**Novo default method forEach**

O novo método forEach recebe como parâmetro a interface Consumer, que possui um único método abstrato chamado accept.

class ImprimeNaLinha implements Consumer<String>{

@Override

public void accept(String s) {

System.out.println(s);

}

}

Consumer<String> consumidor = new ImprimeNaLinha();

palavras.forEach(consumidor);

**Ordenando com o método sort**

Usando o default method sort, existente na própria lista e um new Comparator para o palavras.sort() antes de fazer o seu forEach.

class ComparadorPorTamanho implements Comparator<String> {

@Override

public int compare(String s1, String s2) {

if (s1.length() < s2.length())

return -1;

if (s1.length() > s2.length())

return 1;

return 0;

}

}

ComparadorPorTamanho comparador = new ComparadorPorTamanho();

palavras.sort(comparador);

System.out.println(palavras);

**Vantagens dos default methods**

A grande vantagem é que agora uma interface pode evoluir sem quebrar compatibilidade.

**Código mais sucinto com Method references**

Com os lambdas e métodos default, conseguimos escrever a ordenação das Strings de uma forma bem mais sucinta:

palavras.sort((s1, s2) -> Integer.compare(s1.length(), s2.length()));

**Métodos default em Comparator**

Há vários métodos auxiliares no Java 8. Até em interfaces como o Comparator. E você pode ter um método default que é estático. Esse é o caso do Comparator.comparing, que é uma fábrica, uma factory, de Comparator. Passamos o lambda para dizer qual será o critério de comparação desse Comparator, repare:

palavras.sort(Comparator.comparing(s -> s.length()));

Veja a expressividade da linha, está escrito algo como "palavras ordene comparando s.length". Podemos quebrar em duas linhas para ver o que esse novo método faz exatamente:

Comparator<String> comparador = Comparator.comparing(s -> s.length());

palavras.sort(comparador);

Dizemos que Comparator.comparing recebe um lambda, mas essa é uma expressão do dia a dia. Na verdade, ela recebe uma instância de uma interface funcional. No caso é a interface [Function](http://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html" \t "_blank) que tem apenas um método, o apply. Para utilizarmos o Comparator.comparing, nem precisamos ficar decorando os tipos e assinatura do método dessas interfaces funcionais. Essa é uma vantagem dos lambdas. Você também vai acabar programando dessa forma. É claro que, com o tempo, você vai conhecer melhor as funções do pacote java.util.functions. Vamos quebrar o código mais um pouco. Não se esqueça de dar os devidos imports.

Function<String, Integer> funcao = s -> s.length();

Comparator<String> comparador = Comparator.comparing(funcao);

palavras.sort(comparador);

A interface Function vai nos ajudar a passar um objeto para o Comparator.comparing que diz qual será a informação que queremos usar como critério de comparação. Ela recebe dois tipos genéricos. No nosso caso, recebe uma String, que é o tipo que queremos comparar, e um Integer, que é o que queremos extrair dessa string para usar como critério. Poderia até mesmo criar uma classe anônima para implementar essa Function e seu método apply, sem utilizar nenhum lambda. O código ficaria grande e tedioso.

Quisemos quebrar em três linhas para que você pudesse enxergar o que ocorre por trás exatamente. Sem dúvida o palavras.sort(Comparator.comparing(s -> s.length())) é mais fácil de ler. Dá para melhorar ainda mais? Sim!

**Method reference**

É muito comum escrevermos lambdas curtos, que simplesmente invocam um único método. É o exemplo de s -> s.length(). Dada uma String, invoque e retorne o método length. Por esse motivo, há uma forma de escrever esse tipo de lambda de uma forma ainda mais reduzida. Em vez de fazer:

palavras.sort(Comparator.comparing(s -> s.length()));

Fazemos uma referência ao método (method reference):

palavras.sort(Comparator.comparing(String::length));

São equivalentes nesse caso! Sim, é estranho ver String::length e dizer que é equivalente a um lambda, pois não há nem a -> e nem os parênteses de invocação ao método. Por isso é chamado de method reference. Ela pode ficar ainda mais curta com o import static:

import static java.util.Comparator.\*;

palavras.sort(comparing(String::length));

Vamos ver melhor a semelhança entre um lambda e seu method reference equivalente. Veja as duas declarações a seguir:

Function<String, Integer> funcao1 = s -> s.length();

Function<String, Integer> funcao2 = String::length;

Elas ambas geram a mesma função: dada um String, invoca o método length e devolve este Integer. As duas serão avaliadas/resolvidas (*evaluated*) para Functions equivalentes.

