Linguagem de Programação Orientada a Objetos

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

UniFil

Unidade 3

Introdução da Unidade

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

Objetivos

- Aprender sobre agrupamentos e coleções;
- Apresentar as classes genéricas;
- Introduzir sobre as funcionalidades *Streams* e *Lambdas*;
- Abordar sobre as funções *map*, *filter* e *reduce*.

Conteúdo programático

Aula 01 – Agrupando objetos

Aula 02 – Conceitos avançados

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 - Agrupando Objetos

O foco principal deste capítulo é apresentar algumas das maneiras pelas quais os objetos podem ser agrupados em coleções. Em particular, discutimos a classe *ArrayList* como um exemplo de coleções de tamanho flexível. Intimamente associada às coleções está a necessidade de iterar sobre os elementos que elas contêm. Para esse propósito, apresentamos duas novas estruturas de controle: o *loop for-each* e o *while*.

Este capítulo é longo e importante. Você pode não conseguir se tornar um bom programador sem entender completamente o conteúdo deste capítulo. Você provavelmente precisará de mais tempo para estudá-lo do que nos capítulos anteriores. Não fique tentado a se apressar; não se apresse e estude-o completamente.

Criando temas

Além de introduzir o novo material sobre coleções e iteração, também revisaremos dois dos principais temas introduzidos na aula 02, da unidade 2: abstração e interação com objetos. Vimos que a abstração nos permite simplificar um problema, identificando componentes discretos que podem ser vistos como um todo, ao invés de nos preocuparmos com seus detalhes.

Veremos esse princípio em ação quando começarmos a usar as classes de bibliotecas disponíveis em Java. Embora essas classes não sejam, estritamente falando, uma parte da linguagem, algumas delas estão tão intimamente associadas à criação de programas Java que geralmente são pensadas dessa maneira. A maioria das pessoas que cria programas em Java verifica constantemente as bibliotecas para ver se alguém já escreveu uma classe da qual possa fazer uso. Dessa forma, eles economizam uma enorme quantidade de esforço que pode ser melhor utilizada em outras partes do programa. O mesmo princípio se aplica à maioria das outras linguagens de programação, que também tendem a ter bibliotecas de classes úteis. Portanto, vale a pena familiarizar-se com o conteúdo da biblioteca e como usar as classes mais comuns. O poder da abstração acontece, pois geralmente não precisamos saber muito (se é que há alguma coisa) sobre como a classe se parece por dentro para poder usá-la efetivamente.

Se usarmos uma classe de biblioteca, seguiremos escrevendo código que cria instâncias dessas classes e nossos objetos estarão interagindo com os objetos da biblioteca. Portanto, a interação com os objetos também será muito importante neste capítulo.

Você verá que este conteúdo revisita continuamente e se baseia em temas que foram introduzidos nas aulas anteriores. Nós nos referimos a isso como uma "abordagem iterativa". Uma vantagem particular da abordagem é que ela o ajudará a aprofundar gradualmente sua compreensão dos tópicos à medida que avança no material.

Nesta aula, também estendemos nossa compreensão da abstração para ver que ela não significa apenas ocultar detalhes, mas também significa ver os recursos e padrões comuns que se repetem nos programas. Reconhecer esses padrões significa que geralmente podemos reutilizar parte ou todo o método ou classe que escrevemos anteriormente em uma nova situação. Isso se aplica particularmente ao examinar coleções e iteração.

4.2 A abstração da coleção

Uma das abstrações que exploraremos neste capítulo é a ideia de uma coleção - a noção de agrupar coisas para que possamos nos referir a elas e gerenciá-las todas juntas. Uma coleção pode ser grande (todos os alunos de uma universidade), pequena (os cursos que um dos alunos está fazendo) ou até vazia (as pinturas de Picasso que eu possuo).

Se possuirmos uma coleção de selos, autógrafos, pôsteres de shows, enfeites, música ou o que for, então algumas ações comuns que desejaremos executar de vez em quando na coleção, independentemente do que coletamos. Por exemplo, é provável que desejemos adicionar à coleção, mas também podemos reduzi-la - digamos, se temos duplicatas ou queremos arrecadar dinheiro para compras adicionais. Também podemos querer organizá-lo de alguma forma - por data de aquisição ou valor, talvez. O que estamos descrevendo aqui são operações típicas em uma coleção.

Em um contexto de programação, a abstração de coleção se torna uma classe de algum tipo, e as operações seriam métodos dessa classe. Uma coleção específica (minha coleção de músicas) seria uma instância da classe. Além disso, os itens armazenados em uma instância de coleção seriam eles próprios os objetos.

Aqui estão alguns exemplos de coleção mais obviamente relacionados a um contexto de programação:

Os calendários eletrônicos armazenam notas de eventos sobre compromissos, reuniões, aniversários etc. Novas notas são adicionadas à medida que eventos futuros são organizados e notas antigas são excluídas porque os detalhes de eventos passados não são mais necessários. As bibliotecas registram detalhes sobre os livros e periódicos que eles possuem. O catálogo muda à medida que novos livros são comprados e os antigos são armazenados ou descartados.

4.3 Um organizador para arquivos de música

As universidades mantêm registros de estudantes. Cada ano acadêmico adiciona novos registros à coleção, enquanto os registros daqueles que foram deixados são movidos para uma coleção de arquivos. Listar subconjuntos da coleção será comum: todos os alunos que cursam o primeiro ano do ensino médio ou todos os alunos que se formarão este ano, por exemplo. O número de itens armazenados em uma coleção varia de tempos em tempos. Até o momento, não conhecemos nenhum recurso do Java que nos permitisse agrupar números arbitrários de itens. Poderíamos, talvez, definir uma classe com muitos campos individuais para cobrir um número fixo, mas muito grande de itens, pois os programas geralmente precisam de uma solução mais geral do que a fornecida. Uma solução adequada não exigiria que soubéssemos antecipadamente quantos itens queremos agrupar ou que fixássemos um limite superior para esse número.

Portanto, iniciaremos nossa exploração da biblioteca Java observando uma classe que fornece a maneira mais simples possível de agrupar objetos, uma lista de tamanho flexível, sem classificação, mas ordenada: *ArrayList*. Nas próximas seções, usaremos o exemplo de acompanhar uma coleção de músicas pessoais para ilustrar como podemos agrupar um número arbitrário de objetos em um único objeto de contêiner.

Vamos escrever uma aula que pode nos ajudar a organizar nossos arquivos de música armazenados em um computador. Nossa turma não armazena os detalhes do arquivo; em vez disso, delegará essa responsabilidade à classe da biblioteca *ArrayList* padrão, o que economizará muito trabalho. Então, por que precisamos escrever nossa própria classe? Um ponto importante a ser lembrado ao lidar com as classes da biblioteca é que elas não foram escritas para nenhum cenário de aplicativo específico - são classes de uso geral. Um *ArrayList* pode armazenar objetos de registro do aluno, enquanto outro armazena lembretes de eventos. Isso significa que são as classes que escrevemos para usar as classes da biblioteca que fornecem as operações específicas do cenário, como o fato de estarmos lidando com arquivos de música ou reproduzindo um arquivo armazenado na coleção.

Por uma questão de simplicidade, a primeira versão deste projeto simplesmente funcionará com os nomes de arquivo de faixas de música individuais. Não haverá detalhes separados de título, artista, tempo de reprodução etc. Isso significa que pediremos apenas ao *ArrayList* para armazenar os objetos *String* que representam os nomes dos arquivos. Manter as coisas simples nesse estágio ajudará a evitar obscurecer os principais conceitos que estamos tentando ilustrar, que são a criação e o uso de um objeto de coleção. Mais adiante nesta aula, adicionaremos mais sofisticação, para tornar um organizador e reprodutor de música mais viável. Assumiremos que cada arquivo de música representa uma única faixa. Os arquivos de exemplo que fornecemos ao projeto têm o nome do artista e o título da faixa incorporados no nome do arquivo, e usaremos esse recurso mais tarde. Por enquanto, aqui estão as operações básicas que teremos na versão inicial do nosso organizador:

Permite adicionar faixas à coleção.

Ele não tem limite predeterminado no número de faixas que pode armazenar, além do limite de memória da máquina em que é executada.

Ele nos dirá quantas faixas existem na coleção.

Ele listará todas as faixas.

Veremos que a classe *ArrayList* facilita muito o fornecimento dessa funcionalidade a partir de nossa própria classe.

Observe que não estamos sendo muito ambiciosos nesta primeira versão. Esses recursos serão suficientes para ilustrar o básico da criação e uso da classe *ArrayList*, e as versões posteriores criarão outros recursos incrementados até termos algo mais sofisticado. (Mais importante, talvez, mais tarde adicionaremos a possibilidade de reproduzir os arquivos de música. Nossa primeira versão não poderá fazer isso.) Essa abordagem modesta e incremental tem muito mais probabilidade de levar ao sucesso do que tentar implementar tudo uma vez. Antes de analisarmos o código fonte necessário para fazer o uso dessa classe, é útil explorar o comportamento inicial do organizador da música.

Usando uma classe de biblioteca

O código 4.1 mostra a definição completa da nossa classe *MusicOrganizer*, que utiliza a classe da biblioteca *ArrayList*. Observe que as classes da biblioteca não aparecem no diagrama de classes *BlueJ*.

Bibliotecas de classes: um dos recursos das linguagens orientadas a objetos que as tornam poderosas é que elas geralmente são acompanhadas por bibliotecas de classes. Essas bibliotecas normalmente contêm muitas centenas ou milhares de classes diferentes que se mostraram úteis para desenvolvedores em uma ampla variedade de projetos diferentes. Java chama seus pacotes de bibliotecas. As classes da biblioteca são usadas exatamente da mesma maneira que usamos nossas próprias classes. As instâncias são construídas usando *new* e as classes têm campos, construtores e métodos.

Código 4.1 A classe *MusicOrganizer*

```
import java.util.ArrayList;

/**
* A class to hold details of audio files.
*
* @author David J. Barnes and Michael Kölling
* @version 2016.02.29
*/
public class MusicOrganizer
{
  // An ArrayList for storing the file names of music files.
  private ArrayList<String> files;

  /**
   * Create a MusicOrganizer
   */
  public MusicOrganizer()
  {
    files = new ArrayList<>();
  }

  /**
   * Add a file to the collection.
   * @param filename The file to be added.
   */
  public void addFile(String filename)
  {
    files.add(filename);
  }

  /**
   * Return the number of files in the collection.
   * @return The number of files in the collection.
   */
  public int getNumberOfFiles()
  {
    return files.size();
  }

  /**
   * List a file from the collection.
   * @param index The index of the file to be listed.
   */
  public void listFile(int index)
  {
    if(index >= 0 && index < files.size()) {
      String filename = files.get(index);
      System.out.println(filename);
    }
  }
```

```
/**
 * Remove a file from the collection.
 * @param index The index of the file to be removed.
 */
public void removeFile(int index)
{
    if(index >= 0 && index < files.size()) {
        files.remove(index);
    }
}
}
```

4.4.1 Importando uma classe de biblioteca

A primeira linha do arquivo de classe ilustra a maneira como obtemos acesso a uma classe de biblioteca em Java, por meio de uma declaração de importação:

import java.util.ArrayList;

Isso torna a classe *ArrayList* do pacote java.util disponível para nossa definição de classe. As instruções de importação sempre devem ser colocadas antes das definições de classe em um arquivo. Depois que um nome de classe é importado de um pacote, dessa maneira, podemos usar essa classe como se fosse uma de nossas próprias classes. Então, usamos *ArrayList* na "cabeça" da classe *MusicOrganizer* para definir um campo de arquivos:

private ArrayList<String> files;

Aqui, vemos uma nova construção: a menção de *String* entre colchetes angulares: <>. A necessidade disso foi mencionada na Seção 4.3, onde observamos que *ArrayList* é uma classe de coleção de uso geral - isto é, não restrita ao que pode armazenar. Quando criamos um objeto *ArrayList*, no entanto, precisamos ser específicos sobre o tipo de objetos que serão armazenados nessa instância específica. Podemos armazenar qualquer tipo que escolhermos, mas precisamos designar esse tipo ao declarar uma variável *ArrayList*. Classes como *ArrayList*, que são parametrizadas com um segundo tipo, são chamadas de classes genéricas (discutiremos mais detalhadamente mais adiante). Portanto, ao usar coleções, sempre precisamos especificar dois tipos: o tipo da coleção em si (aqui: *ArrayList*) e o tipo dos elementos que planejamos armazenar na coleção (aqui: *String*). Podemos ler a definição de tipo completa *ArrayList* <> como "*ArrayList of String*". Usamos essa definição de tipo como o tipo para nossa variável de arquivos. Como você deve esperar agora, vemos uma conexão estreita entre o corpo do construtor e os campos da classe, porque o construtor é responsável por inicializar os campos de cada instância. Assim, como o *ClockDisplay* criou objetos *NumberDisplay* para seus dois campos, aqui vemos o construtor do *MusicOrganizer* criando um objeto do tipo *ArrayList* e armazenando-o no campo files.

4.4.2 Notação de diamante

Observe que, ao criar a instância *ArrayList*, escrevemos a seguinte instrução: files = new *ArrayList* <> ();

Essa é a chamada notação de diamante (porque os dois colchetes angulares criam uma forma de diamante) e parece incomum. Vimos anteriormente que a nova declaração tem a seguinte forma: new type-name (parameters), e também vimos que o nome completo do tipo para nossa

coleção: ArrayList <String>. Portanto, com uma lista de parâmetros vazia, a instrução para criar o novo objeto de coleção deve ter a seguinte aparência:

files = new ArrayList<String>();

De fato, escrever esta declaração também funcionaria. A primeira versão, usando a notação de diamante (deixando de fora a menção do tipo *String*) é apenas uma notação de atalho por conveniência. Se a criação do objeto de coleção for combinada com uma atribuição, o compilador poderá calcular o tipo dos elementos da coleção a partir do tipo da variável no lado esquerdo da atribuição, e Java nos permitirá pular a definição novamente. O tipo de elemento é deduzido automaticamente do tipo de variável.

O uso deste formulário não altera o fato de que o objeto que está sendo criado poderá armazenar apenas objetos *String*; é apenas uma conveniência que nos poupa digitação e reduz nosso código.

4.4.3 Principais métodos de *ArrayList*

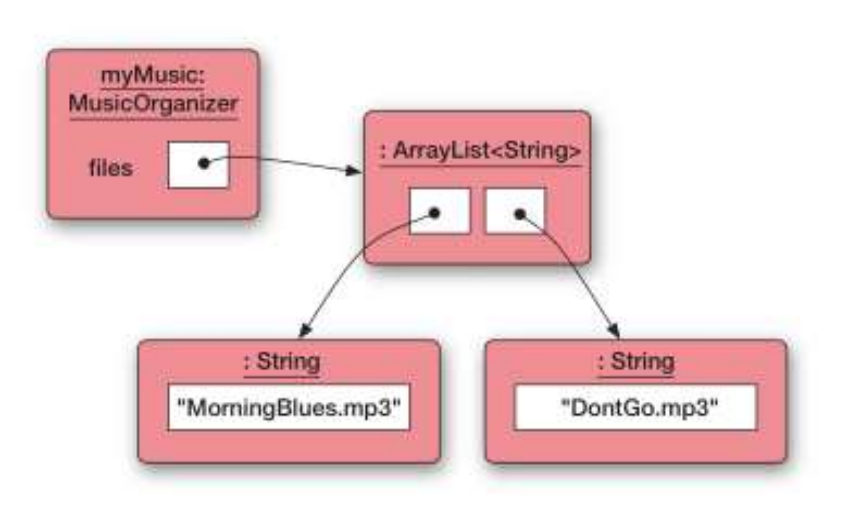
A classe *ArrayList* define muitos métodos, mas usaremos apenas quatro nesse estágio, para dar suporte à funcionalidade necessária: adicionar, dimensionar, obter e remover. Os dois primeiros são ilustrados nos métodos *addFile* e *getNumberOfFiles*, relativamente simples, respectivamente. O método *add* de um *ArrayList* armazena um objeto na lista, e o método *size* informa quantos itens estão armazenados nele atualmente. Veremos como os métodos "*get"* e "remove" funcionam na Seção 4.7, embora você provavelmente tenha alguma ideia, de antemão, simplesmente lendo o código dos métodos *listFile* e *removeFile*.

