**Expressões do Lambda**

Um problema com classes anônimas é que se a implementação de sua classe anônima é muito simples, como uma interface que contém apenas um método, então a sintaxe de classes anônimas pode parecer difícil e pouco clara. Nesses casos, você geralmente está tentando passar a funcionalidade como um argumento para outro método, como qual ação deve ser tomada quando alguém clica em um botão. As expressões do Lambda permitem que você faça isso, trate a funcionalidade como argumento de método ou código como dados.

A seção anterior, [Classes Anônimas](https://docs.oracle.com/javase/tutorial/java/javaOO/anonymousclasses.html), mostra como implementar uma classe base sem dar um nome a ela. Embora isso seja geralmente mais conciso do que uma classe nomeada, para classes Com apenas um método, até mesmo uma classe anônima parece um pouco excessivo e pesado. As expressões do Lambda permitem expressar instâncias de classes de método único de forma mais compacta.

Esta seção aborda os seguintes tópicos:

* [Caso de uso ideal para expressões Lambda](https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html#use-case)
  + [Abordagem 1: Criar métodos que procuram membros que correspondem a uma característica](https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html#approach1)
  + [Abordagem 2: Criar métodos de pesquisa mais generalizados](https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html#approach2)
  + [Abordagem 3: Especificar código de critérios de pesquisa em uma classe local](https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html#approach3)
  + [Abordagem 4: Especificar código de critérios de pesquisa em uma classe anônima](https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html#approach4)
  + [Abordagem 5: Especificar código de critérios de pesquisa com uma expressão do Lambda](https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html#approach5)
  + [Abordagem 6: Usar interfaces funcionais padrão com expressões do Lambda](https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html#approach6)
  + [Abordagem 7: Usar expressões do Lambda em todo o aplicativo](https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html#approach7)
  + [Abordagem 8: Use genéricos mais extensivamente](https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html#approach8)
  + [Abordagem 9: Usar operações agregadas que aceitam expressões do Lambda como parâmetros](https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html#approach9)
* [Expressões do Lambda em aplicativos GUI](https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html#lambda-expressions-in-gui-applications)
* [Sintaxe de expressões Lambda](https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html#syntax)
* [Acessando variáveis locais do escopo de inclusão](https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html#accessing-local-variables)
* [Digitação de destino](https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html#target-typing)
  + [Tipos de destino e argumentos de método](https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html#target-types-and-method-arguments)
* [Serialização](https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html#serialization)

**Caso de uso ideal para expressões Lambda**

Suponha que você esteja criando um aplicativo de rede social. Você deseja criar um recurso que permita que um administrador execute qualquer tipo de ação, como o envio de uma mensagem, sobre os membros do aplicativo de rede social que satisfazem certos critérios. A tabela a seguir descreve esse caso de uso em detalhes:

|  |  |
| --- | --- |
| **Campo** | **Descrição** |
| Nome | Executar ação em membros selecionados |
| Ator Principal | Administrador |
| Pré-condições | O administrador está conectado ao sistema. |
| Pós-condições | A ação é executada somente em membros que se encaixam nos critérios especificados. |
| Principal Cenário de Sucesso | 1. O administrador especifica os critérios dos membros para executar uma determinada ação. 2. O administrador especifica uma ação a ser executada nesses membros selecionados. 3. O administrador seleciona o botão **Enviar**. 4. O sistema localiza todos os membros que correspondem aos critérios especificados. 5. O sistema executa a ação especificada em todos os membros correspondentes. |
| Extensões | 1º. O administrador tem a opção de visualizar os membros que correspondem aos critérios especificados antes de especificar a ação a ser executada ou antes de selecionar o botão **Enviar**. |
| Frequência de Ocorrência | Muitas vezes durante o dia. |

Suponha que os membros deste aplicativo de rede social sejam representada pela seguinte classe [Person](https://docs.oracle.com/javase/tutorial/java/javaOO/examples/Person.java):

public class Person {

public enum Sex {

MALE, FEMALE

}

String name;

LocalDate birthday;

Sex gender;

String emailAddress;

public int getAge() {

// ...

}

public void printPerson() {

// ...