Quer um outro exemplo? Vejamos o nosso forEach, que recebe um Consumer:

palavras.forEach(s -> System.out.println(s));

Dada uma String, invoque o System.out.println passando-a como argumento. É possível usar method reference aqui também! Queremos invocar o println de System.out:

palavras.forEach(System.out::println);

Novamente pode parecer estranho. Não há os parênteses, não há a flechinha (->), nem os argumentos que o Consumer recebe. Fica tudo implícito. Dessa vez, o argumento recebido (isso é, cada palavra dentro da lista palavras), não será a variável onde o método será invocado. O Java 8 consegue perceber que tem um println que recebe objetos, e invocará esse método, passando a String da vez.

Quando usar lambda e quando usar method reference? Algumas vezes não é possível usar method references. Se você tiver, por exemplo, um lambda que dada uma String, pega os 5 primeiros caracteres, faríamos s -> s.substring(0, 5). Esse lambda não pode ser escrito como method reference! Pois não é uma simples invocação de métodos onde os parâmetros são os mesmos que os do lambda.

**Streams: trabalhando melhor com coleções**

Para que a gente possa elaborar exemplos mais reais e interessantes, vamos criar uma simples classe que representa um curso do Alura. Com ela, vamos refazer alguns passos dos últimos capítulos do curso, de forma mais rápida, a fim de revê-los. Ela terá apenas dois atributos: seu nome e a quantidade de alunos, além de um construtor e seus respectivos getters:

class Curso {

private String nome;

private int alunos;

public Curso(String nome, int alunos) {

this.nome = nome;

this.alunos = alunos;

}

public String getNome() {

return nome;

}

public int getAlunos() {

return alunos;

}

}

Em uma classe com o main, criamos alguns cursos e inserimos em uma lista:

List<Curso> cursos = new ArrayList<Curso>();

cursos.add(new Curso("Python", 45));

cursos.add(new Curso("JavaScript", 150));

cursos.add(new Curso("Java 8", 113));

cursos.add(new Curso("C", 55));

Se quisermos ordenar esses cursos pela quantidade de alunos, podemos utilizar o Comparator da forma antiga, fazendo lista.sort(new Comparator<Curso>() { .... } e implementando nosso critério de comparação dentro da classe anônima.

E a forma nova, como fazemos? Podemos já utilizar o Comparator.comparing. E indicaremos que queremos que o getAlunos seja utilizado como critério de comparação:

cursos.sort(Comparator.comparingInt(c -> c.getAlunos()));

Esse é o caso que podemos usar um method reference para ficar ainda mais sucinto e legível:

cursos.sort(Comparator.comparingInt(Curso::getAlunos));

Pronto! Você já pode fazer um forEach e imprimir os nomes dos cursos para ver se o resultado é o esperado.

**Streams: trabalhando com coleções no java 8**

E se quisermos fazer outras tarefas com essa coleção de cursos? Por exemplo, filtrar apenas os cursos com mais de 100 alunos. Poderíamos fazer um loop que, dado o critério desejado seja atendido, adicionamos este curso em uma nova lista, a lista filtrada.

No Java 8, podemos fazer de uma forma muito mais interessante. Há como invocar um filter. Para sua surpresa, esse método não se encontra em List, nem em Collection, nem em nenhuma das interfaces já conhecidas. Ela está dentro de uma nova interface, a Stream. Você pode pegar um Stream de uma coleção simplesmente invocando cursos.stream():

Stream<Curso> streamDeCurso = cursos.stream();

O que fazemos com ele? O Stream devolvido por esse método tem uma dezena de métodos bastante úteis. O primeiro é o filter, que recebe um predicado (um critério), que deve devolver verdadeiro ou falso, dependendo se você deseja filtrá-lo ou não. Utilizaremos um lambda para isso:

Stream<Curso> streamDeCurso = cursos.stream().filter(c -> c.getAlunos() > 100);

Repare que o filtro devolve também um Stream! É um exemplo do que chamam de fluent interface. Vamos fazer um forEach e ver o resultado dos cursos:

Stream<Curso> streamDeCurso = cursos.stream().filter(c -> c.getAlunos() > 100);

cursos.forEach(c -> System.out.println(c.getNome()));

A saída será:

Python

C

Java 8

Java Script

Estranho. Filtramos apenas os que tem mais de 100 alunos, e ele acabou listando todos! Por quê? Pois **modificações em um stream não modificam a coleção/objeto que o gerou**. Tudo que é feito nesse fluxo de objetos, nesse Stream, não impacta, não tem efeitos colaterais na coleção original. A coleção original continua com os mesmos cursos!

Para imprimir os cursos filtrados, podemos usar o forEach que existe em Stream:

Stream<Curso> streamDeCurso = cursos.stream().filter(c -> c.getAlunos() > 100);

streamDeCurso.forEach(c -> System.out.println(c.getNome()));

Ou melhor ainda, podemos eliminar essa variável temporária, fazendo tudo em uma mesma linha:

cursos.stream().filter(c -> c.getAlunos() > 100).forEach(c -> System.out.println(c.getNome()));

Por uma questão de legibilidade, vamos espaçar esse código em algumas linhas, mas continuando um único statement:

cursos.stream()

.filter(c -> c.getAlunos() > 100)

.forEach(c -> System.out.println(c.getNome()));

Interessante não? E por que criaram uma nova interface e não colocar o método de filtrar dentro de List? Há vários motivos. Mas repare já em um primeiro: numa coleção tradicional, o que o filter faria? Alteraria a coleção em questão, ou manteria intacta, devolvendo uma nova coleção? Coleções no java podem ser mutáveis e imutáveis, o que complicaria ler esses métodos. No Stream, sabemos que esses métodos nunca alterarão a coleção original.

Vamos além. Vamos ver as outras funcionalidades. E se quisermos, dados esses cursos filtrados no nosso fluxo (Stream) de objetos, um novo fluxo apenas com a quantidade de alunos de cada um deles? Utilizamos o map:

cursos.stream()

.filter(c -> c.getAlunos() > 100)

.map(c -> c.getAlunos());

Se você reparar, esse map não devolve um Stream<Curso>, e sim um Stream<Integer>! Faz sentido. Podemos concatenar a invocação ao forEach para imprimirmos os dados:

cursos.stream()

.filter(c -> c.getAlunos() > 100)

.map(c -> c.getAlunos())

.forEach(x -> System.out.println(x));

Aproveitamos para recapitular o que já vimos: temos a oportunidade de usar o recurso de method references duas vezes. Tanto pra invocação de getAlunos quanto a do println. Vamos alterar:

cursos.stream()

.filter(c -> c.getAlunos() > 100)

.map(Curso::getAlunos)

.forEach(System.out::println);

O lambda passado para o filter não pode ser representado como um method reference, pois não é uma simples invocação de um único método: ele compara com um número. Pode ser que, no começo, você prefira os lambdas, pela sintaxe ser utilizada mais frequentemente. Com o tempo, verá que o method reference não impõe problema de legibilidade algum, muito pelo contrário.

Outro ponto que podemos notar: nem vimos qual é o tipo de interface que Map recebe! É uma Function, mas repare que usamos o lambda e nem foi necessário conhecer a fundo quais eram os parâmetros que ele recebia. Foi natural. É claro que, com o tempo, é importante que você domine essa nova API, mesmo que acabe utilizando majoritariamente as interfaces como lambdas.

