

Práctico de Programación

Programación funcional

Uni10.es

Índice de contenidos

Uni10.es

1. Conceptos sobre programación funcional
2. Introducción a las funciones lambda
3. Gestión de colecciones

¿Qué es la programación funcional?

- Paradigma de programación **declarativo**, no imperativo
 - Se dice cómo es el problema a resolver, en lugar de los pasos a seguir para resolverlo
- Ejemplos de lenguajes funcionales puros: *Miranda*, *Haskell*
- Ejemplos de lenguajes funcionales híbridos (también adaptados a otros paradigmas): *Clojure*, *Scala*
- La mayoría de lenguajes populares actuales no se pueden considerar funcionales, ni puros ni híbridos, pero han adaptado su sintaxis y funcionalidad para ofrecer parte de este paradigma

Características principales

- **Transparencia referencial**: la salida de una función debe depender sólo de sus argumentos. Si la llamamos varias veces con los mismos argumentos, debe producir siempre el mismo resultado.
- **Inmutabilidad de los datos**: los datos deben ser inmutables para evitar posibles efectos colaterales.
- **Composición de funciones**: las funciones se tratan como datos, de modo que la salida de una función se puede tomar como entrada para la siguiente.
- **Funciones de primer orden**: funciones que permiten tener otras funciones como parámetros, a modo de *callbacks*.

Transparencia referencial e inmutabilidad

- Si llamamos repetidamente a esta función con el parámetro 1, cada vez producirá un resultado distinto (3, 4, 5...)

```
class Prueba
{
    static int valorExterno = 1;

    static int UnaFuncion(int parametro)
    {
        valorExterno++;
        return valorExterno + parametro;
    }
}
```

Imperativo vs Declarativo

- Ejemplo en **Java**: obtener una sublista con los mayores de edad de entre una lista de personas

```
List<Persona> adultos = new ArrayList<>();  
for (int i = 0; i < personas.size(); i++)  
{  
    if (personas.get(i).getEdad() >= 18)  
        adultos.add(personas.get(i));  
}
```

IMPERATIVO

```
List<Persona> adultos =  
    personas.stream()  
        .filter(p -> p.getEdad() >= 18)  
        .collect(Collectors.toList());
```

DECLARATIVO

Más compacto

Menos propenso a errores

composición de funciones



Introducción a las funciones lambda

- Expresiones breves que simplifican la implementación de elementos más costosos en cuanto a líneas de código
- Normalmente aplicados a la implementación de interfaces, aunque en algunos lenguajes tienen más utilidades prácticas
- En algunos lenguajes se les suele denominar "funciones flecha" (*arrow functions*) ya que en su sintaxis es característica una flecha, que separa la cabecera de la función de su cuerpo.

Ejemplo en Java: comparaciones

- API del método `List.sort` de Java:

```
default void sort(Comparator<? super E> c)
```

- La interfaz `Comparator` pide implementar un método `compare`, que recibe dos datos del tipo a tratar (T), y devuelve un entero indicando si el primero es menor, mayor, o son iguales (de forma similar al método `compareTo` de la interfaz `Comparable`)

```
int compare(T o1,  
            T o2)
```


Ejemplo en Java: comparaciones

- Ordenar una lista de personas de mayor a menor edad SIN lambdas: se puede emplear **Comparable** o **Comparator**.


```
class Persona
{
    private String nombre;
    private int edad;
```

```
class ComparadorPersona implements Comparator<Persona>
{
    @Override
    public int compare(Persona p1, Persona p2)
    {
        return p2.getEdad() - p1.getEdad();
    }
}
```

```
ArrayList<Persona> personas = new ArrayList<>();
personas.add(new Persona("Nacho", 42));
personas.add(new Persona("Ana", 38));
personas.add(new Persona("Juan", 70));
personas.add(new Persona("Mario", 7));
personas.add(new Persona("Laura", 4));

personas.sort(new ComparadorPersona());

for (int i = 0; i < personas.size(); i++)
    System.out.println(personas.get(i).mostrar());
```



Ejemplo en Java: comparaciones

- Implementación CON lambdas:

```
ArrayList<Persona> personas = new ArrayList<>();
personas.add(new Persona("Nacho", 42));
personas.add(new Persona("Ana", 38));
personas.add(new Persona("Juan", 70));
personas.add(new Persona("Mario", 7));
personas.add(new Persona("Laura", 4));

personas.sort((p1, p2) -> p2.getEdad() - p1.getEdad());

for (int i = 0; i < personas.size(); i++)
    System.out.println(personas.get(i).mostrar());
```