}

}

Suponha que os membros do seu aplicativo de rede social são armazenados em uma instância.List<Person>

Esta seção começa com uma abordagem ingênua para este caso de uso. Ele melhora essa abordagem com classes locais e anônimas e, em seguida, termina com uma abordagem eficiente e concisa usando expressões lambda. Encontre os trechos de código descritos nesta seção no exemplo [RosterTest](https://docs.oracle.com/javase/tutorial/java/javaOO/examples/RosterTest.java" \t "_blank).

**Abordagem 1: Criar métodos que procuram membros que correspondem a uma característica**

Uma abordagem simplista é criar vários métodos; Cada método procura membros que correspondam a uma característica, como sexo ou idade. O método a seguir imprime membros que são mais antigos do que um especificado idade:

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

for (Person p : roster) {

if (p.getAge() >= age) {

p.printPerson();

}

}

}

**Nota**: Uma [Lista](https://docs.oracle.com/javase/8/docs/api/java/util/List.html) é uma [Coleção](https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html) ordenada. Uma *coleção* é um objeto que agrupa vários elementos em uma única unidade. As coleções são usado para armazenar, recuperar, manipular e comunicar agregados dados. Para obter mais informações sobre coleções, consulte a trilha [Coleções](https://docs.oracle.com/javase/tutorial/collections/index.html).

Essa abordagem pode potencialmente tornar seu aplicativo *frágil*, que é a probabilidade de um aplicativo não funcionar devido à introdução de atualizações (como tipos de dados mais recentes). Suponha que você atualize seu aplicativo e altere a estrutura da classe de modo que ela contenha variáveis de membro diferentes; Talvez a classe registre e meça idades com um tipo de dados ou algoritmo diferente. Você teria que reescrever muito da sua API para acomodar essa alteração. Além disso, essa abordagem é desnecessariamente restritiva; e se você Queria imprimir membros com menos de uma certa idade, por exemplo?Person

**Abordagem 2: Criar métodos de pesquisa mais generalizados**

O método a seguir é mais genérico que ; Ele imprime membros dentro de uma faixa etária 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 específico ou uma combinação de um gênero e faixa etária especificados? E se você decidir alterar a classe e adicionar outros atributos, como status de relacionamento ou localização geográfica? Embora esse método seja mais genérico do que o , tentar criar um método separado para cada possível consulta de pesquisa ainda pode levar a um código frágil. Em vez disso, você pode separar o código que especifica os critérios para os quais deseja pesquisar em uma classe diferente.PersonprintPersonsOlderThan

**Abordagem 3: Especificar código de critérios de pesquisa em uma classe local**

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

public static void printPersons(

List<Person> roster, CheckPerson tester) {

for (Person p : roster) {

if (tester.test(p)) {

p.printPerson();

}

}

}

Esse método verifica cada instância contida no parâmetro se ela satisfaz os critérios de pesquisa especificados no parâmetro invocando o método . Se o método retornar um valor, o método será chamado na instância.PersonListrosterCheckPersontestertester.testtester.testtrueprintPersonsPerson

Para especificar os critérios de pesquisa, implemente a interface:CheckPerson

interface CheckPerson {

boolean test(Person p);

}

A classe a seguir implementa a interface especificando uma implementação para o método . Esse método filtra os membros qualificados para o Serviço Seletivo nos Estados Unidos: ele retorna um valor se seu parâmetro for masculino e tiver entre 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 essa classe, você cria um novo instância dele e invoque o método:printPersons

printPersons(

roster, new CheckPersonEligibleForSelectiveService());

Embora essa abordagem seja menos frágil — você não precisa reescrever métodos se alterar a estrutura do — você ainda tem código adicional: uma nova interface e uma classe local para cada pesquisa que planeja executar 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: Especificar código de critérios de pesquisa em uma classe anônima**

Um dos argumentos da seguinte invocação do método é uma classe anônima que filtra os membros elegíveis para o Serviço Seletivo nos Estados Unidos: aqueles que são do sexo masculino e têm 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 executar. No entanto, a sintaxe de classes anônimas é volumosa, considerando que a interface contém apenas um método. Neste maiúsculas e minúsculas, você pode usar uma expressão lambda em vez de uma classe anônima, como descrito na próxima seção.CheckPerson

