

Java 8: Iniciando o desenvolvimento com a Streams API

Java 8: Iniciando o desenvolvimento com a Streams API

Postado em Dezembro 2016
Por Carlos Silva

Conheça a Streams API, novo recurso do Java que facilita o desenvolvimento, reduz o tamanho do código e simplifica o uso do paralelismo

Neste artigo mostraremos como trabalhar com uma das principais novidades do Java 8, a Streams API, recurso que traz novas classes e métodos que ajudam a manipular coleções de maneira mais simples e eficiente, lançando mão do estilo de programação funcional. Aprenderemos, a partir dos conceitos e da elaboração de um exemplo prático, como utilizar essa API e conhecer os vários benefícios que ela pode trazer. Com base nesse estudo o leitor terá uma base sólida para incorporar essa nova opção no seu dia a dia para o desenvolvimento de soluções.

Desde o lançamento da primeira versão do Java, mais de 20 anos se passaram. Nesse período, mudanças significativas na linguagem foram ocorrer apenas em meados de 2004, com a chegada do Java 5. A inclusão de Generics, Enums e annotations foram algumas das novidades.

Após essa versão, novas mudanças de impacto voltaram a acontecer na linguagem mais de uma década depois, com o lançamento da versão 8, feito pela Oracle em 2014. Ao todo são mais de 80 novas funcionalidades listadas, que vão além de apenas mudanças singelas na máquina virtual (JVM), causando impactos naquilo que mais afeta o desenvolvedor: a forma como ele escreve o código. Na sintaxe, por exemplo, houveram alterações significativas com a incorporação de conceitos originários de linguagens que seguem o paradigma funcional, oferecendo assim mais facilidade em tarefas que antes demandavam maior complexidade e muitas linhas de código.

Uma das novas *features* recebe o nome de Expressões Lambda (EL). Essa novidade proporcionou ao Java a adição de recursos muito disseminados em linguagens funcionais, facilitando em muito a vida de quem desenvolve sistemas. Para aqueles que já se arriscaram em linguagens como Groovy, Scala e Clojure, que executam sobre a JVM, e que estão acostumados, por exemplo, a passar comportamentos, isto é, funções como argumentos em uma chamada de método, certamente essas mudanças causarão menos impacto do que naqueles que nunca tiveram esse contato e cujo costume é escrever métodos que recebem apenas objetos ou tipos primitivos como parâmetro.

Outro recurso adicionado à nova versão da linguagem é a API para lidar com datas, a *Date and Time*, baseada na famosa biblioteca JodaTime. O novo pacote, **java.time**, possui várias classes para trabalhar com objetos que armazenam apenas datas, horas ou mesmo ambos de forma simultânea.

Essa versão trouxe também o recurso denominado *Default Methods*, que foi introduzido para possibilitar a evolução de interfaces ao permitir que interfaces já existentes ofereçam métodos novos sem que os códigos que as implementem também tenham que fornecer uma implementação para esses métodos. Diante disso, note que um dos focos dessa nova versão continua sendo manter a compatibilidade com códigos legados e ser o menos intrusivo possível, ou seja, afetar o menos possível as antigas APIs. Essa nova *feature* foi usada para incluir o método **stream()** na API de Collections, por exemplo, e possibilitar que todas as coleções sejam fontes de dados para streams. Além deste, outros métodos padrão também foram incorporados à API de coleções, como o **removelf()** na interface **Collection**, e em **Comparator**, o método **reversed()**, que retorna um novo comparador que realiza a ordenação ao contrário.

A Streams API traz uma nova opção para a manipulação de coleções em Java seguindo os princípios da programação funcional. Combinada com as expressões lambda, ela proporciona uma forma diferente de lidar com conjuntos de elementos, oferecendo ao desenvolvedor uma maneira simples e concisa de escrever código que resulta em facilidade de manutenção e paralelização sem efeitos indesejados em tempo de execução.

A proposta em torno da Streams API é fazer com que o desenvolvedor não se preocupe mais com a forma de se programar o comportamento, deixando a parte relacionada ao controle de fluxo e loop a cargo da API. É algo muito parecido com o que é feito com Threads, onde os aspectos mais complexos ficam encapsulados em APIs e as regras de negócio passam a ser a única responsabilidade do desenvolvedor.

Outro ponto a se destacar sobre a Streams API diz respeito à eficiência do processamento. Com o aperfeiçoamento constante do hardware, sobretudo a proliferação das CPUs multicore, a API levou isso em consideração e com o apoio do paradigma funcional, suporta a paralelização de operações para processar os dados – abstraindo a lógica de baixo nível para se ter um código *multithreading* – e deixa o desenvolvedor concentrar-se totalmente nas regras existentes.

A paralelização de operações consiste basicamente em dividir uma tarefa maior em subtarefas menores, processar essas subtarefas em paralelo e, em seguida, combinar os resultados para obter o resultado final. Em sua estrutura, a API de Streams fornece um mecanismo similar para trabalhar com a Java Collections, convertendo a coleção em uma stream, em um primeiro momento, processando os vários elementos em paralelo em seguida e, por fim, reunindo os elementos resultantes em uma coleção.

Com base no que foi visto até aqui, o objetivo desse artigo é ensinar, de forma prática e objetiva, como trabalhar com Streams e como usar os diferentes tipos de operações ofertadas por esse novo recurso.

Collections e Streams

Você já parou para pensar as dificuldades que teríamos se as coleções não existissem? Hoje em dia, quase todos os aplicativos Java fazem uso e processam recursos relacionados à interface **Collection**. Por exemplo, o desenvolvedor pode ter a necessidade de criar uma coleção para armazenar as vendas realizadas em uma loja e em seguida processar esses dados com o intuito de descobrir em que momento do dia ocorre o maior volume de vendas. Com base nessas informações, ações podem ser tomadas, como colocar um número maior de funcionários nesse período ou mesmo alocar os funcionários mais qualificados.

O framework de coleções do Java possui mais de 18 anos, tendo surgido com a versão 2 da linguagem, mas no decorrer desse tempo sofreu apenas algumas evoluções. Até o Java 8, a principal evolução havia sido a adição de recursos devido à criação de Generics, no Java 5, o que permitiu a criação de coleções “tipadas”.

A sua estabilidade se justifica, sobretudo, devido à realidade existente até alguns anos atrás, quando a evolução dos processadores era focada no aumento do poder de processamento de um núcleo único (core). No entanto, avanços ocorreram e o foco da indústria mudou, concentrando-se na capacidade de termos vários núcleos de processamento em um mesmo processador.

Para que esse novo cenário possa trazer ganhos ao usuário, no entanto, não apenas a evolução em nível de hardware se faz necessária. Os softwares também precisam suportar esse novo paradigma, que realmente possibilita a execução de vários processos simultaneamente.

Com isso em mente e levando em consideração as mudanças realizadas para a implementação das Expressões Lambda, o framework de coleções do Java passou por muitas alterações e melhorias.

Isso ocorreu principalmente em função da disponibilização dos *Default Methods* que, como mencionado, proporcionam a evolução de interfaces já muito conhecidas pelos desenvolvedores.

A nova opção de as interfaces terem *Default Methods* possibilitou ao Java 8 criar uma grande quantidade de métodos na Collections API. Esses métodos são “herdados” por aqueles que implementarem a interface e construções mais eficientes podem ser adicionadas às classes quando apropriado.

