



Tema 3.9

Expresiones

Lambda y Java 8

Análisis y Diseño de Software

2º Ingeniería Informática

Universidad Autónoma de Madrid

Indice

- **Nuevos conceptos en interfaces**
 - Métodos default.
 - Métodos estáticos.
- Expresiones Lambda
- Ejercicios
- Conclusiones y bibliografía

Nuevos conceptos en interfaces

- Considera la siguiente interfaz

```
public interface Arbol<T>{  
    T getElemento();  
    Arbol<T> hijoIzq();  
    Arbol<T> hijoDer();  
    boolean esHoja();  
    boolean esVacio();  
    T search(T o);  
}
```

- Para facilitar su uso, podríamos dar una implementación por defecto para algunos métodos.
- Esta implementación puede usar otros métodos de la interfaz.
- Las clases no necesitan dar una implementación de un método default, pero pueden hacerlo.

Métodos default

```
public interface Arbol<T>{
    T getElemento();
    Arbol<T> hijoIzq();
    Arbol<T> hijoDer();
    default boolean esHoja() {
        return this.hijoIzq().esVacio() && this.hijoDer().esVacio();
    }
    boolean esVacio();
    default T search(T o) {
        if (this.getElemento().equals(o)) return this.getElemento();
        else {
            T result = null;
            if (! this.hijoIzq().esVacio() ) result = this.hijoIzq().search(o);
            if (result != null) return result;
            if (! this.hijoDer().esVacio() ) result = this.hijoDer().search(o);
            if (result != null) return result;
        }
        return null;
    }
}
```

Métodos default: motivación

- Poder añadir métodos a una interfaz sin romper código que ya funciona (los nuevos métodos de la interfaz serían métodos default).
- Especificar métodos que son opcionales:
 - Dar una implementación que devuelve una excepción.
- Facilitar la implementación de interfaces (similar a una clase abstracta con implementaciones de referencia).

Diferencias con clases abstracta

- Una interfaz no tiene estado interno:
 - No puede declarar atributos (variables de instancia), sólo constantes.
- Una clase sólo puede heredar de una clase, mientras que puede implementar varias interfaces.
- El propósito de las interfaces sigue siendo especificar “qué” (firmas de métodos) y no “cómo” (código en los métodos).

Métodos default y herencia múltiple

- Sin métodos default, no hay problemas de colisión de nombres con herencia múltiple o implementación múltiple.
- El método tiene un solo código, implementado en la clase.

```
interface Alfa {  
    int metodo1();  
}
```

```
interface Beta {  
    int metodo1();  
}
```

```
public class Prueba implements Alfa, Beta{  
    @Override public int metodo1() {return 42; } // OK!  
}
```

Métodos default y herencia múltiple

- Si un método default se implementa en una clase, dicha implementación tiene preferencia.

```
interface Alfa {  
    default int metodo1() { return 23;}  
}  
  
interface Beta {  
    int metodo1();  
}  
  
public class Prueba implements Alfa, Beta{  
    @Override public int metodo1() {return 42; }  
  
    public static void main(String... args) {  
        System.out.println(new Prueba().metodo1());  
        // el de la clase toma preferencia  
    }  
}
```


Métodos default y herencia múltiple

- Si hay colisión de nombres y alguno tiene código, la clase ha de dar una implementación.

```
interface Alfa {  
    default int metodo1() { return 23;}  
}
```

```
interface Beta {  
    default int metodo1() { return 89; }  
}
```

```
public class Prueba implements Alfa, Beta{ // Error!  
                                           // hay que dar implementacion  
    public static void main(String... args) {  
        System.out.println(new Prueba().metodo1());  
    }  
}
```

Métodos default y herencia múltiple

- Si hay colisión de nombres y alguno tiene código, la clase ha de dar una implementación.

```
interface Alfa {  
    default int metodo1() { return 23;}  
}
```

```
interface Beta {  
    default int metodo1() { return 89; }  
}
```

```
public class Prueba implements Alfa, Beta{  
    // Usar el de Alfa...  
    @Override public int metodo1() {return Alfa.super.metodo1(); }  
  
    public static void main(String... args) {  
        System.out.println(new Prueba().metodo1()); // 23  
    }  
}
```

Métodos default y herencia múltiple

- La notación `SuperType.super.metodo()` es usable también en interfaces.

```
interface Alfa {  
    default int metodo1() { return 23;}  
}
```

```
interface Beta extends Alfa {  
    default int metodo1() { return Alfa.super.metodo1()+1; }  
}
```

```
public class Prueba implements Beta{  
    public static void main(String... args) {  
        System.out.println(new Prueba().metodo1()); // 24  
    }  
}
```