4.5 Estruturas de objetos com coleções

Para entender como um objeto de coleção, como um *ArrayList*, opera, é útil examinar um diagrama de objetos. A Figura 1 ilustra como um objeto *MusicOrganizer* pode aparecer com duas cadeias de nome de arquivo armazenadas nele. Compare a Figura 1 com a Figura 2, na qual um terceiro nome de arquivo foi armazenado. Há pelo menos três recursos importantes da classe *ArrayList* que você deve observar:

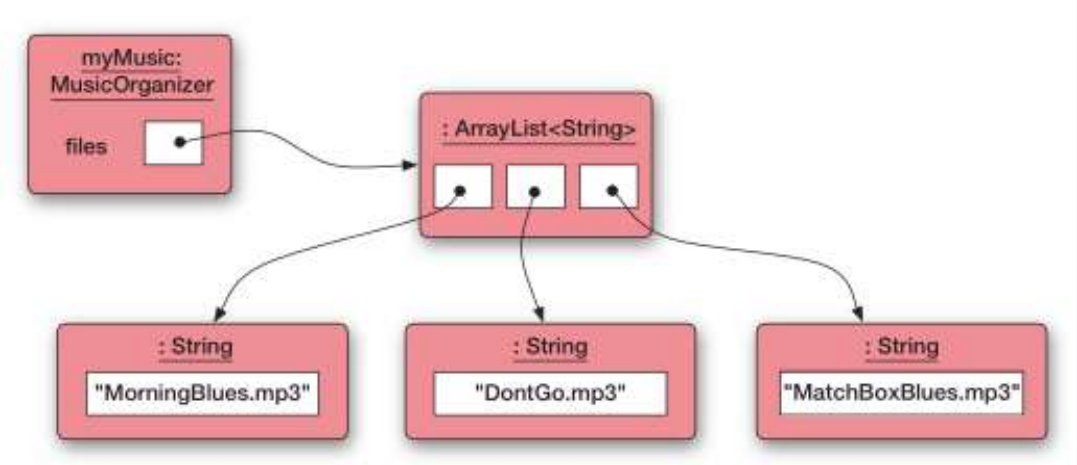
- Ele é capaz de aumentar sua capacidade interna conforme necessário: à medida que mais itens são adicionados, ele simplesmente abre espaço para eles;
- Ele mantém sua própria contagem privada de quantos itens está armazenando no momento. Seu método de tamanho retorna essa contagem;
- Ele mantém a ordem dos itens inseridos nele. O método *add* armazena cada novo item no final da lista. Mais tarde, você pode recuperá-los na mesma ordem.

Figura 1 – Como um objeto *MusicOrganizer* pode aparecer com duas cadeias de nome de arquivo armazenadas nele



Fonte: BlueJ.

Figura 2 – Exemplo 2 do objeto *MusicOrganizer*



Fonte: BlueJ.

Percebemos que o objeto *MusicOrganizer* parece bastante simples - ele possui apenas um único campo que armazena um objeto do tipo *ArrayList* <>. Todo o trabalho difícil é realizado no objeto *ArrayList*. Essa é uma das grandes vantagens do uso de classes de biblioteca: alguém investiu tempo e esforço para implementar algo útil, e estamos obtendo acesso a essa funcionalidade quase de graça usando essa classe. Nesta fase, não precisamos nos preocupar sobre como um *ArrayList* é capaz de suportar esses recursos. É suficiente apreciar o quão útil é essa capacidade. Lembre-se: Essa supressão de detalhes é um benefício que a abstração nos proporciona; significa que podemos utilizar *ArrayList* para escrever qualquer número de classes diferentes que exijam armazenamento de um número arbitrário de objetos.

O segundo recurso - o objeto *ArrayList*, que mantém sua própria contagem de objetos inseridos, tem consequências importantes na maneira como implementamos a classe *MusicOrganizer*.

Embora um organizador tenha um método *getNumberOfFiles*, na verdade não definimos um campo específico para registrar essas informações. Em vez disso, um organizador delega a responsabilidade de acompanhar o número de itens em seu objeto *ArrayList*.

Isso significa que um organizador não duplica as informações disponíveis de outros lugares. Se um usuário solicitar ao organizador informações sobre o número de nomes de arquivos, o organizador passará a pergunta para o objeto de arquivos e retornará a resposta que receber dele.

Duplicação de informações ou comportamento é algo que muitas vezes trabalhamos duro para evitar. A duplicação pode representar um esforço desperdiçado e pode levar a inconsistências, onde duas coisas que devem ser idênticas acabam não sendo, por erro. Teremos muito mais a dizer sobre duplicação de funcionalidades nos próximos capítulos.

4.6 Classes genéricas

A nova notação usando os colchetes angulares merece um pouco mais de discussão. O tipo do nosso campo de arquivos foi declarado como:

ArrayList <String>

A classe que estamos usando aqui é simplesmente chamada *ArrayList*, mas requer que um segundo tipo seja especificado como parâmetro, quando é usado para declarar campos ou outras variáveis. Classes que requerem esse parâmetro de tipo são chamadas de classes genéricas. Classes genéricas, em contraste com outras classes que vimos até agora, não definem um único tipo em Java, mas potencialmente muitos tipos.

A classe ArrayList, por exemplo, pode ser usada para especificar um *ArrayList* de *String*, um *ArrayList* de Person, um *ArrayList* de *Rectangle* ou um *ArrayList* de qualquer outra classe que tenhamos disponível. Cada *ArrayList* em particular é um tipo separado que pode ser usado em declarações de campos, parâmetros e valores de retorno. Poderíamos, por exemplo, definir os dois campos a seguir:

private ArrayList<Person> members;

private ArrayList<TicketMachine> machines;

Essas definições afirmam que os membros se referem a um *ArrayList* que pode armazenar objetos Person, enquanto as máquinas podem se referir a um *ArrayList* para armazenar objetos *TicketMachine*. Observe que *ArrayList* <*Person*> e *ArrayList* <*TicketMachine*> são tipos diferentes. Os campos não podem ser atribuídos um ao outro, mesmo que seus tipos tenham sido derivados da mesma classe *ArrayList*.

Classes genéricas são usadas para vários propósitos; coleções como *ArrayList* e algumas outras coleções que encontraremos em breve são as únicas classes genéricas com as quais precisamos lidar.

4.7 Numeração nas coleções

Ao explorar o projeto “music-organizer-v1”, observamos que era necessário usar valores de parâmetros começando em 0 para listar e remover nomes de arquivos na coleção. A razão por

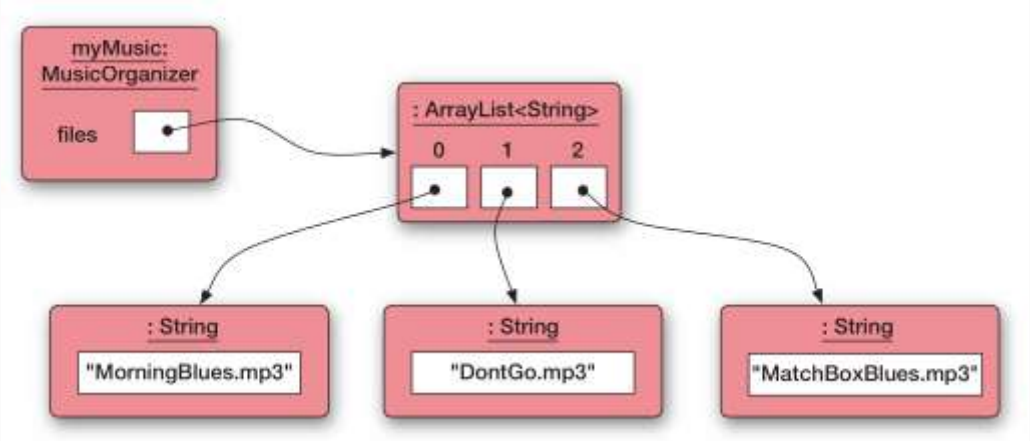
trás desse requisito é que os itens armazenados nas coleções *ArrayList* têm uma numeração ou posicionamento implícito que começa em 0. A posição de um objeto em uma coleção é mais conhecida como seu índice. O primeiro item adicionado a uma coleção recebe o número de índice 0, o segundo, o número de índice 1 e assim por diante. A Figura 3 ilustra a mesma situação que acima, com os números de índice mostrados no objeto *ArrayList*.

Isso significa que o último item de uma coleção tem o tamanho do índice-1. Por exemplo, em uma lista de 20 itens, o último estará no índice 19.

Os métodos *listFile* e *removeFile* ilustram a maneira como um número de índice é usado para obter acesso a um item em um *ArrayList*: um pelo método "*get"* e o outro pelo método "remove". Observe que os dois métodos garantem que seu valor de parâmetro esteja no intervalo de valores de índice válidos [0. . . size () - 1] antes de passar o índice para os métodos *ArrayList*. Esse é um bom hábito de validação estilística a ser adotado, pois evita a falha de uma chamada de método de classe de biblioteca ao transmitir valores de parâmetros que poderiam ser inválidos.

Armadilha se você não tomar cuidado; tente acessar um elemento de coleção que esteja fora dos índices válidos do *ArrayList*. Ao fazer isso, você receberá uma mensagem de erro e o programa será encerrado. Esse erro é chamado de erro de índice fora dos limites. Em Java, você verá uma mensagem sobre uma *IndexOutOfBoundsException*.

Figura 3 - Os números de índice mostrados no objeto *ArrayList*



Fonte: BlueJ.

4.7.1 O efeito da remoção na numeração

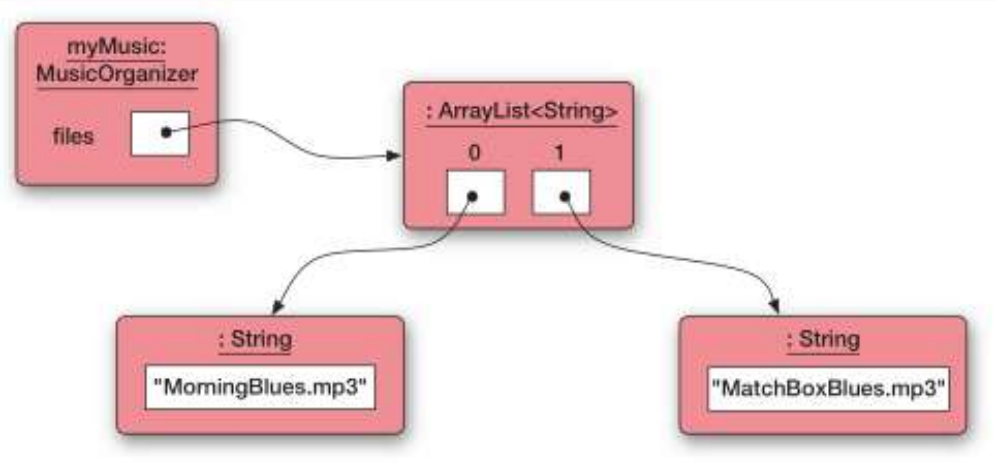
Além de adicionar itens a uma coleção, é comum querer remover itens, como vimos com o método *removeFile* no Código 4.1. A classe *ArrayList* possui um método remove que leva o índice do objeto a ser removido. Um detalhe do processo de remoção a ser observado é que ele pode alterar os valores de índice nos quais outros objetos da coleção são armazenados. Se um item com um número baixo de índice for removido, a coleção moverá todos os itens subsequentes

por uma posição para preencher a lacuna. Como consequência, seus números de índice serão reduzidos em 1.

A Figura 4 ilustra a maneira como alguns dos valores de índice de itens em um *ArrayList* são alterados pela remoção de um item do meio da lista. Começando com a situação representada na Figura 3, o objeto com o índice 1 foi removido. Como resultado, o objeto originalmente no número de índice 2 foi alterado para 1, enquanto o objeto no número de índice 0 permanece inalterado.

Além disso, veremos mais adiante que é possível inserir itens em um *ArrayList* em uma posição diferente do final dela. Isso significa que os itens que já estão na lista podem ter seus números de índice aumentados quando um novo item é adicionado. Os usuários precisam estar cientes dessa possível alteração de índices ao adicionar ou remover itens.

Figura 4 - Valores de índice de itens em um *ArrayList* são alterados pela remoção de um item do meio da lista



Fonte: BlueJ.

4.7.2 A utilidade geral da numeração com coleções

O uso de valores de índice inteiro para acessar objetos em uma coleção é algo que veremos repetidamente - não apenas com *ArrayList*s, mas também com vários tipos diferentes de coleções. Portanto, é importante entender o que vimos até agora: que os valores do índice começam em zero; que os objetos são numerados sequencialmente; e que geralmente não há lacunas nos valores de índice de objetos consecutivos na coleção.

O uso de valores inteiros como índices também facilita a expressão de exibições no código do programa, como "o próximo item" e "o item anterior" com relação a um item da coleção. Se um item estiver no índice p, então o “próximo” estará no índice (p + 1) e o “anterior” agora estará no índice (p – 1). Também podemos mapear seleções de idioma natural, como "os três primeiros", para a terminologia relacionada ao programa. Por exemplo, "os itens nos índices 0, 1 e 2" ou "os quatro últimos" podem ser "os itens nos índices (list.size() - 4) a (list.size() - 1)").

Poderíamos até imaginar percorrendo toda a coleção, tendo uma variável de índice inteiro cujo valor é inicialmente definido como zero e aumentado sucessivamente em 1, passando seu valor

para o método *get* para acessar cada item da lista em ordem (parando quando vai além do valor final do índice da lista). Mas estamos nos adiantando um pouco. No entanto, daqui a pouco, veremos como tudo isso funciona na prática, quando analisarmos os *loops* e a iteração.

Se o parâmetro não for válido, ele deverá imprimir uma mensagem de erro informando qual é o intervalo válido. Se o índice for válido, ele não imprime nada. Teste seu método no banco de objetos com parâmetros válidos e inválidos. Seu método ainda funciona quando você verifica um índice se a coleção estiver vazia?

4.8 Reproduzindo os arquivos de música

Um bom recurso do nosso organizador, além de manter uma lista de arquivos de música, nos permitiria reproduzi-los. Mais uma vez, podemos usar a abstração para nos ajudar aqui. Se tivermos uma aula que foi escrita especificamente para reproduzir arquivos de áudio, nossa turma organizadora não precisaria saber nada sobre como fazer isso; poderia simplesmente entregar o nome do arquivo para a classe *player* e deixá-lo fazer o resto.

Infelizmente, a biblioteca Java padrão não possui uma classe adequada para reproduzir arquivos mp3, que é o formato de áudio com o qual queremos trabalhar. No entanto, muitos programadores individuais estão constantemente escrevendo suas próprias classes úteis e as disponibilizando para uso de outras pessoas. Elas são frequentemente chamadas de “bibliotecas de terceiros” e são importadas e usadas da mesma maneira que as classes de biblioteca Java padrão.

Para a próxima versão do nosso projeto, usamos um conjunto de classes do javazoom.net para escrever nossa própria classe de tocador de música. Você pode encontrar isso na versão chamada “music-organizer-v2”. Os três métodos da classe *MusicPlayer* que usaremos são *playSample*, *startPlaying* e *stop*. Os dois primeiros levam o nome do arquivo de áudio para reproduzir.

O primeiro reproduz alguns segundos do início do arquivo e retorna quando termina a reprodução, enquanto o segundo inicia sua reprodução em segundo plano e retorna imediatamente o controle de volta ao organizador - daí a necessidade do método *stop*, caso você queira, para cancelar a reprodução. O código 4.2 mostra os novos elementos da classe *MusicOrganizer* que acessam parte dessa funcionalidade de reprodução.

Código 4.2

```
import java.util.ArrayList;

/**
* A class to hold details of audio files.
* This version can play the files.
*
* @author David J. Barnes and Michael Kölling
* @version 2016.02.29
*/
public class MusicOrganizer
{
  // An ArrayList for storing the file names of music files.
  private ArrayList<String> files;
  // A player for the music files.
  private MusicPlayer player;

  /**
   * Create a MusicOrganizer
   */
  public MusicOrganizer()
  {
     files = new ArrayList<>();
     player = new MusicPlayer();
  }

  /**
   * Start playing a file in the collection.
   * Use stopPlaying() to stop it playing.
   * @param index The index of the file to be played.
   */
  public void startPlaying(int index)
  {
     String filename = files.get(index);
     player.startPlaying(filename);
  }


  /**
   * Stop the player.
   */
  public void stopPlaying()
  {
     player.stop();
  }
}
```

4.8.1 Resumo do organizador da música

Fizemos um bom progresso com o básico de organização de nossa coleção de músicas. Podemos armazenar os nomes de qualquer número de arquivos de música e até reproduzi-los. Fizemos isso com pouco esforço de codificação, porque conseguimos pegar a funcionalidade fornecida pelas classes da biblioteca: *ArrayList* da biblioteca Java padrão e um *music player* que usa uma biblioteca de classes de terceiros. Também conseguimos fazer isso com relativamente pouco conhecimento do funcionamento interno dessas classes de biblioteca; foi suficiente conhecer os nomes, tipos de parâmetros e tipos de retorno dos principais métodos.

No entanto, ainda faltam algumas funcionalidades importantes, se queremos um programa realmente útil - o mais óbvio é a falta de uma maneira de listar toda a coleção, por exemplo. Este será o tópico da próxima seção, à medida que apresentarmos a primeira de várias estruturas de controle de *loop* Java.

4.9 Processando uma coleção inteira

No final da última seção, dissemos que seria útil ter um método no organizador de músicas que lista todos os nomes de arquivos armazenados na coleção. Sabendo que cada nome de arquivo na coleção possui um número de índice exclusivo, uma maneira de expressar o que queremos é dizer que desejamos exibir o nome do arquivo armazenado em números de índice crescentes consecutivos a partir de zero.

```
System.out.println(files.get(0));

System.out.println(files.get(1));

System.out.println(files.get(2));

etc.
```

Quantas instruções *println* seriam necessárias para concluir o método?

