Interfaces Funcionales

1. Introducción 1
2. Justificación 2
3. Lambda Expresiones 4
4. Predicate<T> 5
5. BiPredicate<T,U> 6
6. Function<T,R> 6
7. BiFunction<T,U,R> 8
8. Consumer<T> 8
9. BiConsumer<T,U> 9
10. Supplier<T> 9
11. UnaryOperator<T> 9
12. BinaryOperator<T> 10

1. Introducción

Las interfaces funcionales son un potente mecanismo que proporciona Java 8 para poder pasar funciones como argumentos a los métodos. Realmente esta posibilidad ya estaba presente en versiones anteriores de Java por ejemplo con la interfaz Comparator.

¿Qué es una interfaz funcional? Una interfaz funcional define "objetos" que no guardan valores como los objetos tradicionales sino que sirven para guardar "funciones". Nótese que tanto "objetos" como "funciones" están entrecomillados porque realmente no son tales pero sí es un mecanismo para que un método reciba argumentos de tipo funcional. Volviendo al caso de la interfaz Comparator, ¿Qué se está definiendo con un Comparator? Se está definiendo un criterio de ordenación, de forma que el método compare nos dice dados dos objetos cuál es menor de los dos. Si se pasa un objeto de tipo Comparator como argumento a un método, estamos proporcionando a ese método una función tal que dados dos objetos de un tipo devuelve un entero que nos indica cuál es el orden de los dos objetos. El método es "genérico" con respecto a ese orden y debe estar preparado para recibir cualquier criterio y desarrollar su funcionalidad de acuerdo al orden que tiene como argumento de entrada.

De esta manera, por ejemplo, el método sort de la clase Collections, puede recibir un argumento que es un objeto de tipo Comparator. Previamente se debe haber definido una clase de tipo Comparator que implemente un método compare para definir el orden de los objetos de la clase a comparar.

Ejemplo. La clase Comparator que ordena objetos de tipo Vuelo por precio de menor a mayor es:

```
public class ComparadorVueloPrecio implements Comparator<Vuelo> {

	public int compare(Vuelo v1, Vuelo v2){
		return v1.getPrecio().compareTo(v2.getPrecio());
	}
}
```

Y la invocación podría ser:

```
Collections.sort(listaVuelos, new ComparadorVueloPrecio());
```

Como se puede observar el método sort recibe un argumento de tipo funcional, ya que el objeto de tipo Comparator que recibe le sirve a sort para saber cómo debe ordenar los valores de listaVuelos. El código del método sort de la clase Collections lógicamente debe estar preparado para ordenar por distintos criterios. Para ello tiene el método sort entre sus líneas una invocación genérica al método compare del objeto que recibe como argumento. De esta manera si recibe un Comparator para ordenar por precio, ordenara por precios la lista de Vuelos, pero si recibe un Comparator que ordene por número de pasajeros, este será el criterio de ordenación.

Java 8 lo que hace es extender el número de interfaces funcionales y su aplicabilidad definiendo un conjunto de métodos cuyos argumentos de entrada son estas interfaces.

2. Justificación

Estudiemos otra posible aplicación de las interfaces funcionales distinta del Comparator. Hay numerosos algoritmos que usan una condición booleana en su esquema. Uno de los más simples es el patrón contador, algoritmo que nos devuelve el número de elementos de una colección que cumplen una determinada condición: ¿Cuántos vuelos completos salen hoy? ¿Cuántos vuelos hay a Madrid esta semana? etc. Sabemos que el esquema de este algoritmo es el siguiente:

```
Esquema contador
        Para todo elemento de la colección
                Si cumple condición
                        Incrementa contador
                Finsi
        FinPara
}
```