Métodos estáticos en interfaces

- Es posible añadir métodos estáticos en una interfaz, de manera similar a como se hace en una clase.
- Útil para definir librerías.
 - Ejemplo: creación de algunos comparadores útiles en `Comparator<T>`

```
public interface Comparator<T> {  
    int compare(T o1, T o2);  
    static <T extends Comparable<? super T>> Comparator<T> naturalOrder() { ... }  
  
    static <T> Comparator<T> nullsFirst(Comparator<? super T> comparator) { ... }  
    static <T> Comparator<T> nullsLast(Comparator<? super T> comparator) { ... }  
    //...  
}
```

Indice

- Nuevos conceptos en interfaces

- **Expresiones Lambda**

- Introducción y ejemplos

- Expresiones lambda

- Streams

- Ejercicios

- Conclusiones y bibliografía



Introducción y Ejemplos

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 λ -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:

parámetro

cuerpo de la expresión

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

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<T>
- 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());  
});
```

Referencias a métodos

- Una referencia a un método es una “abreviatura” para una lambda que usa dicho método.
- El método no se llama en ese momento, es simplemente una lambda
- Sintaxis: <QualifiedName>::<methodName>

```
class Producto {  
    private int precio;  
    private String nombre;  
  
    public Producto(int p, String n) { this.precio = p; this.nombre = n; }  
    public int getPrecio() { return this.precio; }  
    public String getNombre() { return this.nombre; }  
}  
  
// Obtener el nombre de los productos que empiezan por S,  
// sin repetición
```

```
Set<String> descs2 = productos.stream().  
    map(Producto::getNombre).    // una referencia a método  
    filter( s -> s.startsWith("S")).  
    collect(Collectors.toSet());
```

Referencias a métodos

```
class Producto {  
    private int precio;  
    private String nombre;  
  
    public Producto(int p, String n) { this.precio = p; this.nombre = n; }  
    public int getPrecio() { return this.precio; }  
    public String getNombre() { return this.nombre; }  
}
```

```
Set<String> descs2 = productos.stream().  
    map(p -> p.getNombre()). // equivalente  
    filter( s -> s.startsWith("S")).  
    collect(Collectors.toSet());
```

Referencias a métodos

Tipos

<code><NombreTipo>::<metodoEstatico></code>	Una referencia a un método estático de una clase, interfaz o enum
<code><refObjeto>::<metodoInstancia></code>	Una referencia a un método de instancia del objeto
<code><NombreClase>::<metodoInstancia></code>	Una referencia a un método de instancia de la clase
<code><NombreTipo>.super:: <metodoInstancia></code>	Una referencia a un método de la superclase del objeto actual
<code><NombreClass>::new</code>	Una referencia al constructor de la clase
<code><NombreTipoArray>::new</code>	Una referencia al constructor del tipo de array

Referencias a métodos de Objeto

```
class Almacen {  
    private List<Producto> productos = new ArrayList<Producto>();  
  
    public Almacen(Producto...productos) {  
        this.productos.addAll(Arrays.asList(productos));  
    }  
    public String getNombre(Producto p) { return p.getNombre(); }  
    public Stream<Producto> getProductos() { return this.productos.stream(); }  
}
```

```
class Producto { /* como antes */}
```

```
Almacen alm = new Almacen(new Producto(20, "Sal"));
```

```
Set<String> descs2 = alm.getProductos().  
    map(alm::getNombre).// una referencia a método de objeto  
    filter( s -> s.startsWith("S")).  
    collect(Collectors.toSet());
```

Ambigüedad y casting

```
@FunctionalInterface interface IntegerReduce {
    int join (int x, int y);
}
@FunctionalInterface interface StringReduce {
    String join (String x, String y);
}

public class Joiner {
    public String doJoin (StringReduce sj) { return sj.join("Java", "8");}
    public int doJoin(IntegerReduce ij) { return ij.join(64, 128); }

    public static void main(String[] args) {
        Joiner j = new Joiner();
        System.out.println(j.doJoin((x, y) -> x + y));
        // Error: The method doJoin(StringReduce) is ambiguous for the type Joiner
    }
}
```

Ambigüedad y casting

```
@FunctionalInterface interface IntegerReduce {
    int join (int x, int y);
}
@FunctionalInterface interface StringReduce {
    String join (String x, String y);
}

public class Joiner {
    public String doJoin (StringReduce sj) { return sj.join("Java", "8"); }
    public int doJoin(IntegerReduce ij) { return ij.join(64, 128); }

    public static void main(String[] args) {
        Joiner j = new Joiner();
        System.out.println(j.doJoin((StringReduce)(x, y) -> x + y));
    }
}
```

Ambigüedad y casting

```
@FunctionalInterface interface IntegerReduce {
    int join (int x, int y);
}
@FunctionalInterface interface StringReduce {
    String join (String x, String y);
}

public class Joiner {
    public String doJoin (StringReduce sj) { return sj.join("Java", "8");}
    public int doJoin(IntegerReduce ij) { return ij.join(64, 128); }

    public static void main(String[] args) {
        Joiner j = new Joiner();
        System.out.println(j.doJoin((String x, String y) -> x + y));
        // Equivalente a lo anterior
    }
}
```


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}
```

