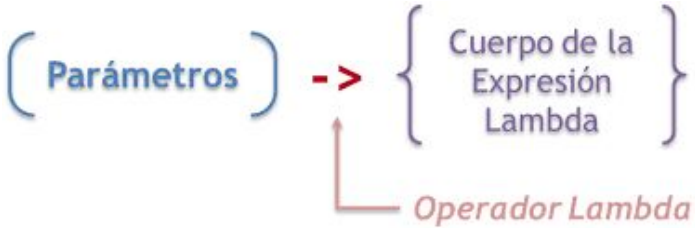


Expresiones lambda

UNJu - Programación Orientada a Objetos

Ing. José Zapana

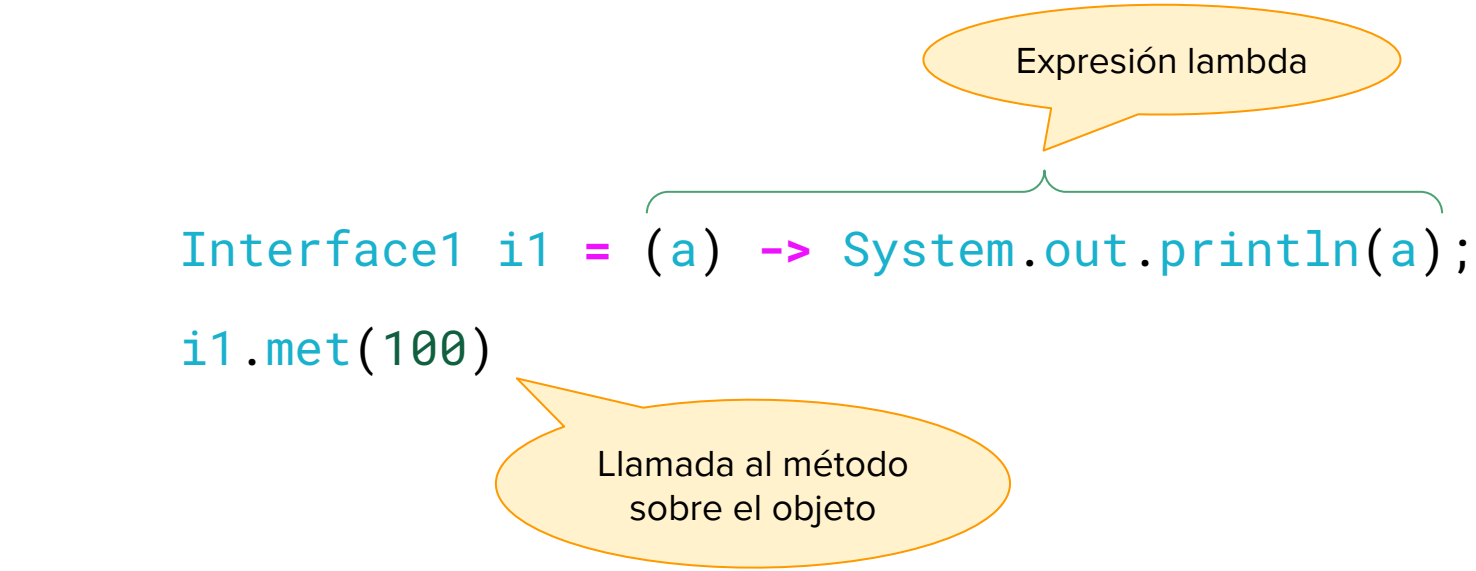
Interfaz funcional

- Es necesario definirlas dado que las expresiones lambda están estrechamente relacionadas.
- Proporcionan un único método abstracto

```
public interface runnable {
    void run();
}
public interface Interface1{
    void met(int data);
    default int res(){return 1;}
}
public interface Interface2{
    boolean process(int n, String pt);
    static void print(){}
}
```

¿Qué es una expresión lambda?

- Es una implementación de una interfaz funcional
- Proporciona código al único método abstracto de la interfaz, a la vez que genera un objeto que implementa la misma.



