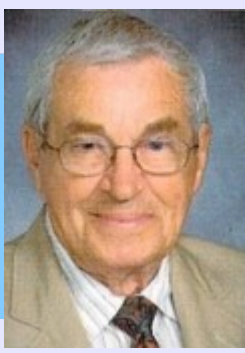




75-62 Técnicas de Programación
Concurrentes II
Lic. Ing. Osvaldo Clúa
2014

Facultad de Ingeniería
Universidad de Buenos Aires

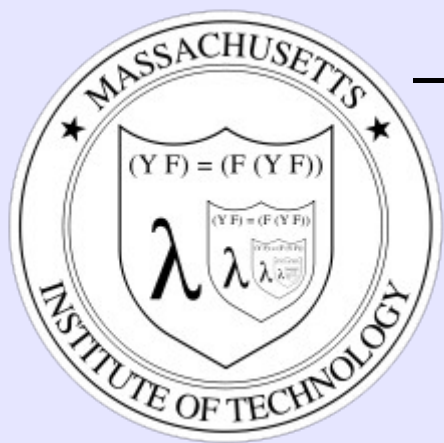
Introducción a
Lambda en java



Cálculo Lambda



- Sistema lógico formal para expresar cálculos basados en abstracción y aplicación de funciones, usando sustitución y enlace (binding) de variables.
 - Introducido por Alonzo Church (1903-1995)
 - Implementado entre otros en Lisp y Scheme



Expresiones Lambda

- En un lenguaje funcional (Mit Scheme) una expresión lambda se puede escribir
 - $(\text{lambda } (x) (* x x))$
 - Que significa que una variable libre x se multiplica por si misma $(*)$ y se produce el resultado de esta operación.
- Y se aplica luego a un valor
 - $((\text{lambda } (x) (* x x)) 3)$
 - Produciendo un 9

Lambda en Java

- En Java se escribe
 - `(Integer x)->{return x*x;}`
 - `(Integer x)->x*x;`
- O si el tipo puede inferirse
 - `(x)->x*x`
- Y se lo usa donde que java espera una función.
 - Pero en Java no hay objetos-función...
 - ...y la expresión lambda ni siquiera es un objeto.

Interfaces funcionales

- Son interfaces con un solo método abstracto.


```
@FunctionalInterface
interface Calc {
    Integer op(Integer a, Integer b);
}
```

- La annotation es optativa
 - Pueden tener mas métodos concretos (default).
- Y ahora se puede asignar la expresion lambda

```
Calc sum =(Integer x, Integer y)->{return x+y;};
Calc mult = (Integer x, Integer y) -> x * y;
Calc rest =(x,y)->x-y;
```

Ejemplo Completo

```
public class Lambda01 {  
    /*    *cálculo lambda    */  
  
    @FunctionalInterface  
    interface Calc {  
        Integer op(Integer a, Integer b);  
    }  
  
    public static void main(String[] args) {  
        System.out.print(" suma lambda on site 4+2=");  
        Calc sum = (Integer x, Integer y) -> {  
            return x + y;  
        };  
        Calc mult = (Integer x, Integer y) -> x * y;  
        Calc rest = (x, y) -> x - y;  
        System.out.println("sum " + sum.op(6, 4));  
        System.out.println("rest " + rest.op(6, 4));  
        System.out.println("mult " + mult.op(6, 4));  
    }  
}
```



sum 10
rest 2
mult 24

java.util.function

- Son "functional interfaces"
- Sirven como "variables" para expresiones lambda.
- T es el tipo del parámetro, R del resultado
 - Function ($T \rightarrow R$)
 - Consumers ($T \rightarrow \text{void}$)
 - Predicate ($T \rightarrow \text{boolean}$)
 - Supplier ($\text{nil} \rightarrow R$)

Tipos de Functional Interfaces

Functional Interface	Parameter Types	Return Type	Description
Supplier<T>	None	T	Supplies a value of type T
Consumer<T>	T	void	Consumes a value of type T
BiConsumer<T, U>	T, U	void	Consumes values of types T and U
Predicate<T>	T	boolean	A Boolean-valued function
ToIntFunction<T> ToLongFunction<T> ToDoubleFunction<T>	T	int long double	An int-, long-, or double-valued function
IntFunction<R> LongFunction<R> DoubleFunction<R>	int long double	R	A function with argument of type int, long, or double
Function<T, R>	T	R	A function with argument of type T
BiFunction<T, U, R>	T, U	R	A function with arguments of types T and U
UnaryOperator<T>	T	T	A unary operator on the type T
BinaryOperator<T>	T, T	T	A binary operator on the type T

Aggregate Operations


- Son operaciones sobre un conjunto de datos (una collection por ejemplo).
 - Como resultado puede dar un valor u otro conjunto de datos
- En java se pueden componer.
 - Como los pipes del shell
- Están en el paquete `Java.util.stream`

Java.util.stream

- El método `stream()` de `Collection` crea un stream a partir de una `Collection`.
 - Hay otras formas de crearlo, por ejemplo `generate(Supplier s)`
- Hay métodos intermedios que crean un stream a partir de otro.
- Hay métodos que recorren el stream operando sobre cada elemento.
- Hay métodos finales que reducen el stream a un valor
- `Object [] toArray()` devuelve el stream en un array.

Ejemplo

```
List <Integer> li= Arrays.asList(5, 7, 10, 25, 74);  
int suma=li.stream().reduce(0,Integer::sum);  
System.out.println("Suma de la lista "+suma);  
  
long pares=li.stream().filter(x->x % 2 ==0).count();  
System.out.println("Cantidad de pares "+pares);  
  
Integer[] arrInt=Stream.of(23,45,67,88,2,27)  
    .filter(x->x % 2 ==0)  
    .toArray(Integer[]::new);  
Arrays.stream(arrInt).forEach(System.out::println);
```



```
Suma de la lista 121  
Cantidad de pares 2  
88  
2
```

Características

- Un Stream no almacena sus elementos.
- Un Stream no cambia sus elementos, crea un nuevo Stream a partir de ellos.
- Los Streams son lazy. Solo actúan cuando se los pide desde la salida.

```
static void imp(int n) {  
    System.out.print(" " + n);  
}
```

```
0 10 20 30 40 50 60 70 80 90 100 110 120 130 140  
0 10 20 30 40
```

```
Stream.iterate(0, n -> n + 10).limit(15).forEach(Stream_examples::imp);  
System.out.println("");
```

```
Stream<Integer> numbers = Stream.iterate(0, n -> n + 10);  
numbers.limit(5).forEach(Stream_examples::imp);  
System.out.println("");
```

Optional

- Es un container que puede o no tener un valor.
- Ver en el ejemplo que los Stream no pueden reusarse

```
List <String> qacL=Arrays.asList("Arriba", "Quilmes");  
Stream <String> qacSt=qacL.stream();  
List <String> cerveL=Arrays.asList("Arriba", "Cerveceros");
```

```
Optional <String> conQ=qacSt.filter(s->s.startsWith("Q")).findFirst();  
System.out.println(qacL.stream().reduce(" ",String::concat)+" tiene empezando con Q "+conQ);
```

```
conQ=cerveL.stream().filter(s->s.startsWith("Q")).findFirst();  
System.out.println(cerveL.stream().reduce(" ",String::concat)+" tiene empezando con Q "+conQ);
```



ArribaQuilmes tiene empezando con Q Optional[Quilmes]
ArribaCerveceros tiene empezando con Q Optional.empty