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:

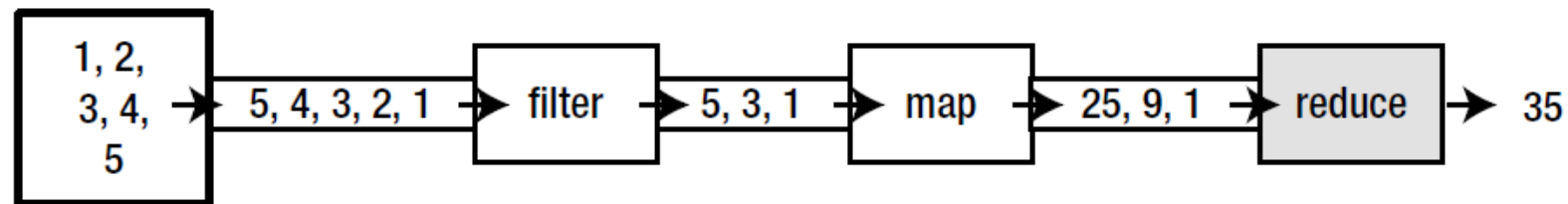
```
Machine Light : [off, on]  
switch: from [off] to [on]  
    Machine variables: {num=1}  
switch: from [on] to [off]  
    Machine variables: {num=2}
```

Tipos intersección y lambdas

- Un tipo intersección (nuevo en Java8) es una intersección o subtipo de múltiples tipos.
- La expresión: (Type1 & Type2 & Type3) es un tipo nuevo, que es la intersección de los 3 tipos.

```
Serializable comp = (Comparator<Persona> & Serializable)  
                    (x, y) -> x.getEdad() - y.getEdad();
```

Streams



```
numbers.stream( ).filter(n -> n % 2 == 1).map(n -> n * n).reduce(0, Integer::sum)
```

Streams

- Es una secuencia de datos que soporta operaciones secuenciales o paralelas de agregación.
 - Calcular la suma de sus elementos.
 - Mapear el nombre de todas las personas de una lista a su longitud.
- Similar a una colección, pero:
 - Diseñados para **programación declarativa**/funcional (en contraste con el código más imperativo de las colecciones).
 - Soportan **iteración interna**.
 - **No** tienen **almacenamiento** (saca los elementos de una fuente de datos **bajo demanda**).
 - Pueden representar una **secuencia infinita**.
 - Diseñados para facilitar **paralelización** de las operaciones.
 - Soportan **operaciones “perezosas”**.
 - Streams ordenados (e.g, fuente de datos ordenada, como un List, o bien porque se ha ordenado con sort) y desordenados.

Iteración interna vs externa

■ Iteración externa:

```
List<Integer> numbers =  
    Arrays.asList(1, 2, 3, 4, 5);
```

```
int sum = 0;  
for (int n : numbers) {  
    if (n % 2 == 1) {  
        int square = n * n;  
        sum = sum + square;  
    }  
}
```

```
System.out.println(sum);
```

- El cliente extrae los elementos (for), los itera y les aplica un algoritmo.

■ Iteración interna:

```
List<Integer> numbers =  
    Arrays.asList(1, 2, 3, 4, 5);
```

```
int sum = numbers.stream()  
    .filter(n -> n % 2 == 1)  
    .map(n -> n * n)  
    .reduce(0, Integer::sum);
```

```
System.out.println(sum);
```

- El cliente pasa el algoritmo al stream.
- El stream aplica el algoritmo, iterando internamente.

Iteración interna vs externa

Paralelización

■ Iteración externa:

```
List<Integer> numbers =  
    Arrays.asList(1, 2, 3, 4, 5);
```

```
int sum = 0;  
for (int n : numbers) {  
    if (n % 2 == 1) {  
        int square = n * n;  
        sum = sum + square;  
    }  
}
```

```
System.out.println(sum);
```

- El cliente extrae los elementos (for), los itera y les aplica un algoritmo.