**Streams primitivos**

Trabalhar com Streams vai ser frequente no seu dia a dia. Há um cuidado a ser tomado: com os tipos primitivos. Quando fizemos o map(Curso::getAlunos), recebemos de volta um Stream<Integer>, que acaba fazendo o autoboxing dos ints. Isto é, utilizará mais recursos da JVM. Claro que, se sua coleção é pequena, o impacto será irrisório. Mas é possível trabalhar só com ints, invocando o método mapToInt:

IntStream stream = cursos.stream()

.filter(c -> c.getAlunos() > 100)

.mapToInt(Curso::getAlunos);

Ele devolve um IntStream, que não vai gerar autoboxing e possui novos métodos específicos para trabalhar com inteiros. Um exemplo? A soma:

int soma = cursos.stream()

.filter(c -> c.getAlunos() > 100)

.mapToInt(Curso::getAlunos)

.sum()

Em uma única linha de código, pegamos todos os cursos, filtramos os que tem mais de 100 e somamos todos os alunos. Há também versões para double e long de Streams primitivos. Até mesmo o Comparator.comparing possui versões como Comparator.comparingInt, que recebe uma IntFunction e não necessita do boxing. Em outras palavras, todas as interfaces funcionais do novo pacote java.util.functions possuem versões desses tipos primitivos.

<https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html>

**Mais Streams, Collectors e APIs**

Os Streams possibilitam trabalhar com dados de uma maneira funcional. Normalmente, são dados e objetos que vêm de uma collection do Java. Por que não adicionaram esses métodos diretamente nas Collections? Justo para não ser dependente delas, não ter efeitos colaterais e não entupir de métodos as interfaces.

Vamos conhecer outros métodos interessantes dos Streams. Um exemplo seria: quero um curso que tenha mais de 100 alunos! Pode ser qualquer um deles. Há o método findAny

cursos.stream()

.filter(c -> c.getAlunos() > 100)

.findAny();

O que será que devolve o findAny? Um Curso? Não! Um Optional<Curso>.

# Optional

Optional é uma importante nova classe do Java 8. É com ele que poderemos trabalhar de uma maneira mais organizada com possíveis valores null. Em vez de ficar comparando if(algumaCoisa == null), o Optional já fornece uma série de métodos para nos ajudar nessas situações. Por que o findAny utiliza esse recurso? Pois pode não haver nenhum curso com mais de 100 alunos! Nesse caso, o que seria retornado? null? uma exception?

Vamos ver as vantagens de se trabalhar com Optional. Primeiro vamos atribuir o resultado do findAny a uma variável:

Optional<Curso> optional = cursos.stream()

.filter(c -> c.getAlunos() > 100)

.findAny();

Dado um Optional, podemos pegar seu conteúdo invocando o get. Ele vai devolver o Curso que queremos. Mas e se não houver nenhum? Uma exception será lançada.

Curso curso = optional.get();

Há métodos mais interessantes. O orElse diz que ele deve devolver o curso que existe dentro desse optional, ou então o que foi passado como argumento:

Curso curso = optional.orElse(null);

Nesse caso ou ele devolve o curso encontrado, ou null, caso nenhum seja encontrado. Mesmo assim, ainda não está tão interessante. Há como evitar tanto o null, quanto as exceptions, quanto os ifs. O método ifPresent executa um lambda (um Consumer) no caso de existir um curso dentro daquele optional:

optional.ifPresent(c -> System.out.println(c.getNome()));

Claro que, no dia a dia, não teríamos a variável temporária curso. Podemos fazer isso

cursos.stream()

.filter(c -> c.getAlunos() > 100)

.findAny()

.ifPresent(c -> System.out.println(c.getNome()));

Outros métodos devolvem Optional nos Streams. Um deles é o average em IntStream. Por que? Pois pode não existir nenhum element, e aí a média poderia realizar uma divisão por zero.

Você vai encontrar Optional não somente na API de Streams. Vale a pena conhecer e utilizá-la no seu próprio código e entidades.

# Gerando uma coleção a partir de um Stream

Invocar métodos no stream de uma coleção não altera o conteúdo da coleção original. Ele não gera efeitos colaterais. Como então obter uma coleção depois de alterar um Stream?

Tentar fazer List<Curso> novaLista = lista.stream().filter(...) não compila, pois um Stream **não é** uma coleção. Para fazer algo parecido com isso, utilizamos o método collect, que coleta elementos de um Stream para produzir um outro objeto, como uma coleção.

