Lenguaje Java

Funciones Lambda

https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html
https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html

https://www.adictosaltrabajo.com/2015/12/04/expresiones-lambda-con-java-8/
https://www.ecodeup.com/entendiendo-paso-a-paso-las-expresiones-lambda-en-java/
https://www.arquitecturajava.com/el-concepto-de-java-8-reference-method/

Interfaz funcional

- Se crean en **java 8**
- Son útiles para **definir operaciones, de forma rápida**, que afectarán a cada uno de los objetos de un conjunto (lista, vector, fichero,..). Por ejemplo sumar 1 a todos ellos o elegir solo algunos de ellos
- Una **interfaz es funcional** si :
  - **Solo tiene un método abstracto**. (Puede tener otros métodos static o implementados por defecto (con default))

Se utiliza la anotación ***@FunctionalInterface*** para indicar al compilador que la interfaz es funcional y que haga las comprobaciones pertinentes.

Se suelen implementar con una **clase anónima**.

Hay muchos interfaces funcionales, por ejemplo, Comparator:

```
Collections.sort(lista, new Comparator<String>() {    //Ordenar la cadena por su longitud
   @Override
   public int compare(String str1, String str2) {
      return str1.length()-str2.length();
   }
});
```

Ejemplo Interfaz funcional

```
public class UsoInterfazFuncional {
   @FunctionalInterface
    public interface IFuncionLambda {
    //método abstracto para sumar 2 números, que lo implementará el programador
a partir de una expresión Lambda
        public void suma(int a, int b);
    }

public static void main(String[] args) {
  int x = 10;
  int y = 5;
  //se implementa el método de la interfaz con una expresión lambda
  IFuncionLambda iflambda = (a, b) -> { System.out.println(a + b); };
  //se utiliza el método con la implementación y se le envía x e y
  iflambda.suma(x, y);
  //se puede declarar otra diferente
  IFuncionLambda iflambda2 = (a, b) -> { System.out.println(2*a + b); };
  //se utiliza el método con la nueva implementación y se le envían
 // los valores 100 e y
  iflambda2.suma(100, y);
}
}
```

Funciones lambda

- Se introducen en Java 8 para facilitar la codificación y evitar crear clases anónimas de un solo método.
- Como java sólo permite clases e interfaces, las expresiones o funciones lambda son interfaces que solo soportan un método abstracto.
- Pueden ser utilizadas donde el tipo aceptado sea una interfaz funcional.

  Collections.sort(lista, **(str1, str2)-> str1.length()-str2.length()**);

- Se utilizan con varios tipos de interfaces funcionales como el *Consumer*<T> (método *void accept (T t)).*

Ejemplo en el que el método accept recibe un dato y no devuelve nada, lo que es compatible con la función lambda indicada

Consumer<*String*> *consumidor =* ***(x) -> System.out.println(x)****;*

*consumidor.****accept****("hola");*

Funciones lambda

- Las funciones lambda son funciones anónimas reducidas, pueden tener parámetros, tienen instrucciones y pueden devolver valores (no es necesario, aunque se puede, usar *return*)

Formato: **(parámetros) -> {cuerpo}**

Se puede definir el tipo de los parámetros o se puede omitir si se puede inferir del entorno.

Los paréntesis de los parámetros son opcionales si hay 1 parámetro (si hay 0 o más de 1 son obligatorios)

Las llaves del cuerpo son opcionales cuando solo hay 1 instrucción.

Ordenación con funciones Lambda

```
public class OrdenarArrays {
static void mostrar(Persona v[]) {
  for (int i=0; i<v.length;i++)
    System.out.println(v[i]);
}
```

```
public class Persona {
  String nombre;   int edad;
public String toString() {
  return "Persona [nombre=" +
nombre + ", edad=" + edad + "]";
}
…
}
```

```
public static void main(String[] args) {
  Persona vp[]= {new Persona("uno",33), new Persona("dos",22),
new Persona("tres",11)};
  Arrays.sort(vp, (Persona a, Persona b) -> (b.edad - a.edad) );
//también se puede utilizar el formato, mas flexible:
  Arrays.sort(vp,(Persona a, Persona b) -> {
    int r=b.edad - a.edad;
    return r;}); //fin exp. lambda
 mostrar(vp);
}}
```

Funciones lambda

- Desde Java-11 se admite el parámetro var en funciones lambda;
- //antes de java 11

(x, y) -> x.metodo(y)

- //desde java 11: también

(var x, var y) -> x.metodo(y)

- Hay que escoger entre usar var o usar el tipo explicito, ya que si se usan así:

(var x, int y) -> x.metodo(y)

tendremos una excepción.