■ Iteración interna:

```
List<Integer> numbers =  
    Arrays.asList(1, 2, 3, 4, 5);
```

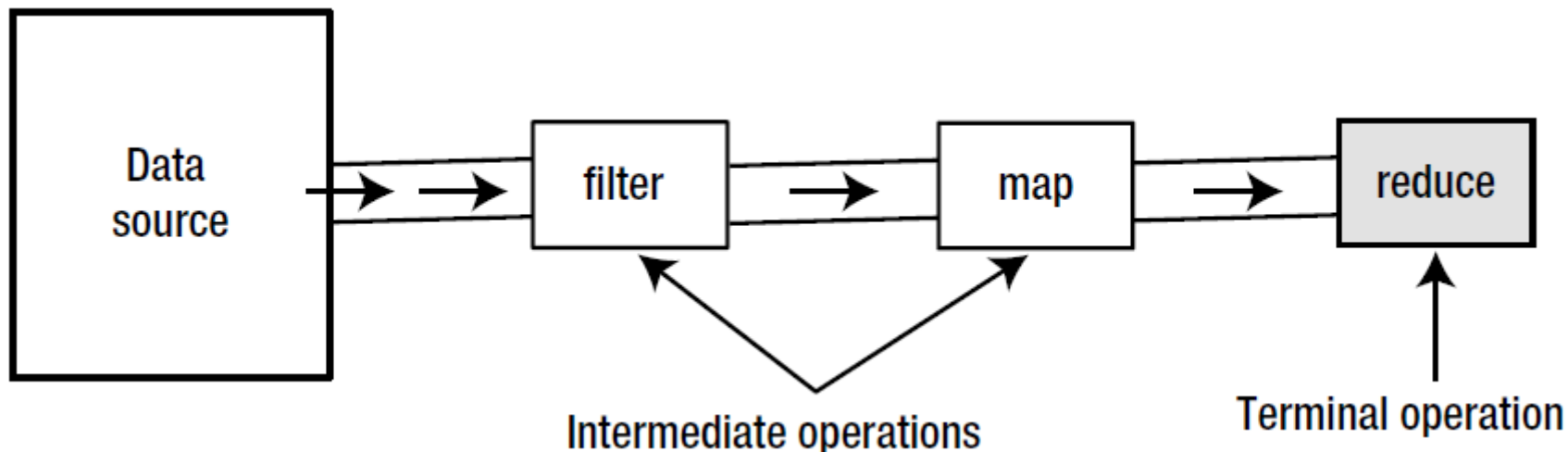
```
int sum = numbers.parallelStream()  
    .filter(n -> n % 2 == 1)  
    .map(n -> n * n)  
    .reduce(0, Integer::sum);
```

```
System.out.println(sum);
```

- Podemos paralelizar las operaciones sobre el stream para sacar partido de los procesadores con varios cores.

Operando sobre streams

- Dos tipos de operación
 - Operaciones intermedias, que son perezosas (lazy).
 - Operaciones terminales, que son codiciosas (eager).
- Una operación lazy no saca elementos del stream hasta que no se llama a una eager.
- Cadena de operaciones lazy aplicadas a un stream
 - Cada una produce un nuevo stream
- La operación terminal saca las entradas del stream, **inicia el cómputo** y produce un resultado.



Creando streams

- Se pueden crear:
 - ☐ Vacíos.
 - ☐ De valores.
 - ☐ De funciones.
 - ☐ De arrays.
 - ☐ De colecciones.
 - ☐ De ficheros.
 - ☐ De otras fuentes.

Desde valores

- `<T> Stream<T> of(T t)`
- `<T> Stream<T> of(T...values)`

```
// Calcular la media de letras en las palabras de un String
Stream<String> words = Stream.of("Java 8 es superguay".split("\\s+"));

OptionalDouble media = words.mapToInt(String::length).average();

System.out.println("media: "+media.orElse(0.0));
```

Desde valores

- `<T> Stream<T> of(T t)`
- `<T> Stream<T> of(T...values)`

```
// Más estadísticas: no es posible re-procesar un Stream
long palabras = Stream.of("Java 8 es superguay".split("\\s+")).
    collect(Collectors.counting());

long suma = Stream.of("Java 8 es superguay".split("\\s+")).
    mapToInt(String::length).sum();

OptionalInt minimo = Stream.of("Java 8 es superguay".split("\\s+")).
    mapToInt(String::length).min();

OptionalInt maximo = Stream.of("Java 8 es superguay".split("\\s+")).
    mapToInt(String::length).max();

OptionalDouble media = Stream.of("Java 8 es superguay".split("\\s+")).
    mapToInt(String::length).average();

System.out.println(palabras+" "+suma+" "+
    minimo.orElse(0)+" "+maximo.orElse(0)+" "+media.orElse(0.0));
```

Desde valores

- `<T> Stream<T> of(T t)`
- `<T> Stream<T> of(T...values)`

```
IntSummaryStatistics estadisticas =  
    Stream.of("Java 8 es superguay".split("\\s+")).  
        collect(Collectors.summarizingInt(String::length));
```

```
System.out.println(estadisticas);
```

```
// Salida:
```

```
// IntSummaryStatistics{count=4, sum=16, min=1, average=4.000000, max=9}
```