Com o auxílio desses novos recursos os projetistas da Oracle disponibilizaram uma nova abstração no Java 8 para trabalhar com coleções. De nome **Stream**, trata-se de uma poderosa solução para processar coleções de maneira declarativa, ao invés da tradicional e burocrática forma imperativa. Na forma imperativa, para realizar uma iteração simples, por exemplo, o desenvolvedor tem que se preocupar não apenas com o que deve ser feito em cada elemento, isto é, com as regras associadas ao processamento dos elementos da lista, mas também com a maneira de realizar essa iteração.

Assim, esse procedimento possibilita que a mesma iteração tenha resultados diferentes em função de alterações imprevistas em variáveis de controle ou mesmo mutações no decorrer da repetição.

O trecho de código apresentado na **Listagem 1** mostra um exemplo de uso da forma imperativa. Nele deseja-se percorrer uma lista finita de números inteiros e calcular a soma de todos os números pares. Nesse caso três dificuldades podem ser encontradas em virtude dessa abordagem.

Listagem 1. Cálculo da soma de valores inteiros pares antes do Java 5.

 Copy

```
private static int somaIterator(List list) {
    Iterator it = list.iterator();
    int soma = 0;
    while (it.hasNext()) {
        int num = it.next();
        if (num % 2 == 0) {
            soma += num;
        }
    }
    return soma;
}
```

A primeira diz respeito à forma como a lista de valores será iterada, isto é, o algoritmo utilizado para varrer a lista. A segunda dificuldade é viabilizar uma maneira eficiente e paralela para percorrer a lista, sobretudo quando existe a necessidade de manipular uma quantidade extensa de dados. Por fim, o terceiro entrave é a verbosidade do código, ou seja, a grande quantidade de código para realizar tarefas tidas como simples.

Com o advento do Java 5, uma nova maneira de percorrer um array ou coleção (ou qualquer tipo de objeto que implemente a interface `java.lang.Iterable`) foi criada, deixando o código-fonte mais curto, como mostra a **Listagem 2**. No entanto, note que mesmo com esse novo recurso, implementar o código para percorrer uma lista é algo que acaba se tornando tedioso e cansativo. Quantas vezes um desenvolvedor Java já escreveu um `for` desse tipo em sua carreira?

Listagem 2. Cálculo da soma de valores inteiros pares utilizando o laço for melhorado do Java 5.

 Copy

```
private static int somaIterator(ListList<Integer> list) {
    int soma = 0;
    for (int num : list) {
        if (num % 2 == 0) {
            soma += num;
        }
    }
    return soma;
}
```

Pensando em solucionar esse problema, a programação funcional oferece uma nova maneira de pensar no fluxo do algoritmo, introduzindo o conceito de programação declarativa. Esta possibilita trabalhar com fluxos a partir do encadeamento de métodos e da passagem de comportamentos via expressões lambda.

Dentre os principais benefícios dessa abordagem podemos citar a obtenção de códigos mais concisos, a imutabilidade de variáveis e o fim dos *side-effects*, uma vez que a garantia de que uma variável não vai mudar assegura, dentro de um fluxo, que, quando o código sofrer paralelização não existirão efeitos indesejáveis como, por exemplo, modificar o objeto original ao invés de se criar um novo.

Um código conciso significa um código breve, preciso e claro. Em suma, expressa muito com poucas palavras. Com todas essas características reunidas, o resultado será um código muito mais simples com relação à sua manutenção, esteticamente mais agradável e menos suscetível a erros.

Agora, com as novas *features* da linguagem, temos uma separação do que fazer do como fazer, ou seja, a partir da versão 8 o desenvolvedor precisa apenas focar em “o que” fazer com os elementos, deixando de lado o “como” percorrer a lista. Além disso, temos, em nível de API, o processamento paralelo, que permite aproveitar as arquiteturas de núcleos múltiplos sem ter que programar linhas de código multiprocessos.

Antes de explorar em detalhes o que pode ser feito com Streams, para que o leitor tenha uma percepção do novo estilo de programação com Java 8, vamos apresentar um exemplo. O propósito desse exemplo é encontrar em uma lista de ordens de serviço todas aquelas que estão relacionadas à ativação, por exemplo, de um serviço de telefonia, e por fim retornar todos os identificadores dessas ordens classificados de maneira decrescente segundo o valor cobrado pelo serviço. No Java 7, o código pode ser escrito conforme a **Listagem 3**. Já no Java 8, pode ser empregado o código mostrado na **Listagem 4**.

Listagem 3. Abordagem no Java 7 ou anterior para trabalhar com coleções.

 Copy

```
List<Ordem>
OrdensAtivacao = new ArrayList<>(); for(Ordem o: ordens) {
if(o.getType() == Ordem.ATIVACAO)
{ OrdensAtivacao.add(t); }
}
Collections.sort(OrdensAtivacao, new Comparator<>{
public int compare(Ordem t1, Ordem t2){
    return t2.getValue().compareTo(t1.getValue());
}});
List<Integer> ordensIDs = new ArrayList<>();
for(Ordem o: OrdensAtivacao) {
ordensIDs.add(t.getId());
}
}
```

Listagem 4. Abordagem no Java 8 utilizando Streams para trabalhar com coleções.

 Copy

```
List<Integer> ordensIDs = Ordem.stream()
    .filter(o -> o.getType() == Ordem.ATIVACAO)
    .sorted(comparing(Ordem::getValue).reversed())
    .map(Ordem::getId)
    .collect(toList());
```

Perceba que o acesso aos elementos de uma stream é diferente de como é feito em coleções. Em uma coleção é possível navegar até os elementos de diferentes maneiras, tanto de forma sequencial quanto por meio de índices. Já em uma **Stream** o acesso aos elementos é sequencial, não sendo possível alcançá-los através de índices, pois inexiste uma estrutura de dados para armazenar os elementos que, por sua vez, são processados sob demanda.

A **Figura 1** mostra o fluxo de operações que acontece durante a execução do código da **Listagem 4**. Primeiramente, obtemos uma stream da lista de dados usando o método **stream()**, da interface **Collection**. Depois, uma série de operações são aplicadas. O método **filter()**, por exemplo, retorna apenas as ordens que são do tipo **ATIVACAO**. Sua saída é processada pela operação **sorted()**, que ordenará as operações de forma decrescente levando em consideração o valor da operação. Em seguida, o resultado de **sorted()** será manipulado pelo método **map()**, que obterá todas as informações que desejamos, ou seja, todos os identificadores das ordens da lista de operações. Por fim, o método **collect()** devolve uma lista de inteiros, em oposição aos demais, que sempre retornam uma nova stream como resultado do processamento.

Saiba que a aparente complexidade do código não deve ser uma preocupação. O importante neste momento é entender como ocorre o processamento de uma stream. Nas próximas seções vamos analisar em detalhes seu funcionamento e o leitor saberá usar a API para implementar códigos similares ao da **Listagem 4** visando sempre construir consultas eficientes.

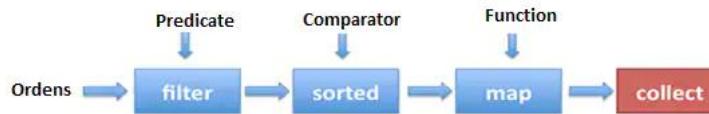


Figura 1. Fluxo de processamento de uma stream.

