Bloque 4

# Interfaces Funcionales y el Tipo Stream

Fundamentos de Programación
Departamento de Lenguajes y Sistemas Informáticos

2023/2024

# Interfaces Funcionales y el tipo Stream

## *Objetivos* de este bloque 4 (Interfaces Funcionales y tipo Stream)

- Hemos aprendido hasta ahora, a grandes rasgos, lo siguiente:
  - *Diseñar* tipos (mediante clases y record)
  - *Manejar* objetos de los tipos diseñados.
  - *Diseñar* un tipo de objetos que denominamos "*contenedores*" (que permiten almacenar colecciones).
  - *Leer ficheros* cuyos registros responde a determinada estructura de un tipo de objeto (mediante "parseos" pasamos de una línea de texto a un objeto) y cada "objeto parseado" se añade a tipos contenedores.
  - Mediante recorridos secuenciales se realizan "analíticas de datos" sobre objetos contenido en las colecciones.
- *Se trata ahora de realizar las "analíticas de datos" mediante el tipo Stream*

# Tipo Stream

2023/24

## Definición

El tipo *Stream<T>* permite trabajar de una forma *cómoda y simple* las colecciones de objetos para realizar "analítica de datos".

Collection<*Vuelo*>

[v1,v2,v3,…,$v_n$]

stream()

Stream<*Vuelo*>

v1 v2 v3 … $v_n$

### *Constructores:*

Stream<T> *stream*()
Stream<T> *of* (obj1, obj2,…)
Stream<T> *concat* (Stream s1, Stream s2)

### *Operaciones intermedias:*

*Stream<T> limit (long n)*
*Stream<T> skip (long n)*
*Stream<T> distinct ()*
*Stream<T> filter (Predicate<T>)*
Stream<R> *map (Function<T, R>)*
*Stream<T> sorted*()
*Stream<T> sorted* (Comparator<T>)

### *Operaciones terminales:*

*boolean allMatch (Predicate<T>)*
*boolean anyMatch (Predicate<T>)*
*long count()*
*long sum()*
*OptionalDouble average()*
*T max(Comparator<T>)*
*T min(Comparator<T>)*
*void forEach(Consumer<T>)*

### *Operación Inversa:*

*T collect (Collectors)*

# Tipo Stream

2023/24

## Ejemplo de máximo:

### Con recorrido secuencial

```
public Vuelo vueloMásDuración () {
  Vuelo res=null;
  for (Vuelo v:this.vuelos()) {
    if (res==null || v.duración().compareTo(res.duración())>0) {
      res=v;
    }
  }
  return res;
}
```

### Con Stream

```
public Vuelo vueloMásDuración () {
  return this.vuelos().stream()
             .max(Comparator.comparing(Vuelo::duración)).orElse(null);
}
```

# Tipo Stream

2023/24

## Ejemplo de obtención de un mapa:

### Con recorrido secuencial

```
public Map<Compañía, Set<String>> distintosDestinosPorCompañía (){
  Map<Compañía, Set<String>> res=new HashMap<Compañía, Set<String>>();
  for (Vuelo v:this.vuelos()) {
    if (!res.containsKey(v.compañía())){
      Set<String> aux=new HashSet<String>();
      aux.add(v.destino());
      res.put(v.compañía(),aux);
    }
  else {
      Set<String> aux=res.get(v.compañía());
      aux.add(v.destino());
    res.put(v.compañía(),aux);
    }
  }
  return res;
}
```

# Tipo Stream

2023/24

## Ejemplo de obtención de un mapa:

### Con Stream

```
public Map<Compañía, Set<String>> distintosDestinosPorCompañía (){
  return this.vuelos().stream()
        .collect(Collectors.groupingBy(Vuelo::compañía,
         Collectors.mapping(Vuelo::destino,Collectors.toSet())));
}
```

# Tipo Stream

2023/24

## Constructores:

Stream<T> stream()
Stream<T> of (obj1, obj2,…)
Stream<T> concat (Stream s1, Stream s2)

