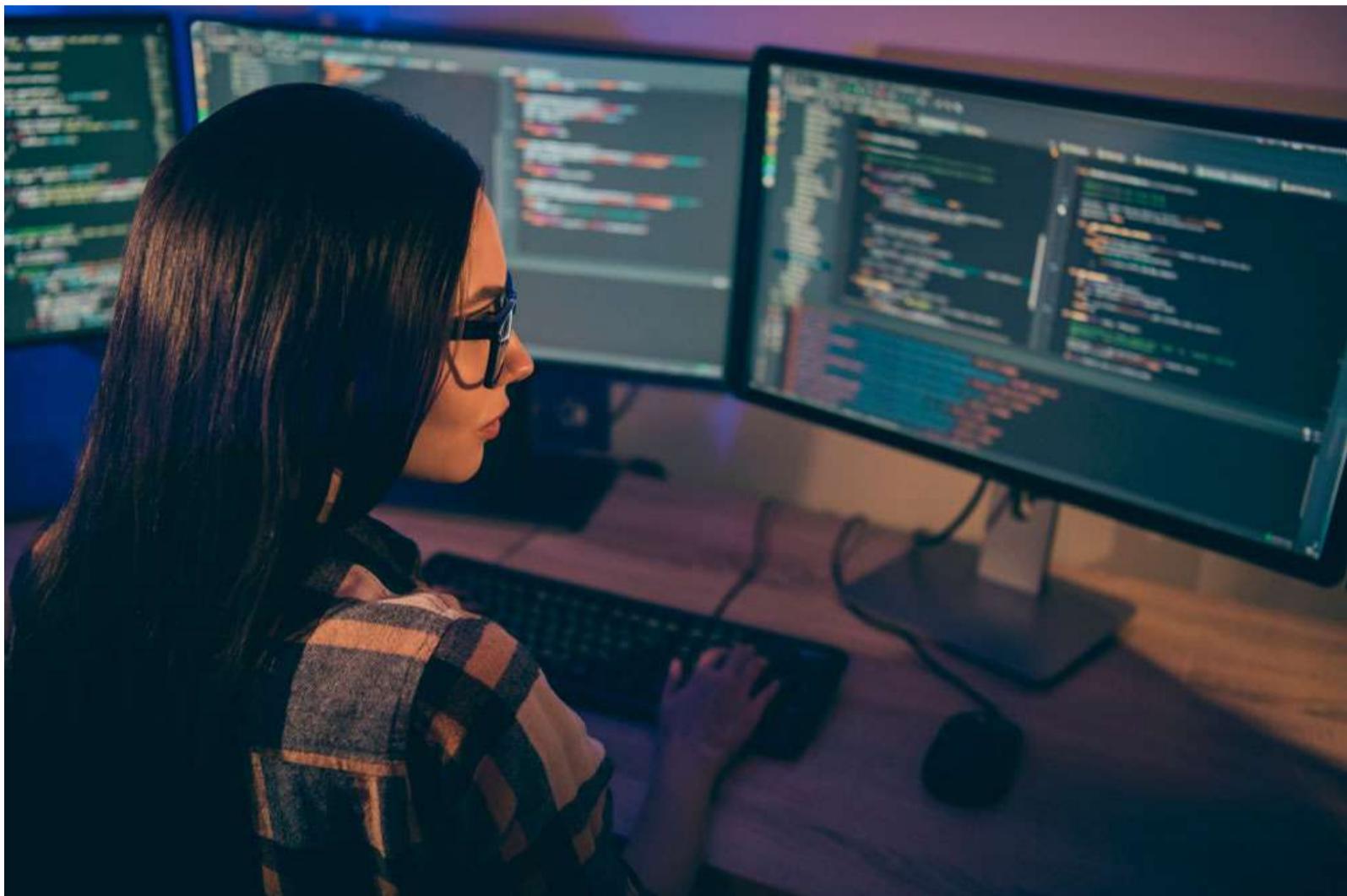
BANCO DE DADOS RELACIONAL E NOSQL

Jesimar da Silva Arantes

Ver anotações

UTILIZAÇÃO DE BANCO DE DADOS

Criação de banco de dados relacional para armazenamento das informações da rota do robô e sua integração com Java.



Fonte: Shutterstock.

Deseja ouvir este material?

Áudio disponível no material digital.

SEM MEDO DE ERRAR

A *startup* em que você trabalha lhe passou um conjunto de atividades, como:

- Instalar o XAMPP para acessar o MySQL e o phpMyAdmin.
- Criar um banco de dados para o robô.
- Criar uma tabela nesse banco para armazenar a rota do robô.

- Integrar a aplicação em Java com o banco que foi criado utilizando o phpMyAdmin.

Após instalados o XAMPP com MySQL e phpMyAdmin, você decidiu criar um BD chamado bd_simulador_robo e uma tabela chamada rota por meio do phpMyAdmin; para isso, você escreveu o Código 4.13 a seguir.

Código 4.13 | Comandos SQL para criação do BD e da tabela no MySQL

```
1 CREATE DATABASE `bd_simulador_robo`;  
2  
3 CREATE TABLE `bd_simulador_robo`.`rota`(  
4     `posx` DECIMAL(10) DEFAULT NULL,  
5     `posy` DECIMAL(10) DEFAULT NULL,  
6     `codigo` INT(10) NOT NULL AUTO_INCREMENT,  
7     PRIMARY KEY(`codigo`)  
8 ) ENGINE = INNODB AUTO_INCREMENT = 1 DEFAULT CHARSET = latin1;
```

Fonte: elaborado pelo autor.

Após isso, você decidiu partir do Código 4.6 e adicionar os recursos em Java que armazenam as posições do robô no BD. Dessa maneira, você escreveu o Código 4.14 (repare que foram destacadas apenas as partes em que houve alteração, e o código completo pode ser acessado no GitHub do autor). Nas linhas 4 a 6, temos a declaração três novas variáveis para comunicação com o BD; nas linhas 12, 17, 22 e 27, temos uma invocação do BD para inserir a posição atual do robô toda vez que ele se mover, dessa maneira, no BD, ficará armazenado toda a rota percorrida pelo robô; por fim, nas linhas 34 a 49, temos a declaração da classe Consultas, em que definimos o método inserir posição, e um programa pronto com todas as alterações requisitadas.

