Introdução ao Java Lambda

Uma expressão lambda é uma função sem nome com parâmetros e um corpo.

O corpo da expressão lambda pode ser uma instrução de bloco ou uma expressão.

-> separa os parâmetros e o corpo.

(int x) -> x + 1 usa um parâmetro int e retorna o valor do parâmetro incrementado em 1.

(int x, int y) -> x + y Utiliza dois parâmetros int e devolve a soma.

(String msg)->{System.out.println(msg);} pega um parâmetro String e o imprime na saída padrão.

msg->System.out.println(msg) toma um parâmetro e imprime-o na saída padrão. É idêntico ao código acima.

() -> "hi" não usa parâmetros e retorna uma cadeia de caracteres.

(String str) -> str.length() leva uma String e retorna seu comprimento.

O lambda a seguir usa dois parâmetros int e retorna o máximo dos dois.

(int x, int y) -> {

int max = x > y ? x : y;

return max;

}

Por que as expressões Lambda

As expressões lambda nos permitem passar a lógica de forma compacta.

O código a seguir usa uma classe interna anônima para adicionar manipulador de eventos para uma ação de clique no botão.

button.addActionListener(**new** ActionListener() {

**public** **void** actionPerformed(ActionEvent e) {

System.out.println(**"hi"**);

}

});

O manipulador de ações imprime uma mensagem quando o botão é clicado.

Usando uma expressão lambda, podemos adicionar o manipulador de ação a um evento de clique no botão em uma única linha de código.

button.addActionListener(e -> System.out.println("hi"));

Nota

Em vez de passar uma classe interna que implementa uma interface, estamos passando em um bloco de código.

e é o nome de um parâmetro,

-> separa o parâmetro do corpo de a expressão lambda.

Na expressão lambda, o parâmetro não é declarado com um tipo. é inferir o tipo de de seu contexto, a assinatura de . ejavaceaddActionListener

Não precisamos escrever explicitamente o digite quando for óbvio. Os parâmetros do método lambda ainda são tipados estaticamente.

Sintaxe Java Lambda

A sintaxe geral para usar expressões lambda é

(Parameters) -> { Body }

-> separa parâmetros e corpo de expressão lambda.

Os parâmetros são colocados entre parênteses, o que é da mesma forma que para os métodos enquanto o corpo da expressão lambda é um bloco de código entre chaves.

Nota

O corpo da expressão lambda pode ter variáveis locais, instruções. Podemos usar quebrar, continuar e retornar no corpo de expressão lambda. Podemos até jogar exceções fora do corpo de expressão lambda.

Uma expressão lambda não tem um nome, pois representa uma classe interna anônima.

O tipo de retorno de uma expressão lambda é inferido pelo compilador.

Uma expressão lambda não pode ter uma cláusula throws como um método.

Uma expressão lambda não pode ser genérica enquanto o genérico é definido no funcional interface.

Expressão lambda explícita e implícita

Uma expressão lambda que não declara o tipo de seus parâmetros é chamada de expressão lambda implícita.

Uma expressão lambda explícita é uma expressão lambda que declara o tipo de seus parâmetros.

O compilador irá inferir os tipos de parâmetros para expressão lambda implícita.

Exemplo

O código a seguir Cria interface com método único e a usa como tipo de expressão lambda. Ao criar a expressão lambda, declaramos o tipo do parâmetro como tem o tipo. s1 Integer

**public** **class** Main {

/\*w w w. ja va 2 s . com\*/

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

MyIntegerCalculator myIntegerCalculator = (**Integer** s1) -> s1 \* 2;

System.out.println(**"1- Result x2 : "** + myIntegerCalculator.calcIt(5));

}

}

**interface** MyIntegerCalculator {

**public** **Integer** calcIt(**Integer** s1);

}

O código acima gera o seguinte resultado.



Exemplo 2

Aqui está a demonstração novamente sem usar o tipo. Ao ignorar o digite o compilador tem que descobrir isso.