Ejemplo en C#

- El mismo ejemplo anterior en **C#** CON lambdas quedaría así con el método **Sort** de la clase **List**, que necesita una interfaz **IComparer**, e implementar su método **Compare**:

```
List<Persona> personas = new List<Persona>();
personas.Add(new Persona("Nacho", 42));
personas.Add(new Persona("Ana", 38));
personas.Add(new Persona("Juan", 70));
personas.Add(new Persona("Mario", 7));
personas.Add(new Persona("Laura", 4));

personas.Sort((p1, p2) => p2.getEdad() - p1.getEdad());

for (int i = 0; i < personas.Count; i++)
    Console.WriteLine(personas[i].mostrar());
```




Estructura de una expresión lambda

Parámetros del método
a implementar (sin tipo
de dato)

"Flecha", separador de los
parámetros y el cuerpo de la
función

Código de la función a
implementar. Si es un simple
"return", pueden omitirse las
llaves y el "return"



```
(p1, p2) => p2.getEdad() - p1.getEdad()
```

- Los paréntesis del lado izquierdo pueden omitirse si sólo hay un parámetro, por norma general, en casi todos los lenguajes que usan este tipo de expresiones
- Si el código a la derecha de la flecha necesita hacer más que un simple "return", se pone entre llaves

Ejercicio 1

- Supongamos que tenemos implementada una clase Libro que tiene los atributos de *título*, *autor* y *precio*, con el correspondiente constructor y sus *getters*
- Supongamos también que tenemos una lista de libros en una variable *libros*
- Se pide implementar:
 - Un fragmento de código en **Java** que muestre los títulos de los libros por pantalla, ordenados de menor a mayor
 - Un fragmento de código en **C#** que muestre los 3 libros más caros de la colección, ordenados de mayor a menor precio

Gestión de colecciones con streams en Java

- Desde Java 8, permiten procesar grandes cantidades de datos aprovechando la paralelización que permita el sistema.
- No modifican la colección original, sino que crean copias.
- Dos tipos de operaciones
 - **Intermedias:** devuelven otro *stream* resultado de procesar el anterior de algún modo (filtrado, mapeo), para ir enlazando operaciones
 - **Finales:** cierran el stream devolviendo algún resultado (colección resultante, cálculo numérico, etc).
- Muchas de estas operaciones tienen como parámetro una interfaz, que puede implementarse muy brevemente empleando expresiones lambda

Operaciones intermedias streams: filtrado

- El método **filter** es una operación intermedia que permite quedarnos con los datos de una colección que cumplan el criterio indicado como parámetro.

```
ArrayList<Persona> personas = new ArrayList<>();  
// Aquí rellenaríamos la lista de personas  
  
Stream<Persona> adultos =  
    personas.stream()  
        .filter(p -> p.getEdad() >= 18);
```

filter recibe como parámetro una interfaz **Predicate**, cuyo método **test** recibe como parámetro un objeto y devuelve si ese objeto cumple o no una determinada condición



Aquellas personas "p" de la colección cuya edad sea mayor o igual que 18

Operaciones intermedias streams: mapeo

- El método **map** es una operación intermedia que permite transformar la colección original para quedarnos con cierta parte de la información o crear otros datos .

```
ArrayList<Persona> personas = new ArrayList<>();  
// Aquí rellenaríamos la lista de personas  
  
Stream<Integer> edades =  
    personas.stream()  
        .map(p -> p.getEdad());
```

map recibe como parámetro una interfaz **Function**, cuyo método **apply** recibe como parámetro un objeto y devuelve otro objeto diferente, normalmente derivado del parámetro



Las edades de aquellas personas "p" de la colección

Operaciones intermedias streams: combinar

- Se pueden combinar operaciones intermedias (composición de funciones) para producir resultados más complejos. Por ejemplo, las edades de las personas adultas:

```
ArrayList<Persona> personas = new ArrayList<>();  
// Aquí rellenaríamos la lista de personas  
  
Stream<Integer> edadesAdultos =  
    personas.stream()  
        .filter(p -> p.getEdad() >= 18)  
        .map(p -> p.getEdad());
```

Operaciones intermedias streams: ordenar

- El método **sorted** es una operación intermedia que permite ordenar los elementos de una colección según cierto criterio. Por ejemplo, ordenar las personas adultas por edad:

```
Stream<Persona> personasOrdenadas =  
    personas.stream()  
        .filter(p -> p.getEdad() >= 18)  
        .sorted((p1, p2) -> p1.getEdad() - p2.getEdad());
```

sorted recibe como parámetro una interfaz **Comparator**, que ya conocemos.