No es posible utilizar variables externas dentro de la expresión lambda a menos que estas sean final.

High Order Functions (HOF)

- Funciones que recibe una función (o varias) como parámetro **o** devuelve una función como salida.
- En general podemos tener funciones asociadas a variables de nuestra aplicación.

```
Predicate<String> filtro3 = (nombre) -> nombre.length() <= 3;
```

- Y luego utilizarla en otra llamada a función:

```
lista.stream().filter(filtro3).forEach(nombre ->
System.out.println(nombre));  // List<String> lista
```

- https://www.arquitecturajava.com/utilizando-java-high-order-functions/
- https://medium.com/@avicsebooks/functional-programming-in-java-8b3f73b11df0

High Order Functions (HOF)

```
public class EjHighOF1 {
List<String> listaNombres;

EjHighOF1(){
listaNombres = new
ArrayList<String>();
listaNombres.add("Pedro");
listaNombres.add("Miguel");
listaNombres.add("Ana");
listaNombres.add("Isabel");
listaNombres.add("MariaPilar");
}
void listado2() {
imprimir
(listaNombres,System.out::println,3);
System.out.println("*********");
imprimir
(listaNombres,System.out::println,5);
System.out.println("*********");
imprimir
(listaNombres,System.out::println,7);
}
```

```
public void imprimir (List<String>listaNombres
,Consumer<String> consumidor, int size) {
listaNombres.stream().filter(filtroSize(size))
.forEach(consumidor);
}

public static Predicate<String>
filtroSize(final int longitud) {
return texto -> texto.length() <= longitud;
}
```

Utilizamos 2 HOF (imprimir y filtroSize) para simplificar el código

Referencias a métodos

- La sintaxis:

  referenciaObjetivo::nombreDelMetodo

Nos permite reutilizar un método como expresión lambda

Ejem:

File::canRead // en lugar de File f -> f.canRead();

las referencias a los métodos permiten una anotación más rápida para expresiones lambda simples

- Método estático:

  (String info) -> System.out.println(info) *// Expresión lambda sin referencias.*

  System.out::println *// Expresión lambda con referencia a método estático.*

- Método de instancia de un tipo:

  (Estudiante estudiante, int pos) -> estudiante.getNota(pos) *// Expresión lambda sin referencias.*

  // Expresión lambda con referencia a método de un tipo.

```
IntFuncInstancia if2=new Estudiante("Pepe")::getNota;
int not=if2.getNota(3);
```

  Veremos más adelante su uso

Ejemplo uso referencias

```
public class ReferenciaMetodosEstaticos{
  public static void main(String[] args) {
   List names = new ArrayList();
     names.add("Uno");
     names.add("Dos");
     names.add("Tres");
  //Método estático
     names.forEach(System.out::println);
 }
}

//Método no estático
 Function<Persona, String> ftoString=
Persona::toString;
 System.out.println(ftoString.apply(new
Persona("Santiago,18));
```

```
// referencia a mensajes
LinkedList<Integer> lst=new
LinkedList<Integer>(Arrays.asList(1, 2, 3));
Supplier<Integer> funcion3 = lista::removeLast;
System.out.println(funcion3.get()); // 3
lista.forEach(System.out::println);

//referencia a constructores
Supplier< Persona > per= Persona::new;
//Construye un objeto de tipo Persona que es
devuelto por método get(); del interfaz funcional
Supplier
Persona pers=per.get();
```