Código 4.14 | Adaptações na implementação para suportar armazenar a rota do robô

```
1 ... // diversas importações foram omitidas
2 public class AppGUI extends Application {
3     ... // diversas declarações foram omitidas
4     private final ConexaoBD conBD = new ConexaoBD();
5     private final Connection connection = conBD.getConexao();
6     private final Consultas consultas = new Consultas();
7     ... // o método main foi omitido
8     public void start(Stage janela) {
9         cena.setOnKeyPressed((evt) -> {
10             if (evt.getCode() == KeyCode.UP) {
11                 ... // algumas linhas foram omitidas
12                 consultas.inserirPos(connection,
13                     robo.getPosicaoX(), robo.getPosicaoY());
14             }
15             if (evt.getCode() == KeyCode.DOWN) {
16                 ... // algumas linhas foram omitidas
17                 consultas.inserirPos(connection,
18                     robo.getPosicaoX(), robo.getPosicaoY());
19             }
20             if (evt.getCode() == KeyCode.LEFT) {
21                 ... // algumas linhas foram omitidas
22                 consultas.inserirPos(connection,
23                     robo.getPosicaoX(), robo.getPosicaoY());
24             }
25             if (evt.getCode() == KeyCode.RIGHT) {
26                 ... // algumas linhas foram omitidas
27                 consultas.inserirPos(connection,
28                     robo.getPosicaoX(), robo.getPosicaoY());
29             }
30         });
31     }
32 }
33
34 import java.sql.Connection;
35 import java.sql.SQLException;
36 import java.sql.Statement;
37 public class Consultas {
38     public void inserirPos(Connection conn, float posX, float posY){
39         try {
40             Statement st = conn.createStatement();
41         }
42     }
43 }
```

```
41         String sql = "INSERT INTO `rota` (posx, posy) " +
42                     "VALUES (" + posX + ", " + posY + ")";
43         st.executeUpdate(sql);
44         st.close();
45     } catch (SQLException ex) {
46         System.out.println(ex);
47     }
48 }
49 }
50
51 //Foi adicionado uma classe chamada ConexaoBD que é semelhante à
//mostrada no Código 4.12 nas linhas 19 a 60.
```

0

Ver anotações

Fonte: elaborado pelo autor.

PROGRAMAÇÃO EM JAVA USANDO *THREADS*

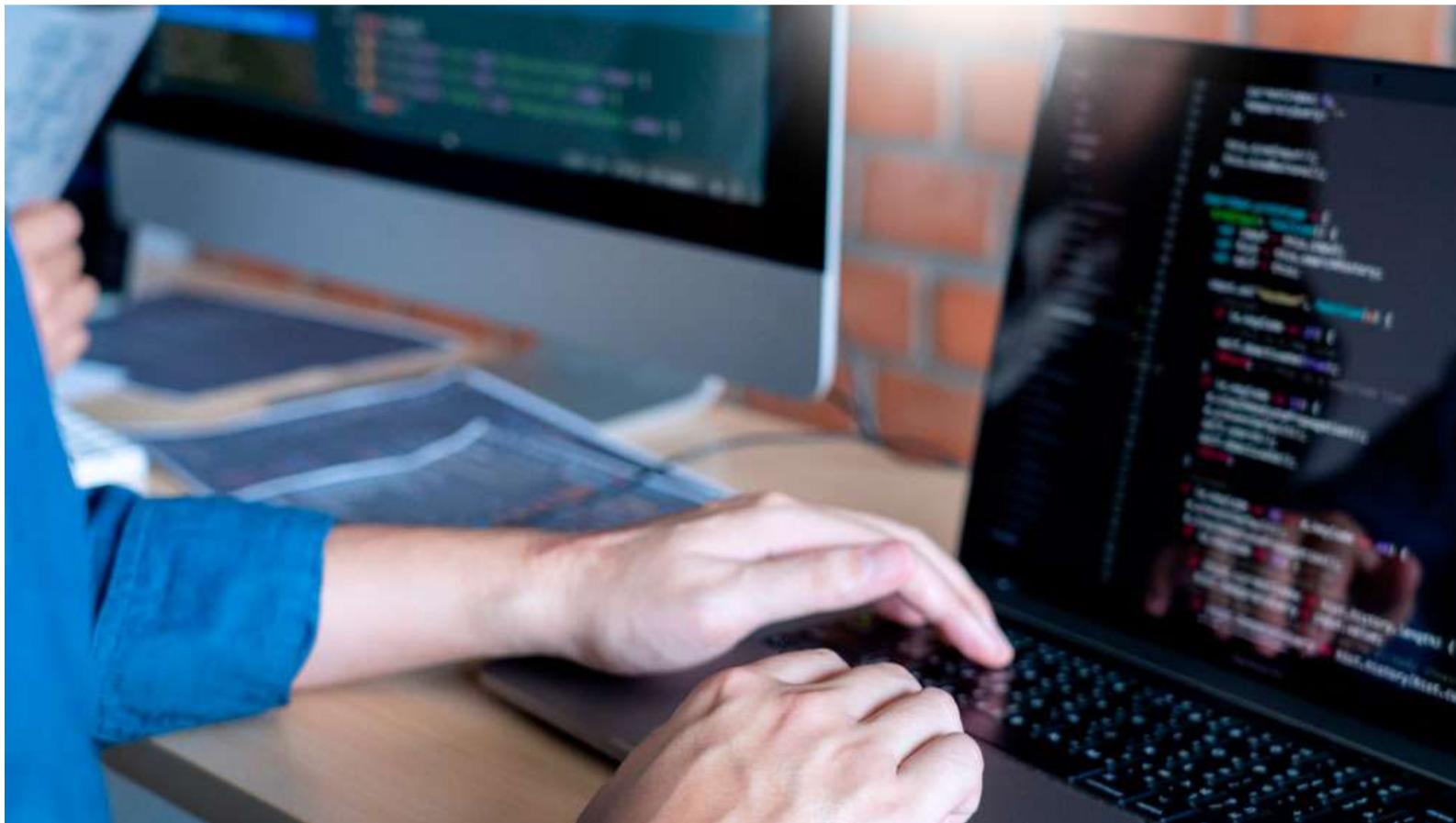
Jesimar da Silva Arantes

0

[Ver anotações](#)

APLICAÇÕES QUE UTILIZAM *THREADS*

As *threads* são utilizadas para destravar parte da aplicação quando algum processamento pesado está sendo executado e acelerar o processamento em alguma aplicação que possa ser paralelizada.



Fonte: Shutterstock.

Deseja ouvir este material?

Áudio disponível no material digital.

PRATICAR PARA APRENDER

Caro aluno, bem-vindo à terceira seção da quarta e última unidade de estudos sobre Linguagem Orientada a Objetos. Qual desenvolvedor nunca desejou criar uma aplicação que execute em paralelo ou que não trave quando um processo pesado inicia a sua execução? Acredito que muitos desenvolvedores já desejaram isso. Atualmente, a maioria dos processadores vendidos são *multic平ores*, ou seja, possuem mais de um núcleo de processamento, mas pouco adianta ter um *hardware* robusto sem que o *software* consiga extrair o máximo desse *hardware*. Nesta seção, estudaremos as *threads* que nos auxiliam a criar programas que executem em paralelo ou concorrentemente.

o

Ver anotações

De forma a contextualizar a sua aprendizagem, lembre-se de que você está trabalhando em um simulador de robô e que seu gestor reviu o seu código e percebeu que na sala em que o robô opera, você ainda não colocou, na sua simulação, a máquina que carrega as caixas da área de depósito. Ele explicou que o seu simulador ficará mais realista se possuir uma pequena animação dessa máquina, tal como na aplicação que desenvolveu usando a ferramenta Greenfoot. Diante disso, ele pediu a você que colocasse a máquina carregadora de caixas e criasse uma pequena animação em zigue-zague para ela, bem como recomendou que utilize *threads* para isso, de forma que a animação da máquina carregadora seja independente do movimento executado pelo robô.

