



# Tema 2.9

# Expresiones

# Lambda

Análisis y Diseño de Software

2º Ingeniería Informática

Universidad Autónoma de Madrid



# Indice

- Introducción y ejemplos
- Expresiones lambda
- Streams
- Ejercicios
- Conclusiones y bibliografía



# Indice

- Introducción y ejemplos
- Expresiones lambda
- ~~Streams~~ (mirad las transparencias completas)
- Ejercicios
- Conclusiones y bibliografía

# Expresiones lambda. ¿Qué son?

- Funciones como conceptos de primer nivel.
  - Anónimas, no llevan nombre.
  - Podemos pasarlas como parámetros.
- En Java 8 se llaman expresiones lambda.
- El nombre proviene del  $\lambda$ -cálculo (Alonzo Church).
- En otros lenguajes (e.j, Ruby) las *closures* son un concepto similar.
- Promueven un estilo de programación más cercano al paradigma funcional.
  - Concatenación de funciones, que operan sobre streams.
  - Más fácilmente paralelizable (útil para procesar grandes volúmenes de datos).
  - Código más intencional y menos verboso.

# Expresiones lambda. Ejemplo.

- En Swing, es frecuente tener que configurar los componentes gráficos con métodos callback, que se ejecutan cuando sucede un evento.
- Antes de Java 8, había que definir una clase para poder definir el método.

Método de interés  
para el botón

```
button.addActionListener(new ActionListener() {  
    public void actionPerformed(ActionEvent event) {  
        System.out.println("button clicked");  
    }  
});
```

Clase anónima

- Una expresión lambda permite una sintaxis más concisa:

```
button.addActionListener(event -> System.out.println("button clicked"));
```

parámetro

cuerpo de la expresión

expresión lambda

# Otro ejemplo: filtrando una lista

## *Programación estilo Java 7*

```
class Producto {  
    private int precio;  
    public Producto(int p) { this.precio = p; }  
    public int getPrecio() { return this.precio; }  
}
```

```
List<Producto> productos = Arrays.asList(  
    new Producto(20),  
    new Producto(40),  
    new Producto (5));
```

```
List<Producto> descuentos = new ArrayList<Producto>();
```

```
for (Producto p : productos)  
    if (p.getPrecio()>10.0)  
        descuentos.add(p);
```

# Otro ejemplo: filtrando una lista

## *Programación con lambdas*

```
class Producto {  
    private int precio;  
    public Producto(int p) { this.precio = p; }  
    public int getPrecio() { return this.precio; }  
}
```

```
List<Producto> productos = Arrays.asList(  
    new Producto(20),  
    new Producto(40),  
    new Producto (5));
```

```
List<Producto> descuentos = productos.stream().  
    filter(p -> p.getPrecio()>10.0).    // filtramos los > 10  
    collect(Collectors.toList());        // los ponemos en una lista
```

# Azúcar sintáctico...

*(y alguna cosa más)*

```
Stream<Producto> filter(Predicate<? super Producto> a)
```

@FunctionalInterface

```
public interface Predicate<T> {  
    //... más cosas  
    boolean test(T t);  
}
```

El compilador genera  
una clase anónima

```
List<Producto> descs2 = productos.stream().  
    filter(new Predicate<Producto>() {  
        @Override public boolean test(Producto a) {  
            return a.getPrecio()>10.0;  
        }  
    }).collect(Collectors.toList());
```





# Expresiones Lambda

# Expresiones Lambda

## ¿Qué son?

- Bloque de código sin nombre, formado por:
  - Lista de parámetros formales,
  - Separador “->”
  - Cuerpo.

`(int x) -> x + 1`

- Parece un método, pero no lo es: es una instancia de una interfaz funcional.
- Más precisamente, es una notación compacta para una instancia de una clase anónima, tipada por una interfaz funcional.

# Expresiones Lambda

## ¿Qué son?

- Una interfaz funcional es una interfaz con un único método no default.
- Algunas interfaces funcionales importantes:

Nombre	Argumentos	Retorno	Método funcional	Ejemplo
Predicate<T>	T	boolean	test(T t)	¿Tiene descuento el producto?
Consumer<T>	T	void	accept(T t)	Imprimir un valor
Function<T,R>	T	R	apply(T t)	Obtener precio de un Producto
Supplier<T>	None	T	get()	Creación de un objeto
UnaryOperator<T>	T	T	apply(T t)	Negación lógica (!)
BinaryOperator<T>	(T, T)	T	apply(T t, T u)	Multiplicar dos números (*)

- Algunas tienen especializaciones: IntConsumer
- Otras contienen métodos default y static de utilidad.