Depende de quantos nomes de arquivos estão na lista no momento em que são impressos. Se houver três, serão necessárias três instruções *println*; se houver quatro, serão necessárias quatro declarações; e assim por diante. Os métodos *listFile* e *removeFile* ilustram que o intervalo de números de índice válidos a qualquer momento é [0 a (tamanho () - 1)]. Portanto, um método *listAllFiles* também teria que levar em conta esse tamanho dinâmico para fazer seu trabalho. O que temos aqui é o requisito de fazer algo várias vezes, mas o número de vezes depende de circunstâncias que podem variar - nesse caso, o tamanho da coleção. Nós atenderemos a esse tipo de requisito em quase todos os programas que escrevemos, e a maioria das linguagens de programação tem várias maneiras de lidar com isso através do uso de instruções de loop, também conhecidas como estruturas de controle iterativas.

O primeiro loop que apresentaremos para listar os arquivos é especial para uso com coleções, o que evita completamente a necessidade de usar uma variável de índice: isso é chamado de *loop* para cada um.

4.9.1 O *loop for-each*

Um *loop for-each* é uma maneira de executar um conjunto de ações repetidamente nos itens de uma coleção, mas sem ter que escrever essas ações mais de uma vez. Podemos resumir a sintaxe Java e as ações de um *loop for-each* no seguinte pseudocódigo:

```
for(ElementType element : collection) {

  loop body

}
```

A principal parte nova do Java é a palavra para. A linguagem Java possui duas variações do loop for: uma é o *loop for-each*, que estamos discutindo aqui; o outro é simplesmente chamado de

loop for e será discutido no Capítulo 7. Um *loop for-each* tem duas partes: um cabeçalho de loop (a primeira linha da instrução loop) e um corpo de loop após o cabeçalho. O corpo contém as afirmações que desejamos executar repetidamente. O *loop for-each* recebe o nome da maneira como podemos lê-lo: se lermos a palavra-chave para "*for each*" e os dois pontos no cabeçalho do loop como "*in*", a estrutura de código mostrada acima começará a gerar mais sentido, como neste pseudocódigo:

para cada elemento da coleção faça: {

corpo do laço

}

Ao comparar esta versão com o pseudocódigo original da primeira versão, você percebe que o elemento foi gravado na forma de uma declaração de variável como elemento *ElementType*. Esta seção realmente declara uma variável que é usada para cada elemento da coleção, por sua vez. Antes de discutir mais, vejamos um exemplo de código Java real.

O código 4.3 mostra uma implementação de um método *listAllFiles* que lista todos os nomes de arquivos atualmente no *ArrayList* do organizador que usam esse *loop* para cada um.

Código 4.3

```
/**
 * Show a list of all the files in the collection.
 */
public void listAllFiles()
{
    for(String filename : files) {
        System.out.println(filename);
    }
}
```

Nesse *loop for-each*, o corpo do *loop* - consistindo em uma única instrução System.out.*println* é executado repetidamente, uma vez para cada elemento nos arquivos *ArrayList*. Se, por exemplo, houvesse quatro *strings* na lista, a instrução *println* seria executada quatro vezes.

Cada vez que a instrução é executada, o nome do arquivo da variável é definido para conter um dos elementos da lista: primeiro o do índice 0, depois o do índice 1 e assim por diante. Assim, cada elemento da lista é impresso. Vamos dissecar o *loop* em um pouco mais detalhadamente. A palavra-chave “para” apresenta o *loop*. É seguido por um par de parênteses, dentro do qual os detalhes do *loop* são definidos. O primeiro dos detalhes é a declaração *String filename*; isso declara um novo nome de arquivo de variável local que será usado para manter os elementos da lista em ordem. Chamamos essa variável de variável de *loop*.

Podemos escolher o nome dessa variável, assim como o de qualquer outra variável; não precisa ser chamado de "nome do arquivo". O tipo da variável de loop deve ser o mesmo que o tipo de elemento declarado da coleção que vamos usar - *String* no nosso caso. Em seguida, segue dois pontos e a variável que contém a coleção que queremos processar. Nesta coleção, cada elemento será atribuído à variável de *loop*, por sua vez; e para cada uma dessas atribuições, o corpo do *loop* é executado uma vez. Nele, usamos a variável *loop* para se referir a cada elemento. Para testar sua compreensão de como esse *loop* funciona.

Vimos agora como podemos usar um *loop for-each* para executar alguma operação (o corpo do *loop*) em todos os elementos de uma coleção. Este é um grande passo em frente, mas não resolve todos os nossos problemas. Às vezes precisamos de um pouco mais de controle, e o Java fornece uma construção de *loop* diferente para nos permitir fazer mais: o *loop while*.

4.9.2 Processamento seletivo de uma coleção

O método *listAllFiles* ilustra a utilidade fundamental de um *loop for-each*, este fornece acesso a todos os elementos de uma coleção, em ordem, por meio da variável declarada no cabeçalho do *loop*. Ele não nos fornece a posição de índice de um elemento, mas nem sempre precisamos disso, portanto isso não é necessariamente um problema. Ter acesso a todos os itens da coleção, no entanto, não significa que precisamos fazer a mesma coisa com todos; podemos ser mais seletivos que isso. Por exemplo, podemos querer listar apenas a música de um determinado artista ou precisar encontrar todas as músicas com uma frase específica no título. Não há nada que nos impeça de fazer isso, porque o corpo de um *loop for-each* é apenas um bloco comum e podemos usar as instruções Java que desejamos dentro dele. Portanto, deve ser fácil usar uma instrução *if* no corpo para selecionar os arquivos que queremos.

O código 4.4 mostra um método para listar apenas os nomes de arquivo na coleção que contêm uma sequência específica.

```
/**
 * List the names of files matching the given search string.
 * @param searchString The string to match.
 */
public void listMatching(String searchString)
{
    for(String filename : files) {
        if(filename.contains(searchString)) {
            // A match.
            System.out.println(filename);
        }
    }
}
```

Usando uma instrução *if* e o resultado booleano do método *contains* da classe *String*, podemos "filtrar" quais nomes de arquivos devem ser impressos e quais não. Se o nome do arquivo não corresponder, simplesmente o ignoramos - nenhuma outra parte é necessária. O critério de filtragem (o teste na instrução *if*) pode ser o que quisermos.

4.9.3 Uma limitação do uso de *strings*

A essa altura, podemos ver que simplesmente ter sequências de nomes de arquivos contendo todos os detalhes da faixa de música não é realmente satisfatório. Por exemplo, suponha que queremos encontrar todas as faixas com a palavra "amor" no título. Usando a técnica de correspondência não sofisticada descrita acima, também encontraremos faixas de artistas cujos nomes possuem essa sequência de caracteres (por exemplo, Glover). Embora isso possa não parecer um problema particularmente grande, ele parece um pouco "barato", e deve ser possível fazer melhor com apenas um pouco mais de esforço. O que realmente precisamos é de

uma classe separada, como *Track*, digamos que armazene os detalhes do artista e do título, independentemente do nome do arquivo. Então poderíamos combinar títulos, mais facilmente, separadamente dos artistas. O *ArrayList* <> no organizador se tornaria um *ArrayList* <*Track*>.

À medida que desenvolvemos o projeto organizador de música nas seções posteriores, eventualmente avançaremos para uma melhor estrutura, introduzindo uma classe *Track*.

4.9.4 Resumo do *loop for-each*

O *loop for-each* é sempre usado para iterar sobre uma coleção. Ele nos fornece uma maneira de acessar todos os itens da coleção em sequência, um por um, e processar esses itens da maneira que desejarmos. Podemos optar por fazer o mesmo com cada item (como fizemos ao imprimir a lista completa) ou podemos ser seletivos e filtrar a lista (como fizemos quando imprimimos apenas um subconjunto da coleção). O corpo do *loop* pode ser tão complicado quanto acreditamos. Com sua simplicidade essencial, necessariamente, vêm algumas limitações. Por exemplo, uma restrição é que não podemos alterar o que é armazenado na coleção enquanto iteramos sobre ela, adicionando novos itens a ela ou removendo seus itens. Isso não significa, no entanto, que não possamos alterar os estados dos objetos que já estão na coleção. Também vimos que o *loop for-each* não nos fornece um valor de índice para os itens da coleção. Se queremos um, temos que declarar e manter nossa própria variável local. A razão para isso tem a ver com abstração, novamente. Ao lidar com coleções e iterá-las, vale a pena ter duas coisas em mente:

Um *loop for-each* fornece uma estrutura de controle geral para iterar sobre diferentes tipos de coleção.

Existem alguns tipos de coleções que naturalmente não associam índices inteiros aos itens que eles armazenam. Vamos encontrar alguns deles nas próximas aulas. Portanto, o *loop for-each* abstrai a tarefa de processar uma coleção completa, elemento por elemento, e é capaz de lidar com diferentes tipos de coleção. Não precisamos saber os detalhes de como isso é gerenciado.

Uma das perguntas que não fizemos é se um *loop for-each* pode ser usado se quisermos parar parcialmente no processamento da coleção. Por exemplo, suponha que, em vez de tocar todas as faixas de nosso artista escolhido, apenas quiséssemos encontrar a primeira e tocá-la, sem precisar avançar. Embora, em princípio, seja possível fazer isso usando um *loop for-each*, nossa prática e conselhos não é usar um *loop for-each* para tarefas que talvez não precisem processar toda a coleção. Em outras palavras, recomendamos o uso de um *loop for-each* apenas se você definitivamente quiser processar a coleção inteira. Novamente, declarado de outra maneira, uma vez iniciado o *loop*, você sabe ao certo quantas vezes o corpo será executado - isso será igual ao tamanho da coleção. Esse estilo geralmente é chamado de iteração definida. Para tarefas nas quais você pode querer parar no meio do caminho, há *loops* mais apropriados para usar - por exemplo, o *loop while*, que apresentaremos a seguir. Nesses casos, o número de vezes que o corpo do *loop* será executado é menos certo; normalmente depende do que acontece durante a iteração. Esse estilo costuma ser chamado de iteração indefinida, e nós o exploraremos a seguir.

4.10 Iteração indefinida

O uso de um *loop for-each* nos deu nossa primeira experiência com o princípio de realizar algumas ações repetidamente. As instruções dentro do corpo do *loop* são repetidas para cada item na coleção associada e a iteração para quando atingimos o final da coleção. Um *loop for-each* fornece iteração definida; dado o estado de uma coleção específica, o corpo do *loop* será executado o número de vezes que corresponde exatamente ao tamanho dessa coleção. Mas há muitas situações em que queremos repetir algumas ações, mas não podemos prever com antecedência exatamente quantas vezes isso pode acontecer. Um *loop for-each* não nos ajuda nesses casos.

Imagine, por exemplo, que você perdeu suas chaves e precisa encontrá-las antes de poder sair de casa. Sua pesquisa modelará uma iteração indefinida, porque haverá muitos lugares diferentes para procurar e você não poderá prever com antecedência quantos lugares precisará pesquisar antes de encontrar as chaves; afinal, se você pudesse prever isso, iria direto para onde eles estão! Então, você fará algo como compor mentalmente uma lista de possíveis lugares em que eles poderiam estar e depois visitará cada lugar até encontrá-los. Uma vez encontrado, você deseja parar de procurar em vez de completar a lista (o que seria inútil).

O que temos aqui é um exemplo de iteração indefinida: a ação (pesquisa) será repetida um número imprevisível de vezes, até que a tarefa seja concluída. Cenários semelhantes à pesquisa de teclas são comuns em situações de programação. Embora nem sempre procuremos algo, situações em que queremos continuar fazendo algo até que a repetição não seja mais necessária são frequentemente encontradas. De fato, eles são tão comuns que a maioria das linguagens de programação fornece pelo menos uma - e geralmente mais de uma - construção de *loop* para expressá-las. Como o que estamos tentando fazer com essas construções de *loop* é tipicamente mais complexo do que apenas iterar uma coleção completa do começo ao fim, elas exigem um pouco mais de esforço para serem entendidas. Mas esse esforço será bem recompensado pela maior variedade de coisas que podemos conseguir com eles. Nosso foco aqui será o *loop while* do Java, que é semelhante aos *loops* encontrados em outras linguagens de programação.

4.10.1 O *loop while*

Um *loop while* consiste em um cabeçalho e um corpo; o corpo deve ser executado repetidamente. Aqui está a estrutura de um *loop while* em que a condição booleana e o corpo do *loop* são pseudocódigo, mas todo o resto é a sintaxe Java:

```
while (condição booleana) {
  corpo do laço
}
```

O loop é introduzido com a palavra-chave *while*, seguida por uma condição booleana. Em última análise, a condição é o que controla quantas vezes um *loop* específico irá iterar. A condição é avaliada quando o controle do programa atinge o *loop* pela primeira vez e é reavaliada cada vez que o corpo do *loop* é executado. É isso que dá ao *loop while* seu caráter indefinido - o processo de reavaliação. Se a condição for avaliada como verdadeira, o corpo será executado; e uma vez que é avaliado como falso, a iteração é concluída. O corpo do *loop* é então pulado e a execução continua com o que se segue imediatamente após o *loop*. Observe que a condição pode

realmente ser avaliada como falsa na primeira vez em que é testada. Se isso acontecer, o corpo não será executado. Essa é uma característica importante do *loop while*: o corpo pode ser executado zero vezes, ao invés de sempre pelo menos uma vez.

Antes de olharmos para um exemplo Java adequado, vejamos uma versão em pseudocódigo da busca de chaves descrita anteriormente, para tentar desenvolver uma sensação de como um *loop while* funciona. Aqui está uma maneira de expressar a pesquisa:

```
while (as chaves estão faltando) {
  olhe no próximo lugar
}
```

Quando chegamos ao *loop* pela primeira vez, a condição é avaliada: as chaves estão ausentes. Isso significa que entramos no corpo do laço e olhamos para o próximo lugar em nossa lista mental.

Feito isso, retornamos à condição e a reavaliamos. Se tivermos encontrado as chaves, o *loop* estará concluído e podemos pular o loop e sair de casa. Se as chaves ainda estiverem faltando, voltaremos ao corpo do *loop* e olharemos para o próximo lugar. Esse processo repetitivo continua até que as chaves não estejam mais ausentes.

Observe que poderíamos expressar da mesma forma a condição do loop, da seguinte maneira:

```
while (não (as chaves foram encontradas)) {
  olhe no próximo lugar
}
```

A distinção é sutil - uma expressa como um status a ser alterado e a outra como uma meta que ainda não foi alcançada. Reserve um tempo para ler as duas versões cuidadosamente, para ter certeza de que entende como cada uma delas funciona. Ambos são igualmente válidos e refletem as escolhas de expressão que teremos que fazer ao escrever loops reais. Nos dois casos, o que escrevemos dentro do loop quando as chaves são finalmente encontradas significa que as condições do loop “mudam” de verdadeiro para falso na próxima vez que forem avaliadas.

4.10.2 Iterando com uma variável de índice

Para o nosso primeiro *loop while* no código Java correto, escreveremos uma versão do método *listAllFiles* mostrado no Código 4.3. Isso realmente não ilustra o caráter indefinido dos *loops while*, mas fornece uma comparação útil com o equivalente, familiar para cada exemplo. A versão *while-loop* é mostrada no Código 4.5. Um recurso importante é a maneira como uma variável inteira (índice) é usada para acessar os elementos da lista e controlar o comprimento da iteração.

Código 4.5

```
/**
 * Show a list of all the files in the collection.
 */
public void listAllFiles()
{
    int index = 0;
    while(index < files.size()) {
        String filename = files.get(index);
        System.out.println(filename);
        index++;
    }
}
```

É imediatamente óbvio que a versão *while-loop* requer mais esforço de nossa parte para programá-la.

Considerar: temos que declarar uma variável para o índice da lista e precisamos inicializá-la como 0 para o primeiro elemento da lista. A variável deve ser declarada fora do *loop*. Temos que descobrir como expressar a condição do *loop* para garantir que ele pare no momento certo.

Os elementos da lista não são buscados automaticamente da coleção e atribuídos a uma variável para nós. Em vez disso, precisamos fazer isso sozinhos, usando o método *get* do *ArrayList*. O nome do arquivo variável será local para o corpo do loop.

