Expresiones lambda e interfaces funcionales

Desde java 8, se pueden programar algunos aspectos siguiendo el paradigma de programación funcional que para algunos contextos es más apropiada que las técnicas de programación imperativa. Dos conceptos básicos en la programación funcional en Java son *expresión lambda* e *interfaz funcional*.

El interface funcional no es un concepto del paradigma de programación funcional, pero es importante para java ya que forma parte del mecanismo que permite la existencia de expresiones lambda en Java.

EXPRESIONES LAMBDA

Se componen de tres partes

Argument List	Arrow Token	Body
`(int x, int y)`	`->`	`x + y`

Ejemplos de expresiones lambda

```
(int x, int y) -> x + y

() -> 42

(String s) -> { System.out.println(s); }
```

Supongo que la sintaxis de una expresión lambda te recordará a la definición de una función matemática … de hecho es, o mejor dicho, se comporta como una función. Una función sin nombre pero una función al fin.

Parámetros en las lambda

Pueden tener 0 parámetros

```
() -> System.out.println("lambda sin parámetros");
```

un parámetro

```
(p) -> System.out.println("Un parámetro: " + p);
```

cuando toma sólo un parámetro podemos omitir los paréntesis

```
p -> System.out.println("Un parámetro: " + p);
```

varios parámetros, separados por comas

```
(p1,p2) -> System.out.println("parámetro 1: " + p1+" parámetro 2: " + p2 );
```

declaración de tipos de los parámetros

con expresiones lambda java hace siempre que es posible "inferencia de tipos" y no es necesario declarar el tipo en muchas situaciones. Si por el contexto, no es posible inferir el tipo hay que declararlo

```
(MiClase mc) -> System.out.println("Nombre de mc: " + mc.getNombre());
```

Cuerpo en las lambda

Cuando es multilínea necesitamos {} similar al cuerpo de un método

cuerpo de una línea

```
() -> System.out.println("linea 1");
```

cuerpo de varias líneas

```
() ->{ System.out.println("linea 1");
        System.out.println("linea 2");
}
```

Retorno en las lambda

No hay que declarar el tipo de retorno como en los métodos ya que se infiere de lo que devuelve el cuerpo de la lambda.

Con return.

Si usamos return el cuerpo tiene que llevar {} incluso aunque sólo tenga una instrucción

```
() ->{ System.out.println("Hola");
        return "Adios";
}

() ->{ return "Adios";}
```

Sin return

Si la expresión tiene una sola línea. Esta línea será una expresión que devuelve un valor y no es necesario escribir return y por tanto podemos omitir los {}

```
() -> "Adios"
() -> 3>2
```

aunque te sorprenda, en la práctica es raro usar lambdas con cuerpo de más de 1 instrucción y por lo tanto podemos olvidarnos de llaves y return

¿Qué es una expresión lambda en programación?

Es una forma de escribir una función. Más concretamente es una forma de escribir una función anónima.

x -> x+y ¡no tiene nombre!

En Kotlin y Python podemos trabajar con:

- funciones con nombre. Las que se declaran con fun en Kotlin o def en python
- funciones sin nombre con expresiones lambda.

En java no hay funciones con nombre, qué pena.

¿Es un método java una función?

Aparentemente, en java lo más parecido a una función matemática o a una función de programación funcional es un *método,* debido a su estructura de argumentos+cuerpo+retorno, pero esto no es suficiente para que se pueda comportar como una función matemática o de programación funcional ya que por ejemplo:

- no se puede asignar directamente un método a una variable
- no se puede pasar directamente un método como parámetro.

Recuerda que discutimos que el código de un método se almacena o encapsula dentro de cada objeto, por ejemplo, el código del método len() de los objetos String va dentro de cada objeto String que se cree en nuestro programa. Esta cuestión de almacenamiento, fue acertada en su momento para ganar eficiencia

pero hoy en día genera limitaciones como por ejemplo no poder soportar el hecho de que a una variable se le pueda asignar un método. Observa que no hay forma de indicar en java la sentencia de asignación:
variable= metodo.
sólo puedo almacenar un tipo primitivo o un objeto completo pero no se puede referenciar directamente el método

¿Qué es una expresión lambda en programación java?: Una función que se representa mediante un objeto
Vimos antes el concepto de expresión lambda en programación en general, y concluimos que era una función anónima, pero además si trabajamos en Java conviene tener presente que "hay detrás" de una lambda.

