Dia 26 – Expressão Lambida

Conteudo

Introdução:

Vamos mostrar agora 8 casos ideias de como usar e porque usar Expressão lambda

- Abordagem 1: Criar métodos que buscam membros que correspondam a uma característica
- Abordagem 2: Crie métodos de pesquisa mais generalizados
- Abordagem 3: Especifique o código de critérios de pesquisa em uma classe local
- Abordagem 4: Especifique o código de critérios de pesquisa em uma classe anônima
- Abordagem 5: Especifique o código de critérios de pesquisa com uma expressão lambda
- Abordagem 6: Use interfaces funcionais padrão com expressões lambda
- Abordagem 7: Use expressões lambda ao longo de sua aplicação
- Abordagem 8: Use genéricos mais extensivamente
- Abordagem 9: Use operações agregadas que aceitam expressões lambda como parâmetros

Abordagem 1:

Abordagem 1: Criar métodos que buscam membros que correspondam a uma característica

Uma abordagem simplista é criar vários métodos; cada método procura por membros que correspondam a uma característica, como sexo ou idade. O método a seguir imprime membros com mais de idade do que uma idade especificada:

```
public static void printPersonsOlderThan(List<Person> roster, int age) {
```

```
   for (Person p : roster) {
      if (p.getAge() >= age) {
         p.printPerson();
      }
   }
}
```

Nota: Uma Lista é uma coleçãoordenada . Uma coleção é um objeto que agrupa vários elementos em uma única unidade. As coleções são usadas para armazenar, recuperar, manipular e comunicar dados agregados. Para obter mais informações sobre coletas, consulte a trilha de Coleções.

Essa abordagem pode potencialmente tornar seu aplicativo frágil,o que é a probabilidade de um aplicativo não funcionar por causa da introdução de atualizações (como tipos de dados mais novos). Suponha que você atualize seu aplicativo e altere a estrutura da classe de modo que contenha diferentes variáveis de membro; talvez os registros de classe e medidas idades com um tipo de dados ou algoritmo diferente. Você teria que reescrever muito de sua API para acomodar essa mudança. Além disso, essa abordagem é desnecessariamente restritiva; e se você quisesse imprimir membros mais jovens do que uma certa idade, por exemplo?Person

Abordagem 2: Crie métodos de pesquisa mais generalizados

O seguinte método é mais genérico do que; ele imprime membros dentro de uma faixa de idades especificada:`printPersonsOlderThan`

```
public static void printPersonsWithinAgeRange(
    List<Person> roster, int low, int high) {
    for (Person p : roster) {
        if (low <= p.getAge() && p.getAge() < high) {
            p.printPerson();
        }
    }
}
```

E se você quiser imprimir membros de um sexo especificado, ou uma combinação de um gênero e faixa etária especificados? E se você decidir mudar a classe e adicionar outros atributos, como status de relacionamento ou localização geográfica? Embora este método seja mais genérico do que , tentar criar um método separado para cada possível consulta de pesquisa ainda pode levar a código frágil. Em vez disso, você pode separar o código que especifica os critérios para os quais você deseja pesquisar em uma classe diferente.`PersonprintPersonsOlderThan`

Abordagem 3: Especifique o código de critérios de pesquisa em uma classe local

O método a seguir imprime membros que correspondem aos critérios de pesquisa que você especifica:

```
public static void printPersons(
    List<Person> roster, CheckPerson tester) {
    for (Person p : roster) {
        if (tester.test(p)) {
            p.printPerson();
        }
```

```
        }
    }
```

Este método verifica cada instância contida no parâmetro se satisfaz os critérios de pesquisa especificados no parâmetro invocando o método . Se o método retornar um valor, então o método será invocado na instância.`PersonListrosterCheckPersontestertester.testtester.testtrueprint PersonsPerson`

Para especificar os critérios de pesquisa, você implementa a interface:`CheckPerson`

```
interface CheckPerson {
    boolean test(Person p);
}
```

A classe a seguir implementa a interface especificando uma implementação para o método . Este método filtra os membros elegíveis para o Serviço Seletivo nos Estados Unidos: ele retorna um valor se seu parâmetro for masculino e entre as idades de 18 e 25 anos:`CheckPersontesttruePerson`

```
class CheckPersonEligibleForSelectiveService implements CheckPerson {
    public boolean test(Person p) {
        return p.gender == Person.Sex.MALE &&
            p.getAge() >= 18 &&
            p.getAge() <= 25;
    }
}
```