Temos que lembrar de incrementar a variável contador (índice), para garantir que a condição do *loop* acabe se tornando falsa quando chegarmos ao final da lista. A declaração final no corpo do *loop while* ilustra um operador especial para incrementar uma variável numérica em 1:

index ++;

Isso é equivalente a

índice = índice + 1;

Até agora, o *loop for-each* é claramente melhor para o nosso propósito. Foi menos difícil escrever e é mais seguro. A razão pela qual é mais seguro é que sempre é garantido que ele chegue ao fim. Em nossa versão *while-loop*, é possível cometer um erro que resulta em um *loop* infinito. Se esquecermos de incrementar a variável de índice (a última linha no corpo do *loop*), a condição do *loop* nunca se tornaria falsa e ele iteraria indefinidamente. Este é um erro de programação típico que pega até programadores experientes de tempos em tempos. O programa será executado para sempre. Se o *loop* em tal situação não contiver uma declaração de saída, o programa parecerá "travar": parece não fazer nada e não responde a nenhum clique do mouse ou pressionamento de tecla. Na realidade, o programa faz muito. Ele executa o *loop* repetidamente, mas não podemos ver nenhum efeito disso, e o programa parece “ter morrido”. No BlueJ, isso geralmente pode ser detectado pelo fato de o indicador de "execução" listrado em vermelho e branco permanecer aceso enquanto o programa parece não estar fazendo nada.

Então, quais são os benefícios de um *loop while* sobre um *loop for-each*? Eles são duplos: primeiro, o *loop while* não precisa estar relacionado a uma coleção (podemos fazer *loop* em qualquer condição que possamos escrever como expressão booleana); segundo, mesmo se estivermos usando o *loop* para processar a coleção, não precisamos processar todos os elementos. Em vez disso, poderíamos parar mais cedo, se quiséssemos incluir outro componente na condição do *loop* que expressa por que queremos parar. É claro que, estritamente falando, a condição do *loop* realmente expressa se queremos continuar, e é a negação disso que faz com que o *loop* pare.

Uma vantagem de ter uma variável de índice explícita é que podemos usar seu valor dentro e fora do *loop*, o que não estava disponível para nós nos exemplos de cada um. Assim, podemos incluir o índice na lista, se desejarmos. Isso facilitará a escolha de uma faixa por sua posição na lista. Por exemplo:

```
int índice = 0;

while (index < files.size()) {

  String nomedoarquivo = files.get (index);

  // Prefixe o nome do arquivo com o índice da faixa.

  System.out.println (index + ": " + nomedoarquivo);

  index ++;

}
```

Ter uma variável de índice local pode ser particularmente importante ao pesquisar em uma lista, pois pode fornecer um registro de onde o item foi localizado, que ainda estará disponível quando o *loop* terminar. Veremos isso na próxima seção.

4.10.3 Pesquisando uma coleção

A pesquisa é uma das formas mais importantes de iteração que você encontrará. É vital, portanto, ter uma boa compreensão de seus elementos essenciais. O tipo de estrutura de *loop* resultante ocorre repetidamente em situações práticas de programação.

A principal característica de uma pesquisa é que ela envolve iteração indefinida; isso é necessariamente verdade, porque se soubéssemos exatamente onde procurar, não precisaríamos de uma pesquisa! Em vez disso, temos que iniciar uma pesquisa e será necessário um número desconhecido de iterações antes de obtermos sucesso. Isso implica que um *loop for-each* é inadequado para uso durante a pesquisa, porque ele completará todo o seu conjunto de iterações.

Em situações reais de pesquisa, precisamos levar em conta o fato de que a pesquisa pode falhar: podemos ficar sem lugares para procurar. Isso significa que normalmente temos duas possibilidades de conclusão a serem consideradas ao escrever um *loop* de pesquisa:

A pesquisa é bem-sucedida após um número indefinido de iterações.

A pesquisa falha após esgotar todas as possibilidades.

Ambos devem ser levados em consideração ao escrever a condição do *loop*. Como a condição deve ser avaliada como verdadeira se quisermos iterar mais uma vez, cada um dos critérios de acabamento deve, por si só, fazer com que a condição seja avaliada como falsa para interromper o *loop*. O fato de acabarmos pesquisando a lista inteira quando a pesquisa falha não transforma uma pesquisa com falha em um exemplo de iteração definida. A principal característica da iteração definida é que você pode determinar o número de iterações quando o *loop* iniciar. Este não será o caso de uma pesquisa. Se estivermos usando uma variável de índice para percorrer elementos sucessivos de uma coleção, é fácil identificar uma pesquisa com falha: a variável de índice terá sido incrementada além do item final da lista. Essa é exatamente a situação abordada no método *listAllFiles* no Código 4.5, onde a condição é:

while (index <files.size ())

A condição expressa que queremos continuar enquanto o índice estiver dentro do intervalo de índice válido da coleção; assim que tiver sido incrementado fora da faixa, queremos que o *loop* pare. Essa condição funciona mesmo se a lista estiver completamente vazia. Nesse caso, o índice será inicializado como zero e a chamada para o método *size* retornará zero também. Como zero não é menor que zero, o corpo do *loop* não será executado, o que queremos.

Também precisamos adicionar uma segunda parte à condição que indica se já encontramos o item de pesquisa e para a pesquisa quando o encontramos. Vimos na Seção 4.10.1 que geralmente podemos expressar isso de maneira positiva ou negativa, por meio de variáveis booleanas definidas adequadamente:

Uma variável chamada *search* (ou ausente, digamos) inicialmente definida como "*true"* pode manter a pesquisa em andamento até que ela seja "false" dentro do *loop* quando o item for encontrado. Uma variável chamada "encontrada", inicialmente definida como falsa e usada na condição como! Encontrada, pode manter a pesquisa até que seja definida como verdadeira quando o item for encontrado. Aqui estão os dois fragmentos de código correspondentes que expressam a condição completa nos dois casos:

int index = 0;

boolean searching = true;

while (index < files.size() && searching)

ou

int índice = 0;

boolean encontrado = false;

while (índice < files.size () && !encontrado)

Reserve um tempo para entender esses dois fragmentos, que realizam exatamente o mesmo controle de *loop*, mas expressos de maneiras ligeiramente diferentes. Lembre-se de que a condição como um todo deve ser verdadeira se quisermos continuar procurando e falsa se quisermos parar de procurar, por qualquer motivo. Discutimos o operador "*and*" && na aula 01, da unidade 2, que avalia apenas como verdadeiro se ambos os operandos forem verdadeiros.

A versão completa de um método para procurar o primeiro nome de arquivo correspondente a uma determinada sequência de pesquisa pode ser vista no Código 4.6 (music-organizer-v4). O método retorna o índice do item como resultado. Observe que precisamos encontrar uma

maneira de indicar ao responsável pela chamada do método se a pesquisa falhou. Nesse caso, optamos por retornar um valor que não pode representar um local válido na coleção - um valor negativo. Essa é uma técnica comumente usada em situações de pesquisa: o retorno de um valor fora dos limites para indicar falha.

Código 4.6

```
/**
 * Find the index of the first file matching the given
 * search string.
 * @param searchString The string to match.
 * @return The index of the first occurrence, or -1 if
 *         no match is found.
 */
public int findFirst(String searchString)
{
    int index = 0;
    // Record that we will be searching until a match is found.
    boolean searching = true;

    while(searching && index < files.size()) {
        String filename = files.get(index);
        if(filename.contains(searchString)) {
            // A match. We can stop searching.
            searching = false;
        }
        else {
            // Move on.
            index++;
        }
    }
    if(searching) {
        // We didn't find it.
        return -1;
    }
    else {
        // Return where it was found.
        return index;
    }
}
```

Pode ser tentador tentar ter apenas uma condição no *loop*, mesmo que haja duas razões distintas para finalizar a pesquisa. Uma maneira de fazer isso seria organizar artificialmente o valor do índice para ser muito grande se encontrarmos o que estamos procurando. Essa é uma prática que desencorajamos, porque torna os critérios de terminação do *loop* enganosos e a clareza é sempre preferida.

4.10.4 Alguns exemplos de não coleta

Os *loops* não são usados apenas com coleções. Existem muitas situações em que queremos repetir um bloco de declarações que não envolva uma coleção. Aqui está um exemplo que imprime todos os números pares de 0 a 30:

```
int index = 0;

while(index <= 30) {

  System.out.println(index);

  index = index + 2;

}
```

De fato, esse é o uso de um *loop while* para iteração definida, porque fica claro no início quantos números serão impressos. No entanto, não podemos usar um *loop for-each*, porque eles só podem ser usados ao iterar sobre coleções. Mais tarde, encontraremos um terceiro *loop* relacionado (o *loop* for) que seria mais apropriado para este exemplo em particular. Para testar sua própria compreensão dos *loops while*, além das coleções.

4.11 Melhorando a estrutura - a classe *Track*

Vimos em alguns lugares que o uso de *strings* para armazenar todos os detalhes da faixa não é totalmente satisfatório e dá ao nosso *music player* uma sensação bastante barata. Qualquer *player* comercial nos permitirá procurar faixas por artista, título, álbum, gênero, etc. e provavelmente incluirá detalhes adicionais, como tempo de reprodução e número da faixa. Um dos poderes da orientação a objetos é que ela permite projetar classes que modelam de perto a estrutura e os comportamentos inerentes às entidades do mundo real que estamos frequentemente tentando representar. Isso é conseguido através da escrita de classes cujos campos e métodos correspondem aos atributos. Já sabemos o suficiente sobre como escrever classes básicas com campos, construtores e métodos de acessador e mutador que podemos projetar facilmente uma classe *Track* que possui campos para armazenar informações separadas sobre artistas e títulos, por exemplo. Dessa forma, poderemos interagir com os objetos no organizador da música de uma maneira que pareça mais natural.

Portanto, é hora de deixar de armazenar os detalhes da faixa como *strings*, porque ter uma classe *Track* separada é a maneira mais apropriada de representar os principais itens de dados - faixas de música - que estamos usando no programa. No entanto, não seremos muito ambiciosos. Um dos obstáculos óbvios a serem superados é como obter as informações separadas que desejamos armazenar em cada objeto *Track*. Uma maneira seria pedir ao usuário para inserir o artista, título, gênero etc., sempre que adicionar um arquivo de música ao organizador. No entanto, isso seria bastante lento e trabalhoso, portanto, para este projeto, escolhemos um conjunto de arquivos de música que possuem o artista e o título como parte do nome do arquivo. Criamos uma classe auxiliar para o nosso aplicativo (chamada *TrackReader*) que procurará arquivos de música em uma pasta específica e usará seus nomes para preencher partes dos objetos correspondentes da faixa. Não vamos nos preocupar com os detalhes de como isso é feito nesta fase. (Mais adiante neste livro, discutiremos as técnicas e classes de biblioteca usadas na classe *TrackReader*.) Uma implementação desse design está no music-organizer-v5.

Aqui estão alguns dos principais pontos a serem procurados nesta versão:

O principal a analisar são as alterações que fizemos na classe *MusicOrganizer*, passando de armazenar objetos *String* no *ArrayList* para armazenar objetos *Track* (Código 4.7). Isso afetou a maioria dos métodos que desenvolvemos anteriormente. Ao listar detalhes das faixas em *listAllTracks*, solicitamos que o objeto *Track* retorne uma *String* contendo seus detalhes. Isso mostra que projetamos a classe *Track* para ser responsável por fornecer os detalhes a serem impressos, como artista e título. Este é um exemplo do que é chamado de design orientado à responsabilidade, que abordamos com mais detalhes em um capítulo posterior. No método *playTrack*, agora precisamos recuperar o nome do arquivo do objeto *Track* selecionado antes de repassá-lo ao *player*. Na biblioteca de músicas, adicionamos o código para ler automaticamente da pasta de áudio e algumas instruções de impressão para exibir algumas informações.

4.11 Melhorando a estrutura - a classe *Track*

Código 4.7

```
import java.util.ArrayList;
/**
* A class to hold details of audio tracks.
* Individual tracks may be played.
*
* @author David J. Barnes and Michael Kölling
* @version 2016.02.29
*/
public class MusicOrganizer
{
    // An ArrayList for storing music tracks.
    private ArrayList<Track> tracks;
    // A player for the music tracks.
    private MusicPlayer player;
    // A reader that can read music files and load them as tracks.
    private TrackReader reader;


    /**
     * Create a MusicOrganizer
     */
    public MusicOrganizer()
```

```
{
    tracks = new ArrayList<>();
    player = new MusicPlayer();
    reader = new TrackReader();
    readLibrary("../audio");
    System.out.println("Music library loaded. " + getNumberOfTracks() + " tracks.");
    System.out.println();
}


/**
 * Add a track to the collection.
 * @param track The track to be added.
 */
public void addTrack(Track track)
{
    tracks.add(track);
}

/**
 * Play a track in the collection.
 * @param index The index of the track to be played.
 */
public void playTrack(int index)
{
    if(indexValid(index)) {
        Track track = tracks.get(index);
        player.startPlaying(track.getFilename());
        System.out.println("Now playing: " + track.getArtist() + " - " + track.getTitle());
    }
}
```

```
    /**
     * Show a list of all the tracks in the collection.
     */
    public void listAllTracks()
    {
        System.out.println("Track listing: ");


        for(Track track : tracks) {
            System.out.println(track.getDetails());
        }
        System.out.println();
    }
}
```

Embora possamos ver que a introdução de uma classe *Track* tornou alguns dos métodos antigos um pouco mais complicados, trabalhar com objetos *Track* especializados resulta em uma estrutura muito melhor para o programa como um todo e nos permite desenvolver a classe *Track* da melhor maneira possível o nível de detalhe apropriado para representar mais do que apenas nomes de arquivos de áudio. A informação não é apenas melhor estruturada; o uso de uma classe *Track* também nos permite armazenar informações mais ricas, se quisermos, como uma imagem da capa de um álbum.

Com uma melhor estruturação das informações da faixa, agora podemos fazer um trabalho muito melhor na busca de faixas que atendam a critérios específicos. Por exemplo, se quisermos encontrar todas as faixas que contenham a palavra "amor" em seu título, podemos fazê-lo da seguinte maneira, sem correr o risco de incompatibilidades nos nomes dos artistas:

```
/ **
* Liste todas as faixas que contêm a sequência de pesquisa especificada em seu título.

* @param searchString A string de pesquisa a ser encontrada.
* /
public void findInTitle(String searchString)
{
  for(Track track : tracks) {
    String title = track.getTitle();
    if(title.contains(searchString)) {
      System.out.println(track.getDetails());
    }
  }
}
```

4.12 O tipo *Iterator*

A iteração é uma ferramenta vital em quase todos os projetos de programação, portanto, não surpreende descobrir que as linguagens de programação geralmente oferecem uma ampla gama de recursos que a suportam, cada uma com suas próprias características específicas, adequadas para diferentes situações.

Agora discutiremos uma terceira variação de como iterar sobre uma coleção que está um pouco no meio entre o *loop while* e o *loop for-each*. Ele usa um *loop while* para executar a iteração, e um objeto "iterador" em vez de uma variável de índice inteiro para acompanhar a posição na lista. Temos que ter muito cuidado com a nomeação neste momento, porque o "Iterator" (observe o I maiúsculo) é uma classe Java, mas também encontraremos um método chamado "iterador" (i minúsculo), portanto, preste muita atenção a essas diferenças quando lendo esta seção e ao escrever seu próprio código.

Examinar todos os itens de uma coleção é tão comum que já vimos uma estrutura de controle especial - o *loop for-each* - feito sob medida para esse fim. Além disso, as várias classes de bibliotecas de coleções do Java fornecem um tipo comum personalizado para oferecer suporte à iteração, e o *ArrayList* é típico a esse respeito.

O método "iterador" de *ArrayList* retorna um objeto "Iterator". O "iterador" também é definido no pacote *java.util*, portanto, devemos adicionar uma segunda instrução de importação ao arquivo de classe para usá-lo:

```
import java.util.ArrayList;

import java.util.Iterator;
```

Um "iterador" fornece apenas quatro métodos, e dois deles são usados para iterar sobre uma coleção: *hasNext* e *next*. Nenhum deles aceita um parâmetro, mas ambos têm tipos de retorno não nulos; portanto, eles são usados em expressões. A maneira como costumamos usar um "iterador" pode ser descrita no pseudocódigo da seguinte maneira:

```
Iterator<ElementType> it = myCollection.iterator();

while(it.hasNext ()) {
```

chame it.next() para obter o próximo elemento

faça algo com esse elemento

}

Nesse fragmento de código, primeiro usamos o método “iterador” da classe *ArrayList* para obter um objeto “Iterator”. Observe que o “iterador” também é um tipo genérico; portanto, o parametrizamos com o tipo de elementos na coleção sobre a qual estamos iterando. Em seguida, usamos esse “iterador” para verificar repetidamente se existem mais elementos, *it.hasNext* (), e para obter o próximo elemento, *it.next* (). Um ponto importante a ser observado é que é o objeto Iterator que solicitamos que retorne o próximo item, e não o objeto de coleção. De fato, tendemos a não nos referir diretamente à coleção no corpo do *loop*; toda a interação com a coleção é feita via “Iterator”.

Usando um “iterador”, podemos escrever um método para listar as faixas, conforme mostrado no Código 4.8. De fato, o “Iterator” começa no início da coleção e avança progressivamente, um objeto de cada vez, assim que chamamos seu próximo método.

