Programação I

Streams e Funções de Alta Ordem

Samuel da Silva Feitosa

Aula 25
2022/1

Streams

- Uma das grandes adições do Java 8 foi o pacote java.util.stream destinado a execução de operações em massa:
  - filtragem (filter), mapeamento (map), redução (reduce) de coleções.
  - Funcionalidade também conhecida como aplicação de funções de alta ordem (higher-order functions).
- O conteúdo da coleção é percorrido sequencialmente, sem repetição de qualquer um de seus elementos, de modo que as transformações sejam aplicadas a todos.

Streams

- Streams são diferentes de coleções, pois:
  - Não armazenam os elementos, apenas transmitem sequencialmente o conteúdo de suas fontes.
  - Têm natureza funcional, dado que produzem resultados sem modificar suas fontes.
  - São consumíveis, ou seja, os elementos da fonte são visitados apenas uma vez durante o ciclo de vida do stream, requerendo um novo stream para que sejam revisitados.
- A forma mais simples de obter um stream é através do método stream() disponível nas classes derivadas de Collection<T>.

Navegação

- A navegação pelo conteúdo de coleções é uma tarefa frequentemente realizada.
  - Muito comum o emprego de Iterator<T> e laços de repetição, como vimos na aula anterior.

```
Iterator<?> iterator = colecao.iterator();
while (iterator.hasNext()) {
    System.out.println(iterator.next());
}
```

- Usando o novo método forEach(Consumer<T>), é possível simplificar a navegação de uma coleção usando uma expressão lambda.

```
colecao.forEach(e -> System.out.println(e));
```

Exemplo: Classe Pessoa

```java
public class Pessoa {
    private String nome;
    private String sobrenome;
    private Calendar dataNasc;
    public Pessoa(String nome, String sobrenome) {
        this.nome = nome;
        this.sobrenome = sobrenome;
        this.dataNasc = Calendar.getInstance();
    }
    public String getNome() {
        return nome;
    }
    public void setNome(String nome) {
        this.nome = nome;
    }
    public String getSobrenome() {
        return sobrenome;
    }
    public void setSobrenome(String sobrenome) {
        this.sobrenome = sobrenome;
    }
```

```java
    public String getDataNasc() {
        Date dataNasc = this.dataNasc.getTime();
        DateFormat f = DateFormat.getDateInstance(DateFormat.MEDIUM);
        return f.format(dataNasc);
    }
    public void setDataNasc(int dia, int mes, int ano) {
        this.dataNasc = Calendar.getInstance();
        this.dataNasc.set(ano, mes, dia);
    }
    @Override
    public String toString() {
        return "Nome: " + this.nome + " Sobrenome: " + this.sobrenome
                + " Data de Nascimento: " + this.getDataNasc();
    }
```

Exemplo Navegação

```
public class Navegacao {
    public static void mostraInfo(Pessoa p) {
        System.out.println("Informações da pessoa:");
        System.out.println("Nome = " + p.getNome());
        System.out.println("Sobrenome = " + p.getSobrenome());
        System.out.println("Data de Nascimento = " + p.getDataNasc());
    }
    public static void main(String[] args) {
        List<Pessoa> pessoas = new ArrayList<>();

        Pessoa p1 = new Pessoa("João", "Silva");
        Pessoa p2 = new Pessoa("Maria", "Santos");

        pessoas.add(p1);
        pessoas.add(p2);

        System.out.println("Mostrando todas as informações");
        pessoas.forEach(p -> System.out.println(p.toString()));
        System.out.println("Mostrando apenas nome e sobrenome");
        pessoas.forEach(p ->
                System.out.println(p.getNome() + " " + p.getSobrenome()));
        System.out.println("Usando referência para método");
        pessoas.forEach(Navegacao::mostraInfo);
    }
}
```

Filtragem

- A filtragem (filter) consiste na seleção de um subconjunto de elementos de uma estrutura de dados atendendo a determinado critério.

```
Iterator<Double> iterator = colecao.iterator();
while (iterator.hasNext()) {
    double elem = iterator.next();
    if (elem > LIMITE) {
        System.out.println(elem);
    }
}
```

- Um resultado similar poderia ser obtido usando um stream, o método filter e uma expr. lambda.

```
colecao.stream().filter(elem -> elem > LIMITE);
```