O método Collect recebe um Collector, uma interface não tão trivial de se implementar. Podemos usar a classe Collectors (repare o s no final), cheio de factory methods que ajudam na criação de coletores. Um dos coletores mais utilizados é o retornado por Collectors.toList():

List<Curso> resultados = cursos.stream()

.filter(c -> c.getAlunos() > 100)

.collect(Collectors.toList());

Pronto! É através dos coletores que podemos "retornar" de um Stream para uma Collection. Certamente poderia ter usado a mesma variável, a List<Curso> cursos que temos:

cursos = cursos.stream()

.filter(c -> c.getAlunos() > 100)

.collect(Collectors.toList());

Pronto. Alteramos a referência antiga para apontar para essa nova coleção, depois de filtrada!

Um exemplo mais complicado? Podemos gerar mapas! Queremos um mapa que, dado o nome do curso, o valor atrelado é a quantidade alunos. Um Map<String, Integer>. Utilizamos o Collectors.toMap. Ele recebe duas Functions. A primeira indica o que vai ser a chave, e a segunda o que será o valor:

Map mapa = cursos

.stream()

.filter(c -> c.getAlunos() > 100)

.collect(Collectors.toMap(c -> c.getNome(), c -> c.getAlunos()));

# Outras vantagens do Stream

Os Streams foram desenhados de uma forma a tirar proveito da programação funcional. Se você utilizá-los da forma que vimos por aqui, eles nunca gerarão efeitos colaterais. Isso é, apenas o stream será alterado, e nenhum outro objeto será impactado.

Dada essa premissa, podemos pedir para que nosso stream seja processado em paralelo. Ele mesmo vai decidir quantas threads usar e fazer todo o trabalho, utilizando APIs mais complicadas (como a de fork join) para ganhar performance. Para fazer isso, basta utilizar parallelStream() em vez de stream()!

Tome cuidado. Para streams pequenos, o custo de cuidado dessas threads e manipular os dados entre elas é alto e pode ser bem mais lento que o Stream tradicional.

**A nova API de datas**

Vamos começar criando uma data, a data de hoje. Para representar uma data em java agora eu posso utilizar a classe LocalDate, presente no pacote java.time. Repare como é facil ter a data atual utilizando o método now():

public class Datas {

public static void main(String[] args) {

LocalDate hoje = LocalDate.now();

System.out.println(hoje);

}

}

Assim como este existem diversos outros métodos estáticos nas novas classes da API de datas. Vamos conhecer vários no decorrer do capítulo.

Repare que o valor impresso neste caso será 2014-05-28, logo veremos como formatar essa saída de outras formas.

# Conhecendo mais da API

Vamos agora criar uma nova data para representar as Olimpíadas do Rio, por exemplo. Para fazer isso podemos utilizar o método of passando o dia, mês e ano:

LocalDate olimpiadasRio = LocalDate.of(2016, Month.JUNE, 5);

Repare que existe uma enumeração pra representar o mês, mas se você preferir pode utilizar sim um numero inteiro.

Vamos agora calcular a diferença de anos entre essas duas datas. Uma forma de fazer isso na mão seria subtraindo o método getYear das datas, algo como:

int anos = olimpiadasRio.getYear() - hoje.getYear();

System.out.println(anos);

# Trabalhando com Period

Ao executar esse código temos o resultado esperado, que neste caso é 2 anos. Mas e se quisessemos descobrir a diferença de dias e meses também? Daria pra fazer da mesma forma, mas sempre que você tiver que fazer um trabalho dessa na mão você pode ter certeza que já existe algo pronto pra te ajudar de alguma forma. Nesse caso podemos utilizar a classe Period.

Para saber a diferença entre duas datas podemos utilizar seu método between, da seguinte forma:

Period periodo = Period.between(hoje, olimpiadasRio);

System.out.println(periodo);

Repare que a saída desse nosso println vai ser um pouco estranha, um exemplo seria: P2Y8D.

Isso significa um período de 2 anos e 8 dias. Mas claro, poderíamos imprimir apenas as propriedades, da segunite forma:

Period periodo = Period.between(hoje, olimpiadasRio);

System.out.println(periodo.getYears());

System.out.println(periodo.getMonths());

System.out.println(periodo.getDays());

Assim como esses existem diversos getters pra passar informações importantes a respeito desse período.