Desde funciones

- `<T> Stream<T> iterate(T seed, UnaryOperator<T> f)`
- `<T> Stream<T> generate(Supplier<T> s)`

```
Stream.iterate(0L, n -> n + 2)    // stream infinito [0, 2, 4,...)
    .limit(5)                      // cogemos los 5 primeros
    .forEach(System.out::println); // los imprimimos
```

```
Stream.generate(Math::random)    // stream infinito de nº aleatorios
    .limit(5)                     // cogemos los 5 primeros
    .forEach(System.out::println);
```

```
Stream.iterate(0L, n -> n + 2)
    .skip(100)                    // saltamos 100
    .limit(5)
    .forEach(System.out::println);
```

Desde arrays y colecciones

- `Arrays.stream`
- Método `stream()` y `parallelStream()` sobre colecciones

```
IntStream numbers = Arrays.stream(new int[]{1, 2, 3});
```

```
Stream<String> words =  
    Arrays.asList("esto", "se", "convierte", "en",  
                  "lista").stream();
```


Valores opcionales (Optional)

- El valor null se usa para representar una ausencia de valor.
- Dificulta la concatenación de operaciones sobre un stream.
- Optional<T> representa un valor de tipo T que puede ser null.
 - isPresent(): true si no es null
 - ifPresent(Consumer<? super T> action): ejecuta la lambda si no es null

```
// Create an Optional for the string "Hello"
Optional<String> str = Optional.of("Hello");
// Print the value in the Optional, if present
str.ifPresent(value -> System.out.println("Optional contains " + value));
```

Algunas operaciones sobre Streams (1/2)

Operación	Tipo	Descripción
distinct	intermedia	Devuelve un stream con los elementos distintos
filter	Intermedia	Devuelve un stream con los elementos que cumplen el predicado
flatMap	Intermedia	Devuelve un stream con el resultado de aplicar una función sobre los elementos del stream. La función produce un stream por cada elemento, que se aplana.
limit	Intermedia	Devuelve un stream de longitud menor o igual que el límite que se le pasa
map	Intermedia	Devuelve un stream con el resultado de aplicar una función sobre los elementos del stream
peek	Intermedia	Devuelve este stream, pero aplica una acción al consumir elementos (útil para debug)
skip	Intermedia	Descarta los n primeros elementos y devuelve el stream con los siguientes.
sorted	intermedia	Devuelve un stream ordenado de acuerdo al orden natural o a un Comparator

Algunas operaciones sobre Streams (2/2)

Operación	Tipo	Descripción
allMatch	terminal	Devuelve true si todos los elementos del stream cumplen el predicado
anyMatch	terminal	Devuelve true si algún elemento del stream cumplen el predicado
findAny	terminal	Devuelve elemento del stream. Se devuelve un Optional vacío si el stream está vacío.
findFirst	terminal	Devuelve el primer elemento del stream (si está desordenado es uno arbitrario)
noneMatch	terminal	Devuelve true si ningún elemento del stream cumple el predicado
forEach	terminal	Aplica una acción a cada elemento del stream
reduce	terminal	Aplica una operación de reducción que calcula un valor único para el stream

Ejemplo debug

```
int sum = Stream.of(1, 2, 3, 4, 5)
    .peek(e -> System.out.println("Taking integer: " + e))
    .filter(n -> n % 2 == 1)
    .peek(e -> System.out.println("Filtered integer: " + e))
    .map(n -> n * n)
    .peek(e -> System.out.println("Mapped integer: " + e))
    .reduce(0, Integer::sum);
```

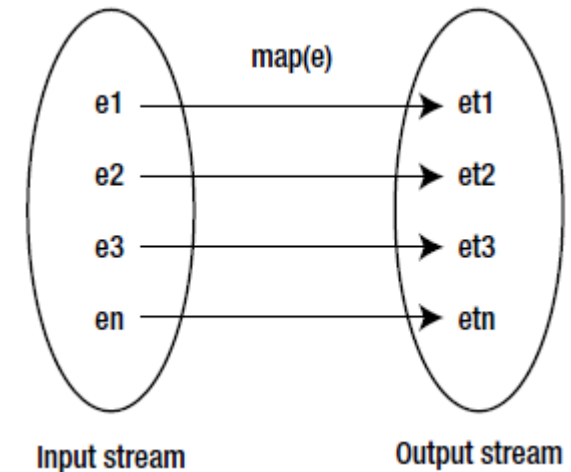
```
System.out.println("Sum = " + sum);
```

```
Taking integer: 1
Filtered integer: 1
Mapped integer: 1
Taking integer: 2
Taking integer: 3
Filtered integer: 3
Mapped integer: 9
Taking integer: 4
Taking integer: 5
Filtered integer: 5
Mapped integer: 25
Sum = 35
```