Este esquema tiene como entrada la colección de elementos y la condición que deben cumplir y como salida el contador. Hasta ahora si queremos hacer un método para implementar un contador, éste debía recibir los valores necesarios para implementar la condición. Por ejemplo, veamos el código de los métodos de Aeropuerto que cuentan el número de vuelos a un destino y el número de vuelos a partir de una determinada fecha.

```
public Integer contadorVuelosFecha(Fecha f){
        Integer contador=0;
        for(Vuelo v:vuelos){
                if (v.getFecha().compareTo(f)>0){
                        contador++;
                }
        }
        return contador;
}

public Integer contadorVuelosDestino(String d){
        Integer contador=0;
        for(Vuelo v:vuelos){
                if (v.getDestino().equals(d)){
                        contador++;
```

```
                }
            }
            return contador;
        }
```

Como se puede observar el código es exactamente el mismo salvo la condición del if y lógicamente los argumentos necesarios para su codificación: la Fecha f en el primer caso y el destino d en el segundo. Este código se invocaría:

```
Integer cont1 = aeropuertoSVQ.contadorVuelosDestino("Madrid");
System.out.println("El número de vuelos a Madrid es "+cont1);
Integer cont2 = aeropuertoSVQ.contadorVuelosFecha(new Fecha(16,07,2014));
System.out.println("El número de vuelos a partir del 16 de julio es "+cont2);
```

Supongamos que se pudiera pasar la condición del if como argumento al método. Entonces el método contador podría generalizarse y su código sería algo así como:

```
    public Integer contadorVuelosGenerico(Condicion<Vuelo> filtro[1]){
        Integer contador=0;
        For (Vuelo v:vuelos){
            if (condición de filtro sobre v[2]){
                contador++;
            }
        }
        return contador;
    }
```

De esta manera, tendríamos un método contador genérico para Aeropuerto que una vez codificado, podría invocarse con distintas expresiones booleanas y tener diferentes funcionalidades. Para poder pasar una expresión booleana como argumento necesitaríamos un tipo (una interfaz) Condicion que implementara un método que recibiera un objeto del tipo implicado (Vuelo en este caso) y devolviera un valor de tipo Boolean con la condición requerida. Esta posibilidad es la interfaz funcional Predicate que será el primer ejemplo que se estudiará en la siguiente sección.

La invocación de este método genérico sería:

```
Integer cont1 = aeropuertoSVQ.contadorVueloGenerico
                                        (v->v.getDestino.equals("Madrid")[3]);
System.out.println("El número de vuelos a Madrid es "+cont1);

Fecha f = new Fecha(16,07,2014)
Integer cont2 = aeropuertoSVQ.contadorVueloGenerico
                                        (v->v.getFecha().compareTo(f)>0[4]);
System.out.println("El número de vuelos a partir del 16 de julio es "+cont2);
```

[1] Con posterioridad este supuesto tipo Condicion será realmente el tipo Predicate
[2] Esta expresión será realmente filtro.test(v), ya que test es el método que implementa Predicate
[3] Esta es una lambda expresión (ver siguiente apartado) que indica que la "condición" o "filtro" que se pasa como argumento indica que cada Vuelo v devuelve una condición sobre si su destino es Madrid
[4] Esta es otra lambda expresión que para cada Vuelo v devuelve cierto si la fecha de v es posterior al 14 de julio

3. Lambda Expresiones

Hay otra cuestión que también cambia en Java 8 y es la forma de proporcionar la interfaz funcional como argumento en la invocación de un método. Como hemos visto en el ejemplo anterior para usar un Comparator en Java 7, se define una clase externa con el método compare y la invocación se realiza con un objeto de la clase que o bien se crea antes o se construye directamente en la invocación. Java 8 tiene dos mecanismos más flexibles para definir las interfaces funcionales: las lambda expresiones y las referencias a métodos.

Una lambda expresión es una simplificación de un método en el que los parámetros de entrada y la expresión de salida se separan por un operador flecha '->'. Los argumentos de entrada se escriben entre paréntesis separados por comas. Si la interfaz sólo tiene un argumento de entrada no es necesario poner los paréntesis. Así el principio de una lambda expresión es de la forma () -> si no hay argumento de entrada, x-> si sólo tiene un argumento o (x, y) -> para dos argumentos. Normalmente no es necesario definir los tipos de los argumentos porque Java es capaz de deducirlos del contexto en el que se definen. A continuación del operador -> se debe escribir la expresión que será el valor que devuelve la interfaz que se está declarando.