**public** **class** Main {

// ww w. ja v a 2 s .c o m

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

MyIntegerCalculator myIntegerCalculator = (s1) -> s1 \* 2;

System.out.println(**"1- Result x2 : "** + myIntegerCalculator.calcIt(5));

}

}

**interface** MyIntegerCalculator {

**public** **Integer** calcIt(**Integer** s1);

}

O código acima gera o seguinte resultado.



Omitindo tipos de parâmetro

Podemos optar por omitir tipos de parâmetros em expressões lambda.

Na expressão lambda, os tipos de parâmetros são declarados. (int x, int y) -> { return x + y; }

Podemos reescrever com segurança a expressão lambda omitindo os tipos de parâmetros como

(x, y) -> { return x + y; }

Se optarmos por omitir os tipos de parâmetros, teremos que omitir tipos para todos os parâmetros.

**public** **class** Main {

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

Processor stringProcessor = (str) -> str.length();

String name = **"Java Lambda"**;

**int** length = stringProcessor.getStringLength(name);

System.out.println(length);//from ww w.ja v a 2 s . c om

}

}

@FunctionalInterface

**interface** Processor {

**int** getStringLength(String str);

}

O código acima gera o seguinte resultado.



Parâmetro único

Para uma expressão lambda de parâmetro único, podemos omitir os parênteses à medida que omitimos o tipo de parâmetro.

A expressão lambda tem tudo. (String msg) -> {System.out.println(msg);}

Então podemos omitir o tipo de parâmetro a ter

(msg)->{System.out.println(msg);}

Podemos ainda omitir o tipo de parâmetro e os parênteses da seguinte maneira.

msg -> { System.out.println(msg); }

**public** **class** Main {

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

Processor stringProcessor = str -> str.length();

String name = **"Java Lambda"**;

**int** length = stringProcessor.getStringLength(name);

System.out.println(length);/\* w w w. jav a2s .c o m\*/

}

}

@FunctionalInterface

**interface** Processor {

**int** getStringLength(String str);

}

O código acima gera o seguinte resultado.



Sem parâmetros

Para uma expressão lambda sem parâmetros, ainda precisamos dos parênteses.

() -> { System.out.println("hi"); }

O exemplo a seguir mostra como usar o .BooleanSupplier

**import** java.util.function.BooleanSupplier;

//from w w w.j a va 2s . c o m

**public** **class** Main {

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

BooleanSupplier bs = () -> true;

System.out.println(bs.getAsBoolean());

**int** x = 0, y= 1;

bs = () -> x > y;

System.out.println(bs.getAsBoolean());

}

}

O código acima gera o seguinte resultado.

Uma imagem contendo Forma

Descrição gerada automaticamente

Modificador final

Você pode usar modificador na declaração de parâmetro para expressões lambda explícitas. final

As expressões lambda a seguir usam o modificador final.

(final int x, final int y) -> { return x + y; }

Podemos usar apenas um modificador da seguinte maneira.

(int x, final int y) -> {return x + y;}

**public** **class** Main {

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

Processor stringProcessor = (**final** String str) -> str.length();

String name = **"Java Lambda"**;

**int** length = stringProcessor.getStringLength(name);

System.out.println(length);/\* w w w . ja v a 2s .co m\*/

}

}

@FunctionalInterface

**interface** Processor {

**int** getStringLength(String str);

}

O código acima gera o seguinte resultado.



Corpo de expressões lambda

O corpo da expressão lambda pode ser uma instrução block ou uma única expressão.

Uma instrução block é colocada entre chaves, enquanto uma única expressão pode existir sem chaves.

Na instrução block podemos usar a instrução para retornar o valor. return

A expressão lambda a seguir usa uma instrução block e uma instrução use para retornar a soma. return

(int x, int y) -> { return x + y; }

Enquanto o lambda a seguir usa uma expressão:

(int x, int y) -> x + y

A expressão não requer os aparelhos.