Sintaxis para la creación de una expresión lambda

- Tiene 2 partes: la lista de parámetros del método y la implementación

```
parámetros -> implementación
```

- Los parámetros pueden indicar o no el tipo
- La lista de parámetros se puede indicar o no entre paréntesis.
- Sin embargo si hay 2 o más parámetros los paréntesis son obligatorios al igual que el tipo de datos
- En caso de devolver un resultado, la implementación puede omitir la palabra ***return*** si consta de una sola expresión.

Ejemplos

```
() -> 3
(int a) -> System.out.println(“hello”)
x -> x * x
(n1, n2) -> {
   n1 += 20;
   System.out.println(n1 + n2);
}
```

```
-> 3
int a ->
System.out.println(“hello”)
x -> return x*x; //se
requieren llaves con return
n1, n2 ->
System.out.println(n1 + n2)
```

Inferencia de tipos

- Es posible inferir el tipo en los parámetros de las expresiones lambda:

```
(var a) -> System.out.println(a)
```

- Aunque no se puede combinar inferencia de tipos y tipos específicos en una misma expresión

```
(var a, int c) -> a + c  // error de compilación
```

- ¿Qué utilidad tiene si ya es posible no identificar el tipo en los parámetros?

```
(@NotNull var c) -> … //OK
```

```
(@NotNull c) -> … // Error de compilación
```

Interfaces java.util.function

- Conjunto de interfaces funcionales incorporadas en Java SE 8 dentro del paquete java.util.function
- Utilizadas como argumentos en métodos que manipulan datos para el establecimiento de criterios de filtrado, operación sobre elementos, transformación, etc.
- Implementadas habitualmente mediante expresiones lambda

Interfaz predicate <T>

- Dispone del método abstracto test, que a partir de un objeto realiza una comprobación y devuelve un boolean:

```
boolean test (T t)
```

- Utilizada para la definición de criterios de filtrado, por ejemplo método `removeIf()` de Collection:

```
//Elimina los elementos que cumplen
//con la condición del filtro
boolean removeIf(Predicate <? super T> filtro)
```

```
//Eliminar números pares
//de una collection
lista.removeIf(n -> n % 2 == 0)
```

- Variante de BiPredicate <T,U> con dos parámetros

```
boolean test (T t, U u)
```

Interfaz function <T, R>

- Método abstracto apply, que a partir de un objeto realiza una operación y devuelve un resultado:

```
R apply (T t)
```

- Utilizando operaciones de transformación de datos. Por ejemplo el método map() de Stream

```
//Transforma cada elemento del Stream de tipo T
//en otro de tipo R
Stream <R> map(Function <? super T,? extends R> mapper)
```

```
//Genera un nuevo Stream con la longitud de las cadenas
//del Stream original
st.map( cad -> cad.length());
```

Interfaz consumer <T>

- Dispone del método abstracto accept, que realiza algún tipo de procesamiento con el objeto recibido:

```
void accept (T t)
```

- Utilizada en operaciones de procesamiento de datos. Por ejemplo, método forEach() de listas y conjuntos

```
//Aplica las operaciones del método a cada elemento
//de la lista
void forEach(Consumer <? super T> action)
```

```
//Imprime el contenido de la lista
lista.forEach(n -> System.out.println(n));
```

Interfaz Supplier <T>

- Dispone de un método abstracto get, que no recibe ningún parámetro y devuelve como resultado un objeto:

```
T get()
```

- Utilizada para implementar operaciones de extracción de datos. Por ejemplo generate() de Stream:

```
//Generar una secuencia infinita de elementos
//proporcionados por llamadas a get()
static Stream <T> generate(Supplier <T> s)
```

```
//Devuelve un Stream de números aleatorios entre 1 y 500
Stream.generate(() -> (int)(Math.random()*500+1));
```

Ejemplo

- Dada la siguiente lista, escribir un bloque de código que muestre solamente los números pares ordenados de mayor a menor:

```
List<Integer> nums = List.of(10, 4, 21, 3, 17, 8, 20, 11);
```

- Respuesta:

```
List<Integer> nums = List.of(10, 4, 21, 3, 17, 8, 20, 11);
List<Integer> datos = new ArrayList<>(nums);
datos.removeIf(n -> n % 2! = 0);
datos.sort((a,b) -> b - a);
datos.forEach(n -> System.out.println(n));
```

Referencias

- https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html