Ejemplo 1. Una interfaz funcional que reciba un Vuelo y devuelva su precio sería:

```
x-> x.getPrecio();
```

Ejemplo 2. Una interfaz funcional que reciba un String representado un número entero y devuelva un Integer con el valor correspondiente, se escribe:

```
x -> new Integer(x);
```

Ejemplo 3. Pongamos un ejemplo de nuevo con la interfaz Comparator. La invocación al método sort de Collections se puede hacer construyendo el Comparator mediante una lambda expresión directamente en el mismo código de la llamada:

```
Collections.sort(listaVuelos,(x,y)->x.getPrecio().compareTo(y.getPrecio()));
```

La lambda expresión está formado por los argumentos, en este caso dos: x e y encerrados entre paréntesis y separados por una coma. Los argumentos son dos porque el método compare de Comparator tiene dos argumentos. Como se puede observar x e y son referencias de tipo Vuelo aunque formalmente no hace falta ponerlo porque el compilador de Java es capaz de "comprenderlo" por el contexto, ya que si estamos ordenando un List<Vuelo>, el Comparator debe ser de Vuelo y por tanto los argumentos del método compare han de ser de tipo Vuelo. A continuación y después del símbolo flecha '->' se escribe la expresión que debe devolver el método compare. En nuestro caso, una expresión de tipo int con la comparación de los precios de los vuelos.

Otra forma de referenciar al Comparator es invocando al método que devuelve la propiedad por la que queremos comparar. Por ejemplo Java 8 permite esta otra invocación:

```
Collections.sort(vuelos,Comparator.comparing(Vuelo::getPrecio));
```

En esta segunda invocación la interfaz Comparator invoca al método estático comparing que tiene como argumento una interfaz funcional de tipo Function que se puede definir simplemente haciendo referencia al método que devuelve la propiedad por la que queremos comparar.

Las lambda expresiones suelen usarse directamente en la llamada del método que tiene como argumento una interfaz funcional, pero si una misma lambda expresión se va a usar en varias ocasiones, se puede declarar mediante un identificador.

Ejemplo. Para definir si un Vuelo está completo escribimos:

```
Predicate<Vuelo> vueloCompleto = x->
                                 x.getNumPasajeros().equals(x.getNumPlazas());
```

De esta forma, el identificador vueloCompleto puede sustituir a una interfaz Predicate en todas aquellas invocaciones a métodos que tengan un argumento de tipo Predicate.

Si la interfaz funcional va a tener argumentos de entrada distintos de los propios de la interfaz a la que define entonces la mejor manera es definirla como si de un método se tratara.

Ejemplo. Si se necesita definir una interfaz Predicate para Vuelo que reciba un argumento de tipo Fecha para preguntar si un Vuelo es posterior a una determinada fecha, se define:

```
Predicate<Vuelo> fechaIgual(Fecha f){
      return x -> x.getFecha().compareTo(f)>0;
}
```

4. Predicate<T>

Como se ha señalado anteriormente la interfaz Predicate implementa una condición lógica para métodos que necesiten un filtro o condición. Predicate implementa un método denominado **test** que devuelve un boolean a partir de un objeto de tipo T. Por tanto, el tipo Predicate se usa para clasificar los objetos de un tipo T según cumplan o no una determinada propiedad. Por ejemplo, dado un Vuelo si está completo, dado un Libro si tiene su título contiene una determinada palabra, dada una Canción si dura más de x minutos, dada un String si comienza por un determinado carácter.

Ejemplo 1. El Predicate que devuelve si un Vuelo está completo es:

```
Predicate<Vuelo> vueloCompleto = x->
                                 x.getNumPasajeros().equals(x.getNumPlazas());
```

Ejemplo 2. Si se necesita definir una condición a partir de un parámetro se le puede poner un argumento a la interfaz funcional. Por ejemplo, si necesitamos preguntar si un Vuelo es de una determinada fecha, se define:

```
Predicate<Vuelo> fechaIgual(Fecha f){
      return x -> x.getFecha().equals(f);
}
```

Existen también las interfaces especializadas DoublePredicate, IntPredicate y LongPredicate para obtener un valor lógico a partir de objetos de tipos básicos.