Para usar esta classe, você cria uma nova instância dela e invoca o método:`printPersons`

```
printPersons(
    roster, new CheckPersonEligibleForSelectiveService());
```

Embora essa abordagem seja menos frágil — você não precisa reescrever métodos se você alterar a estrutura do — você ainda tem um código adicional: uma nova interface e uma classe local para cada pesquisa que você planeja realizar em seu aplicativo. Como implementa uma interface, você pode usar uma classe anônima em vez de uma classe local e ignorar a necessidade de declarar uma nova classe para cada pesquisa.`PersonCheckPersonEligibleForSelectiveService`

Abordagem 4: Especifique o código de critérios de pesquisa em uma classe anônima

Um dos argumentos da seguinte invocação do método é uma classe anônima que filtra membros elegíveis para o Serviço Seletivo nos Estados Unidos: aqueles que são homens e entre 18 e 25 anos:`printPersons`

```
printPersons(
    roster,
    new CheckPerson() {
        public boolean test(Person p) {
            return p.getGender() == Person.Sex.MALE
                && p.getAge() >= 18
                && p.getAge() <= 25;
        }
    }
);
```

Essa abordagem reduz a quantidade de código necessária porque você não precisa criar uma nova classe para cada pesquisa que deseja realizar. No entanto, a sintaxe de classes anônimas é volumosa considerando que a interface contém apenas um método. Neste caso, você pode usar

uma expressão lambda em vez de uma classe anônima, como descrito na próxima seção.`CheckPerson`

Abordagem 5: Especifique o código de critérios de pesquisa com uma expressão lambda

A interface é uma *interface funcional.* Uma interface funcional é qualquer interface que contenha apenas um método abstrato. (Uma interface funcional pode conter um ou mais métodos padrão ou estáticos.) Como uma interface funcional contém apenas um método abstrato, você pode omitir o nome desse método quando o implementar. Para fazer isso, em vez de usar uma expressão de classe anônima, você usa uma *expressão lambda*, que é destacada no seguinte método de invocação:`CheckPerson`

```
printPersons(
    roster,
    (Person p) -> p.getGender() == Person.Sex.MALE
        && p.getAge() >= 18
        && p.getAge() <= 25
);
```

Consulte Syntax of Lambda Expressions para obter informações sobre como definir expressões lambda.

Você pode usar uma interface funcional padrão no lugar da interface, o que reduz ainda mais a quantidade de código necessária.`CheckPerson`

Abordagem 6: Use interfaces funcionais padrão com expressões lambda

Reconsidere a interface:`CheckPerson`

```
interface CheckPerson {
    boolean test(Person p);
}
```

Esta é uma interface muito simples. É uma interface funcional porque contém apenas um método abstrato. Este método pega um parâmetro e retorna um valor. O método é tão simples que pode não valer a pena definir um em sua aplicação. Consequentemente, o JDK define várias interfaces funcionais padrão, que você pode encontrar no pacote .`booleanjava.util.function`

Por exemplo, você pode usar a interface no lugar de . Esta interface contém o método:`Predicate<T>CheckPersonboolean test(T t)`

```
interface Predicate<T> {
    boolean test(T t);
}
```

A interface é um exemplo de interface genérica. (Para obter mais informações sobre genéricos, consulte a lição Genéricos (Atualizados). Os tipos genéricos (como interfaces genéricas) especificam um ou mais parâmetros de tipo dentro dos suportes angulares (). Esta interface contém apenas um parâmetro de tipo, . Quando você declara ou instancia um tipo genérico com argumentos de tipo real, você tem um tipo parametrizado. Por exemplo, o tipo parametrizado é o seguinte:`Predicate<T><>TPredicate<Person>`

```
interface Predicate<Person> {
    boolean test(Person t);
}
```

Este tipo parametrizado contém um método que tem o mesmo tipo de retorno e parâmetros que . Consequentemente, você pode usar no lugar de como o seguinte método demonstra:`CheckPerson.boolean test(Person p)Predicate<T>CheckPerson`

```
public static void printPersonsWithPredicate(
    List<Person> roster, Predicate<Person> tester) {
    for (Person p : roster) {
        if (tester.test(p)) {
            p.printPerson();
        }
    }
}
```

Como resultado, a invocação do método a seguir é a mesma de quando você invocou na Abordagem 3: Especificar código de critérios de pesquisa em uma classe local para obter membros elegíveis para serviço seletivo:`printPersons`

```
printPersonsWithPredicate(
    roster,
    p -> p.getGender() == Person.Sex.MALE
        && p.getAge() >= 18
        && p.getAge() <= 25
);
```

Este não é o único lugar possível neste método para usar uma expressão lambda. A abordagem a seguir sugere outras formas de usar expressões lambda.

