Expresiones LAMBDA

Introducción

JAVA introduce los conceptos de **interfaces funcionales**, **expresiones lambdas** y **referencias a métodos** para facilitar la **creación de funciones**.

Originariamente las **clases anónimas** fueron la única manera de **crear objetos función en JAVA**:

Demasiada verborragia

```
Collections.sort(words, new Comparator<String>() {
    public int compare(String s1, String s2) {
        return Integer.compare(s1.length(), s2.length());
    }
});
System.out.println(words);
```

Ordena una lista de Strings de acuerdo a su longitud usando una **clase anónima.** Opera como una **función de comparación**

Las **clases anónimas** fueron una solución adecuada para implementar **patrones de diseño OO que requerían de objetos funciones.** Ejemplo es el **patrón Strategy**: la interface **Comparator** representa una **estrategia abstracta para ordenar** y la **clase anónima una estrategia concreta** para ordenar strings como el ejemplo de arriba.

Los objetos función en JAVA

```
interface MathOperation {
    double operation(double a, double b);
}
```

```
class Addition implements MathOperation {

    public double operation(double a, double b) {
        return a + b;
    }
}
```

Es muy largo!
Creamos una clase para algo muy simple

```
MathOperation addition = new MathOperation() {
    public double operation(double a, double b) {
        return a + b;
    }
};
```

Clases anónimas, también demasiado detalle!

```
MathOperation addition = (int a, int b) -> a + b;
```

Código conciso: expresión lambda

Interfaces funcionales

En JAVA 8 se formaliza la idea que las **interfaces con un único método** son especiales y requieren de un tratamiento especial. A estas interfaces se las denomina **interfaces funcionales**.

Es posible **crear instancias de interfaces funcionales** con **expresiones lambdas** o simplemente con **lambdas.**

Las **expresiones lambdas** cumplen una **función similar a las clases anónimas** pero son más **concisas**.

La expresión lambda es una instancia de la interface Comparator <String>

```
Collections.sort(words,
        (s1, s2) -> Integer.compare(s1.length(), s2.length()));
```

```
Collections.sort(words, new Comparator<String>() {
    public int compare(String s1, String s2) {
        return Integer.compare(s1.length(), s2.length());
    }
});
System.out.println(words);
```

El compilador deduce los tipos por contexto usando inferencia de tipos

En lambdas **no** están presentes los tipos de datos:
- el tipo del lambda: Comparator<String>
- el tipo de los parámteros s1 y s2: String
- el tipo del valor del retorno: int

El **compilador** para hacer inferencias usa la información aportada por los **tipos genéricos**.

Inferencia de tipos

El **compilador** para hacer **inferencias de tipos** usa la información aportada por los **tipos genéricos**.

Si se usan **raw types**, el **compilador no puede hacer inferencia de tipos** y es necesario explicitar los tipos de datos y hacer casting - > **versión más verbose.**

Versión concisa

```
public static void testLambdas_congenericos() {
    List<String> words = new ArrayList<>();
    words.add("hola");
    words.add("chau");
    words.add("genia");
    words.add("ok");
    words.add("1");
    Collections.sort(words, (s1, s2) -> Integer.compare(s1.length(), s2.length()));
}
```

Versión verbose

```
public static void testLambdas_singenericos() {
    List words = new ArrayList<>();
    words.add("hola");
    words.add("chau");
    words.add("genia");
    words.add("ok");
    words.add("1");
    Collections.sort(words,
      (String s1, String s2) -> Integer.compare(((String) s1).length(), ((String) s2).length()));

}
```

¿Qué es y cómo se escribe un lambda?

Una **expresión Lambda** en JAVA es una **función**, **toma parámetros de entrada y devuelve un valor.** No tiene nombre, no pertenece a una clase, se puede pasar como parámetro y ejecutarse bajo demanda.