**Abordagem 5: Especificar código de critérios de pesquisa com uma expressão do Lambda**

A interface é uma *interface funcional*. Um funcional interface é qualquer interface que contém apenas um [método abstrato](https://docs.oracle.com/javase/tutorial/java/IandI/abstract.html). (Uma interface funcional pode conter um ou mais métodos [padrão](https://docs.oracle.com/javase/tutorial/java/IandI/defaultmethods.html) ou [métodos estáticos](https://docs.oracle.com/javase/tutorial/java/IandI/defaultmethods.html#static).) Porque Uma interface funcional contém apenas um método abstrato, você pode omitir o nome desse método quando você implementá-lo. Para fazer isso, em vez de usar um anônimo expressão de classe, você usa um *lambda expressão*, que é destacado na seguinte chamada de método:CheckPerson

printPersons(

roster,

**(Person p) -> p.getGender() == Person.Sex.MALE**

**&& p.getAge() >= 18**

**&& p.getAge() <= 25**

);

Consulte Sintaxe de expressões lambda para obter informações sobre como definir [expressões lambda](https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html#syntax).

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

**Abordagem 6: Usar interfaces funcionais padrão com expressões do Lambda**

Reconsidere a interface:CheckPerson

interface CheckPerson {

boolean test(Person p);

}

Este é um interface simples. É uma interface funcional porque contém apenas um método abstrato. Esse método usa um parâmetro e retorna um valor. O método é tão simples que pode não valer a pena ele para definir um em seu aplicativo. 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 . Este contém o método :Predicate<T>CheckPersonboolean test(T t)

interface Predicate<T> {

boolean test(T t);

}

A interface é um exemplo de uma interface genérica. (Para obter mais informações sobre genéricos, consulte a lição [Genéricos (Atualizado](https://docs.oracle.com/javase/tutorial/java/generics/index.html)).) Tipos genéricos (como interfaces genéricas) especificam um ou mais parâmetros de tipo entre colchetes angulares (). Essa 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);

}

Esse 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: Especifique o Código de Critérios de Pesquisa em uma Classe Local](https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html#approach3) para obter membros qualificados para o 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 maneiras de usar expressões lambda.

**Abordagem 7: Usar expressões do Lambda em todo o aplicativo**

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();

}

}

}

Esse método verifica cada instância contida no parâmetro se ela satisfaz os critérios especificados no parâmetro . Se a instância satisfizer os critérios especificados pelo , o método será chamado na instância.PersonListrosterPredicatetesterPersontesterprintPersonPerson

Em vez de invocar o método , você pode especificar uma ação diferente a ser executada nas instâncias que satisfazem os critérios especificados pelo . Você pode especificar essa ação com uma expressão lambda. Suponha que você queira uma expressão lambda semelhante a , que usa um argumento (um objeto do tipo ) e retorna void. Lembre-se, para usar uma expressão lambda, você precisa implementar uma interface funcional. Nesse caso, você precisa de uma interface funcional que contenha um método abstrato que possa usar um argumento de tipo e retornar void. A interface contém o método , que tem essas características. O O método a seguir substitui a invocação por um instância de 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 um 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](https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html#approach3) para obter membros qualificados para o Serviço Seletivo. A expressão lambda usada para Imprimir membros é destacado: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 queira validar os perfis dos membros ou recuperar suas informações de contato? Neste caso, você precisa de um que contém um método abstrato que retorna um valor. A interface contém o método. O método a seguir recupera os dados especificado 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 email 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 está 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 email dos membros qualificados para o Serviço Seletivo, chame o método da seguinte maneira:processElements

processElements(

roster,

p -> p.getGender() == Person.Sex.MALE

&& p.getAge() >= 18

&& p.getAge() <= 25,

p -> p.getEmailAddress(),

email -> System.out.println(email)

);

Essa invocação de método executa as seguintes ações:

1. Obtém uma origem de objetos da coleção . Neste exemplo, ele obtém uma origem de objetos da coleção . Observe que a coleção , que é uma coleção do 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 qualificados para o Serviço Seletivo.PredicatetesterPredicate
3. Mapeia cada objeto filtrado para um valor conforme especificado pelo objeto . Neste exemplo, o objeto é uma expressão lambda que retorna o endereço de email de um membro.FunctionmapperFunction
4. Executa uma ação em cada objeto mapeado conforme especificado pelo objeto . Neste exemplo, o objeto é uma expressão lambda que imprime uma cadeia de caracteres, que é o endereço de email retornado pelo objeto.ConsumerblockConsumerFunction

