Linguagem de Programação Orientada a Objetos

Caros alunos, as vídeo aulas desta disciplina encontram-se no AVA (Ambiente Virtual de Aprendizagem).

UniFil

Unidade 4

Introdução da Unidade

Nesta unidade continuamos a desvendar o mundo da orientação objeto, começando com uma discussão sobre coleções e agrupamentos.

Objetivos

- Aprender sobre como escrever a documentação de seu código;
- Exemplificar sobre os modificadores de acesso público e privado;
- Abordar sobre os métodos estáticos e Constantes;
- Apresentar os métodos de classe.

Conteúdo programático

Aula 01 – Comportamentos sofisticados

Aula 02 – Público versus Privado

Referências

BARNES, D. J.; KLLING, M. **Objects First with Java**: A Practical Introduction Using BlueJ. 6th edition. USA: Prentice Hall Press, 2017.

BlueJ - A free Java Development Environment designed for beginners. Disponível em: <https://bluej.org>. Acesso em: 18 maio. 2020.

Repositório de Projetos. Disponível em: <https://www.bluej.org/objects-first/resources/projects.zip>. Acesso em: 18 maio. 2020.

The BlueJ Tutorial (Portuguese). PDF. Translated by João Luiz Silva Barbosa. Disponível em: <https://www.bluej.org/tutorial/tutorial-portuguese.pdf>. Acesso em: 18 maio. 2020.

Aula 01 - Comportamentos sofisticados

Na aula 01, da unidade 3, introduzimos a classe *ArrayList* da biblioteca de classes Java. Discutimos como isso nos permitiu fazer algo que seria difícil de alcançar (nesse caso, armazenar um número arbitrário de objetos).

Este foi apenas um exemplo único de uma classe útil da biblioteca Java. A biblioteca consiste em milhares de classes, muitas das quais geralmente são úteis para o seu trabalho, e muitas delas você provavelmente nunca vai utilizar.

Para um bom programador Java, é essencial poder trabalhar com a biblioteca Java e fazer escolhas informadas sobre quais classes usar. Depois de iniciar o trabalho com a biblioteca, você verá, rapidamente, que ela permite executar muitas tarefas com mais facilidade do que seria possível. Aprender a trabalhar com as aulas da biblioteca é o tópico principal deste capítulo.

Os itens da biblioteca não são apenas um conjunto de classes arbitrárias e não relacionadas que todos temos que aprender individualmente, mas geralmente são organizados em relacionamentos, explorando características comuns. Aqui, novamente encontramos o conceito de abstração para nos ajudar a lidar com uma grande quantidade de informações. Algumas das partes mais importantes da biblioteca são as coleções, das quais a classe *ArrayList* faz parte. Discutiremos outros tipos de coleções nesta aula e veremos que eles compartilham muitos atributos entre si, para que possamos abstrair os detalhes individuais de uma coleção específica e conversar sobre as classes de coleção em geral.

Novas classes de coleção, bem como algumas outras classes úteis, serão introduzidas e discutidas. Ao longo deste capítulo, trabalharemos na construção de um único aplicativo (o sistema *TechSupport*), que utiliza várias classes de bibliotecas diferentes. Uma implementação completa contendo todas as ideias e códigos fonte discutidos aqui, bem como várias versões intermediárias, está incluída nos projetos, que podem ser encontrados nas referências. Embora isso permita que você estude a solução completa. Depois de uma breve olhada no programa completo, eles começarão com uma versão inicial muito simples do projeto e, gradualmente, desenvolverão e implementarão a solução completa.

O aplicativo faz uso de várias novas classes e técnicas de bibliotecas, cada uma exigindo estudo individual, como mapas de *hash*, conjuntos, “tokenização” de *strings* e uso adicional de números aleatórios. Você deve estar ciente de que este não é um capítulo para ser lido e entendido em um único dia, mas contém várias seções que merecem alguns dias de estudo cada.

6.1 Documentação para classes de biblioteca

A biblioteca de classes padrão do Java é extensa. Ele consiste em milhares de classes, cada uma das quais possui muitos métodos, com e sem parâmetros e com e sem tipos de retorno. É impossível memorizar todos eles e todos os detalhes que os acompanham. Em vez disso, um bom programador Java deve saber: algumas das classes mais importantes e seus métodos da biblioteca por nome (*ArrayList* é uma daquelas importantes) e como descobrir outras classes e procurar os detalhes (como métodos e parâmetros).

Aqui, apresentaremos algumas das classes importantes da biblioteca de classes e outras classes serão apresentadas ao longo do material. Mais importante, mostraremos como você pode explorar e entender a biblioteca por conta própria. Isso permitirá que você escreva programas muito mais interessantes. Felizmente, a biblioteca Java está muito bem documentada. Esta documentação está disponível no formato HTML (para que possa ser lida em um navegador da *web*). Usaremos isso para descobrir sobre as classes da biblioteca.

Ler e entender a documentação é a primeira parte de nossa introdução às aulas da biblioteca. Iremos dar um passo adiante nessa abordagem e também discutiremos como preparar nossas próprias classes para que outras pessoas possam usá-las da mesma maneira que usariam classes de biblioteca padrão. Isso é importante para o desenvolvimento de *software* no mundo real, onde as equipes precisam lidar com grandes projetos e manutenção de *software* ao longo do tempo.

Uma coisa que você deve ter notado sobre a classe *ArrayList* é que a usamos sem nunca olhar para o código fonte. Não verificamos como foi implementado. Isso não era necessário para utilizar sua funcionalidade. Tudo o que precisávamos saber era o nome da classe, os nomes dos métodos, os parâmetros e tipos de retorno desses métodos, e o que exatamente esses métodos fazem. Nós realmente não nos importamos com o trabalho. Isso é típico para o uso de classes de biblioteca.

O mesmo vale para outras classes em projetos de *software* maiores. Normalmente, várias pessoas trabalham juntas em um projeto lidando em diferentes partes. Cada programador deve se concentrar em sua própria área e não precisa entender os detalhes de todas as outras partes (discutimos isso na Seção 3.2, onde falamos sobre abstração e modularização).

De fato, cada programador deve poder usar as classes de outros membros da equipe como se fossem classes da biblioteca, fazendo uso informado delas sem a necessidade de saber como elas funcionam internamente.

Para que isso funcione, cada membro da equipe deve escrever uma documentação sobre sua classe semelhante à documentação da biblioteca padrão Java, que permite que outras pessoas usem a classe sem a necessidade de ler o código. Este tópico também será discutido neste capítulo.

6.2 O sistema *TechSupport*

Como sempre, exploraremos essas questões com um exemplo. Desta vez, usaremos o aplicativo *TechSupport*. Você pode encontrá-lo nos projetos do material, sob o nome *tech-support1.*

O *TechSupport* é um programa destinado a fornecer suporte técnico para clientes da empresa de *software "*DodgySoft" fictícia. Há algum tempo, a "DodgySoft" tinha um departamento de suporte técnico com pessoas sentadas nos telefones. Os clientes podem ligar para obter aconselhamento e ajudar com seus problemas técnicos com os produtos da empresa. Recentemente, porém, os negócios não estão indo tão bem, e a "DodgySoft" decidiu se livrar do departamento de suporte técnico para economizar dinheiro. Agora, eles desejam desenvolver o sistema *TechSupport* para dar a impressão de que o suporte ainda é fornecido. O sistema deve imitar as respostas que uma pessoa de suporte técnico pode dar. Os clientes podem se comunicar com o sistema de suporte técnico on-line.

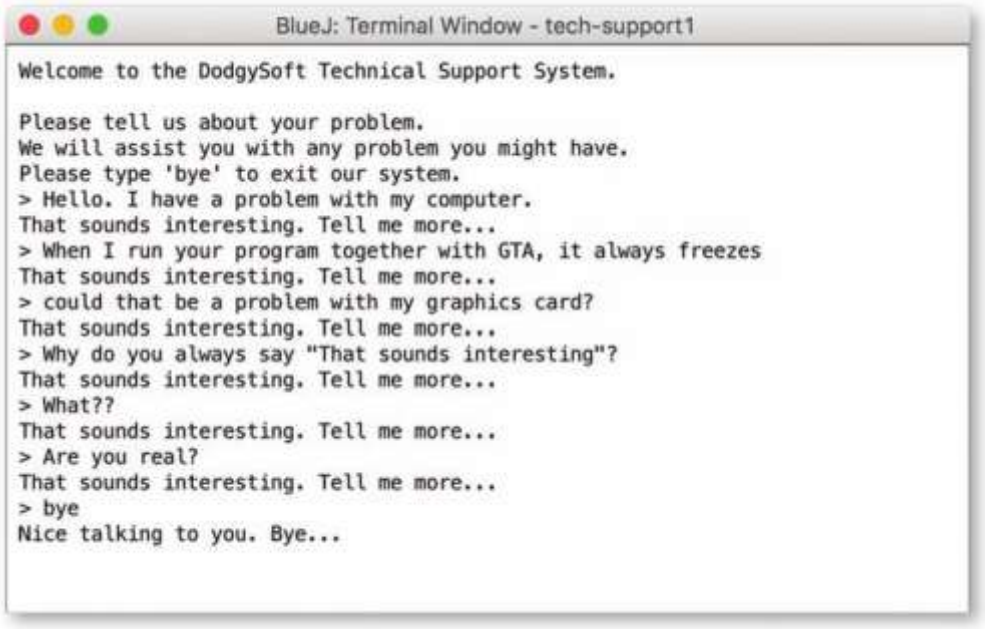
6.2.1 Explorando o sistema *TechSupport*

Digite alguns problemas que você pode estar tendo com seu *software* para testar o sistema. Veja como se comporta. Digite "tchau" quando terminar. Você não precisa examinar o código fonte nesta fase. Este projeto é a solução completa que teremos desenvolvido até o final desta aula. O objetivo deste exercício é apenas dar uma ideia do que planejamos alcançar.

A ideia do projeto *TechSupport* é baseada no inovador programa de inteligência artificial Eliza, desenvolvido por Joseph Weizenbaum no Massachusetts Institute of Technology na década de 1960. Você pode descobrir mais sobre o programa original pesquisando na web "Eliza" e "Weizenbaum". Agora iniciaremos nossa exploração mais detalhada usando o projeto *tech-support1*. É uma primeira implementação rudimentar do nosso sistema. Nós o melhoraremos ao longo do capítulo.

Dessa forma, devemos chegar a uma melhor compreensão de todo o sistema do que apenas lendo a solução completa.

Figura 1 – Modelo de perguntas do projeto



Fonte: BlueJ.

O usuário pode digitar uma pergunta e o sistema responde. Tente o mesmo com a nossa versão protótipo do projeto, *tech-support1*.

Na versão completa do *TechSupport*, o sistema consegue produzir respostas razoavelmente variadas, às vezes até parecem fazer sentido! Na versão do protótipo que estamos usando como ponto de partida, as respostas são muito mais restritas (Figura 1). Você notará muito rapidamente que a resposta é sempre a mesma:

"Isso parece interessante. Conte-me mais..."

Na verdade, isso não é muito interessante e nem muito convincente ao tentar fingir que temos uma pessoa de suporte técnico no outro extremo do diálogo. Em breve, tentaremos melhorar isso. No entanto, antes de fazer isso, exploraremos mais o que temos até agora.

O diagrama do projeto mostra três classes: *SupportSystem*, *InputReader* e Responder (Figura 2). *SupportSystem* é a classe principal, que usa o *InputReader* para obter alguma entrada do terminal e o Respondente para gerar uma resposta.

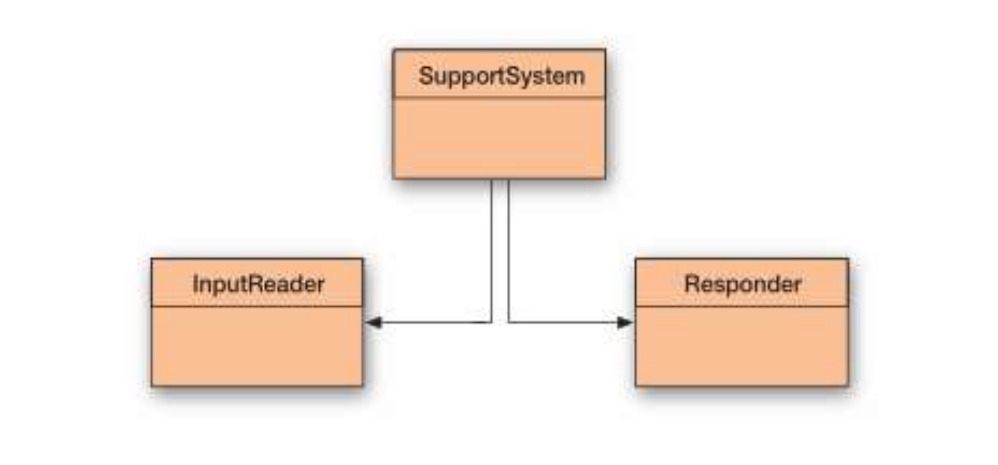
Examine o *InputReader* ainda mais, criando um objeto dessa classe e examinando os métodos do objeto. Você verá que ele tem apenas um único método disponível, chamado *getInput*, que retorna uma *string*. Experimente. Esse método permite digitar uma linha de entrada no terminal e, em seguida, retorna o que você digitou como resultado do método. Não examinaremos como isso funciona internamente neste momento, mas observe que o *InputReader* possui um método *getInput* que retorna uma *string*.