Código 4.8

```
/**
 * Show a list of all the tracks in the collection.
 */
public void listAllTracks()
{
  Iterator<Track> it = tracks.iterator();
  while(it.hasNext()) {
    Track t = it.next();
    System.out.println(t.getDetails());
  }
}
```

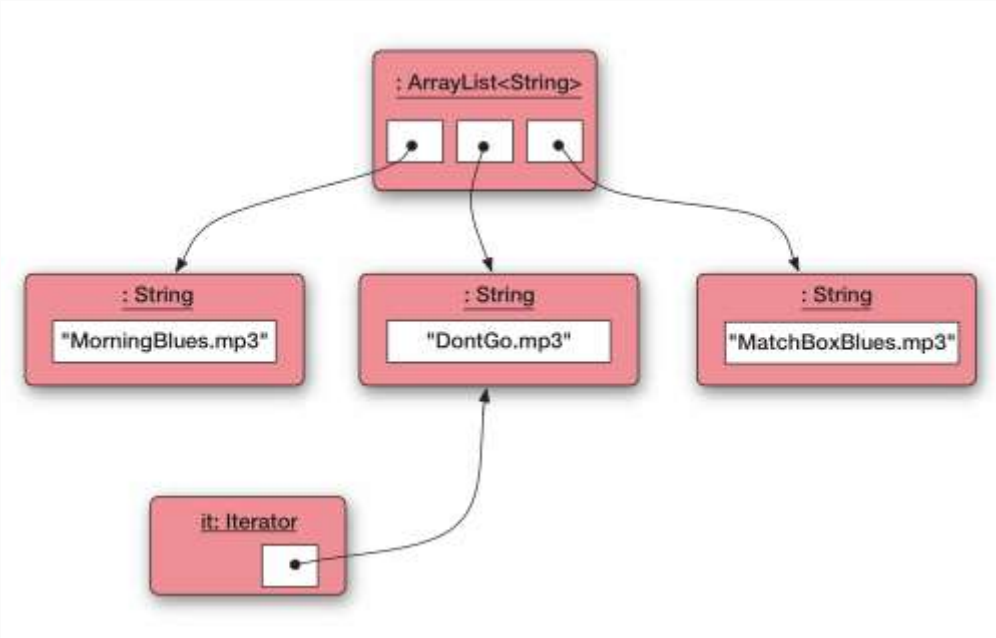
Reserve um tempo para comparar esta versão com a que usa um *loop for-each* no Código 4.7 e as duas versões de *listAllFiles* mostradas no Código 4.3, e no 4.5 do código. Um ponto específico a ser observado sobre a versão mais recente é que usamos um *loop while*, mas não precisamos cuidar da variável *index*. Isso ocorre porque o “Iterator” controla até que ponto ele progrediu através da coleção, para que ele saiba se há mais itens restantes (*hasNext*) e qual retornar (próximo) se houver outro.

Uma das chaves para entender como um “Iterator” funciona é que a chamada para o próximo faz com que o “Iterator” retorne o próximo item da coleção e depois passe esse item. Portanto, chamadas sucessivas para o próximo em um “iterador” sempre retornarão itens distintos; você não pode voltar ao item anterior depois que o próximo for chamado. Eventualmente, o “iterador” chega ao final da coleção e retorna falso de uma chamada para *hasNext*. Depois que *hasNext* retornar “*false*”, seria um erro tentar chamar a seguir esse objeto “Iterator” específico - com efeito, o objeto “Iterator” foi "usado" e não tem mais uso. Diante disso, ele parece não oferecer vantagens óbvias em relação às maneiras anteriores que vimos para iterar sobre uma coleção, mas as duas seções a seguir fornecem razões pelas quais é importante saber usá-lo.

4.12.1 Acesso ao índice versus “iteradores”

Vimos que temos pelo menos três maneiras diferentes pelas quais podemos iterar sobre um *ArrayList*. Podemos usar um *loop for-each* (como visto na Seção 4.9.1), o método *get* com uma variável de índice inteiro (Seção 4.10.2) ou um objeto “Iterator” (esta seção).

Figura 5 – Iterar sobre um *ArrayList*



Fonte: BlueJ.

Pelo que sabemos até agora, todas as abordagens parecem iguais em qualidade. O primeiro foi talvez um pouco mais fácil de entender, porém o menos flexível.

A primeira abordagem, usando o *loop for-each*, é a técnica padrão usada para processar todos os elementos de uma coleção (ou seja, iteração definida), porque é a mais concisa para esse caso. As duas últimas versões têm o benefício de que a iteração pode ser mais facilmente interrompida no meio do processamento (iteração indefinida); portanto, elas são preferíveis ao processar apenas uma parte da coleção.

Para um *ArrayList*, os dois últimos métodos (usando os *loops while*) são de fato igualmente bons. Este nem sempre é o caso, porém. Java fornece muito mais classes de coleção além do *ArrayList*. Veremos vários deles nos próximos capítulos. Para algumas coleções, é impossível ou muito ineficiente acessar elementos individuais fornecendo um índice. Portanto, nossa primeira versão do *loop while* é uma solução específica para a coleção *ArrayList* e pode não funcionar para outros tipos de coleções.

A solução mais recente, usando um “Iterator”, está disponível para todas as coleções na biblioteca de classes Java e, portanto, é um padrão de código importante que usaremos novamente em projetos posteriores.

4.12.2 Removendo elementos

Outra consideração importante ao escolher a estrutura de *loop* a ser usada ocorre quando precisamos remover elementos da coleção durante a iteração. Um exemplo, pode ser quando queremos remover todas as faixas de nossa coleção pertencentes a um artista em que não estamos mais interessados.

Podemos escrever isso facilmente no pseudocódigo:

```
para cada faixa na coleção {
  se track.getArtist() é o artista desfavorecido:
  collection.remove(faixa)
}
```

Acontece que essa operação perfeitamente razoável não é possível com um *loop for-each*. Se tentarmos modificar a coleção usando um dos métodos de remoção da coleção enquanto estiver no meio de uma iteração, o sistema reportará um erro (chamado *ConcurrentModificationException*). Isso acontece porque alterar a coleção no meio de uma iteração tem o potencial de confundir completamente a situação. E se o elemento removido fosse o que estávamos trabalhando atualmente? Se já foi removido, como devemos encontrar o próximo elemento? Geralmente, não há boas respostas para esses problemas em potencial; portanto, o uso do método de remoção da coleção simplesmente não é permitido durante uma iteração com o *loop for-each*. A solução adequada para remover durante a iteração é usar um "iterador". Seu terceiro método (além de *hasNext* e *next*) é removido. Não requer parâmetro e possui um tipo de retorno nulo. Chamar "remover" removerá o item retornado pela chamada mais recente para a próxima. Aqui está um código de exemplo:

```
Iterator<Track> it = tracks.iterator();
while(it.hasNext()) {
  Track t = it.next();
  String artist = t.getArtist();
  if(artist.equals(artistToRemove)) {
    it.remove();
  }
}
```

Mais uma vez, observe que não usamos a variável de coleção de trilhas no corpo do *loop*. Embora o *ArrayList* e o "Iterator" tenham métodos de remoção, devemos usar o método de remoção do "Iterator", e não o *ArrayList*.

Usar a remoção do "iterador" é menos flexível - não podemos remover elementos arbitrários. Podemos remover apenas o último elemento que recuperamos do próximo método do "iterador". Por outro lado, o uso da remoção do "iterador" é permitido durante uma iteração. Como o próprio "Iterator" é informado sobre a remoção (e faz isso por nós), ele pode manter a iteração corretamente sincronizada com a coleção.

Essa remoção não é possível com o *loop for-each*, porque não temos um "Iterator" para trabalhar. Nesse caso, precisamos usar o *loop while* com um "iterador". Tecnicamente, também

podemos remover elementos usando o método *get* da coleção com um índice para a iteração. Isso não é recomendado, no entanto, porque os índices do elemento podem mudar quando adicionamos ou excluímos elementos, e é muito fácil obter a iteração com índices incorretos quando modificamos a coleção durante a iteração. O uso de um "iterador" nos protege de tais erros.

Resumo do projeto organizador de música

No organizador de músicas, vimos como podemos usar um objeto *ArrayList*, criado a partir de uma classe da biblioteca, para armazenar um número arbitrário de objetos em uma coleção. Não precisamos decidir com antecedência quantos objetos queremos armazenar, e o objeto *ArrayList* controla automaticamente o número de itens armazenados nele. Discutimos como podemos usar um loop para iterar sobre todos os elementos da coleção. Java tem várias construções de *loop* - as duas que usamos aqui são o *loop for-each* e *while*. Normalmente, usamos um *loop for-each* quando queremos processar toda a coleção e o *loop while* quando não podemos predeter quantas iterações precisamos ou quando precisamos remover elementos durante a iteração.

Com um *ArrayList*, podemos acessar elementos por índice ou iterar sobre todos os elementos usando um objeto "Iterator". Vale a pena revisar as diferentes circunstâncias sob as quais os diversos tipos de *loop* (para cada um e enquanto) são apropriados e por que um "Iterator" deve ser preferido em relação a um índice inteiro, por que esses tipos de decisões terão que ser tomados repetidamente. Acertá-los pode facilitar muito a solução de um problema específico.

Quando você deseja adicionar um novo objeto de Associação ao objeto Clube, a partir do banco de objetos, há duas maneiras de fazer isso. Crie um novo objeto *Membership* no banco de objetos, chame o método *join* no objeto Clube e clique no objeto *Membership* para fornecer o parâmetro ou chame o método *join* no objeto Clube e digite na caixa de diálogo do método: new Membership ("nome do membro...", mês, ano) Cada vez que você adicionar um, use o método *numberOfMembers* para verificar se o método *join* está adicionando à coleção e se o método *numberOfMembers* está fornecendo o resultado correto.

O pacote *java.util* contém a classe *Random* cujo método *nextInt* irá gerar um número inteiro aleatório positivo dentro de um intervalo limitado. Escreva um método na classe *MusicOrganizer* para selecionar uma única faixa aleatória da lista e reproduzi-la. Dica: Você precisará importar o *Random* e criar um objeto *Random*, diretamente no novo método ou no construtor e armazenado em um campo. Você precisará encontrar a documentação da "API" para a classe *Random* e verificar seus métodos para escolher a versão correta do *nextInt.*

Dica: Uma maneira de fazer isso seria embaralhar a ordem das faixas da lista - ou, talvez melhor, uma cópia da lista - e depois reproduzir do começo ao fim. Outra maneira seria fazer uma cópia da lista e escolher repetidamente uma faixa aleatória da lista, reproduzi-la e removê-la da lista até que ela esteja vazia. Tente implementar uma dessas abordagens. Se você tentar o primeiro, quão fácil é embaralhar a lista para que ela fique genuinamente em uma nova ordem aleatória? Existem métodos de biblioteca que poderiam ajudar com isso?

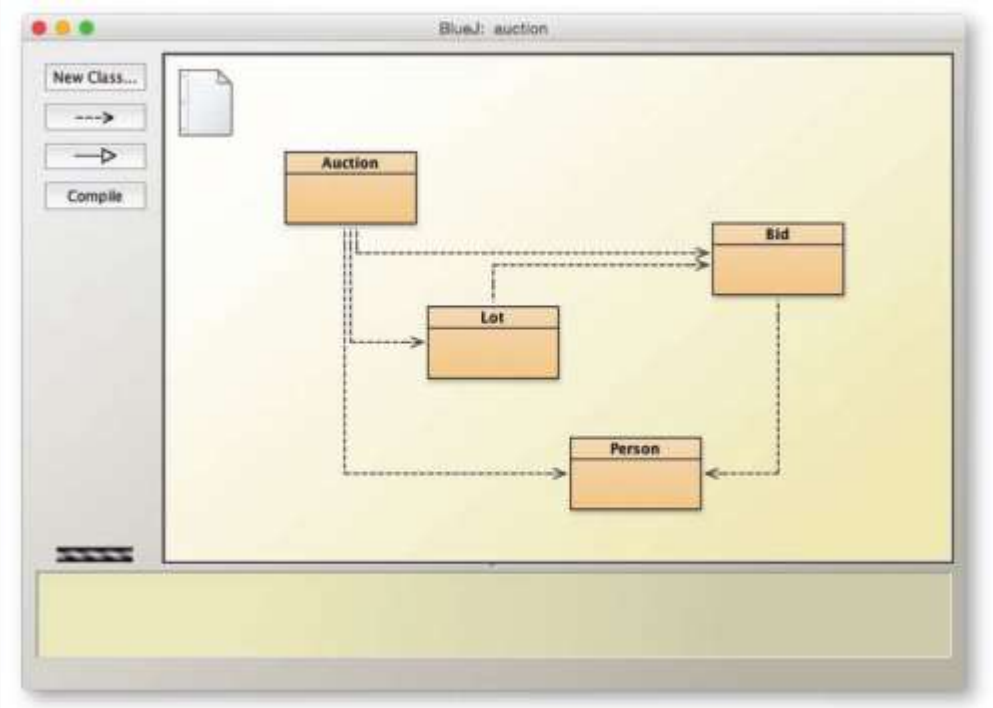
4.14 Outro exemplo: um sistema de leilão

Nesta seção, acompanharemos algumas das novas ideias que introduzimos nesta aula, analisando-as novamente em um contexto diferente.

O projeto de leilão modela parte da operação de um sistema de leilão online. A ideia é que um leilão consista em um conjunto de itens oferecidos para venda. Esses itens são chamados de "lotes" e cada um recebe um número de lote exclusivo pelo programa. Uma pessoa pode tentar comprar muito, oferecendo uma quantia em dinheiro por isso. Nossos leilões são ligeiramente diferentes de outros leilões, porque o nosso oferece todos os lotes por um período limitado de tempo. No final desse período, o leilão é encerrado. No fim do leilão, considera-se que a pessoa que fez o lance mais alto por um lote e o comprou. Os lotes para os quais não houver ofertas permanecem não vendidos no fechamento. Lotes não vendidos podem ser oferecidos em um leilão posterior, por exemplo.

O projeto de leilão contém as seguintes classes: Leilão, Lance, Lote e Pessoa. Uma análise mais detalhada do diagrama de classes para este projeto, (Figura 4.6) revela que os relacionamentos entre as várias classes são um pouco mais complicados do que vimos em projetos anteriores. Isso influenciará na maneira como as informações são acessadas durante as atividades de leilão. Por exemplo, o diagrama mostra que os objetos de leilão conhecem todos os outros tipos de objetos: lance, lote e pessoa. Os objetos de lote conhecem os objetos de lance e os objetos de lance conhecem os objetos de pessoa. O que o diagrama não pode nos dizer é exatamente como as informações armazenadas em um objeto Bid, por exemplo, são acessadas por um objeto *Auction*. Para isso, temos que olhar o código das várias classes.

Figura 6 – Código com várias classes



Fonte: BlueJ.

4.14.1 Introdução ao projeto

Nesse estágio, valeria a pena abrir o projeto de leilão e explorar o código fonte antes de ler mais. Além de ver o uso familiar de um *ArrayList* e *loops*, você provavelmente encontrará várias coisas que não entende bem a princípio, mas isso é de se esperar à medida que avançamos para novas ideias e novas maneiras de fazer as coisas. Um objeto *Auction* é o ponto de partida do projeto. As pessoas que desejam vender itens entram no leilão por meio do método *enterLot*, mas fornecem apenas uma descrição da *string*. O objeto "Leilão" cria um objeto "Lote" para cada lote inserido. Isso modela a maneira como as coisas funcionam no mundo real, como neste caso, é o site de leilão, e não os vendedores, por exemplo, que atribui números de lote ou códigos de identificação aos itens. Portanto, um objeto "Lote" é a representação do site de leilão de um item para venda.

Para licitar lotes, as pessoas devem se registrar na casa de leilões. Em nosso programa, um licitante em potencial é representado por um objeto "Pessoa". Esses objetos devem ser criados independentemente no banco de objetos do BlueJ. Em nosso projeto, um objeto "Pessoa" simplesmente contém o nome da pessoa. Quando alguém quer fazer um lance por muito, ele chama o método *makeABid* do objeto *Auction*, inserindo o número do lote em que está interessado, o objeto da pessoa do licitante e o quanto ele deseja fazer um lance. Observe que eles passam o número do lote, ao invés do objeto "Lot"; os objetos de lote permanecem internos ao objeto de leilão e são sempre referenciados externamente por seu número de lote.

Assim como o objeto Leilão cria objetos "Lote", ele também transforma um valor de lance monetário em um objeto "Lance", que registra o valor e a pessoa que licitou esse valor. É por isso que vemos um *link* da classe "*Bid*" para a classe "*Person*" no diagrama de classes. No entanto, observe que não há *link* entre "Lance" e "Lote"; *o link* no diagrama é o contrário, porque um lote registra o que atualmente é o lance mais alto para esse lote. Isso significa que o objeto "Lote" substituirá o objeto "Lance" armazenado toda vez que um lance mais alto for feito. O que descrevemos aqui revela uma cadeia bastante aninhada de referências a objetos. Objetos de leilão armazenam objetos de lote; cada objeto "Lot" pode armazenar um objeto "Bid"; cada objeto "Bid" oferece um objeto "*Person*". Como essas cadeias são muito comuns nos programas, esse projeto oferece uma boa oportunidade para explorar como elas funcionam na prática.

