



UD 2: Modelo Vista Controlador

9.- Expresiones Lambda, Streams y Collectors



Desarrollo Web en Entorno Servidor - Joseramon.profesor@gmail.com

Objetivos de la sesión:

- Entender que son las **expresiones Lambda**
- Entender que son las **interfaces funcionales**
- Distinguir los distintos **tipos de expresiones Lambda**: Consumidores, Proveedores, Funciones, Predicados y Referencias a métodos.
- Aprender a utilizar expresiones Lambda para **recorrer una lista**
- Aprender a utilizar expresiones Lambda para **comparar datos**
- Aprender a utilizar **Streams** y sus métodos **para filtrar, ordenar o extraer información de una colección de datos**:

Filter

Distinct

AnyMatch

Reduce

Sorted

FindFirst

Map

Collect(Collectors):toList(), toSet(), toMap()

Count



UD 2: Modelo Vista Controlador

9.- Expresiones Lambda, Streams y Collectors

Desarrollo Web en Entorno Servidor - joseramon.profesor@gmail.com



En esta sección entenderemos que son las expresiones Lambda, los Streams y como se entrelazan pudiendo utilizar Collectors para facilitarnos la programación.



¿Que son las empresiones Lambda?



UD 2: Modelo Vista Controlador

9.- Expresiones Lambda, Streams y Collectors

Desarrollo Web en Entorno Servidor - Jose Ramon.profesor@gmail.com



• Expresiones Lambda

El JDK 8 incorporó una característica nueva a Java que mejora su poder expresivo, las expresiones lambda. La expresión lambda se basa en el **operador lambda** (“->”) u **operador de flecha**.

(parámetros) -> { cuerpo-lambda }

Por ejemplo: (a,b) -> { System.out.println(a + b); }

El operador lambda (“->”) divide una expresión lambda en **dos partes**:

1. El lado izquierdo especifica los **parámetros requeridos** por la expresión lambda.
2. En el lado derecho está el **cuerpo lambda**, que especifica las **acciones** de la expresión lambda.

No te preocupes si no entiendes el código mostrado en la parte derecha :
¡¡En breve lo entenderas!!

```
InterfazFuncionalApp.java X
InterfazFuncionalApp.java > InterfazFuncionalApp > main(String[])
1  import java.util.Scanner;
2
3  @FunctionalInterface
4  public interface IFuncionalCalculo {
5      public void calcula(int a, int b);
6  }
7
8  public class InterfazFuncionalApp {
9      Run | Debug
10     public static void main(String[] args) {
11         Scanner leer = new Scanner(System.in);
12         IFuncionalCalculo suma=(a, b) -> {
13             System.out.println(a + b);
14         };
15         IFuncionalCalculo esDivisor=(a, b) -> {
16             System.out.println(a%b==0);
17         };
18         System.out.println("Dame el 1º número:");
19         int valor1=leer.nextInt();
20         System.out.println("Dame el 2º número:");
21         int valor2=leer.nextInt();
22         System.out.print("Suma=");
23         suma.calcula( valor1, valor2);
24         System.out.print("¿El 2º es divisor del 1º?");
25         esDivisor.calcula(valor1, valor2);
26         leer.close();
27     }
```



UD 2: Modelo Vista Controlador

9.- Expresiones Lambda, Streams y Collectors



Desarrollo Web en Entorno Servidor - joseramon.profesor@gmail.com

- **Expresiones Lambda** (parámetros) -> { cuerpo-lambda }

Parámetros:

- Cuando se tiene un solo parámetro no es necesario utilizar los paréntesis.
 `a -> a*2`
- Cuando no se tienen parámetros, o cuando se tienen dos o más, es necesario utilizar paréntesis.

`() -> System.out.println("Hola")`

`(int base, int altura) -> base*altura`

Cuerpo-lambda:

Cuando el cuerpo de la expresión lambda tiene una única línea no es necesario utilizar las llaves y no necesitan especificar la clausula return en el caso de que deba devolver valores. Sin embargo, cuando el cuerpo de la expresión lambda tiene más de una línea se hace necesario utilizar las llaves y es necesario incluir la clausula return en el caso de que la función deba devolver un valor .

```
(n) -> { //calcula el divisor más pequeño
    int res=1;
    n = n<0 ? -n:n;// Obtenga el valor absoluto de n.
    for (int i=2; i<=n/i;i++)
        if ((n%i)==0) {
            res = i;
            break;
        }
    return res;
};
```




UD 2: Modelo Vista Controlador

9.- Expresiones Lambda, Streams y Collectors

Desarrollo Web en Entorno Servidor - Joseramon.profesor@gmail.com



· Expresiones Lambda (parámetros) -> { cuerpo-lambda }

Una **expresión lambda** es básicamente un **método anónimo** (es decir, sin nombre). Las expresiones lambda también se conocen comúnmente como **cierres (closures)**. En **Javascript** se les suele dar el nombre de **expresiones flecha**.

¿Cual es el motivo para complicarnos con esta nueva estructura? ¿Que ventaja nos aporta esta forma de declarar “pseudométodos anónimos” ?

Para entender el valor añadido que aporta las expresiones lambda vamos a ver ejemplos de expresiones lambda **asociadas a interfaces funcionales y su utilización final , los Streams. Con los Streams le sacaremos todo el provecho.**



¿Que es una interfaz funcional?



UD 2: Modelo Vista Controlador

9.- Expresiones Lambda, Streams y Collectors



Desarrollo Web en Entorno Servidor - Jose Ramon.profesor@gmail.com

· Expresiones Lambda: Interfaces funcionales

Una interfaz funcional es una interfaz que **SOLO tiene un método y dicho método es abstracto**. Recordemos que a partir del JDK 8 una interfaz podía implementar un método (dejando de ser abstracto) antenponiendo la palabra default.

Aunque es **opcional** se puede declarar la anotación **@FunctionalInterface**. Esta anotación le indica al compilador que se trata de una interfaz funcional.

Normalmente, este método especifica el propósito previsto de la interfaz. Además, una interfaz funcional permite definir el objetivo de la expresión lambda.