Operación map

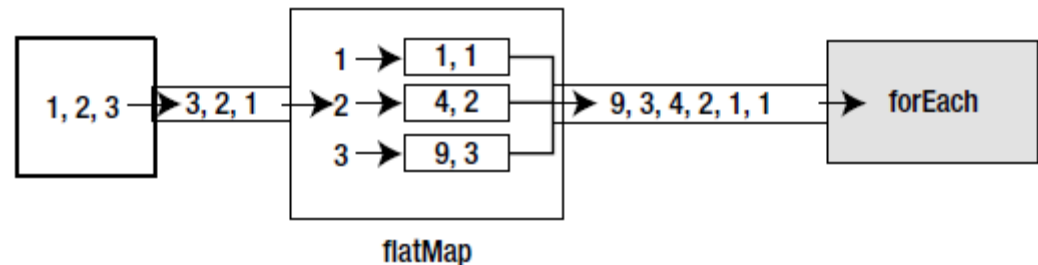
- Aplica una función a cada elemento del stream
- Versiones especializadas, que devuelven subclases de Stream:
 - `<R> Stream<R> map(Function<? super T, ? extends R> mapper)`
 - `DoubleStream mapToDouble(ToDoubleFunction<? super T> mapper)`
 - `IntStream mapToInt(ToIntFunction<? super T> mapper)`
 - `LongStream mapToLong(ToLongFunction<? super T> mapper)`

```
IntStream.rangeClosed(1, 5)
    .map(n -> n * n)
    .forEach(System.out::println);
```



Operación flatMap

- Aplica una función que produce un stream a cada elemento del stream
- Aplana el stream de streams resultante.



```
Stream.of(1, 2, 3)
    .flatMap(n -> Stream.of(n, n * n))
    .forEach(System.out::println);
```

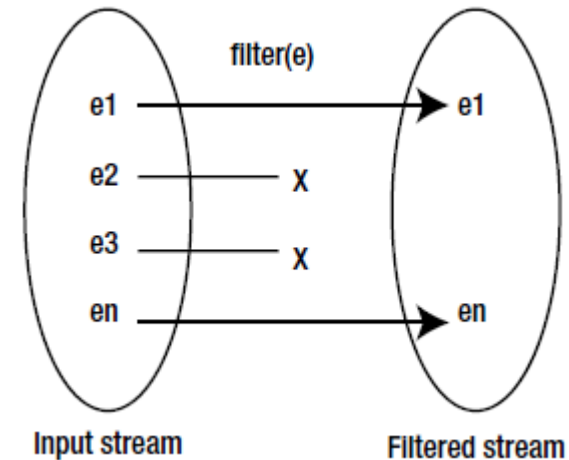
Salida:

1
1
2
4
3
9

Operación filter

- Produce un stream con los elementos que cumplen el predicado.

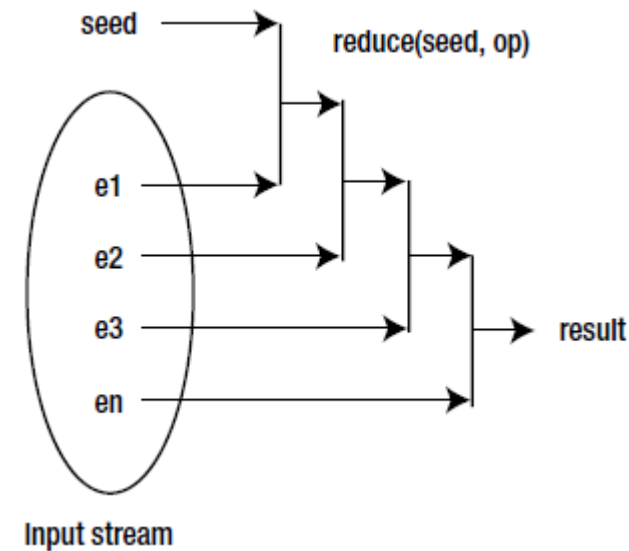
```
class Producto{  
    private int precio;  
    private String nombre;  
    public Producto(int p, String n) {  
        this.precio = p;  
        this.nombre = n;  
    }  
    public int getPrecio() { return this.precio; }  
    public String getNombre() { return this.nombre; }  
}
```



```
Stream<Producto> sprod = Stream.of( new Producto(20, "Sal"),  
                                     new Producto(40, "Azucar"),  
                                     new Producto (5, "Vino"));  
sprod.filter( s -> s.getPrecio() > 10).  
    map(Producto::getNombre).  
    forEach(System.out::println);  
// Salida: Sal Azucar
```

Operación reduce

- Combina todos los elementos del stream en un único valor.
- Toma un valor inicial y un acumulador
- Un tercer parámetro es una lambda “combinadora” para combinar resultados de múltiples threads en caso de ejecución paralela.



```
String conc = Stream.of("voy ", "a ", " concatenar").  
                    reduce("Concatenación: ", String::concat);  
  
System.out.println(conc); // Salida: Concatenación: voy a  concatenar
```