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

**Abordagem 9: Usar operações agregadas que aceitam expressões do Lambda como parâmetros**

O exemplo a seguir usa operações agregadas para imprimir os endereços de email dos membros contidos na coleção que são qualificados para o Serviço Seletivo:roster

roster

.stream()

.filter(

p -> p.getGender() == Person.Sex.MALE

&& p.getAge() >= 18

&& p.getAge() <= 25)

.map(p -> p.getEmailAddress())

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

A tabela a seguir mapeia cada uma das operações que o método executa com a operação agregada correspondente:processElements

|  |  |
| --- | --- |
| **processElements Ação** | **Operação Agregada** |
| Obter uma origem de objetos | Stream<E> **stream**() |
| Filtrar objetos que correspondem a um objetoPredicate | Stream<T> **filter**(Predicate<? super T> predicate) |
| Mapeie objetos para outro valor, conforme especificado por um objetoFunction | <R> Stream<R> **map**(Function<? super T,? extends R> mapper) |
| Executar uma ação conforme especificado por um objetoConsumer | void **forEach**(Consumer<? super T> action) |

As operações , e são *operações agregadas*. As operações agregam elementos de processo de um fluxo, não diretamente de uma coleção (razão pela qual o primeiro método invocado neste exemplo é ). Um *fluxo* é uma sequência de elementos. Ao contrário de uma coleção, ela não é uma estrutura de dados que armazena elementos. Em vez disso, um fluxo carrega valores de uma origem, como coleção, por meio de um pipeline. Um *pipeline* é uma sequência de operações de fluxo, que neste exemplo é - -. Além disso, as operações agregadas normalmente aceitam expressões lambda como parâmetros, permitindo que você personalize como elas se comportam.filtermapforEachstreamfiltermapforEach

Para obter uma discussão mais completa das operações agregadas, consulte a lição [Operações agregadas](https://docs.oracle.com/javase/tutorial/collections/streams/index.html).

**Expressões do Lambda em aplicativos GUI**

Para processar eventos em um aplicativo de interface gráfica do usuário (GUI), como teclado ações, ações do mouse e ações de rolagem, que você normalmente cria manipuladores de eventos, que geralmente envolve a implementação de um determinado interface. Muitas vezes, as interfaces do manipulador de eventos são funcionais Interfaces; eles tendem a ter apenas um método.

No exemplo JavaFX [HelloWorld.java](https://docs.oracle.com/javase/8/javafx/get-started-tutorial/hello_world.htm) (discutido na seção anterior [Classes anônimas](https://docs.oracle.com/javase/tutorial/java/javaOO/anonymousclasses.html)), você pode Substitua a classe anônima realçada por uma expressão lambda neste declaração:

btn.setOnAction(**new EventHandler<ActionEvent>() {**

**@Override**

**public void handle(ActionEvent event) {**

**System.out.println("Hello World!");**

**}**

**}**);

A invocação do método especifica o que acontece quando você seleciona o botão representado pelo objeto. Este método requer um objeto do tipo . A interface contém apenas um método, . Essa interface é uma interface funcional, portanto, você pode usar a seguinte expressão lambda realçada para substituí-la:btn.setOnActionbtnEventHandler<ActionEvent>EventHandler<ActionEvent>void handle(T event)

btn.setOnAction(

**event -> System.out.println("Hello World!")**

);

**Sintaxe de expressões Lambda**

Uma expressão lambda consiste no seguinte:

* Um Lista separada por vírgulas de parâmetros formais incluída em parênteses. O método contém um parâmetro, , que representa uma instância da classe.CheckPerson.testpPerson

**Nota**: Você pode Omita o tipo de dados dos parâmetros em uma expressão do Lambda. Em Além disso, você pode omitir os parênteses se houver apenas um parâmetro. Por exemplo, a seguinte expressão lambda também é válido:

p -> p.getGender() == Person.Sex.MALE

&& p.getAge() >= 18

&& p.getAge() <= 25

* O token de seta, ->
* Um corpo, que consiste em uma única expressão ou um bloco de instrução. Este exemplo usa a seguinte expressão:
* p.getGender() == Person.Sex.MALE
* && p.getAge() >= 18