Java es un lenguaje orientado a objetos así que va a haber una relación entre objeto y lambda. **Una lambda es un objeto que representa y por tanto es, una función.**
Hay objetos que representan coches, personas, …, y con la sintaxis lambda generamos un objeto que representa una función anónima.

Relación lambda/interface funcional
Sabemos que en Java todo objeto tiene que tener un tipo(String, Persona, Coche, …). Una expresión lambda es un objeto, pero no un objeto de cualquier tipo, tiene que ser un objeto instancia de una clase que implementa un interface funcional.

el contexto de uso más básico como veremos en los próximos ejemplos es asignar una lambda a una variable, siendo la variable de tipo "interface funcional"

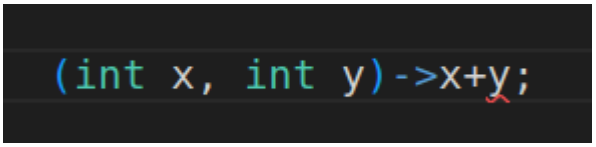
```
@FunctionalInterface
interface MiComparador{
   public boolean esMenor(int a1, int a2);
}
class App{
    public static void main(String[] args) {
       MiComparador mc=(a1,a2) -> a1<a2;
       System.out.println(mc.esMenor(5,2));
   }
}
```

¡Conseguimos en java asignar una función a una variable gracias a la expresión lambda!

¿donde se usan las lambdas?
Similar a un objeto cualquiera ya que una lambda genera un objeto. Teniendo en cuenta que el objeto que genera la lambda tiene que tener un tipo que debe corresponderse con un interface funcional
Por lo tanto igual que un *new Coche()* una lambda puede aparecer en

- en expresión de asignación varRef=objeto.
- en un argumento de método
- en un return

Una expresión lambda no se puede usar directamente como una sentencia. Si escribes una expresión lambda como si fuera una sentencia el compilador nos muestra error



Las funciones con nombre de kotlin o python en cambio sí pueden constituirse en una sentencia, ya que, al usar su nombre "invocamos a la función"

```
fun saludar(){//saludar es una función con nombre
   print("Hola mundo")
}
fun main() {
  saludar();//invocamos a una función con nombre
}
```

INFERENCIA DE TIPOS

El mecanismo de inferencia de tipos de Java es fundamental cuando se usan expresiones lambda. Recuerda que en ciertas situaciones Java hace una deducción o inferencia del tipo que le corresponde a una variable por el contexto de uso. Al trabajar con genéricos se utiliza esta característica y al trabajar con expresiones lambda también.

Recuerda el ejemplo

```
@FunctionalInterface
interface MiComparador{
    public boolean esMenor(int a1, int a2);
}
class App{
     public static void main(String[] args) {
        MiComparador mc=(a1,a2) -> a1<a2;
        System.out.println(mc.esMenor(5,2));
    }
}
```

el tipo de a1 y a2 de la expresión lambda se deducen como int ya que hay una sentencia de asignación

```
MiComparador mc=(a1,a2) -> a1<a2;
```

Y java revisa la definición de MiComparador y deduce que a1 y a2 son de tipo int

INTERFACE FUNCIONAL

No es un concepto nuevo, es más bien una terminología nueva para simplemente designar a los interfaces que constan de un sólo método. Lo que sí va a ser novedoso es la relación que se establece en Java entre un interface funcional y una expresión lambda.

Ejemplo de interface funcional:

```
@FunctionalInterface
interface MiComparador{
    public boolean esMenor(int a1, int a2);
}
```

MiComparador tiene un único método, por tanto es un interface funcional.

RELACIÓN ENTRE INTERFAZ FUNCIONAL, LAMBDA Y OBJETOS

Ya que java es un lenguaje OO, finalmente el funcionamiento de las lambda expressions se traduce a la mecánica de POO. Cuando escribimos una *lambda expression* lo que realmente ocurre "interna y automáticamente "es que:

1. Implementamos un interface funcional y por consiguiente sobrescribimos su método.
2. Se crea una instancia de dicha implementación

itodo esto lo hace automáticamente java cuando escribimos la expresión lambda!.

Establecer estas relaciones es importante para nosotros porque trabajamos en java cuyo funcionamiento se basa en la interacción entre objetos, pero si estuviéramos programando en un lenguaje de programación funcional puro las explicaciones anteriores serían detalles de bajo nivel sin interés alguno.