4.14.2 A palavra-chave nula

A partir da discussão acima, deve ficar claro que um objeto de "Lance" é criado apenas quando alguém realmente faz um lance para um "Lote". O objeto de lance, recém-criado, armazena a pessoa que faz o lance. Isso significa que o campo "Pessoa" de cada objeto de "Lance" pode ser inicializado no construtor de Lances, e o campo sempre conterá um objeto de "Pessoa" válido.

Por outro lado, quando um objeto "Lote" é criado, isso significa simplesmente que ele foi inserido no leilão e ainda não possui licitantes. No entanto, ele ainda possui um campo Lance, para registrar o lance mais alto para o lote. Qual valor deve ser usado para inicializar esse campo no construtor "Lot"?

O que precisamos é de um valor para o campo que deixe claro que atualmente "nenhum objeto" está sendo referido por essa variável. Em certo sentido, a variável está "vazia". Para indicar isso, Java fornece a palavra-chave nula. Portanto, o construtor de "Lot" possui a seguinte instrução: maximumBid = null;

Um princípio muito importante é que, se uma variável contiver o valor nulo, uma chamada de método não deverá ser feita nela. A razão para isso deve ser clara: como os métodos pertencem a objetos, não podemos chamar um método se a variável não se referir a um objeto. Isso significa que às vezes precisamos usar uma instrução *if* para testar se uma variável contém nulo ou não, antes de chamar um método nessa variável. A falha em fazer esse teste levará ao erro de tempo de execução muito comum, chamado *NullPointerException*. Você verá alguns exemplos desse teste nas classes "Lot" e "*Auction*".

De fato, se não conseguirmos inicializar um campo do tipo de objeto, ele receberá automaticamente o valor nulo. Nesse caso em particular, no entanto, preferimos fazer a atribuição explicitamente para que não haja dúvidas na mente do leitor do código, que esperamos que o *maximumBid* seja nulo quando um objeto Lot for criado.

4.14.3 A classe "Lot"

A classe "Lot" armazena uma descrição do lote, um número de lote e detalhes da oferta mais alta recebida até o momento. A parte mais complexa da classe é o método "*bidFor*" (código 4.9). Isso lida com o que acontece quando uma pessoa faz uma oferta pelo lote. Quando um lance é feito, é necessário verificar se o novo lance tem um valor mais alto do que qualquer lance existente nesse lote. Se for maior, o novo lance será armazenado como o lance mais alto atual no lote. Aqui, primeiro verificamos se esse lance é o lance mais alto. Esse será o caso se não houver lance anterior ou se o lance atual for maior que o melhor lance até o momento. A primeira parte da verificação envolve o seguinte teste:

maximumBid == null

Código 4.9

```
public class Lot
{
    // The current highest bid for this lot.
    private Bid highestBid;


    /**
     * Attempt to bid for this lot. A successful bid
     * must have a value higher than any existing bid.
     * @param bid A new bid.
     * @return true if successful, false otherwise
     */
    public boolean bidFor(Bid bid)
    {
        if(highestBid == null) {
            // There is no previous bid.
            highestBid = bid;
            return true;
        }
        else if(bid.getValue() > highestBid.getValue()) {
            // The bid is better than the previous one.
            highestBid = bid;
            return true;
        }
        else {
            // The bid is not better.
            return false;
        }
    }

    ... Outros métodos omitidos

}
```

Este é um teste para saber se a variável mais alta está atualmente se referindo a um objeto ou não. Conforme descrito na seção anterior, até que um lance seja recebido para esse lote, o campo *maximumBid* conterá o valor nulo. Se ainda for nulo, essa é a primeira oferta para esse lote específico e deve claramente ser a mais alta. Se não for nulo, temos que comparar seu valor com o novo lance. Observe que a falha do primeiro teste nos fornece algumas informações muito úteis: agora sabemos com certeza que o *maximumBid* não é nulo; portanto, sabemos que é seguro chamar um método nele. Não precisamos fazer um teste nulo novamente nesta segunda condição. A comparação dos valores dos dois lances nos permite escolher um novo lance mais alto ou rejeitar o novo lance, se não for melhor.

4.14.4 A classe *Auction*

A classe *Auction* (Código 4.10) fornece uma ilustração mais detalhada dos conceitos *ArrayList* e de *loop* para cada um que discutimos anteriormente neste capítulo.

Código 4.10

```
import java.util.ArrayList;

/**
* A simple model of an auction.
* The auction maintains a list of lots of arbitrary length.
*
* @author David J. Barnes and Michael Kölling
* @version 2016.02.29
*/
public class Auction
{
  // The list of Lots in this auction.
  private ArrayList<Lot> lots;
  // The number that will be given to the next lot entered
  // into this auction.
  private int nextLotNumber;

  /**
   * Create a new auction.
   */
  public Auction()
  {
     lots = new ArrayList<>();
     nextLotNumber = 1;
  }

  /**
   * Enter a new lot into the auction.
   * @param description A description of the lot.
   */
  public void enterLot(String description)
  {
     lots.add(new Lot(nextLotNumber, description));
     nextLotNumber++;
  }
```

```
    /**
     * Show the full list of lots in this auction.
     */
    public void showLots()
    {
        for(Lot lot : lots) {
            System.out.println(lot.toString());
        }
    }

    /**
     * Make a bid for a lot.
```

```
     * A message is printed indicating whether the bid is
     * successful or not.
     *
     * @param lotNumber The lot being bid for.
     * @param bidder The person bidding for the lot.
     * @param value  The value of the bid.
     */
    public void makeABid(int lotNumber, Person bidder, long value)
    {
        Lot selectedLot = getLot(lotNumber);
        if(selectedLot != null) {
            Bid bid = new Bid(bidder, value);
            boolean successful = selectedLot.bidFor(bid);
            if(successful) {
                System.out.println("The bid for lot number " +
                                   lotNumber + " was successful.");
            }
            else {
                // Report which bid is higher.
                Bid highestBid = selectedLot.getHighestBid();
                System.out.println("Lot number: " + lotNumber +
                                   " already has a bid of: " +
                                   highestBid.getValue());
            }
        }
    }
```

```
    /**
     * Return the lot with the given number. Return null
     * if a lot with this number does not exist.
     * @param lotNumber The number of the lot to return.
     */
    public Lot getLot(int lotNumber)
    {
        if((lotNumber >= 1) && (lotNumber < nextLotNumber)) {
            // The number seems to be reasonable.
            Lot selectedLot = lots.get(lotNumber - 1);
            // Include a confidence check to be sure we have the
            // right lot.
            if(selectedLot.getNumber() != lotNumber) {
                System.out.println("Internal error: Lot number " +
                                   selectedLot.getNumber() +
                                   " was returned instead of " +
                                   lotNumber);
                // Don't return an invalid lot.
                selectedLot = null;
            }
            return selectedLot;
        }
        else {
            System.out.println("Lot number: " + lotNumber +
                               " does not exist.");
            return null;
        }
    }
```

4.14 Outro exemplo: um sistema de leilão

O campo de lotes é um *ArrayList* usado para armazenar os lotes oferecidos neste leilão. Os lotes são inseridos no leilão, passando uma descrição simples para o método *enterLot*. Um novo lote é criado passando a descrição e um número de lote exclusivo para seu construtor. O novo objeto "Lote" é adicionado à coleção. As seções a seguir, discutem alguns recursos adicionais, comumente encontrados, ilustrados na classe *Auction*.

4.14.5 Objetos anônimos

O método *enterLot* no *Auction* ilustra um idioma comum - objetos anônimos. Vemos isso na seguinte declaração:

lots.add(new Lot(nextLotNumber, description));

Aqui, estamos fazendo duas coisas:

Estamos criando um novo objeto "Lot".

Também estamos passando esse novo objeto para o método "*add*" do *ArrayList*.

Poderíamos ter escrito a mesma declaração em duas linhas, para tornar as etapas separadas mais explícitas:

Lot newLot = new Lot(nextLotNumber, description);

lots.add(newLot);

Ambas as versões são equivalentes, mas se não houver mais uso para a variável *newLot*, a versão original evita definir uma variável com um uso tão limitado. Com efeito, criamos um objeto anônimo - um objeto sem nome - passando diretamente para o método que o utiliza.

4.14.6 Chamadas do método de encadeamento

Em nossa introdução ao projeto de leilão, observamos uma cadeia de referências a objetos: Os objetos de leilão armazenam objetos de lote; cada objeto "Lot" pode armazenar um objeto "Bid"; cada objeto "Bid" oferece um objeto "Person". Se o objeto Leilão precisar identificar quem atualmente tem o lance mais alto em um lote, será necessário solicitar ao Lote que devolva o objeto Lance para esse lote. Em seguida, peça ao objeto "Lance" a pessoa que fez o lance e, em seguida, peça o objeto "Pessoa", seu nome. Ignorando a possibilidade de referências a objetos nulos, podemos ver algo como a seguinte sequência de instruções para imprimir o nome de um licitante:

Bid bid = lot.getHighestBid();

Person bidder = bid.getBidder();

String name = bidder.getName();

System.out.println(name);

Como as variáveis de lance, licitante e nome estão sendo usadas aqui simplesmente como postagens temporárias para chegar ao nome do licitante, é comum ver sequências, como essa,

compactadas pelo uso de referências de objetos anônimos. Por exemplo, podemos obter o mesmo efeito com o seguinte:

System.out.println(lot.getHighestBid().getBidder().getName());

Parece que os métodos estão chamando métodos, mas não é assim que isso deve ser lido. Tendo em mente que os dois conjuntos de instruções são equivalentes, a cadeia de chamadas de método deve ser lida estritamente da esquerda para a direita:

lot.getHighestBid().getBidder().getName()

A chamada para *getHighestBid* retorna um objeto de lance anônimo e o método *getBidder* é chamado nesse objeto. Da mesma forma, *getBidder* retorna um objeto "Pessoa" anônimo, portanto, *getName* é chamado nessa pessoa.

Tais cadeias de chamadas de método podem parecer complicadas, mas podem ser desmarcadas se você entender as regras subjacentes. Mesmo que você opte por não escrever seu código dessa maneira mais concisa, você deve aprender a lê-lo, pois poderá encontrá-lo no código de outros programadores.

4.14.7 Usando coleções

A classe *ArrayList* (e outras similares) é uma importante ferramenta de programação, porque muitos problemas de programação envolvem o trabalho com coleções de objetos de tamanho variável. Antes de passar para os próximos capítulos, é importante que você se familiarize e se sinta confortável em como trabalhar com eles.

TERMOS INTRODUZIDOS NESTA AULA

Coleção, "iterador", para cada *loop*, enquanto *loop*, índice, instrução de importação, biblioteca, pacote, objeto anônimo, iteração definida e iteração indefinida.

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 - Conceitos Avançados

Esta aula apresenta alguns conceitos avançados. As seções avançadas deste material (haverá outras posteriormente) podem ser puladas na primeira leitura, se você desejar. Aqui, discutiremos algumas técnicas alternativas para realizar as mesmas tarefas discutidas nas aulas anteriores. Essas técnicas não estavam disponíveis na linguagem Java anterior à versão 8.

5.1 Uma visão alternativa dos temas da aula 01

Na aula 01, desta unidade, começamos a examinar técnicas para processar coleções de objetos. Manipular coleções é uma das técnicas fundamentais em programação; quase todos os programas de computador processam coleções de algum tipo, e veremos muito mais disso ao longo deste material. Ser capaz de fazer isso bem é essencial.

No entanto, o aprendizado sobre coleções em programação tornou-se mais complicado nos últimos anos com a introdução de novas técnicas de programação em nas principais linguagens. Essas técnicas podem tornar seu programa mais simples, mas acarretam o aprendizado mais trabalhoso - há simplesmente mais construções diferentes para dominar.

As "novas" técnicas não são realmente novas - elas são apenas novas neste tipo particular de linguagem. Sempre houve diferentes tipos de linguagens: Java, por exemplo, é uma linguagem imperativa orientada a objetos, e o estilo de processamento de coleção que começamos a introduzir na aula anterior 4 é a maneira imperativa típica de fazer as coisas.

Outra classe de linguagem são as linguagens funcionais; elas existem há muito tempo e usam um estilo diferente de programação. Acontece que, por razões que discutiremos mais adiante, que o estilo funcional de processar coleções tem vantagens em algumas situações em relação ao estilo imperativo. Como resultado, os designers da linguagem Java decidiram em algum momento adicionar alguns elementos de programação funcional à linguagem Java. Eles fizeram isso na versão 8 da linguagem (Java 8, lançado em 2014). As novas técnicas adicionadas são fluxos e lambdas (às vezes também chamadas de fechamentos).

Isso faz do Java tecnicamente uma linguagem híbrida: uma linguagem principalmente imperativa com alguns elementos funcionais.

Para programadores proficientes, isso é ótimo. As novas construções de linguagem funcional nos permitem escrever programas mais elegantes e expressivos. Para os alunos, isso cria mais trabalho: agora temos que estudar duas maneiras diferentes de alcançar um objetivo semelhante.

O novo estilo funcional, que abordamos neste capítulo, está rapidamente se tornando popular e provavelmente será o estilo dominante de escrever novo código em Java dentro de um curto espaço de tempo. O estilo imperativo abordado no capítulo anterior, no entanto, ainda é a maneira como a maioria dos códigos é escrita atualmente e ainda é considerada a maneira Java "padrão" de fazer as coisas. Quando você trabalha com código existente escrito por outros programadores (como na maioria das vezes), é essencial que você entenda e possa trabalhar com o estilo imperativo. Ao escrever seu próprio código, você pode escolher seu estilo: os estilos funcionais e imperativos são duas maneiras diferentes de resolver o mesmo problema e uma

pode substituir a outra. No futuro, muitas vezes preferimos o estilo funcional, porque é mais conciso e mais elegante.

Em resumo, para que um aluno se torne um bom programador, é necessário familiarizar-se com os dois estilos de processamento de coleções. Neste material, apresentamos o estilo imperativo primeiro e introduzimos o estilo funcional nas seções "avançadas" (como este capítulo). Essas seções avançadas podem ser ignoradas (agora você pode pular para a próxima aula, se quiser) e ainda aprenderá como programar e ainda poderá resolver os problemas que apresentamos. Eles também podem ser lidos fora de sequência (pule-os agora, mas volte mais tarde) se você tiver pouco tempo e quiser progredir com outras construções primeiro. Para uma imagem completa, eles podem ser lidos em sequência, onde estão, e você aprenderá as construções alternativas à medida que avança.

5.2 Monitorando populações de animais

Como antes, usaremos um programa de exemplo para introduzir e discutir as novas construções. Aqui, veremos um sistema para monitorar populações de animais.

Um elemento importante da conservação animal e da proteção de espécies ameaçadas é a capacidade de monitorar o número da população e detectar quando os níveis estão saudáveis ou em declínio. Nosso projeto processa relatórios de avistamentos de diferentes tipos de animais, devolvidos por observadores de vários lugares diferentes. Cada relatório de avistamento consiste no nome do animal que foi visto, quantos foram vistos, quem está enviando o relatório (um número inteiro), em que área o avistamento foi realizado (um número inteiro) e uma indicação de quando foi feita a observação (um valor numérico simples que pode ser o número de dias desde o início do experimento). Os requisitos simples de dados facilitam para os observadores em locais remotos enviarem quantidades relativamente pequenas de informações valiosas - talvez na forma de uma mensagem de texto - para uma base que é capaz de agregar os dados e criar relatórios ou direcionar os trabalhadores de campo para diferentes áreas.

Essa abordagem se encaixa bem nos dois projetos altamente gerenciados - como os que podem ser realizados em um parque nacional e envolvem câmeras acionadas por movimento e pessoas avaliando as filmagens - e com atividades pouco organizadas de fornecimento de multidões - como dias nacionais de contagem de pássaros, onde um grande grupo de voluntários usa aplicativos de telefone para enviar dados.

Fornecemos uma implementação parcial desse sistema sob o nome *animal-Monitoring-v1* nos projetos, incluso nas referências.

O Código 5.1, mostra a classe de “Avistamento” que usaremos para registrar detalhes de cada relatório de avistamento, de um único observador para um animal em particular, depois que os detalhes deste forem processados. Neste capítulo, não nos preocuparemos diretamente com o formato dos dados enviados pelo observador. De acordo com o tema, esta aula explora conceitos Java mais avançados; você deve ler o código fonte completo do projeto para encontrar coisas que ainda não exploramos em detalhes, e usá-las para seu próprio desenvolvimento pessoal. A classe *Sighting*, no entanto, é muito direta.