&& p.getAge() <= 25

Se você especifique uma única expressão e, em seguida, o Java runtime avalia o e, em seguida, retorna seu valor. Alternativamente, você pode usar uma declaração de retorno:

p -> {

return p.getGender() == Person.Sex.MALE

&& p.getAge() >= 18

&& p.getAge() <= 25;

}

Uma instrução return não é uma expressão; em uma expressão lambda, você deve colocar instruções entre chaves (). No entanto, não é necessário incluir uma invocação de método void em chaves. Por exemplo, a seguir está uma expressão lambda válida:{}

email -> System.out.println(email)

Observe que uma expressão lambda se parece muito com um método declaração; Você pode considerar expressões do Lambda como anônimas Métodos — Métodos sem nome.

O exemplo a seguir, [Calculadora](https://docs.oracle.com/javase/tutorial/java/javaOO/examples/Calculator.java), é um exemplo de expressões lambda que usam mais de um parâmetro formal:

public class Calculator {

interface IntegerMath {

int operation(int a, int b);

}

public int operateBinary(int a, int b, IntegerMath op) {

return op.operation(a, b);

}

public static void main(String... args) {

Calculator myApp = new Calculator();

IntegerMath addition = (a, b) -> a + b;

IntegerMath subtraction = (a, b) -> a - b;

System.out.println("40 + 2 = " +

myApp.operateBinary(40, 2, addition));

System.out.println("20 - 10 = " +

myApp.operateBinary(20, 10, subtraction));

}

}

O método executa um operação matemática em dois operandos inteiros. A operação é especificado por uma instância de . O exemplo define duas operações com expressões lambda e . O exemplo imprime o seguinte:operateBinaryIntegerMathadditionsubtraction

40 + 2 = 42

20 - 10 = 10

**Acessando variáveis locais do escopo de inclusão**

Como classes locais e anônimas, as expressões lambda podem [capturar variáveis](https://docs.oracle.com/javase/tutorial/java/javaOO/localclasses.html#accessing-members-of-an-enclosing-class); eles têm o mesmo acesso às variáveis locais do escopo de encerramento. No entanto, ao contrário das classes locais e anônimas, as expressões lambda não têm problemas de [sombreamento (consulte Sombreamento](https://docs.oracle.com/javase/tutorial/java/javaOO/nested.html#shadowing) para obter mais informações). As expressões lambda têm escopo lexical. Isso significa que eles não herdam nenhum nome de um supertipo ou introduzem um novo nível de escopo. As declarações em uma expressão lambda são interpretadas exatamente como no ambiente de encerramento. O exemplo a seguir, [LambdaScopeTest](https://docs.oracle.com/javase/tutorial/java/javaOO/examples/LambdaScopeTest.java" \t "_blank), demonstra isso:

import java.util.function.Consumer;

public class LambdaScopeTest {

public int x = 0;

class FirstLevel {

public int x = 1;

void methodInFirstLevel(int x) {

int z = 2;

Consumer<Integer> myConsumer = (y) ->

{

// The following statement causes the compiler to generate

// the error "Local variable z defined in an enclosing scope

// must be final or effectively final"

//

// z = 99;

System.out.println("x = " + x);

System.out.println("y = " + y);

System.out.println("z = " + z);

System.out.println("this.x = " + this.x);

System.out.println("LambdaScopeTest.this.x = " +

LambdaScopeTest.this.x);

};

myConsumer.accept(x);

}

}

public static void main(String... args) {

LambdaScopeTest st = new LambdaScopeTest();

LambdaScopeTest.FirstLevel fl = st.new FirstLevel();

fl.methodInFirstLevel(23);

}

}

Este exemplo gera a seguinte saída:

x = 23

y = 23

z = 2

this.x = 1

LambdaScopeTest.this.x = 0

Se você substituir o parâmetro no lugar de na declaração da expressão lambda , o compilador gerará um erro:xymyConsumer

Consumer<Integer> myConsumer = (x) -> {

// ...