Faça o mesmo com a classe Respondente. Você descobrirá que ele possui um método *generateResponse* que sempre retorna a *string* "Isso parece interessante. Diga-me mais..." (Em inglês: *That sounds interesting. Tell me more*…).

Isso explica o que vimos no diálogo anterior.

Agora, vejamos a classe *SupportSystem* um pouco mais de perto.

Figura 2 – As classes



Fonte: BlueJ.

6.2.2 Lendo o código

O código fonte completo da classe *SupportSystem* é mostrado no Código 6.1. O código 6.2 mostra o código fonte da classe Respondente.

Código 6.1

```
/**
* This class implements a technical support system. It is the top level class
* in this project. The support system communicates via text input/output
* in the text terminal.
*
* This class uses an object of class InputReader to read input from the user,
* and an object of class Responder to generate responses. It contains a loop
* that repeatedly reads input and generates output until the users wants to
* leave.
*
* @author    Michael Kölling and David J. Barnes
* @version   0.1 (2016.02.29)
*/
public class SupportSystem
{
  private InputReader reader;
  private Responder responder;

  /**
   * Creates a technical support system.
   */
  public SupportSystem()
  {
     reader = new InputReader();
     responder = new Responder();
  }


  /**
   * Start the technical support system. This will print a welcome
   * message and enter into a dialog with the user, until the user
   * ends the dialog.
   */
  public void start()
  {
     boolean finished = false;

     printWelcome();

     while(!finished) {
        String input = reader.getInput();

        if(input.startsWith("bye")) {
           finished = true;
        }
        else {
           String response = responder.generateResponse();
           System.out.println(response);
        }
     }

     printGoodbye();
  }


  /**
   * Print a welcome message to the screen.
   */
  private void printWelcome()
  {
     System.out.println("Welcome to the DodgySoft Technical Support System.");
     System.out.println();
     System.out.println("Please tell us about your problem.");
     System.out.println("We will assist you with any problem you might have.");
     System.out.println("Please type 'bye' to exit our system.");
  }


  /**
   * Print a good-bye message to the screen.
   */
  private void printGoodbye()
  {
     System.out.println("Nice talking to you. Bye...");
  }
}
```

Observando o Código 6.2, vemos que a classe Respondente é trivial. Ele possui apenas um método e sempre retorna a mesma *string*. Isso é algo que melhoraremos mais tarde. Por enquanto, vamos nos concentrar na classe *SupportSystem*.

O *SupportSystem* declara dois campos de instância para armazenar um *InputReader* e um objeto "Responder", e atribui esses dois objetos ao construtor.

No final, ele tem dois métodos chamados *printWelcome* e *printGoodbye*. Eles simplesmente imprimem algum texto, uma mensagem de boas-vindas e uma mensagem de despedida, respectivamente.

A parte mais interessante do código é o método no meio: *start*. Vamos discutir esse método com mais detalhes.

Código 6.2

```
/**
* The responder class represents a response generator object.
* It is used to generate an automatic response to an input string.
*
* @author    Michael Kölling and David J. Barnes
* @version   0.1 (2016.02.29)
*/
public class Responder
{
   /**
    * Construct a Responder - nothing to do
    */
   public Responder()
   {
   }


   /**
    * Generate a response.
    * @return   A string that should be displayed as the response
    */
   public String generateResponse()
   {
      return "That sounds interesting. Tell me more...";
   }
}
```

Na parte superior do método, há uma chamada para *printWelcome* e, no final, uma chamada para *printGoodbye*. Essas duas chamadas cuidam da impressão dessas seções de texto nos horários apropriados. O restante deste método consiste em uma declaração de uma variável booleana e um *loop while*. A estrutura é a seguinte:

```
boolean finished = false;
while(!finished) {
 faça algo
 if(condição de saída) {
   finished = true;
 }
```

```
  else {
    faca algo mais
  }
}
```

Esse padrão de código é uma variação do idioma *while-loop* discutido na Seção 4.10. Usamos terminar como um sinalizador que se torna verdadeiro quando queremos terminar o *loop* (e com ele, o programa inteiro). Garantimos que seja inicialmente falso. (Lembre-se de que o ponto de exclamação não é um operador!)

A parte principal do loop, a parte que é feita repetidamente enquanto desejamos continuar, consiste em três instruções, se retirarmos a verificação da condição de saída:

```
String input = reader.getInput ();
...
String response = responder.generateResponse();
System.out.println(response);
```

Assim, o loop repetidamente:

- Lê alguma entrada do usuário;
- Pede ao respondente para gerar uma resposta;
- Exibe essa resposta.

(Você deve ter notado que a resposta não depende da entrada! Isso certamente é algo que teremos que melhorar mais tarde.)

A última parte a examinar é a verificação da condição de saída. A intenção é que o programa termine assim que o usuário digitar a palavra "*bye*". A seção relevante do código fonte que encontramos na classe Iê:

```
String input = reader.getInput();
if(input.startsWith("bye")) {
  finished = true;
}
```

Se você entender essas partes isoladamente, é uma boa ideia examinar novamente o método de início completo no Código 6.1 e verificar se você pode entender todos juntos.

No último fragmento de código examinado acima, é usado um método chamado *beginWith*. Como esse método é chamado na variável de entrada, que contém um objeto *String*, ele deve ser um método da classe *String*. Mas o que esse método faz? E como descobrimos? Podemos adivinhar, simplesmente vendo o nome do método, que ele testa se a *string* de entrada começa com a palavra "*bye*". Podemos verificar isso por experiência. Execute o sistema *TechSupport* novamente e digite "*bye bye*" ou "*bye everyone*". Você notará que ambas as versões fazem com que o sistema saia.

Observe, no entanto, que digitar "*Bye*" ou " *bye*" começando com uma letra maiúscula ou com um espaço na frente da palavra não é reconhecido como começando com "*bye*". Isso pode ser um pouco chato para um usuário, mas podemos resolver esses problemas se soubermos um pouco mais sobre a classe *String*.

Como descobrimos mais informações sobre o método *winsWith* ou outros métodos da classe *String*?

6.3 Lendo a documentação da classe

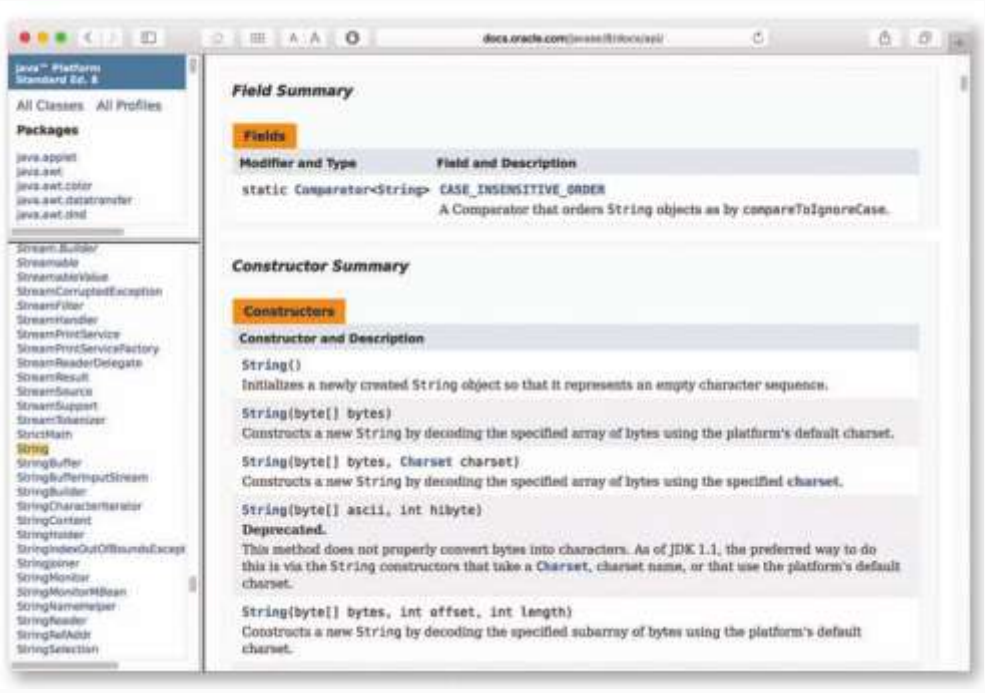
A classe *String* é uma das classes da biblioteca de classes Java padrão. Podemos descobrir mais detalhes lendo a documentação da biblioteca para a classe *String*.

Para fazer isso, escolha o item Bibliotecas de classes Java, no menu Ajuda do BlueJ. Isso abrirá um navegador da Web que exibe a página principal da documentação da API Java (interface de programação de aplicativos).

Por padrão, essa função acessa a documentação através da Internet. Isso não funcionará se sua máquina não tiver acesso à rede. No entanto, o BlueJ pode ser configurado para usar uma cópia local da documentação da API Java. Isso é recomendado, porque acelera o acesso e pode funcionar sem conexão à Internet.

O navegador da web exibirá três quadros. Na parte superior esquerda, você vê uma lista de pacotes. Abaixo disso, há uma lista de todas as classes na biblioteca Java. O quadro grande à direita é usado para exibir detalhes de um pacote ou classe selecionada.

Figura 3 – Documentação Java



Fonte: BlueJ.

Na lista de classes à esquerda, localize e selecione a classe *String*. O quadro à direita exibe a documentação da classe *String* (Figura 3).

6.3.1 Interfaces versus implementação

Você verá que a documentação inclui diferentes informações. Eles são, entre outras coisas:

- O nome da classe;
- Uma descrição geral do objetivo da classe;
- Uma lista dos construtores e métodos da classe;
- Os parâmetros e tipos de retorno para cada construtor e método;
- Uma descrição do objetivo de cada construtor e método.

Essas informações, juntas, são chamadas de interface de uma classe. Observe que a interface não mostra o código fonte que implementa a classe. Se uma classe estiver bem descrita (ou seja, se sua interface estiver bem escrita), um programador não precisará ver o código fonte para poder usar a classe. Ver a interface fornece todas as informações necessárias. Isso é a abstração em ação novamente.

O código fonte nos bastidores, que faz a classe funcionar, é chamado de implementação da classe. Normalmente, um programador trabalha na implementação de uma classe de cada vez, enquanto faz uso de várias outras classes através de suas interfaces.

Essa distinção entre a interface e a implementação é um conceito muito importante, e surgirá repetidamente neste e nos próximos capítulos deste material.

Nota: palavra interface possui vários significados no contexto de programação e Java.

É usado para descrever a parte publicamente visível de uma classe (que é como acabamos de usá-la aqui), mas também tem outros significados. A interface do usuário (geralmente uma interface gráfica do usuário) às vezes é chamada apenas de interface, mas o Java também possui uma construção de linguagem chamada interface.

É importante ser capaz de distinguir entre os diferentes significados da palavra interface em um contexto específico.

A terminologia da interface também é usada para métodos individuais. Por exemplo, a documentação *String* mostra a interface do método *length*:

public int length ()

Retorna o comprimento dessa *string*. O comprimento é igual ao número de unidades de código “Unicode” na cadeia de caracteres.

Especificado por: comprimento na interface CharSequence.

Retorna: o comprimento da sequência de caracteres representados por este objeto.

A interface de um método consiste no cabeçalho do método e um comentário (mostrado aqui em itálico). O cabeçalho de um método inclui (nesta ordem):

- Um modificador de acesso (aqui público), que discutiremos abaixo;
- O tipo de retorno do método (aqui int);
- O nome do método;

- Uma lista de parâmetros (que está vazia neste exemplo); o nome e os parâmetros juntos também são chamados de assinatura do método.

A interface fornece tudo o que precisamos saber para fazer uso desse método.

6.3.2 Usando métodos de classe de biblioteca

Voltar ao nosso sistema *TechSupport*. Agora, queremos melhorar um pouco o processamento de entrada.

Vimos na discussão acima que nosso sistema não é muito tolerante: se digitarmos "*Bye*" ou " *bye*" em vez de "*bye*", por exemplo, a palavra não será reconhecida. Queremos mudar isso ajustando o texto lido de um usuário para que essas variações sejam reconhecidas como "*tchau*".

A documentação da classe *String* nos diz que ele possui um método chamado “trim” para remover espaços no início e no final da *string*. Podemos usar esse método para lidar com o segundo caso de problema.

Anote o cabeçalho desse método. Anote uma chamada de exemplo para esse método em uma variável *String* chamada texto.

Um detalhe importante sobre os objetos *String* é que eles são imutáveis, ou seja, não podem ser modificados depois de criados. Observe com cuidado que o método “trim”, por exemplo, retorna uma nova *string*; não modifica a cadeia original. Preste muita atenção ao seguinte comentário "*Pitfall*".

Armadilha: um erro comum em Java é tentar modificar uma *string*, por exemplo, escrevendo

input.toUpperCase();

Isso está incorreto (as *strings* não podem ser modificadas), mas infelizmente isso não produz um erro.

