**Java - Trabalhando com lambdas, streams e Spring Framework**

**Entendendo Maven e Gradle**

Para entender a diferença entre **Maven** e **Gradle**, precisamos primeiro compreender o que eles são e para que servem. Maven e Gradle ***são ferramentas de automação de compilação e gerenciamento de dependências muito populares na comunidade Java***. Eles ajudam a simplificar e organizar o processo de construção, teste e implantação de projetos Java, tornando o desenvolvimento mais eficiente.

**O que é Maven?**

Principais conceitos do Maven:

* **Dependency Management**: O Maven é ótimo para gerenciar as dependências de um projeto. As dependências são declaradas no **pom.xml** e o Maven é responsável por baixá-las automaticamente do repositório central do Maven ou de repositórios personalizados.
* **Convenção sobre Configuração**: O Maven segue um conjunto de convenções, o que significa que muitas configurações são predefinidas e podem ser usadas fora da caixa sem a necessidade de configuração adicional.
* **Ciclo de Vida Padrão**: O Maven possui um ciclo de vida padrão com diferentes fases (por exemplo, compile, test, package, install, deploy), e cada fase executa uma série de metas. O ciclo de vida padrão permite que você construa e teste seu projeto facilmente usando comandos como **mvn compile**, **mvn test** e **mvn package**.
* **Central Repository**: O Maven possui um repositório central que contém uma grande variedade de bibliotecas Java prontas para serem utilizadas. Esse repositório pode ser visualizado nesse [link](https://mvnrepository.com/).

**O que é Gradle?**

O Gradle é outra ferramenta de construção e automação de projetos Java que ganhou popularidade ao longo dos anos. Ele usa uma linguagem de domínio específico (DSL) baseada em Groovy ou Kotlin para definir a estrutura do projeto e as tarefas de construção.

Principais conceitos do Gradle:

* **Flexibilidade**: O Gradle é altamente flexível e permite que você defina suas próprias tarefas de construção e configure o processo de construção de acordo com suas necessidades.
* **Build by Convention**: O Gradle também segue algumas convenções, mas oferece mais liberdade do que o Maven na forma como você organiza e configura o projeto.
* **Dependency Management**: Assim como o Maven, o Gradle também gerencia as dependências do projeto e pode baixá-las de repositórios remotos.
* **Incremental Builds**: O Gradle é projetado para realizar compilações incrementais, o que significa que ele pode construir apenas as partes do projeto que foram alteradas desde a última compilação, tornando o processo mais rápido.

Agora que temos uma compreensão básica do que são Maven e Gradle, vamos ver suas semelhanças, diferenças, vantagens e desvantagens.

**Semelhanças**

Tanto o Maven quanto o Gradle fornecem convenções para a estrutura do diretório do projeto, dependência de gerenciamento e plugins de construção. Eles também são amplamente suportados em IDEs e ferramentas de Integração Contínua (CI).

**Diferenças**

A principal diferença entre Maven e Gradle é a maneira como eles gerenciam as dependências e como eles descrevem a lógica de construção. Maven usa arquivos XML para gerenciar as dependências e descreve a lógica de construção usando plugins, enquanto Gradle usa um formato de script e descreve a lógica de construção como código.

**Vantagens e Desvantagens**

Maven é fácil de aprender e tem um grande ecossistema. A desvantagem é que os arquivos XML podem se tornar muito grandes e difíceis de gerenciar para projetos complexos.

Gradle, por outro lado, permite scripts de construção mais poderosos e é mais flexível. No entanto, é mais difícil de aprender e o ecossistema ainda não é tão grande quanto o do Maven.

A escolha entre Maven e Gradle depende do seu projeto e preferências. Ambas as ferramentas são poderosas e amplamente utilizadas, e ambas têm vantagens e desvantagens. O Maven é mais rígido e segue uma abordagem "opinião sobre configuração", o que pode ser uma vantagem para projetos menores e mais simples. Já o Gradle é mais flexível e personalizável, sendo uma escolha popular para projetos maiores e complexos que requerem configurações específicas.

Em resumo, tanto Maven quanto Gradle podem ser usados com eficiência para projetos Java, e a escolha dependerá do contexto e das preferências da equipe de desenvolvimento.

**A interface CommandLineRunner**

A interface **CommandLineRunner** é um recurso poderoso dentro do universo do Spring Framework, amplamente utilizado no desenvolvimento de aplicações Java. Ela permite que executemos alguma ação logo após a inicialização de nossa aplicação. Pode ser muito útil, por exemplo, se quisermos carregar alguns dados em nosso banco de dados logo na inicialização de nossa aplicação.

**Como funciona?**

Quando uma aplicação Spring Boot é lançada pode ocorrer várias operações automáticas, como a criação de beans, configuração de banco de dados, entre outros. A abertura para customização destas operações é limitada, e é aqui que a interface **CommandLineRunner** entra em cena.

A interface **CommandLineRunner** representa uma tarefa a ser executada após a inicialização do Spring Boot, ou seja, permite definir código para ser executado automaticamente quando o aplicativo é iniciado.

**Como Usar?**

A utilização é bem simples. Na própria classe principal da aplicação, podemos incluir para que se implemente a interface **CommandLineRunner**. Veja um exemplo:

@SpringBootApplication

**public** **class** **MyCommandLineRunner** **implements** **CommandLineRunner** {

@Override

**public** **void** **run**(String... args) **throws** Exception {

System.out.println("Olá, Mundo!");

}

}

Note que no exemplo acima, criamos uma classe chamada **MyCommandLineRunner** que implementa a interface **CommandLineRunner**. No método "**run()**" inserimos a ação que desejamos que seja executada logo depois que a aplicação for iniciada, nesse caso, apenas será impressa a mensagem: "Olá, Mundo!".

**Quando Usar?**

A interface **CommandLineRunner** é bem versátil e pode ser usada em diversas situações. Conforme mencionado anteriormente, ela pode ser usada para carregar dados para um banco de dados. Também pode ser usada para iniciar recursos, como conexões de rede, e para checar a integridade de determinados componentes ou serviços com os quais a aplicação irá interagir.

É importante lembrar que a **CommandLineRunner** é executada apenas na inicialização da aplicação, então não deve ser utilizada para tarefas que precisam ocorrer periodicamente durante o funcionamento da aplicação, para isso, Spring oferece outras ferramentas que serão mais adequadas.

**Motivação**

Vamos imaginar uma situação em que temos que carregar uma grande quantidade de dados em nosso banco de dados assim que nossa aplicação Spring iniciar. Bem, manualmente isso seria desafiador e demorado, no entanto, a interface **CommandLineRunner** torna essa tarefa extremamente mais simples.

Ao aprofundar seu conhecimento no Spring, você terá diversas opções para otimizar suas aplicações e tornar seu código mais limpo e eficaz. O Spring é um framework que facilita o desenvolvimento de aplicações em Java. Ele oferece um modelo de programação abrangente e simplificado, escondendo muitos dos detalhes de baixo nível. Como resultado, você pode se concentrar em escrever o seu código, sem se preocupar com uma infinidade de detalhes técnicos.

**Código para consumir API**

No próximo passo, iremos consumir uma API para busca de dados de séries. Para tal, criaremos a classe chamada **ConsumoAPI**, dentro de um pacote chamado **service**. Nele teremos um método chamado **ObterDados**, que devolve uma String com o json que corresponde à resposta da requisição.

Como é um procedimento muito similar ao que foi feito no último curso da formação Java Orientação a Objetos, vamos deixar aqui já disponibilizado para que você copie o código desse método e cole em sua classe, a fim de agilizar os estudos. Mas caso você queira voltar na formação anterior para rever a explicação, [clique para acessar o link da aula](https://cursos.alura.com.br/course/java-consumindo-api-gravando-arquivos-lidando-erros/task/126731).