Ejemplo para ilustrar estos conceptos:

```
@FunctionalInterface
interface MiComparador{
   public boolean esMenor(int a1, int a2);
}

class App{
    public static void main(String[] args) {
       //escribimos expresiones lambda que implementan MiComparador

       //en el cuerpo tenemos dos intruciones la  1ª se imprime frase y la 2ª se devuelve boolean a1<a2.
       // es muy infrecuente que haya más de una instrucción, es tan sólo un ejemplo
       MiComparador mc=(a1,a2) -> {System.out.print("comparando "+a1+" y "+ a2+": ");return a1<a2;};
       boolean result = mc.esMenor(2, 5);
       System.out.println(result);

       //no es necesario realmente llaves y return
       mc=(int a1,int a2) -> {return a1<a2;};
       System.out.println(mc.esMenor(5,2));

       //si sólo hay una instrucción no es necesario return ni llaves
       mc=(int a1,int a2) -> a1<a2;
       System.out.println(mc.esMenor(5,2));

       // no es necesario declarar los tipos de los parámetros pues se infieren
       mc=(a1,a2) -> a1<a2;
       System.out.println(mc.esMenor(5,2));

       //en la 5ª  siempre se devuelve false
       mc=(a1,a2) -> false;
       System.out.println(mc.esMenor(5,2));
   }
}
```

Explicación paso a paso:

1. definimos la interface funcional.

```
@FunctionalInterface
interface MiComparador{
   public boolean esMenor(int a1, int a2);
}
```

2. Con una Lambda expresión, conseguimos un triple efecto:

- crea de alguna manera interna una clase que implementa el interface
- y por tanto sobreescribe el método de la interface funcional. El cuerpo de la lambda será el cuerpo del método.
- crea instancia de esa clase "que no vemos" que implementa el interface

```
MiComparador mc=(a1,a2) -> {System.out.println("comparando "+a1+" y "+ a2);return a1<a2;};
```

Observa cómo el compilador al ver que la expresión lambda se asocia a algo de tipo MiComparador puede inferir automáticamente que a1 y a2 son de tipo int

Demostramos que se creó una instancia por qué invocamos al método

```
boolean result = mc.esMenor(2, 5);
```

¿A que clase pertenece el método esMenor()? A una clase sin nombre generada internamente por la expresión lambda, lo que sí sabemos es que esa clase implementa el interface MiComparador.

Escribimos un ejemplo similar al anterior, **pero ahora sin utilizar expresiones lambda para observar el trabajo que hacen automáticamente las lambda**.

```
@FunctionalInterface
interface MiComparador{
    public boolean esMenor(int a1, int a2);
}

class MiComparadorConcreto1 implements MiComparador{

    @Override
    public boolean esMenor(int a1, int a2) {
       System.out.println("comparando "+a1+" y "+ a2);
       return a1<a2;
    }
}

class MiComparadorConcreto2 implements MiComparador{

    @Override
    public boolean esMenor(int a1, int a2) {
       return a1<a2;
    }
}
class MiComparadorConcreto3 implements MiComparador{
    @Override
    public boolean esMenor(int a1, int a2) {
       return false;
    }
}
class App {
     public static void main(String[] args) {
       MiComparador mc=new MiComparadorConcreto1();

       boolean result = mc.esMenor(2, 5);
       System.out.println(result);


       mc=new MiComparadorConcreto2();
       System.out.println(mc.esMenor(5,2));


       mc=new MiComparadorConcreto3();
       System.out.println(mc.esMenor(5,2));
    }
}
```

Expresiones lambda vs implementar interface expresamente
El ejemplo anterior, sin utilizar expresiones lambda, el código resultante es más laborioso. Cuando veamos clases anónimas se puede mejorar un poco pero, con todo, las expresiones lambda son superiores en concisión.
Además las expresiones lambda nos permitirán utilizar recursos de la programación funcional que iremos viendo a lo largo del curso.

Ejercicio U8_B2_E1

```
@FunctionalInterface
interface TransformadorNumerico{
    public int transformar(int x);
}
```

con una expresión lambda, sobre el interface anterior genera una implementación que por ejemplo devuelva el cuadrado de un número.

EL PAQUETE java.util.function
Ya sabemos que en java a una expresión lambda siempre tiene que corresponderle un interface funcional. No suele ser necesario escribir interfaces funcionales ya que para la mayor parte de los casos ya están escritas en el API, además si utilizamos estas interfaces nuestro código será más standard y legible.