A instrução simplesmente não tem efeito e a sequência de entrada permanece inalterada.

O método *toUpperCase*, assim como outros métodos de sequência, não modifica a sequência original, mas retorna uma nova sequência semelhante a original, com algumas alterações aplicadas (aqui, com caracteres alterados para maiúsculas). Se queremos que nossa variável de entrada seja alterada, precisamos atribuir esse novo objeto de volta à variável (descartando o original), assim:

input = input.toUpperCase ();

O novo objeto também pode ser atribuído a outra variável ou processado de outras maneiras.

Depois de estudar a interface do método “trim”, podemos ver que podemos remover os espaços de uma *string* de entrada com a seguinte linha de código:

input = input.trim ();

Esse código solicitará que o objeto *String* armazenado na variável de entrada crie uma nova sequência semelhante com os espaços à esquerda e à direita, removidos. A nova *String* é então

armazenada na variável de entrada porque não temos mais uso para a antiga. Portanto, após essa linha de código, a entrada se refere a uma sequência sem espaços em cada extremidade.

Agora podemos inserir esta linha em nosso código fonte para que ele leia:

```
String input = reader.getInput();
input = input.trim();
if (input.startsWith("bye")) {
  finished = true;
}
else {
  Código omitido.
}
```

As duas primeiras linhas também podem ser mescladas em uma única linha:

String input = reader.getInput (). Trim ();

O efeito dessa linha de código é idêntico ao das duas primeiras linhas acima. O lado direito deve ser lido como se estivesse entre parênteses da seguinte maneira:

(reader.getInput()).trim()

Qual versão você prefere é principalmente uma questão de gosto. A decisão deve ser tomada principalmente com base na legibilidade: use a versão que você achar mais fácil de ler e entender.

Frequentemente, programadores iniciantes preferem a versão de duas linhas, enquanto programadores mais experientes se acostumam ao estilo de uma linha.

Agora, resolvemos o problema causado pelos espaços ao redor da entrada, mas ainda não resolvemos o problema com letras maiúsculas. No entanto, uma investigação mais aprofundada da documentação da classe *String* sugere uma possível solução, porque descreve um método chamado: toLowerCase.

6.3.3 Verificando a igualdade das *strings*

Uma solução alternativa seria verificar se a sequência de entrada é a sequência "*bye*", ao invés de começar com a sequência "*bye*". Uma tentativa (incorreta!) De escrever esse código pode ter a seguinte aparência:

```
if (input == "bye") {// nem sempre funciona!
  ...
}
```

O problema aqui é que é possível a existência de vários objetos *String* independentes, que representam o mesmo texto. Dois objetos *String*, por exemplo, podem conter os caracteres "*tchau*". O operador de igualdade (==) verifica se cada lado do operador se refere ao mesmo objeto, não se eles têm o mesmo valor! Essa é uma diferença importante.

Em nosso exemplo, estamos interessados na questão de saber se a variável de entrada e a constante de *string* "*bye*" representam o mesmo valor, não se elas se referem ao mesmo objeto. Portanto, o uso do operador == está errado. Pode retornar “false”, mesmo que o valor da

variável de entrada seja "*bye*". A solução é usar o método *equals*, definido na classe *String*. Este método testa corretamente se o conteúdo de dois objetos *String* é o mesmo. O código correto diz:

```
if (input.equals("bye")) {
  ...
}
```

Obviamente, isso também pode ser combinado com os métodos “trim” e “*toLowerCase*”.

Armadilha: a comparação de *strings* com o operador “==” pode levar a resultados não intencionais. Como regra geral, as *strings* quase sempre devem ser comparadas como iguais, em vez de com o operador “==”.

6.4 Adicionando comportamento aleatório

Até o momento, fizemos uma pequena melhoria no projeto *TechSupport*, mas, no geral, ele permanece muito básico. Um dos principais problemas é que ele sempre dá a mesma resposta, independentemente da entrada do usuário. Agora, melhoraremos isso definindo um conjunto de frases plausíveis com as quais responder. Em seguida, o programa escolherá aleatoriamente um deles cada vez que se espera que ele responda. Esta será uma extensão da classe Respondente em nosso projeto.

Para fazer isso, usaremos um *ArrayList* para armazenar algumas *strings* de resposta, gerar um número inteiro aleatório e usar o número aleatório como um índice na lista de respostas para escolhermos uma de nossas frases. Nesta versão, a resposta ainda não dependerá da entrada do usuário (faremos isso mais tarde), mas pelo menos variará a resposta e parecerá muito melhor.

Primeiro, temos que descobrir como gerar um número inteiro aleatório.

Aleatório e pseudoaleatório

A geração de números aleatórios em um computador não é realmente tão fácil de fazer como se poderia pensar inicialmente. Como os computadores operam de uma maneira determinística muito bem definida, que se baseia no fato de que todo o cálculo é previsível e repetível, eles fornecem pouco espaço para um comportamento aleatório real.

Os pesquisadores, ao longo do tempo, propuseram muitos algoritmos para produzir sequências aparentemente aleatórias de números. Esses números geralmente não são aleatórios, mas seguem regras complicadas. Eles são, portanto, referidos como: números pseudoaleatórios.

Em uma linguagem como Java, a geração de números pseudoaleatórios felizmente foi implementada em uma classe de biblioteca, então tudo o que precisamos fazer para receber um número pseudoaleatório é fazer algumas chamadas para a biblioteca.

Se você quiser ler mais sobre isso, faça uma pesquisa na *web* por "números pseudoaleatórios".

6.4.1 A classe aleatória

A biblioteca de classes Java contém uma delas chamada *Random,* que será útil para o nosso projeto.

Em que pacote está? O que isso faz? Como você constrói uma instância? Como você gera um número aleatório? Observe que você provavelmente não entenderá tudo o que está indicado na documentação. Apenas tente descobrir o que você precisa saber.

Para gerar um número aleatório, precisamos:

- Criar uma instância da classe Random;
- Fazer uma chamada para um método dessa instância para obter um número.

Observando a documentação, vemos que existem vários métodos chamados *next Something* para gerar valores aleatórios de vários tipos. O que gera um número inteiro aleatório é chamado *nextInt*.

A seguir, ilustra o código necessário para gerar e imprimir um número inteiro aleatório:

```
Random randomGenerator;

randomGenerator = new Random();

int index = randomGenerator.nextInt();

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

Esse fragmento de código cria uma nova instância da classe *Random* e a armazena na variável *randomGenerator*. Em seguida, chama o método *nextInt* para receber um número aleatório, armazena-o na variável de índice e, eventualmente, o imprime.

Sua classe deve criar apenas uma única instância da classe *Random* (em seu construtor) e armazená-la em um campo. Não crie uma nova instância aleatória sempre que desejar um novo número.

6.4.2 Números aleatórios com alcance limitado

Os números aleatórios que vimos até agora foram gerados a partir de todo o intervalo de números inteiros Java (-2147483648 a 2147483647). Isso é bom como experimento, mas raramente é útil.

Mais frequentemente, queremos números aleatórios dentro de um determinado intervalo limitado.

A classe *Random* também oferece um método para suportar isso. É novamente chamado *nextInt*, mas possui um parâmetro para especificar o intervalo de números que gostaríamos de usar.

6.4.3 Gerando respostas aleatórias

Agora, podemos estender a classe “Respondente” para selecionar uma resposta aleatória de uma lista de frases predefinidas. O código 6.2 mostra o código fonte da classe “Respondente”, como está em nossa primeira versão.

Agora, adicionaremos o código a esta primeira versão:

- Declare um campo do tipo *Random* para conter o gerador de números aleatórios;
- Declare um campo do tipo *ArrayList* para conter nossas possíveis respostas;
- Crie os objetos *Random* e *ArrayList* no construtor "Responder";
- Preencha a lista de respostas com algumas frases;
- Selecione e retorne uma frase aleatória quando *generateResponse* for chamado.

O código 6.3 mostra uma versão do código fonte do "Respondente" com essas adições.

Código 6.3

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

/**
* The responder class represents a response generator object. It is used
* to generate an automatic response. This is the second version of this
* class. This time, we generate some random behavior by randomly selecting
* a phrase from a predefined list of responses.
*
* @author     Michael Kölling and David J. Barnes
* @version    0.2 (2016.02.29)
*/
public class Responder
{
  private Random randomGenerator;
  private ArrayList<String> responses;

  /**
   * Construct a Responder
   */
  public Responder()
  {
     randomGenerator = new Random();
     responses = new ArrayList<>();
     fillResponses();
  }

  /**
   * Generate a response.
   *
   * @return  A string that should be displayed as the response
   */
  public String generateResponse()
  {
     // Pick a random number for the index in the default response
     // list. The number will be between 0 (inclusive) and the size
     // of the list (exclusive).
     int index = randomGenerator.nextInt(responses.size());
     return responses.get(index);
  }
```

```
    /**
     * Build up a list of default responses from which we can pick one
     * if we don't know what else to say.
     */
    private void fillResponses()
    {
        responses.add("That sounds odd. Could you describe this in more detail?");
        responses.add("No other customer has ever complained about this \n" +
                      "before. What is your system configuration?");
        responses.add("I need a bit more information on that.");
        responses.add("Have you checked that you do not have a dll conflict?");
        responses.add("That is covered in the manual. Have you read the manual?");
        responses.add("Your description is a bit wishy-washy. Have you got \n" +
                      "an expert there with you who could describe this better?");
        responses.add("That's not a bug, it's a feature!");
        responses.add("Could you elaborate on that?");
        responses.add("Have you tried running the app on your phone?");
        responses.add("I just checked StackOverflow - they don't know either.");
    }
}
```

Nesta versão, colocamos o código que preenche a lista de respostas em seu próprio método, chamado *fillResponses*, que é chamado a partir do construtor. Isso garante que a lista de respostas seja preenchida assim que um objeto "Respondente" for criado, mas o código fonte para preencher a lista é mantido separado para facilitar a leitura e compreensão da classe.

O segmento de código mais interessante dessa classe está no método *generateResponse*.

Deixando de fora os comentários, ele lê:

```
public String generateResponse()
{
  int index = randomGenerator.nextInt(answers.size());
  return answers.get(index);
}
```

A primeira linha de código neste método faz três coisas:

- É obtido o tamanho da lista de respostas chamando seu método de tamanho;
- Gere um número aleatório entre 0 (inclusive) e o tamanho (exclusivo);
- Armazena esse número aleatório no índice de variável local.

Se isso parecer muito código para uma linha, você também pode escrever:

```
int listSize = answers.size ();

int index = randomGenerator.nextInt (listSize);
```

Este código é equivalente à primeira linha acima. Qual versão você prefere, novamente, depende qual você acha mais fácil de ler.

É importante observar que esse segmento de código irá gerar um número aleatório no intervalo "0" para *listSize*-1 (inclusive). Isso se encaixa perfeitamente com os índices legais de um *ArrayList*.

Lembre-se de que o intervalo de índices para um *ArrayList* de tamanho *listSize* é “0” para *listSize*-1. Assim, o número aleatório calculado nos fornece um índice perfeito para acessar aleatoriamente um do conjunto completo dos elementos da lista.

A última linha do método lê:

return answers.get (index);

Essa linha faz duas coisas:

- Recupera a resposta no índice de posição usando o método *get*.
- Ele retorna a *string* selecionada como resultado do método, usando a instrução *return*.

Se você não tomar cuidado, seu código pode gerar um número aleatório que esteja fora dos índices válidos do *ArrayList*. Quando você tenta usá-lo como um índice para acessar um elemento da lista, você obtém uma *IndexOutOfBoundsException*.

6.4.4 Lendo a documentação para classes parametrizadas

Até o momento, solicitamos que você analise a documentação da classe *String* do pacote *java*.*lang* e a classe *Random* do pacote *java*.*util*. Você deve ter notado ao fazer isso que alguns nomes de classe na lista de documentação parecem um pouco diferentes, como *ArrayList* <E> ou *HashMap* <K, V>. Ou seja, o nome da classe é seguido por algumas informações extras entre colchetes angulares. Classes que se parecem com isso são chamadas de classes parametrizadas ou classes genéricas. As informações entre colchetes nos dizem que, quando usamos essas classes, devemos fornecer um ou mais nomes de tipo entre colchetes angulares para concluir a definição. Já vimos essa ideia anteriormente, onde usamos *ArrayList* parametrizando com nomes de tipos, como *String*. Eles também podem ser parametrizados com qualquer outro tipo:

private ArrayList<String> notes;

private ArrayList<Student> students;

Como podemos parametrizar um *ArrayList* com qualquer outro tipo de classe que escolhermos, isso se reflete na documentação da API. Portanto, se você olhar a lista de métodos para *ArrayList* <E>, verá métodos como:

boolean add(E o)

E get(int index)

Isso nos diz que o tipo de objetos que podemos adicionar a um *ArrayList* depende do tipo usado para parametrizá-lo, e o tipo dos objetos retornados de seu método *get* depende desse tipo da mesma maneira. Com efeito, se criarmos um objeto *ArrayList* <> e a documentação nos informar que o objeto possui os dois métodos a seguir:

boolean add(String o)

String get(int index)

Enquanto que, se criarmos um objeto *ArrayList* <*Student*>, ele terá estes dois métodos:

boolean add(Student o)

Student get (int index)

Solicitaremos que você consulte a documentação para obter outros tipos de parâmetros nas seções posteriores.