**public** **String** **obterDados**(String endereco) {

**HttpClient** client = **HttpClient**.**newHttpClient**();

**HttpRequest** request = **HttpRequest**.**newBuilder**()

.**uri**(URI.**create**(endereco))

.**build**();

**HttpResponse**<**String**> response = **null**;

**try** {

response = client

.**send**(request, **HttpResponse**.BodyHandlers.**ofString**());

} **catch** (**IOException** e) {

**throw** **new** **RuntimeException**(e);

} **catch** (**InterruptedException** e) {

**throw** **new** **RuntimeException**(e);

}

**String** json = response.**body**();

**return** json;

}

Gotaria de compartilhar que recebi mais um certificado da trilha de formação **Java Developer** sobre testes unitários com **#JUnit**. Obrigado [**DIO**](https://www.linkedin.com/in/marcelodsoares/overlay/create-post/) pela oportunidade de aprendizado.

**Incluindo a dependência do Jackson no pom.xml**

Você está desenvolvendo um projeto Java que precisa manipular dados em formato JSON, e decidiu utilizar a biblioteca **Jackson** para realizar essa tarefa. No arquivo pom.xml, é necessário incluir a dependência do Jackson para que o projeto possa utilizá-lo. Com seria esta implementação no arquivo pom.xml?

<dependency>

<groupId>com.fasterxml.jackson.core</groupId>

<artifactId>jackson-databind</artifactId>

<version>2.15.2</version>

</dependency>

Essa é a maneira correta de incluir a dependência do Jackson no arquivo pom.xml.

**JsonAlias e JsonProperty**

**@JsonAlias** e **@JsonProperty** são anotações em **Jackson**, uma biblioteca Java para processar **JSON**, que ajudam a mapear propriedades de classe para campos JSON. No entanto, eles possuem funções diferentes. Vamos conhecer melhor cada uma dessas anotações?

**@JsonProperty**

Essa anotação é usada para definir o nome da propriedade JSON que está associada ao campo Java.

Quando o JSON é serializado, isto é, convertido de objetos Java para JSON, o nome especificado em @JsonProperty será usado como a chave para o campo no JSON de saída. Da mesma forma, quando o JSON é desserializado (convertido de JSON para objetos Java), a biblioteca procura pelo nome especificado em @JsonProperty para mapear o valor JSON para o campo Java.

Então, se você quiser usar um nome diferente no JSON, pode usar o @JsonProperty(“nomeNoJson”). Isso é útil, por exemplo, quando você precisa trabalhar com um JSON que segue uma convenção de nomenclatura diferente da sua classe Java.

Por exemplo, se você tiver uma classe Java com a propriedade "nomeCompleto" e quiser que ela seja mapeada para o JSON como "nome", você pode usar o @JsonProperty("nome") para especificar o nome correto no JSON:

**public** **class** **Pessoa** {

@JsonProperty("nome")

**private** String nomeCompleto;

}

**@JsonAlias**

O @JsonAlias é usado para definir um ou mais apelidos para o nome da propriedade JSON associada ao campo Java.

Ao desserializar, o @JsonAlias permite que a biblioteca encontre o valor JSON correspondente, mesmo que o nome da propriedade no JSON não corresponda exatamente ao nome do campo Java.

Isso é útil quando você está trabalhando com diferentes versões de um JSON ou quando deseja permitir que uma propriedade seja referenciada por nomes diferentes.

Por exemplo, se você tiver uma classe Java com a propriedade "nomeCompleto" e o JSON usa "nome" em vez disso, você pode usar o @JsonAlias("nome") para mapear corretamente a propriedade. Dessa forma, tanto "nomeCompleto" quanto "nome" serão aceitos ao fazer o mapeamento:

**public** **class** **Pessoa** {

@JsonAlias({"nomeCompleto", "nome"})

**private** String nomeCompleto;

}

Para mais informações sobre as anotações do Jackson, você pode consultar a documentação oficial aqui: [Documentação do Jackson Annotations](https://github.com/FasterXML/jackson).

**Generics**

Em Java, generics permitem criar classes, interfaces e métodos que podem trabalhar com tipos desconhecidos ou parâmetros genéricos. Eles fornecem uma forma de escrever código flexível e reutilizável, tornando-o independente de tipos específicos e permitindo que ele funcione com diferentes tipos de dados.

Para criar um método ou classe genérico, você precisa usar parâmetros de tipo (tipos genéricos) que são representados entre colchetes angulares < >. Geralmente, usamos letras maiúsculas únicas para representar os tipos genéricos, mas você pode usar qualquer identificador válido em Java. Aqui está um exemplo de uma classe genérica chamada Caixa, que armazena um valor de um tipo desconhecido:

**public** **class** **Caixa**<**T**> {

**private** T conteudo;

**public** T **getConteudo**() {

**return** conteudo;

}

**public** **void** **setConteudo**(T conteudo) {

**this**.conteudo = conteudo;

}

}

No exemplo acima, podemos criar um objeto do tipo Caixa e armazenar qualquer tipo de valor no mesmo, veja um exemplo:

**public** **class** **TestaCaixa** {

**public** **static** void **main**(String[] args) {

**Caixa**<**String**> caixaDeTexto = **new** **Caixa**();

caixaDeTexto.**setConteudo**("Guardando texto na minha caixa!");

**Caixa**<**Integer**> caixaDeIdade = **new** **Caixa**();

caixaDeIdade.**setConteudo**(30);

**Caixa**<**Double**> caixaDeValor = **new** **Caixa**<>();

caixaDeValor.**setConteudo**(150.50);

}

}

Observe que podemos utilizar a classe caixa para incluir valores de tipos diferentes. Para a variável **caixaDeTexto**, o compilador irá garantir que apenas valores do tipo String possam ser armazenados. Ja para a variável caixaDeIdade, o compilador irá garantir que apenas valores do tipo Integer possam ser armazenados, e assim sucessivamente.

**Método Genérico**

Para criar um método genérico, você pode usar a mesma sintaxe com parâmetros de tipo entre colchetes angulares. Aqui está um exemplo de um método genérico, que pertence à classe Caixa e que deverá somar o **valor** passado por parâmetro ao conteúdo da caixa:

**public** <T> T **somaConteudoNaCaixa**(T valor) {

**if** (this.conteudo **instanceof** **Integer** c && valor **instanceof** **Integer** i) {

**Integer** resultado = c + i;

**return** (T) resultado;

} **else** **if** (this.conteudo **instanceof** **Double** c && valor **instanceof** **Double** d) {

**Double** resultado = c + d;

**return** (T) resultado;

} **else** **if** (this.conteudo **instanceof** **String** c && valor **instanceof** **String** s) {

**String** resultado = c + "\n" + s;

**return** (T) resultado;

}

**return** **null**;

}

O objetivo do método acima é realizar a soma entre o conteúdo atual da caixa (this.conteudo) e o valor passado como parâmetro (valor). O método é genérico e pode ser usado para diferentes tipos de conteúdo que podem ser somados, como Integer, Double e String.

Vamos descrever o que acontece passo a passo:

public <T> T somaConteudoNaCaixa(T valor) {

O método é genérico e recebe um parâmetro valor do tipo genérico T, que é o mesmo tipo que será retornado como resultado da soma.

Logo em seguida, o método começa com uma série de condicionais if que verificam o tipo do conteúdo atual da caixa (this.conteudo) e o tipo do valor passado como parâmetro (valor).

**if** (this.conteudo **instanceof** Integer c && valor **instanceof** Integer i) {

// Realiza a soma entre os valores e armazena o resultado em uma variável

**Integer** resultado = c + i;

// Retorna o resultado como tipo genérico T (Integer no caso)

**return** (T) resultado;

}

A verificação é realizada usando os operadores **instanceof** e os operadores de pattern matching (instanceof com pattern variables) disponíveis a partir do Java 16.