Outro diferencial é verificado quando se tem a necessidade de manipular grandes quantidades de dados. Nesses casos a Streams API oferece a possibilidade de trabalhar com esses dados de forma paralela, viabilizando uma melhora de desempenho ao tirar proveito do poder de processamento dos computadores modernos.

Tomando como exemplo o código da **Listagem 4**, para que o desenvolvedor consiga fazer uso da paralelização, basta trocar o método **stream()** por **parallelStream()**. Dessa forma a Streams API irá decompor as ações em várias subtarefas, e as operações serão processadas em paralelo, explorando os recursos oferecidos pelos diversos núcleos do processador.

Características de uma Stream Uma Stream pode ser definida como uma “sequência de elementos de uma fonte de dados que oferece suporte a diferentes tipos de operações de agregação”. A seguir, vamos entender cada um dos componentes dessa definição:

Sequência de elementos: Uma stream provê uma interface para um conjunto sequencial de valores de um determinado tipo. Contudo, streams não armazenam elementos. Eles são processados sob demanda;

Fonte de dados: Streams consomem dados de uma fonte, como coleções, arrays ou mesmo recursos de E/S (entrada e saída);

Operações de agregação: Streams suportam operações comuns a linguagens de programação funcionais, como filtrar, modificar, transformar o elemento em outro e assim por diante. Essas operações podem ser realizadas em série ou em paralelo.

Além disso, as operações relacionadas a streams têm duas características fundamentais que as tornam muito diferentes das operações sobre coleções. São elas:

Pipeline: Streams são projetadas de tal maneira que a maior parte de suas operações retornam novas streams. Dessa forma, é possível criar uma cadeia de operações que formam um fluxo de processamento. A isso damos o nome de pipeline;

Iteração interna: Tradicionalmente as coleções usam loops ou iteradores para percorrer seus elementos. Esse tipo de operação é conhecido como iteração externa e é claramente perceptível no código. A **Listagem 1** demonstra esse tipo de abordagem. Já a partir do Java 8, através da Streams API, é possível encontrar métodos como **map()**, **forEach()**, **filter()**, entre outros, que percorrem uma sequência de elementos internamente focando apenas no que “fazer” com os elementos, livrando o desenvolvedor de ter que se preocupar com a forma de iterar sobre a lista e como manipular cada um dos seus elementos. Ou seja, o modo como ocorre a iteração/loop agora fica encapsulado na API.

Como já mencionado, as operações realizadas sobre streams podem ser classificadas como intermediárias ou terminais e quando combinadas formam uma estrutura de processo chamada *pipeline*, que é composta por uma fonte de dados, seguida por zero ou mais operações intermediárias e uma operação terminal. Essa distinção é importante porque as operações intermediárias servem como entrada para outras operações intermediárias ou para a operação terminal e as intermediárias não realizam qualquer ação até o momento em que uma operação terminal seja invocada. Esse mecanismo é conhecido como *lazy evaluation* (avaliação tardia ou “preguiçosa”) e possibilita que o processamento da cadeia de operações seja executado de forma mais performática, postergando a computação até um ponto em que o resultado seja, de fato, importante, evitando assim cálculos desnecessários.

Outra otimização que a Stream API oferece ocorre com a ajuda das operações de curto-círcuito (*short-circuiting*). Essas operações possibilitam que o resultado final seja obtido antes que todos os elementos da stream sejam processados. Um exemplo é a chamada ao método `limit(n)` onde apenas os n primeiros elementos da stream serão processados.

Saiba que as operações intermediárias sempre retornam uma nova stream, de modo que seja possível realizar o encadeamento de múltiplas operações intermediárias. Já as operações terminais, como o próprio nome sugere, residem no final da cadeia de operações e seu objetivo é fechar o processo. Elas retornam um resultado diferente de uma stream, que pode ser um valor ou um objeto. Em suma, a Streams API trabalha convertendo uma fonte de dados em uma **Stream**. Em seguida, realiza o processamento dos dados através das operações intermediárias e, por fim, retorna uma nova coleção ou valor reduzido (*map-reduce*) com a chamada a uma operação terminal.

Tomando como base a **Listagem 4**, `filter()`, `sorted()` e `map()` são tidas como operações intermediárias. Já `collect()` é classificada como terminal, pois encerra a stream e retorna uma lista. As operações intermediárias ainda podem ser divididas em dois grupos: *stateless* e *stateful*. As operações *stateless*, como `filter()` e `map()`, não armazenam o estado do elemento manipulado anteriormente ao processar um novo elemento. Dessa forma, cada elemento pode ser processado independentemente das operações dos outros elementos. Já as operações *stateful*, como `distinct()` e `sorted()`, podem incorporar o estado do elemento processado anteriormente no processamento de novos elementos.

Após conhecer as principais características de uma stream, vamos focar no estudo dos principais recursos que compõem a API. Esses recursos serão analisados de acordo com o fluxo de processamento apresentado na **Figura 1**. Sendo assim, primeiro será mostrado como criar e obter uma stream. Logo após, focaremos nas operações intermediárias e então analisaremos algumas das mais difundidas operações terminais. Por fim, outros recursos da Streams API serão abordados através do desenvolvimento de uma aplicação exemplo.

Trabalhando com a Streams API A Streams API foi desenvolvida sob o pacote `java.util.stream`. Deste modo, este disponibiliza classes e interfaces que suportam as mais variadas operações funcionais que podemos aplicar sobre os dados com o Java 8. O tipo mais importante dentro desse pacote é a interface **Stream**. Dito isso, vamos deixar a teoria de lado e iniciar nosso estudo prático sobre streams.

Como criar streams O primeiro passo para se trabalhar com streams é saber como criá-las. A forma mais comum é através de uma coleção de dados, tendo em vista que o principal propósito dessa API é tornar mais flexível e eficiente o processamento de coleções. A **Listagem 5** mostra como criar uma stream ao invocar o método `stream()` a partir da interface `java.util.Collection`.

Nesse trecho de código, primeiramente uma lista de strings é definida e três objetos são adicionados a ela. Em seguida, uma stream de strings é obtida ao chamar o método `items.stream()`, na linha 7. Outra forma de criar streams é invocando o método `parallelStream()`, que possibilitará paralelizar o seu processamento.

Listagem 5. Criação de uma stream a partir de um List.

 Copy

```
List<String> items = new ArrayList<String>();
    items.add("um");
    items.add("dois");
    items.add("três");
Stream<String> stream = items.stream();
```

O método `stream()` também foi adicionado à interface `java.util.map`. A **Listagem 6** mostra um exemplo de como criar uma stream a partir dessa interface.

Listagem 6. Criação de Stream a partir de um Map

 Copy

```
Map <String, String> map = new HashMap<String, String>();
    map.put("key1", "abacate");
    map.put("key2", "melancia");
    map.put("key3", "manga");
Stream<String> stream = map.values().stream();
```

Além disso, uma stream pode ser gerada a partir de I/O, arrays e valores. Para obter uma stream a partir de valores ou arrays é muito simples: basta chamar os métodos estáticos `Stream.of()` ou `Arrays.stream()`, como mostra o código a seguir:

 Copy

```
Stream<Integer> numbersFromValues = Stream.of(1, 2, 3, 4, 5);
IntStream numbersFromArray = Arrays.stream(new int[] {1, 2, 3, 4, 5});
```

Por sua vez, para criar uma stream de linhas a partir do conteúdo de um arquivo texto (I/O), podemos chamar o método estático **Files.lines(Path path)**. No código a seguir, por exemplo, é possível descobrir a quantidade de linhas que um arquivo possui:

Copy

```
Stream <String> lines= Files.lines(Paths.get("myFile.txt"),
Charset.defaultCharset());
long numbersLines = lines.count();
```

Após conhecer alguns dos diferentes modos para criar e obter streams, vamos focar agora em como processá-las. Para isso, apresentaremos nos próximos tópicos a transformação e o processamento de streams fazendo uso de diferentes operações da interface **Stream**.