Ejemplo uso referencias

```
class Estudiante {
 int []notas= {4,5,7,6,2,3};
 String nombre;
 static int totEstudiantes=0;
 Estudiante(String n){
 nombre=n;
 totEstudiantes++;
 }
 static int getTotEstudiantes() {
 return totEstudiantes;
 }
 int getNota (int pos) {
 return notas[pos];
 }
}
```

```
@FunctionalInterface
public interface IntFuncTotEst {
public int getTotEstudiantes();
}

@FunctionalInterface
public interface IntFuncInstancia {
int getNota (int pos);
}
```

```
public class Notacion2Ptos {
public static void main(String[] args) {
       Estudiante e1= new Estudiante ("Juan");
       Estudiante e2= new Estudiante ("Ana");

       IntFuncTotEst ife=Estudiante::getTotEstudiantes;
       int tot= ife.getTotEstudiantes();
       System.out.println("tot estudiantes:"+tot);

       IntFuncInstancia if2=new
Estudiante("Pepe")::getNota;
       int not=if2.getNota(3);
       System.out.println("La 3ª nota es:"+ not);
}
}
```

Tipos de expresiones lambda

- Proveedores
- Consumidores.
- Funciones.
  - Operadores Unarios.
  - Operadores Binarios.
- Predicados.

Proveedores (*Supplier)*

- No tienen parámetros y devuelven un valor.
- Interface Funcional ***Supplier<T>*** *método* ***T get()***

Ejem.

```
Supplier<String> cadena = () -> "Ejemplo de Proveedor";
System.out.println(cadena.get());
```

Hay interfaces para:

- IntSupplier
- LongSupplier
- DoubleSupplier
- BooleanSupplier

Ejemplo uso proveedor(Supplier)

```
public class Persona {
  private String nombre;
  private int edad;
public Persona(){}
public Persona(String nombre, int e) {
  this.nombre = nombre;
  this.edad = e;
}
public String getNombre() {
  return nombre;
}
public void setNombre(String nombre) {
  this.nombre = nombre;
}
public int getEdad() {
  return edad;
}
public void setEdad(int edad) {
  this.edad = edad;
}
public String toString() {
  return "nombre=" + nombre + ", edad=" + edad + "]";
}
}
```

```
import java.util.function.Supplier;
public class LambdaPersona {
public static Persona crearPersona(){
  return new Persona("Pablo", 32);
}
public static void main(String[] args) {
//se crea un proveedor de tipo Persona, el cual obtiene una persona
  Supplier<Persona> supplier = LambdaPersona::crearPersona; //obtiene desde el proveedor la persona y la asigna a per
  Persona per = supplier.get();
        // imprime el nombre
  System.out.println(per.getNombre());
}
}
```

Consumidores (*Consumer)*

- Aceptan un solo valor y no devuelven ningún valor.

  Interfaz Funcional ***Consumer<T>*** *método* ***void accept (T)***

Ejem.

```
int a -> System.out.println(a);
```

Ejem.

```
Consumer<Persona> persona = (p) -> System.out.println("Hola, " +
p.getNombre());
persona.accept(new Persona("Jorge", "Valladares","Quito"));
```

- BiConsumidores. Reciben dos valores como parámetro y no devuelven resultado.

  Interfaz Funcional BiConsumer**<T,U>** **void accept(T t, U u)**

Ejem.

```
(int a, String msg) -> System.out.println(msg+ a);
```

Con consumer.andThen (expr Lambda) puedes componer consumidores

Consumidores

Se suelen utilizar para imprimir :

```
 miLista.stream().
filter((p)>p.getNombre().equals("Alicia")).
forEach(System.out::println);
```

```
static void imprimir( List<Persona> l,
Consumer c){
    for (Persona p:l)
       c.accept(p);
}
```

Con consumer.andThen (expr Lambda) puedes componer consumidores

```
 System.out.println("nombres mas 18");
 Consumer<Persona> nombre= p->System.out.println(p.getNombre());
 miLista.stream().filter((p)->p.edad>18).forEach(nombre);
 Consumer <Persona> nombreEdad = nombre.andThen(p ->
System.out.println(" Edad: "+
   p.getEdad()));
 System.out.println("edades y nombres mas 18");
 miLista.stream().filter((p)->p.edad>=18).forEach(nombreEdad);
imprimir (miLista, nombreEdad);
```