Collectors

- Se usan cuando hay que guardar los resultados de operar en un Stream en una Colección.
- O hay que aplicar lógica compleja al resumir la información de un Stream.
 - `<R> R collect(Supplier<R> supplier, BiConsumer<R,? super T> accumulator, BiConsumer<R,R> combiner)`
 - `<R,A> R collect(Collector<? super T,A,R> collector)`

```
List<String> nombres =  
    sprod.map(Producto::getNombre).  
        collect(ArrayList::new, ArrayList::add, ArrayList::addAll);
```

```
System.out.println(nombres);  
// Salida: [Sal, Azucar, Vino]
```

Collectors

- Se usan cuando hay que guardar los resultados de operar en un Stream en una Colección.
- O hay que aplicar lógica compleja al resumir la información de un Stream.
 - `<R> R collect(Supplier<R> supplier, BiConsumer<R,? super T> accumulator, BiConsumer<R,R> combiner)`
 - `<R,A> R collect(Collector<? super T,A,R> collector)`

```
List<String> nombres =  
    sprod.map(Producto::getNombre).  
        collect(Collectors.toList()); // equivalente a lo anterior
```

```
System.out.println(nombres);  
// Salida: [Sal, Azucar, Vino]
```

Agrupando datos en Mapas

- `groupBy(Function<? super T,? extends K> classifier)`
- `groupBy(Function<? super T,? extends K> classifier, Collector<? super T,A,D> downstream)`

```
class Producto{  
    enum TipoProducto {ALIMENTO, BEBIDA}  
    private TipoProducto tp;  
    private int precio;  
    private String nombre;  
    ...}
```

```
Stream<Producto> sprod = Stream.of(  
    new Producto(20, "Sal", TipoProducto.ALIMENTO),  
    new Producto(40, "Azucar", TipoProducto.ALIMENTO),  
    new Producto (5, "Vino", TipoProducto.BEBIDA));
```

```
Map<TipoProducto, List<Producto>> prodsPorTipo =  
    sprod.collect(Collectors.groupingBy(Producto::getTipo));
```

```
System.out.println(prodsPorTipo);
```

```
// Salida: {BEBIDA=[Vino (5€)], ALIMENTO=[Sal (20€), Azucar (40€)]}
```

Agrupando datos en Mapas

- `groupBy(Function<? super T,? extends K> classifier)`
- `groupBy(Function<? super T,? extends K> classifier, Collector<? super T,A,D> downstream)`

```
class Producto{  
    enum TipoProducto {ALIMENTO, BEBIDA}  
    private TipoProducto tp;  
    private int precio;  
    private String nombre;  
    ...}
```

```
Stream<Producto> sprod = Stream.of(  
    new Producto(20, "Sal", TipoProducto.ALIMENTO),  
    new Producto(40, "Azucar", TipoProducto.ALIMENTO),  
    new Producto (5, "Vino", TipoProducto.BEBIDA));
```

```
Map<TipoProducto, Integer> prodsPorTipo =  
    sprod.collect(Collectors.groupingBy(Producto::getTipo, Collectors.counting()));
```

```
System.out.println(prodsPorTipo);  
// Salida: {ALIMENTO=2, BEBIDA=1}
```

Agrupando datos en Mapas

- `groupBy(Function<? super T,? extends K> classifier)`
- `groupBy(Function<? super T,? extends K> classifier, Collector<? super T,A,D> downstream)`

```
Stream<Producto> sprod = Stream.of(  
    new Producto(20, "Sal", TipoProducto.ALIMENTO),  
    new Producto(40, "Azucar", TipoProducto.ALIMENTO),  
    new Producto (5, "Vino", TipoProducto.BEBIDA));
```

```
Map<TipoProducto, List<String>> prodsPorTipo =  
    sprod.collect(Collectors.groupingBy(Producto::getTipo,  
        Collectors.mapping(Producto::getNombre,  
            Collectors.toList())));
```

```
System.out.println(prodsPorTipo);  
// Salida: {BEBIDA=[Vino], ALIMENTO=[Sal, Azucar]}
```

Indice

- Nuevos conceptos en interfaces
- Expresiones Lambda
- **Ejercicios**
- Conclusiones y bibliografía

Ejercicios (1/6)

- Sea el siguiente programa para el manejo de pedidos:

```
package productos;

public enum TipoProducto {
    ALIMENTO (6), TABACO (21), ALCOHOL (21);

    private double iva;

    private TipoProducto(double imp) { this.iva = imp;}

    public double getIva() { return this.iva;}
}
```