Métodos default.

La interfaz Predicate tiene tres métodos que implementan las operaciones lógicas: negate(), and(Predicate) y or(Predicate). Por ejemplo, si se necesita un argumento de tipo Predicate que diga si un Vuelo de una determinada fecha f está completo se escribiría:

```
fechaIgual(f).and(vueloCompleto)
```

5. BiPredicate<T,U>

La interfaz BiPredicate obtiene un valor lógico a partir de dos argumentos de distinto tipo. Por ejemplo, dado un String representando un destino y un Vuelo devuelve si el vuelo tiene ese destino, dada una canción y una duración devuelve si la duración de la canción es menor de la proporcionada, etc.

Ejemplo. La interfaz que devuelve si un Vuelo sale en una determinada fecha sería:

```
BiPredicate<Vuelo, Fecha> getCoincidencia = (x,y)->
                                            y.equals(x.getFecha());
```

6. Function<T,R>

Function es una interfaz con un método **apply** que recibe un argumento de tipo T y devuelve un objeto de tipo R. Se usa principalmente para transformar objetos de un tipo en alguna propiedad derivada u operación entre ellas. Por ejemplo, el autor de un Libro, la duración de una Cancion, la recaudación de un Vuelo, etc. Java 8 proporciona un conjunto de interfaces especializadas según el tipo del argumento de entrada o de salida. Por ejemplo, ToDoubleFunction<T>, ToIntFunction<T>, ToLongFunction<T> están especializadas en recibir un objeto de algún tipo T y devolver el tipo especificado en el nombre de la interfaz. Estas interfaces implementan un método denominado **applyAsX** donde X equivale a Double, Int o Double repsectivamente. De forma recíproca las interfaces LongFunction<R>, IntFunction<R> y DoubleFunction<R> reciben un valor del tipo especificado en el nombre y devuelven un objeto de tipo R, mediante un método **apply**. Finalmente existen seis interfaces más de nombre XToYFunction donde X, Y toman los valores Double, Int o Long, haciendo X el tipo del

argumento de entrada e Y el de salida. El método que implementan es ApplyAsY, donde Y es el tipo del argumento de salida.

Ejemplo 1: Dado un Vuelo la Function que dado un Vuelo devuelve su duración se puede definir:

```
Function<Vuelo, Duracion> functionDuracion = x->x.getDuracion();
```

En este caso si el tipo Vuelo tiene definido el método getDuracion se puede usar el operador :: en la invocación del método que use el tipo Function como argumento de entrada:

```
Vuelo::getDuracion
```

Por supuesto, si la expresión de la Function no se va a usar más de una vez, la lambda expresión x->x.getDuracion() puede ser argumento de entrada en los métodos que necesiten una Function.

Ejemplo 2. Dado un Vuelo la Function que devuelve el ratio de ocupación del mismo es:

```
Function<Vuelo,Double>functOcupa=x->1.*x.getNumPasajeros()/x.getNumPlazas();
```

Este caso es un claro ejemplo de función especializada:

```
ToDoubleFunction<Vuelo> functOcupa(){
      return x->1.*x.getNumPasajeros()/x.getNumPlazas();
}
```

Métodos default

La interfaz Function tiene dos métodos que permiten operar funciones mediante la composición: compose(Function) y andThen(compose). La diferencia entre ambos es el orden de aplicación de las funciones que intervienen. La función resultado de la aplicación del método f.compose(g) es primero aplicar g y después f, mientras que f.andThen(g) es el resultado de primero aplicar f y después g.