Para ilustrar cada uma dessas operações, primeiramente vamos criar a classe **Pessoa** com quatro atributos básicos: **id**, **nome**, **nacionalidade** e **idade**. A **Listagem 7** mostra a implementação dessa classe, que traz, também, o método **populaPessoas()**, criado para preencher uma lista com alguns objetos.

Listagem 7. Implementação da classe Pessoa.

Copy

```
public class Pessoa {
    String id;
    String nome;
    String nacionalidade;
    int idade; //gets e sets omitidos
    public Pessoa(){}
    public Pessoa (String id, String nome, String nacionalidade, int idade){
        this.id = id;
        this.nome = nome;
        this.nacionalidade = nacionalidade;
        this.idade = idade;
    }
    public List<Pessoa> populaPessoas(){
        Pessoa pessoal = new Pessoa("p1" , "Matheus Henrique", "Brasil", 18);
        Pessoa pessoa2 = new Pessoa("p2" , "Hernandez Roja", "Mexico", 21);
        Pessoa pessoa3 = new Pessoa("p3" , "Mario Fernandes","Canada", 22);
        Pessoa pessoa4 = new Pessoa("p4" , "Neymar Junior", "Brasil", 22);
        List<Pessoa> list = new ArrayList<Pessoa>();
        list.add(pessoal);
        list.add(pessoa2);
        list.add(pessoa3);
        list.add(pessoa4);
        return list;
    }
    @Override public String toString() { return this.nome; } }
```

Operações intermediárias

Algumas das operações intermediárias mais utilizadas são: **filter()**, **map()**, **sorted()**, **limit()** e **distinct()**. Portanto, nossos primeiros passos nessa nova API demonstrarão como utilizar essas operações.

Filter O método **filter()** é usado para filtrar elementos de uma stream de acordo com uma condição (predicado). Para isso, ele recebe como parâmetro um objeto que implementa a interface **Predicate<T>** (**interface funcional**) que define uma função com valor de retorno igual a um **boolean** e retorna uma nova stream contendo apenas os elementos que satisfazem à condição.

O código a seguir mostra um exemplo de uso dessa operação. Primeiramente é criada uma lista com alguns objetos do tipo **Pessoa**. Em seguida, com a chamada ao método **stream()** é criada a stream. Logo após, o método **filter()** recebe como parâmetro uma condição, representada por uma expressão lambda, que tem por objetivo buscar todas as pessoas que nasceram no Brasil.

Copy

```
List<String> pessoas = new Pessoa().populaPessoas();
Stream<String> stream = pessoas.stream().filter(pessoa -> pessoa.getNacionalidade().equals("Brasil"));
```

Map

Diante de algumas situações se faz necessário realizar transformações em uma lista de dados. O método **map()** permite realizar essas mudanças sem a necessidade de variáveis intermediárias, apenas utilizando como argumento uma função do tipo **java.util.function.Function**, que, assim como **Predicate<T>**, também é uma interface funcional. Essa função toma cada elemento de uma stream como parâmetro e retorna o elemento processado como resposta. O resultado será uma nova stream contendo os elementos mapeados a partir da stream original.

A **Listagem 8** mostra um exemplo desse tipo de operação. Na linha 4, na nova stream obtida a partir da operação de filtragem, é realizado um mapeamento com o intuito de obter apenas a idade das pessoas presentes no fluxo de dados.

Listagem 8. Exemplo de uso do método **map()**.

 Copy

```
List<String> pessoas = new Pessoa().populaPessoas();
Stream<Integer> stream = pessoas.stream()
    .filter(pessoa -> pessoa.getNacionalidade().equals("Brasil"))
    .map(Pessoa::getIdade);
```

Porém, nesse trecho de código podemos ter um problema com a utilização do método **map()**, haja vista que seu retorno é do tipo **Stream<Integer>**. Esse fatogera boxing dos valores inteiros, isto é, a necessidade de converter o tipo primitivo retornado pelo método **getIdade()** em seu correspondente objeto wrapper. Sendo assim, teremos um *overhead* indesejado, sobretudo quando se tratar de listas grandes.

Pensando nisso, a Streams API oferece implementações para os principais tipos primitivos, a saber: **IntStream**, **DoubleStream** e **LongStream**. Em nosso exemplo, portanto, podemos usar o **IntStream** para evitar o autoboxing e chamar o método **mapToInt()** ao invés do **map()**.

Outro ponto a observar nesse exemplo é a possibilidade de tirar proveito da sintaxe de *method reference*, simplificando ainda mais o nosso código. Para verificar isso, note como, na linha 4, é passado o método **getIdade()** da classe **Pessoa** como parâmetro.

Nota: *Method Reference* é um novo recurso do Java 8 que permite fazer referência a um método ou construtor de uma classe (de forma funcional) e assim indicar que ele deve ser utilizado num ponto específico do código, deixando-o mais simples e legível. Para utilizá-lo, basta informar uma classe ou referência seguida do símbolo “`:`” e o nome do método sem os parênteses no final.

Sorted

A ordenação de elementos em coleções é uma tarefa recorrente no dia a dia de todo desenvolvedor. No Java 8, felizmente, isso foi bastante facilitado, eliminando a necessidade de implementar o verboso **Comparator**, assim como as classes internas anônimas, proporcionando ao código clareza e simplicidade. Para isso, a Streams API oferece a operação **sorted()**. Esse método retorna uma nova stream contendo os elementos da stream original ordenados de acordo com algum critério.

A **Listagem 9** mostra um exemplo que faz uso desse método. Após a filtragem, já explicada, na linha 4 a ordenação é realizada utilizando o método **comparing()** da interface **Comparator**. Esse método recebe uma **Function** como parâmetro e devolve um valor chave que será utilizado na ordenação. Nesse caso, a classificação das informações é feita com base no nome da pessoa, utilizando a ordem natural (alfabética) definida na interface **Comparator** para classificar Strings.

Note que nesse caso também é possível tirar proveito da sintaxe de *method reference*, uma vez que é passado uma referência do método **getNome()** da classe **Pessoa** como parâmetro para a operação **Comparator.comparing()**.

Listagem 9. Exemplo de uso do método **sorted()**.