Se o conteúdo atual (this.conteudo) e o valor (valor) forem ambos do mesmo tipo, é feita a soma ou a concatenação, como no caso da String. Caso o tipo da variável valor seja diferente do tipo do conteúdo, devolvemos o valor anterior do conteúdo. Vejamos como ficaria em nossa classe TestaCaixa:

**public** **static** **void** **main**(String[] args) {

Caixa<String> caixaDeTexto = **new** Caixa();

caixaDeTexto.setConteudo("Guardando texto na minha caixa!");

System.**out**.println(caixaDeTexto.somaConteudoNaCaixa("Mais uma linha"));

Caixa<Integer> caixaDeIdade = **new** Caixa();

caixaDeIdade.setConteudo(30);

System.**out**.println(caixaDeIdade.somaConteudoNaCaixa(26));

Caixa<Double> caixaDeValor = **new** Caixa<>();

caixaDeValor.setConteudo(150.50);

System.**out**.println(caixaDeValor.somaConteudoNaCaixa(350.50));

System.**out**.println(caixaDeValor.somaConteudoNaCaixa("texto"));

}

Ao executar o código acima, teremos como saída em nosso terminal, os seguintes valores:

Guardando texto na minha caixa!

Mais uma linha

56

501.0

**null**

Repare que na última linha do código, ao tentar incluir uma String “texto” em nossa **caixaDeValor**, ao executar esse código tivemos um retorno **null**, pois só realizamos a soma caso ambos os tipos fossem iguais.

**Trabalhando com tipos genéricos**

Imagine que você é um desenvolvedor em uma clínica médica e foi designado para desenvolver a API da empresa com o Spring Boot. Notando que há muitos códigos similares sendo usados várias vezes, você decide aplicar os conceitos de tipos genéricos para evitar essa repetição.

Qual das soluções abaixo seriam as mais adequadas para substituir os códigos repetitivos utilizando tipos genéricos? Selecione duas alternativas.

**public** **class** **GenericRepository**<T> {

**public** T **save**(T t) {

// Código para salvar 't' no banco de dados

}

}

Esta é a solução correta porque utiliza de forma correta o conceito de tipos genéricos, possibilitando que o repository salve diferentes tipos de dados, diminuindo a repetição de código.

**public** **class** **GenericRepository** {

**public** <T> T **save**(T t) {

// Código para salvar 't' no banco de dados

}

}

Esta solução também está correta. Embora a definição de T esteja ocorrendo no método ao invés da classe, ela continua aplicando os conceitos de tipos genéricos, permitindo que diversos tipos de dados sejam salvos diminuindo a repetição de códigos.

**Iterando para buscar temporadas de uma série**

Aprendemos a iterar para buscar todas as temporadas de uma série específica, a usar as anotações **@JsonAlias** e **@JsonIgnoreProperties**, a atualizar links durante a iteração e a imprimir lista das temporadas buscadas de APIs. Agora, imagine que precisamos buscar todas as temporadas de uma determinada série da API. Considerando que você já buscou os dados da série e alimentou-os na variável **dadosSerie**, como você implementaria esse recurso?

List<DadosTemporada> temporadas = **new** **ArrayList**<>();

**for**(**int** i = 1; i <= dadosSerie.**totalTemporadas**(); i++) {

json = consumoApi.**obterDados**("https://www.omdbapi.com/?t="+ serie + "&season=" + i + "&apikey=6585022c");

DadosTemporada dadosTemporada = conversor.**obterDados**(json, DadosTemporada.**class**);

temporadas.**add**(dadosTemporada);

}

temporadas.**forEach**(System.out::println);

**Coleções**

As coleções são estruturas de dados fundamentais no Java, que nos permitem armazenar e manipular conjuntos de elementos de forma eficiente. Elas são implementadas pela API de coleções do Java, que faz parte do pacote **java.util**. As coleções fornecem uma variedade de classes e interfaces para armazenar e organizar dados de diferentes maneiras, atendendo a diferentes necessidades e cenários.

A API de coleções do Java inclui **interfaces**, **classes abstrata**s e **classes concretas** para representar diferentes tipos de coleções. Algumas das principais interfaces de coleções são:

* **List**: Uma coleção ordenada que permite elementos duplicados. Os elementos são acessados por índices.
* **Set**: Uma coleção que não permite elementos duplicados e normalmente não possui ordem definida.
* **Queue**: Uma coleção que representa uma fila, onde os elementos são adicionados no final e removidos do início.
* **Map**: Uma coleção de pares chave-valor, onde cada chave é única e mapeada para um valor correspondente.

Além dessas interfaces, a API de coleções também inclui várias **classes concretas** que implementam essas **interfaces**, como **ArrayList**, **LinkedList**, **HashSet**, **TreeSet**, **HashMap**, **TreeMap**, entre outras.

Umas das interfaces mais utilizadas é o **List**. Conforme citado anteriormente, o **List** é uma interface que define uma sequência ordenada de elementos, onde cada elemento possui uma posição específica. Ao declarar uma variável desse tipo, estamos indicando que iremos trabalhar com uma coleção que mantém a ordem dos elementos.

Uma das principais razões para usarmos o **List** é a flexibilidade que ele oferece. Podemos adicionar, remover e acessar elementos de forma fácil e intuitiva. Além disso, ele também permite a duplicação de elementos, ou seja, podemos ter elementos repetidos na coleção. Outra vantagem é a possibilidade de percorrer os elementos de forma sequencial, utilizando laços de repetição, como o **for-each**, isso facilita a manipulação dos elementos e a realização de operações em massa. Veja um exemplo prático do uso do List em um código Java:

import java.util.List;

import java.util.ArrayList;

**public** **class** **ExemploList** {

**public** **static** **void** **main**(String[] args) {

// Criando um objeto do tipo List para armazenar números inteiros

List<Integer> numeros = **new** ArrayList<>();

// Adicionando elementos ao List

numeros.**add**(10);

numeros.**add**(20);

numeros.**add**(30);

// Acessando elementos do List

System.**out**.println("Primeiro elemento: " + numeros.**get**(0)); // Saída: 10

System.**out**.println("Segundo elemento: " + numeros.**get**(1)); // Saída: 20

System.**out**.println("Terceiro elemento: " + numeros.**get**(2)); // Saída: 30

// Percorrendo os elementos do List

**for** (Integer numero : numeros) {

System.**out**.println(numero);

}

// Removendo um elemento do List

numeros.**remove**(1); // Remove o elemento de índice 1 (20)

// Verificando o tamanho do List

System.**out**.println("Tamanho do List: " + numeros.size()); // Saída: 2

}

}

Além disso, o Java também possui outras coleções, como o **Set** e o **Map**. O **Set** é uma coleção que não permite elementos duplicados, enquanto o **Map** é uma coleção que associa chaves a valores, permitindo a recuperação rápida de um elemento através de sua chave.

As coleções do Java são extremamente úteis em diversas situações, como armazenar dados em memória, realizar operações de busca, ordenação e filtragem. Elas nos ajudam a organizar e manipular grandes quantidades de dados de forma eficiente e elegante.

Se você deseja se aprofundar no assunto, confira o curso: [Java Collections: Dominando Listas, Sets e Mapas](https://cursos.alura.com.br/course/java-collections).

**Buscando somente episódio pares**

Imagine que você é uma pessoa desenvolvedora e está trabalhando em uma aplicação que utiliza a API do OMDB para buscar informações de séries de TV. Sua tarefa é buscar uma determinada série e temporada e retornar apenas os episódios pares. Sua equipe está usando um trecho de repetição (loop) para resolver essa questão. Como você modificaria o código base para imprimir apenas os episódios pares de uma dada série e temporada?

**for** (DadosEpisodio episodio : dadosTemporada.episodios()) {

**if** (episodio.numero() % 2 == 0) {

**System**.out.println(episodio);

}

}

**Constantes**

As constantes desempenham um papel importante na programação, permitindo que valores fixos e imutáveis sejam armazenados e utilizados ao longo do código. Elas são especialmente úteis quando temos valores que não devem ser alterados durante a execução do programa.

Ao declarar uma constante, é importante seguir algumas boas práticas. A nomenclatura das constantes deve ser clara e descritiva, utilizando letras maiúsculas e separando as palavras por underscore (\_), seguindo o padrão conhecido como "**snake\_case**". Isso torna o código mais legível e compreensível para outros desenvolvedores.

Em Java, podemos declarar uma constante utilizando a palavra-chave **final**. Por exemplo:

**final** **int** ANO\_ATUAL = 2022;

**final** **String** NOME\_EMPRESA = "Alura";

Nesse exemplo, **ANO\_ATUAL** e **NOME\_EMPRESA** são constantes que armazenam um valor inteiro e uma string, respectivamente. O uso da palavra-chave final indica que essas variáveis não podem ter seu valor alterado após a atribuição inicial.

