

Java

Expressões Lambda (Java 8)

UA, DETI, Programação III
José Luis Oliveira, Carlos Costa
2017/18

1

Cálculo lambda

- As linguagens de programação funcional são baseadas no cálculo lambda (cálculo- λ).
 - Lisp, Haskell, Scheme
- O cálculo lambda pode ser visto como uma linguagem de programação abstrata em que funções podem ser combinadas para formar outras funções.
- Ideia geral: formalismo matemático
 - $x \rightarrow f(x)$ i.e. x é transformado em $f(x)$
- O cálculo lambda trata as funções como *elementos de primeira classe*
 - podem ser utilizadas como argumentos e retornadas como valores de outras funções.

Sintaxe

- Uma expressão lambda descreve uma função anónima
- Representa-se na forma:
 - `(argument) -> (body)`
`(int a, int b) -> { return a + b; }`
- Pode ter zero, um, ou mais argumentos
 - `() -> { body }`
`() -> System.out.println("Hello World");`
 - `(arg1, arg2...) -> { body }`
- O tipo dos argumentos pode ser explicitamente declarado ou inferido
 - `(type1 arg1, type2 arg2...) -> { body }`
`(int a, int b) -> { return a + b; }`
`a -> return a*a // um argumento - podemos omitir (a)`
- O corpo (body) pode ter uma ou mais instruções

3

Exemplos

lambda expression	equivalent method
<code>() -> { System.gc(); }</code>	<code>void nn() { System.gc(); }</code>
<code>(int x) -> { return x+1; }</code>	<code>int nn(int x) return x+1; }</code>
<code>(int x, int y) -> { return x+y; }</code>	<code>int nn(int x, int y) { return x+y; }</code>
<code>(String... args) ->{return args.length;}</code>	<code>int nn(String... args) { return args.length; }</code>
<code>(String[] args) -> { if (args != null) return args.length; else return 0; }</code>	<code>int nn(String[] args) { if (args != null) return args.length; else return 0; }</code>

4

Como usar?

- Uma expressão lambda não pode ser isoladamente

```
(n) -> (n % 2) == 0 // Erro de compilação
```

- Precisamos de outro mecanismo adicional
 - Interfaces funcionais
 - onde as expressões lambda passam a ser implementações de métodos abstratos.
 - O compilador Java converte uma expressão lambda num método privado da classe (isto é um processo interno).

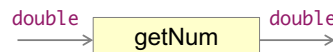
5

Functional interfaces

- Uma interface funcional contém apenas um método/função abstrata
 - Método abstrato numa interface? Não são todos?
 - A partir do JDK 8 passa a ser possível definir um comportamento por omissão nos métodos de uma interface (*default method*)
- Exemplo

```
@FunctionalInterface
interface MyNum {
    double getNum(double n);
}

public class Lambda1 {
    public static void main(String[] args) {
        MyNum n1 = (x) -> x+1;
        // qualquer expressão que transforme double em double
        System.out.println(n1.getNum(10));
        n1 = (x) -> x*x;
        System.out.println(n1.getNum(10));
    }
}
```



```
11.0
100.0
```

6

Exemplos

```
@FunctionalInterface
interface Ecra {
    void escreve(String s);
}

public class Lambda2 {

    public static void main(String[] args) {

        Ecra xd = (String s) -> {
            if (s.length() > 2)
                System.out.print(s);
            else
                System.out.print("-");
        };
        xd.escreve("Lambda print");
        xd.escreve("?");
    }
}
```

interface funcional

Lambda print-

7

Exemplos

```
// Another functional interface.
interface NumericTest {
    boolean test(int n);
}

class Lambda3 {
    public static void main(String args[]) {
        // A lambda expression that tests if a number is even.
        NumericTest isEven = (n) -> (n % 2) == 0;
        if (isEven.test(10)) System.out.println("10 is even");
        if (!isEven.test(9)) System.out.println("9 is not even");
        // Now, use a lambda expression that tests if a number is non-
        negative.
        NumericTest isNonNeg = (n) -> n >= 0;
        if (isNonNeg.test(1)) System.out.println("1 is non-negative");
        if (!isNonNeg.test(-1)) System.out.println("-1 is negative");
    }
}
```

10 is even
9 is not even
1 is non-negative
-1 is negative

8

Exemplos

```
// Demonstrate a lambda expression that takes two parameters.
interface NumericTest2 {
    boolean test(int n, int d);
}

public class Lambda4 {
    public static void main(String args[]) {
        // This lambda expression determines if one number is
        // a factor of another.
        NumericTest2 isFactor = (n, d) -> (n % d) == 0;
        if (isFactor.test(10, 2))
            System.out.println("2 is a factor of 10");
        if (!isFactor.test(10, 3))
            System.out.println("3 is not a factor of 10");
    }
}
```

```
2 is a factor of 10
3 is not a factor of 10
```

9

Exemplos

```
// A block lambda that computes the factorial of an int value.
interface NumericFunc {
    int func(int n);
}

class Lambda5 {
    public static void main(String args[]) {
        // This block lambda computes the factorial of an int value.
        NumericFunc factorial = (n) -> {
            int result = 1;
            for (int i = 1; i <= n; i++)
                result = i * result;
            return result;
        };
        System.out.println("The factorial of 3 is " + factorial.func(3));
        System.out.println("The factorial of 5 is " + factorial.func(5));
    }
}
```

```
The factorial of 3 is 6
The factorial of 5 is 120
```