 Copy

```
List<String> pessoas = new Pessoa().populaPessoas();
Stream<String> stream = pessoas.stream()
    .filter(pessoa -> pessoa.getNacionalidade().equals("Brasil"))
    .sorted(Comparator.comparing(Pessoa::getNome));
```

Distinct

A operação **distinct()** retorna uma stream contendo apenas elementos que são exclusivos, isto é, que não se repetem, de acordo com a implementação do método **equals()**. O código a seguir mostra um exemplo de uso:

```
List<String> pessoas = new Pessoa().populaPessoas(); Stream<String> stream = pessoas.stream().distinct();
```

Limit Já o método **limit()** é utilizado para limitar o número de elementos em uma stream. É uma operação conhecida como curto-círcuito devido ao fato de não precisar processar todos os elementos. Como exemplo, o código a seguir demonstra como retornar uma stream com apenas os dois primeiros elementos:

```
List<String> pessoas = new Pessoa().populaPessoas(); Stream<String> stream = pessoas.stream().limit(2);
```

Operações terminais Esse tipo de operação pode ser identificado pelo tipo de retorno do método, uma vez que uma operação terminal nunca retorna uma interface **Stream**, mas sim um resultado (List, String, Long, Integer, etc.) ou void. A seguir, veremos em detalhes alguns dos métodos mais importantes e em quais situações eles podem ser empregados.

ForEach Através do método **forEach()** é possível realizar um loop sobre todos os elementos de uma stream e executar algum tipo de processamento. No exemplo a seguir, o parâmetro que o método **forEach()** recebe é uma expressão lambda que invoca o método **getNome()** do objeto **pessoa** e imprime o seu retorno no console. Assim, serão exibidos os nomes de todas as pessoas presentes na coleção.

```
List<String> pessoas = new Pessoa().populaPessoas(); pessoas.stream().forEach(pessoa -> System.out.println(pessoa));
```

Average Com o objetivo de auxiliar sua utilização, as implementações de **Stream** para tipos primitivos (**IntStream**, **DoubleStream** e **LongStream**) oferecem vários métodos. Um deles é o **average()**, que permite calcular a média dos valores dos elementos. A **Listagem 11** mostra um exemplo de uso dessa operação, no qual é calculada a média de idade de todas as pessoas que nasceram no Brasil.

Na linha 4, a operação **mapToInt()** é empregada para que possamos obter uma nova stream, composta apenas por valores inteiros. A partir dessa stream, a operação **average()** irá realizar o cálculo da média. Por fim, na linha 6, o método **getAsDouble()** converte o valor retornado por **average()** para o tipo numérico **double**.

A operação **getAsDouble()** é utilizada porque **average()** não retorna um valor numérico e sim um objeto da classe **java.util.Optional**, introduzida no Java 8. Essa classe permite que tratemos algumas situações excepcionais de forma simples como, por exemplo, quando o número de elementos da stream for igual a zero. Nesse caso, o resultado da média será um positivo infinito (qualquer número dividido por 0 é igual a infinito) e certamente não é um valor que o desenvolvedor deseja manipular.

Nota: Observe que o método **average()** utiliza todos os elementos da stream para retornar um único valor. Operações desse tipo são conhecidas como operações de redução (reduction).

Listagem 11. Exemplo de uso do método **average()**.

 Copy

```
List<String> pessoas = new Pessoa().populaPessoas();
double media = pessoas.stream()
    .filter(pessoa -> pessoa.getNacionalidade().equals("Brasil"))
    .mapToInt(pessoa -> pessoa.getIdade())
    .average()
    .getAsDouble();
```

Collect

O método **collect()** possibilita coletar os elementos de uma stream na forma de coleções, convertendo uma stream para os tipos **List**, **Set** ou **Map**. Um exemplo de uso dessa operação pode ser visto no trecho de código a seguir:

 Copy

```
List<String>
pessoas = new Pessoa().populaPessoas();
List<String>
pessoasComM = pessoas.stream().filter(pessoa ->
    pessoa.startsWith("M")).collect(Collectors.toList());
```

Note que após gerar a stream e aplicar um filtro sobre ela, chamamos o método **collect()**, o qual recebe como argumento **Collector.toList()**, que reuni o resultado da stream e os retorna na forma de uma lista.

Count O método **count()** retorna a quantidade de elementos presentes em uma stream. Portanto, assim como **average()**, também é classificado como uma operação de redução (*reduction*). Como exemplo, o trecho de código a seguir mostra como obter o número de pessoas em uma lista cujo nome começa com a letra “N”:

Copy

```
List<String>
pessoas = new Pessoa().populaPessoas();
long qt = pessoas.stream().filter(pessoa ->
pessoa.startsWith("N")).count();
```

AllMatch

Um padrão de processamento comum em aplicações consiste em verificar se os elementos de uma coleção correspondem a um determinado predicado, isto é, a uma característica (propriedades de um objeto).

Com esse objetivo, o método **allMatch()** verifica se todos os elementos de uma stream atendem a um critério passado como parâmetro, através de um **Predicate**, e retorna um valor booleano. O exemplo a seguir apresenta essa situação. Nesse código, cada elemento da stream é submetido a uma condição, que nesse caso é verificar se a pessoa nasceu no México. Se todos os elementos obedecem a essa condição, será retornado **true**. Caso algum dos elementos não satisfaça ao predicado, será retornado **false**.

Copy

```
List<String>
pessoas = new Pessoa().populaPessoas();

boolean todosMexicanos = pessoas.stream().allMatch(pessoa -> pessoa.getNacionalidade().equals("Mexico"));
```

Desenvolvendo uma aplicação com a Streams API

Para apresentar a Streams API na prática, implementaremos uma aplicação exemplo utilizando a versão mais recente do Java e o NetBeans 8.1, IDE que proporcionará mais qualidade e produtividade na codificação, uma vez que suporta as funcionalidades do JDK 8.

Na seção **Links** você encontrará os endereços para download do Java 8 e do NetBeans. Após baixar os arquivos e fazer a instalação, será necessário ativar o suporte do JDK 8 no ambiente de desenvolvimento. Para isso, vamos registrar o Java no IDE conforme os passos a seguir:

Com o NetBeans aberto, no menu principal, selecione *Ferramentas > Plataformas Java*;

Logo após, na caixa de diálogo *Gerenciador de plataforma Java*, clique em *Adicionar Plataforma Java*;

Na caixa de diálogo *Adicionar Plataforma Java*, escolha *Edição Padrão Java (Java Standard Edition)* e clique em *Próximo*;

Informe o diretório que contém o Java 8 e em seguida clique em *Próximo*, como sugere a **Figura 2**:

Clique em *Finalizar* para que a caixa de diálogo *Adicionar Plataforma Java* seja fechada;

Por fim, assegure-se que o JDK 1.8 esteja selecionado (verifique a lista *Plataformas*) e clique em *Fechar*, conforme a **Figura 3**.

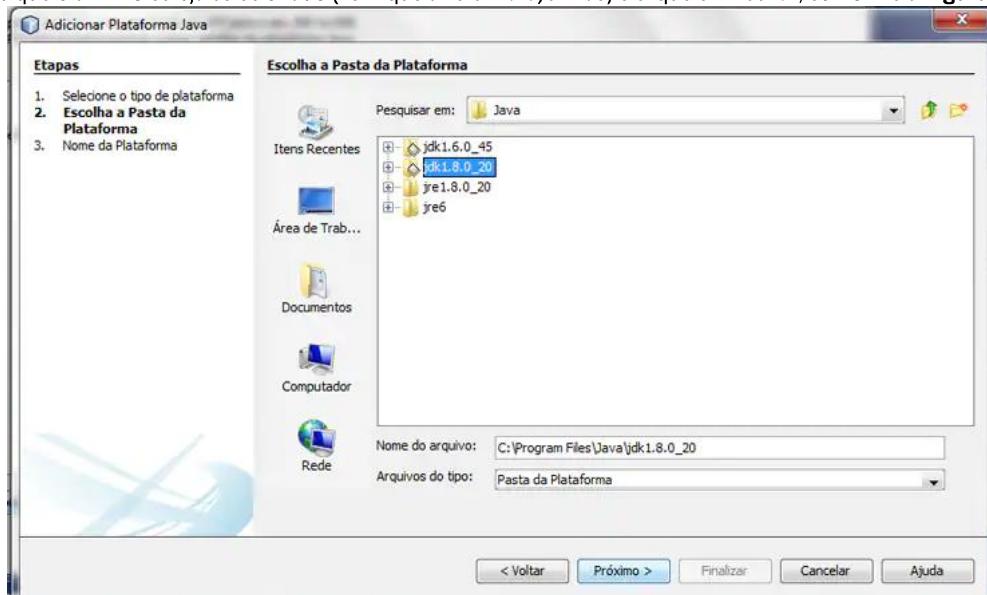


Figura 2.Informando o diretório de instalação do Java 8.