O lambda não é necessário para retornar um valor. As duas expressões lambda a seguir apenas produzem o parâmetro para saída padrão e não retornar nada.

(String msg)->{System.out.println(msg);}// a block statement

(String msg)->System.out.println(msg) //an expression

Exemplo:

**public** **class** Main {

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

Processor stringProcessor = (String str) -> str.length();

String name = **"Java Lambda"**;

**int** length = stringProcessor.getStringLength(name);

System.out.println(length);// www . j a va 2 s. co m

}

}

@FunctionalInterface

**interface** Processor {

**int** getStringLength(String str);

}

O código acima gera o seguinte resultado.



Inferência de tipo Java Lambda

Uma expressão lambda representa uma instância de uma interface funcional.

Uma expressão lambda pode ser mapeada para diferentes tipos de interface funcional dependendo do contexto.

O compilador infere o tipo de uma expressão lambda.

Exemplo

No código a seguir, há duas interfaces funcionais e .Processor SecondProcessor

Processor tem um método chamado que aceita uma cadeia de caracteres como parâmetro e retorna e .getStringLength int

SecondProcessor tem um método chamado que também aceita uma cadeia de caracteres como parâmetro e retorna um arquivo .noNameint

Pelo código, podemos ver que podemos atribuir duas expressões lambda idênticas para eles.

**public** **class** Main {

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

Processor stringProcessor = (String str) -> str.length();

SecondProcessor secondProcessor = (String str) -> str.length();

//from w w w. ja va 2s . c o m

**//stringProcessor = secondProcessor; //compile error**

String name = **"Java Lambda"**;

**int** length = stringProcessor.getStringLength(name);

System.out.println(length);

}

}

@FunctionalInterface

**interface** Processor {

**int** getStringLength(String str);

}

@FunctionalInterface

**interface** SecondProcessor {

**int** noName(String str);

}

O código acima gera o seguinte resultado.



Nota

Processor ou é chamado de tipo de destino. SecondProcessor

O processo de inferir o tipo de uma expressão lambda é chamado de digitação de destino.

O compilador usa as seguintes regras para determinar se um A expressão lambda pode ser atribuída ao seu tipo de destino:

* Deve ser uma interface funcional.
* Os parâmetros de expressão lambda devem corresponder ao método abstrato na interface funcional.
* O tipo de retorno da expressão lambda é compatível com o tipo de retorno de o método abstrato em interface funcional.
* As exceções verificadas lançadas da expressão lambda devem ser compatível com a cláusula de arremessos declarados do método abstrato em interface funcional.

Parâmetro de comportamento do Java Lambda

Podemos passar expressões lambda para métodos como argumentos.

Exemplo

O código a seguir cria uma interface funcional chamada .Calculator

Dentro do existe um método chamado qual aceita dois parâmetros e retorna um valor.Calculatorcalculateintint

Na classe há um método que aceita o interface funcional como parâmetro. E chama o método do e produz o resultado.MainengineCalculatorcalculateCalculator

No método principal, chamamos os métodos do motor quatro vezes com diferentes expressões lambda.

**public** **class** Main {

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

engine((x,y)-> x + y);// w w w .j av a 2s. c om

engine((x,y)-> x \* y);

engine((x,y)-> x / y);

engine((x,y)-> x % y);

}

**private** **static** **void** engine(Calculator calculator){

**int** x = 2, y = 4;

**int** result = calculator.calculate(x,y);

System.out.println(result);

}

}

@FunctionalInterface

**interface** Calculator{

**int** calculate(**int** x, **int** y);

}

O código acima gera o seguinte resultado.

Texto

Descrição gerada automaticamente com confiança baixa

Nota

O resultado do método depende de Expressões lambda passaram para ele. engine

O comportamento do motor é parametrizado.

Alterar o comportamento de um método através de seus parâmetros é chamado de parametrização de comportamento.

Na parametrização do comportamento passamos a lógica encapsulado em expressões lambda para métodos como se fosse dados.