Por ejemplo, la interfaz estándar Runnable es una interfaz funcional porque define solo un método: run(). Entonces, run() define la acción de Runnable.

A veces a las interfaces funcionales se les conoce como tipo SAM, donde **SAM** significa **Single Abstract Method**.

```
InterfazFuncionalApp.java x
InterfazFuncionalApp.java > InterfazFuncionalApp > main(String[])
1  import java.util.Scanner;
2
3  @FunctionalInterface
4  public interface IFuncionalCalculo {
5      public void calcula(int a, int b);
6  }
7
8  public class InterfazFuncionalApp {
9      Run | Debug
10     public static void main(String[] args) {
11         Scanner leer = new Scanner(System.in);
12         IFuncionalCalculo suma=(a, b) -> {
13             System.out.println(a + b);
14         };
15         IFuncionalCalculo esDivisor=(a, b) -> {
16             System.out.println(a%b==0);
17         };
18         System.out.println("Dame el 1º número:");
19         int valor1=leer.nextInt();
20         System.out.println("Dame el 2º número:");
21         int valor2=leer.nextInt();
22         System.out.print("Suma=");
23         suma.calcula( valor1, valor2);
24         System.out.print(";El 2º es divisor del 1º?");
25         esDivisor.calcula(valor1, valor2);
26         leer.close();
27     }
```



UD 2: Modelo Vista Controlador

9.- Expresiones Lambda, Streams y Collectors

Desarrollo Web en Entorno Servidor - Joseramon.profesor@gmail.com



· Expresiones Lambda: Tipos

Las expresiones lambda se pueden clasificar en :

- **Consumidores**
- **Proveedores**
- **Funciones**
 - **Operadores Unarios**
 - **Operadores Binarios**
- **Predicados**
- **Referencias a métodos**

En el Drive están las aplicaciones con el código de ejemplo para que el alumno pueda realizar pruebas y trazas.

¡¡Es recomendable ejecutar las aplicaciones en modo depuración y realizar una traza para entender los conceptos de esta sesión!!



• Expresiones Lambda: Tipos: Consumidores

Son expresiones que aceptan un solo valor (consumen un valor) y no devuelven valor alguno. Si tienen 2 parámetros se suelen llamar Biconsumidoras. Suelen utilizarse mucho con la **interfaz Consumer<T>** (a partir del JDK 1.8) y su método asociado **accept(T)** para ejecutar el método:

```
EjemploConsumidorApp.java X
EjemploConsumidorApp.java > EjemploConsumidorApp > main(String[])
1  import java.util.function.Consumer;
2
3  class Persona {
4      private String nombre;
5      private String apellido;
6      private String direccion;
7
8  > public Persona(String nombre, String apellido, String direccion)
13 > public String getNombre() { ...
17 > public void setNombre(String nombre) { ...
21 > public String getApellido() { ...
25 > public void setApellido(String apellido) { ...
29 > public String getDireccion() { ...
33 > public void setDireccion(String direccion) { ...
37 }
38
39 @FunctionalInterface
40 interface IFuncionalSaludo {
41     public void imprime(String nom);
42 }
43
44 @FunctionalInterface
45 interface IFuncionalSaludoCompleto {
46     public void imprime(String nom, String apel);
47 }
48
```

```
EjemploConsumidorApp.java X
EjemploConsumidorApp.java > ...
48
49 public class EjemploConsumidorApp {
50     Run | Debug
51     public static void main(String[] args) {
52         // Ejemplos expresiones Lambda consumidoras con Interfaces funcionales
53         IFuncionalSaludo saludo = (nombre) -> System.out.println("Hola "
54                                     + nombre);
55         saludo.imprime("Jose");
56         IFuncionalSaludoCompleto saludoCompleto = (nombre, apellido) -> System.out
57             .println("Hola " + nombre + " " + apellido);
58         saludoCompleto.imprime("Jose Ramón", "Cebolla");
59
60         // Ejemplo expresiones Lambda con Interfaz Consumer<T>
61         Consumer<String> saludoConConsumer = (nombre) -> System.out.println("Hola "
62                                     + nombre);
63         saludoConConsumer.accept("Jose");
64         Consumer<Persona> persona = (p) -> System.out.println("Hola, "
65                                     + p.getNombre());
66         persona.accept(new Persona("Jose Ramón ", "Cebolla", "Rotova"));
67     }
68 }
```



```
joseramon@Notebook-PC:~/Escritorio/2020.Simarro/SRV/UD2/ejemplos$
Hola Jose
Hola Jose Ramón Cebolla
Hola Jose
Hola, Jose Ramón
```




• Expresiones Lambda: Tipos: Proveedores

Son expresiones que no tienen parámetros pero devuelven resultados (proveen un resultado). Suelen utilizarse mucho con la **interfaz Supplier<T>** (a partir del JDK 1.8) y su método asociado **get()** para proveer el resultado:

EjemploProveedorApp.java ×

```

1  import java.util.function.Supplier;
2
3
4  class Persona {
5      private String nombre;
6      private String apellido;
7      private String direccion;
8
9      public Persona(){}
10
11 > public Persona(String nombre, String apellido, String direccion) {}
12
13 > public String getNombre() { ... }
14
15 > public void setNombre(String nombre) { ... }
16
17 > public String getApellido() { ... }
18
19 > public void setApellido(String apellido) { ... }
20
21 > public String getDireccion() { ... }
22
23 > public void setDireccion(String direccion) { ... }
24
25 > public String toString() { ... }
26
27 }

```

EjemploProveedorApp.java ×

```

47  @FunctionalInterface
48  interface IFuncionalProveedora {
49      public String getStr();
50  }
51
52  public class EjemploProveedorApp {
53
54      public static Persona personaPorDefecto(){
55          return new Persona("nombre", "apellido", "direccion");
56      }
57
58      Run | Debug
59      public static void main(String[] args) {
60          // Ejemplos expresiones Lambda proveedoras con Interfaces funcionales
61          IFuncionalProveedora hola = () -> "Hello!!";
62          System.out.println(hola.getStr());
63
64          // Ejemplo expresiones Lambda con Interfaz Supplier<T>
65          Supplier<String> holaConSupplier = () -> "Hola Supplier!!";
66          System.out.println(holaConSupplier.get());
67          Supplier<Persona> supplierPersona=()-> personaPorDefecto();
68          Persona p=supplierPersona.get();
69          System.out.println(p.getNombre()+ " "+p.getApellido()
70              + " vive en " + p.getDireccion());
71      }