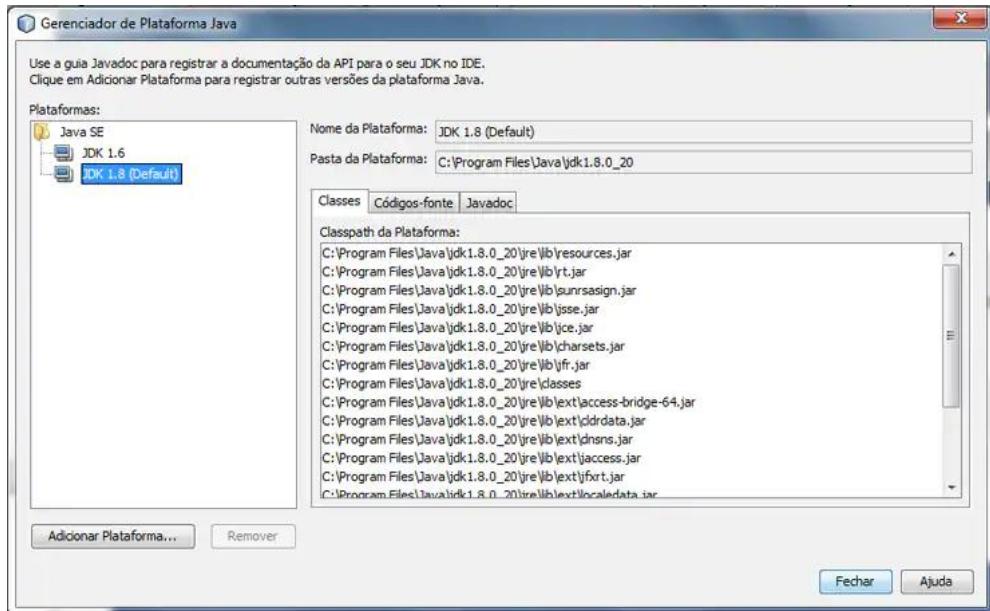


Figura 3.Confirmando a seleção do JDK 1.8.

Feito isso, o NetBeans será executado utilizando as bibliotecas da versão 8 do Java e nenhuma outra configuração será necessária.

Cadastro de jogadores de futebol

Com o ambiente de desenvolvimento pronto, criaremos agora uma pequena aplicação que manipulará dados referentes ao último campeonato brasileiro de futebol. Nessa aplicação o enfoque será dado às informações referentes aos jogadores, a saber: nome, posição em que atua, idade, time em que disputou o campeonato e o número de gols marcados. A **Figura 4** mostra a representação gráfica dessa classe.



Figura 4. Representação gráfica da classe Jogador.

Criando a aplicação no NetBeans Com o NetBeans aberto, crie um projeto Java do tipo Desktop. Para isso, clique no botão *Novo Projeto*, selecione a categoria *Java Desktop* e depois clique em *Próximo*. Na tela que surgir, no campo *Nome do Projeto*, informe “JavaApplicationStream” e clique em *Finish* para confirmar a construção do projeto.

A classe Jogador

Agora que temos a estrutura do projeto pronta e todas configurações realizadas, vamos criar a classe que representará o domínio do sistema, ou seja, a entidade que a aplicação manipulará: **Jogador**. Portanto, no pacote **br.com.cadastro.model**, crie a classe **Jogador** e a implemente conforme o código da **Listagem 12**.

Listagem 12. Implementação da classe Jogador.

Copy

```

package br.com.cadastro.model;

public class Jogador {
    private String nome;
    private String posicao;
    private int idade;
    private String timeAtual;
    private int golsMarcados;
    //métodos gets e sets omitidos...
    @Override
    public String toString() {
  
```

```
        return this.nome + " " + this.posicao + " " + this.getTimeAtual();
    }
}
```

Nessa classe temos os atributos do jogador e os respectivos métodos de acesso. Além disso, temos o método sobrescrito **toString()**, que retorna uma representação mais informativa, concisa, do objeto com o qual estamos trabalhando.

Classe com as operações a serem disponibilizadas O passo seguinte é implementar a classe e as operações que serão realizadas sobre o cadastro. Sendo assim, no pacote **br.com.cadastro.negocio**, crie a classe **JogadorImpl** conforme a **Listagem 13**.

Para simplificar o exemplo, as informações referentes aos jogadores serão obtidas de um arquivo texto, apresentado na **Figura 5**. Seu formato respeita a seguinte disposição: NomeJogador,Posição,Idade,TimeAtual,GolsMarcados.

```
Hernane,Atacante,29,Sport,3
Lucas Pratto,Atacante,27,Atlético-MG,13
Nene,Atacante,33,Vasco,8
Ricardo Oliveira,Atacante,34,Santos,20
Bruno,Lateral-Direito,23,São Paulo,1
Egídio,Lateral-Esquerdo,27,Palmeiras,3
Renan,Goleiro,33,Goiás,0
Ernando,Zagueiro,31,Internacional,3
Lugano,Zagueiro,35,São Paulo,1
Paolo Guerreiro,Atacante,34,Fluminense,8
Cícero,Meio-Campo,28,Fluminense,4
Elias,Volante,27,Corinthias,7
Arrascaeta,Meio-Campo,30,Cruzeiro,3
Luan,Atacante,23,Atlético-MG,8
Mayke,Lateral-Direito,22,Cruzeiro,2
Dedé,Zagueiro,28,Cruzeiro,0
Carlinhos,Lateral-Esquerdo,31,São Paulo,2
Edu Dracena,Zagueiro,33,Palmeiras,4
Willians,Volante,32,Cruzeiro,2
Marcos Rocha,Lateral-Direito,26,Atlético-MG,6
Victor,Goleiro,29,Atlético-MG,0
Rogerio Ceni,Goleiro,41,São Paulo,7
Marcelo Grohe,Goleiro,29,Grêmio,0
Negueba,Atacante,27,Coritiba,4
Bressan,Zagueiro,22,Grêmio,2
Rever,Zagueiro,29,Internacional,5
Michel Bastos,Meio-Campo,33,São Paulo,7
Vitinho,Atacante,22,Internacional,11
William,Atacante,28,Cruzeiro,11
Apodi,Lateral-Direito,27,Chapecoense,5
Arouca,Volante,29,Palmeiras,1
Cristaldo,Atacante,31,Palmeiras,5
Vágner Love,Atacante,37,Corinthias,14
Lucas Lima,Meio-Campo,25,Santos,7
Rodrigo Caio,Volante,22,São Paulo,2
Jefferson,Goleiro,34,Botafogo,0
Marcos Guilherme,Meio-Campo,22,Atlético-PR,4
Henrique Almeida,Atacante,27,Coritiba,12
Thiago Mendes,Volante,28,São Paulo,4
Centurión, Atacante, 27, São Paulo,2
Fernando Prass,Goleiro,35,Palmeiras,0
Zé Roberto,Meio-Campo,38,Palmeiras,2
Malcom,Atacante,18,Corinthias,7
```

Figura 5. Conteúdo do arquivo jogadores.txt.