## Operaciones intermedias:

Stream<T> limit (long n)
Stream<T> skip (long n)
Stream<T> distinct ()
Stream<T> filter (Predicate<T>)
Stream<R> map (Function<T, R>):
    Stream<Integer> mapToInt(Function<T, Integer>)
    Stream<Long> mapToLong (Function<T, Long>)
    Stream<Double> mapToDouble (Function<T, Double>)
Stream<T> sorted()
Stream<T> sorted (Comparator<T>)

## Operaciones terminales:

boolean allMatch (Predicate<T>)
boolean anyMatch (Predicate<T>)
long count()
long sum()
OptionalDouble average()
T max(Comparator<T>)
T min(Comparator<T>)
void forEach(Consumer<T>)

## Operación Inversa:

T collect (Collectors)

2023/24

# Interfases Funcionales y el tipo Stream

## Índice

- Expresiones *Lamba*
- Expresiones *Referencia a Métodos*
- *Interfaces funcionales*
  - Comparator<T>
  - Suplier<T>
  - Consumer<T>
  - Function<T,R>
  - BiFunction<T,R,S>
  - Predicate<T>
- Tipo *Stream*<T>

2023/24

# Expresiones Lambda

Las expresiones *lambda*, es una *forma cómoda y breve* de expresar una función de manera formal en determinados lenguajes.

Estas expresiones *se utilizan como argumento de métodos del tipo Stream*

A partir de un número determinado de parámetros de entrada, que se escriben entre paréntesis y separados por coma (,), se obtendrá como salida, el resultado de aplicar una o varias operaciones a dichos parámetros. En la sintaxis se separa la entrada (los parámetros), de la salida (las operaciones) por el operador flecha **->**.

*Sintaxis*:

- () **->** expresión
- (parámetro/s) **->** expresión
- (parámetro/s) **->** **{**sentencia/s;**}** (*cuando son sentencias se encierran en un bloque*)

# Expresiones Lambda

*Ejemplos*: Expresiones lambda que:

- Suma dos valores: *(x,y)->x+y*
- Eleva al cubo un valor *x->Math.pow(x,3)* -Observar que cuando hay un único parámetro se puede omitir los paréntesis-.
- Multiplica un valor por la suma de los otros dos: *(x,y,z)->x*(y+z)*
- Obtiene la duración de una Canción: *c->c.getDuracion()*
- Obtiene una expresión trigonométrica: *(x,y)->Math.sin(x*x)+Math.cos(y)*
- Ejecutar una sentencia: *(x,y)->{System.out.println(x+y);}* -Observar que se encierran entre llaves y termina en ; -.
- Crear una lista de enteros: *( )->new ArrayList<Integer>()* -Observar que cuando la operación no depende de ningún parámetro sólo se abre y cierran los paréntesis -.

# Expresiones Lambda

*Ejercicio*: Expresiones lambda que:

- Obtenga la hora de llegada de un vuelo (designado por v):
- El número de plazas libres de un vuelo (designado por vuelo)
- El porcentaje de una cantidad "c" sobre un total "t"
- Crear un SortedMap cuyas claves esté ordenadas por el orden natural del tipo Familia, en que a cada Familia de animales le asocie el promedio de las edades medias


# Expresiones Lambda

*Ejercicio*: Expresiones lambda que:

- Obtenga la hora de llegada de un vuelo (designado por v):

  *v->v.fechaHoraLlegada().toLocalTime()*

- El número de plazas libres de un vuelo (designado por vuelo)

  *vuelo->vuelo.númeroPlazas()-vuelo.númeroPasajeros()*

- El porcentaje de una cantidad "c" sobre un total "t"

  *(c,t)->100.0*c/t*

- Crear un SortedMap cuyas claves esté ordenadas por el orden natural del tipo Familia, en que a cada Familia de animales le asocie el promedio de las edades medias