# Expresiones Lambda

## ¿Qué son?

- Las expresiones lambda no tienen:
  - ☐ Nombre
  - ☐ Declaración del tipo de retorno (se infiere).
  - ☐ Cláusula throws (se infiere)
  - ☐ Declaración de tipos genéricos
- Los tipos de los parámetros formales se pueden omitir (lambdas implícitas vs. explícitas).
  - ☐ O se omiten todos los parámetros o ninguno.
- Si se incluyen los tipos, se puede añadir el modificador final a los parámetros.

# Parámetros y retorno

## ■ Con cero parámetros y sin retorno

```
Runnable noArguments = () -> System.out.println("Hello World");  
noArguments.run();
```

```
/* Equivalente a  
Runnable noArguments = new Runnable() {  
    @Override public void run() {  
        System.out.println("Hello World");  
    }  
};  
*/
```

# Parámetros y retorno

- La siguiente sintaxis es incorrecta:



Runnable noArguments = -> System.out.println("Hello World");

# Parámetros y retorno

- Con un parámetro, varias instrucciones y sin retorno:

```
Consumer<Producto> consumer = p -> {  
    p.incrementaPrecio(10);  
    System.out.println(p.getNombre()+": "+p.getPrecio());  
};
```

```
List<Producto> productos = Arrays.asList(  
    new Producto(20, "Sal"),  
    new Producto(40, "Azucar"),  
    new Producto (5, "Vino"));
```

```
productos.forEach(p -> {  
    p.incrementaPrecio(10);  
    System.out.println(p.getNombre()+": "+p.getPrecio());  
});
```

```
productos.forEach(consumer); // equivalente a lo anterior
```

# Parámetros y retorno

- Las siguientes sintaxis son equivalentes:



```
Consumer<Producto> consumer = p -> {    // lambda implícita
    p.incrementaPrecio(10);
    System.out.println(p.getNombre()+": "+p.getPrecio());
};
```



```
Consumer<Producto> consumer = (p) -> {
    p.incrementaPrecio(10);
    System.out.println(p.getNombre()+": "+p.getPrecio());
};
```



```
Consumer<Producto> consumer = (Producto p) -> { // lambda explícita
    p.incrementaPrecio(10);
    System.out.println(p.getNombre()+": "+p.getPrecio());
};
```



# Parámetros y retorno

- La siguiente sintaxis es incorrecta:



```
Consumer<Producto> consumer = Producto p -> {  
    p.incrementaPrecio(10);  
    System.out.println(p.getNombre()+": "+p.getPrecio());  
};
```

# Parámetros y retorno

## ■ Con dos parámetros y con retorno:

```
List<Integer> numeros = Arrays.asList(1, 1, 2, 3, 5, 8, 13);
```

```
// Optional es un tipo que admite un resultado o null...
```

```
// ...permite una notación “funcional” para if...then...else
```

```
Optional<Integer> result =
```

```
    numeros.stream().
```


```
        reduce((x, y) -> x+y); // BinaryOperator<Integer>
```

```
System.out.println("Suma="+result.orElse(0));
```


```
// si no hay resultado, imprime 0
```

# Parámetros y retorno

- Las siguientes sintaxis son equivalentes:



```
Optional<Integer> result =  
    numeros.stream().  
        reduce((x, y) -> { return x+y; });
```



```
Optional<Integer> result =  
    numeros.stream().  
        reduce((Integer x, Integer y) -> { return x+y; });
```

# Variables del contexto

- En una lambda, podemos usar variables del contexto externo que sean finales...

```
List<Producto> productos = Arrays.asList(new Producto(20, "Sal"), new  
Producto(40, "Azucar"), new Producto (5, "Vino"));
```

```
final int incremento = 10;
```

```
productos.forEach(p -> {  
    p.incrementaPrecio(incremento); //incremento es final, podemos usarla  
    System.out.println(p.getNombre()+": "+p.getPrecio());  
});
```

# Variables del contexto

## ■ ... o efectivamente finales

```
List<Producto> productos = Arrays.asList(new Producto(20, "Sal"), new  
Producto(40, "Azucar"), new Producto (5, "Vino"));
```

```
int incremento = 10;
```

```
productos.forEach(p -> {  
    p.incrementaPrecio(incremento); // no cambiamos incremento, OK!  
    System.out.println(p.getNombre()+": "+p.getPrecio());  
});
```