A primeira operação, declarada na linha 7, tem o objetivo de verificar se o arquivo *jogadores.txt* existe, e para isso, utilizaremos os recursos da Streams API.

No Java 8, a classe **java.nio.file.Files** sofreu algumas alterações para possibilitar trabalhar com streams. Uma das operações adicionadas foi a **Files.list()**, chamado na linha 10, que retorna todos os arquivos de uma pasta através de um **Stream<Path>**. Nesse caso, cada elemento dessa stream representará um objeto do tipo **Path**, possibilitando assim manipular o conteúdo do sistema de arquivos.

Com a stream em mãos, pode ser aplicado um filtro para verificar se o nome de algum dos arquivos da pasta equivale ao valor passado como parâmetro, o que é feito na linha 11, com o predicado **p -> p.toString().endsWith("jogadores.txt")**. Por sua vez, o método **findAny()** checa se alguma ocorrência da característica representada pelo predicado está presente na nova stream. Caso seja confirmada sua existência, é dada continuidade à execução da aplicação. Caso o arquivo não esteja presente, é informada sua ausência e finalizada a execução.

Dando sequência ao processamento, na linha 19 declaramos o método **getListaDeJogadores()**, que lê os dados do arquivo e constrói uma lista com todos os jogadores. Essa lista será recebida pelos demais métodos como parâmetro e servirá de fonte de dados para a criação das streams. Primeiramente, na linha 20, a operação **Files.lines()** cria uma stream com todas as linhas do arquivo informado. Como esse método considera o encoding UTF-8 como padrão e o arquivo que utilizamos está com o tipo ISO-8859-1, existe a necessidade de informar ao método o encoding correto.

A partir dessa stream é possível obter uma lista com todas as linhas do arquivo, onde cada elemento da lista representa uma linha. Para isso, o método **collect()** deve ser chamado, como mostra a linha 21. Então, a lista será percorrida para obtermos as características (atributos) de cada atleta e criar objetos do tipo **Jogador**. Como podemos verificar na **Figura 5**, cada linha contém as informações referentes a um jogador separadas por vírgula.

Obtida a lista de jogadores, nossa fonte de dados, iremos focar na construção de métodos que ilustram o processamento de dados com streams.

Listagem 13. Implementação da classe JogadorImpl.

 Copy

```
package br.com.cadastro.negocio;
//imports omitidos
public class JogadorImpl {
    public boolean verificarArquivoExiste(Path caminho){
        boolean ret = false;
        try {
            Stream<Path> stream = Files.list(caminho);
            Optional<Path> arq = stream.filter(p -> p.toString().endsWith("jogadores.txt")).findAny();
            ret = arq.isPresent(); //informa se o arquivo está presente
        } catch (IOException ex) {
            ex.printStackTrace();
        }
        return ret;
    }
    public List<Jogador> getListaDeJogadores(Path caminho) throws IOException {
        Stream<String> linhas = Files.lines(caminho, StandardCharsets.ISO_8859_1);
        List<String> listaDeLinhas = linhas.collect(Collectors.toList());
        List<Jogador> listaDeJogadores = new ArrayList<>();
        Jogador jogador;
        Iterator it = listaDeLinhas.listIterator();
        String str = null;
        while (it.hasNext()) {
            str = (String) it.next();
        }
        <blockquote>
        <blockquote>
```

```

    public int imprimirSomatorioGols (List<Jogador> jogadores) {
        int soma = jogadores.stream().mapToInt(jogador -> jogador.getGolsMarcados()).sum();
        return soma;
    }

    public void agruparJogadoresPeloTime(List<Jogador> jogadores) {
        Map<String, List<Jogador>> groupByTime = jogadores.stream().collect(
            Collectors.groupingBy(Jogador::getTimeAtual));
        System.out.println(groupByTime);
    }

    public void ordenarJogadoresGols(List<Jogador> jogadores) {
        jogadores.stream().sorted(Comparator.comparingInt(Jogador::getGolsMarcados).reversed()).forEach(System.out::println);
    }
}

```

O método imprimirJogadores()

O primeiro desses métodos é o **imprimirJogadores()**, declarado na linha 43. Note que ele cria uma stream e imprime todos os elementos utilizando a operação **forEach()**, que recebe como parâmetro o método **println()**. Dessa forma, o compilador sabe que ao iterar internamente pela stream, a cada passo sempre teremos um elemento do tipo **Jogador**, pois inicialmente foi criada uma stream do tipo **Stream<Jogador>**. A passagem do método **println()** indica a ação, ou seja, o que será feito sobre cada um dos elementos da lista. Nesse caso, será impressa a descrição textual de cada objeto através da chamada a **toString()**, método que deverá ser sobreescrito conforme expõe a linha 13 da **Listagem 12**.

O método imprimirJogadoresTime() Na linha 47, o método **imprimirJogadoresTime()** recebe como parâmetro, além da lista de jogadores, o nome de um time, e imprime apenas os jogadores associados a ele. Para isso, o filtro realizado sobre a stream inicial permite recuperar apenas os jogadores do time desejado, graças à condição indicada pela expressão *lambda jogador -> jogador.getTimeAtual().equals(time)*, passada como parâmetro para o método **filter()**. A partir da nova stream é realizada uma iteração sobre os elementos com o objetivo de imprimir uma representação textual de cada objeto. Com proposta semelhante, o método **imprimirJogadoresTimeGol()**, declarado na linha 51, imprime os nomes dos jogadores de um dado time, porém, considerando somente aqueles que marcaram mais de dez gols.

O método calcularMediaIdade() Na linha 60, o método **calcularMediaIdade()** calcula a média de idade de todos os jogadores cadastrados. Com isso em mente, no fluxo de processamento da linha 61, primeiramente criamos uma stream a partir da lista de jogadores. Em seguida, chamamos o método **mapToInt()**, que produz uma stream de inteiros cujos elementos correspondem à idade dos atletas. Por fim, é invocado o método **average()**. Como essa operação não retorna um valor numérico, e sim um objeto da classe **Optional**, declaramos o método **getAsDouble()** para recuperar o resultado.

O método imprimirJogadorMaisVelho() Outro método implementado foi o **imprimirJogadorMaisVelho()** – veja a linha 64. Para alcançar esse requisito utilizamos a operação **max()**, que para saber o que avaliar recebe **Comparator.comparingInt()** como parâmetro. Esse método, por sua vez, aceita como parâmetro uma função, **Jogador::getIdade**, que extrai um valor chave, nesse caso a idade dos jogadores, e o utiliza como critério de classificação para comparar os elementos da stream.

Aproveitando essa lógica, criamos o método **imprimirJogadorMaisNovo()** – veja a linha 69. A diferença em relação ao método anterior está na substituição do método **max()** pelo **min()**, para retornar o menor valor.