Código 5.1

```
/**
* Details of a sighting of a type of animal by an individual spotter.
*
* @author David J. Barnes and Michael Kölling
* @version 2016.02.29
*/
public class Sighting
{
   // The animal spotted.
   private final String animal;
   // The ID of the spotter.
   private final int spotter;
   // How many were seen.
   private final int count;
   // The ID of the area in which they were seen.
   private final int area;
   // The reporting period.
   private final int period;

   /**
    * Create a record of a sighting of a particular type of animal.
    * @param animal The animal spotted.
    * @param spotter The ID of the spotter.
    * @param count How many were seen (>= 0).
    * @param area The ID of the area in which they were seen.
    * @param period The reporting period.
    */
   public Sighting(String animal, int spotter, int count, int area, int period)
   {
      this.animal = animal;
      this.spotter = spotter;
      this.count = count;
      this.area = area;
      this.period = period;
   }

   // Alguns métodos omitidos
```

```
   /**
    * Return a string containing details of the animal, the number seen,
    * where they were seen, who spotted them and when.
    * @return A string giving details of the sighting.
    */
   public String getDetails()
   {
      return animal +
            ", count = " + count +
            ", area = " + area +
            ", spotter = " + spotter +
            ", period = " + period;
   }

}
```

O código 5.2, mostra parte da classe *AnimalMonitor* usada para agregar os avistamentos individuais em uma lista. Neste ponto, todos os avistamentos de todos os diferentes observadores e áreas são mantidos juntos em uma única coleção. Entre outras coisas, a classe contém métodos para imprimir a lista, contar o número total de avistamentos de um determinado animal, listar todos os avistamentos de um observador específico, remover registros que não contêm avistamentos e assim por diante. Todos esses métodos foram implementados usando as técnicas descritas na aula anterior para manipulação básica de lista: iteração na lista completa; processar elementos selecionados da lista com base em alguma condição (filtragem); e remoção de elementos. Os métodos implementados aqui não estão completos para um aplicativo desse tipo, mas foram implementados para mostrar exemplos desses tipos diferentes de processamento de lista.

Código 5.2

```
import java.util.ArrayList;
import java.util.Iterator;

/**
* Monitor counts of different types of animal.
* Sightings are recorded by spotters.
*
* @author David J. Barnes and Michael Kölling
* @version 2016.02.29 (imperative)
*/
public class AnimalMonitor
{
  // Records of all the sightings of animals.
  private ArrayList<Sighting> sightings;

  /**
   * Create an AnimalMonitor.
   */
  public AnimalMonitor()
  {
     this.sightings = new ArrayList<>();
  }

  /**
   * Add the sightings recorded in the given filename to the current list.
   * @param filename A CSV file of Sighting records.
   */
  public void addSightings(String filename)
  {
     SightingReader reader = new SightingReader();
     sightings.addAll(reader.getSightings(filename));
  }

  /**
   * Print details of all the sightings.
   */
  public void printList()
  {
     for(Sighting record : sightings) {
        System.out.println(record.getDetails());
     }
  }
```

```
    // Outros métodos omitidos
}
```

Até agora, esse código usa as técnicas de processamento de coleção imperativas apresentadas na aula 01.

Usaremos isso como base para fazer alterações gradualmente, a fim de substituir o código de processamento da lista por construções funcionais.

5.3 Um primeiro olhar sobre as lambdas

Central para o novo estilo funcional de processar uma coleção é o conceito de um lambda. A nova ideia básica é que podemos passar um segmento de código como parâmetro para um método, que pode executar esse trecho de código mais tarde, quando necessário.

Até agora, vimos vários tipos de parâmetros - números inteiros, sequências de caracteres, objetos, mas todos esses eram pedaços de dados, não pedaços de código. A capacidade de passar código como parâmetro era nova no Java 8 e nos permite fazer algumas coisas muito úteis.

5.3.1 Processando uma coleção com uma lambda

Quando estávamos processando nossas coleções, escrevemos um *loop* para recuperar os elementos desta, um por um, e depois fizemos algo para cada elemento. A estrutura do código é sempre semelhante ao pseudocódigo a seguir:

loop (para cada elemento da coleção):

pegue um elemento;

faça algo com o elemento;

end loop

Não importa se usamos um *loop for*, um *loop* para cada um ou um *while*, ou se usamos um “iterador” ou um índice para acessar os elementos - o princípio é o mesmo.

Ao usar um lambda, abordamos o problema de maneira diferente. Passamos um pouco de código para a coleção e dizemos "Faça isso para cada elemento da coleção". A estrutura do código se parece com isso:

collection.doThisForEachElement(algum código);

Podemos ver que o código parece muito mais simples. Mais importante: o *loop* desapareceu completamente. Na verdade, ele não se foi - acabou de ser movido para o método *doThisForEachElement* - mas desapareceu do nosso próprio código; não precisamos mais escrever isso sempre.

Tornamos as coleções mais poderosas: em vez de apenas podermos entregar seus elementos para nós, para que possamos trabalhar com eles, a coleção agora pode trabalhar com os elementos para nós. Isso facilita a nossa vida.

Podemos pensar assim: imagine que você precisa cortar todas as crianças de uma turma escolar. No estilo antigo, você diz ao professor: envie-me o primeiro filho. Então você dá à criança um corte de cabelo. Então você diz: me envie o próximo filho. Outra sessão de corte de cabelo. E assim continua, até que não haja mais filhos. Este é o *loop* de estilo antigo do próximo item/processo.

No novo estilo, faça o seguinte: Escreva instruções sobre como cortar o cabelo e, em seguida, dê ao professor e diga: faça isso para todas as crianças da sua classe. (O professor, aqui, representa a coleção.) De fato, um efeito colateral interessante dessa abordagem é que nem precisamos saber como o professor concluirá a tarefa. Por exemplo, ao invés de cortar os cabelos eles mesmos, eles podem decidir subcontratar a tarefa para uma pessoa separada para cada criança, para que o cabelo de cada criança seja cortado ao mesmo tempo. O professor apenas passava as instruções que você lhes deu.

Sua vida é subitamente muito mais fácil. O professor está fazendo muito do trabalho para você. É exatamente isso que as novas coleções no Java 8 podem fazer por você.

5.3.2 Sintaxe básica de um lambda

Um lambda é, de certa forma, semelhante a um método: é um segmento de código definido para fazer alguma coisa, mas não imediatamente executado. No entanto, diferentemente de um método, ele não pertence a uma classe e não lhe daremos um nome. É útil comparar a sintaxe de um lambda com a de um método que executa uma tarefa semelhante.

Um método para imprimir os detalhes de uma observação pode ser assim:

```
public void printSighting(Sighting record)
{
  System.out.println(record.getDetails());
}
```

O lambda equivalente é assim:

```
(Sighting record) ->
{
  System.out.println(record.getDetails());
}
```

Agora você pode ver facilmente as diferenças e semelhanças. As diferenças são:

Não há palavra-chave pública ou privada - a visibilidade é assumida como pública.

Não há nenhum tipo de retorno especificado. Isso ocorre porque isso pode ser resolvido pelo compilador a partir da declaração final no corpo do lambda. Nesse caso, o método *println* é nulo; portanto, o tipo de retorno do lambda é nulo.

Não há nome dado ao lambda; começa com a lista de parâmetros. Uma seta (->) separa a lista de parâmetros do corpo do lambda - esta é a notação distinta para ele. Você verá que os lambdas são frequentemente usados para tarefas pontuais e relativamente simples, que não exigem a

complexidade de uma classe completa. Algumas das informações sobre um lambda são fornecidas por padrão ou inferidas, a partir do contexto - sua visibilidade e tipo de retorno - e não temos escolha sobre isso. Além disso, como um lambda não possui classe associada, também não há campos de instância, nem construtor. Isso contribui para o uso especializado de lambdas. Por causa da falta de um nome, os lambdas também são conhecidos como funções anônimas.

Iremos encontrar lambdas e parâmetros de função várias vezes, mais adiante neste material, por exemplo, quando introduzirmos interfaces funcionais e quando escrevermos interfaces gráficas de usuário (GUIs) nas próximas aulas. Por enquanto, entretanto, exploraremos o uso de lambdas com nossas coleções.

5.4 O método de coleta *forEach*

Ao introduzir a ideia de um lambda acima, mencionamos o conceito de um método *doThisForEachElement,* na classe de coleção. Este método em Java é realmente chamado *forEach.*

Se tivermos uma coleção chamada *myList*, podemos escrever *myList.forEach* (algum código a ser aplicado a cada elemento da lista). O parâmetro desse método será um lambda - um pedaço de código - com uma sintaxe, como nós vimos acima. O método *forEach* executará o lambda para cada elemento da lista, passando como um parâmetro para o lambda por sua vez. Vamos olhar para algum código concreto. O método *printList* da nossa classe *AnimalMonitor* se parece com isso:

```
/ **
* Imprima detalhes de todos os avistamentos.
* /

public void printList()
{
  for(Sighting record : sightings) {
    System.out.println(record.getDetails());
  }
}
```

Agora podemos escrever a seguinte versão baseada em lambda deste método:

```
/ **
* Imprima detalhes de todos os avistamentos.
* /
public void printList()
{
  sightings.forEach(
    (
      System.out.println(record.getDetails());
    }
  );
}
```

O corpo desta nova versão do *printList* consiste em uma única instrução (mesmo que abranja seis linhas), que é uma chamada ao método *forEach* da lista de aparições. É certo que isso ainda não parece mais simples do que antes, mas chegaremos lá em um momento. É importante observar que não há nenhuma declaração de loop explícita nesta versão, diferente da versão anterior - a iteração é tratada implicitamente pelo método *forEach*.

O parâmetro do *forEach* é um lambda. O lambda em si possui um parâmetro chamado *record*. Cada elemento da lista de avistamentos será usado como parâmetro para este lambda, por sua vez, e o corpo do lambda será executado. Dessa forma, os detalhes de cada elemento serão impressos. Esse estilo de sintaxe é completamente diferente de qualquer outro encontrado nos capítulos anteriores deste material, portanto, dedique algum tempo para entender esse primeiro exemplo de lambda em ação. Embora em breve mostremos algumas pequenas modificações na sintaxe usada aqui, ela ilustra um uso típico de lambdas, e as ideias fundamentais serão repetidas em todos os exemplos futuros:

- Um lambda começa com uma lista de parâmetros;

- Um símbolo de seta segue os parâmetros;

- As instruções a serem executadas seguem o símbolo de seta.

5.4.1 Variações da sintaxe lambda

Um objetivo da sintaxe lambda em Java é permitir escrever código de forma clara e concisa. A capacidade de omitir o modificador de acesso público, por exemplo, e o tipo de retorno, é o começo disso. O compilador, no entanto, permite-nos ir além: vários outros elementos podem ser omitidos, onde o compilador pode resolvê-los sozinho.

O primeiro é o tipo do parâmetro para o lambda (registro, no nosso caso). Como o tipo do parâmetro lambda deve sempre corresponder ao tipo dos elementos da coleção, o compilador pode resolver e podemos omitir isso. Nosso código fica assim:

```
sightings.forEach (
  (record) ->
  {
    System.out.println(record.getDetails());
  }
);
```

A próxima simplificação diz respeito a parênteses e colchetes:

-Se a lista de parâmetros contiver apenas um único parâmetro, os parênteses poderão ser omitidos;

-Se o corpo do lambda contiver apenas uma única instrução, os colchetes poderão ser omitidos.

Aplicando essas duas simplificações, o código agora se torna curto o suficiente para caber em uma única linha, e podemos escrever a chamada *forEach* como:

```
sightings.forEach(record -> System.out.println(record.getDetails()));
```

Esta é a versão que preferimos em nosso próprio código.

5.5 Fluxos

O método *forEach* de *ArrayList* (e classes de coleção relacionadas) é apenas um exemplo específico de um método que utiliza o recurso de fluxos introduzidos no Java 8. Os fluxos são um pouco como coleções, mas com algumas diferenças importantes:

Os elementos em um fluxo não são acessados por meio de um índice, mas geralmente, sequencialmente. O conteúdo e a ordem do fluxo não podem ser alterados - as alterações exigem a criação de um novo fluxo. Um fluxo pode ser infinito!

O conceito de fluxo é usado para unificar o processamento de conjuntos de dados, independentemente de onde esse conjunto de dados vem. Pode ser o processamento de elementos de uma coleção (como discutimos), o processamento de mensagens vindas de uma rede, o processamento de linhas de texto de um arquivo de texto ou todos os caracteres de uma *string*. Não importa qual é a fonte dos dados - podemos ver qualquer um deles como um fluxo e, em seguida, usar as mesmas técnicas e métodos para processar os elementos.

Os fluxos adicionam mais um novo recurso significativo sobre os recursos anteriores, em Java, para o processamento de coleções: o potencial para processamento paralelo seguro. Há muitas situações nas quais os elementos de pedido em uma coleção são processados não importam ou quando o processamento de um elemento na coleção é amplamente independente dos outros elementos. Onde essas condições se aplicam, a paralelização do processamento da coleção pode oferecer uma aceleração significativa ao lidar com grandes quantidades de dados. No entanto, não discutiremos mais isso aqui. Um *ArrayList* não é, por si só, um fluxo, mas podemos solicitar que forneça a fonte de um fluxo chamando seu método. O fluxo retornado conterá todos os elementos do *ArrayList* na ordem do índice. Se ignorarmos a questão da paralelização, os fluxos não nos permitirão fazer nada que ainda não possamos programar, usando recursos de linguagem que já conhecemos. No entanto, os fluxos capturam ordenadamente muitas das tarefas mais comuns que queremos executar nas coleções, de uma forma que as torna muito mais simples de programar do que estamos acostumados, além de serem mais fáceis de entender.

5.5.1 Filtros, mapas e reduções

Observamos acima que o conteúdo de um fluxo não pode ser modificado, portanto, as alterações exigem que um novo fluxo seja criado. Muitos tipos variados de processamento de fluxos podem ser alcançados com apenas três tipos diferentes de funções: filtrar, mapear e reduzir. Ao variar e combinar isso, podemos fazer a maior parte do que precisaremos.

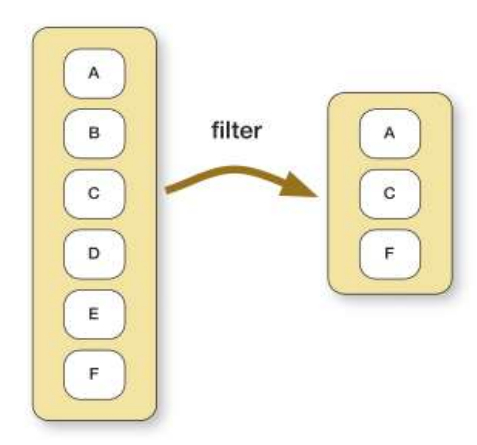
Antes de examinar os métodos Java específicos, discutiremos a ideia básica dessas funções. Todos eles são aplicados a um fluxo para processá-lo de uma certa maneira. Todos são muito comuns e úteis.

A função *filter*

A função de filtro pega um fluxo, seleciona alguns dos elementos e cria um novo fluxo apenas com os elementos selecionados (Figura 5.1). Alguns elementos são filtrados. O resultado é um fluxo com menos elementos (um subconjunto do original).

Um exemplo de nossa coleção de avistamentos de animais, poderia ser selecionar os avistamentos de elefantes dentre os demais registrados. Começamos com todos os avistamentos, aplicamos um filtro que seleciona apenas elefantes e ficamos com um fluxo de todos os avistamentos de elefantes.

Figura 1 – Filtro para avistamento de elefante

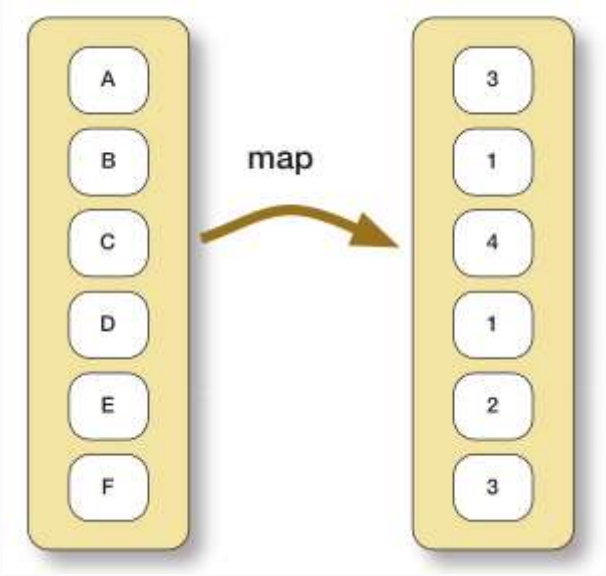


Fonte: BlueJ.

A função map

A função map pega um fluxo e cria um novo fluxo, onde cada elemento do fluxo original é mapeado para um novo elemento diferente no novo fluxo (Figura 2). O novo fluxo terá o mesmo número de elementos, mas o tipo e o conteúdo de cada elemento podem ser diferentes; é derivado de alguma maneira do elemento original. Em nosso projeto, um exemplo é substituir todos os objetos de mira no fluxo original pelo número de animais localizados nessa mira. Começamos com um fluxo de objetos de mira e terminamos com um fluxo de números inteiros.

Figura 2 - Mapeamento



Fonte: BlueJ.

A função *reduction*

A função de redução pega um fluxo e recolhe todos os elementos em um único resultado (Figura 3). Isso pode ser feito de maneiras diferentes, por exemplo, adicionando todos os elementos juntos ou selecionando apenas o menor elemento do fluxo. Começamos com um fluxo e terminamos com um único resultado.