# Incrementando e decrementando suas datas

Outra coisa bem comum em nosso dia a dia é quando queremos saber o dia anterior ou posterior a uma data. Por exemplo como saber qual a data de amanhã? Há diversos métodos pra nos ajudar com isso, vamos encontrar na API diversos métodos minus ou plus pras diferentes unidades de tempo, como por exemplo:

System.out.println(hoje.minusYears(1));

System.out.println(hoje.minusMonths(4));

System.out.println(hoje.minusDays(2));

System.out.println(hoje.plusYears(1));

System.out.println(hoje.plusMonths(4));

System.out.println(hoje.plusDays(2));

Ou seja, pra saber a data de amanhã bastaria fazer hoje.plusDays(1).

# Uma API imutável

Sabendo disso podemos escrever o seguinte código para incrementar 4 anos na data atual, para saber quando será a próxima Olimpíada, por exemplo.

olimpiadasRio.plusYears(4);

System.out.println(olimpiadasRio);

Mas repare que a saída desse código ainda será a data atual. Porque isso ocorreu? Da mesma forma que as novas API's, como o Stream, os métodos da API de datas sempre vão retornar uma nova instancia da sua data. Portanto precisamos fazer algo como:

LocalDate proximasOlimpiadas = olimpiadasRio.plusYears(4);

System.out.println(proximasOlimpiadas);

Ou seja, toda a API de datas é imutável. Ela nunca vai alterar a data original.

Ao executar esse código, um exemplo da saída seria 2020-06-25.

Mas note que não é esse o formato que estamos acostumados a trabalhar, podemos então trabalhar com os diversos formatadores de datas existentes.

# Formatando suas datas

Para formatar nossas datas podemos utilizar o DateTimeFormatter. Existem diversos já prontos, mas há ainda a alternativa de você criar o seu próprio formatador no padrão já conhecido de dd/MM/yyyy.

Para fazer isso basta você utilizar o método ofPattern:

DateTimeFormatter formatador = DateTimeFormatter.ofPattern("dd/MM/yyyy");

Agora podemos a partir da nossa data, neste caso proximasOlimpiadas, chamar o método format passando esse formatador:

String valorFormatado = proximasOlimpiadas.format(formatador);

System.out.println(valorFormatado);

Agora sim, ao executar temos o resultado 05/06/2020

# Trabalhando com medida de tempo

Por enquanto só estamos trabalhando com datas, fazendo formatações e manipulando seu resultado. Mas é muito comum eu também precisar trabalhar com horas, minutos e segundos. Ou seja, trabalhar com uma medida de data com tempo.

Para isso podemos utilizar a classe LocalDateTime, de forma bem similar podemos fazer:

LocalDateTime agora = LocalDateTime.now();

Podemos criar um novo formatador para mostrar as horas, minutos e segundos para conseguirmos ver o resultado já formatado:

DateTimeFormatter formatadorComHoras = DateTimeFormatter.ofPattern("dd/MM/yyyy hh:mm:ss");

LocalDateTime agora = LocalDateTime.now();

System.out.println(agora.format(formatadorComHoras));

Pronto! Agora o resultado será algo como 05/06/2014 12:24:10.

# Lidando com modelos mais específicos

É muito comum ignorarmos valores quando precisamos apenas de algumas medidas de tempo, como por exemplo ano e mês. Nessa caso no lugar de criarmos um LocalDate ou algo assim e ignorar o seu valor de dia, podemos trabalhar com os modelos mais específicos da nova API.

Neste exemplo podemos usar o YearMonth, da seguinte forma:

YearMonth anoEMes = YearMonth.of(2015, Month.JANUARY);

Ou seja, existem diversas novas classes para expressar bem nossas intenções.

Outro exemplo, para trabalharmos apenas com tempo podemos utilizar o LocalTime. Representar o horario do nosso intervalo de almoço, por exemplo, poderia ser feito com:

LocalTime intervalo = LocalTime.of(12, 30);

System.out.println(intervalo);

Nesse caso a saída seria exatamente 12:30. Não quer trabalhar com esse formato? Tudo bem, você pode criar um formatador como nós já fizemos, contanto que ele só tenha as medidas de tempo que existem, que neste caso são hora e minuto. Caso contrário uma exception será lançada.