Diante do desafio que lhe foi apresentado, como você criará essa animação da máquina? Como você criará essa *thread* em Java? O que é uma *thread*? Para que servem as *threads*? Esta seção o auxiliará na resposta de tais perguntas.

Muito bem, agora que você foi apresentado à sua nova situação-problema, estude esta seção e compreenda como a linguagem Java trata as *threads*; esse conceito é fundamental para que você consiga fazer grandes projetos de programação.

E aí, vamos juntos compreender esses conceitos e resolver esse desafio?

Bom estudo!

CONCEITO-CHAVE

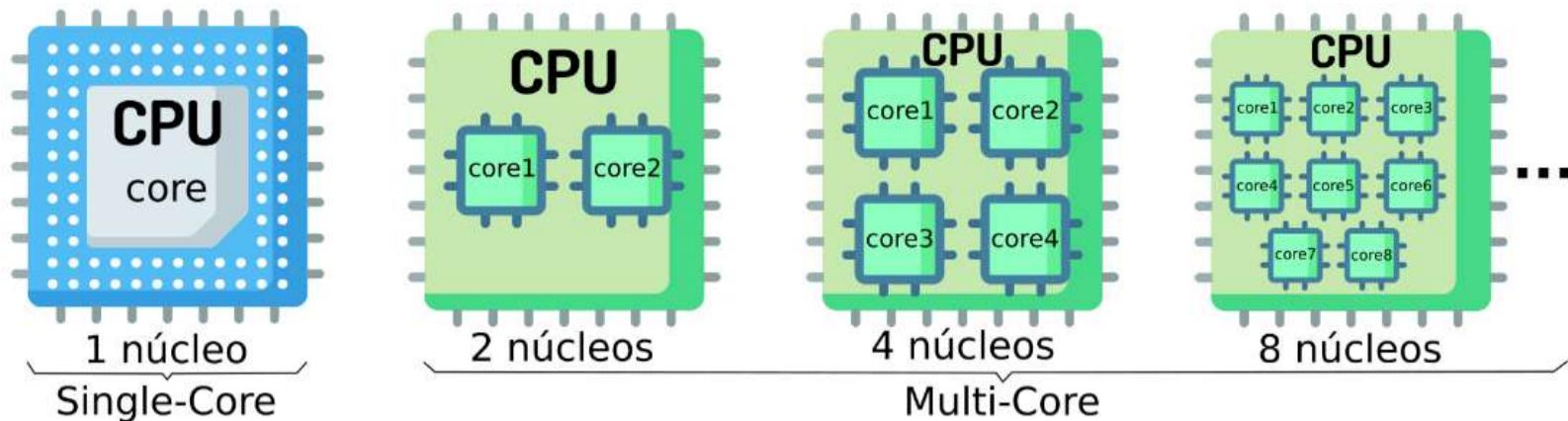
A presente seção tem por objetivo lhe mostrar como criar aplicações em Java para serem executadas em paralelo ou concorrentemente, logo, vamos estudar alguns aspectos de *hardware* e *software* que permitem a concorrência e o paralelismo. Os aspectos de *hardware* estudados são processadores *single-core* e *multi-core*, já os aspectos de *software* estudados são processos e *threads*.

Um computador, de forma simples, possui um conjunto de dispositivos de *hardware* que auxiliam no seu funcionamento, como processador, memória cache, memória RAM, disco rígido e dispositivos de entrada e saída de dados.

PROCESSADORES

De forma a avançarmos nesta seção, vamos nos concentrar apenas no processador. Existem, basicamente, dois tipos de processadores, que são: *single-core* (com apenas um núcleo) ou *multi-core* (com mais de um núcleo), como ilustra a Figura 4.11.

Figura 4.11 | Esquema representando os processadores *single-core* e *multi-core*



Fonte: elaborada pelo autor.

0

Ver anotações

Na Figura 4.11, à esquerda, temos uma ilustração de um processador com um único núcleo (*core*). Os processadores *single-core* são mais simples e só conseguem executar uma tarefa por vez; eles foram criados primeiro e não suportam a execução em paralelo devido ao único núcleo; apesar disso, diversas aplicações podem ser executadas concorrentemente por meio de mecanismos de compartilhamento do tempo de processamento.

Já os processadores à direita, na Figura 4.11, possuem 2, 4 e 8 núcleos (*cores*), logo, são processadores *multi-core*. Esse tipo de processador foi criado depois e é um pouco mais complexo; nesse tipo de arquitetura de *hardware*, é permitida a execução em paralelo.

Os processadores *multi-core* estão ficando cada vez mais presentes no mercado e o número de núcleos tem aumentado, tornando o poder de processamento cada vez mais poderoso. Atualmente, a maioria dos computadores, celulares e dispositivos de computação embarcada, como Raspberry Pi, é *multi-core*.

DICA

Leitor, procure descobrir quantos núcleos o seu processador possui, pois a resposta a essa questão o auxiliará na compreensão do restante da seção e na construção de aplicações paralelas.

PROCESSOS E *THREADS*

Bem, até aqui, falamos dos aspectos de *hardware*, agora, vamos falar de dois conceitos importantes de software, que são: os processos e as *threads*. Um **processo** tem o seu próprio ambiente de execução e seu próprio espaço de memória, além disso, é muito comum associar um processo a um programa ou aplicação, no entanto, isso não é exatamente correto e tais detalhes são estudados na disciplina de Sistemas Operacional (SO). Apesar disso, neste livro, para a

simplificarmos as coisas, podemos pensar em um processo como sendo uma aplicação, sem grandes perdas de generalidade, pois a maioria das aplicações em Java roda em apenas um processo. Uma **thread** é uma linha de execução; ao criarmos uma *thread*, esta cria também um ambiente de execução, mas compartilha recursos como memória e arquivos abertos. É importante que saiba que cada processo tem, pelo menos, uma *thread*, mas que é possível criar mais *threads* dentro desse processo se desejar. Um dos aspectos interessantes das *threads* é que elas são mais leves do que os processos, e a execução de uma *thread* pode ser interrompida quantas vezes forem necessárias, continuando, sempre, do ponto em que parou. É uma tarefa do SO, em específico do escalonador de processos, decidir qual *thread* executará e em qual núcleo do processador.

Inicialmente, vamos mostrar uma simples aplicação em Java que apenas imprime o nome da *thread* principal que está em execução. Dessa maneira, analise o Código 4.15.