  *( )->new TreeMap<Familia,Double>()*

2023/24

# Interfases Funcionales y el tipo Stream

## Índice

- Expresiones *Lamba*
- Expresiones *Referencia a Métodos*
- *Interfaces funcionales*
  - Comparator<T>
  - Suplier<T>
  - Consumer<T>
  - Function<T,R>
  - BiFunction<T,R,S>
  - Predicate<T>
- Tipo *Stream*<T>


# Expresiones Referencia a Método

Hemos visto como ejemplo "Obtener la duración de una Canción, mediante la expresión lambda: *c->c.getDuracion()*"

Cuando las expresiones lambda hacen llamadas a métodos que no requieren de paso de parámetros y no hay concatenación de métodos, pueden ser sustituidas por expresiones denominadas de "*Referencia a Método*"

En estas expresiones de *Referencia a Método*:

- Se sustituye el operador fecha (**->**) por **::**
- A la izquierda el nombre del Tipo donde se implementa el método
- A la derecha el nombre del método. **Observar** que se dice "nombre" del método, sin los paréntesis.

De esta forma la *expresión lambda*: *c->c.getDuracion()*

Se puede escribir como *referencia a método: Canción::getDuración*

2023/24

# Interfases Funcionales y el tipo Stream

## Índice

- Expresiones *Lamba*
- Expresiones *Referencia a Métodos*
- *Interfaces funcionales*
  - Comparator<T>
  - Suplier<T>
  - Consumer<T>
  - Function<T,R>
  - BiFunction<T,R,S>
  - Predicate<T>
- Tipo *Stream*<T>


# Interfaces Funcionales

Una interfaz funcional es una interfaz que contiene:

- *Al menos un método estático*: se invocan utilizando el nombre de la Interfaz y se programa dentro de la/s clase/s, que implementen la interfaz.
- *Métodos por defecto también denominados abstractos*, que necesitan de un objeto para invocarlos y que se programan en la propia interfaz (no se programa dentro de una clase).

No profundizamos en ello, *lo más importante de las interfaces funcionales es que sus métodos recibirán funciones como parámetros*.

*Estas funciones* serán: *expresiones lambda* o *referencia a método*

2023/24

# Interfases Funcionales y el tipo Stream

## Índice

- Expresiones *Lamba*
- Expresiones *Referencia a Métodos*
- *Interfaces funcionales*
  - Comparator<T>
  - Suplier<T>
  - Consumer<T>
  - Function<T,R>
  - BiFunction<T,R,S>
  - Predicate<T>
- Tipo *Stream*<T>

2023/24

# Interfaz Comparator

Definición: Hemos visto que cuando creamos un Tipo (por ejemplo, el tipo Vuelo), normalmente, se establece un criterio de orden natural para el tipo que se implementaba programando el método ***compareTo*** *(en nuestro tipo Vuelo se ha establecido como orden natural el del código y a igualdad de código, desempatan por el destino y, por último, si siguen empatando, por la fecha de salida*)

No obstante, sobre el mismo Tipo se pueden establecer otros criterios de ordenación alternativos. Estos otros criterios se implementarán mediante la *Interfaz funcional Comparator*.

2023/24

# Interfaz Comparator

## Definición de la Interfaz Comparator

```
public interface Comparator<T> {

int compare(T o1, T o2);

static <T,U extends Comparable<? super U>> Comparator<T>
comparing(Function<? super T,? extends U> keyExtractor);

static <T extends Comparable<? super T>> Comparator<T>
naturalOrder();

default Comparator<T> reversed();

static <T extends Comparable<? super T>> Comparator<T>
reverseOrder();

default Comparator<T> thenComparing(Comparator<? super T> other);

  . . .

}
```

método no estático ni por defecto que se programa en la clase que implemente la interfaz

# Interfaz Comparator

## Definición

- *compare*: es el método estático que se programaría en una clase que implemente la interfaz.
- *comparing*: Permite construir un comparador.
- *reversed*: Aplicado a un comparador ya creado invierte su ordenación.
- *thenComparing*: Aplicado a un comparador permite establecer un siguiente criterio de desempate.
- *naturalOrder*: Construye un comparador por el orden natural del tipo al que se aplica.
- *reverseOrder*: Construye un comparador por el orden natural inverso del tipo al que se aplica.