Além disso, é uma boa prática declarar as constantes como **static** caso elas pertençam a uma classe e sejam compartilhadas por vários objetos. Dessa forma, as constantes podem ser acessadas diretamente através do nome da classe, sem a necessidade de instanciar um objeto.

**public** **class** **ExemploConstantes** {

**public** **static** **final** **int** ANO\_ATUAL = 2022;

**public** **static** **final** **String** NOME\_EMPRESA = "Alura";

}

No exemplo acima, as constantes **ANO\_ATUAL** e **NOME\_EMPRESA** são declaradas como **static**, permitindo que sejam acessadas diretamente através da classe **ExemploConstantes**. Veja por exemplo como poderíamos acessá-las a partir de nosso método **main**.

**public** **class** **Principal** {

**public** **static** void **main**(String[] args) {

**System**.out.**println**(“**Eu** trabalho na empresa “ + **ExemploConstantes**.NOME\_EMPRESA);

}

}

O uso de constantes traz benefícios como facilitar a manutenção do código, evitar erros de digitação e tornar o código mais legível. Além disso, elas ajudam a evitar a repetição de valores em diferentes partes do código, promovendo a consistência e a reutilização.

Lembre-se de que as constantes devem ser utilizadas para valores que não devem ser alterados durante a execução do programa.

**Funções Lambda**

As **funções lambda** são um recurso muito útil na linguagem de programação Java. Elas podem ajudar a simplificar nosso código e torná-lo mais fácil de ler e compreender. Mas antes de avançarmos, vamos começar pelo início.

**O que são funções Lambda?**

As funções Lambda - ***chamadas de funções anônimas*** - são uma maneira de definir funções que são frequentemente usadas uma única vez, direto no local onde elas serão usadas.

Na programação convencional, normalmente definimos uma função em algum lugar do nosso código e, em seguida, chamamos essa função por seu nome em outro lugar, sempre que precisamos usá-la.

Porém, às vezes precisamos de uma função que seja usada em apenas um lugar no nosso código. Para esses casos, pode ser mais simples e direto definir essa função diretamente no lugar onde ela será usada, sem lhe dar um nome, ou seja, usando uma função Lambda.

A principal vantagem do uso de funções lambda é a simplificação do código e a melhora na legibilidade, já que a função é definida logo onde será usada.

**Como usamos funções Lambda no Java?**

Em Java, uma função lambda é definida da seguinte maneira:

(argumentos) -> { corpo-da-função }

Por exemplo, podemos definir uma função lambda que adicione dois números da seguinte maneira:

(a, b) -> { **return** a + b; }

Aqui, **a** e **b** são os argumentos da nossa função. O corpo da função, que é o código que será executado quando a função for chamada, está entre chaves **{}**. Neste caso, o corpo da função é apenas uma linha que retorna a soma de **a** e **b**.

Em Java, as funções lambda são, geralmente, usadas com interfaces funcionais. ***Uma interface funcional é uma interface que contém apenas um único método***. A função lambda então fornece a implementação desse único método.

**Exemplos de uso de funções Lambda**

Vamos a um exemplo concreto para entender melhor. Suponhamos que temos uma lista de números e queremos printar apenas os números pares dessa lista. Sem o uso de funções lambda, poderíamos fazer algo assim:

List<Integer> lista = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9);

**for**(Integer i: lista) {

**if**(i % 2 == 0) {

System.out.println(i);

}

}

Porém, com o uso de funções lambda, podemos simplificar esse código:

List<Integer> lista = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9);

lista.stream().filter(i -> i % 2 == 0).forEach(System.out::println);

No código acima, criamos um stream da nossa lista, filtramos esse stream para incluir apenas os números pares (isso é feito pela função lambda **i -> i % 2 == 0**), e finalmente usamos o método **forEach** para printar cada elemento do stream filtrado.

Agora ficou mais claro como as funções Lambda em Java podem nos ajudar a simplificar nosso código e torná-lo mais legível!

**Ignorando propriedades no Java**

Imagine que você está desenvolvendo o ADOPET, um site para adoção de animais que se conecta a uma API, que retorna informações sobre os animais disponíveis para adoção.

As informações da API são retornadas em JSON, e você precisa manipulá-las para serem apresentadas no site. Durante o desenvolvimento, você se deparou com a necessidade de ignorar algumas propriedades que são retornadas, mas não serão usadas no seu site.

Como você pode ignorar os retornos de propriedades indesejadas da API no seu código em Java?

@JsonIgnoreProperties(ignoreUnknown = **true**)

A anotação **@JsonIgnoreProperties** é a maneira correta de ignorar as propriedades desejadas no Java, colocando seu atributo **ignoreUnknown** como **true**, visto que por ***padrão ele é setado como false***.

**Operações intermediárias e finais**

As streams são uma nova forma de trabalhar com coleções de dados no Java, introduzidas a partir do Java 8. Elas permitem realizar operações de forma mais eficiente e concisa, utilizando uma abordagem funcional.

Uma stream é uma sequência de elementos que pode ser processada em paralelo ou em série. Ela pode ser criada a partir de uma coleção, um array, um arquivo, entre outros. A partir daí, podemos realizar diversas operações nessa stream, como filtrar, mapear, ordenar, entre outras.

As ***operações intermediárias******são aquelas que podem ser aplicadas em uma stream e retornam uma nova stream como resultado***. Essas operações não são executadas imediatamente, mas apenas quando uma operação final é chamada. Vamos ver alguns exemplos de operações intermediárias:

* **Filter**: permite filtrar os elementos da stream com base em uma condição. Por exemplo, podemos filtrar uma lista de números para retornar apenas os números pares.

List<**Integer**> numeros = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

List<**Integer**> numerosPares = numeros.stream()

.**filter**(n -> n % 2 == 0)

.**collect**(Collectors.toList());

System.out.println(numerosPares);

// Output: [2, 4, 6, 8, 10]

* **Map**: permite transformar cada elemento da stream em outro tipo de dado. Por exemplo, podemos transformar uma lista de strings em uma lista de seus respectivos tamanhos.

List<String> palavras = Arrays.asList("Java", "Stream", "Operações", "Intermediárias");

List<Integer> tamanhos = palavras.stream()

.**map**(s -> s.length())

.collect(Collectors.toList());

System.out.println(tamanhos);

// Output: [4, 6, 11, 17]

As ***operações finais são aquelas que encerram a stream e retornam um resultado concreto***. Algumas operações finais comuns são o forEach, collect e count. Vamos ver alguns exemplos de operações finais:

* **ForEach**: permite executar uma ação em cada elemento da stream. Por exemplo, podemos imprimir cada elemento da lista.

List<**String**> nomes = Arrays.**asList**("João", "Maria", "Pedro", "Ana");

nomes.**stream**()

.**forEach**(nome -> System.out.**println**("Olá, " + nome + "!"));

// Output:

// Olá, João!

// Olá, Maria!

// Olá, Pedro!

// Olá, Ana!

* **Collect**: permite coletar os elementos da stream em uma coleção ou em outro tipo de dado. Por exemplo, podemos coletar os números pares em um conjunto.

List<**Integer**> numeros = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

**Set**<**Integer**> numerosPares = numeros.stream()

.**filter**(n -> n % 2 == 0)

.**collect**(Collectors.toSet());

System.out.println(numerosPares);

// Output: [2, 4, 6, 8, 10]

Por fim, as streams e suas operações intermediárias e finais são uma forma poderosa de manipular coleções de dados no Java de maneira mais eficiente e concisa. Elas permitem escrever código mais legível e expressivo, facilitando o processamento e a transformação de dados.

Além das operações intermediárias e finais mencionadas, existem muitas outras disponíveis, como **distinct** (que remove elementos duplicados), **limit** (que limita o número de elementos da stream), **skip** (que pula os primeiros elementos da stream), **reduce** (que combina os elementos da stream em um único resultado) e muitas outras.