Abordagem 7: Use expressões lambda ao longo de sua aplicação

Reconsidere o método para ver onde mais você poderia usar expressões lambda:`printPersonsWithPredicate`

```
public static void printPersonsWithPredicate(
    List<Person> roster, Predicate<Person> tester) {
    for (Person p : roster) {
        if (tester.test(p)) {
            p.printPerson();
        }
    }
}
```

Este método verifica cada instância contida no parâmetro se satisfaz os critérios especificados no parâmetro . Se a instância satisfazer os critérios especificados, o método será invocado na instância.`PersonListrosterPredicatetesterPersontesterprintPersronPerson`

Em vez de invocar o método, você pode especificar uma ação diferente para executar nessas instâncias que satisfaçam os critérios especificados por . Você pode especificar esta ação com uma expressão lambda. Suponha que você queira uma expressão lambda semelhante a , uma que pega um argumento (um objeto de tipo) e retorna o vazio. Lembre-se, para usar uma expressão lambda, você precisa implementar uma interface funcional. Neste caso, você precisa de uma interface funcional que contenha um método abstrato que pode levar um argumento de tipo e devoluções anuladas. A interface contém o método, que tem essas características. O método a seguir substitui a invocação por uma instância que invoca o método `:printPersonPersontesterprintPersonPersonPersonConsumer<T>void accept(T t)p.printPerson()Consumer<Person>accept`

```
public static void processPersons(
    List<Person> roster,
    Predicate<Person> tester,
    Consumer<Person> block) {
        for (Person p : roster) {
            if (tester.test(p)) {
```

```
                block.accept(p);
            }
        }
    }
```

Como resultado, a invocação do método a seguir é a mesma de quando você invocou na Abordagem 3: Especificar código de critérios de pesquisa em uma classe local para obter membros elegíveis para o Serviço Seletivo. Destaca-se a expressão lambda usada para imprimir membros:`printPersons`

```
processPersons(
     roster,
     p -> p.getGender() == Person.Sex.MALE
         && p.getAge() >= 18
         && p.getAge() <= 25,
     p -> p.printPerson()
);
```

E se você quiser fazer mais com os perfis de seus membros do que imprimi-los. Suponha que você deseja validar os perfis dos membros ou recuperar suas informações de contato? Neste caso, você precisa de uma interface funcional que contenha um método abstrato que retorne um valor. A interface contém o método . O método a seguir recupera os dados especificados pelo parâmetro e, em seguida, executa uma ação nele especificada pelo parâmetro:`Function<T,R>R apply(T t)mapperblock`

```
public static void processPersonsWithFunction(
    List<Person> roster,
    Predicate<Person> tester,
    Function<Person, String> mapper,
    Consumer<String> block) {
    for (Person p : roster) {
        if (tester.test(p)) {
            String data = mapper.apply(p);
            block.accept(data);
        }
    }
}
```

O método a seguir recupera o endereço de e-mail de cada membro contido em quem é elegível para o Serviço Seletivo e, em seguida, imprime-o:`roster`

```
processPersonsWithFunction(
    roster,
    p -> p.getGender() == Person.Sex.MALE
        && p.getAge() >= 18
        && p.getAge() <= 25,
    p -> p.getEmailAddress(),
    email -> System.out.println(email)
);
```

Abordagem 8: Use genéricos mais extensivamente

Reconsidere o método. A seguir, uma versão genérica dele que aceita, como parâmetro, uma coleção que contém elementos de qualquer tipo de dados:`processPersonsWithFunction`

```
public static <X, Y> void processElements(
    Iterable<X> source,
    Predicate<X> tester,
    Function <X, Y> mapper,
    Consumer<Y> block) {
    for (X p : source) {
        if (tester.test(p)) {
            Y data = mapper.apply(p);
```

```
                block.accept(data);
            }
        }
    }
```

Para imprimir o endereço de e-mail dos membros elegíveis para o Serviço Seletivo, invoque o método da seguinte forma:`processElements`

```
processElements(
    roster,
    p -> p.getGender() == Person.Sex.MALE
        && p.getAge() >= 18
        && p.getAge() <= 25,
    p -> p.getEmailAddress(),
    email -> System.out.println(email)
);
```