Código 4.15 | Aplicação básica com um processo e uma *thread*

```
1 public class SimplesApp {  
2     public static void main(String[] args) {  
3         Thread thFluxo = Thread.currentThread();  
4         System.out.printf("Nome Thread: %s%n", thFluxo.getName());  
5     }  
6 }
```

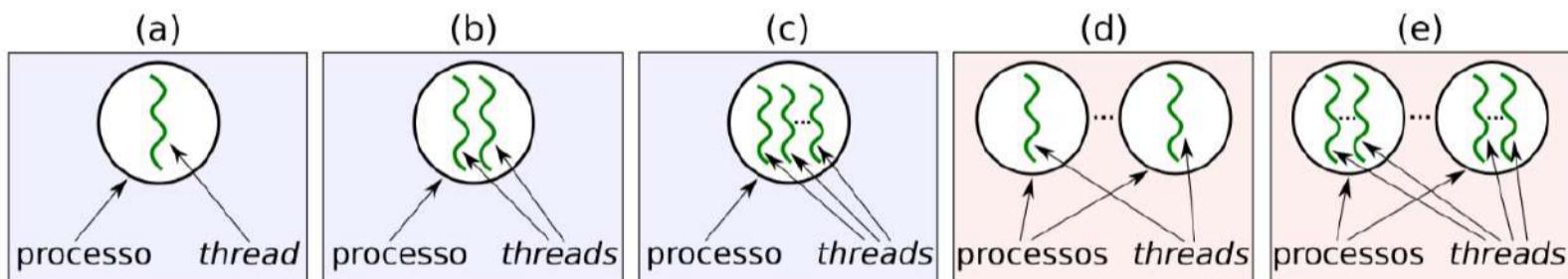
Fonte: elaborado pelo autor.

Nas linhas 1 e 2 do Código 4.15, temos a declaração da classe e a definição do método principal (*main*); na linha 3, foi criado um objeto chamado *thFluxo*, que recebe a *thread* atual; e na linha 4, é impresso na tela o nome da *thread* atual em execução.

Bem, até aqui, discutimos o código, mas não vimos o seu comportamento durante o fluxo de execução, assim, imagine que você desenvolveu esse código e clicou no botão para iniciar a sua execução; dessa maneira, a JVM criará um processo para que essa aplicação possa ser executada (toda aplicação está associada a um processo); em seguida, a JVM criará, também, uma *thread* dentro desse processo, e é importante que se lembre de que cada processo possui pelo menos uma *thread*, e a partir desse ambiente criado (processo + *thread*), a aplicação possa ser executada. Dessa forma, essa aplicação imprimirá o nome da *thread*, que se chama “main”. (Experimente implementar e executar esse código.)

A Figura 4.12 é uma ilustração que nos auxilia a entender as relações possíveis entre processos e *threads*. Dito isso, analise esta figura:

Figura 4.12 | Esquema representando as possíveis relações entre processos e *threads*



Fonte: elaborada pelo autor.

Na Figura 4.12 (a), temos a relação mais simples possível na construção de uma aplicação em que temos um processo e uma *thread* que está vinculada a ele. Esse tipo de relação ilustra, por exemplo, o Código 4.15 e todos os outros códigos anteriormente vistos neste livro. A Figura 4.12 (b) nos mostra um processo que possui duas *threads* vinculadas a ele; já a Figura 4.12 (c) nos mostra, de forma genérica, que um processo pode estar vinculado a várias *threads*.

A seguir, neste livro, construiremos exemplos de aplicações que funcionam como representado na Figura 4.12 (b) e (c). Na Figura 4.12 (d), temos a relação em que uma aplicação cria vários processos e cada processo possui apenas uma *thread*; na Figura 4.12 (e), temos uma última relação possível na construção de aplicações em

que podemos criar vários processos e cada processo pode possuir várias *threads*; já as Figuras 4.12 (d) e (e) nos mostram relações mais complexas e não serão tratadas neste livro, pois necessitam de um curso mais avançado de Java.

THREADS

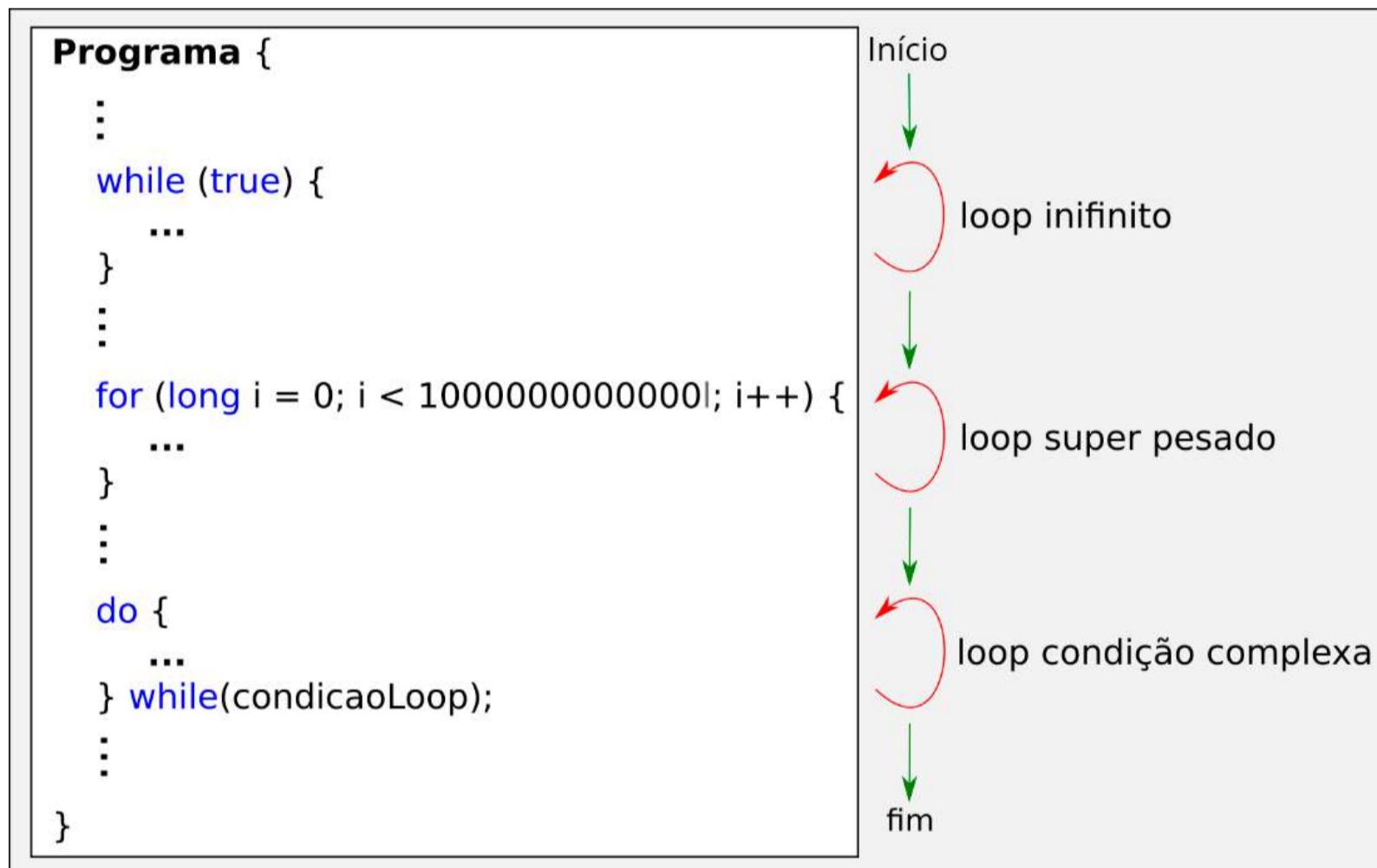
O exemplo do Código 4.15 foi utilizado apenas para discutirmos as ideias de processos e *threads*; a partir de agora, vamos nos concentrar apenas nas *threads* e trabalharmos com apenas o processo principal.