Parametrização do comportamento Ambiguidade

Nem sempre é possível para o compilador inferir o tipo de uma expressão lambda.

Uma dessas situações é passar expressões lambda para métodos sobrecarregados.

Há duas interfaces funcionais no código a seguir. Uma delas é pelo valor cálculo e o outro é por valor. intlong

Na classe Main existem métodos sobrecarregados chamados . Um é esperado e outro é para .engineIntCalculatorLongCalculator

No método principal temos que indicar os parâmetros de expressão lambda para indicar compilador qual função sobrecarregada queremos usar.

**public** **class** Main {

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

engine((**int** x,**int** y)-> x + y);

engine((**long** x, **long** y)-> x \* y);

engine((**int** x,**int** y)-> x / y);

engine((**long** x,**long** y)-> x % y);

}/\* w ww . java 2 s . com\*/

**private** **static** **void** engine(IntCalculator calculator){

**int** x = 2, y = 4;

**int** result = calculator.calculate(x,y);

System.out.println(result);

}

**private** **static** **void** engine(LongCalculator calculator){

**long** x = 2, y = 4;

**long** result = calculator.calculate(x,y);

System.out.println(result);

}

}

@FunctionalInterface

**interface** IntCalculator{

**int** calculate(**int** x, **int** y);

}

@FunctionalInterface

**interface** LongCalculator{

**long** calculate(**long** x, **long** y);

}

O código acima gera o seguinte resultado.

Imagem em branco e preto

Descrição gerada automaticamente com confiança média

Nota 1

Para resolver a ambiguidade, podemos alterar a expressão lambda implícita para explicitado especificando o tipo dos parâmetros. Isso é o que foi feito para o código acima.

Ou podemos usar um elenco da seguinte forma. Ao chamar o motor pela primeira vez, lançamos o lambda expressão para .IntCalculator

**public** **class** Main {

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

engine((IntCalculator) ((x,y)-> x + y));

engine((**long** x, **long** y)-> x \* y);

engine((**int** x,**int** y)-> x / y);

engine((**long** x,**long** y)-> x % y);

}/\* w w w . java 2 s .c o m\*/

**private** **static** **void** engine(IntCalculator calculator){

**int** x = 2, y = 4;

**int** result = calculator.calculate(x,y);

System.out.println(result);

}

**private** **static** **void** engine(LongCalculator calculator){

**long** x = 2, y = 4;

**long** result = calculator.calculate(x,y);

System.out.println(result);

}

}

@FunctionalInterface

**interface** IntCalculator{

**int** calculate(**int** x, **int** y);

}

@FunctionalInterface

**interface** LongCalculator{

**long** calculate(**long** x, **long** y);

}

O código acima gera o seguinte resultado.

Texto

Descrição gerada automaticamente com confiança média

Nota 2

Ou podemos evitar o uso da expressão lambda diretamente como parâmetro. Podemos atribuir a expressão lambda a uma interface funcional, e em seguida, passe a variável para o método. O código a seguir mostra essa técnica.

**public** **class** Main {

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

IntCalculator iCal = (x,y)-> x + y;

engine(iCal);/\*from www .java2s . c om\*/

engine((**long** x, **long** y)-> x \* y);

engine((**int** x,**int** y)-> x / y);

engine((**long** x,**long** y)-> x % y);

}

**private** **static** **void** engine(IntCalculator calculator){

**int** x = 2, y = 4;

**int** result = calculator.calculate(x,y);

System.out.println(result);

}

**private** **static** **void** engine(LongCalculator calculator){

**long** x = 2, y = 4;

**long** result = calculator.calculate(x,y);

System.out.println(result);

}

}

@FunctionalInterface

**interface** IntCalculator{

**int** calculate(**int** x, **int** y);

}

@FunctionalInterface

**interface** LongCalculator{

**long** calculate(**long** x, **long** y);

}

O código acima gera o seguinte resultado.

Uma imagem contendo Padrão do plano de fundo

Descrição gerada automaticamente