```

joseramon@Notebook-PC:~/Escritorio/2020.Simarro/SRV/UD2/ejemplos\$
Hello!!
Hola Supplier!!
nombre apellido vive en direccion



· Expresiones Lambda: Funciones

Son expresiones que consumen un argumento y proveen un valor como resultado y cuyos tipos no tienen porque ser iguales. Las **BiFunciones** son aquellas expresiones de tipo función que aceptan dos argumentos y devuelven un resultado. Suelen utilizarse mucho con la **interfaz** **Function<TipoArgumento, TipoRespuesta>** (a partir del JDK 1.8) y su método asociado **apply()** :

```
EjemploFuncionApp.java X
EjemploFuncionApp.java > ...
1  import java.util.function.Function;
2
3  class Persona {
4      private String nombre;
5      private String apellido;
6      private String direccion;
7
8  >   public Persona(String nombre, String apellido, String direccion) {
13
14 >   public String getNombre() {
17
18 >   public void setNombre(String nombre) {
21
22 >   public String getApellido() {
25
26 >   public void setApellido(String apellido) {
29
30 >   public String getDireccion() {
33
34 >   public void setDireccion(String direccion) {
37 }
38
39 @FunctionalInterface
40 interface IFuncionalCalculo {
41     public double getResultado(int radio);
42 }
```

```
EjemploFuncionApp.java X
EjemploFuncionApp.java > ...
43
44 public class EjemploFuncionApp {
45     Run | Debug
46     public static void main(String[] args) {
47         // Ejemplos expresiones Lambda consumidoras con Interfaces funcionales
48         IFuncionalCalculo area = radio -> (double)radio*radio*3.141516;
49         System.out.println("Area de un circulo de radio 5= " +
50             area.getResultado(5));
51
52         // Ejemplos expresiones Lambda con Interfaz Function<T,R>
53         Function<Integer, Double> areaConFunction=radio-> radio*radio*3.141516;
54         System.out.println("AreaConFunction de un circulo de radio 5= " +
55             areaConFunction.apply(5));
56
57         Persona personal=new Persona("nom1", "apellido1", "direccion1");
58         Function<Persona,String> nombrePersona= p-> p.getNombre();
59         System.out.println("Nombre de la persona= " +
60             nombrePersona.apply(personal));
61     }
62 }
```

joseramon@Notebook-PC:~/Escritorio/2020.Simarro/SRV/UD2/ejemplos\$
Area de un circulo de radio 5= 78.53790000000001
AreaConFunction de un circulo de radio 5= 78.53790000000001
Nombre de la persona= nom1



· Expresiones Lambda: Predicados

Se trata de expresiones que aceptan/consumen un parámetro y devuelven/proveen un valor lógico (verdadero o falso, de ahí el predicado). Suelen utilizarse mucho con la **interfaz Predicate<T>** (a partir del JDK 1.8) y su método asociado **test()** para devolver el predicado (verdadero o falso) :

```
EjemploPredicadosApp.java X
EjemploPredicadosApp.java > EjemploPredicadosApp > main(String[])
1  import java.util.function.Predicate;
2
3  > class Persona { ...
45
46  @FunctionalInterface
47  interface IFuncionalPredicado {
48      public Boolean comprobar(int num);
49  }
50
51  public class EjemploPredicadosApp {
52
53      Run | Debug
54      public static void main(String[] args) {
55          // Ejemplos expresiones Lambda Predicado con Interfaces Predicado
56          IFuncionalPredicado mayorEdad = edad -> edad >= 18;
57          System.out.println("¿Con 20 años eres mayor de edad? " +
58              mayorEdad.comprobar(20));
59
60          // Ejemplo expresiones Lambda con Interfaz Predicate<T>
61          Predicate<Integer> mayorEdadConPredicado = edad -> edad >= 18;
62          System.out.println("¿Con 10 años eres mayor de edad? " +
63              mayorEdadConPredicado.test(10));
64          System.out.println("¿Con 10 años eres menor de edad? " +
65              mayorEdadConPredicado.negate().test(10));
66          Persona personal = new Persona("nom", "apellido", "direccion");
67          Predicate<Persona> nombreCorto = p -> p.getNombre().length() < 5;
68          System.out.println("¿Nombre persona '" + personal.getNombre() +
69              "' es demasiado corto? " + nombreCorto.test(personal));
70      }
71  }
```

```
joseramon@Notebook-PC:~/Escritorio/2020.Simarro/SRV/UD2/ejemplos$
¿Con 20 años eres mayor de edad? true
¿Con 10 años eres mayor de edad? false
¿Con 10 años eres menor de edad? true
¿Nombre persona 'nom' es demasiado corto? true
```