Streams no Java são uma poderosa ferramenta para manipulação de coleções de dados. As operações intermediárias permitem filtrar, mapear e ordenar os elementos da stream, enquanto as operações finais encerram a stream e retornam um resultado concreto. Combinando essas operações, podemos realizar transformações e ações de forma concisa e eficiente.

**Uso de construtores personalizados em Java**

Vamos explorar um tema bastante interessante e muito útil em Java: os Construtores Personalizados. Eles são como um comando especial que usamos para criar e inicializar objetos de uma classe. O construtor personalizado nos permite definir como queremos criar nossos objetos, permitindo que inicializemos as propriedades da classe com os valores que especificamos. No final, isso nos dá mais controle sobre a maneira como nossos objetos são criados.

**Quando os Construtores Personalizados são úteis?**

Vamos considerar um exemplo simples. Digamos que você esteja criando uma classe de Carro em Java, onde cada Carro tem uma marca, um modelo e um ano.

Sem um construtor personalizado, você teria que criar um carro assim:

**Carro** meuCarro = **new** **Carro**();

meuCarro.setMarca("Toyota");

meuCarro.setModelo("Corolla");

meuCarro.setAno(2015);

Isso não é eficiente e também pode levar a erros, pois nada impede de criar um carro sem definir todas as propriedades.

Agora, usando um construtor personalizado:

**public** **class** **Carro** {

**private** String marca;

**private** String modelo;

**private** **int** ano;

// Construtor personalizado

**public** **Carro**(String marca, String modelo, **int** ano) {

this.marca = marca;

this.modelo = modelo;

this.ano = ano;

}

}

Agora, você pode criar um objeto **meuCarro** de forma muito mais fácil e segura:

**Carro** meuCarro = **new** **Carro**("Toyota", "Corolla", 2015);

O uso de construtores personalizados garante que todos os objetos que criamos têm todas as propriedades necessárias definidas.

**Possíveis problemas ao usar Construtores Personalizados**

Apesar de úteis, os construtores personalizados devem ser usados corretamente para evitar problemas.

Um problema comum é a **sobrecarga de construtores**, onde você tem muitos construtores na mesma classe. Isso pode tornar seu código confuso e difícil de manter.

Outro problema pode surgir ao **usar a herança**. Se a classe pai tem um construtor personalizado, a classe filha deve chamar explicitamente esse construtor, ou o compilador retornará um erro.

Além disso, se você **não definir um construtor padrão** (um construtor sem argumentos), o Java não criará um para você se você tiver qualquer construtor personalizado. Isso pode ser um problema se você precisar criar um objeto sem inicializar qualquer propriedade.

Em resumo, construtores personalizados em Java são uma ferramenta muito útil na criação de objetos, garantindo que todas as propriedades necessárias sejam inicializadas. No entanto, eles devem ser usados com cuidado para evitar problemas de sobrecarga e herança. Como sempre, a melhor estratégia é usar a ferramenta certa para o trabalho certo e manter seu código simples e limpo.

**Imutabilidade dos dados**

A imutabilidade dos dados é um conceito fundamental na programação, que se refere à característica de um objeto ou valor não poder ser alterado após ser criado. Isso traz diversos benefícios, como segurança e confiabilidade do código, uma vez que os dados imutáveis não podem ser acidentalmente modificados ou corrompidos.

Além disso, a imutabilidade promove a concorrência segura em ambientes multithread, pois cada thread pode trabalhar com cópias dos dados imutáveis sem interferir nas outras. A imutabilidade também torna o código mais legível e facilita a manutenção, pois o fluxo de informações é mais claro e previsível.

Em Java, ***podemos criar objetos imutáveis usando classes finais, atributos finais e métodos que não modificam o estado interno do objeto***. Essa prática é essencial para desenvolver programas mais robustos e confiáveis.

Vimos no vídeo da aula anterior, que ao utilizar streams, temos como uma das operações finais a possibilidade de colocar os dados gerados em outra lista. Vimos também que se utilizarmos diretamente o **toList()** para esse fim, é gerada uma lista imutável. Sendo assim, quando precisamos que a nova lista possa ser modificada, temos que utilizar o **collect(Collectors.toList())**.

Em Java, algumas classes nativas já são imutáveis. Alguns exemplos incluem:

* **String**: A classe **java.lang.String** é imutável. Uma vez criada, uma instância de String não pode ser modificada.
* **Classes Wrapper**: As classes que representam os tipos primitivos, como **Integer, Double, Character,** etc., também são imutáveis.
* **Enum**: As enumerações (enum) em Java são imutáveis. As instâncias de um enum são criadas apenas uma vez e não podem ser modificadas.

Além das classes nativas, a API de coleções do Java também oferece algumas classes imutáveis na classe utilitária **Collections**. Essas classes são usadas para criar coleções imutáveis, como listas, conjuntos e mapas. Alguns exemplos incluem:

* **Collections.unmodifiableList**: Cria uma lista imutável a partir de uma lista existente.
* **Collections.unmodifiableSet**: Cria um conjunto imutável a partir de um conjunto existente.
* **Collections.unmodifiableMap**: Cria um mapa imutável a partir de um mapa existente.

Essas classes permitem que você crie versões imutáveis de coleções existentes, tornando-as seguras para compartilhar e evitar a modificação acidental.

Veja esse código de exemplo:

**public** **class** **ImutabilidadeExemplo** {

**public** **static** void **main**(String[] args) {

**List**<**String**> listaOriginal = **new** **ArrayList**<>();

listaOriginal.**add**("A");

listaOriginal.**add**("B");

**List**<**String**> listaImutavel = **Collections**.**unmodifiableList**(listaOriginal);

// Tentativa de adicionar um elemento na lista imutável resultará em exceção

listaImutavel.**add**("C"); // Lançará UnsupportedOperationException

}

}

**Filtrando e Ordenando Episódios com Streams**

Você trabalha desenvolvendo aplicações em um banco digital e uma de suas tarefas é gerenciar e analisar de maneira eficiente uma enorme quantidade de dados referentes aos investimentos e saldos dos clientes, a fim de que se possa oferecer a eles linhas de crédito apropriadas ao saldo que possuem em conta.

Neste contexto, você deve filtrar os clientes com conta-corrente e ordená-los considerando o saldo, ordenando do maior para o menor, e, então, exibir os cinco clientes com maior saldo em conta.

Considerando que a variável dadosClientes já está populada com a lista completa de clientes e que os métodos para obter o tipo da conta e o saldo são o getTipoConta() e getSaldo(), respectivamente, qual dos seguintes códigos fará a funcionalidade corretamente?

dadosClientes.stream()

.filter(c -> c.getTipoConta().equalsIgnoreCase("corrente"))

.sorted(Comparator.comparingDouble(Conta::getSaldo).reversed())

.limit(5);

Este código utiliza corretamente o método stream para iniciar a manipulação da lista. Então, o filtro é aplicado para buscar as contas que tem o tipo “corrente”. A lista é ordenada em ordem decrescente de saldo e, finalmente, são obtidos os cinco primeiros elementos.

**Filtrando e classificando dados**

Você precisa exibir uma lista dos 10 melhores filmes lançados num determinado ano. Para isso, você levará em consideração o total de visualizações, que deve ser maior ou igual a 10 mil, e avaliação dos clientes, que deverá ser maior ou igual a 9.4. Considere a classe **Filme** abaixo:

**public** **class** **Filme** {

**private** String titulo;

**private** Integer totalVisualizacao;

**private** Double avaliacao;

**private** Integer anoLancamento;

//getter, setters e toString() omitidos

}

Considerando que **List<Filme> filmes** já está com todos os filmes populados e você quer saber os melhores filmes de 2022, como você faria o código para realizar essa tarefa?

filmes.stream()

.filter(filme -> filme.getAnoLancamento() == 2022)

.filter(filme -> filme.getTotalVisualizacao() >= 10000 && filme.getAvaliacao() >= 9.4)

.sorted(Comparator.comparingDouble(Filme::getAvaliacao).reversed())

.limit(10)

.forEach(System.out::println);

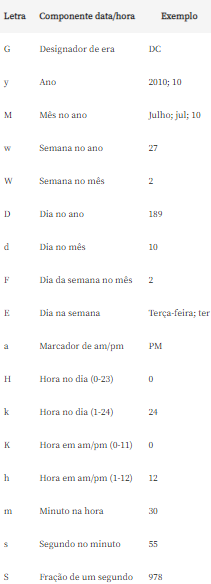
Criamos um stream e utilizamos duas vezes o método **filter**, uma para pegar somente filmes lançados em 2022 e outra para filtrar os filmes de acordo com os critérios determinados. Depois utilizamos o **sorted** para ordenar pela avaliação de forma decrescente e então utilizamos o método limit para limitar a lista em 10 filmes. Por fim, utilizamos a operação final forEach para imprimir os dados do filme.