Para cada pareja de personas p1 y p2, ordénalas en función de la resta de la edad de p1 menos la edad de p2

Operaciones finales streams: colección

- El método **collect** es una operación final que permite obtener algún tipo de colección a partir de los datos procesados por las operaciones intermedias. Por ejemplo, una lista con las edades de las personas adultas:

```
ArrayList<Persona> personas = new ArrayList<>();  
// Aquí rellenaríamos la lista de personas  
  
List<Integer> edadesAdultos =  
    personas.stream()  
        .filter(p -> p.getEdad() >= 18)  
        .map(p -> p.getEdad())  
        .collect(Collectors.toList());
```


Operaciones finales streams: cadena

- El método **collect** también permite obtener una cadena de texto que una los elementos resultantes, a través de un separador común. En la función *Collectors.joining* se puede indicar también un prefijo y un sufijo para el texto.
- Por ejemplo, los nombres de las personas adultas.

```
ArrayList<Persona> personas = new ArrayList<>();  
// Aquí rellenaríamos la lista de personas  
  
String nombresAdultos =  
    personas.stream()  
        .filter(p -> p.getEdad() >= 18)  
        .map(p -> p.getNombre())  
        .collect(Collectors.joining(", ", "Adultos: ", ""));
```


Operaciones finales streams: forEach

- El método **forEach** permite recorrer cada elemento del *stream* resultante, y hacer lo que se necesite con él. Por ejemplo, sacar por pantalla en líneas separadas los nombres de las personas adultas.

```
ArrayList<Persona> personas = new ArrayList<>();  
// Aquí rellenaríamos la lista de personas  
  
personas.stream()  
    .filter(p -> p.getEdad() >= 18)  
    .map(p -> p.getNombre())  
    .forEach(p -> System.out.println(p));
```

Operaciones finales streams: media

- El método **average** permite, junto con la operación intermedia **mapToInt**, obtener una media de un *stream* que haya producido una colección resultante numérica. Por ejemplo, la media de edades de las personas adultas.

```
ArrayList<Persona> personas = new ArrayList<>();  
// Aquí rellenaríamos la lista de personas  
  
double mediaEdadAdultos =  
    personas.stream()  
        .filter(p -> p.getEdad() >= 18)  
        .mapToInt(p -> p.getEdad()).average().getAsDouble();
```

Gestión de colecciones con LINQ en C#

- LINQ (*Language-INtegrated Query*) es una potente librería para manipular colecciones de datos en C#
- Sintaxis similar a SQL
- Además, cuenta con métodos basados en predicados, para filtrar, ordenar y hacer otras operaciones avanzadas
- Se necesita incluir el espacio de nombres `System.Linq`.
- Lo que se devuelve en la manipulación se puede asignar normalmente a una lista (`List`), o también a un `IEnumerable` (interfaz implementada por la clase `List`)

Filtrado con "where"

- Listado de personas mayores de edad

```
IEnumerable<Persona> adultos =  
    from persona in personas  
    where persona.Edad >= 18  
    select persona;
```

```
foreach(Persona p in adultos)  
    Console.WriteLine(p.Nombre);
```



Obtenemos un IEnumerable
que se puede recorrer con un
foreach

Mapeado con "select"

- Listado de edades de las personas mayores de edad (también se puede recorrer con [foreach](#))
- Es la última instrucción que debe aparecer en la sentencia LINQ

```
IEnumerable<Int32> edadesAdultas =  
    from persona in personas  
    where persona.Edad >= 18  
    select persona.Edad;
```

Ordenación con "orderby"

- Listado de edades de las personas mayores de edad, ordenado de mayor a menor

```
IEnumerable<Int32> edadesOrdenadas =  
    from persona in personas  
    where persona.Edad >= 18  
    orderby persona.Edad  
    select persona.Edad
```

Sobre el objeto IEnumerable/List: métodos

- El objeto **IEnumerable/List** que se obtiene con la consulta LINQ dispone de otros métodos muy útiles para hacer filtrados, mapeos y otras operaciones complejas:
 - **Where** permite establecer un filtrado según una condición, similar al **filter** de los streams en Java
 - **Select** permite seleccionar un atributo dado de la colección, similar al **map** de los streams en Java
 - **OrderBy** y **OrderByDescending** permiten ordenar por un atributo de la colección, en orden ascendente o descendente, de forma similar al **sorted** de los streams en Java
 - **ToList** obtiene una lista, típicamente resultado de algunas las operaciones anteriores.
 - Otros métodos útiles: **Average**, **Min**, **Max**, **Any**...