# Variables del contexto

## ■ ... o efectivamente finales

```
List<Producto> productos = Arrays.asList(new Producto(20, "Sal"), new  
Producto(40, "Azucar"), new Producto (5, "Vino"));
```

```
int incremento = 10;
```

```
productos.forEach(p -> {  
    incremento += 3; // ERROR!!  
    p.incrementaPrecio(incremento);  
    System.out.println(p.getNombre()+": "+p.getPrecio());  
});
```



# Ejercicio

Filters y maps

# Ejercicio

- Diseña una clase que almacene secuencialmente datos de cualquier tipo, y que pueda devolver una secuencia filtrada de esos datos por criterios configurables.



# Ejercicio: *currying*

- En programación funcional, el *currying* es una técnica muy utilizada para reducir los parámetros de una función
- Dada una función  $f: X \times Y \rightarrow Z$ ,  $\text{curry}(f)$  devuelve una función  $h: X \rightarrow (Y \rightarrow Z)$ .
- Es decir,  $h$  toma un argumento de tipo  $X$ , y devuelve una función  $Y \rightarrow Z$ , definida de forma que  $h(x)(y) = f(x, y)$ .
- Para hacer:
  - Implementa *curry* usando expresiones lambda.
  - Sugerencia:
    - se puede utilizar la interfaz `BiFunction<X, Y, Z>` para modelar  $f$ , y `Function<Y, Z>` para modelar  $h$ .

# Ejercicio: *currying*

- En programación funcional es una técnica muy utilizada para transformar una función
- Dada una función  $h: X \rightarrow Y$  se puede transformar en una función  $h'$  que devuelve una función  $h'(x): Y \rightarrow Z$ .
- Es decir,  $h$  toma un argumento  $x$  y devuelve una función  $h'(x): Y \rightarrow Z$ .  
 $h'(x)(y) = f(x, y)$ .
- Para hacer:
  - ☐ Implementar la transformación de currying.
  - ☐ Sugerencia:
    - se puede usar la función `curry` de `Function`.



una técnica muy utilizada para transformar una función  $h: X \rightarrow Y$  en una función  $h'$  que devuelve una función  $h'(x): Y \rightarrow Z$ .  
Es decir,  $h$  toma un argumento  $x$  y devuelve una función  $h'(x): Y \rightarrow Z$ .  
 $h'(x)(y) = f(x, y)$ .

lambda.

$Z$  para modelar  $f$ ,

# Solución

```
package currying;

import java.util.function.*;

public class Currying {
    private <X, Y, Z> Function<X, Function<Y, Z>> curry(BiFunction<X, Y, Z> f){
        return x -> (y -> f.apply(x, y));
    }

    public static void main(String ...args) {
        Currying c = new Currying();
        Integer result =
            c.<Integer, Integer, Integer>curry((x, y) -> x + y ).
                apply(3).
                apply(4);
        System.out.println(result);
    }
}
```

# Interfaces funcionales

- Una interfaz funcional es una interfaz que tiene exactamente un método abstracto.
- No cuentan para definir la interfaz:
  - Métodos default
  - Métodos estáticos
  - Métodos heredados de Object
- Se puede anotar de manera opcional con `@FunctionalInterface` (en `java.lang`).
  - El compilador chequea que efectivamente la interfaz declarada es funcional.

# Ejemplo (1/2)

// Un sistema de objetos con métodos dinámicos  
// embebido en Java

```
@FunctionalInterface interface Method {  
    void exec(ProtoObject o);  
}
```

```
public class ProtoObject {  
    private HashMap<String, Object> slots = new HashMap<>();  
    private HashMap<String, Method> methods = new HashMap<>();  
  
    public void add (String name, Method m) { this.methods.put(name, m); }  
    public void add (String name, Object v) { this.slots.put(name, v); }  
    public Object get (String name) { return this.slots.get(name); }  
    public void exec (String name) { this.methods.get(name).exec(this); }  
    @Override public String toString() { return this.slots.toString(); }  
}
```