**Trabalhando com datas**

Para trabalhar com datas, podemos importar o pacote que possui diversas classes de data e hora, o **java.time**. Essa API de datas foi introduzida no Java 8 com a ideia de resolver os problemas das antigas classes e facilitar o uso de datas no Java. Vamos conhecer algumas funcionalidades?

**Padrões**

Os formatos de data e hora são especificados por strings padrão de data e hora, a tabela a seguir mostra a sintaxe padrão que deve ser utilizada para o padrão de formatação:



Exemplo:

**import** java.text.SimpleDateFormat;

**import** java.util.Date;

**public** **class** **Datas** {

**public** **static** **void** **main**(String[] args) {

**String** pattern = "E, dd MMM yyyy HH:mm:ss z";

**SimpleDateFormat** simpleDateFormat = **new** **SimpleDateFormat**(pattern);

**String** date = simpleDateFormat.format(**new** **Date**());

System.out.println(date);

}

}

Este código produzirá:

sá**b**., 19 nov. 2022 02:58:14 BRT”:

**Local Date**

O local date representa uma data. Uma data sem fuso horário no sistema de calendário ISO-8601, tem como formato padrão yyyy-MM-dd. Essa classe não armazena ou representa uma hora ou fuso horário. Em vez disso, é uma descrição da data, normalmente usada para aniversários.

**Data atual**

Vamos utilizar o Local Date para criar a data atual, a de hoje:

**import** java.time.LocalDate;

**public** **class** **Datas** {

**public** **static** **void** **main**(String[] args) {

**LocalDate** hoje = LocalDate.now();

System.out.println(hoje);

}

}

Perceba o resultado é a data atual no formato yyyy-MM-dd :

2022-11-19

**Criando datas**

Agora, vamos criar uma nova data para representar a data de um aniversário. Para isso, importaremos o pacote **java.time.month**;

**import** java.time.LocalDate;

**import** java.time.Month;

**public** **class** **Datas** {

**public** **static** **void** **main**(String[] args) {

**LocalDate** hoje = LocalDate.now();

System.out.println(hoje);

**LocalDate** aniversarioAlice = LocalDate.of(2005, Month.SEPTEMBER, 15);

System.out.println(aniversarioAlice);

}

}

O resultado também será no formato yyyy-MM-dd:

2022-11-19

2005-09-15

Também podemos brincar com esse código e calcular a idade atual da aniversariante. Uma forma de fazer isso na mão seria subtraindo o método **getYear** das datas, dessa forma:

**import** java.time.LocalDate;

**import** java.time.Month;

**public** **class** **Datas** {

**public** **static** **void** **main**(String[] args) {

**LocalDate** hoje = LocalDate.now();

System.out.println(hoje);

**LocalDate** aniversarioAlice = LocalDate.of(2005, Month.SEPTEMBER, 15);

System.out.println(aniversarioAlice);

**int** idade = hoje.getYear() - aniversarioAlice.getYear();

System.out.println(idade);

}

}

O resultado também será:

2022-11-19

2005-09-15

17

**Período**

Ao executar esse código temos o resultado esperado, que neste caso é 17 anos. Mas e se quiséssemos descobrir a diferença de dias e meses também? Daria pra fazer da mesma forma utilizando o get, no entanto, existe algo pronto para nos ajudarmos. Nesse caso, podemos utilizar a classe **Period**.

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

**import** java.time.LocalDate;

**import** java.time.Month;

**import** java.time.Period;

**public** **class** **Datas** {

**public** **static** **void** **main**(String[] args) {

**LocalDate** hoje = LocalDate.now();

System.out.println(hoje);

**LocalDate** aniversarioAlice = LocalDate.of(2005, Month.SEPTEMBER, 15);

System.out.println(aniversarioAlice);

**int** idade = aniversarioAlice.getYear() - hoje.getYear();

System.out.println(idade);

**Period** periodo = Period.between(hoje, aniversarioAlice);

System.out.println(periodo);

}

}

Repare que a saída desse nosso **println** vai ser um pouco estranha, um exemplo seria: P-17Y-2M-4D. Isso significa um período de 17 anos, 2 meses e 4 dias.

**Local Time**

O local time representa um horário, seu formato padrão é hh:mm:ss.zzz.

**Horário atual**

Assim como fizemos com o local date, dessa vez vamos imprimir o horário atual:

**import** java.time.LocalTime;

**public** **class** **Horarios** {

**public** **static** void **main**(String[] args) {

**LocalTime** hoje = **LocalTime**.**now**();

**System**.out.**println**(hoje);

}

}

Perceba o resultado é a data atual no formato hh:mm:ss.zzz:

**06:02:21.029460400**

**Criando horários**

Agora, vamos criar um novo horário para representar as horas em que Alice nasceu:

**import** java.time.LocalTime;

**public** **class** **Horarios** {

**public** **static** **void** **main**(String[] args) {

**LocalTime** hoje = LocalTime.now();

System.out.println(hoje);

**LocalTime** aniversarioHoraAlice = LocalTime.of(22, 33, 15);

System.out.println(aniversarioHoraAlice);

}

}

O resultado será no formato hh:mm:ss, pois colocamos dessa forma no **LocalTime.of()**

**22:33:15**

**Local Date Time**

**Horário e data atual**

O localDateTime representa uma data com a hora, visto como ano-mês-dia-hora-minuto-segundo. O tempo é representado como **yyyy-MM-dd-HH-mm-ss.zzz**.

**import** java.time.LocalDateTime;

**public** **class** **DataHorario** {

**public** **static** **void** **main**(String[] args) {

**LocalDateTime** hoje = LocalDateTime.now();

System.out.println(hoje);

}

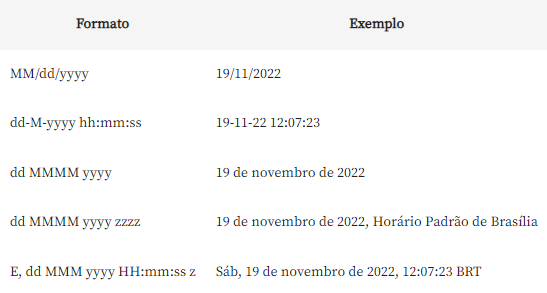
}

Resultado:

**2022-11-19T04:11:17.108565200**

**Date Time Formatter**

Você deve ter notado que os resultados estão em formatos que não estamos acostumados a trabalhar. Podemos então trabalhar com diversos formatadores de datas existentes!



Além disso, também podemos mudar a forma de exibir o mês. Se utilizarmos letras em quantidades diferentes, consequentemente o resultado também é. Vejamos:

Interface gráfica do usuário, Aplicativo

Descrição gerada automaticamente

Para formatarmos, vamos utilizar o **DateTimeFormatter**. Existem diversos já prontos, mas há ainda a alternativa de você criar o seu próprio formato no padrão já conhecido de **dd/MM/yyyy hh:mm:ss**.

Para fazer isso basta você utilizar o método **ofPattern**. Como vimos nas tabelas anteriores, podemos escolher como será impresso.