6.5 Pacotes e importação

Ainda existem duas linhas na parte superior do arquivo de origem que precisamos discutir:

import java.util.ArrayList;

import java.util.Random;

Encontramos a declaração de importação pela primeira vez nas aulas anteriores. Agora é a hora de analisá-la um pouco mais de perto.

As classes Java armazenadas na biblioteca de classes não estão disponíveis automaticamente para uso, como as outras classes no projeto atual. Em vez disso, devemos declarar em nosso código fonte que gostaríamos de usar uma classe da biblioteca. Isso é chamado de importação da classe e é feito usando a instrução de importação. A declaração de importação tem o formato:

import nome-de-classe-qualificado;

Como a biblioteca Java contém milhares de classes, é necessária alguma estrutura na organização da biblioteca para facilitar o tratamento do grande número de classes. Java usa pacotes para organizar as classes da biblioteca em grupos que pertencem um ao outro. Pacotes podem ser aninhados (ou seja, pacotes podem conter outros pacotes).

As classes *ArrayList* e *Random* estão no pacote *java.util*. Esta informação pode ser encontrada na documentação da classe. O nome completo ou o nome qualificado de uma classe é o nome de seu pacote, seguido por um ponto, seguido pelo nome da classe. Portanto, os nomes qualificados das duas classes que usamos aqui são *java.util.ArrayList* e *java.util.Random*.

Java também nos permite importar pacotes completos com instruções no formulário:

import nome do pacote.*;

Portanto, a instrução a seguir importaria todos os nomes de classe do pacote *java.util*:

import java.util.*;

Listar todas as classes usadas separadamente, como em nossa primeira versão, é um pouco mais trabalhoso em termos de digitação, mas serve também como uma parte da documentação. Indica claramente quais classes são realmente usadas por nossa classe. Portanto, tenderemos a usar o estilo do primeiro exemplo, listando todas as classes importadas separadamente.

Há uma exceção a essas regras: algumas classes são usadas com tanta frequência que quase todas as classes as importam. Essas classes foram colocadas no pacote *java.lang* e esse pacote é importado automaticamente para todas as classes. Portanto, não precisamos escrever instruções de importação para classes em *java.lang*. A classe *String* é um exemplo dessa classe.

6.6 Usando mapas para associações

Agora temos uma solução para o nosso sistema de suporte técnico que geram respostas aleatórias.

Isso é melhor que a nossa primeira versão, mas ainda não é muito convincente. Em particular, a entrada do usuário não influencia a resposta de forma alguma. É nesta área que agora queremos melhorar.

O plano é que tenhamos um conjunto de palavras que provavelmente ocorrerão em perguntas típicas e associaremos essas palavras a respostas específicas. Se a entrada do usuário contiver uma de nossas palavras conhecidas, podemos gerar uma resposta relacionada. Esse ainda é um método muito grosseiro, porque não capta nenhum significado da entrada do usuário nem reconhece um contexto. Ainda assim, pode ser surpreendentemente eficaz e é um bom próximo passo.

Para fazer isso, usaremos um *HashMap*. Você encontrará a documentação para a classe *HashMap* na documentação da biblioteca Java. *HashMap* é uma especialização de um Mapa, que você também encontrará documentado. Você verá que precisa ler a documentação de ambos para entender o que é um *HashMap* e como ele funciona.

6.6.1 O conceito de mapa

Um mapa é uma coleção de pares de objetos chave/valor. Como no *ArrayList*, um mapa pode armazenar um número flexível de entradas. Uma diferença entre o *ArrayList* e um mapa é que, com um mapa, cada entrada não é um objeto, mas um par de objetos. Este par consiste em um objeto-chave e um objeto de valor.

Em vez de procurar entradas nesta coleção usando um índice inteiro (como fizemos com o *ArrayList*), usamos o objeto-chave para procurar o objeto de valor.

Um exemplo diário de um mapa é uma lista de contatos. Uma lista de contatos contém entradas e cada entrada é um par: um nome e um número de telefone. Você usa uma lista de contatos procurando um nome e obtendo um número de telefone. Nós não usamos um índice da posição da entrada na lista de contatos para encontrá-lo.

Um mapa pode ser organizado de forma que seja fácil procurar um valor para uma chave. No caso de uma lista de contatos, isso é feito usando a classificação alfabética. Armazenando as entradas na ordem alfabética de suas chaves, é fácil encontrar a chave e procurar o valor. A pesquisa inversa (encontrar a chave para um valor, ou seja, encontrar o nome para um determinado número de telefone) não é tão fácil com um mapa. Como em uma lista de contatos, a pesquisa inversa em um mapa é possível, mas leva um tempo comparativamente longo. Assim, os mapas são ideais para uma pesquisa unidirecional, onde conhecemos a chave de pesquisa e precisamos conhecer um valor associado a essa chave.

6.6.2 Usando um *HashMap*

O *HashMap* é uma implementação específica do Map. Os métodos mais importantes da classe *HashMap* são colocados e obtidos.

O método *put* insere uma entrada no mapa e obtém o valor de uma determinada chave.

O seguinte fragmento de código cria um *HashMap* e insere três entradas nele. Cada entrada é um par de chave/valor que consiste em um nome e um número de telefone.

```
HashMap<String, String> contacts = new HashMap<>();

contacts.put("Charles Nguyen", "(531) 9392 4587");

contacts.put("Lisa Jones", "(402) 4536 4674");

contacts.put("William H. Smith", "(998) 5488 0123");
```

Como vimos em *ArrayList*, ao declarar uma variável *HashMap* e criar um *HashMap* objeto, temos que dizer que tipo de objetos serão armazenados no mapa e, adicionalmente, quais tipos de objetos serão usados para a chave. Para a lista de contatos, usaremos cadeias de caracteres para os dois chaves e valores, mas os dois tipos às vezes serão diferentes.

Como vimos na Seção 4.4.2, ao criar objetos de classes genéricas e atribuí-los a uma variável, precisamos especificar os tipos genéricos (aqui <*String, String*>) apenas uma vez no lado esquerdo da atribuição, e pode usar o operador de diamante na construção do objeto à direita; os tipos genéricos usados para a construção do objeto são copiados da declaração da variável.

O código a seguir, encontrará o número de telefone de Lisa Jones e o imprimirá.

```
String number = contacts.get("Lisa Jones");

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

Observe que você passa a chave (o nome "Lisa Jones") para o método *get* para receber

o valor (o número de telefone).

Leia a documentação dos métodos *get* e *put* da classe *HashMap* novamente e veja se a explicação corresponde ao seu entendimento atual.

6.6.3 Usando um mapa para o sistema *TechSupport*

No sistema *TechSupport*, podemos fazer bom uso de um mapa usando palavras conhecidas como chaves e respostas associadas como valores. O código 6.4 mostra um exemplo no qual um *HashMap* chamado *responseMap* é criado e três entradas são feitas. Por exemplo, a palavra "lento" está associada ao texto "Acho que isso tem a ver com o seu *hardware*. A atualização do processador deve resolver todos os problemas de desempenho. Você tem algum problema com o nosso software?" Agora, sempre que alguém entra em uma pergunta que contém a palavra "lento", podemos procurar e imprimir essa resposta. Observe que a sequência de respostas no código fonte abrange várias linhas, mas é concatenada com o operador +, portanto, uma única sequência é inserida como um valor no *HashMap*.

Código 6.4

```
    private HashMap<String, String> responseMap;
...
/**
 * Construct a Responder
 */
public Responder()
{
    responseMap = new HashMap<>();
    defaultResponses = new ArrayList<>();
    fillResponseMap();
    fillDefaultResponses();
    randomGenerator = new Random();
}

/**
 * Enter all the known keywords and their associated responses
 * into our response map.
 */
private void fillResponseMap()
{
    responseMap.put("crash",
                "Well, it never crashes on our system. It must have something\n" +
                "to do with your system. Tell me more about your configuration.");
    responseMap.put("crashes",
                "Well, it never crashes on our system. It must have something\n" +
                "to do with your system. Tell me more about your configuration.");
    responseMap.put("slow",
                "I think this has to do with your hardware. Upgrading your processor\n" +
                "should solve all performance problems. Have you got a problem with\n" +
                "our software?");
    responseMap.put("performance",
                "Performance was quite adequate in all our tests. Are you running\n" +
                "any other processes in the background?");
    responseMap.put("bug",
                "Well, you know, all software has some bugs. But our software engineers\n" +
                "are working very hard to fix them. Can you describe the problem a bit\n" +
                "further?");
    responseMap.put("buggy",
                "Well, you know, all software has some bugs. But our software engineers\n" +
                "are working very hard to fix them. Can you describe the problem a bit\n" +
                "further?");
    responseMap.put("windows",
                "This is a known bug to do with the Windows operating system. Please\n" +
                "report it to Microsoft. There is nothing we can do about this.");
    responseMap.put("mac",
                "This is a known bug to do with the Mac operating system. Please\n" +
                "report it to Apple. There is nothing we can do about this.");
    responseMap.put("expensive",
                "The cost of our product is quite competitive. Have you looked around\n" +
                "and really compared our features?");
    responseMap.put("installation",
                "The installation is really quite straight forward. We have tons of\n" +
                "wizards that do all the work for you. Have you read the installation\n" +
                "instructions?");
    responseMap.put("memory",
                "If you read the system requirements carefully, you will see that the\n" +
                "specified memory requirements are 1.5 giga byte. You really should\n" +
                "upgrade your memory. Anything else you want to know?");
    responseMap.put("linux",
                "We take Linux support very seriously. But there are some problems.\n" +
                "Most have to do with incompatible glibc versions. Can you be a bit\n" +
                "more precise?");
    responseMap.put("bluej",
                "Ahhh, BlueJ, yes. We tried to buy out those guys long ago, but\n" +
                "they simply won't sell... Stubborn people they are. Nothing we can\n" +
                "do about it, I'm afraid.");
}
```

Uma primeira tentativa de escrever um método para gerar as respostas agora pode se parecer com o método *generateResponse,* abaixo. Aqui, para simplificar as coisas no momento, supomos que apenas uma única palavra (por exemplo, "lenta") seja inserida pelo usuário.

```
public String generateResponse(String word)
{
  String response = responseMap.get(word);
  if(response != null) {
    return response;
  }
  else {
    // If we get here, the word was not recognized. In
    // this case, we pick one of our default responses.
    return pickDefaultResponse();
  }
}
```

Nesse fragmento de código, procuramos a palavra inserida pelo usuário em nosso mapa de respostas. Se encontrarmos uma entrada, a usamos como resposta. Se não encontrarmos uma entrada para essa palavra, chamaremos um método chamado *pickDefaultResponse*. Agora, esse método pode conter o código da versão anterior do *generateResponse*, que seleciona aleatoriamente uma das respostas padrão (conforme mostrado no Código 6.3). A nova lógica, então, é que escolhemos uma resposta apropriada se reconhecermos uma palavra ou uma resposta aleatória da nossa lista de respostas padrão, se não reconhecermos.

Essa abordagem de associação de palavras-chave a respostas funciona muito bem, desde que o usuário não insira perguntas completas, mas apenas palavras únicas. A melhoria final para concluir o aplicativo é permitir que o usuário insira as perguntas completas novamente e escolha respostas correspondentes, se reconhecermos alguma das palavras nas perguntas.

Isso coloca o problema de reconhecer as palavras-chave na frase que foi inserida pelo usuário. Na versão atual, a entrada do usuário é retornada pelo *InputReader* como uma única sequência. Agora vamos mudar isso para uma nova versão na qual o *InputReader* retorna a entrada como um conjunto de palavras. Tecnicamente, esse será um conjunto de cadeias, em que cada cadeia no conjunto representa uma única palavra que foi inserida pelo usuário.

Se pudermos fazer isso, podemos passar o conjunto inteiro para o “Respondente”, que poderá verificar todas as palavras do conjunto para ver se é conhecido e tem uma resposta associada.

Para conseguir isso em Java, precisamos saber sobre duas coisas: como cortar uma única *string* contendo uma frase inteira em palavras e como usar conjuntos. Essas questões serão discutidas nas próximas duas seções.

6.7 Usando conjuntos

A biblioteca padrão Java inclui diferentes variações de conjuntos implementados em diferentes classes. A classe que usaremos é chamada *HashSet*.

Os dois tipos de funcionalidade que precisamos são a capacidade de inserir elementos no conjunto e recuperar os elementos posteriormente. Felizmente, essas tarefas quase não contêm novidades para nós.

Considere o seguinte fragmento de código:

```
import java.util.HashSet;
```

```
...
HashSet<String> mySet = new HashSet<>();
mySet.add("um");
mySet.add("dois");
mySet.add("três");
```

Compare esse código com as instruções necessárias para inserir elementos em um *ArrayList*. Quase não há diferença, exceto que criamos um *HashSet* dessa vez, em vez de um *ArrayList*.

Agora, vamos analisar a iteração sobre todos os elementos:

```
for(String item : mySet) {
  faça algo com esse item
}
```

Novamente, essas instruções são as mesmas que usamos para iterar sobre um *ArrayList* nas aulas anteriores.

Em resumo, o uso de coleções em Java é bastante semelhante para diferentes tipos de coleções. Depois de entender como usar um deles, você poderá usá-los todos. As diferenças realmente estão no comportamento de cada coleção. Uma lista, por exemplo, manterá todos os elementos inseridos na ordem desejada, fornece acesso aos elementos por índice e pode conter o mesmo elemento várias vezes. Um conjunto, por outro lado, não mantém nenhuma ordem específica (os elementos podem ser retornados em um *loop for-each* em uma ordem diferente daquela em que foram inseridos) e garante que cada elemento esteja no conjunto no máximo uma vez. Inserir um elemento uma segunda vez simplesmente não tem efeito.

6.8 Dividindo *strings*

Agora que vimos como usar um conjunto, podemos investigar como podemos cortar a sequência de entrada em palavras separadas para serem armazenadas em um conjunto de palavras. A solução é mostrada em uma nova versão do método *getInput* do *InputReader* (código 6.5).

Código 6.5

```
/**
 * Read a line of text from standard input (the text terminal),
 * and return it as a set of words.
 *
 * @return  A set of Strings, where each String is one of the
 *          words typed by the user
 */
public HashSet<String> getInput()
{
   System.out.print("> ");              // print prompt
   String inputLine = reader.nextLine().trim().toLowerCase();


   String[] wordArray = inputLine.split(" ");  // split at spaces


   // add words from array into hashset
   HashSet<String> words = new HashSet<>();
   for(String word : wordArray) {
      words.add(word);
   }
   return words;
}
```

Aqui, além de usar um *HashSet*, também usamos o método *split*, que é um método padrão da classe *String*.

O método *split* pode dividir uma *string* em *substrings* separadas e retornar aquelas em uma matriz de *strings*. O parâmetro para o método *split* define em que tipo de caracteres a *string* original deve ser dividida. Definimos que queremos cortar nossa *string* em cada caractere de espaço.

As próximas linhas de código criam um *HashSet* e copiam as palavras da matriz para o conjunto antes de retornar o conjunto (veja a próxima nota, para uma versão mais curta).

Nota: existe uma maneira mais curta e elegante de fazer isso. Poder-se-ia escrever palavras *HashSet* <> = novo *HashSet* <> (*Arrays*.*asList* (*wordArray*)); para substituir todas as quatro linhas de código. Isso usa a classe *Arrays* da biblioteca padrão e um método estático (também conhecido como método de classe) que ainda não queremos discutir. Se você estiver curioso, leia sobre os métodos de classe na Seção 6.15 e tente usar esta versão.

6.9 Finalizando o sistema *TechSupport*

Para juntar tudo, também precisamos ajustar as classes *SupportSystem* e “Responder” para lidar corretamente com um conjunto de palavras em vez de uma única sequência. O código 6.6 mostra a nova versão do método *start* da classe *SupportSystem*. Não mudou muito. As mudanças são:

A variável de entrada que recebe o resultado de *reader.getInput* () agora é do tipo *HashSet*.

A verificação para finalizar o aplicativo é feita usando o método *contains* da classe *HashSet*, ao invés de um método *String*. (Consulte esse método na documentação.) A classe *HashSet* deve ser importada usando uma instrução de importação (não mostrada aqui).

Por fim, precisamos estender o método *generateResponse* na classe "Respondente" para aceitar um conjunto de palavras como parâmetro. Em seguida, ele precisa iterar essas palavras e verificar cada uma delas com o nosso mapa de palavras conhecidas. Se alguma das palavras for reconhecida, retornamos imediatamente a resposta associada. Se não reconhecermos nenhuma das palavras, como antes, escolhemos uma de nossas respostas padrão. O código 6.7 mostra a solução.

Esta é a última alteração no aplicativo discutido aqui neste capítulo. A solução no projeto *tech-support-complete* contém todas essas alterações. Ele também contém mais associações de palavras a respostas do que as mostradas neste capítulo.

Muitas outras melhorias para esta aplicação são possíveis.

Código 6.6

```
public void start()
{
    boolean finished = false;

    printWelcome();

    while(!finished) {
        HashSet<String> input = reader.getInput();

        if(input.contains("bye")) {
            finished = true;
        }
        else {
            String response = responder.generateResponse(input);
            System.out.println(response);
        }
    }
    printGoodbye();
}
```

Código 6.7

```
public String generateResponse(HashSet<String> words)
{
    for (String word : words) {
        String response = responseMap.get(word);
        if(response != null) {
            return response;
        }
    }

    // If we get here, none of the words from the input line was recognized.
    // In this case we pick one of our default responses (what we say when
    // we cannot think of anything else to say...)
    return pickDefaultResponse();
}
```

Videoaula 1

Utilize o QR Code para assistir!

Videoaula 2

Utilize o QR Code para assistir!

Videoaula 3

Utilize o QR Code para assistir!

Aula 02 - Público versus Privado

6.10 Classes de *autobox* e *wrapper*

Vimos que, com uma parametrização adequada, as classes de coleção podem armazenar objetos de qualquer tipo de objeto. Resta um problema: o Java possui alguns tipos que não são do tipo objeto.

Como sabemos, os tipos simples, como *int*, booleano e char, são separados dos tipos de objetos. Seus valores não são instâncias de classes e normalmente não seria possível adicioná-los a uma coleção.

Há situações em que podemos criar uma lista de valores *int* ou um conjunto de valores de caracteres, por exemplo. O que podemos fazer? A solução de Java para esse problema são as classes *wrapper* (embrulhador). Todo tipo primitivo em Java tem uma classe de *wrapper* correspondente que representa o mesmo tipo, mas é um tipo de objeto real. A classe de *wrapper* para *int*, por exemplo, é chamada *Integer*.

A instrução a seguir, explicitamente, agrupa o valor da variável *int* primitiva *ix,* em um objeto Inteiro:

Integer iwrap = new Integer(ix);

E agora o *iwrap* obviamente poderia ser facilmente armazenado em uma coleção *ArrayList* <Integer>, por exemplo. No entanto, o armazenamento de valores primitivos em uma coleção de objetos é ainda mais fácil por meio de um recurso do compilador conhecido como "*autoboxing*".

Sempre que um valor de um tipo primitivo é usado em um contexto que requer um tipo de *wrapper*, o compilador agrupa automaticamente o valor do tipo primitivo em um objeto de *wrapper* apropriado.

Isso significa que valores do tipo primitivo podem ser adicionados diretamente a uma coleção:

```
private ArrayList<Integer> markList;
...
public void storeMarkInList(int mark)
{
  markList.add(mark);
}
```

A operação reversa de remoção da *unboxing* também é executada automaticamente; portanto, a recuperação de uma coleção pode ser assim:

int firstMark = markList.remove(0);

A *autobox* também é aplicada sempre que um valor do tipo primitivo é passado como parâmetro para um método que espera um tipo de invólucro e quando um valor do tipo primitivo é armazenado em uma variável do tipo invólucro. Da mesma forma, o desempacotamento é aplicado quando um valor do tipo *wrapper* é passado como parâmetro para um método que espera um valor do tipo primitivo e quando armazenado em uma variável do tipo primitivo. Vale a pena notar que isso quase faz parecer que tipos primitivos podem ser armazenados em

coleções. No entanto, o tipo da coleção ainda deve ser declarado usando o tipo de *wrapper* (por exemplo, *ArrayList* <*Integer*>, não *ArrayList* <*int*>).

6.10.1 Mantendo contagens de uso

A combinação de um mapa com a caixa automática fornece uma maneira fácil de manter as contagens de objetos. Por exemplo, suponha que a empresa que administra o sistema *TechSupport* esteja recebendo reclamações de seus clientes de que algumas de suas respostas não têm relação com a pergunta que está sendo feita. A empresa pode decidir analisar as palavras que estão sendo usadas nas perguntas e adicionar respostas específicas adicionais para aquelas que são usadas com mais frequência para as quais não tem respostas prontas. O código 6.8, mostra parte da classe *WordCounter* que pode ser encontrada no projeto de análise de suporte técnico.

Código 6.8

```
import java.util.HashMap;
import java.util.HashSet;

/**
* Keep a record of how many times each word was
* entered by users.
*
* @author     Michael Kölling and David J. Barnes
* @version    1.0 (2016.02.29)
*/
public class WordCounter
{
  // Associate each word with a count.
  private HashMap<String, Integer> counts;

  /**
   * Create a WordCounter
   */
  public WordCounter()
  {
     counts = new HashMap<>();
  }

  /**
   * Update the usage count of all words in input.
   * @param input A set of words entered by the user.
   */
  public void addWords(HashSet<String> input)
  {
     for(String word : input) {
        int counter = counts.getOrDefault(word, 0);
        counts.put(word, counter + 1);
     }
  }
}
```

O método *addWords* recebe o mesmo conjunto de palavras que são transmitidas ao "Respondente", para que cada palavra possa ser associada a uma contagem. As contagens são armazenadas em um mapa dos objetos *String* para Inteiro.

Observe o uso do método *getOrDefault* do *HashMap*, que usa dois parâmetros: uma chave e um valor padrão. Se a chave já estiver em uso no mapa, esse método retornará seu valor associado, mas se a chave não estiver em uso, retornará o valor padrão em vez de nulo. Isso evita a necessidade de escrever duas ações de acompanhamento diferentes, dependendo de a chave estar em uso ou não. Usando *get*, teríamos que escrever algo como:

```
Integer counter = counts.get(word);
if(counter == null) {
  counts.put(word, 1);
}
else {
  counts.put(word, counter + 1);
}
```

Observe como a caixa automática e a *unboxing* são usadas várias vezes nesses exemplos.

6.11 Escrevendo documentação da classe

Ao trabalhar em seus projetos, é importante escrever documentação para suas aulas à medida que você desenvolve o código fonte. É comum que os programadores não levem a documentação a sério o suficiente, e isso frequentemente cria sérios problemas posteriormente.

Se você não fornecer uma documentação suficiente, pode ser muito difícil para outro programador (ou você mesmo algum tempo depois) entender suas aulas. Normalmente, o que você precisa fazer nesse caso é ler a implementação da classe e descobrir o que ela faz. Isso pode funcionar com um projeto de aluno pequeno, mas cria problemas em projetos do mundo real.

Não é incomum que aplicativos comerciais consistam em centenas de milhares de linhas de código em vários milhares de classes. Imagine que você tinha que ler tudo isso para entender como um aplicativo funciona! Você nunca teria sucesso.

Quando usamos as classes da biblioteca Java, como *HashSet* ou *Random*, contamos exclusivamente com a documentação para descobrir como usá-las. Nunca vimos a implementação dessas classes. Isso funcionou porque essas classes estavam suficientemente bem documentadas (embora até essa documentação possa ser aprimorada). Nossa tarefa teria sido muito mais difícil se tivéssemos que ler a implementação das classes antes de usá-las.

Em uma equipe de desenvolvimento de software, a implementação de classes geralmente é compartilhada entre vários programadores. Embora você possa ser responsável por implementar a classe *SupportSystem* em nosso último exemplo, outra pessoa pode implementar o *InputReader*. Assim, você pode escrever uma classe enquanto faz chamadas para métodos de outras classes.

Os sistemas Java incluem uma ferramenta chamada "javadoc" que pode ser usada para gerar uma descrição dessa interface a partir dos arquivos de origem. A documentação da biblioteca

padrão que usamos, por exemplo, foi criada a partir dos arquivos de origem das classes pelo “javadoc”.

6.11.1 Usando “javadoc” no BlueJ

O ambiente BlueJ usa “javadoc” para permitir que você crie documentação para sua classe de duas maneiras:

Você pode visualizar a documentação de uma única classe alternando o seletor *pop-up,* no canto superior direito da janela do editor de código fonte para documentação (ou alternar a exibição de documentação no menu Ferramentas do editor).

Você pode usar a função Documentação do Projeto, no menu Ferramentas da janela principal, para gerar documentação para todas as classes no projeto.

O tutorial do BlueJ fornece mais detalhes se você estiver interessado. Você pode encontrar o tutorial do BlueJ no menu Ajuda do BlueJ.

6.11.2 Elementos da documentação da classe

A documentação de uma classe deve incluir pelo menos:

- O nome da classe;
- Um comentário descrevendo o objetivo geral e as características da classe;
- A versão *number* 6.11.

Escrevendo documentação da classe

- O nome do autor (ou nomes dos autores);
- Documentação para cada construtor e cada método.

A documentação para cada construtor e método deve incluir:

- O nome do método;
- O tipo de retorno;
- Os nomes e tipos de parâmetros;
- Uma descrição da finalidade e função do método;
- Uma descrição de cada parâmetro;
- Uma descrição do valor retornado.

Além disso, cada projeto completo deve ter um comentário geral do projeto, geralmente contido em um arquivo "Leia-me". No BlueJ, este comentário do projeto é acessível através da nota de texto exibida no canto superior esquerdo do diagrama de classes.

Alguns elementos da documentação, como nomes e parâmetros de métodos, sempre podem ser extraídos do código fonte. Outras partes, como comentários descrevendo a classe, métodos

e parâmetros, precisam de mais atenção, pois podem ser facilmente esquecidas, incompletas ou incorretas.

Em Java, os comentários “javadoc” são gravados com um símbolo de comentário especial no início:

/ **

Este é um comentário do “javadoc”.

* /

O símbolo de início do comentário deve ter dois asteriscos para ser reconhecido como um comentário “javadoc”.

Esse comentário imediatamente anterior à declaração da classe é lido como um comentário da classe. E se o comentário está diretamente acima de uma assinatura de método, é considerado um comentário de método.

Os detalhes exatos de como a documentação é produzida e formatada são diferentes em diferentes linguagens de programação e ambientes. O conteúdo, no entanto, deve ser mais ou menos o mesmo.

Em Java, usando “javadoc”, vários símbolos especiais de chave estão disponíveis para formatar a documentação. Esses símbolos principais começam com o símbolo @ e incluem:

@versão

@autor

@param

@Retorna

6.12 Público versus privado

É hora de discutir com mais detalhes um aspecto das classes que já encontramos várias vezes, mas sem falar muito sobre isso: modificadores de acesso.

Modificadores de acesso são as palavras-chave *public* ou *private* no início das declarações de campo e assinaturas de método. Por exemplo:

```
// declaração de campo
private int numberOfSeats;
// métodos
public void setAge (int replaceAge)
{...
}
private int computeAverage ()
{...
}
```

Campos, construtores e métodos podem todos ser públicos ou privados, embora até agora vimos principalmente campos privados e construtores e métodos públicos. Voltaremos a isso abaixo.

Modificadores de acesso definem a visibilidade de um campo, construtor ou método. Se um método, por exemplo, for público, poderá ser chamado de dentro da mesma classe ou de qualquer outra classe.

Métodos particulares, por outro lado, podem ser invocados somente de dentro da classe em que são declarados. Eles não são visíveis para outras classes.

Agora que discutimos a diferença entre a interface e a implementação de uma classe (Seção 6.3.1), podemos entender mais facilmente o objetivo dessas palavras-chave.

Lembre-se: a interface de uma classe é o conjunto de detalhes que outro programador que usa a classe precisa ver. Ele fornece informações sobre como usar a classe. A interface inclui assinaturas e comentários do construtor e do método. Também é chamada de parte pública de uma classe. Seu objetivo é definir o que a classe faz.

A implementação é a seção de uma classe que define com precisão como a classe funciona.

Os corpos do método, contendo as instruções Java, e a maioria dos campos fazem parte da implementação. A implementação também é chamada de parte privada de uma classe.

O usuário de uma classe não precisa saber sobre a implementação. De fato, existem boas razões pelas quais um usuário deve ser impedido de saber sobre a implementação (ou pelo menos de fazer uso desse conhecimento). Este princípio é chamado de ocultação de informações.

A palavra-chave pública declara que um elemento de uma classe (um campo ou método) faz parte da interface (ou seja, visível publicamente); a palavra-chave privada declara que faz parte da implementação (ou seja, oculta do acesso externo).

6.12.1 Ocultar informações

Em muitas linguagens de programação orientadas a objetos, os elementos internos de uma classe e sua implementação estão ocultos de outras classes. Existem dois aspectos para isso. Primeiro, um programador que faz uso de uma classe não precisa conhecer os elementos internos; segundo, um usuário não deve ter permissão para conhecer os internos.

O primeiro princípio que não precisamos saber tem a ver com abstração e modularização, conforme discutido anteriormente. Se fosse necessário conhecer todos os elementos internos de todas as classes que precisamos usar, nunca concluiríamos a implementação de grandes sistemas.

O segundo princípio que não é permitido conhecer é diferente. Também tem a ver com modularização, mas em um contexto diferente. A linguagem de programação não permite acesso à seção privada de uma classe por instruções em outra classe. Isso garante que uma classe não dependa exatamente de como outra classe é implementada.

Isso é muito importante para o trabalho de manutenção. É uma tarefa muito comum para um programador de manutenção alterar posteriormente ou estender a implementação de uma classe para fazer melhorias ou corrigir bugs. Idealmente, alterar a implementação de uma classe

não deve tornar necessário alterar outras classes também. Esse problema é conhecido como acoplamento. Se as alterações em uma parte do programa não exigirem também alterações em outra parte do programa, isso é conhecido como acoplamento fraco ou acoplamento frouxo. O acoplamento frouxo é bom, porque facilita muito o trabalho de um programador de manutenção. Em vez de entender e mudar muitas classes, ela pode precisar entender e mudar apenas uma classe. Por exemplo, se um programador de sistemas Java fizer uma melhoria na implementação da classe *ArrayList*, você esperaria não precisar alterar seu código usando esta classe. Isso funcionará, porque você não fez nenhuma referência à implementação do *ArrayList* em seu próprio código.

Portanto, para ser mais preciso, a regra de que um usuário não deve ter permissão para conhecer os elementos internos de uma classe não se refere ao programador de outra classe, mas à própria classe. Geralmente, não é um problema se um programador conhece os detalhes da implementação, mas uma classe não deve "conhecer" (depender) dos detalhes internos de outra classe. O programador de ambas as classes pode até ser a mesma pessoa, mas as classes ainda devem ser fracamente acopladas.

As questões de acoplamento e ocultação de informações são muito importantes, e teremos mais a dizer sobre elas nas próximas aulas.

Por enquanto, é importante entender que a palavra-chave privada impõe a ocultação de informações, não permitindo que outras classes acessem essa parte da classe. Isso garante um acoplamento flexível e torna a aplicação mais modular e fácil de manter.

6.12.2 Métodos particulares e campos públicos

A maioria dos métodos que vimos até agora era pública. Isso garante que outras classes possam chamar esses métodos. Às vezes, porém, usamos métodos privados. Na classe *SupportSystem* do sistema *TechSupport*, por exemplo, vimos os métodos *printWelcome* e *printGoodbye* declarados como métodos privados.

A razão para ter as duas opções é que os métodos são realmente usados para diferentes propósitos.

Eles são usados para fornecer operações aos usuários de uma classe (métodos públicos) e são usados para dividir uma tarefa maior em várias menores para facilitar o manuseio da tarefa grande. No segundo caso, as subtarefas não devem ser invocadas diretamente de fora da classe, mas são colocadas em métodos separados apenas para facilitar a leitura da implementação de uma classe. Nesse caso, esses métodos devem ser privados. Os métodos *printWelcome* e *printGoodbye* são exemplos disso.

Outro bom motivo para ter um método privado é para uma tarefa necessária (como uma subtarefa) em vários métodos de uma classe. Em vez de escrever o código várias vezes, podemos escrevê-lo uma vez em um único método privado e depois chamá-lo de vários lugares diferentes. Veremos um exemplo disso mais tarde.

Em Java, os campos também podem ser declarados privados ou públicos. Até o momento, não vimos exemplos de campos públicos e há uma boa razão para isso. Declarar campos públicos quebra o princípio de ocultação de informações. Torna uma classe dependente dessas informações vulnerável a operação incorreta se a implementação for alterada. Mesmo que a

linguagem Java nos permita declarar campos públicos, consideramos esse estilo ruim e não faremos uso dessa opção.

Algumas outras linguagens orientadas a objetos não permitem campos públicos.

Uma outra razão para manter os campos privados é que ele permite que um objeto mantenha maior controle sobre seu estado. Se o acesso a um campo privado for canalizado por meio de métodos acessadores e mutadores, um objeto poderá garantir que o campo nunca seja configurado com um valor que seria inconsistente com seu estado geral. Este nível de integridade não é possível se os campos forem tornados públicos.

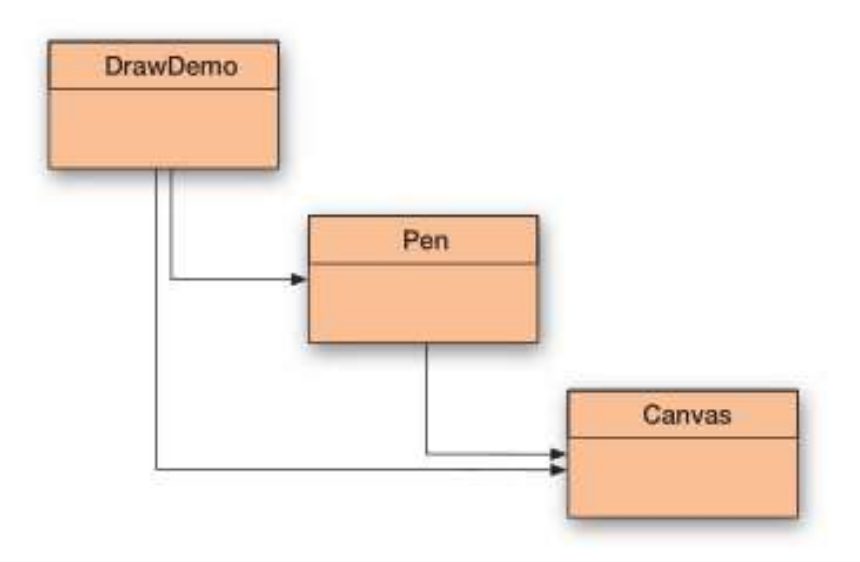
Em suma, os campos devem sempre ser privados.

Java tem mais dois níveis de acesso. Um é declarado usando a palavra-chave protegida como modificador de acesso; o outro é usado se nenhum modificador de acesso for declarado. Discutiremos isso nos próximos capítulos.

6.13 Aprendendo sobre classes a partir de suas interfaces

Discutiremos brevemente outro projeto para revisitar e praticar os conceitos discutidos nesta aula. O projeto tem o nome rabisco, do inglês *scribble*, e você pode encontrá-lo na pasta do Capítulo 6 dos projetos, que pode ser encontrado nas referências.

Figura 1 – Projeto rabisco



Fonte: BlueJ.

6.13.1 A demonstração do projeto rabisco (*scribble*)

O projeto de rabisco fornece três classes: *DrawDemo, Pen e Canvas* (Figura 1).

A classe *Canvas* fornece uma janela na tela que pode ser usada para desenhar. Possui operações para desenhar linhas, formas e texto. Uma tela pode ser usada criando uma instância

interativamente ou de outro objeto. A classe *Canvas* não deve precisar de nenhuma modificação. Provavelmente, é melhor tratá-lo como uma classe de biblioteca: abra o editor e mude para a exibição da documentação. Isso exibe a interface da classe com a documentação do "javadoc".

A classe Pen fornece um objeto de caneta que pode ser usado para produzir desenhos na tela, movendo a caneta pela tela. A caneta em si é invisível, mas desenha uma linha quando movida na tela.

A classe *DrawDemo* fornece alguns pequenos exemplos de como usar um objeto de caneta para produzir um desenho na tela. O melhor ponto de partida para entender e experimentar este projeto é a classe *DrawDemo*.

Alguns dos métodos nas classes Caneta e Tela referem-se a parâmetros do tipo Cor.

Esse tipo é definido na classe *Color* no pacote *java.awt* (portanto, seu nome completo é *java.awt.Color*). A classe *Color* define algumas constantes de cores, às quais podemos nos referir da seguinte maneira:

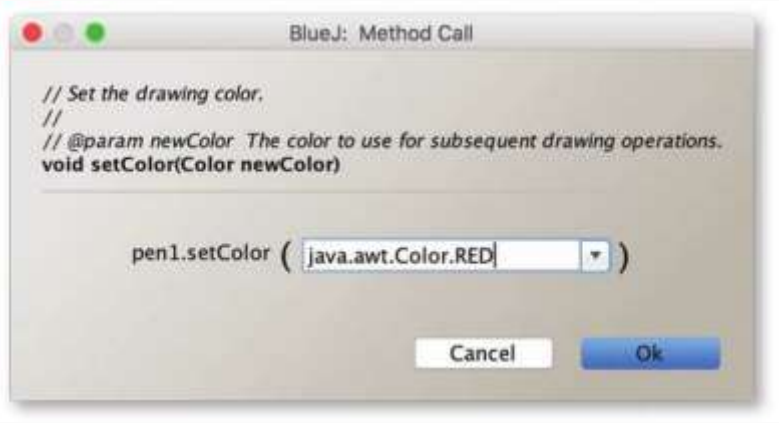
Color.RED

O uso dessas constantes requer que a classe *Color* seja importada na classe *using*.

Ao chamar métodos interativamente que esperam parâmetros da classe Color, precisamos nos referir à classe de maneira um pouco diferente. Como o diálogo interativo não possui declaração de importação (e, portanto, a classe Color não é conhecida automaticamente), precisamos escrever o nome completo da classe para se referir à classe (Figura 2). Isso permite que o sistema Java encontre a classe sem usar uma instrução de importação.

Agora que sabemos como mudar a cor de canetas e telas.

Figura 2 – Mudar a cor de telas e canetas



Fonte: BlueJ.

Como você viu, podemos desenhar diretamente na tela ou usar um objeto de caneta para desenhar. A caneta nos fornece uma abstração que mantém uma posição, rotação e cor atuais,

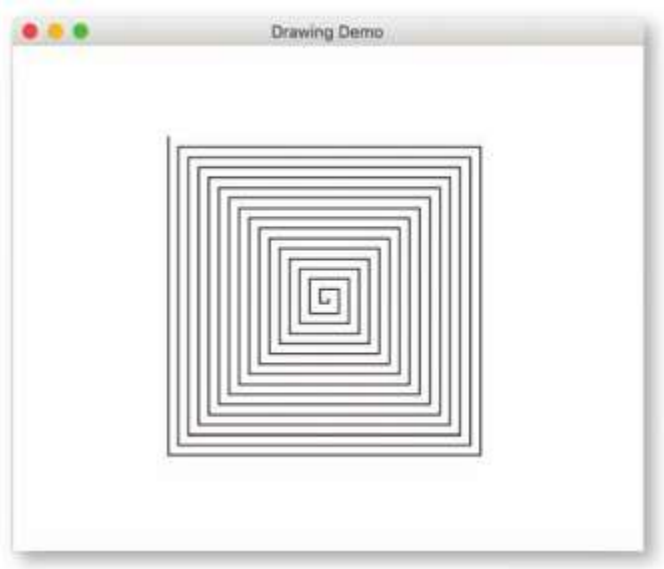
e isso facilita a produção de alguns tipos de desenhos. Vamos experimentar isso um pouco mais, desta vez escrevendo código em uma classe em vez de usar chamadas interativas.

6.13.2 Conclusão do código

Frequentemente, estamos razoavelmente familiarizados com uma classe de biblioteca que estamos usando, mas ainda não conseguimos lembrar os nomes exatos de todos os métodos ou parâmetros exatos. Para essa situação, os ambientes de desenvolvimento geralmente oferecem alguma ajuda: conclusão do código.

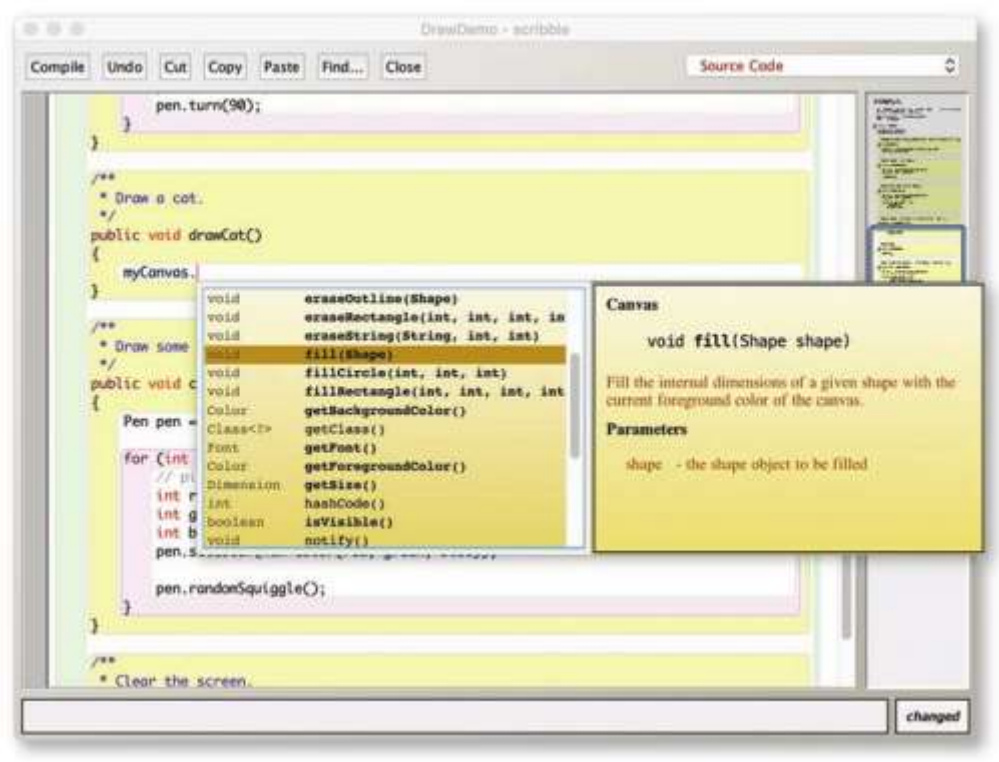
A conclusão de código é uma função que está disponível no editor do BlueJ, quando o cursor está atrás do ponto de uma chamada de método. Nessa situação, digitar CTRL-*space* exibirá uma *pop-up* listando todos os métodos na interface do objeto que estamos usando na chamada (Figura 4).

Figura 3 - *Pop-up* listando todos os métodos na interface do objeto que estamos usando na chamada



Fonte: BlueJ.

Figura 4 - *Pop-up* listando todos os métodos na interface do objeto



Fonte: BlueJ.

Quando o *pop-up* de conclusão do código é exibido, podemos digitar o início do nome do método para restringir a lista de métodos. Pressionar *Return* insere a chamada do método selecionado em nosso código fonte. A conclusão de código também pode ser usada sem um objeto anterior para chamar métodos locais.

O uso do preenchimento de código não deve substituir a leitura da documentação de uma classe, porque ela não inclui todas as informações (como o comentário introdutório da classe).

Porém, uma vez que estamos razoavelmente familiarizados com uma classe em geral, o preenchimento de código é uma grande ajuda para recuperar mais facilmente os detalhes de um método e inserir a chamada em nossa fonte.

6.14 Variáveis e constantes de classe

Até agora, não analisamos a classe *BouncingBall*. Se você estiver realmente interessado em entender como essa animação funciona, também poderá estudar esta classe. É razoavelmente simples. O único método que requer algum esforço para entender é o movimento, onde a bola muda de posição para a próxima posição em seu caminho.

Deixaremos em grande parte ao leitor estudar esse método, exceto por um detalhe que queremos discutir aqui.

private static final int GRAVITY = 3;

Essa é uma construção que ainda não vimos. Essa linha, de fato, introduz duas novas palavras-chave, que são usadas juntas: estática e final.

6.14.1 A palavra-chave estática

A palavra-chave *static* é a sintaxe do Java para definir variáveis de classe. Variáveis de classe são campos armazenados na própria classe, não em um objeto. Isso as torna fundamentalmente diferentes das variáveis de instância (os campos com os quais lidamos até agora). Considere este segmento de código (uma parte da classe *BouncingBall*):

