

Ejercicios programación funcional

[Descargar estos ejercicios](#)

Índice

- [Ejercicio 1](#)
- [Ejercicio 2](#)
- [Ejercicio 3](#)
- [Ejercicio 4](#)
- [Ejercicio 5](#)

Ejercicio 1

Crea una aplicación que a partir de una Lista de enteros, te muestre los múltiplos de un número introducido por teclado que existan en la lista, usando **funciones-λ**. Resuelve de dos formas distintas, con clausura y sin clausura.

Ejercicio 2

Crea una aplicación que sirva para **buscar coincidencias en una lista de cadenas**. Para ello, definiremos una **función-λ** que reciba una lista y una cadena y sobre la lista con el método `filter` busque la cadena.

Ten en cuenta que el método `filter` necesitará un predicado para el cual utilizaremos otra función-λ para formarlo.

✦ **Nota:** Puede serte de utilidad la función `contains` sobre cadenas. Por último, muestra, con `forEach` la lista resultante.

Ejercicio 3

Vamos a realizar una serie de operaciones funcionales usando funciones-λ con el patrón

Map – Filter – Fold

Partiremos de la siguiente lista de números reales:

```
List<double> reales = new List<double> {  
    0.5, 1.6, 2.8, 3.9, 4.1, 5.2, 6.3, 7.4, 8.1, 9.2  
};
```

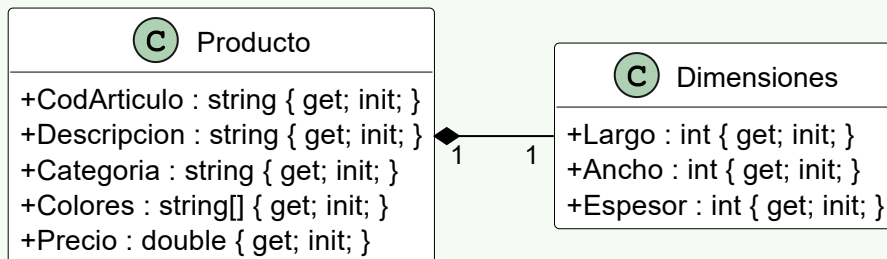
Vamos a realizar las siguientes operaciones:

1. Mostrar la lista usando el método `forEach(action:Consumer<T>)` de lista. Pasando a la función-λ action, una **clausura** de la variable string texto, en la que iremos componiendo su contenido separado por un espacio en blanco.
2. Cuenta aquellos elementos cuya **parte decimal es menor que 0.5**
 - **Map:** Paso del valor real a su parte decimal. Ej: $2.8 \rightarrow 0.8$
 - **Filter:** Filtro aquellas partes decimales que cumplen el predicado: $d < 0.5$
 - **Fold:** Contar los elementos en la secuencia resultante.
3. Calcular la suma de todos los valores de la secuencia cuya **parte entera sea múltiplo de 3**.
 - **Map:** Mapea el valor real a un objeto anónimo con la parte entera y el propio valor real de la secuencia. Ej: $2.8 \rightarrow \text{new } \{ e = 2, r = 2.8 \}$
 - **Filter:** Filtro aquellas partes enteras que cumplen el predicado: $o.e \% 3 == 0$
 - **Fold:** Suma todos los `o.r` de la secuencia resultante.
4. Calcular el máximo valor de la secuencia cuya parte decimal es mayor que **0.5**
 - **Map:** Mapea el valor real a un objeto anónimo con la parte decimal y el propio valor real de la secuencia. Ej: $2.8 \rightarrow \text{new } \{ d = 0.8, r = 2.8 \}$
 - **Filter:** Filtro aquellas partes decimales que cumplen el predicado: $o.d > 0.5$
 - **Fold:** Me quedo con el máximo de todos los `o.r` de la secuencia resultante.

Ejercicio 4

En [baseEjercicio4.kt](#) de este bloque de ejercicios, encontrarás definidas las siguientes clases

...



En la propiedad estática `productos` de la clase `Datos` te devolverá una secuencia de productos (`IEnumerable<Producto>`) sobre la que realizar las consultas.

Además, en el programa principal tienes un '*esqueleto*' a completar con descripción de cada consulta. Por ejemplo, para la primera consulta tendríamos ...

```
println(SeparadorConsulta);
println(
    "Consulta 1: Usando las funciones filter y map.\n" +
    "Mostrar CodArticulo, Descripcion y Precio .\n" +
    "de productos con Precio entre 10 y 30 euros\n"
);
val consulta1 : List<Any> = listOf() //A cambiar
println(consulta1.joinToString(separator = "\n"));
```

Nosotros deberemos rellenar la consulta de acuerdo a las especificaciones de la descripción, cuidando la presentación y sangría para que sean legibles. Por ejemplo ...

```

...
val consulta1 = Datos.productos.filter { it.precio >= 10 && it.precio <= 30 }
    .map { p: Producto ->
        object {
            val codArticulo = p.codArticulo
            val descripcion = p.descripcion
            val precio = p.precio
            override fun toString() = "$codArticulo $descripcion $precio"
        }
    }
...

```

Una vez completadas todas las consultas. Al ejecutar el programa la salida por pantalla del programa deberá ser ...