É muito importante sabermos quando utilizar as *threads*; no geral, elas são utilizadas para:

- Destrarvar alguma parte da aplicação quando algum processamento pesado está sendo executado. Neste caso, pode-se utilizar *threads* em computadores com apenas um único núcleo ou *multicores*.
 - Acelerar o processamento em alguma aplicação que possa ser paralelizada. Neste caso, faz sentido utilizar as *threads* apenas em computadores que possuem mais de um núcleo, caso contrário, não haverá qualquer ganho de desempenho.

De forma a ilustrar o primeiro caso de aplicação, vamos considerar a Figura 4.13 mostrada a seguir.

Figura 4.13 | Exemplo de aplicação com *loops* que necessita de *threads* para destravar



Fonte: elaborada pelo autor.

A Figura 4.13 nos mostra o rascunho de código de um programa qualquer em que as reticências na vertical indicam linhas de código que possuem processamento leve e que não queremos destacar; em seguida, nesse código, temos um *loop* infinito que executa alguma verificação rotineira, porém importante; posteriormente, temos um *loop* que executa algum cálculo super pesado que demora alguns minutos ou até horas para ser executado; por fim, temos um *loop* com uma condição que não sabemos quando será satisfeita.

o

Ver anotações

Esse código da Figura 4.13, apesar de ser abstrato, pode representar um código que desejamos implementar em alguma aplicação do mundo real, no entanto, de acordo com os estudos, sabemos que o código nunca sairá do primeiro *loop*, pois se trata de um *loop* infinito. As *threads*, por sua vez, solucionam problemas de travamento de código como esse, logo, basta colocar uma *thread* para executar cada um dos *loops* mostrados, e é importante destacarmos que duas ou mais tarefas podem ser executadas ao mesmo tempo, mesmo que haja apenas um processador. Esse tipo de execução é chamado de concorrente, pois a aplicação concorre à utilização do processador, e isso é feito de forma automática pelo sistema operacional, que escalona as tarefas a ocuparem o núcleo do processador.

O Código 4.16 a seguir nos mostra como seria uma implementação da ideia sugerida na Figura 4.13. Por esse código ser simples, não será explicado.

Implemente este código e veja qual será a saída.

Código 4.16 | Aplicação construída sobre o exemplo da Figura 4.13 sem *threads*

```

1  public class ProgramaLoopSemThread {
2
3      public static void main(String[] args) {
4
5          ProgramaLoopSemThread p = new ProgramaLoopSemThread();
6
7          p.programa();
8
9      }
10
11     public void programa() {
12
13         System.out.println("inicio");
14
15         while (true) {
16
17             if (1 % 2 == 2) break;//artimanha p/ compilar o código
18
19             System.out.println("loop infinito");
20
21         }
22
23         System.out.println("passou do primeiro loop");
24
25         for (long i = 0; i < 1000000000000L; i++) {
26
27             System.out.println("loop super pesado");
28
29         }
30
31         System.out.println("passou do segundo loop");
32
33         boolean condicaoLoop = true;
34
35         do {
36
37             System.out.println("loop condição complexa");
38
39         } while (condicaoLoop);
40
41         System.out.println("fim");
42
43     }
44 }
```

0

Ver anotações

Fonte: elaborado pelo autor.

Você deve ter percebido que a aplicação ficou travada no primeiro *loop*, assim, as linhas 12 a 21 do Código 4.16 nunca serão executadas. Vamos, agora, mostrar como colocar *threads* nessa aplicação genérica para que os três *loops* sejam executados concorrentemente ou paralelamente (se tiver mais de um núcleo em seu processador).

Analise o código 4.17 a seguir.

Código 4.17 | Aplicação construída sobre o exemplo da Figura 4.13 com *threads*

0

[Ver anotações](#)

```
1 public class ProgramaLoopComThread {
2     public static void main(String[] args) {
3         ProgramaLoopComThread p = new ProgramaLoopComThread();
4         p.programa();
5     }
6     public void programa() {
7         System.out.println("inicio");
8         Thread thread1 = new Thread(loop1());
9         thread1.start();
10        System.out.println("passou do primeiro loop");
11        Thread thread2 = new Thread(loop2());
12        thread2.start();
13        System.out.println("passou do segundo loop");
14        Thread thread3 = new Thread(loop3());
15        thread3.start();
16        System.out.println("fim");
17    }
18    public Runnable loop1() {
19        Runnable run1 = new Runnable() {
20            @Override
21            public void run() {
22                while (true) {
23                    if (1 % 2 == 2) break;//artimanha p/ compilar
24                    System.out.println("loop infinito");
25                }
26            }
27        };
28        return run1;
29    }
30    public Runnable loop2() {
31        Runnable run2 = new Runnable() {
32            @Override
33            public void run() {
34                for (long i = 0; i < 1000000000001; i++) {
35                    System.out.println("loop super pesado");
36                }
37            }
38        };
39        return run2;
40    }
}
```

```

41     public Runnable loop3() {
42
43         Runnable run3 = new Runnable() {
44
45             @Override
46
47             public void run() {
48
49                 boolean condicaoLoop = true;
50
51                 do {
52
53                     System.out.println("loop condição complexa");
54                 } while (condicaoLoop);
55
56             }
57
58         };
59
60         return run3;
61
62     }
63

```

0

[Ver anotações](#)

Fonte: elaborado pelo autor.

Nas linhas 18 a 29 do Código 4.17, colocamos o *loop* infinito dentro de um objeto do tipo *Runnable* (na Unidade 3, Seção 2, estudamos um pouco sobre a interface *Runnable*), que se trata de uma interface que possui um método chamado *run*, que deve ser obrigatoriamente sobrescrito; de forma geral, colocamos dentro do método *run* tudo o que queremos paralelizar ou executar de forma concorrente. Na linha 28, por sua vez, retornamos o objeto *Runnable* criado; de forma semelhante, nas linhas 30 a 40, o *loop* super pesado foi colocado dentro de um objeto também do tipo *Runnable*, e o mesmo foi feito nas linhas 41 a 52 com o *loop* que possui a condição complexa. Já na linha 8, criamos um objeto do tipo *Thread* que recebe como argumento um objeto *Runnable* (nesse caso, a primeira *thread* executará o *loop* infinito); na linha 9, mandamos a *thread* para ser executada, de fato, por meio do método *start*; nas linhas 11 e 12, criamos uma nova *thread* que executará o *loop* super pesado; por fim, nas linhas 14 e 15, criamos a nossa última *thread*, que executará o *loop* com condição complexa.

Implemente esse código e analise a saída impressa; compare essa saída com a saída do Código 4.16, que não possuía *threads*; faça diversas execuções e repare que, cada vez que executar o programa, a saída será diferente, pois, a cada vez, o escalonador de processos do SO selecionará as *threads* de outra forma.

Vamos, agora, criar uma aplicação em que desejamos realizar algum processamento em paralelo. Para isso, vamos criar um simples programa que calcula se um número de entrada é primo ou não; caso seja primo, então, o

número será impresso na tela. Analise o Código 4.18 a seguir.