```
public class BouncingBall
{
  // Efeito da gravidade.
  private static final int GRAVITY = 3;
  private int xPosition;
  private int yPosition;
  ...Outros campos e métodos omitidos.
}
```

Agora imagine que criamos três instâncias de *BouncingBall*. A situação resultante é mostrada na Figura 5.

Como podemos ver no diagrama, as variáveis de instância (*xPosition* e *yPosition*) são armazenadas em cada objeto. Como criamos três objetos, temos três cópias independentes dessas variáveis.

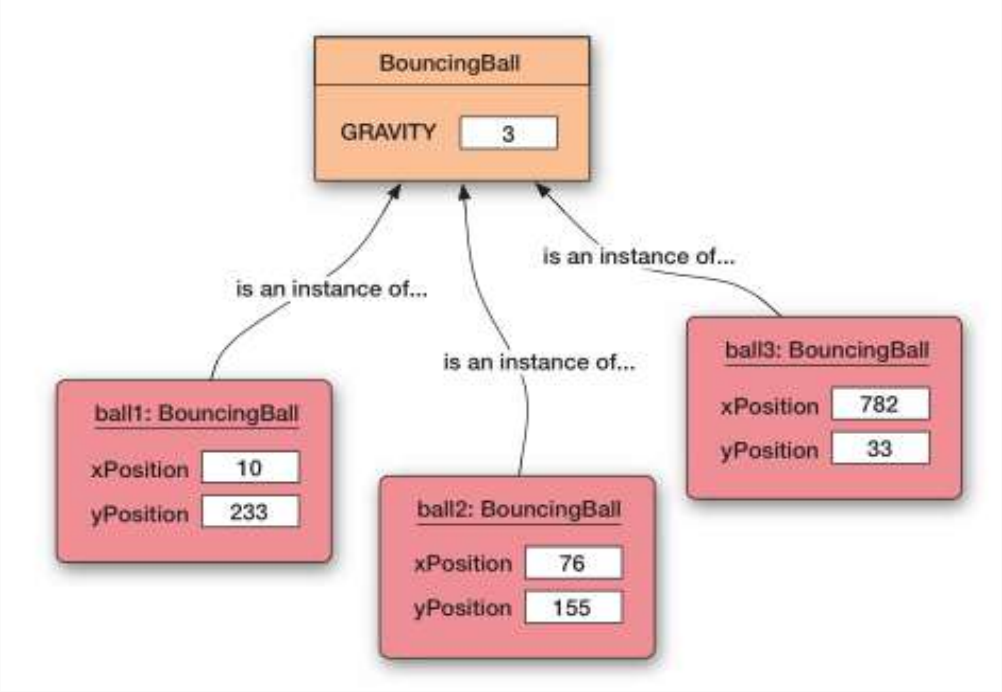
A variável de classe *GRAVITY*, por outro lado, é armazenada na própria classe. Como resultado, sempre há exatamente uma cópia dessa variável independentemente do número de instâncias criadas.

O código fonte da classe pode acessar (ler e definir) esse tipo de variável, assim como uma variável de instância. A variável de classe pode ser acessada a partir de qualquer uma das instâncias da classe. Como resultado, os objetos compartilham essa variável.

Variáveis de classe são frequentemente usadas se um valor sempre deve ser o mesmo para todas as instâncias de uma classe. Em vez de armazenar uma cópia do mesmo valor em cada objeto, o que seria um desperdício de espaço e pode ser difícil de coordenar, um único valor pode ser compartilhado entre todas as instâncias.

Java também suporta métodos de classe (também conhecidos como métodos estáticos), que são métodos que pertencem a uma classe.

Figura 5 - Três instâncias de *BouncingBall*



Fonte: BlueJ.

6.14.2 Constantes

Um uso frequente da palavra-chave estática é definir constantes. As constantes são semelhantes às variáveis, mas não podem alterar seu valor durante a execução de um aplicativo. Em Java, constantes são definidas com a palavra-chave final. Por exemplo:

```
private final int SIZE = 10;
```

Aqui, definimos uma constante denominada *SIZE* com o valor 10. Observamos que as declarações constantes são semelhantes às declarações de campo, com duas diferenças:

- Eles incluem a palavra-chave final antes do nome do tipo;
- Eles devem ser inicializados com um valor no ponto da declaração.

Se um valor pretende não mudar, é uma boa ideia declará-lo final. Isso garante que não possa ser acidentalmente alterado posteriormente. Qualquer tentativa de alterar um campo constante resultará em uma mensagem de erro em tempo de compilação. As constantes são, por convenção, frequentemente escritas em letras maiúsculas.

Seguiremos essa convenção neste material.

Na prática, é frequente o caso de constantes que se aplicam a todas as instâncias de uma classe. Nesta situação, declaramos constantes de classe. Constantes de classe são campos de classe constantes. Eles são declarados usando uma combinação das palavras-chave estáticas e finais. Por exemplo:

```
private static final int SIZE = 10;
```

A definição de *GRAVITY* do nosso projeto de bola quicando é outro exemplo dessa constante. Este é o estilo no qual constantes são definidas na maioria das vezes. As constantes específicas da instância são usadas com muito menos frequência.

Encontramos mais dois exemplos de constantes no projeto de rabisco. O primeiro exemplo foram duas constantes usadas na classe *Pen* para definir o tamanho do "rabisco aleatório" (volte para o projeto e encontre-as). O segundo exemplo foi o uso das constantes de cores nesse projeto, como Color. VERMELHO. Nesse caso, não definimos as constantes, mas usamos constantes definidas em outra classe.

A razão pela qual podemos usar as constantes da classe Color é que elas foram declaradas públicas.

Ao contrário de outros campos (sobre os quais comentamos anteriormente que eles nunca deveriam ser declarados públicos), declarar constantes públicas é geralmente sem problemas e às vezes, útil.

Uma variável privada usada para indicar uma marca de aprovação, com o valor inteiro 40.

Uma variável de caráter público usada para indicar que o comando help é 'h'.

6.15 Métodos de classe

Até agora, todos os métodos que vimos foram métodos de instância: eles são chamados em uma instância de uma classe. O que distingue métodos de classe de métodos de instância é que métodos de classe podem ser chamados sem que uma instância possua a classe é suficiente.

6.15.1 Métodos estáticos

Na Seção 6.14, discutimos variáveis de classe. Os métodos de classe são conceitualmente relacionados e usam uma sintaxe relacionada (a palavra-chave estática em Java). Assim como as variáveis de classe pertencem à classe e não a uma instância, o mesmo acontece com os métodos de classe.

Um método de classe é definido adicionando a palavra-chave estática na frente do nome do tipo no cabeçalho do método:

```
public static int getNumberOfDaysThisMonth()
{
  ...
}
```

Esse método pode ser chamado especificando o nome da classe na qual está definido, antes do ponto na notação de ponto usual. Se, por exemplo, o método acima for definido em uma classe chamada Calendário, a chamada a seguir será chamada:

```
int dias = Calendar.getNumberOfDaysThisMonth();
```

Observe que o nome da classe é usado antes do ponto que nenhum objeto foi criado.

6.15.2 Limitações dos métodos de classe

Como os métodos de classe estão associados a uma classe e não a uma instância, eles têm duas limitações importantes. A primeira limitação é que um método de classe pode não acessar nenhum campo de instância definido na classe. Isso é lógico, porque os campos da instância estão associados a objetos individuais. Em vez disso, os métodos de classe são restritos ao acesso a variáveis de classe de sua classe. A segunda limitação é como a primeira: um método de classe não pode chamar um método de instância da classe. Um método de classe pode chamar apenas outros métodos de classe definidos em sua classe.

Você verá que escrevemos muito poucos métodos de classe nos exemplos deste material.

Executando sem BlueJ

Quando terminarmos de escrever um programa, talvez desejemos repassá-lo para outra pessoa usar. Para fazer isso, seria bom que as pessoas pudessem usá-lo sem a necessidade de iniciar o BlueJ. Para que isso aconteça, precisamos de mais uma coisa: um método de classe específico conhecido como método principal.

6.16.1 O método principal

Se queremos iniciar um aplicativo Java sem o BlueJ, precisamos usar um método de classe. No BlueJ, normalmente criamos um objeto e chamamos um de seus métodos, mas sem o BlueJ, um aplicativo é iniciado sem nenhum objeto existente. As classes são as únicas coisas que temos inicialmente; portanto, o primeiro método que pode ser chamado deve ser um método de classe.

A definição de Java para iniciar aplicativos é bastante simples: o usuário especifica a classe que deve ser iniciada e o sistema Java chamará um método chamado *main* nessa classe.

Este método deve ter uma assinatura específica:

public static void main(String[] args)

Os métodos de classe podem ser chamados no menu *pop-up* da classe.

6.17 Mais material avançado

No capítulo anterior, introduzimos o material que descrevemos como "avançado" e sugerimos que esse material pudesse ser ignorado na primeira leitura, se desejado. Nesta seção, apresentamos outros materiais avançados e os mesmos conselhos se aplicam.

6.17.1 Tipos de coleção polimórfica (avançado)

Até agora, introduzimos várias classes de coleção diferentes do pacote *java.util*, como *ArrayList*, *HashMap* e *HashSet*. Cada um possui vários recursos distintos que o tornam apropriado para situações específicas. No entanto, eles também têm um número considerável de semelhanças

e métodos em comum. Isso sugere que eles podem estar relacionados de alguma forma. Na programação orientada a objetos, os relacionamentos entre classes são expressos por herança.

O mesmo argumento discutido para as classes da biblioteca é verdadeiro para as classes que você escreve: se podemos usar as classes sem precisar ler e entender a implementação completa, nossa tarefa se torna mais fácil. Como nas classes da biblioteca, queremos ver apenas a interface pública da classe, em vez da implementação. Portanto, é importante escrever uma boa documentação de classe para nossas próprias classes.

Ao tentar entender como as várias classes de coleção são usadas e os relacionamentos entre elas, é necessário prestar muita atenção em seus nomes. Pelo nome, é tentador supor que um *HashSet* deve ser muito semelhante a um *HashMap*. De fato, como ilustramos, um *HashSet* é realmente muito mais próximo de um *ArrayList*.

Seus nomes consistem em duas partes, por exemplo, "matriz" e "lista". A segunda metade nos diz com que tipo de coleção estamos lidando (Lista, Mapa, Conjunto) e a primeira nos diz como é implementada (por exemplo, usando uma matriz ou *hash*).

Para usar coleções, o tipo da coleção (a segunda parte) é o mais importante. Já discutimos antes que podemos abstrair da implementação; ou seja, não precisamos pensar nisso com muitos detalhes. Assim, para nossos propósitos, um *HashSet* e um *TreeSet* são muito semelhantes. Ambos são conjuntos, então eles se comportam da mesma maneira. A principal diferença está na implementação, que é importante apenas quando começamos a pensar em eficiência: uma implementação executará algumas operações muito mais rapidamente que a outra. No entanto, as preocupações com eficiência surgem muito mais tarde, e somente quando temos coleções ou aplicativos muito grandes nos quais o desempenho é crítico.

Uma consequência dessa semelhança é que geralmente podemos ignorar detalhes da implementação ao declarar uma variável que se refere a um objeto de coleção específico. Em outras palavras, podemos declarar que a variável é de um tipo mais abstrato, como *List*, *Map* ou *Set*, em vez do tipo mais concreto, *ArrayList*, *LinkedList*, *HashMap*, *TreeSet*, etc.

```
List<Track> tracks = new LinkedList<>();
```

ou

```
Map<String, String> responseMap = new HashMap<>();
```

Aqui, a variável é declarada como do tipo *List*, mesmo que o objeto criado e armazenado nela seja do tipo *LinkedList* (e o mesmo para *Map* e *HashMap*).

Variáveis como *tracks* e *responseMap* nesses exemplos são chamadas de variáveis polimórficas, porque são capazes de se referir a objetos de tipos diferentes, mas relacionados. As faixas variáveis podem se referir a um *ArrayList* ou a *LinkedList*, enquanto *responseMap* pode se referir a um *HashMap* ou a um *TreeMap*. O ponto principal por trás de declará-las assim, é que a maneira como as variáveis são usadas no restante do código é independente do tipo exato e concreto do objeto ao qual elas se referem. Uma chamada para adicionar, limpar, obter, remover ou dimensionar faixas, tem a mesma sintaxe e o mesmo efeito na coleção associada. O tipo exato raramente importa em qualquer situação. Uma vantagem de usar variáveis polimórficas dessa maneira é que, se quisermos mudar de um tipo concreto para outro, o único local no código que requer uma alteração é o local em que o objeto é criado. Outra vantagem é que o código que escrevemos não precisa saber o tipo concreto que pode ter sido usado no

código escrito por outra pessoa - precisamos apenas conhecer o tipo abstrato: Lista, Mapa, Conjunto, etc. Ambas as vantagens reduzem o acoplamento entre diferentes partes de um programa.

6.17.2 O método de coleta de fluxos (avançado)

Nesta seção, continuamos a discussão sobre fluxos e lambdas do Java 8 que iniciamos no capítulo anterior (Avançado). Explicamos lá que é importante saber que cada operação do fluxo deixa seu fluxo de entrada inalterado. Cada operação cria um novo fluxo, com base em sua entrada e na operação, para transmitir a próxima operação na sequência. No entanto, geralmente queremos usar operações de fluxo para criar uma versão nova de uma coleção original que seja uma cópia modificada, geralmente filtrando o conteúdo original para selecionar os objetos desejados ou rejeitar aqueles que não são necessários. Para isso, precisamos usar o método de coleta de um fluxo. O método de coleta é uma operação terminal e, portanto, aparecerá como a operação final em um pipeline de operações. Ele pode ser usado para inserir os elementos resultantes de uma operação de fluxo novamente em uma nova coleção.

Na verdade, existem duas versões do método *collect*, mas vamos nos concentrar apenas no mais simples. De fato, optaremos por focar apenas em seu papel, permitindo-nos criar um novo objeto de coleta como a operação final de um pipeline e deixar uma exploração adicional para o leitor interessado. Até o uso mais básico envolve vários conceitos já avançados para este estágio relativamente inicial deste material: fluxos, métodos estáticos e polimorfismo.

O método *collect* usa um *Collector* como parâmetro. Essencialmente, isso deve ser algo que possa acumular progressivamente os elementos de seu fluxo de entrada de alguma maneira. Há semelhanças nessa ideia de acumulação com a operação de redução de fluxo que abordamos na aula anterior. Lá, vimos que um fluxo de números poderia ser reduzido a um único número; aqui vamos reduzir um fluxo de objetos para um único objeto, que é outra coleção. Como muitas vezes não nos preocupamos particularmente se a coleção é um *ArrayList*, um *LinkedList* ou alguma outra forma de tipo de lista que não conhecemos antes, podemos solicitar ao coletor que retorne à coleção na forma do tipo polimórfico *List*.

A maneira mais fácil de obter um coletor é usar um dos métodos estáticos definidos na classe *Collectors* do pacote *java.util.stream*: *toList, toMap e toSet*.

O código 6.9 mostra um método do projeto do *national-park* (parque nacional) que examinamos na aula 01. Esse método filtra a lista completa de objetos de mira para criar uma nova lista apenas para um animal específico. Em nossa versão no projeto *animal-Monitoring-v1*, esse método foi escrito no estilo iterativo, usando um *loop for-each*. Aqui, ele é reescrito no estilo funcional, usando o método de coleta.

Código 6.9

```
/**
 * Return a list of all sightings of the given type of animal
 * in a particular area.
 * @param animal The type of animal.
 * @return A list of sightings.
 */
public List<Sighting> getSightingsOf(String animal)
{
    return sightings.stream()
                .filter(record -> animal.equals(record.getAnimal()))
                .collect(Collectors.toList());
}
```

Um ponto sutil a ser observado é que o método *toList* dos coletores não retorna realmente um objeto *List* a ser passado para o método *collect*. Em vez disso, ele retorna um coletor que leva à criação de um objeto *List* concreto adequado.

6.17.3 Referências de método (avançado)

Suponha que desejemos coletar os objetos de um fluxo filtrado em um tipo concreto específico de coleção, como um *ArrayList*, em vez de um tipo polimórfico. Isso requer uma variação da abordagem usada na seção anterior. Ainda precisamos passar um *Collector* para a operação de coleta, mas precisamos obtê-lo de uma maneira diferente. O código 6.10 mostra como isso pode ser feito.

Código 6.10

```
/**
 * Return a list of all sightings of the given type of animal
 * in a particular area.
 * @param animal The type of animal.
 * @return A list of sightings.
 */
public List<Sighting> getSightingsOf(String animal)
{
    return sightings.stream()
                .filter(record -> animal.equals(record.getAnimal()))
                .collect(Collectors.toCollection(ArrayList::new));
}
```

Desta vez, chamamos o método estático *toCollection* de *Collectors*, que requer um único parâmetro, cujo tipo é *Supplier*. O parâmetro que usamos introduz uma nova notação, chamada referência de método, cuja sintaxe distinta é um par de caracteres de dois pontos adjacentes.

As referências de método são uma abreviação conveniente para alguns lambdas. Eles são frequentemente usados onde o compilador pode inferir facilmente a chamada completa de um método, sem precisar especificar seus detalhes explicitamente. Existem quatro casos de referências de método:

a) Uma referência a um construtor, como usamos aqui:

ArrayList::new

(Observe que, estritamente falando, um construtor não é realmente um método.) Essa é uma abreviação para o lambda:

() -> new ArrayList<>()

b) Uma referência a uma chamada de método em um objeto específico:

System.out::println

Isso seria uma abreviação para o lambda:

str -> System.out.println(str)

Observe que *System.out* é uma variável estática pública da classe *System* e o parâmetro *str* seria fornecido pelo contexto em que a referência do método é usada, por exemplo, como a operação do terminal de um pipeline de fluxo:

forEach(System.out::println)

c) Uma referência a um método estático:

Math::abs

Isso seria uma abreviação para:

x -> Math.abs(x)

O parâmetro x seria fornecido pelo contexto em que a referência do método é usada, por exemplo, como parte do mapeamento de um fluxo:

map(Math::abs)

d) Uma referência a um método de instância de um tipo específico:

String::length

Isso seria uma abreviação para:

str -> str.length()

Mais uma vez, *str* seria fornecido pelo contexto. Observe que é importante distinguir entre esse uso “a”, que envolve um método de instância, e o uso “b”, que envolve uma instância específica, e o uso “c”, que envolve um método estático.

Aqui ilustramos apenas os exemplos mais simples para o uso de referências de métodos. Casos mais complexos envolvendo vários parâmetros também serão encontrados nos exemplos mais gerais.

TERMOS INTRODUZIDOS NESTE CAPÍTULO

Interface, implementação, mapa, conjunto, caixa automática, classes de wrapper, javadoc, modificador de acesso, ocultação de informações, variável de classe, método de classe, método de classe, método principal, variável estática, constante, final, polimórfica e referência de método.

Videoaula 1

Utilize o QR Code para assistir!

Videoaula 2

Utilize o QR Code para assistir!

Videoaula 3

Utilize o QR Code para assistir!

Encerramento

Chegamos ao final de nossa unidade, parabéns!

Lidar com bibliotecas de classes e interfaces de classe é essencial para um programador competente.

Existem dois aspectos nesse tópico: ler descrições da biblioteca de classes (especialmente interfaces de classe) e escrevê-las.

É importante conhecer algumas classes essenciais da biblioteca de classes Java padrão e poder descobrir mais quando necessário. Aqui, apresentamos algumas das classes mais importantes e discutimos como navegar na documentação da biblioteca.

Também é importante ser capaz de documentar qualquer classe que seja escrita, no mesmo estilo das classes da biblioteca, para que outros programadores possam usá-la facilmente sem a necessidade de entender a implementação. Esta documentação deve incluir bons comentários para cada projeto, classe e método. O uso do “javadoc” com programas Java o ajudará a fazer isso.

Bom trabalho!

Videoaula Encerramento

Utilize o QR Code para assistir!

Assista agora ao vídeo de encerramento de nossa disciplina.

Esperamos que este guia o tenha ajudado compreender a organização e o funcionamento de seu curso. Outras questões importantes relacionadas ao curso serão disponibilizadas pela coordenação.

Grande abraço e sucesso!