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

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

**import** java.time.LocalDateTime;

**import** java.time.format.DateTimeFormatter;

**public** **class** **Formatando** {

**public** **static** **void** **main**(String[] args) {

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

**LocalDateTime** agora = LocalDateTime.now();

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

}

}

Agora sim, ao executar temos o resultado:

19/11/2022 04:38:11

**Entendendo o fluxo**

Como temos estudado no curso, Streams nos permitem manipular coleções de elementos (como Lists ou Sets) de forma mais sofisticada e eficiente do que os métodos tradicionais. Eles são especialmente úteis quando precisamos realizar operações em massa nesses elementos, como filtrar, transformar ou agregar.

**E a função Peek, o que é?**

A função “peek” é uma operação intermediária de uma stream. Uma operação intermediária é aquela que processa os dados da stream, mas não a finaliza, ou seja, ela permite que mais operações sejam feitas depois dela. A função peek é usada para "espiar" os elementos da stream sem alterá-los, o que pode ser muito útil para depuração. Essa função vai receber cada elemento da stream e realizar alguma operação de efeito colateral (como imprimir o elemento), mas vai devolver o mesmo elemento sem alterar nada.

**Por que eu deveria usar isso?**

O uso de Streams e a função peek podem te ajudar a escrever códigos mais elegantes, fáceis de ler, e diminuir a possibilidade de erros. Além disso, eles podem fazer seu código rodar mais rápido em certos casos, pois os Streams permitem a paralelização das operações, ou seja, executar várias operações ao mesmo tempo.

**Exemplos de código**

Vamos ver agora um exemplo simples de como podemos usar a função peek com streams. Imagine que temos uma lista com números e queremos multiplicar cada um deles por 2 e depois somar os resultados.

List<Integer> numeros = Arrays.asList(1, 2, 3, 4, 5);

**int** soma = numeros.stream()

.peek(n -> System.out.println("Elemento: " + n))

.map(n -> n \* 2)

.peek(n -> System.out.println("Conteúdo depois do map: " + n))

.reduce(0, (total, numero) -> total + numero);

System.out.println("A soma dos números é: " + soma);

Nesse código, nós criamos uma stream com a lista de números, espiamos cada número para imprimir (com a função.peek), multiplicamos o valor por dois, espiamos também para conferir o valor da multiplicação e depois somamos os resultados. Veja como aparece no terminal após a execução da aplicação:

Elemento: 1

Conteú**do** depois **do** map: 2

Elemento: 2

Conteú**do** depois **do** map: 4

Elemento: 3

Conteú**do** depois **do** map: 6

Elemento: 4

Conteú**do** depois **do** map: 8

Elemento: 5

Conteú**do** depois **do** map: 10

A soma dos números é: 30

Process finished **with** **exit** code 0

Isso nos ajuda a ter certeza de que o processo está sendo feito corretamente. Aqui conseguimos perceber que a soma dos números 2 + 4 + 6 + 8 + 10 foi calculada corretamente, resultando em 30. Ou seja, a função lambda em nosso método reduce está fazendo exatamente o que era planejado. E pudemos conferir isso, com o peek.

Esse é apenas um exemplo básico, mas lembre-se que a função peek pode ser usada em qualquer lugar dentro da sua stream, sempre que você quiser ver quais elementos estão passando por aquele ponto.

**Prática de Debug com a função 'peek'**

Você está trabalhando como desenvolvedor(a) no Gatito Petshop. O projeto atual envolve a utilização do Spring Boot API e Java Streams. Durante sua atividade, você se lembra que a função peek pode ser bastante útil para acompanhar o processamento dos streams e facilitar o processo de debug. Você decide então aplicar essa função, mas precisa recordar de como ela funciona.

Como você aplicaria a função 'peek' em uma lista de objetos Pet (animais) para imprimir cada elemento enquanto ele é processado?

pets.stream()

.peek(System.out::println)

.collect(Collectors.toList());

A função 'peek' é usada corretamente. Ela recebe uma função lambda como parâmetro e atua sobre cada elemento da stream, neste caso imprimindo-os. A operação 'collect' transforma a stream de volta em uma lista.

**Utilizando um container de dados - Explorando o Optional**

A programação Java usa Containers para armazenar, acessar e manipular dados. Um exemplo particularmente útil é o **Optional** - ***um container especial que pode conter ou não um valor***. Então, vamos descobrir mais sobre o **Optional**, como utilizá-lo e quais práticas seguir.

**O que é o Optional?**

O **Optional** é um único valor de container introduzido no Java 8. Ele pode conter um valor único ou nenhum valor. O principal uso do **Optional** é fornecer um tipo de retorno alternativo quando um método pode não retornar um valor.

Usar **null** é uma prática comum, mas pode levar a erros como **NullPointerException**. O **Optional** ajuda a evitar esses erros fornecendo uma maneira mais segura de lidar com valores que podem ou não estar presentes.

**Como usar o Optional?**

Vamos criar um método que pode retornar um valor null. Sem o **Optional**, ele pode causar erros indesejados, mas com o **Optional**, ele é mais seguro:

**public** Optional<String> **getNome**() {

// O nome pode ser null

**return** Optional.ofNullable(nome);

}

Aqui, **Optional.ofNullable(nome)** criará um **Optional** que contém o valor de nome se ele não for **null**. Se for **null**, ele criará um **Optional** vazio.

Agora, para acessar valor dentro do **Optional**, podemos usar **ifPresent** e **orElse** assim:

Optional<String> optionalNome = getNome();

optionalNome.ifPresent(System.out::println); // Só irá printar o nome se não for null

**String** nome = optionalNome.orElse("Nome não disponível"); // Irá retornar "Nome não disponível" caso nome seja null

**Melhores práticas**

Embora o **Optional** seja um aliado útil, há algumas coisas que devem ser levadas em consideração para usá-lo de maneira eficaz:

1. **Prefira o retorno Optional em vez de retornar null:** Isso torna suas intenções claras e evita erros.
2. **Não use Optional.get() sem Optional.isPresent():** O **Optional.get()** lançará um erro se o valor não estiver presente. Portanto, é melhor verificar antes se o valor está presente.
3. **Não use Optional para campos da classe ou parâmetros do método:** O **Optional** deve ser usado principalmente para retornos de métodos que podem não ter valor.

Entender e usar corretamente o **Optional** pode levar a um código mais limpo e menos propenso a erros, então vale a pena investir seu tempo para aprender.

**Trabalhando em atividades paralelas**

Trabalhar em atividades paralelas é uma prática essencial na programação, pois permite executar várias tarefas simultaneamente, aumentando a eficiência do código. Uma das formas de realizar programação paralela em Java é utilizando threads.

As threads são unidades independentes de execução que podem ser criadas dentro de um programa. Elas permitem que diferentes partes do código sejam executadas ao mesmo tempo, melhorando o desempenho e a responsividade do programa.

**findAny**

Uma operação útil para trabalhar com threads é o método **findAny** em coleções de dados. Esse método é utilizado para encontrar qualquer elemento que satisfaça uma determinada condição em uma coleção. Ele retorna um **Optional**, que pode conter o elemento encontrado ou ser vazio caso nenhum elemento satisfaça a condição.

Ao utilizar o **findAny** em uma coleção com threads, cada thread pode buscar um elemento da coleção de forma paralela, tornando a busca mais rápida e eficiente. É importante ressaltar que o **findAny** ***não garante que sempre será retornado o mesmo elemento, pois a ordem de busca pode variar entre as threads***.

Para utilizar o **findAny** com threads, você pode criar uma implementação de **Predicate** que define a condição que os elementos devem satisfazer. Em seguida, você pode utilizar o método **parallelStream** em uma coleção para criar uma stream paralela e chamar o **findAny**, passando o **Predicate** como argumento. Vejamos um exemplo:

**public** **class** **ExemploFindAnyParallelStream** {

**public** **static** **void** **main**(String[] args) {

List<Integer> numeros = **new** ArrayList<>();

**for** (int i = 1; i <= 100; i++) {

numeros.**add**(i);

}

// Utilizando parallelStream para encontrar um elemento qualquer em paralelo

Optional<Integer> numeroQualquer = numeros.parallelStream()

.filter(numero -> numero % 10 == 0) // Filtra os números que são múltiplos de 10

.findAny();

**if** (numeroQualquer.isPresent()) {

System.**out**.println("Encontrado: " + numeroQualquer.**get**());

} **else** {

System.**out**.println("Nenhum número encontrado.");

}

}

}

Neste exemplo, temos uma lista de números de 1 a 100. Utilizando **parallelStream**, filtramos os números que são múltiplos de 10 e, em seguida, utilizamos **findAny** para encontrar qualquer elemento que atenda a essa condição. Como a busca é feita em paralelo, o resultado pode variar a cada execução do programa, pois diferentes threads podem encontrar elementos diferentes do stream. A saída no console pode ser, por exemplo:

**Encontrado: 70**

ou

**Encontrado: 50**

ou qualquer outro número que seja múltiplo de 10 dentro da lista.