· Expresiones Lambda: Referencias a métodos

Las referencias a los métodos nos permiten *reutilizar un método como expresión lambda*. Para hacer uso de las referencias a métodos basta con utilizar la siguiente **sintaxis**: ***Referencia::nombreDelMetodo***

```
EjemploReferenciaMetodosApp.java X
EjemploReferenciaMetodosApp.java > EjemploReferenciaMetodosApp > main(String[])
1 import java.util.function.Consumer;
2 import java.util.function.Function;
3 class Persona {
4     private String nombre;
5     private String apellido;
6     private String direccion;
7
8 > public Persona(String nombre, String apellido, String direccion) { ...
13
14 > public String getNombre() { ...
17
18 > public void setNombre(String nombre) { ...
21
22 > public String getApellido() { ...
25
26 > public void setApellido(String apellido) { ...
29
30 > public String getDireccion() { ...
33
34 > public void setDireccion(String direccion) { ...
37 }
38
39 @FunctionalInterface
40 interface IFuncionalImpresion {
41     public void imprime(String nom);
42 }
43
```

```
44 public class EjemploReferenciaMetodosApp {
45     Run | Debug
46     public static void main(String[] args) {
47         // Ejemplos expresiones Lambda consumidoras con Interfaces funcionales
48         IFuncionalImpresion nombre = (str) -> System.out.println(str);
49         nombre.imprime("Jose");
50         // Ejemplos expresiones Lambda que hace referencia a métodos
51         IFuncionalImpresion nombreReferencia = System.out::println;
52         // "Pedro" sera 'nom' y se le pasara al println
53         nombreReferencia.imprime("Pedro");
54
55         // Ejemplo expresiones Lambda con Interfaz Consumer<T>
56         Consumer<String> minuscula = (nom) -> nom.toLowerCase();
57         minuscula.accept("Jose");
58         Consumer<String> minusculaRef = String::toLowerCase;
59         // "Pedro" sera la variable 'nom' consumida sobre la que se ejecutará 'toLowerCase()'
60         minusculaRef.accept("Pedro");
61         // Ejemplo expresiones Lambda con Interfaz Function<T,R>
62         Persona personal = new Persona("nom1", "apellido1", "direccion1");
63         Function<Persona, String> nombrePersona = p -> p.getNombre();
64         System.out.println("Nombre de la persona = " +
65             nombrePersona.apply(personal));
66         // 'personal' será el objeto de tipo 'Persona' sobre el que se le invocará 'getNombre'
67         System.out.println("Nombre de la persona = " +
68             nombrePersonaReferencia.apply(personal));
69     }
70 }
```



```
josemon@Notebook-PC:~/Escritorio/2020.Simarro/SRV/UD2/ejemplos$
Jose
Pedro
Nombre de la persona= nom1
Nombre de la persona= nom1
```




UD 2: Modelo Vista Controlador

9.- Expresiones Lambda, Streams y Collectors

Desarrollo Web en Entorno Servidor - Joseramon.profesor@gmail.com



· Expresiones Lambda:

Hemos visto que existen *diferentes tipos de expresiones lambda* y que podemos crear una interfaz funcional para cada tipo.

Es una **buena práctica** no utilizar interfaces funcionales a medida siempre que podamos **escoger una interfaz funcional propia del JDK 8** porque cualquiera que vea nuestro código, sin saber lo que hace lo entienda mejor porqué tiene clara la funcionalidad de dicha interfaz:

· **Interfaz Consumer<T> con su método accept():** Cuando es una expresión que tienen un parámetro, pero no devuelve ningún dato.

```
Consumer<Persona> persona = (p) -> System.out.println("Hola, " + p.getNombre());
persona.accept(new Persona("Jose Ramón ", "Cebolla", "Rotova"));
```

· **Interfaz Supplier<T> con su método get():** Cuando la expresión no tienen parámetros, pero devuelven un dato.

```
Supplier<Persona> supplierPersona = () -> personaPorDefecto();
Persona p = supplierPersona.get();
System.out.println(p.getNombre() + " " + p.getApellido() + " vive en " + p.getDireccion());
```

· **Interfaz Function<T,R> con su método apply():** Cuando la expresión tienen un parámetro y devuelven un dato.

```
Persona personal = new Persona("nom1", "apellido1", "direccion1");
Function<Persona, String> nombrePersona = p -> p.getNombre();
System.out.println("Nombre de la persona = " + nombrePersona.apply(personal));
```

· **Interfaz Predicate<T> con su método test():** Cuando la expresión tienen un parámetro y devuelven un resultado booleano.

```
Persona personal = new Persona("nom", "apellido", "direccion");
Predicate<Persona> nombreCorto = p -> p.getNombre().length() < 5;
System.out.println("¿Nombre persona '" + personal.getNombre() +
    "' es demasiado corto? " + nombreCorto.test(personal));
```



UD 2: Modelo Vista Controlador

9.- Expresiones Lambda, Streams y Collectors

Desarrollo Web en Entorno Servidor - Joseramon.profesor@gmail.com



· Expresiones Lambda:

Esta nueva forma de programar al principio puede resultar muy compleja. Por lo tanto ahora la pregunta podría ser:



¿Que ventaja real le podemos sacar realmente a esta nueva forma de programar?



UD 2: Modelo Vista Controlador

9.- Expresiones Lambda, Streams y Collectors



Desarrollo Web en Entorno Servidor - Joseramon.profesor@gmail.com

· Expresiones Lambda: Recorrer una lista:

Hasta la versión 8 de Java si queríamos recorrer una lista teníamos 3 métodos.

Ahora nos aparece un sistema nuevo, las listas tiene un **método** *forEach()* que admite un **Consumer<T>** como parámetro:

```
Iterable.forEach(Consumer<? super String> action) : void
```

Performs the given action for each element of the Iterable until all elements have been processed or the action throws an exception. Actions are performed in the order of iteration, if that order is specified. Exceptions thrown by the action are relayed to the caller.

The behavior of this method is unspecified if the action performs side-effects that modify the underlying source of elements, unless an overriding class has specified a concurrent modification policy.

- **Parameters:**
 - **action** The action to be performed for each element
- **Throws:**
 - **NullPointerException** - if the specified action is null
- **Since:**
 - 1.8
- **@implSpec**
 - The default implementation behaves as if:

```
for (T t : this)
    action.accept(t);
```

```
RecorrerListaApp.java x
RecorrerListaApp.java > RecorrerListaApp > main(String[])
1  import java.util.ArrayList;
2  import java.util.Iterator;
3  import java.util.List;
4
5  public class RecorrerListaApp {
6      Run|Debug
7      public static void main(String[] args) {
8          List<String> paises = new ArrayList<String>();
9          paises.add("España"); // ocupa la posición 0
10         paises.add("Francia"); // ocupa la posición 1
11         paises.add("Portugal"); // ocupa la posición 2
12         // MÉTODOS TRADICIONALES (ANTES DEL JDK 1.8)
13         // 1º METODO
14         for (int i = 0; i < paises.size(); i++) {
15             System.out.println(paises.get(i));
16         }
17         // 2º METODO
18         for (String pais : paises) {
19             System.out.println(pais);
20         }
21         // 3º METODO
22         // Iterator iter=paises.iterator();
23         // No da error pero es mejor:
24         Iterator<String> iter = paises.iterator();
25         while (iter.hasNext()) { // V si quedan elementos
26             System.out.println(iter.next());
27             // next() retorna el elemento y apunta al siguiente
28         }
29         // RECORRER LISTA UTILIZANDO EXPRESIÓN LAMBDA Consumer<T>
30         paises.forEach(System.out::println);
31     }
32 }
```