Ejemplos con métodos de IEnumerable/List

```
// Listado de personas adultas ordenadas  
// de mayor a menor edad  
  
List<Persona> adultasOrdenadas =  
    personas.Where(p => p.Edad >= 18)  
              .OrderByDescending(p => p.Edad)  
              .ToList();
```

```
// Media de edad de las personas adultas  
  
double mediaEdad =  
    personas.Where(p => p.Edad >= 18)  
              .Average(p => p.Edad);
```

```
// Cadena con los nombres de las personas  
// adultas, separados por comas  
  
List<string> nombresAdultos =  
    personas.Where(p => p.Edad >= 18)  
              .Select(p => p.Nombre)  
              .ToList();  
  
string nombresUnidos =  
    String.Join(", ", nombresAdultos);
```

Ejercicio 2

- Supongamos una clase llamada *Receta* que almacena los datos de una receta de cocina: su *nombre*, su *categoría* (carnes, pastas...) y sus *calorías*, incluyendo su constructor y *getters*.
- Supongamos que tenemos una lista de recetas llamada *recetas*
- Se pide implementar las siguientes consultas en **Java** y **C#**:
 - Recetas de menos de 500 calorías
 - Nombres de las recetas de "carnes", ordenadas alfabéticamente
 - Media de calorías de las recetas de "verduras"
 - Cuántas recetas hay de más de 800 calorías

Prog. funcional en Python

- Disponemos del elemento `lambda` para definir expresiones lambda. Se definen primero los parámetros separados por comas, y tras los dos puntos el resultado a devolver

```
fun = lambda a, b : a+b  
print(fun(3, 4)) # Resultado: 7
```

```
# Listado de personas ordenadas de mayor a menor edad  
personas.sort(key = lambda p: p.edad, reverse = True)
```


Prog. funcional en Python

- Se tienen además funciones como `map` o `filter`, con una funcionalidad similar a las vistas en Java

```
def square(a):  
    return a*a
```

```
data = [1, 3, 5]
```

```
# Map sobre función  
print(list(map(square, data)))
```

```
# Map sobre lambda  
print(list(map(lambda a: a*a, data)))
```

```
def odd_number(num):
```

```
    if num%2 == 0:  
        return True
```

```
    else:  
        return False
```

```
data = [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
# Filter sobre función  
print(list(filter(odd_number, data)))
```

```
# Filter sobre lambda  
print(list(filter(lambda x: x%2 == 0, data)))
```

Ejercicio 3

Uni10.es

- Realiza en **Python** el ejercicio 2

Prog. funcional en C y C++

Uni10.es

- C y C++ son lenguajes muy limitados en este aspecto
- Podemos destacar sobre todo los **punteros a funciones**, que permiten definir funciones de primer orden.
- Se utilizan para llamar a cualquier función que cumpla la estructura del puntero

Prog. funcional en C y C++: ejemplo (1/3)

- Podemos definir una función de ordenación que reciba como parámetros el array a ordenar, y un puntero a una función que marcará el criterio de comparación:

```
void ordenar(struct persona personas[],
            int (*comparador)(struct persona, struct persona))
{
    int i, j;
    struct persona auxiliar;

    for (i = 0; i < TAMANO - 1; i++)
    {
        for (j = i+1; j < TAMANO; j++)
        {
            if (comparador(personas[i], personas[j]) > 0)
            {
                auxiliar = personas[i];
                personas[i] = personas[j];
                personas[j] = auxiliar;
            }
        }
    }
}
```

Prog. funcional en C y C++: ejemplo (2/3)

- Ahora podemos definir dos comparadores:
 - Por nombre de menor a mayor (devolveremos un número negativo si el nombre de la izquierda es menor)
 - Por edad de menor a mayor (devolveremos un número negativo si la edad de la izquierda es menor)

```
int comparaNombres(struct persona p1, struct persona p2)
{
    return strcmp(p1.nombre, p2.nombre);
}

int comparaEdades(struct persona p1, struct persona p2)
{
    return p1.edad - p2.edad;
}
```

Prog. funcional en C y C++: ejemplo (3/3)

- Con esto, podemos llamar a la función de ordenación y ordenar por cualquiera de estos dos criterios:

```
int i;
struct persona personas[TAMANO];

// Ordenar por nombre y mostrar
ordenar(personas, comparaNombres);
printf("Listado ordenado por nombre:\n");
for(i = 0; i < TAMANO; i++)
    printf("%s, %d años\n", personas[i].nombre, personas[i].edad);

// Ordenar por edad y mostrar
ordenar(personas, comparaEdades);
printf("Listado ordenado por edad:\n");
for(i = 0; i < TAMANO; i++)
    printf("%s, %d años\n", personas[i].nombre, personas[i].edad);
```


Ejercicio 4

Uni10.es

- Realiza en **C/C++** un programa que defina un puntero a función que reciba un array de enteros junto con su tamaño como parámetros, y devuelva otro entero. Utiliza este puntero a función para calcular la media, el mínimo y el máximo de un array de enteros dado.