Ejemplo. Supongamos tenemos una función que dado un objeto de tipo Duracion devuelve su conversión a minutos:

```
Function<Duracion,Integer> enMinutos=x->x.getMinutos()+x.getHoras()*60;
```

Y otra función que devuelve la duración de un vuelo:

```
Function<Vuelo,Duracion> getDuracion = Vuelo::getDuracion;
```

Entonces la función que devuelve la duración en minutos de un Vuelo es:

```
Function<Vuelo,Integer> getDuracionEnMinutos=enMinutos.compose(getDuracion);
```

O también:

```
Function<Vuelo,Integer> getDuracionEnMinutos=getDuracion.andThen(enMinutos);
```

7. BiFunction<T,U,R>

BiFunction es una función que recibe dos argumentos de tipo T y U devuelve un argumento de tipo R, mediante un método que se denomina **apply**. Existen también tres interfaces especializadas para devolver un tipo determinado: ToDoubleBiFunction, ToIntBiFunction y ToLongBiFunction, que implementan un método **applyAsX** donde X puede ser Double, Int o Long.

Ejemplo. Para obtener una función que dados una Fecha y un Vuelo devuelva el número de días que faltan desde la fecha hasta que salga el vuelo es:

```
ToIntBiFunction<Vuelo,Fecha> getdias(Vuelo v, Fecha f){
	return (x,y)->y.resta(x.getFecha());
}
```

8. Consumer<T>

La interfaz Consumer es una variante de Function en la que no se devuelve valor, es decir modifica el objeto que invoca mediante un método que se denomina **accept** que recibe un objeto de tipo T y devuelve void. Se usan para definir una acción sobre un objeto. Por ejemplo, incrementa el precio de un Vuelo en un determinado porcentaje, resta a una Fecha un número de días o imprime en la consola un valor. Java 8 proporciona también las interfaces especializadas DoubleConsumer, LongConsumer o IntConsumer también implementando el método **accept**.

Ejemplo 1. Si queremos incrementar el precio de un Vuelo un 10%, definiríamos un Consumer:

```
Consumer<Vuelo> incrementaPrecio10p =  x->x.setPrecio(x.getPrecio()*1.1);
```

Ejemplo 2. Si quisiéramos que el incremento fuera de un determinado porcentaje pasado como argumento, podríamos escribir el siguiente método para el tipo Vuelo:

```
Consumer<Vuelo> incrementaPrecio(Double p){
	return x->x.setPrecio(x.getPrecio()*(1+p/100.));
}
```

Ejemplo 3. Es muy usual encontrarse el siguiente Consumer para sustituir a la expresión System.out.println:

```
Consumer<Vuelo> imprimeVuelo = x->System.out.println(x);
```

Ejemplo 4. Si quisiéramos tener un método de Vuelo que implementara alguna acción sobre un objeto de tipo Vuelo dependiendo de alguna condición, podríamos escribir:

```
public void aplicaAccion(Predicate<Vuelo> cond, Consumer<Vuelo> acc){
	if (cond.test(this)){
		acc.accept(this);
	}
}
```

Una vez creado un objeto v de tipo Vuelo, la invocación del método anterior para incrementar el precio de v si el número de pasajeros es menor de 50 sería:

```
v.aplicaAccion(x->x.getNumPasajeros()<50, x->x.incrementaPrecio(10.));
```

donde incrementaPrecio es el Consumer definido en el ejemplo 2.

9. BiConsumer<T,U>

BiConsumer es una interfaz para representar una acción con dos argumentos de entrada de distinto tipo. Se usa para representar acciones que modifiquen un objeto recibiendo un objeto de otro tipo. Tiene como interfaces especializadas: ObjDoubleConsumer, ObjIntConsumer y ObjLongConsumer que reciben un objeto de tipo T y otro del tipo especificado en el nombre. Todas implementan un método funcional denominado **accept**.

Ejemplo. Para cambiar la duración de un Vuelo, podríamos escribir el siguiente código:

```
BiConsumer<Vuelo, Duracion> cambioDuracion = (x,y)->x.setDuracion(y);
```

10. Supplier<T>

Supplier es una interfaz que porporciona un objeto de tipo T sin recibir ningún argumento mediante un método denominado **get**. Igualmente existen las interfaces especializadas BooleanSupplier, DoubleSupplier, IntSupplier y LongSupplier para proporcionar objetos del tipo indicado. En estos casos el método que implementan se denomina getAsX, donde X es Boolean, Double, Int o Long respectivamente.