· Expresiones Lambda: Comparadores:

Hasta ahora teníamos la **interfaz Comparable** para implementar un criterio de comparación a la hora de ordenar objetos del mismo tipo y una **clase Comparator** para poder indicar otros criterios de ordenación.

Una de las **aplicaciones prácticas de las expresiones Lambda** que esta disponible a partir de Java 8 es la utilización de **comparadores de una manera más simplificada**. Si quisiéramos comparar 2 personas por un criterio distinto al implementado en Comparable no haría falta implementar una clase Comparator:

```
EjemploComparatorApp.java X
EjemploComparatorApp.java > EjemploComparatorApp > main(String[])
1  import java.util.Comparator;
2
3  > class Persona { ...
38
39
40  public class EjemploComparatorApp {
    Run | Debug
41  public static void main(String[] args) {
42      Comparator<Persona> comparaPersonas = (p1, p2) -> p1.getNombre().compareTo(p2.getNombre());
43      Persona p1 = new Persona("Ana", "Gonzalez", "Xativa");
44      Persona p2 = new Persona("Jose", "Martinez", "Xativa");
45      System.out.println("p1>p2= "+comparaPersonas.compare(p1, p2)); // > 0
46      System.out.println("p1<p2= "+comparaPersonas.reversed().compare(p1, p2)); // < 0
47  }
48  }
49
```




UD 2: Modelo Vista Controlador

9.- Expresiones Lambda, Streams y Collectors

Desarrollo Web en Entorno Servidor - Joseramon.profesor@gmail.com



- **Streams:**

Un Stream **representa una secuencia de elementos de un tipo**. En la práctica estas secuencias son subclases de **java.util.Collection** (list, set,...). Los Map de momento no están soportados .

En un Stream podemos distinguir **3 partes**:

- **Fuente:**

Se trata de la lista o colección de donde obtenemos información.

- **Operaciones intermedias:**

Son operaciones cuyo resultado devuelven streams.

Por ejemplo el método filter, que pronto veremos que permite hacer una selección a partir de un predicado.

- **Operaciones terminales:**

Devuelven un resultado de un tipo.

Por ejemplo los métodos max, min, forEach, findFirst etc.



UD 2: Modelo Vista Controlador

9.- Expresiones Lambda, Streams y Collectors

Desarrollo Web en Entorno Servidor - Joseramon.profesor@gmail.com



• Streams:

Veamos algunos de los métodos que podemos invocar sobre un Stream:

Operación:

Intermedia

Terminal

Intermedia

Intermedia

Terminal

Intermedia

Terminal

Terminal

- **Filter** : Acepta un predicado para filtrar todos los elementos del Stream.
- **AnyMatch** : Acepta un predicado y nos devuelve true o false.
- **Sorted** : Devuelve una vista ordenada del stream.
- **Map** : Convierte cda elemento del stream en otro objeto via una funcion.
- **Count** : Devuelve en número de elementos.
- **Distinct** : Permite eliminar los elementos repetidos.
- **Reduce** : Reduce los elementos del stream utilizando una función.
- **FindFirst** : Devuelve un elemento de tipo 'Optional'.
- **Collect (Collectors)** : Convertir elementos del stream a otra estructura (lista, conjunto o mapa).



Streams: Filter:

Filter acepta un predicado para filtrar todos los elementos del Stream. Al tratarse de una operación intermedia debemos invocar a una operación terminal para devolver un resultado, por ejemplo `forEach()` :

```
StreamPaisesFilterApp.java X
StreamPaisesFilterApp.java > ...
1  import java.util.ArrayList;
2  import java.util.List;
3
4  public class StreamPaisesFilterApp {
5      Run|Debug
6      public static void main(String[] args) {
7          // A partir de Java 9 tenemos List.of
8          List<String> paises = new ArrayList<String>(
9              List.of("España", "Francia"));
10
11         //paises que empiezan por E
12         paises.stream().
13             filter(p->p.startsWith("E")).
14             forEach(System.out::println);
15     }
16 }
```



```
josemon@Notebook-PC:~/Escritorio/2020.Simarro/SRV/UD2/ejemplos$
España
```



Streams: AnyMatch:

Es un operador terminal que acepta un predicado y nos devuelve "True" o "False":

```
StreamPaisesAnyMatchApp.java X
StreamPaisesAnyMatchApp.java > StreamPaisesAnyMatchApp > main(String[])
1  import java.util.ArrayList;
2  import java.util.List;
3
4  public class StreamPaisesAnyMatchApp {
5      Run | Debug
6      public static void main(String[] args) {
7          // A partir de Java 9 tenemos List.of
8          List<String> paises = new ArrayList<String>(
9              List.of("España", "Portugal", "Francia"));
10         //Algún país contiene una i?
11         System.out.println(paises.stream().anyMatch(p->p.contains("i")));
12     }
13 }
```



```
joseramon@Notebook-PC:~/Escritorio/2020.Simarro/SRV/UD2/ejemplos$ java StreamPaisesAnyMatchApp
true
```




Streams: Sorted:

Es una operación intermedia que devuelve una vista ordenada del stream. Puede utilizar un Comparator si lo necesitamos. Es importante resaltar que no modifica el orden de la colección original:

```
EjemploStreamSortedApp.java X
EjemploStreamSortedApp.java > EjemploStreamSortedApp > main(String[])
1  import java.util.Comparator;
2  import java.util.List;
3  import java.util.ArrayList;
4
5  > class ComparaDireccion implements Comparator<Persona> { ...
10
11 > class Persona implements Comparable<Persona> { ...
86
87 public class EjemploStreamSortedApp {
    Run | Debug
    public static void main(String[] args) {
88         Comparator<Persona> comparaNombreApellidos = (p1, p2) ->
89         {int compara = p1.getNombre().compareTo(p2.getNombre());
90         if (compara != 0)
91             return compara;
92         else
93             return p1.getApellido().compareTo(p2.getApellido());
94         };
95         Persona persona1 = new Persona("Juan", "Zamora", "Gandia");
96         Persona persona2 = new Persona("Ana", "Zurrieta", "Gandia");
97         Persona persona3 = new Persona("Ana", "Balbastre", "Xativa");
98         List<Persona> lista = new ArrayList<Persona>(
99             List.of(persona1, persona2, persona3));
100         System.out.println("ORDEN por defecto (Comparable por nombre):");
101         lista.stream().sorted().forEach(System.out::println);
102         System.out.println("ORDEN ComparaDireccion:");
103         lista.stream().sorted(new ComparaDireccion()).forEach(System.out::println);
104         System.out.println("ORDEN comparaNombreApellidos:");
105         lista.stream().sorted(comparaNombreApellidos).forEach(System.out::println);
106         System.out.println("La lista realmente no ha cambiado:");
107         lista.forEach(System.out::println);
108     }
109 }
110 }
```



```
joseramon@Notebook-PC:~/Escritorio/2020.Simarro/SRV/UD2/ejemplos$
ORDEN por defecto (Comparable por nombre):
Persona [nombre=Ana, apellido=Zurrieta, direccion=Gandia]
Persona [nombre=Ana, apellido=Balbastre, direccion=Xativa]
Persona [nombre=Juan, apellido=Zamora, direccion=Gandia]
ORDEN ComparaDireccion:
Persona [nombre=Juan, apellido=Zamora, direccion=Gandia]
Persona [nombre=Ana, apellido=Zurrieta, direccion=Gandia]
Persona [nombre=Ana, apellido=Balbastre, direccion=Xativa]
ORDEN comparaNombreApellidos:
Persona [nombre=Ana, apellido=Balbastre, direccion=Xativa]
Persona [nombre=Ana, apellido=Zurrieta, direccion=Gandia]
Persona [nombre=Juan, apellido=Zamora, direccion=Gandia]
La lista realmente no ha cambiado:
Persona [nombre=Juan, apellido=Zamora, direccion=Gandia]
Persona [nombre=Ana, apellido=Zurrieta, direccion=Gandia]
Persona [nombre=Ana, apellido=Balbastre, direccion=Xativa]
```



Streams: Map:

Es una operación intermedia que convierte cada elemento en otro objeto via una función. Posteriormente podemos aplicarle otra operación intermedia o una operación terminal:

```
StreamMapSortedApp.java X
StreamMapSortedApp.java > StreamMapSortedApp > main(String[])
1  import java.util.Comparator;
2  import java.util.List;
3  import java.util.ArrayList;
4
5
6  > class Persona implements Comparable<Persona> { ...
81
82  public class StreamMapSortedApp {
    Run | Debug
83  public static void main(String[] args) {
84      // A partir de Java 9 tenemos List.of
85      List<String> paises = new ArrayList<String>(List.of("España", "Francia"));
86      // convertimos a mayuscula e imprimimos que paises que empiezan por E
87      paises.stream().map(pais->pais.toUpperCase()).
88          filter(p->p.startsWith("E")).forEach(System.out::println);
89
90      //AHORA CON OBJETOS
91      Persona persona1 = new Persona("Juan", "Zamora","Gandia");
92      Persona persona2 = new Persona("Ana", "Zurrieta","Gandia");
93      Persona persona3 = new Persona("Ana", "Balbastre","Xativa");
94      List<Persona> lista= new ArrayList<Persona>(
95          List.of(persona1,persona2,persona3));
96      System.out.println("ORDEN Apellidos de las personas:");
97      lista.stream().map(p->p.getApellido()).sorted().forEach(System.out::println);
98  }
99  }
```

joseramon@Notebook-PC:~/Escritorio/2020.Simarro/SRV/UD2/ejemplos\$
ESPANA
ORDEN Apellidos de las personas:
Balbastre
Zamora
Zurrieta



Streams: Count:

Es una operación terminal que devuelve el número de elementos del Stream como un long:

```
StreamCountMapApp.java x
StreamCountMapApp.java > StreamCountMapApp > main(String[])
1 import java.util.Comparator;
2 import java.util.List;
3 import java.util.ArrayList;
4
5
6 > class Persona implements Comparable<Persona> { ...
81
82 public class StreamCountMapApp {
83     Run | Debug
84     public static void main(String[] args) {
85         // A partir de Java 9 tenemos List.of
86         List<String> paises = new ArrayList<String>(List.of("España", "Francia"));
87         // convertimos a mayuscula y contamos los paises que empiezan por E
88         System.out.println(
89             paises.stream().map(pais->pais.toUpperCase()).
90             filter(p->p.startsWith("E")).count()
91         );
92         //AHORA CON OBJETOS
93         Persona persona1 = new Persona("Juan", "Zamora","Gandia");
94         Persona persona2 = new Persona("Ana", "Zurrieta","Gandia");
95         Persona persona3 = new Persona("Ana", "Balbastre","Xativa");
96         List<Persona> lista= new ArrayList<Persona>(
97             List.of(persona1,persona2,persona3));
98         System.out.println("Cuántas personas viven en Gandia:");
99         System.out.println(lista.stream().map(p->p.getDireccion()).
100             filter(poblacion->poblacion.equals("Gandia")).count());
101     }
102 }
```



```
josemon@Notebook-PC:~/Escritorio/2020.Simarro/SRV/UD2/ejemplos$
1
Cuántas personas viven en Gandia:
2
```



Streams: Distinct:

Es una operación intermedia que permite eliminar los elementos repetidos:

```
StreamDistinctCountMapApp.java X
StreamDistinctCountMapApp.java > StreamDistinctCountMapApp > main(String[])
1  import java.util.List;
2  import java.util.ArrayList;
3
4
5 > class Persona implements Comparable<Persona> { ...
80
81  public class StreamDistinctCountMapApp {
      Run | Debug
82      public static void main(String[] args) {
83          Persona persona1 = new Persona("Juan", "Zamora","Gandia");
84          Persona persona2 = new Persona("Ana", "Zurrieta","Gandia");
85          Persona persona3 = new Persona("Ana", "Balbastre","Xativa");
86          List<Persona> lista= new ArrayList<Persona>(
87              List.of(persona1,persona2,persona3));
88          System.out.println("Cuántas poblaciones distintas hay:");
89          System.out.println(lista.stream().map(p->p.getDireccion()).distinct().count());
90      }
91  }
```



```
joseramon@Notebook-PC:~/Escritorio/2020.Simarro/SRV/UD2/ejemplos$ java StreamDistinctCountMapApp
Cuántas poblaciones distintas hay:
2
```




Streams: Reduce:

Es una operación terminal que reduce los elementos del stream utilizando una función:

```
StreamReduceMapApp.java X
StreamReduceMapApp.java > StreamReduceMapApp > main(String[])
1 import java.util.List;
2 import java.util.Optional;
3 import java.util.ArrayList;
4
5
6 > class Persona implements Comparable<Persona> { ...
81
82 public class StreamReduceMapApp {
83     Run | Debug
84     public static void main(String[] args) {
85         // A partir de Java 9 tenemos List.of
86         List<String> paises = new ArrayList<String>(List.of("Francia",
87             "Francia", "España", "España"));
88         // Lista de paises en mayuscula ordenados sin repetición
89         Optional<String> paisesConA=
90         paises.stream().map(String::toUpperCase).distinct().
91         sorted().reduce((s1, s2) -> s1 + "," + s2);
92         paisesConA.ifPresent(System.out::println);
93
94         //AHORA CON OBJETOS
95         Persona persona1 = new Persona("Ana", "Balbastre","Xativa");
96         Persona persona2 = new Persona("Juan", "Zamora","Gandia");
97         Persona persona3 = new Persona("Ana", "Zurrieta","Gandia");
98         List<Persona> lista= new ArrayList<Persona>(
99             List.of(persona1,persona2,persona3));
100         System.out.println("Lista de poblaciones ordenadas sin repetición:");
101         Optional<String> poblacionesSinRepeticion=
102         lista.stream().map(p->p.getDireccion()).
103         distinct().sorted().reduce((s1, s2) -> s1 + "," + s2);
104         poblacionesSinRepeticion.ifPresent(System.out::println);
105     }
106 }
```



```
joseramon@Notebook-PC:~/Escritorio/2020.Simarro/SRV/UD2/ejemplos$
ESPANA,FRANCIA
Lista de poblaciones ordenadas sin repetición:
Gandia,Xativa
```

En la siguiente diapositiva se explica mejor el concepto de 'Optional'.



Streams: FindFirst:

Es una operación terminal que devuelve un elemento de tipo Optional:

La clase 'Optional' nos evita los problemas ocasionados por valores nulos. Si no hay resultados en vez de devolver nulo, un objeto de tipo Optional devuelve 'false' al consultar su método "isPresent()" para avisar que no habían valores.

```
StreamFindFirstApp.java X
StreamFindFirstApp.java > ...
1 import java.util.List;
2 import java.util.Optional;
3 import java.util.ArrayList;
4
5 > class Persona implements Comparable<Persona> { ...
93
94 public class StreamFindFirstApp {
    Run | Debug
95     public static void main(String[] args) {
96         Persona persona1 = new Persona(111,"Juan", "Zamora","Gandia");
97         Persona persona2 = new Persona(222,"Ana", "Zurrieta","Gandia");
98         Persona persona3 = new Persona(333,"Ana", "Balbastre","Xativa");
99         List<Persona> lista= new ArrayList<Persona>()
100             List.of(persona1,persona2,persona3));
101
102         System.out.println("Buscar a la persona con dni '222':");
103         Optional<Persona> persona= lista.stream().
104             filter(p->p.getId().equals(222)).findFirst();
105         if (persona.isPresent())
106             System.out.println(persona.get().toString());
107         else
108             System.out.println("No se ha encontrado.");
109     }
110 }
```

→

```
joseramon@Notebook-PC:~/Escritorio/2020.Simarro/SRV/UD2/ejemplos$ java StreamFindFirstApp
Buscar a la persona con dni '222':
Persona [nombre=Ana, apellido=Zurrieta, direccion=Gandia]
```



UD 2: Modelo Vista Controlador

9.- Expresiones Lambda, Streams y Collectors



Desarrollo Web en Entorno Servidor - Joseramon.profesor@gmail.com

- **Streams: Collect:**

Stream.collect() es uno de los métodos de la API de Streams de Java 8 que permite reempaquetar los elementos a algún tipo de estructura de datos implementada en las clases del paquete **java.util.stream.Collectors**:

- **toList**
- **toMap**
- **toSet**



- Streams: Collect: `toList()` y `toSet()`
- Veamos los siguientes ejemplos:

```
StreamCollectorsApp.java X
StreamCollectorsApp.java > ...
1 import java.util.List;
2 import java.util.Set;
3 import java.util.ArrayList;
4 import java.util.stream.Collectors;
5
6 > class Persona implements Comparable<Persona> { ...
81
82 public class StreamCollectorsApp {
83     Run | Debug
84     public static void main(String[] args) {
85         Persona persona1 = new Persona("Juan", "Zamora", "Gandia");
86         Persona persona2 = new Persona("Ana", "Zurrieta", "Gandia");
87         Persona persona3 = new Persona("Ana", "Balbastre", "Xativa");
88         List<Persona> lista = new ArrayList<Persona>(
89             List.of(persona1, persona2, persona3));
89
90         System.out.println("Personas de Gandia:");
91         List<Persona> listaPersonasGandia = lista.stream().
92             filter(p -> p.getDireccion().equals("Gandia")).
93             collect(Collectors.toList());
94         listaPersonasGandia.stream().forEach(p -> System.out.println(p));
95
96         System.out.println("Poblaciones de las personas:");
97         Set<String> listaPoblaciones = lista.stream().
98             map(p -> p.getDireccion()).collect(Collectors.toSet());
99         listaPoblaciones.forEach(System.out::println);
100     }
101 }
```



```
joseramon@Notebook-PC:~/Escritorio/2020.Simarro/SRV/UD2/ejemplos$
Personas de Gandia:
Persona [nombre=Juan, apellido=Zamora, direccion=Gandia]
Persona [nombre=Ana, apellido=Zurrieta, direccion=Gandia]
Poblaciones de las personas:
Gandia
Xativa
```