Consulta 1: Usando las funciones filter y map.
Mostrar CodArticulo, Descripcion y Precio .
de productos con Precio entre 10 y 30 euros

```

{ CodArticulo = A01, Descripcion = Uno, Precio = 15,05 }
{ CodArticulo = A02, Descripcion = Dos, Precio = 25,95 }
{ CodArticulo = A04, Descripcion = Cuatro, Precio = 18,45 }

```

Consulta 2: Usando las funciones map, sortedByDescending y take.
Muestra CodArticulo, Descripcion y Precio de los 3 productos.
más caros (ordenando por Precio descendente)

```

{ CodArticulo = A03, Descripcion = Tres, Precio = 30,25 }
{ CodArticulo = A02, Descripcion = Dos, Precio = 25,95 }
{ CodArticulo = A04, Descripcion = Cuatro, Precio = 18,45 }

```

Consulta 3: Usando las funciones groupBy, map, sortedByDescending y last.
Muestra el precio más barato por categoría

```

{ Categoria = C1, PrecioMasBarato = 15,05 }
{ Categoria = C2, PrecioMasBarato = 18,45 }

```

Consulta 4: Usando las funciones `groupBy`, `count`.

¿Cuántos productos hay de cada categoría?

```
{ Categoria = C1, NumeroProductos = 3 }  
{ Categoria = C2, NumeroProductos = 1 }
```

Consulta 5: Usando las funciones `groupBy`, `map` y `filter`

Mostrar las categorías que tengan más de 2 productos

C1

Consulta 6: Usando la función `map`

Mostrar `CodArticulo`, `Descripcion`, `Precio` y `Descuento` redondeado a 2 decimales, siendo `Descuento` el 10% del `Precio`

```
{ CodArticulo = A01, Descripcion = Uno, Precio = 15,05, Descuento = 1,5 }  
{ CodArticulo = A02, Descripcion = Dos, Precio = 25,95, Descuento = 2,6 }  
{ CodArticulo = A03, Descripcion = Tres, Precio = 30,25, Descuento = 3,03 }  
{ CodArticulo = A04, Descripcion = Cuatro, Precio = 18,45, Descuento = 1,84 }
```

Consulta 7: Usando las funciones `filter`, `contains` y `map`.

Mostrar `CodArticulo`, `Descripcion` y `Colores`

de los productos de color verde o rojo

(es decir, que contengan alguno de los dos)

```
{ CodArticulo = A02, Descripcion = Dos, Colores = [blanco, gris, rojo] }  
{ CodArticulo = A03, Descripcion = Tres, Colores = [rojo, gris, verde] }  
{ CodArticulo = A04, Descripcion = Cuatro, Colores = [verde, rojo] }
```

Consulta 8: Usando las funciones `filter` y `map`.

Mostrar `CodArticulo`, `Descripcion` y `Colores`.

de los productos que se fabrican en tres Colores

```
{ CodArticulo = A01, Descripcion = Uno, Colores = [blanco, negro, gris] }  
{ CodArticulo = A02, Descripcion = Dos, Colores = [blanco, gris, rojo] }  
{ CodArticulo = A03, Descripcion = Tres, Colores = [rojo, gris, verde] }
```

Consulta 9: Usando las funciones filter y map.
Mostrar CodArticulo, Descripcion y Dimensiones
de los productos con espesor de 3 cm

```
{ CodArticulo = A01, Descripcion = Uno, Dimensiones = L:4 x A:4 x E:3 }  
{ CodArticulo = A03, Descripcion = Tres, Dimensiones = L:5 x A:5 x E:3 }
```

Consulta 10: Usando las funciones flatMap, distinct y sortBy.
Mostrar los colores de productos ordenados y sin repeticiones

```
blanco  
gris  
negro  
rojo  
verde
```

Ejercicio 5


Vamos a practicar los conceptos del anterior tema, además de añadir HOF y funciones Lambda. Para ellos nos crearemos una **data class Usuario**, con solamente login y password de tipo String. Crearemos una interface sellada **UsuarioEvent**, para gestionar los eventos del usuario:

- **AñadeUsuario** al que le llega un usuario
- **ModificaUsuario** al que le llega un string
- **MuestraUsuario** de tipo object

Por otro lado tendremos una clase donde gestionaremos la lógica de la aplicación llamada **UsuarioViewModel**, que contendrá una lista mutable de usuarios y el método **onUsuarioEvent** al que le llega una **UsuarioEvent** y le da la funcionalidad necesaria a cada evento para que hagan lo que su nombre indica, sobre la lista de usuarios.

Para la parte de la interacción del usuario con la aplicación crearemos una **función usuarioScreen** a la que le llegará una función de nivel superior HOF, con un **UsuarioEvent** de parámetro de entrada y vacío de salida (**usuarioEvent: (UsuarioEvent) -> Unit**). Esta función tendrá un menú que nos permitirá añadir, modificar y mostrar usuarios invocando a los eventos a través del parámetro de entrada de la función. Por ejemplo, para mostrar podría ser algo como **usuarioEvent(UsuarioEvent.MuestraUsuarios)** .

En el programa principal habrá que crear un objeto de tipo **ViewModel** y con este llamar a la función de interacción con el cliente.

 **Tips:** Para realizar la llamada a la función con el objeto **ViewModel** se tendrá que hacer de la siguiente manera:

```
val usuarioViewModel = UsuarioViewModel() usuarioScreen(usuarioViewModel::onUsuarioEvent)
```