10

Exemplos

```
// A block lambda that reverses the characters in a string.
interface StringFunc {
    String func(String n);
}
class Lambda6 {
    public static void main(String args[]) {
        // This block lambda reverses the characters in a string.
        StringFunc reverse = (str) -> {
            String result = "";
            int i;
            for (i = str.length() - 1; i >= 0; i--)
                result += str.charAt(i);
            return result;
        };
        System.out.println("Lambda reversed is " + reverse.func("Lambda"));
        System.out.println("Expression reversed is "
            + reverse.func("Expression"));
    }
}
```

Lambda reversed is adbmaL

Expression reversed is noisserpxE

11

Interfaces funcionais genéricas

- Dos 2 exemplos anteriores

```
// A block lambda that computes the factorial of an int value.
interface NumericFunc { int func(int n); }
```

```
// A block lambda that reverses the characters in a string.
interface StringFunc { String func(String n); }
```

- Repetição! Solução?

```
// A generic functional interface.
```

```
interface SomeFunc<T> {
    T func(T n);
}
```

interface funcional genérica

- Utilização

```
SomeFunc<String> reverse = ...
```

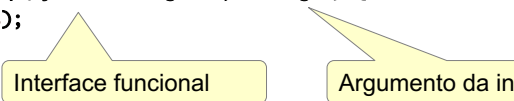
```
SomeFunc<Integer> factorial = ...
```

12

Expressões Lambda como argumento

- Exemplo

```
interface MyFunc<T> {  
    T func(T n);  
}  
...  
// Funções que aceita uma expressão lambda e o seu argumento (T n)  
static String stringOp(MyFunc<String> sf, String s) {  
    return sf.func(s);  
}  
...  
// Outro exemplo  
static Person PersonOp(MyFunc<Person> sf, Person s) {  
    return sf.func(s);  
}
```



13

Expressões Lambda como argumento

- Utilização

```
String inStr = "Lambdas add power to Java";  
String outStr = stringOp((str) -> str.toUpperCase(), inStr);  
System.out.println("The string in uppercase: " + outStr);  
// This passes a block lambda that removes spaces.  
outStr = stringOp((str) -> {  
    String result = "";  
    int i;  
    for(i = 0; i < str.length(); i++)  
        if(str.charAt(i) != ' ')  
            result += str.charAt(i);  
    return result;  
}, inStr);  
System.out.println("The string with spaces removed: " + outStr);
```

```
Here is input string: Lambdas add power to Java  
The string in uppercase: LAMBDA S ADD POWER TO JAVA  
The string with spaces removed: LambdasaddpowertoJava
```

14

Package java.util.function

- Várias interfaces funcionais são fornecidas pelo Java SE 8
 - Servem como ponto de partida para situações standard
- **Predicate**: A property of the object passed as argument
- **Consumer**: An action to be performed with the object passed as argument
- **Function**: Transform a T to a U
- **Supplier**: Provide an instance of a T (such as a factory)
- **UnaryOperator**: A unary operator from T -> T
- **BinaryOperator**: A binary operator from (T, T) -> T

15

java.util.function.Predicate<T>

```
@FunctionalInterface
public interface Predicate<T> {
    boolean test(T t);
    default Predicate<T> and(Predicate<? super T> other) {
        Objects.requireNonNull(other);
        return (t) -> test(t) && other.test(t);
    }
    default Predicate<T> negate() {
        return (t) -> !test(t);
    }
    default Predicate<T> or(Predicate<? super T> other) {
        Objects.requireNonNull(other);
        return (t) -> test(t) || other.test(t);
    }
    static <T> Predicate<T> isEqual(Object targetRef) {
        return (null == targetRef)
            ? Objects::isNull
            : object -> targetRef.equals(object);
    }
}
```

5 métodos? Default?? Static ???

16

Métodos estáticos em Interfaces

```
interface X {  
    static void foo() {  
        System.out.println("foo");  
    }  
}  
  
class Y implements X {  
}  
  
public class Z {  
    public static void main(String[] args) {  
        X.foo();  
        // Y.foo(); // won't compile  
    }  
}
```

17

java.util.function.Predicate

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

Funções genérica para
filtragem da lista

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

Exemplo de filtro

18

Versão genérica

```
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);  
        }  
    }  
}
```

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

Mais sobre isto
quando falarmos
em *Collections*

19

Utilização de expressões lambda

- Interfaces funcionais devem ter um único método abstrato (podem ter outros métodos *default*)
 - Classes anónimas com um único método
 - ActionListener, Runnable, ...
 - Maior versatilidade - podemos criar funções sem "poluir" a interface pública da classe.
- Java Collections

```
//Old way:  
List<Integer> list = Arrays.asList(1, 2, 3, 4, 5, 6, 7);  
for(Integer n: list) {  
    System.out.println(n);  
}  
//New way:  
List<Integer> list = Arrays.asList(1, 2, 3, 4, 5, 6, 7);  
list.forEach(n -> System.out.println(n));  
//or we can use :: double colon operator in Java 8 (method reference)  
list.forEach(System.out::println);
```

20