Esta invocação do método realiza as seguintes ações:

1. Obtém uma fonte de objetos da coleção. Neste exemplo, obtém uma fonte de objetos da coleção. Observe que a coleção , que é uma coleção de tipo , também é um objeto do tipo.`sourcePersonrosterrosterListIterable`
2. Filtra objetos que correspondem ao objeto . Neste exemplo, o objeto é uma expressão lambda que especifica quais membros seriam elegíveis para o Serviço Seletivo.`PredicatetesterPredicate`
3. Mapeia cada objeto filtrado para um valor especificado pelo objeto . Neste exemplo, o objeto é uma expressão lambda que retorna o endereço de e-mail de um membro.`FunctionmapperFunction`
4. Realiza uma ação em cada objeto mapeado conforme especificado pelo objeto . Neste exemplo, o objeto é uma expressão lambda que imprime uma string, que é o endereço de e-mail devolvido pelo objeto.`ConsumerblockConsumerFunction`

Você pode substituir cada uma dessas ações por uma operação agregada.

Vídeo

https://www.youtube.com/watch?v=lbCYLgoVpfQ

Exercício

1) Dado:
   1. List<Integer> list = Arrays.asList(1, 2, 3, 4, 5, 6, 7);
   Qual das alternativa atende melhor a expressão lambda retornando todos os dados
   a) list.forEach(n -> System.out.println(n));
   b) list for(int i=0;i<list.length();i++){ System.out.println(list[i]);}
   c)System.out.println(list);
   d)nem uma das alternativas

2) Dado: Imprimindo apenas os elementos pares de uma lista
   List<Integer> list = Arrays.asList(1, 2, 3, 4, 5, 6, 7);
   Qual das alternativas atende melhor a expressão lambda retornando todos os dados pares
   Qual das alternativa atende melhor a expressão lambda retornando todos os dados
   a) list.forEach(n -> System.out.println(n));
   b) list for(int i=0;i<list.length();i++){ System.out.println(list[i]);}
   c) list.forEach(n -> {
      if (n % 2 == 0) {
         System.out.println(n);
      }

});
d)nem uma das alternativas

3) Dado: Imprimindo apenas os elementos pares de uma lista
List<Integer> list = Arrays.asList(1, 2, 3, 4, 5, 6, 7);
Qual das alternativas atende melhor a expressão lambda retornando todos os dados pares
Qual das alternativa atende melhor a expressão lambda retornando todos os dados
a) list.forEach(n -> System.out.println(n));
b) list for(int i=0;i<list.length();i++){ System.out.println(list[i]);}
c) list.forEach(n -> System.out.println(n * n));
d)nem uma das alternativas

Prova

1) Qual desssas alternativas usa filtros com função lambda
a) List<Pessoa> nomesIniciadosE = listPessoas.stream().filter(p
-> p.getNome().startsWith("E")).collect(Collectors.toList());
nomesIniciadosE.forEach(p -> System.out.println(p.getNome()));
b) List<Pessoa> nomesIniciadosE = listPessoas.stream().folter(p
-> p.getNome().startsWith("E")).collect(Collectors.toList());
nomesIniciadosE.forEach(p -> System.out.println(p.getNome()));
c) List<Pessoa> nomesIniciadosE = listPessoas.stream().fulter(p
-> p.getNome().startsWith("E")).collect(Collectors.toList());
nomesIniciadosE.forEach(p -> System.out.println(p.getNome()));
d) List<Pessoa> nomesIniciadosE = listPessoas.stream().CompareTo(p
-> p.getNome().startsWith("E")).collect(Collectors.toList());
nomesIniciadosE.forEach(p -> System.out.println(p.getNome()));

2) Dado: Imprimindo apenas os elementos pares de uma lista
List<Integer> list = Arrays.asList(1, 2, 3, 4, 5, 6, 7);
Qual das alternativas atende melhor a expressão lambda retornando todos os dados pares
Qual das alternativa atende melhor a expressão lambda retornando todos os dados
a) list.forEach(n -> System.out.println(n));
b) list for(int i=0;i<list.length();i++){ System.out.println(list[i]);}
c) list.forEach(n -> System.out.println(n * n));
d)nem uma das alternativas

3)Qual das alternativas de uso das alternativas esta usando expressão lambda incorretamente.

A) (int a, int b) -> { return a + b; }
B) () -> System.out.println("Hello World");
C) (String s) -> { System.out.println(s); }
D) Nem uma das