Streams: Collect: toMap()

Cuando obtenemos un mapa, no podemos convertirlo a Stream directamente, por lo que obtenemos primero una colección de elementos con “values()” a los que si se puede aplicar Streams:

```
StreamCollectorsToMapApp.java X
StreamCollectorsToMapApp.java > StreamCollectorsToMapApp > main(String[])
1 import java.util.List;
2 import java.util.Map;
3 import java.util.ArrayList;
4 import java.util.stream.Collectors;
5
6 > class Persona implements Comparable<Persona> { ...
94
95 public class StreamCollectorsToMapApp {
    Run|Debug
96     public static void main(String[] args) {
97         Persona persona1 = new Persona(111,"Juan", "Zamora","Gandia");
98         Persona persona2 = new Persona(222,"Ana", "Zurrieta","Gandia");
99         Persona persona3 = new Persona(333,"Ana", "Balbastre","Xativa");
100         List<Persona> lista= new ArrayList<Persona>(
101             List.of(persona1,persona2,persona3));
102
103         System.out.println("Mapa Personas de Gandia:");
104         Map<Integer,Persona> mapaPersonasGandia=lista.stream().
105             filter(p->p.getDireccion().equals("Gandia")).
106             collect(Collectors.toMap(p->p.getId(), p->p));
107         //values sobre un mapa devuelve una colección de Personas
108         mapaPersonasGandia.values().stream().
109             forEach(p->System.out.println(p.toString()));
110     }
111 }
```



```
joseramon@Notebook-PC:~/Escritorio/2020.Simarro/SRV/UD2/ejemplos$
Mapa Personas de Gandia:
Persona [nombre=Ana, apellido=Zurrieta, direccion=Gandia]
Persona [nombre=Juan, apellido=Zamora, direccion=Gandia]
```




EJERCICIO 1:

Abre el fichero “EjerciciosStreamsApp.java” de AULES e implementa el código necesario para devolver los resultados solicitados. **Puedes realizar este ejercicio desde VSCode:**

```
public class EjerciciosStreamsApp {  
    Run | Debug  
    public static void main(String[] args) {  
        Persona persona1 = new Persona(111, "Juan", "Zamora", "Onteniente");  
        Persona persona2 = new Persona(222, "Ana", "Zurrieta", "Gandia");  
        Persona persona3 = new Persona(333, "Ana", "Balbastre", "Xativa");  
        Persona persona4 = new Persona(333, "Pedro", "Gimenez", "Gandia");  
        List<Persona> lista = new ArrayList<Persona>(  
            List.of(persona1, persona2, persona3, persona4));  
  
        System.out.println("Lista de nombres no repetidos que contiene una 'A' ordenados:");  
        //deberá devolver:  
        //Ana  
        //Juan  
  
        System.out.println("Lista de apellidos no repetidos ordenados inversamente en una sola linea:");  
        //deberá devolver:  
        //Zurrieta,Zamora,Gimenez,Balbastre  
  
        System.out.println("Nombre y apellidos de la primera persona de Gandia si lo ordenamos por apellidos:");  
        //Deberá devolver:  
        //Pedro Gimenez  
    }  
}
```




UD 2: Modelo Vista Controlador

9.- Expresiones Lambda, Streams y Collectors



Desarrollo Web en Entorno Servidor - Joseramon.profesor@gmail.com

EJERCICIO 2:

Modifica en AlumnoService.java el método que encuentra un alumno utilizando el dni para utilizar expresiones lambda al buscar el alumno:
public Alumno encontrarAlumnoPorDni(String dni)

Implementa los criterios de ordenación del listado de alumnos y módulos.

Implementa un buscador de Alumnos que permita buscar por el valor exacto del "dni", la "edad", el "ciclo", el "curso", el "horario" o el "pais" del alumno:

Ayuda: Create una clase nueva FiltroAlumno.java en el paquete modelo que contenga "campo" y "valor".

	Nombre	Edad	Ciclo	Curso	Erasmus	Acciones
22222222B	Pedro	32	DAW	2	<input type="checkbox"/>	<button>Modificar</button>
33333333C	Juan	23	ASIR	1	<input type="checkbox"/>	<button>Modificar</button>
11111111A	JoseRa	21	DAM	1	<input type="checkbox"/>	<button>Modificar</button>

Añadir alumno

DWES: Desarrollo Web en Entorno Servidor - profesor: joseramon.profesor@gmail.com



UD 2: Modelo Vista Controlador

9.- Expresiones Lambda, Streams y Collectors

Desarrollo Web en Entorno Servidor - Joseramon.profesor@gmail.com



... continuación EJERCICIO 2:

Sube la aplicación final al moodle.

Para ello:

1º Haz un “Run As \Maven Clean” para dejar solo los fichero fuentes y quitar momentaneamente los necesarios para ejecutar la aplicación (dependencias).

2º Comprime la carpeta de tu aplicación y ponle como nombre al fichero comprimido UD2_practica5_nombreAlumno.tar.gz donde nombreAlumno es el nombre del alumno que entrega la práctica.

3º Súbela al moodle.

IMPORTANTE: No comprimir en RAR, porque Ubuntu no lo lee bien y en clase tenemos Ubuntu. Si tuviesemos Windows, podemos comprimir en ZIP.