Código 4.18 | Aplicação paralelizada para acelerar a detecção de números primos

0

[Ver anotações](#)

0

[Ver anotações](#)

```
1 public class Primo implements Runnable {
2     private final int inicio;
3     private final int fim;
4     public Primo(int inicio, int fim) {
5         this.inicio = inicio;
6         this.fim = fim;
7     }
8     public boolean isPrime(int n) {
9         if (n < 2) {
10             return false;
11         }
12         for (int i = 2; i < (int)(Math.sqrt(n) + 1); i++) {
13             if (n % i == 0) {
14                 return false;
15             }
16         }
17         return true;
18     }
19     @Override
20     public void run() {
21         for (int i = inicio; i < fim; i++) {
22             boolean ehPrimo = isPrime(i);
23             if (ehPrimo) {
24                 System.out.println("é primo: " + i);
25             }
26         }
27     }
28 }
29 public class AppPrimoThread {
30     public static void main(String[] args) {
31         Thread thr1 = new Thread(new Primo(0, 1000000));
32         thr1.start();
33         Thread thr2 = new Thread(new Primo(1000001, 2000000));
34         thr2.start();
35         Thread thr3 = new Thread(new Primo(2000001, 3000000));
36         thr3.start();
37         Thread thr4 = new Thread(new Primo(3000001, 4000000));
38         thr4.start();
39     }
40 }
```

Na linha 1 do Código 4.18, criamos uma classe chamada Primo, que implementa a interface Runnable (lembre-se de que a ideia é, a partir da implementação dessa interface, paralelizarmos os cálculos dos números primos); nas linhas 2 e 3, definimos duas variáveis para armazenar o número de início e fim dos cálculos de números primos; nas linhas 4 a 7, criamos o construtor da classe com a especificação dos valores de início e de fim; nas linhas 8 a 18, definimos um método que, dado um número de entrada, verifica se este é primo ou não; nas linhas 19 a 27, sobrescrevemos o método *run* da interface Runnable e colocamos o cálculo que deverá ser paralelizado (neste caso, queremos calcular todos os números primos do intervalo início ao fim e imprimir, na tela, apenas os que forem primos); nas linhas 29 a 40, criamos uma classe para testar a nossa aplicação; na linha 31, foi criado um objeto do tipo Thread que recebe um objeto do tipo Runnable (neste caso, queremos calcular, nessa *thread*, todos os primos de 0 a 1.000.000); na linha 32, mandamos iniciar essa execução; por fim, o mesmo raciocínio foi posto em prática nas linhas 33 a 38, em que mais três *threads* foram criadas com novos valores de início e fim do cálculo.

o

Ver anotações

ASSIMILE

Quando temos uma aplicação semelhante à mostrada no Código 4.18 e queremos acelerar os cálculos, devemos sempre pensar na arquitetura de *hardware* que executará o programa. Por exemplo: imagine que o processador que executará esse código é um Intel i5 com oito núcleos, assim, o número máximo de *threads* que faz sentido de se criar é oito. Se quisermos, até podemos criar mais *threads*, pois elas são entidades lógicas, porém, se fizermos isso, não conseguiremos extrair o máximo do *hardware*. Lembre-se de que, ao se criar mais *threads* do que o número de núcleos da CPU, um *overhead* é gerado nas trocas de contexto das *threads*. É importante lembrar, também, que as *threads* ficam concorrendo para fazer uso dos núcleos da CPU e executarem o processamento, dessa maneira, caso o seu computador seja um *dual core* com dois núcleos, para obter um maior desempenho, remova duas *threads* do Código 4.18.

No Código 4.18, a classe Primo implementou a interface Runnable e, assim, sobrescreveu o método *run*. Existe outra forma de se fazer a criação da classe Primo, que é herdando da classe Thread em vez de implementar a interface Runnable, e as duas formas são equivalentes e funcionam perfeitamente.

Sugerimos, até mesmo, que você, aluno, após implementar o Código 4.18, faça essa alteração e confirme essa informação.

A seguir, seguem as partes que podem ser alteradas e que obtêm o mesmo efeito na paralelização de código.

Classe Primo implementado Runnable: `public class Primo implements Runnable`

Classe Primo herdando Thread: `public class Primo extends Thread`

o

[Ver anotações](#)

Abaixo, listamos alguns métodos estáticos e não estáticos importantes da classe Thread:

- **Thread.sleep(long milissegundos)**: faz com que o encadeamento da execução adormeça pelo número especificado em milissegundos.
- **Thread.currentThread()**: retorna uma referência ao objeto da *thread* atualmente em execução.
- **getId()**: retorna um identificado da *thread*, e o tipo do retorno é *long*.
- **getName()**: retorna o nome da *thread*, e o tipo do retorno é *String*.
- **getPriority()**: retorna a prioridade da *thread*, e o tipo do retorno é *int*.
- **interrupt()**: interrompe a execução da *thread*.
- **isAlive()**: testa se a *thread* está viva, e o tipo do retorno é *boolean*.
- **isInterrupted()**: testa se a *thread* tem sido interrompida, e o tipo do retorno é *boolean*.
- **join()**: aguarda até que a *thread* atual termine.
- **setName(String nome)**: troca o nome da *thread*.
- **setPriority(int prioridade)**: troca a prioridade da *thread*.
- **Thread.State.getState()**: retorna o estado atual da *thread*.

Caro aluno, procure criar aplicações simples, que utilizem os métodos acima da classe Thread, para ver, na prática, a sua utilização.

EXEMPLIFICANDO

Um método muito importante das *threads* é o *sleep*, que faz com que a *thread* atual durma por um determinado tempo. Esse método recebe como argumento um valor em milissegundos, que indica quanto tempo a *thread* deverá dormir. De forma a ilustrar esse método, analise o Código 4.19.

Código 4.19 | Exemplo básico de utilização do método estático *sleep*

```
1 System.out.println("inicio");
2 try {
3     System.out.println("inicio sleep");
4     Thread.sleep(5000); //valor em milissegundos
5     System.out.println("fim sleep");
6 } catch (InterruptedException ex) {
7     System.out.println(ex);
8 }
9 System.out.println("fim");
```

0

Ver anotações

Fonte: elaborado pelo autor.

No código acima, podemos reparar que o método *sleep* pode lançar uma exceção do tipo *InterruptedException*, e isso se dá quando a *thread* que está dormindo é interrompida por meio do método *interrupted*.

Frente a isso, sugerimos que você, aluno, tente criar uma aplicação simples e que force a interrupção da *thread* enquanto ela estiver dormindo para ver essa exceção sendo lançada.

As *threads*, ao serem criadas, podem assumir qualquer um dos seguintes estados:

- **NEW:** estado da Thread para uma *thread* que ainda não foi iniciada.
- **RUNNABLE:** estado da Thread para uma *thread* que está em execução.
- **TERMINATED:** estado da Thread para uma *thread* que foi encerrada.
- **WAITING:** estado da Thread para uma *thread* que está em espera.
- **TIMED_WAITING:** estado da Thread para uma *thread* que está em espera cronometrada.
- **BLOCKED:** estado da Thread para uma *thread* que está bloqueada, aguardando um bloqueio de monitor.