Exemplo Filtragem

```
public static void main(String[] args) {
    List<Double> lista = Arrays.asList(4.5, 0.3, 2.1, 0.8, 3.6);

    System.out.println(lista);
    // Define um limite para filtragem
    Double LIMITE = 1.5;
    // Obtém um stream a partir da lista
    Stream<Double> stream1 = lista.stream();
    // Aplica filtro com expressão lambda
    Stream<Double> maiores = stream1.filter(e -> e > LIMITE);
    // Mostrando as informações filtradas
    System.out.println("Maiores que o LIMITE");
    maiores.forEach(e -> System.out.println("\t" + e));
    // Obtém (novamente) stream a partir da lista
    Stream<Double> stream2 = lista.stream();
    // Aplica filtro com expr. lambda e adiciona a outra coleção
    List<Double> menores = stream2.filter(e -> e <= LIMITE)
                                  .collect(Collectors.toList());
    // Mostrando as informações com novo filtro
    System.out.println("Menores que o LIMITE");
    menores.forEach(e -> System.out.println("\t" + e));
}
```

Mapeamento

- O mapeamento (map) é a operação de transformar os elementos de uma coleção em outro valor, conforme definido em uma função.
  - Funções map, mapToDouble, mapToInt, mapToLong.

```
DoubleStream resultado = lista.stream().mapToDouble(e -> e / 2.0);
```

- Observa-se aqui o uso de mapToDouble.
  - Expressão lambda equivale a toDoubleFunction<T>.
  - Produz um valor double para cada elemento da lista.
  - Não usa Iterator e nem laços de repetição.
  - Simplifica e melhora a legibilidade do código.

Classe Par

```
public class Par {
    private String primeiro;
    private String segundo;

    public Par() { }
    public Par(String p, String s) {
        this.primeiro = p;
        this.segundo = s;
    }
    public void setPrimeiro(String p) {
        this.primeiro = p;
    }
    public void setSegundo(String s) {
        this.segundo = s;
    }
    public String getPrimeiro() {
        return this.primeiro;
    }
    public String getSegundo() {
        return this.segundo;
    }
}
```

Exemplo Mapeamento

```
public static void main(String[] args) {
    List<Integer> lista = Arrays.asList(1, 2, 3, 4, 5);

    System.out.println(lista);
    // Obtém stream dos quadrados dos elementos
    IntStream quadrados = lista.stream().mapToInt(e -> e * e);
    quadrados.forEach(e -> System.out.println("\t" + e));

    // Criando Stream de Pares
    Stream<Par> streamPares = Stream.of(new Par(81.5, 1.69),
                                        new Par(52.5, 1.62),
                                        new Par(72.1, 1.70));
    // Obtém stream dos IMCs does elementos
    DoubleStream imc = streamPares
        .mapToDouble(p -> p.getPrimeiro() / Math.pow(p.getSegundo(), 2));
    imc.forEach(e -> System.out.println("[imc] " + e));
}
```

Redução

- A redução (reduction ou fold) é uma operação que toma uma sequência de elementos de entrada aplicando uma função de combinação em cada um, produzindo um resultado final.
  - A função de combinação pode totalizar, determinar máximo, calcular alguma medida estatística, ou realizar outra operação desejada.
- Funções que podem ser utilizadas:
  - Em formas gerais, pode-se usar reduce.
  - Streams especializados como DoubleStream, IntStream, etc. permitem usar sum, max, count, etc.

Exemplo Redução

```
public static void main(String[] args) {
    List<Integer> lista = Arrays.asList(1, 2, 3, 4);
    int total = 0;
    // Totalização da lista
    for (int valor: lista) {
        total += valor;
    }
    System.out.println("Total = " + total);
    // Totalização usando sum
    int soma1 = lista.stream().mapToInt(e -> e).sum();
    System.out.println("Soma1 = " + soma1);
    // Mesma totalização usando reduce
    int soma2 = lista.stream().reduce(0, (acum, e) -> acum + e);
    System.out.println("Soma2 = " + soma2);
    // Contagem de elementos usando filtro
    long qtde = lista.stream().filter(e -> e > 1).count();
    System.out.println("Qtde pós filtro = " + qtde);
}
```

Considerações Finais

- Nesta aula estudamos mais uma funcionalidade avançada que foi incorporada a partir do Java 8.
- Streams permitem utilizar funções de alta ordem (conceito vindo das linguagens funcionais) para realizar diversos tipos de operações.
  - Navegação, filtragem, mapeamento e redução.
- Muitos sistemas e linguagens tem se baseado neste formato de codificação, pois além de compactos, permitem implementações que façam o código executar em paralelo.

Exercícios

1. Escreva um programa que leia números, reais ou inteiros, fornecidos pelo usuário, armazenando-os em uma implementação de Set<E>. Após a entrada de dados, use um stream para exibir todos os elementos fornecidos (navegação) e realizar a contagem de elementos (redução).
2. Escreva um programa que armazene nomes e telefones em um Map<K,V>. Utilizem streams para filtrar um telefone a partir de um nome (pesquisa exata) ou todos os nomes e telefones cujos nomes começam com um prefixo dado.