Funciones (*Function)*

- Se usan para hacer transformaciones de objetos.
- Aceptan un argumento y devuelven un valor como resultado. Resultado y argumento pueden ser de diferente tipo.
- Interface Funcional ***Function<T,R> método: R apply (T)***

Ejem.

```
Function<Integer, Integer> suma = x -> x + 10;
System.out.println("La suma de 5 + 10: " + suma.apply(5));
```

- BiFunciones. Reciben dos valores como parámetro y devuelven un resultado
- Interface Funcional BiFunction<T,U,R> método R apply(T t, U u)

Ejem.

```
(int a, int b) -> a+b;
```

Operadores Unarios: Funciones en las que el valor recibido y el devuelto son del mismo tipo.

Operadores Binarios: Funciones en las que los valores recibidos y el devuelto son del mismo tipo.

Funciones Ejem mapeo

```
 static List<Integer> soloEdades (List<Persona> lis, Function<Persona, Integer>
fPerInt){
    List<Integer> res= new ArrayList<Integer>();
    for (Persona p:lis)
    res.add(fPerInt.apply(p));
    return res;
  }
public static void main(String[] args) {
 ArrayList<Persona> miLista= new ArrayList<Persona>();
 miLista.add(new Persona("Miguel",15));
 miLista.add(new Persona("Alicia",34));
 miLista.add(new Persona("Carlos",72));
 miLista.add(new Persona("Alicia",12));

 Function<Persona, Integer> funPerInt=(Persona p)-> {return p.edad;};
 List<Integer> lisEdad=soloEdades(miLista, funPerInt);
 for (Integer i: lisEdad)
   System.out.println(i);
}
```

Se puede hacer más fácilmente con Stream (ver transparencia de stream con map)

Funciones Ejem mapeo

También se pueden combinar con

- andThen: compone 2 funciones
- compose: compone 2 funciones en orden diferente a andThen
- identity: devuelve el argumento que recibe.

Predicados (*Predicate)*

- Aceptan un argumento y devuelven un valor booleano.

Ejem.

```
int a ->   a%2==0;
```

- Bipredicados. Reciben dos valores como parámetro y devuelven un resultado booleano

Interfaz Funcional Predicate<T>: método **boolean test(T n)**

```
Predicate<Persona> jubilado = (p)->p.edad>=65;
Predicate<Persona> menor = (p)->p.edad<=18;
```

Se puede utlizar or, and o negate:

```
Predicate<Persona> noTrabajador= jubilado.or(menor);
Predicate<Persona> noJubilado = jubilado.negate();
```

Ejemplo uso predicado

```
public class Predicados {
public static void evaluar(List<Integer> listaNumeros, Predicate<Integer> predicado) {
  for(Integer n: listaNumeros)  {
    if(predicado.test(n)) {
      System.out.print(n + " ");
    }
  }
  System.out.println();
}

public static void main(String[] args) {
  List<Integer> listaNumeros = Arrays.asList(1, 2, 3, 4, 5, 6, 7,8,9,10);
  System.out.println("Números pares:");
  evaluar(listaNumeros, (n)-> n%2 == 0 );

  System.out.println("Números impares:");
  evaluar(listaNumeros, (n)-> n%2 == 1 );

  System.out.println("Números mayores a 5:");
  evaluar(listaNumeros, (n)-> n > 5 );
}
}
```