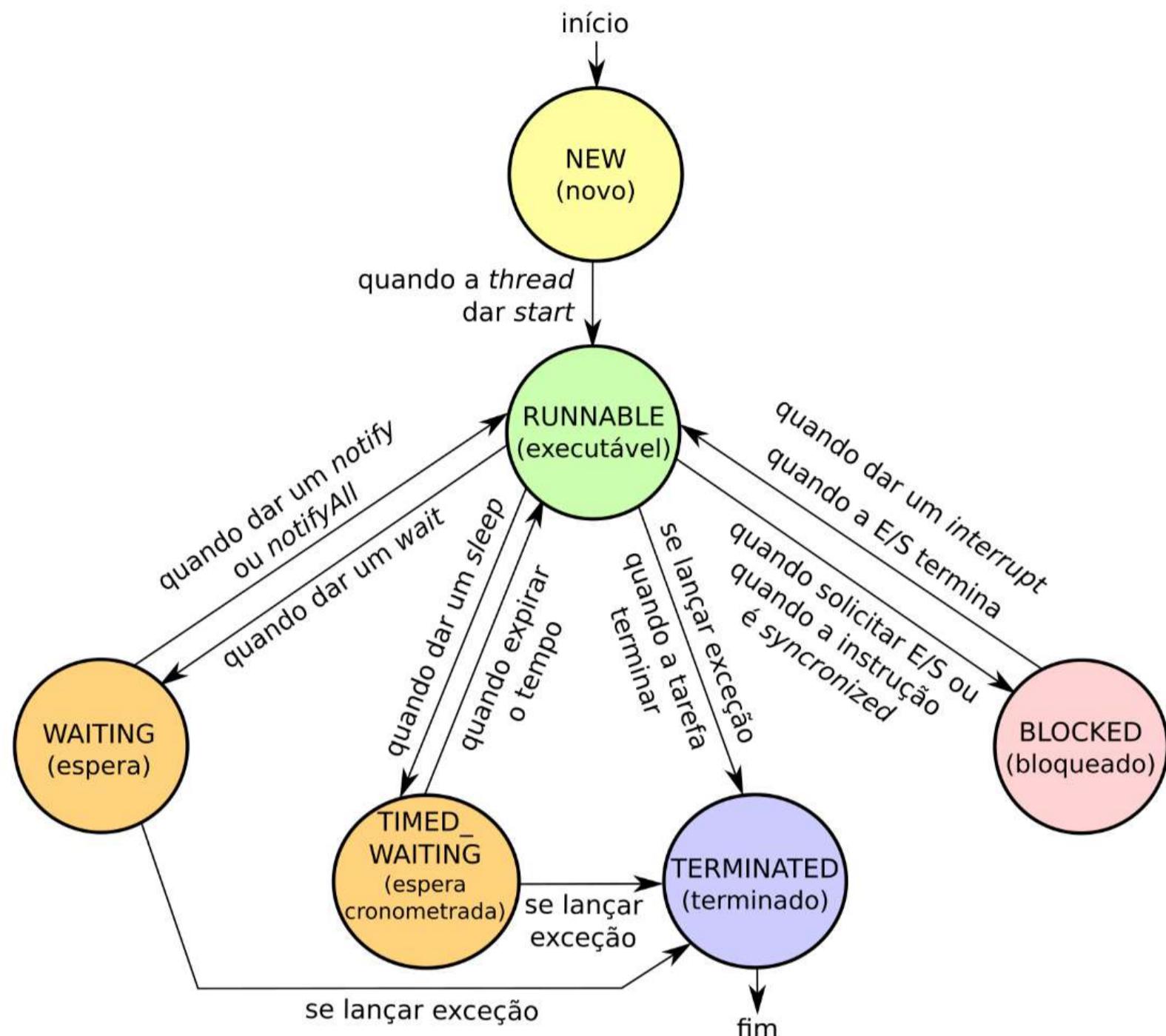
A Figura 4.14 nos mostra o fluxo seguido do ciclo de vida dos estados de uma *thread* dentro da linguagem Java. Inicialmente, quando a *thread* é criada, ela se encontra no estado NEW, em seguida, com o comando *start*, a *thread* muda para o estado RUNNABLE, e ela pode permanecer nesse estado até terminar, indo, então, para TERMINATED. A *thread* pode também ir do estado RUNNABLE para TIMED_WAITING com o comando *sleep*, e quando o intervalo de tempo expirar, cla

voltará para RUNNABLE, bem como pode ir do estado RUNNABLE para WAITING se usado o comando *wait*, e uma vez utilizado o comando *notify*, voltar ao estado RUNNABLE. O estado BLOCKED também pode ser atingido a partir de RUNNABLE por meio de solicitações de Entrada e Saído (E/S) ou quando outra *thread* estiver utilizando o recurso (*synchronized*), e quando essa solicitação terminar, o estado RUNNABLE retornará.

o

[Ver anotações](#)

Figura 4.14 | Ciclo de vida dos estados de uma *thread* dentro do Java



Fonte: adaptada de Deitel; Deitel (2016).

REFLITA

A Figura 4.14 nos mostra os ciclos de vida de uma *thread* a partir do momento de sua criação até o seu término. Diante disso, gostaríamos de convidá-lo a refletir sobre cada uma dessas transições de estados do ciclo de vida; utilize essa figura e tente descrever quais foram os estados ocupados pelas *threads* criadas nos códigos 4.17 e 4.18. Neles, teve algum estado não atingido? Se sim, quais? Reflita também sobre o porquê desses estados não terem sido atingidos.

Caro estudante, nesta seção você estudou os conteúdos relacionados à criação de *threads*; foram dados alguns exemplos de como as *threads* são criadas e como auxiliam na construção de aplicações concorrentes e paralelas na linguagem Java. Frente a isso, lembre-se de que todos os códigos aqui mostrados podem ser acessados no GitHub do autor.

REFERÊNCIAS

ARANTES, J. da S. **Livro-POO-Java**. 2020. Disponível em: <https://bit.ly/3eiUMcE>.

Acesso em: 8 set. 2020.

DEITEL, P. J.; DEITEL, H. M. **Java: como programar**. 10. ed. São Paulo: Pearson Education, 2016.

0

Ver anotações

LOIANE GRONER. **Curso de Java 67:** criando threads + métodos start, run e sleep.

2016. Disponível em: <https://bit.ly/2HaVs9f>. Acesso em: 8 set. 2020.

LOIANE GRONER. **Curso de Java 68:** threads: interface runnable. 2016. Disponível em: <https://bit.ly/32Ghd9m>. Acesso em: 8 set. 2020.

LOIANE GRONER. **Curso de Java 69:** criando várias threads + métodos isAlive e join. Disponível em: <https://bit.ly/3iIR1k5>. 2016. Acesso em: 8 set. 2020.

ORACLE. **Enum Thread State.** [s.d.]. Disponível em: <https://bit.ly/32IZii0>. Acesso em: 8 set. 2020.

ORACLE. **Thread.** [s.d.]. Disponível em: <https://bit.ly/3mDcVYq>. Acesso em: 8 set. 2020.