O método imprimirSomatorioGols() Para calcular a quantidade de gols marcados, implementamos o método **imprimirSomatorioGols()**, iniciado na linha 80. Com esse objetivo, após obter a stream inicial, realizamos uma chamada a **mapInt()** para produzir uma nova stream contendo apenas os valores inteiros que representam os gols. Em seguida, chamamos o método **sum()** para contabilizar os valores e retornar o resultado.

O método agruparJogadoresPeloTime() Outra operação implementada pode ser verificada a partir da linha 85. Trata-se do método **agruparJogadoresPeloTime()**. Sua função é agrupar os jogadores levando em conta o time em que eles atuam. Isso é feito utilizando **Collector.groupingBy()**, factory de Collectors que agrupa os elementos de uma stream de acordo com uma função classificadora. Nesse caso a função empregada foi **Jogador::getTimeAtual()**. O retorno é uma instância da interface **Map**.

O método ordenarJogadoresGols() Por fim, a partir da linha 91 pode ser visto o método **ordenarJogadoresGols()**, que organiza a stream adotando como critério o número de gols marcados, em ordem decrescente (observe o método **reverse()**). Para tanto, da stream inicial é chamado o método **sorted()**, que recebe um **Comparator** com o critério de classificação para realizar a operação.

Codificando a classe principal Neste momento, vamos criar uma classe para executar e testar os vários métodos que implementamos. Sendo assim, no pacote **br.com.cadastro.main**, crie a classe de nome **Principal**, com o código indicado na **Listagem 14**. **Listagem 14**. Implementação da classe Principal.

Copy

```

package br.com.cadastro.main;

//imports omitidos

public class Principal {
    public static void main(String[] args) {
        Principal p = new Principal();
        JogadorImpl jogImpl = new JogadorImpl();
        String enderecoDir = "C:\\\\Users\\\\carlosalberto\\\\Desktop\\\\stream";
        String nomeArquivo = "jogadores.txt";
        List<Jogador> listaDeJogadores = jogImpl.getListaDeJogadores(Paths.get(enderecoDir + "\\\\" + nomeArqui
        if (!jogImpl.verificarArquivoExiste(Paths.get(enderecoDir))){
            System.out.println("Arquivo não encontrado");
        }
        else
        {
            jogImpl.imprimirJogadorArtilheiro(listaDeJogadores);
            jogImpl.imprimirJogadorMaisVelho(listaDeJogadores);
            jogImpl.imprimirJogadorMaisNovo(listaDeJogadores);
        }
    }
}

```

No método **main()**, linhas 9 e 10, definimos, respectivamente, o diretório onde o arquivo com os dados se encontra e o nome desse arquivo. Em seguida, após recuperar a lista de jogadores, invocamos alguns dos métodos da classe **JogadorImpl**, a saber: **imprimirJogadorArtilheiro()**, **imprimirJogadorMaisVelho()** e **imprimirJogadorMaisNovo()**. A **Figura 6** mostra o resultado da execução dessa classe.

```

run:
Jogador Artilheiro: Ricardo Oliveira
Jogador mais velho: Rogério Ceni
Jogador mais novo: Malcom
CONSTRUÍDO COM SUCESSO (tempo total: 0 segundos)

```

Figura 6.Resultado da execução da classe Principal.

Nessa aplicação utilizamos apenas uma pequena amostra da grande quantidade de informações que é produzida ao longo de um campeonato. Caso a situação fosse diferente, mais processamento seria necessário, levando o desenvolvedor a se preocupar com fatores relacionados à performance. Nesse momento, uma boa opção é o emprego do paralelismo, lançando mão dos múltiplos núcleos que os processadores modernos oferecem.

Pensando nisso, a Streams API também foi projetada para tirar proveito do processamento paralelo. Assim, de forma simples, basta o desenvolvedor, ao invés de criar uma stream por meio da chamada ao método **stream()**, solicitar a criação da mesma por meio da chamada ao método **parallelStream()**. Isso possibilitará executar as operações de forma concorrente.

Apesar dessa facilidade, no entanto, é preciso ter cautela e não começar a solucionar todos os problemas de forma paralela, pois isso tem um custo (*overhead*), decorrente do processamento de tarefas adicionais geradas pela paralelização. Portanto, se não houver uma grande quantidade de elementos a serem manipulados, o overhead do processamento natural do paralelismo provavelmente não irá compensar sua adoção.

Note que essa nova forma de escrever código é bem diferente da maneira de processar coleções tradicional, sobretudo devido à incorporação de conceitos oriundos do paradigma funcional, trazendo facilidade em tarefas antes complexas e possibilitando um número menor de linhas de código. Além disso, algumas técnicas ajudam a otimizar o processamento dos dados, como o “processamento preguiçoso”, que faz com que a execução das operações seja realizada apenas quando existir a real necessidade de se obter o resultado, e as operações de curto-circuito (*short-circuiting*), que encurtam o processamento da stream ao trabalhar apenas com a parte necessária para obtenção do resultado.

Por fim, saiba que a introdução de streams na API do Java é um grande avanço e dominar essa nova solução é tão importante quanto conhecer e dominar os conceitos por trás da infraestrutura de Collections.

Links Endereço para download do Java 8.

[www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html\[3\]](http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html[3]) Endereço para download do NetBeans 8.1.
netbeans.org/downloads/ Conteúdo sobre a Streams API.

www.oracle.com/technetwork/pt/articles/java/processamento-streams-java-se-8-2763688-ptb.html Tutorial sobre Java 8.
winterbe.com/posts/2014/07/31/java8-stream-tutorial-examples/ Conteúdo sobre Expressões Lambda e Interfaces Funcionais.
rodrigouchoa.wordpress.com/2014/05/20/novidades-do-java-8-lambda-expressions/ Documentação da Streams API

Livros Paulo Silveira, Rodrigo Turini. Livro Java 8 Prático Lambdas, Streams e os novos recursos da linguagem.

Carlos Silva é Formado em Ciéncia da Computaçao pela Universidade Federal de Uberlândia (UFU), com especializaçao em Desenvolvimento Java pelo Centro Universitário do Triângulo (UNITRI). Trabalha na empresa Algar Telecom como Analista de Sistemas e atualmente é aluno do curso de especializaçao em Análise e Desenvolvimento de Sistemas Aplicados a Gestão Empresarial no Instituto Federal do Triângulo Mineiro (IFTM). Possui as seguintes certificações: OCJP, OCWCD e ITIL.

Este artigo foi revisado pela equipe de produtos Oracle e está em conformidade com as normas e práticas para o uso de produtos Oracle.

Recursos para	Por que a Oracle	Aprendizado	Novidades	Entre em contato
Carreiras	Relatórios de Analistas	O que é computação em nuvem?	Notícias	Vendas: 0800-891-4433
Desenvolvedores	Gartner MQ para Cloud ERP	O que é CRM?	Oracle CloudWorld	Como podemos ajudar?
Investidores	Economia na Nuvem	O que é Docker?	Oracle e Premier League	Inscreve-se para receber emails
Parceiros	Responsabilidade Corporativa	O que é Kubernetes?	Oracle Red Bull Racing	Eventos
Pesquisadores	Diversidade e Inclusão	O que é Python?	Oracle Sustainability	Blogs
Alunos e Educadores	Práticas de segurança	O que é SaaS?	Plataforma de experiência do funcionário	