Otros ejemplos de Func. Lambda I

```
public class Principal1 {
 public static <T> void procesar(
Consumer<T> expresion, T mensaje) {
  expresion.accept(mensaje);
}
```

```
public static void main(String[] args) {
  Consumer<String> consumidor = (x) -> System.out.println(x);
  consumidor.accept("hola");
  procesar ((x)->System.out.println(x),"hola2");
}
}
```

```
public class Principal2 {
 public static <T> void procesar(Consumer<T> expresion, T mensaje) {
   expresion.accept(mensaje);
}
 public static void imprimir(String mensaje) {
   System.out.println("---------"); procesar(Principal2::imprimir,"hola3");

   System.out.println(mensaje);
   System.out.println("---------");
 }
 public static void main(String[] args) {
  Consumer<String> consumidor = (x) -> System.out.println(x);
   consumidor.accept("hola");

   procesar ((x)->System.out.println(x),"hola2");
   procesar(Principal2::imprimir,"hola3");
  procesar(new Impresora()::imprimir,"hola4");
}

}
```

```
public class Impresora {
  public static void imprimir (String mensaje) {
    System.out.println("imprimiendo impresora");
    System.out.println(mensaje);
    System.out.println("imprimiendo impresora");
  }
}
```

Otros ejemplos de Func. Lambda II

```
public class ListaPersonas1 {
  static void ordenarSinLambda(ArrayList<Persona> miLista){
Collections.sort(miLista,new Comparator<Persona>() {
  public int compare(Persona p1,Persona p2) {
    return p1.getNombre().compareTo(p2.getNombre());
      }
});
}
```

```
static void ordenarConLambda(ArrayList<Persona> miLista){
  Collections.sort(miLista,
  (Persona p1,Persona p2)-> p1.getEdad()-p2.getEdad());
  }
static void imprimir(ArrayList<Persona> milista) {
  for (Persona p: milista) {
    System.out.println(p.getNombre()+" "+p.getEdad());
}
}
```

```
public static void main(String[] args) {
ArrayList<Persona> miLista= new ArrayList<Persona>();
miLista.add(new Persona("Miguel",15));
miLista.add(new Persona("Alicia",34));
miLista.add(new Persona("Carlos",32));
ordenarSinLambda(miLista);
imprimir(miLista);
System.out.println("_________________");
ordenarConLambda(miLista);
imprimir(miLista);
 }
}
```

Stream

- Ejecución anidada de funciones.

- Permite concatenar operaciones sobre una lista de datos para facilitar la comprensión de la operación.
- No es más eficaz que la programación tradicional
- https://docs.oracle.com/javase/9/docs/api/java/util/stream/Stream.html

Streams con funciones lambda I

```
public class FiltradoPersona {
  public static void main(String[] args) {
Persona p1= new Persona("Juan",10);
Persona p2= new Persona("Ana",20);
Persona p3= new Persona("Oscar",15);
Persona p4= new Persona("Pepe",18);
List<Persona> lista= new ArrayList<Persona>();
lista.add(p1);
lista.add(p2);
lista.add(p3);
lista.add(p4);
//se van pasando diferentes filtros al stream
Persona filtroPersona=
 lista.stream().filter(elemento->elemento.getEdad()>15).findFirst().get();
//La menor persona de los mayores de 15
 System.out.println(filtroPersona.getNombre()+" "+filtroPersona.getEdad());
  }
}
```

Streams con funciones lambda II

```
public class StreamConLambda1 {
 public static void main(String[] args) {
 ArrayList<Persona> lista= new ArrayList<Persona>();

 lista.add(new Persona("Ana",10));
 lista.add(new Persona("Oscar",45));
 lista.add(new Persona("Carlos",70));
 lista.add(new Persona("Antonio",5));

 double resultado=lista.stream()
 .mapToDouble(pers->pers.getEdad()*12)
 .filter(edad->edad<=120)  //menos de 120 meses
 .sum();

 System.out.println(resultado);
 }
}
```

Streams con map

```
public class StreamConLambda1 {
 public static void main(String[] args) {
 ArrayList<Persona> miLista= new ArrayList<Persona>();
 miLista.add(new Persona("Miguel",15));
 miLista.add(new Persona("Alicia",34));
 miLista.add(new Persona("Carlos",72));
 miLista.add(new Persona("Alicia",12));
 List<String> listaNombres= miLista.stream().
   map((p)->p.getNombre()).  //transformo con Function implicita
   collect(Collectors.toList());  //convierto en lista de String
 for (String s: listaNombres)
   System.out.println(s);
 }
}
```

Ver stream reduction para aplicación en paralelo de reducciones