Ejemplo: IntPredicate
En matemáticas, un predicado es una función que devuelve true o false dependiendo del conjunto de valores sobre los que actúa.
En programación funcional, los predicados son útiles para configurar "filtros". Para escribir filtros sobre valores enteros contamos en el API con IntPredicate. Si consultas el API observas que tienes que sobreescribir el siguiente método

`boolean`	**`test`**`(int value)` Evaluates this predicate on the given argument.

Escribimos un filtro de forma que de una lista de int nos quedamos con los pares. Además aprovechamos para usar la lambda como un parámetro de un método, observa pues cómo conseguimos en Java pasar una función como argumento.

```
import java.util.function.IntPredicate;

public class App{
  public static void main(String[] args){
    int[] lista = {1,2,3,4,5,6,7,8,9};

    System.out.println("Imprimir números int pares:");
    IntPredicate predicate=n-> n%2==0;
    eval(lista, predicate);
    //eval(lista, n-> n%2 == 0 );

   }

  public static void eval(int[] list, IntPredicate predicate) {
```

```
    for(int  n: list) {

      if(predicate.test(n)) {
        System.out.println(n + " ");
      }
    }
  }
}
```

Ejercicio U8_B2_E2: escribir el ejemplo anterior sin expresión λ , es decir, implementando el interface IntPredicate con una clase escrita por nosotros.

Ejercicio U8_B2_E3: usando de nuevo expresiones λ, generar salidas para los números > 5 y para "todos los números"

```
run:
Imprimir números mayores que 5:
6
7
8
9
Imprimir todos los números
1
2
3
4
5
6
7
8
9
BUILD SUCCESSFUL (total time: 0 seconds)
```

EJEMPLO IntConsumer

Este interfaz consta de un método accept() que acepta un int y no devuelve nada

```
import java.util.function.IntConsumer;
public class App{
   public static void main(String[] args) {
      IntConsumer ic = x->System.out.println(x);
      ic.accept(3);
   }
}
```

Puedes consultar en el API que accept tiene que devolver void y también en el API puedes consultar que efectivamente println() devuelve void

Ejercicio U8_B2_E4: Escribir el ejemplo anterior sin expresión λ , es decir, sobreescribiendo expresamente accept().

EJEMPLO IntSupplier

Este interfaz consta de un método getAsInt() sin parámetros y que devuelve un int

```
import java.util.function.IntSupplier;
public class App{
   public static void main(String[] args) {
      IntSupplier i = ()-> Integer.MAX_VALUE;
      System.out.println(i.getAsInt());
   }
}
```

Ejercicio U8_B2_E5: escribir el ejemplo anterior sin expresión λ

Ejercicio U8_B2_E6: hacer una implementación con expresión λ de forma que getAsInt() devuelva un número aleatorio entre 1 y 10

EJEMPLO DE USO DE MÉTODO DEFAULT

Recuerda que vimos que los interfaces desde java 8 pueden contener dos tipos de métodos no abstractos, es decir, con implementación: métodos static y métodos default.

El interface IntPredicate tiene un default negate()

```
import java.util.function.IntPredicate;
class App{
   public static void main(String[] args) {
      IntPredicate menorquecero = x-> x < 0;
      System.out.println(menorquecero.test(123));
      System.out.println(menorquecero.negate().test(123));
   }
}
```

Para entender la última sentencia recuerda que cuando hay varios métodos concatenados por punto se van evaluando de izquierda a derecha, por ejemplo, observa como primero se ejecuta trim() y sobre su resultado que es un Strin se ejecuta length()

```
class App{
   public static void main(String[] args) {
     String saludo=" Hola ";
     System.out.println(saludo.length());
     System.out.println(saludo.trim().length());
   }
}
```

observa que menorquecero.negate() devuelve un intpredicate que niega el predicado i y sobre ese nuevo predicado ejecutamos el método test()

el código anterior puedes escribirlo menos compacto para razonarlo

```
import java.util.function.IntPredicate;
class App{
   public static void main(String[] args) {
      IntPredicate menorquecero = x-> x < 0;
      System.out.println(menorquecero.test(123));
      IntPredicate menorqueceroNegado=menorquecero.negate();
      System.out.println(menorqueceroNegado.test(123));
   }
}
```

Por un mecanismo que maneja el compilador llamado Cierre de funciones menorqueceroNegado conserva internamente una referencia al predicado original

nos podríamos imaginar que el test() de menorquecerNegado es así

```
@Override
public boolean test(int value) {
   return !originalPredicate.test(value);
}
```

El anidamiento de lambdas lo resuelve el compilador pasando la referencia del objeto predicado original al nuevo predicado.De donde sale la referencia

originalPredicate sale de nuestro alcance ya que son mecanismo de compiladores, basta con entender el funcionamiento intuitivamente