Lembrando que, para alguns tipos de stream, como streams de listas ou arrays, a ordem dos elementos é mantida. No entanto, em um stream paralelo, a busca será executada de forma concorrente, e o resultado pode ser qualquer elemento que satisfaça a condição de filtragem. Portanto, em streams paralelos, o método **findAny** pode ser mais útil para obter qualquer elemento sem a garantia de um resultado específico.

**Armazenando dados em um Set**

Imagine que você é um analista de dados no Bytebank Banco Digital, você foi incumbido de monitorar o fluxo de transações bancárias realizadas. O banco utiliza o Java Stream API para processar esses dados. Durante o dia, uma infinidade de operações ocorre, e todas são registradas em tempo real.

Utilizando o Stream API e o conceito de Maps, peek, findFirst e demais recursos', como você criaria um fluxo que filtra as transações com **valores superiores a 5000, imprime uma espécie de log com peek, busca a primeira destas transações encontradas e coleta os resultados em um Set**?

transacoes.stream()

.filter(t -> t.getValor() > 5000)

.peek(System.out::println)

.collect(Collectors.toSet()).stream()

.findFirst();

Este código vai compilar e funcionar conforme o esperado. Ele filtra as transações com valores superiores a 5000, imprime um registro de cada uma com o método peek, coleta os resultados em um Set e retorna o primeiro elemento do Set.

**Classes que facilitam a geração de estatísticas**

Quando se trata de gerar estatísticas em Java, existem classes que podem simplificar bastante o trabalho. Duas delas são **DoubleSummaryStatistics** e **IntSummaryStatistics**, que são usadas para calcular estatísticas básicas em coleções de valores do tipo double e int, respectivamente.

A classe **DoubleSummaryStatistics** é uma classe utilitária que permite calcular estatísticas como soma, média, valor mínimo e máximo, além do total de elementos em uma coleção de valores do tipo double. Ela possui métodos como getSum(), getAverage(), getMin(), getMax() e getCount(), que retornam os respectivos valores estatísticos.

Da mesma forma, a classe **IntSummaryStatistics** é usada para calcular estatísticas básicas em coleções de valores do tipo int. Ela também possui métodos como getSum(), getAverage(), getMin(), getMax() e getCount() para obter as estatísticas desejadas.

Essas classes são especialmente úteis quando você precisa realizar cálculos estatísticos em uma coleção de valores. Por exemplo, se você tem uma lista de preços de produtos e deseja calcular a média, o valor mínimo e o valor máximo, você pode utilizar a classe DoubleSummaryStatistics para obter essas informações de forma simples e eficiente, de forma bem similar ao que fizemos ao pegar as estatísticas dos episódios com maior e menor avaliação, a média e o total de itens levados em consideração na montagem de estatísticas.

Para usar essas classes, você pode criar uma instância delas e, em seguida, percorrer a coleção de valores, atualizando as estatísticas conforme necessário. Ao final do processo, você pode chamar os métodos apropriados para obter as estatísticas desejadas. Vamos ver um exemplo onde temos uma lista de estudantes e queremos pegar dados sobre suas idades.

Para a classe Aluno teremos o seguinte código:

**public** **class** **Aluno** {

**private** String nome;

**private** LocalDate nascimento;

**public** **Aluno**(String nome, LocalDate nascimento) {

**this**.nome = nome;

**this**.nascimento = nascimento;

}

**public** int **getIdade**() {

Period periodo = Period.between(nascimento, LocalDate.now());

**return** periodo.getYears();

}

//getters, setters e toString omitidos

}

Criamos um método getIdade(), onde estamos usando o Period, que assim como o LocalDate é uma classe do pacote java.time, para calcular a idade do estudante de acordo com o dia que a aplicação está sendo executada.

Em nossa classe Principal teremos o seguinte código:

**public** **class** **Principal** {

**public** **static** void **main**(String[] args) {

**List**<**Aluno**> alunos = **Arrays**.**asList**(

**new** **Aluno**("Alice", **LocalDate**.**of**(2002, 10, 20)),

**new** **Aluno**("Bob", **LocalDate**.**of**(1980, 8, 9)),

**new** **Aluno**("Carlos", **LocalDate**.**of**(2001, 01, 28)),

**new** **Aluno**("David", **LocalDate**.**of**(2003, 05, 12)),

**new** **Aluno**("Eva", **LocalDate**.**of**(2005, 12, 03))

);

**IntSummaryStatistics** stats = alunos.**stream**()

.**mapToInt**(**Aluno**::getIdade)

.**summaryStatistics**();

// Exibindo as estatísticas

**System**.out.**println**("Idade média: " + stats.**getAverage**());

**System**.out.**println**("Mínima idade: " + stats.**getMin**());

**System**.out.**println**("Máxima idade: " + stats.**getMax**());

**System**.out.**println**("Total de alunos: " + stats.**getCount**());

}

}

Em resumo, as classes DoubleSummaryStatistics e IntSummaryStatistics são ferramentas úteis para facilitar a geração de estatísticas básicas em coleções de valores do tipo double e int. Elas fornecem métodos simples e eficientes.

**Uso avançado de Java Streams**

O Java Streams é uma característica poderosa que oferece a capacidade de realizar operações de processamento de dados complexas de forma eficiente e em paralelo, sobre collections, arrays e I/O channels. Quando você começa a entender o Java Streams melhor, você nota que ele pode ser usado em muitas maneiras diferentes para fazer seu código mais limpo e mais eficiente. Vamos olhar mais a fundo para os usos avançados de Java Streams e como você pode começar a integrá-los em seus projetos.

**Uso avançado de Java Streams**

Quando trabalhamos com Java Streams, muitas vezes vamos além do uso básico e adentramos em conceitos mais avançados e complexos. Vamos ver alguns exemplos:

**1 - Streams Infinitos:** Streams infinitos, ou “infinite Streams”, são streams que não têm um tamanho definido. Eles são úteis quando queremos gerar uma sequência de números ou valores. Aqui está um exemplo de como criamos um Stream infinito com o método iterate:

Stream.iterate(0, n -> n + 1)

.limit(10)

.forEach(System.out::println);

No exemplo acima, começamos com o número 0 e adicionamos 1 a cada iteração para gerar uma sequência numérica. Utilizamos o método limit() para restringir o Stream infinito a 10 elementos e usamos forEach para imprimi-los.

**2 – FlatMap:** O método flatMap é uma operação intermediária que é usada para transformar um Stream de coleções em um Stream de elementos. Aqui está um exemplo de como o flatMap é usado:

List<List<String>> list = List.of(

List.of("a", "b"),

List.of("c", "d")

);

Stream<String> stream = list.stream()

.flatMap(Collection::stream);

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

Neste exemplo, transformamos um Stream de List para um Stream de Strings.

**3 - Redução de Streams:** Stream.reduce() é uma operação terminal que é utilizada para reduzir o conteúdo de um Stream para um único valor.

List<Integer> numbers = List.of(1, 2, 3, 4, 5);

Optional<Integer> result = numbers.stream().reduce(Integer::sum);

result.ifPresent(System.out::println); //prints 15

No exemplo acima, somamos todos os números da lista usando o método reduce().

Lembre-se que o Java Streams é uma ferramenta poderosa que pode tornar seu código mais elegante e eficiente. Continue praticando e explorando todas as diferentes operações e métodos disponíveis para você com Java Streams para se tornar mais hábil em lidar com dados em suas aplicações.