}

O compilador gera o erro "O parâmetro x da expressão do Lambda não pode redeclarar outra variável local definida em um escopo de delimitação" porque a expressão lambda não introduz um novo nível de escopo. Consequentemente, você pode acessar diretamente campos, métodos e variáveis locais do escopo de delimitação. Por exemplo, a expressão lambda acessa diretamente o parâmetro do método . Para acessar variáveis na classe de delimitação, use a palavra-chave . Neste exemplo, refere-se à variável de membro .xmethodInFirstLevelthisthis.xFirstLevel.x

No entanto, como classes locais e anônimas, uma expressão lambda só pode acessar variáveis locais e parâmetros do bloco de delimitação que são finais ou efetivamente finais. Neste exemplo, a variável é efetivamente final; seu valor nunca é alterado após ser inicializado. No entanto, suponha que você adicione a seguinte instrução de atribuição na expressão lambda :zmyConsumer

Consumer<Integer> myConsumer = (y) -> {

**z = 99;**

// ...

}

Devido a essa instrução de atribuição, a variável não é mais efetivamente final. Como resultado, o compilador Java gera uma mensagem de erro semelhante a "A variável local z definida em um escopo de inclusão deve ser final ou efetivamente final".z

**Digitação de destino**

Como você determina o tipo de uma expressão lambda? Lembrar a expressão lambda que selecionou membros do sexo masculino e entre 18 e 25 anos:

p -> p.getGender() == Person.Sex.MALE

&& p.getAge() >= 18

&& p.getAge() <= 25

Essa expressão lambda foi usada nos dois métodos a seguir:

* public static void printPersons(List<Person> roster, CheckPerson tester) na [Abordagem 3: Especificar código de critérios de pesquisa em uma classe local](https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html#approach3)
* public void printPersonsWithPredicate(List<Person> roster, Predicate<Person> tester) na [Abordagem 6: Usar interfaces funcionais padrão com expressões do Lambda](https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html#approach6)

Quando o Java runtime invoca o método , ele está esperando um tipo de dados de , portanto, o A expressão lambda é desse tipo. Contudo quando o Java runtime invoca o método , ele está esperando um tipo de dados de , Portanto, a expressão lambda é desse tipo. O tipo de dados que Esses métodos esperam é chamado de *tipo de destino*. Para determinar o tipo de um lambda expression, o compilador Java usa o tipo de destino do contexto ou situação em que a expressão lambda foi encontrada. Segue que você só pode usar expressões lambda em situações em que o compilador Java pode determinar um tipo de destino:printPersonsCheckPersonprintPersonsWithPredicatePredicate<Person>

* Declarações de variáveis
* Atribuições
* Instruções de retorno
* Inicializadores de matriz
* Argumentos do método ou do construtor
* Corpos de expressão lambda
* Expressões condicionais, ?:
* Expressões de elenco

**Tipos de destino e argumentos de método**

Para argumentos de método, o compilador Java determina o destino digite com dois outros recursos de linguagem: resolução de sobrecarga e inferência de argumento de tipo.

Considere as duas interfaces funcionais a seguir ( [java.lang.Runnable](https://docs.oracle.com/javase/8/docs/api/java/lang/Runnable.html" \t "_blank) e [java.util.concurrent.Callable<V>](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Callable.html" \t "_blank)):

public interface Runnable {

void run();

}

public interface Callable<V> {

V call();

}

O método não retorna um valor, enquanto faz.Runnable.runCallable<V>.call

Suponha que você tenha sobrecarregado o método da seguinte maneira: (consulte [Definindo métodos](https://docs.oracle.com/javase/tutorial/java/javaOO/methods.html) para obter mais informações sobre métodos de sobrecarga):invoke

void invoke(Runnable r) {

r.run();

}

<T> T invoke(Callable<T> c) {

return c.call();

}

Qual método será invocado na instrução a seguir?

String s = invoke(() -> "done");

O método será invocado porque esse método retorna um valor; o método não. Nesse caso, o tipo da expressão lambda é .invoke(Callable<T>)invoke(Runnable)() -> "done"Callable<T>

**Serialização**

Você pode [serializar](https://docs.oracle.com/javase/tutorial/jndi/objects/serial.html) uma expressão lambda se seu tipo de destino e seus argumentos capturados forem serializáveis. No entanto, como [classes internas](https://docs.oracle.com/javase/tutorial/java/javaOO/nested.html#serialization), a serialização de expressões lambda é fortemente desencorajada.