# Ejemplo (2/2)

```
public class Main {  
    public static void main(String[] args) {  
        ProtoObject p = new ProtoObject();  
        p.add("nombre", "Leonard Nimoy");  
        p.add("edad", 83);  
        p.add("incrementaEdad",  
            self -> {  
                self.add( "edad",  
                    ((Integer)self.get("edad"))+1);  
            }  
        );  
        p.add("imprime",  
            self -> {  
                System.out.println("nombre: "+self.get("nombre")+  
                    "\n"+"edad: "+self.get("edad")+" años.");  
            }  
        );  
        System.out.println(p);  
        p.exec("incrementaEdad");  
        p.exec("imprime");  
        System.out.println(p);  
    }  
}
```

Salida:

```
{nombre=Leonard Nimoy, edad=83}  
nombre: Leonard Nimoy  
edad: 84 años.  
{nombre=Leonard Nimoy, edad=84}
```

# Ejercicio

- Modifica el ejemplo anterior para que:
  - Se pueda clonar un objeto (el prototipo).
  - El objeto creado almacene una referencia a su prototipo
  - Si accedemos a una variable de un objeto, y ese objeto no le ha dado un valor, o no lo tiene, se busca la variable en el prototipo.
  - Añadir un método o variable en el prototipo se refleje en sus clones, pero no al revés.

Algunos lenguajes con un esquema de trabajo similar:

Self: [http://en.wikipedia.org/wiki/Self\\_%28programming\\_language%29](http://en.wikipedia.org/wiki/Self_%28programming_language%29)

JavaScript: <http://en.wikipedia.org/wiki/JavaScript>

# Interfaces funcionales genéricas

- Una interfaz funcional puede tener parámetros genéricos.
- Ejemplo:

```
@FunctionalInterface
public interface Comparator<T> {
    int compare(T o1, T o2);
}
```



# Interfaces funcionales genéricas

```
class Persona {  
    private String nombre;  
    private int edad;  
  
    public Persona(String n, int e) { this.nombre = n; this.edad = e; }  
    public String toString() { return "nombre: "+this.nombre+" edad: "+this.edad; }  
    public int getEdad() { return this.edad; }  
}
```

```
public class Comparar {  
    public static void main(String[] args) {  
        List<Persona> list = Arrays.asList(new Persona("Leonard Simon Nimoy", 83),  
                                            new Persona("William Shatner", 84),  
                                            new Persona("Jackson DeForest", 79));  
  
        Collections.sort(list, (x, y) -> x.getEdad() - y.getEdad());  
        System.out.println(list);  
        Collections.sort(list, (x, y) -> y.getEdad() - x.getEdad());  
        System.out.println(list);  
    }  
}
```

# Uso de Interfaces Funcionales

## *Function y sus especializaciones*

```
import java.util.function.*;

public class FunctionExample {
    public static void main(String[] args) {
        // Usando Function y sus especializaciones
        Function<Integer, Integer> square = x -> x * x;
        IntFunction<String> toStrn = x -> String.valueOf(x); // De entero a String
        ToIntFunction<Float> floor = x -> Math.round(x); // De float a Integer
        UnaryOperator<Integer> square2 = x -> x * x; // De Integer a Integer
        System.out.println(square.apply(5));
        System.out.println(toStrn.apply(5));
        System.out.println(floor.applyAsInt(5f));
        System.out.println(square2.apply(5));
    }
}
```

Salida

25  
5  
5  
25

# Function

## *Algunos métodos default y static*

```
@FunctionalInterface public interface Function<T, R> {  
    R apply(T t); // El método funcional  
    default <V> Function<V, R> compose(Function<? super V, ? extends T> before) {  
        Objects.requireNonNull(before);  
        return (V v) -> apply(before.apply(v));  
    }  
    default <V> Function<T, V> andThen(Function<? super R, ? extends V> after) {  
        Objects.requireNonNull(after);  
        return (T t) -> after.apply(apply(t));  
    }  
    static <T> Function <T, T> identity() {  
        return t->t;  
    }  
}
```

# Componiendo Funciones

```
public class ComposedFunctions {  
    public static void main(String[] args) {  
        // Create two functions  
        Function<Long, Long> square = x -> x * x;  
        Function<Long, Long> addOne = x -> x + 1;  
        // Compose functions from the two functions  
        Function<Long, Long> squareAddOne = square.andThen(addOne);  
        Function<Long, Long> addOneSquare = square.compose(addOne);  
        // Get an identity function  
        Function<Long, Long> identity = Function.<Long>identity();  
        // Test the functions  
        long num = 5L;  
        System.out.println("Number : " + num);  
        System.out.println("Square and then add one: " + squareAddOne.apply(num));  
        System.out.println("Add one and then square: " + addOneSquare.apply(num));  
        System.out.println("Identity: " + identity.apply(num));  
    }  
}
```