Ejercicio U8_B2_E7: escribe el siguiente ejemplo sin expresiones lambda. consulta en el API el método default or().

```
import java.util.function.IntPredicate;
class App{
    public static void main(String[] args) {
        IntPredicate a = x-> x < 0;
        IntPredicate b = x-> x%2== 0;

        System.out.println(a.or(b).test(4));
    }
}
```

INTERFACES FUNCIONALES GENÉRICOS.
Ejemplo con Predicate

Anteriormente usamos IntPredicate. Si queremos usar un predicado genérico podemos usar Predicate<T>. La ventaja es que T puede ser Integer o de cualquier tipo, por tanto podemos escribir predicados sobre objetos de cualquier tipo.

Observa que el tipo genérico de Predicate se concreta en Integer debido al segundo parámetro de eval()

```
import java.util.function.Predicate;
public class App{
   public static void main(String args[]){
      int[] lista = {1, 2, 3, 4, 5, 6, 7, 8, 9};

      System.out.println("Imprimir todos los números:");
      Predicate<Integer> predicado = n -> true;
      eval(lista, predicado);

      System.out.println("Imprimir números pares:");
      eval(lista, n-> n%2 == 0 );


      System.out.println("Imprimir numeros mayores que 3:");
      eval(lista, n-> n > 3 );
   }

   public static void eval(int[] list, Predicate<Integer> predicate) {
      for(int  n: list) {

         if(predicate.test(n)) {
            System.out.println(n + " ");
         }
      }
   }
}
```

Ejercicio U8_B2_E8: Escribe el ejemplo anterior sin utilizar lambda expressions, es decir, creando directamente instancias del interfaz funcional.

Ejercicio U8_B2_E9: Haz un ejemplo similar al anterior con

String[] listaPalabras={"hola","adios","zorros","pimiento"};

Y con ejecución:

```
run:
Imprimir palabras con más de 5 caracteres:
zorros
pimiento
Imprimir palabras menores que chorizo
adios
BUILD SUCCESSFUL (total time: 0 seconds)
```

Ejercicio U8_B2_E10:

Un poco más elaborado, ahora el método eval es genérico de forma que funciona el siguiente main

```
public static void main(String args[]){
     String[] listaPalabras={"hola","adios","zorros","pimiento"};
     Integer[] listaNumeros={3,4,-5,6,-7};

     System.out.println("Imprimir palabras de más  de 5 car:");
     Predicate<String> predicado = s -> s.length()>5;
     eval(listaPalabras, predicado);

     System.out.println("Imprimir numeros positivos: ");
     eval(listaNumeros, i-> i>0 );

  }
```

que produce los resultados

```
Imprimir palabras de más  de 5 car:
zorros
pimiento
Imprimir numeros positivos:
3
4
6
```

Ejemplo con Function

Si consultas el API observaras que Function tienes dos tipos. Un tipo para un parámetro y otro tipo para el retorno.

Observa que el println() comentado es un error de compilación ya que por inferencia de tipos el X debe ser Integer(o poder hacer autoboxing).

```
import java.util.function.Function;
class App{
   public static void main(String[] args){
      // convert centigrade to fahrenheit
      Function<Integer,Double> centigradeToFahrenheitInt = x -> (double)((x*9/5)+32);

      // String to an integer
      Function<String, Integer> stringToInt = x -> Integer.valueOf(x);

      //System.out.println("Centigrade to Fahrenheit: "+centigradeToFahrenheitInt.apply(30.5));
      System.out.println("Centigrade to Fahrenheit: "+centigradeToFahrenheitInt.apply(30));
      System.out.println("String to Int: " + stringToInt.apply("4"));
   }
}
```

Observa que necesitamos el cast (double) en el primer ejemplo ya que la expresión

x*9/5 +32

se evalúa a int, pero ese valor hay que asociarlo a un Double para que por autoboxing
Double d1=2.0 es O.K.
Ya que
Double d2=2 es ERROR

Observa que
double d3=2 es O.K (cast automático de tipo primitivo int a tipo primitivo double).
Pero esto no es aplicable aquí ya que el segundo parámetro de FunctIon es Double no double

Ejercicio U8_B2_E11: Utilizando BiFunction puedes utilizar 3 tipos: 2 parámetros y un retorno. Escribe una expresión lambda que permita elevar un Double a un Integer. Para escribir el cuerpo de la función lambda puedes utilizar Math.pow(x,y)