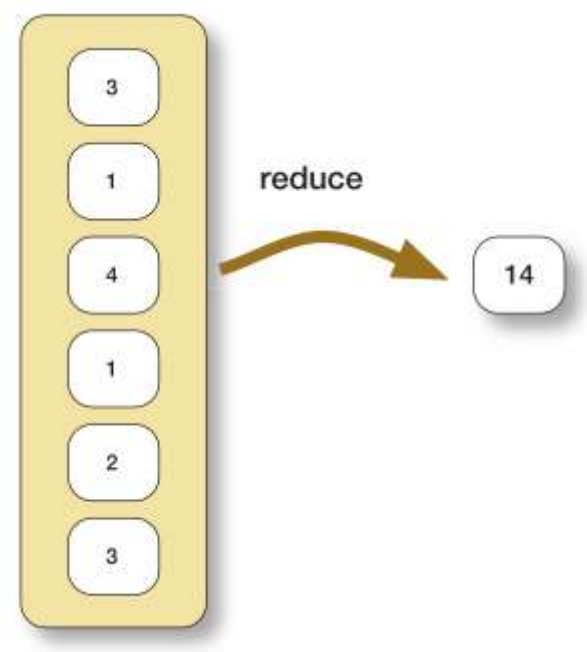
Em nosso projeto de monitoramento de animais, isso seria útil para contar quantos elefantes vimos: Uma vez que tenhamos um fluxo de todos os avistamentos de elefantes, podemos usar o reduzir para somar todos eles. (Lembre-se: cada instância de observação pode ser de vários animais).

5.5.2 Tubulações

Os fluxos e as funções de fluxo se tornam ainda mais úteis quando várias funções são combinadas em um pipeline. Um pipeline é o encadeamento de dois ou mais desses tipos de função, para que sejam aplicados um após o outro.

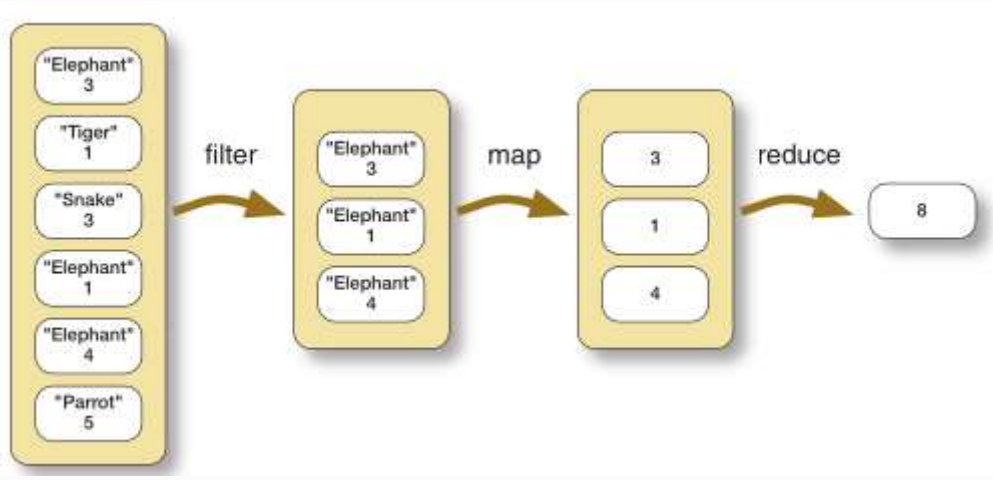
Vamos supor que queremos descobrir quantas aparições de elefantes registramos. Podemos fazer isso aplicando um filtro, mapear e reduzir a função uma após a outra (Figura 4). Neste exemplo, primeiro filtramos todos os avistamentos para selecionar apenas avistamentos de elefantes, depois mapeamos cada objeto de avistamento para o número de elefantes avistados nesse avistamento e, finalmente, adicionamos todos eles com uma função de redução. Observe que, após a aplicação da função de mapa, não estamos mais lidando com um fluxo de objetos *Sighting*, mas com um fluxo de valores inteiros. Essa forma de transformação entre o fluxo de entrada e o fluxo de saída é muito comum na função map.

Figura 3 – Reduzir a função



Fonte: BlueJ.

Figura 4 - Mapear e reduzir a função uma após a outra



Fonte: BlueJ.

Podemos escrever isso usando pseudocódigo, da seguinte maneira:

sightings.filter(name == elephant).map(count).reduce(soma);

Esta não é a sintaxe correta do Java, mas você entendeu. Veremos o código Java correto abaixo. Ao combinar essas funções de maneiras diferentes, muitas operações diversas podem ser alcançadas. Se, por exemplo, quisermos saber quantos relatórios um determinado observador fez em um determinado dia, podemos fazer o seguinte:

```
sightings.filter(spotter == spotterID)
 .filter(period == dayID)
```

```
  .reduce(count elements);
```

Os pipelines sempre começam com uma fonte (um fluxo), seguida por uma sequência de operações. As operações são de dois tipos diferentes: operações intermediárias e operações terminais. Em um pipeline, a última operação deve sempre ser uma operação terminal e todas as outras devem ser operações intermediárias:

```
source.inter1(...).inter2(...).inter3(...).terminal(...);
```

As operações intermediárias sempre produzem um novo fluxo no qual a próxima operação pode ser aplicada, enquanto as operações do terminal produzem um resultado ou são nulas (como *forEach*, que é uma operação do terminal). A documentação para os métodos de fluxo sempre informa se uma operação é intermediária ou terminal.

Agora, discutiremos a sintaxe Java dos métodos que implementam essas funções. É aqui que os fluxos e os lambdas se encontram: o parâmetro para cada um dos métodos é um lambda.

5.5.3 O método de filtro

Conforme discutido, o método de filtro cria um subconjunto do fluxo original. Normalmente, esse subconjunto será o elemento que preenche alguma condição; ou seja, há algo sobre esses elementos específicos que significa que queremos reter e agir sobre eles, e não sobre os outros. Com o projeto organizador de músicas, podemos querer encontrar todas as faixas de um determinado artista. Com o projeto de monitoramento de animais, podemos querer encontrar todos os avistamentos de um determinado animal. Ambos envolvem a iteração sobre suas respectivas listas e o teste de cada elemento para verificar se ele atende a um requisito específico. Se a condição necessária for atendida, continuaremos processando o elemento de alguma maneira.

O método de filtro de um fluxo é uma operação intermediária que transmite ao seu fluxo de saída apenas os elementos do de entrada, que atendem a uma determinada condição. O método pega um lambda que recebe um elemento e retorna um resultado booleano. (Um lambda que pega um elemento e retorna verdadeiro ou falso é chamado de predicado). Se o resultado for verdadeiro, o elemento será incluído no fluxo de saída, caso contrário, será deixado de fora. Por exemplo, podemos querer selecionar apenas os objetos de observação relacionados a elefantes. Aqui está como isso pode ser feito:

```
sightings.stream ()
  .filter (s -> "Elefante".equals(s.getAnimal()))
  . ...
```

Observe que primeiro precisamos criar um fluxo a partir da lista de avistamentos, para poder aplicar a função de filtro. Não precisamos especificar o tipo de parâmetro do lambda, porque o compilador sabe que um *ArrayList* <*Sighting*> fornecerá um fluxo de objetos *Sighting*; portanto, o parâmetro para o predicado terá o tipo *Sighting*. De maneira mais geral, faz sentido tornar a filtragem parte de um método que possa selecionar diferentes tipos de animais; portanto, aqui está como reescreveríamos o método *printSightingsOf* do nosso projeto original em uma versão baseada em fluxo:

```
/ **
* Imprima detalhes de todos os avistamentos de um determinado animal.
```

```
*@param animal O tipo de animal.
*/
public void printSightingsOf (String animal)
{
sightings.stream()
  .filter(s -> animal.equals(s.getAnimal()))
  .forEach(s -> System.out.println(s.getDetails()));
}
```

É importante saber que cada operação do fluxo deixa seu fluxo de entrada não modificado. Cada operação cria um novo fluxo, com base em sua entrada e na operação, para passar para a próxima operação na sequência. Aqui, a operação do terminal é um *forEach* que resulta na impressão dos avistamentos que sobreviveram ao processo de filtragem anterior. O método *forEach* é uma operação de terminal porque não retorna outro fluxo como resultado - é um método nulo.

5.5.4 O método do mapa

O método "*map"* é uma operação intermediária que cria um fluxo com objetos diferentes, possivelmente de um tipo diferente, mapeando cada um dos elementos originais para um novo elemento no novo fluxo.

Para ilustrar isso, considere que, quando estamos imprimindo os detalhes de um avistamento, só estamos realmente interessados na sequência retornada pelo método *getDetails*, e não no próprio objeto "Avistamento". Uma alternativa ao código acima (onde chamamos *getDetails* () na chamada *System.out.println*) seria mapear o objeto *Sighting* para a sequência de detalhes primeiro. A operação do terminal pode apenas imprimir esses detalhes:

```
/ **
* Imprima todos os avistamentos pelo observador.
* @param spotter O ID do observador.
* /
public void printSightingsBy(int spotter)
{
sightings.stream()
  .filter(sighting -> sighting.getSpotter() == spotter)
  .map(sighting -> sighting.getDetails())
  .forEach(details -> System.out.println(details));
}
```

Este exemplo é equivalente à versão que vimos anteriormente. O outro exemplo que mencionamos acima - o mapeamento para o número de animais avistados em cada avistamento - ficaria assim:

```
sightings.stream()
  .filter(sighting ?> sighting.getSpotter() == spotter)
  .map(sighting ?> sighting.getCount())
```

. ...

Esse segmento de código resultaria em um fluxo de números inteiros, que poderíamos processar posteriormente.

5.5.5 O método de redução

As operações intermediárias que vimos até agora tomam um fluxo como entrada e saída de um fluxo. Às vezes, no entanto, precisamos de uma operação que "reduza" seu fluxo de entrada para apenas um único objeto ou valor, e essa é a função do método de redução, que é uma operação terminal.

O código completo para o exemplo dado acima - selecionando todos os animais de um determinado tipo, mapeando o número de animais em cada observação e, finalmente, adicionando todos os números - é o seguinte:

```
public int getCount(String animal)

{
  return sightings.stream()
    .filter(sighting -> animal.equals(sighting.getAnimal()))
    .map(sighting -> sighting.getCount())
    .reduce(0, (total, count) -> return total + count);
}
```

Podemos ver que os parâmetros para o método de redução parecem um pouco mais elaborados do que para os métodos de filtro e mapa. Existem dois aspectos aqui:

-O método de redução possui dois parâmetros. O primeiro em nosso exemplo é 0 e o segundo é um lambda.

-O lambda usado aqui possui dois parâmetros, chamados total e count.

Para entender como isso funciona, pode ser útil examinar o código para adicionar todas as contagens como se estivéssemos escrevendo da maneira tradicional. Escreveríamos um *loop* parecido com este:

```
public int getCount(String animal)
{
  int total = 0;
  for(Sighting sighting : sightings) {
    if(animal.equals(sighting.getAnimal())) {
      total = total + sighting.getCount();
    }
  }
  return total;
}
```

O processo de cálculo da soma envolve as seguintes etapas:

- Declare uma variável para manter o valor final e dê a ele um valor inicial de 0;
- Iterar sobre a lista e determinar quais registros de observação se relacionam com o animal em particular no qual estamos interessados (a etapa de filtragem);
- Obtenha a contagem do avistamento identificado (a etapa de mapeamento);
- Adicione a contagem à variável
- Retorne a soma que foi acumulada na variável ao longo da iteração.

O ponto chave que é relevante para escrever uma versão baseada em fluxo desse processo é reconhecer que a variável total atua como uma forma de "acumulador": cada vez que uma contagem relevante é identificada, ela é adicionada ao valor acumulado até o momento total para fornecer um novo valor que será usado na próxima vez no *loop*. Para que o processo funcione, obviamente, o total precisa receber um valor inicial de 0 para a primeira contagem a ser adicionada.

O método de redução usa dois parâmetros: um valor inicial e um lambda. Formalmente, o valor inicial é chamado de identidade. É o valor para o qual nosso total de execução é inicializado. Em nosso código, passamos “0” para esse parâmetro.

O segundo parâmetro é um lambda com dois parâmetros: um para o total em execução e outro para o elemento atual do fluxo. O lambda que escrevemos em nosso código é:

(total, count) -> return total + count

O corpo do lambda deve retornar o resultado da combinação do total de corrida e do elemento. No nosso exemplo, "combinar" significa apenas adicioná-los. O efeito da aplicação desse método de redução é inicializar um total em execução como zero, adicionar cada elemento do fluxo a ele e retornar o total no final.

Como não há mais um loop explícito em nosso código, pode ser um pouco complicado de entender no começo, mas o processo geral contém todos os mesmos elementos da versão que usa o *loop for-each*.

5.5.6 Removendo de uma coleção

Observamos que o método de filtro não altera realmente a coleção subjacente da qual um fluxo foi obtido. No entanto, lambdas predicadas facilitam a remoção de todos os itens de uma coleção que correspondem a uma condição específica. Por exemplo, suponha que desejamos excluir da lista de avistamentos todos os registros de observação onde a contagem é zero. O código a seguir, utiliza o método *57emovei* da coleção:

```
/ **
* Remova da lista de avistamentos todos esses registros com
* uma contagem de zero.
```

```
* /
public void removeZeroCounts()
{
  sightings.removeIf(sighting -> sighting.getCount() == 0);
}
```

Mais uma vez, a iteração está implícita. O método *removeIf* passa cada elemento da lista, por sua vez, para o predicado lambda e remove todos aqueles para os quais o lambda retorna verdadeiro. Observe que este é um método da coleção (não de um fluxo) e modifica a coleção original.

5.5.7 Outros métodos de fluxo

Dissemos que muitas tarefas comuns podem ser realizadas com esses três tipos de operações: filtrar, mapear e reduzir. Na prática, Java oferece muito mais métodos em fluxos, e veremos alguns deles mais adiante neste material. No entanto, a maioria dessas são apenas variações das três operações introduzidas aqui, mesmo que tenham nomes diferentes.

A classe *Stream*, por exemplo, possui métodos chamados *count*, *findFirst* e *max* - todas são variantes de uma operação de redução - e métodos chamados *limit* e *skip*, que são exemplos de um filtro.

TERMOS INTRODUZIDOS NESTE CAPÍTULO

Programação funcional, lambda, fluxo, pipeline, filtro, mapa e reduzir.

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!

Vimos, na aula 01, que classes como *ArrayList* nos permitem criar coleções contendo um número arbitrário de objetos. A biblioteca Java contém mais coleções como essa, e examinaremos outras na próxima unidade. Você verá que poder usar coleções com confiança é uma habilidade importante para escrever programas interessantes. Dificilmente, existe um aplicativo que veremos a partir de agora que não usa coleções de alguma forma.

Ao usar coleções, surge a necessidade de iterar sobre todos os elementos em uma coleção para fazer uso dos objetos armazenados nelas. Para esse fim, vimos o uso de *loops* e “iteradores”. *Loops* também são um conceito fundamental na computação que você usará em todos os projetos a partir de agora. Certifique-se de se familiarizar o suficiente com os *loops* de escrita - você não irá muito longe sem eles. Ao decidir sobre o tipo de *loop* a ser usado em uma situação específica, geralmente é útil considerar se a tarefa envolve iteração definida ou indefinida. Existe certeza ou incerteza sobre o número de iterações que serão necessárias?

Além disso, mencionamos a biblioteca de classes Java, uma grande coleção de classes úteis que podemos usar para tornar nossas próprias classes mais poderosas. Precisamos estudar a biblioteca com mais detalhes para ver o que mais há nela que devemos conhecer.

Na aula 02, introduzimos alguns conceitos novos que são relativamente avançados para esta etapa do material, mas que estão intimamente associados aos conceitos abordados na aula 01. Examinamos um estilo funcional para processar fluxos de dados. Nesse estilo, não escrevemos *loops* para processar os elementos em uma coleção. Em vez disso, aplicamos um pipeline de operações a um fluxo derivado de uma coleção, a fim de transformar a sequência no formato que queremos. Cada operação no pipeline define o que deve ser feito com um único elemento da sequência. As transformações de sequência mais comuns são executadas através dos métodos de filtro, mapeamento e redução.

As operações em um pipeline geralmente tomam lambdas como parâmetros para especificar o que deve ser feito com cada elemento da sequência. Um lambda é uma função anônima que aceita zero ou mais parâmetros e possui um bloco de código que desempenha a mesma função que um corpo de método.

Olhando para o projeto *animal-Monitoring-v1* em que você está trabalhando, você deve ter notado que existem dois métodos restantes para cada *loop* que ainda não reescrevemos usando fluxos. Esses são os métodos que retornam novas coleções como resultados do método; ainda não discutimos como criar uma nova coleção a partir de um fluxo. Voltaremos a isso na próxima aula, na qual veremos como criar uma nova coleção a partir da sequência transformada.

Bom trabalho!

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!

UniFil