**Salida:**

Number : 5  
Square and then add one: 26  
Add one and then square: 36  
Identity: 5

# Predicate

```
@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);  
    }  
}
```

# Predicate

```
public class Predicates {  
    public static void main(String[] args) {  
        // Create some predicates  
        Predicate<Integer> greaterThanTen = x -> x > 10;  
        Predicate<Integer> divisibleByThree = x -> x % 3 == 0;  
        Predicate<Integer> divisibleByFive = x -> x % 5 == 0;  
        Predicate<Integer> equalToTen = Predicate.isEqual(null);  
        // Create predicates using NOT, AND, and OR on other predicates  
        Predicate<Integer> lessThanOrEqualToTen=greaterThanTen.negate();  
        Predicate<Integer> divisibleByThreeAndFive=divisibleByThree.and(divisibleByFive);  
        Predicate<Integer> divisibleByThreeOrFive=divisibleByThree.or(divisibleByFive);  
        // Test the predicates  
        int num = 10;  
        System.out.println("Number: " + num);  
        System.out.println("greaterThanTen: " + greaterThanTen.test(num));  
        System.out.println("divisibleByThree: " + divisibleByThree.test(num));  
        System.out.println("divisibleByFive: " + divisibleByFive.test(num));  
        System.out.println("lessThanOrEqualToTen: " + lessThanOrEqualToTen.test(num));  
        System.out.println("divisibleByThreeAndFive: " +  
            divisibleByThreeAndFive.test(num));  
        System.out.println("divisibleByThreeOrFive: " +  
            divisibleByThreeOrFive.test(num));  
        System.out.println("equalsToTen: " + equalToTen.test(num));  
    }  
}
```

# Ejercicio (1/2)

- Usando lambdas, crea un simulador para máquinas de estados.
- Una máquina de estados está formada por estados y una serie de variables, que asumimos de tipo Integer.
- Se pasa de un estado a otro cuando sucede un evento (de tipo String).
- Los estados pueden tener asociadas acciones, que se ejecutan al entrar en el estado, y pueden por ejemplo modificar las variables.

# Ejercicio (2/2)

```
public class Main {  
    public static void main(String[] args) {  
        StateMachine sm = new StateMachine("Light", "num"); // nombre y variable  
        State s1 = new State("off");  
        State s2 = new State("on");  
        s1.addEvent("switch", s2);  
        s2.addEvent("switch", s1);  
        s1.action((State s, String e) -> s.set("num", s.get("num")+1) );  
        s2.action((State s, String e) -> s.set("num", s.get("num")+1) );  
  
        sm.addStates(s1, s2);  
        sm.setInitial(s1);  
  
        System.out.println(sm);  
        MachineSimulator ms = new MachineSimulator(sm);  
        ms.simulate(Arrays.asList("switch", "switch"));  
    }  
}
```

Salida:

```
Machine Light : [off, on]  
switch: from [off] to [on]  
    Machine variables: {num=1}  
switch: from [on] to [off]  
    Machine variables: {num=2}
```



# Conclusiones

- Las expresiones lambda introducen flexibilidad y concisión a la hora de especificar operaciones con colecciones de elementos.
- Otras ventajas, como facilidad de paralelización.
- **NO** se han cubierto aspectos avanzados:
  - Referencias a métodos
  - Streams
  - El API de interfaces funcionales y streams es muy extenso.
    - Parte de este API se explorará y practicará en las prácticas.
    - El paradigma funcional (lisp) se estudiará más en detalle en la asignatura de Inteligencia Artificial.
  - Diseño de lenguajes embebidos en Java usando lambdas: ([http://en.wikipedia.org/wiki/Domain-specific\\_language](http://en.wikipedia.org/wiki/Domain-specific_language))
    - El diseño de lenguajes embebidos en Ruby se estudiará en la asignatura de Desarrollo Automatizado de Software.

# Bibliografía

- Java 8 Lambdas. Functional Programming for the masses. O'Reilly. Richard Warburton. 2014.
- Beginning Java 8 Language Features. Kishori Sharan. Apress. Agosto 2014.
- Functional Programming in Java. Harnessing the power of Java 8 Lambda Expressions. The Pragmatic Programmers. V. Subrmanian. 2014.