FOCO NO MERCADO DE TRABALHO

PROGRAMAÇÃO EM JAVA USANDO *THREADS*

Jesimar da Silva Arantes

Ver anotações

CRIAÇÃO DE *THREADS*

Utilização de *threads* na criação de uma animação de movimento para a máquina carregadora de caixas, independente do movimento do robô.



Fonte: Shutterstock.

Deseja ouvir este material?

Áudio disponível no material digital.

SEM MEDO DE ERRAR

A *startup* em que você trabalha está muito satisfeita com o seu simulador de robô.

Como tarefa final de desenvolvimento, o seu gestor pediu a você que desenvolvesse duas tarefas:

- Criar uma animação em zigue-zague que contivesse a máquina carregadora de caixas em operação.
- Fazer utilização de *threads* nessa animação de forma que o movimento (animação) da máquina carregadora fosse independente do movimento do robô.

o

Ver anotações

Após revisar as sugestões de seu gestor, você, então, decidiu partir do Código 4.14 que havia desenvolvido. A primeira coisa que decidiu fazer foi criar uma classe chamada Carregador para conter a modelagem da máquina que operará na sala com o robô. O Código 4.20 abaixo destaca as principais alterações feitas, sendo que, na classe Carregador, temos quatro atributos (linhas 3 a 6), que são: as posições da máquina, um objeto que contém a referência da imagem usada para sua renderização e a velocidade de operação dela. Repare que a classe Carregador implementa a interface Runnable, assim, o método *run* deve ser sobrescrito (linhas 12 a 34). O código do método *run* é uma adaptação do método *act* que foi desenvolvido utilizando-se a ferramenta Greenfoot na Seção 3 da Unidade 1. A ideia geral é fazer com que essa máquina se move para frente e para trás, em zigue-zague. Poucas alterações foram feitas na classe AppGUI, sendo elas a criação de alguns atributos para inserção da máquina carregadora e a instanciação da *thread* que iniciará os movimentos da máquina carregadora (linhas 47 e 48).

Após esses passos, o seu programa ficou pronto, com todas as alterações requisitadas pelo seu gestor. No Código 4.20, diversas partes foram omitidas, mas o código completo pode ser acessado no GitHub do autor.

Código 4.20 | Criação da classe Carregador e novas modificações na classe AppGUI

0

[Ver anotações](#)

```
1 import javafx.scene.image.ImageView;
2 public class Carregador implements Runnable {
3     private float posicaoX;
4     private float posicaoY;
5     private final ImageView viewCarregador;
6     private final int velocidade = 3;
7     public Carregador(float posX, float posY, ImageView view) {
8         this.posicaoX = posX;
9         this.posicaoY = posY;
10        this.viewCarregador = view;
11    }
12    @Override
13    public void run() {
14        boolean isMoveRight = true;
15        while (true) {
16            if (isMoveRight) {
17                posicaoX += velocidade;
18            } else {
19                posicaoX -= velocidade;
20            }
21            if (isMoveRight && posicaoX > 350) {
22                isMoveRight = false;
23            } else if (!isMoveRight && posicaoX < 190) {
24                isMoveRight = true;
25            }
26            viewCarregador.setTranslateX(posicaoX);
27            viewCarregador.setTranslateY(posicaoY);
28            try {
29                Thread.sleep(100);
30            } catch (InterruptedException ex) {
31                System.out.println(ex);
32            }
33        }
34    }
35 }
36
37 ... //as importações foram omitidas
38 public class AppGUI extends Application {
39     ... //algumas declarações foram omitidas
40     private final String IMG_CAR = "recursos/Carregador.png";
```

```

41     private final Image imgCar = new Image(getClass()
42                         .getResourceAsStream(IMG_CAR));
43     private final ImageView viewCar = new ImageView(imgCar);
44     private final Carregador car = new Carregador(250, 270, viewCar);
45     @Override
46     public void start(Stage janela) {
47         Thread threadCarregador = new Thread(car);
48         threadCarregador.start();
49         ... //algumas linhas de código foram omitidas
50         grupo.getChildren().add(viewCarregador);
51         ... //algumas linhas de código foram omitidas
52     }
53 }
```

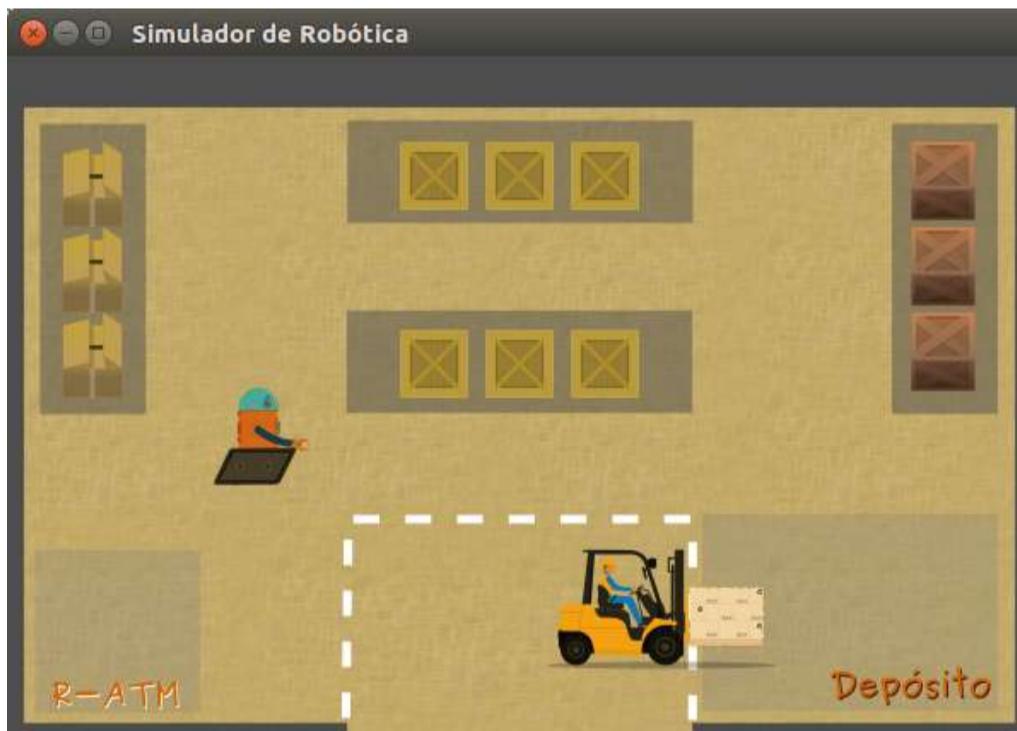
0

[Ver anotações](#)

Fonte: elaborado pelo autor.

Ao executar o Código 4.20, uma tela semelhante à Figura 4.15 deve ser mostrada.

Figura 4.15 | Interface gráfica final do simulador de robótica com a máquina carregadora



Fonte: elaborada pelo autor.

Parabéns, você chegou ao final deste estudo sobre Linguagens Orientadas a Objetos com foco em Java. Acreditamos que o seu estudo sobre OO e Java ainda não terminou, pois ainda há muito o que aprender sobre esse imenso universo, mas acreditamos que as principais bases foram transmitidas.

Boa sorte em sua jornada pelo aprendizado.