Ejercicios (2/6)

```
package productos;
```

```
public class Producto {  
    private String nombre;  
    private double precio;  
    private TipoProducto tipo;
```

```
    public Producto(String n, double p, TipoProducto t) {  
        this.nombre = n; this.precio = p; this.tipo = t;  
    }
```

```
    public double precio() { return this.precio*(1+this.tipo.getIva()*0.01);}
```

```
    @Override public String toString() {  
        return this.nombre+" (" +this.precio+", "+this.tipo.getIva()+"%)" ;  
    }
```

```
    public TipoProducto getTipo() { return this.tipo; }  
}
```


Ejercicios (3/6)

```
package productos;  
import ...;
```

```
public class Pedido {  
    private Map<Producto, Integer> pedido =  
        new LinkedHashMap<Producto, Integer>();  
  
    public Pedido addLineaPedido(int n, Producto p) {  
        this.pedido.put(p, n); return this;  
    }  
    @Override public String toString() {  
        return this.pedido.toString();  
    }  
}
```

Ejercicios (4/6)

```
public static void main(String[] args) {  
    Producto p1 = new Producto("Aceitunas", 2, TipoProducto.ALIMENTO);  
    Producto p2 = new Producto("Cerveza", 1, TipoProducto.ALCOHOL);  
    Producto p3 = new Producto("Ducados", 4, TipoProducto.TABACO);  
    Producto p4 = new Producto("Patatas fritas", 1.5, TipoProducto.ALIMENTO);  
    Producto p5 = new Producto("Jamón", 10.5, TipoProducto.ALIMENTO);  
  
    Pedido pedido = new Pedido().  
        addLineaPedido(1, p1).  
        addLineaPedido(5, p2).  
        addLineaPedido(2, p4).  
        addLineaPedido(1, p5);  
  
    System.out.println(pedido);  
}
```

{Aceitunas (2.0, 6.0%)=1, Cerveza (1.0, 21.0%)=5, Patatas fritas (1.5, 6.0%)=2, Jamón (10.5, 6.0%)=1}

Ejercicios (5/6)

- Modifica la clase Pedido, para que sea posible:
 - Obtener los productos agrupados por tipo (Alimento, Alcohol, Tabaco).
 - Obtener el número de productos de cada tipo.
 - Obtener el precio desglosado por tipo.
 - Obtener el total del pedido (usando stream)
 - Obtener el precio total de los productos que cumplan una condición.

Ejercicios (6/6)

```
public static void main(String[] args) {  
    ...  
    System.out.println("Pedido por tipos: "+  
                        pedido.productosPorTipo());  
    System.out.println("Numero elementos por tipo: "+  
                        pedido.totalProductosPorTipo());  
    System.out.println("Desglose precio por tipo: "+  
                        pedido.totalPrecioPorTipo());  
    System.out.println("Coste de productos con precio neto mayor de 1 euro: "+  
                        pedido.total( p -> p.precio() > 2));  
}
```

{Aceitunas (2.0, 6.0%)=1, Cerveza (1.0, 21.0%)=5, Patatas fritas (1.5, 6.0%)=2, Jamón (10.5, 6.0%)=1}
Total: 22.48 €

Pedido por tipos: {ALCOHOL=[Cerveza (1.0, 21.0%)], ALIMENTO=[Aceitunas (2.0, 6.0%), Patatas fritas (1.5, 6.0%), Jamón (10.5, 6.0%)]}

Numero elementos por tipo: {ALCOHOL=1, ALIMENTO=3}

Desglose precio por tipo: {ALCOHOL=6.05, ALIMENTO=16.43}

Coste de productos con precio neto mayor de 1 euro: 13.25

(Clase Pedido)

Solución

```
public Map<TipoProducto, List<Producto>> productosPorTipo() {  
    return this.pedido.keySet().stream().  
        collect(Collectors.groupingBy(Producto::getTipo,  
                                       Collectors.mapping(p -> p,  
                                                           Collectors.toList())));  
}  
  
public Map<TipoProducto, Long> totalProductosPorTipo() {  
    return this.pedido.keySet().stream().  
        collect(Collectors.groupingBy(Producto::getTipo, Collectors.counting()));  
}  
  
public Map<TipoProducto, Double> totalPrecioPorTipo() {  
    return this.pedido.keySet().stream().  
        collect(Collectors.groupingBy(Producto::getTipo,  
                                       Collectors.summingDouble( (Producto p) ->  
                                                                 p.precio()*this.pedido.get(p))));  
}
```

(Clase Pedido)

Solución

```
public double total() {  
    return this.pedido.keySet().stream().  
        mapToDouble( p -> p.precio()*this.pedido.get(p) ).  
        sum();  
}
```

```
public double total(Predicate<Producto> pred) {  
    return this.pedido.keySet().stream().  
        filter(pred).  
        mapToDouble( p -> p.precio()*this.pedido.get(p) ).  
        sum();  
}
```

Indice

- Nuevos conceptos en interfaces
- Expresiones Lambda
- Ejercicios
- **Conclusiones y bibliografía**

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:
 - 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)
 - 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.