Normalmente las interfaces de tipo Supplier se limitan a invocar constructores. Así por ejemplo, una lambda expresión para invocar al constructor de Vuelo suponiendo que VueloImpl tenga un constructor sin argumento:

```
Supplier<Vuelo> dameVuelo = ()-> new VueloImpl();
```

Si queremos que el Supplier tenga un argumento deberemos recurrir a escribir:

```
Supplier<Vuelo> dameVuelo(String s) {
      return ()->new VueloImpl(s);
}
```

Otra forma habitual de construir un Supplier es mediante una expresión de método:

```
Supplier<Set<Integer>> dameConjunto = HashSet::new;
```

11. UnaryOperator<T>

La interfaz UnaryOperator representa una operación que recibe un solo operando de tipo T y devuelve un resultado del mismo tipo, mediante un método denominado **apply**. Es un caso particular de la interfaz Function con el mismo tipo para el argumento de entrada y salida y Java la implementa como subinterfaz de Function. Java 8 tiene también las interfaces particulares DoubleUnaryOperator, IntUnaryOperator y LongUnaryOperator que implementan el método **applyAsX** siendo X la cadena de caracteres Double, Int o Long respectivamente. Esta interfaz como subinterfaz de Function implementa también los métodos default compose y andThen con el mismo significado.

Ejemplo 1. Si se necesita un operador para modificar un objeto Duracion añadiéndole una cantidad de minutos dada mediante un argumento, debería escribirse:

```
public UnaryOperator<Duracion> añadeMinutos(Integer m){
                return x -> x.suma(new DuracionImpl(0,m));
}
```

12. BinaryOperator<T>

La interfaz BinaryOperator representa una operación que recibe dos operandos de tipo T y devuelve un resultado del mismo tipo mediante un método denominado **apply**. Como se puede ver es un caso particular de la interfaz BiFunction donde los tres tipos T, U y R son el mismo y Java 8 la implementa como subinterfaz de BiFunction. Existen también las especializaciones DoubleBinaryOperator, IntBinaryOperator y LongBinaryOperator para operar con valores numéricos. En estas interfaces el método que implementan se denomina **applyAsX**, donde X puede tomar los nombres Double, Int o Long respectivamente.

Ejemplo 1. Tenemos el tipo Duracion definido que guarda la duración de un Vuelo en horas y minutos. Si el tipo Duracion ya tiene definido el método suma:

```
public Duracion suma(Duracion d) {
      Integer min = getMinutos() + d.getMinutos();
      Integer hor = getHoras() + d.getHoras();
      return new DuracionImpl(hor+min/60,min%60);
}
```

Entonces podemos redefinirlo como un BinaryOperator:

```
BinaryOperator<Duracion> sumadur = (x,y) -> x.suma(y);
```

Equivalente a esta otra expresión:

```
BinaryOperator<Duracion> sumadur = Duracion::suma;
```

Si el método suma no estuviera definido para Duracion se podría definir directamente:

```
BinaryOperator<Duracion> sumadur = (x,y)-> {
      Integer min = x.getMinutos() + y.getMinutos();
```

```
            Integer hor = x.getHoras() + y.getHoras();
            return new DuracionImpl(hor+min/60,min%60);
      };
```

Ejemplo 2. La interfaz DoubleBinaryOperator permite definir funciones reales como composición de otras. Por ejemplo si quisiéramos definir una función h como cociente de dos funciones desconocidas f y g, escribiríamos el código:

```
      public DoubleBinaryOperator funcionH(DoubleBinaryOperator f,
                                           DoubleBinaryOperator g){
            return (x,y)->f.applyAsDouble(x,y)/g.applyAsDouble(x,y);
      }
```

De esta manera una posible invocación para el cociente entre la suma y el producto de dos valores sería:

```
      public Double llamadaFuncionH(Double x, Double y){
            return funcionH((a,b)->a+b,(a,b)->a*b).applyAsDouble(x,y);
      }
```