# Interfaz Comparator

*Ejemplo*:

*Crear* un comparador que ordene los vuelos por la fecha y hora de salida (con *lambda* expresión y con *referencia a método*):

- `Comparator<Vuelo>cmp=Comparator.comparing(v->v.getFechaSalida())`
- `Comparator<Vuelo>cmp=Comparator.comparing(Vuelo::getFechaSalida)`

Para *usarlo*:

Ordenar una lista de vuelos `lVuelos` por la fecha y hora de salida:

```
-Collections.sort(lVuelos,cmp);
```

Crear un SortedSet `cVuelos` en el que los vuelos se recorran por la fecha y hora de salida:

```
-SortedSet <Vuelo> cVuelos=new TreeSet<Vuelo>(cmp);
```

# Interfaz Comparator

*Ejemplo*:

Crear un comparador que ordene los vuelos por la fecha de salida y a igualdad de fecha de salida por el destino, con lambda expresión y con referencia a método):

- `Comparator<Vuelo>cmp1= Comparator.comparing(v->v.getFechaSalida().toLocalDate())`
- `Comparator<Vuelo>cmp2=cmp1.thenComparing(Vuelo::destino)`
- `Comparator<Vuelo>cmp2=cmp1.thenComparing(v->v.destino())`

O hacerlo directamente de una sola vez (*se pueden mezclar las expresiones*)

- `Comparator<Vuelo>cmp2= Comparator.comparing (v->v.getFechaSalida().toLocalDate()).thenComparing(Vuelo::destino)`
- ~~`Comparator<Vuelo>cmp2= Comparator.comparing (Vuelo::getFecha.toLocalDate). thenComparing(v->v.destino())`~~

Para usarlo:

```
Collections.sort(lVuelos,cmp2);
SortedSet <Vuelo> cVuelos=new TreeSet<Vuelo>(cmp2);
```

2023/24

# Interfaz Comparator

*Ejemplo*:

Crear un comparador que ordene los vuelos por el número de pasajeros al revés (de mayor a menor número de pasajeros):

- ```
  Comparator<Vuelo> cmp=
      Comparator.comparing(Vuelo::númeroPasajeros).reversed();
  ```

Crear un comparador que ordene los vuelos por el destino y desempaten por el código.

```
Comparator<Vuelo> cmp= Comparator.comparing(Vuelo::destino)
            .thenComparing(v->v.código());
Comparator<Vuelo> cmp= Comparator.comparing(Vuelo::destino)
            .thenComparing(Vuelo::código);
Comparator<Vuelo> cmp= Comparator.comparing(v->v.destino())
            .thenComparing(v->v.código());
Comparator<Vuelo> cmp= Comparator.comparing(v->v.destino())
            .thenComparing Vuelo::código);
```
```

# Interfaz Comparator

*Ejemplo*:

Crear un comparador que ordene los vuelos por el destino y, a igualdad de destino, desempaten por el orden inverso del código.

No vale:

```
Comparator<Vuelo> cmp= Comparator.comparing(Vuelo::destino)
                .thenComparing(Vuelo::código).reversed();
```

Porque ordena por el destino y desempata por el código, *pero al final lo invierte todo*.

Hay que crear ***comparadores independientes*** y unirlos.

```
Comparator<Vuelo> cmp1= Comparator.comparing(Vuelo::destino);
Comparator<Vuelo> cmp2=Comparator.comparing(Vuelo::código).reversed();
Comparator<Vuelo> cmp=cmp1.thenComparing(cmp2);
```

2023/24

# Ejercicio Aeropuerto

- Realice los ejercicios del ***EnunciadoAeropuerto06***

  *Apartado 17*