Con lambdas el compilador realiza **inferencia de tipos** y deduce los tipos de datos del contexto.

```
Collections.sort(words,
        (s1, s2) -> Integer.compare(s1.length(), s2.length()));
```

Parámetros (podría ser vacío)

Cuerpo del Lambda (implementación de la interface)

Operador Flecha

Corolario: omitir los tipos de datos en los parámetros lambdas a menos que su presencia aporte claridad.

Lambdas son similares a las clases anónimas en cuanto a su función, sin embargo son mucho **más concisas**.

Ejemplos

Interfaces funcionales

```
interface MathOperation {
    int operation(int a, int b);
}
```

```
interface GreetingService {
    void sayMessage(String message);
}
```

```
GreetingService greetService1 = message
            -> System.out.println("Hola " + message);
```

Parámetros sin paréntesis ni declaración de tipos

```
GreetingService greetService2 = (message)
            -> System.out.println("Hola " + message);
```

Parámetros con paréntesis y sin declaración de tipos

```
MathOperation addition = (int a, int b) -> a + b;
```

Parámetros con declaración de tipos

```
MathOperation subtraction = (a, b) -> a - b;
```

Parámetros sin declaración de tipos

Las variables del tipo de las Interfaces almacenan una implementación de las mismas.

```
MathOperation multiplication = (int a, int b) -> {
    return a * b;
};
```

Bloque de código entre llaves y con sentencia return

```
MathOperation division = (int a, int b) -> a / b;
```

Sin sentencia return y sin llaves

Ejemplos

Las **variables del tipo de las interfaces** almacenan una **implementación** de las mismas.

Interfaces funcionales

```
interface MathOperation {
    int operation(int a, int b);
}
```

```
interface GreetingService {
    void sayMessage(String message);
}
```

```
int num1=Integer.valueOf(args[0]);
int num2=Integer.valueOf(args[1]);
MathOperation multiplicacion=(n1,n2)->n1*n2;
int resultado=multiplicacion.operation(num1, num2);
```

```
MathOperation suma=(n1,n2)->{ return n1+n2;};
int resultado2=suma.operation(num1, num2);
```

multiplicacion y **suma** son **funciones**, implementan la interface **MathOperation**.

```
GreetingService servicio= mensaje->System.out.println("Hola "+mensaje);
servicio.sayMessage("Labo 2021");
servicio.sayMessage("Labo 2022");
```

servicio es una **función** que implementan la interface **GreetingService**.

Referencias a métodos

En el caso que lo único que hace la **expresión lambda** es **invocar a un método** con los **parámetros pasados al lambda**, se pueden usar **referencias a métodos y se obtiene un código más conciso.**

```
public interface MyPrinter {
    public void print(String s);
}
```

Expresión Lambda que solo invoca a un método con parámetros:

```
MyPrinter myPrinter = (s) -> { System.out.println(s); };
```

Referencias a métodos:

```
MyPrinter myPrinter = System.out::println;

myPrinter.print("Hola");
```

Los dos puntos dobles le indican al compilador que es **una referencia a un método** y el método al que se hace referencia es el que viene después de **: :**

Referencias a métodos

comparingInt es un método de clase de la **interface Comparator** que se usa para construir **comparadores personalizados** para **enteros primitivos**.

```
Collections.sort(words, Comparator.comparingInt(String::length));
```

```
words.sort(comparingInt(String::length));
```

Referencias a métodos

```
class Persona {
public int getEdad() {
    return edad;
}

}
```

```
List<Persona> personas = new ArrayList<>();
personas.add(new Persona("Lucía", 30));
personas.add(new Persona("Diego", 25));
personas.add(new Persona("Juana", 35));
personas.sort(Comparator.comparingInt(Persona::getEdad));
```

Referencias a métodos

Crea un comparador que compara objetos de tipo **Persona** en función de su edad.
La lista de personas se ordena utilizando dicho comparador resultando en una lista ordenada por edad.

```
package enumerativos;
import java.util.function.DoubleBinaryOperator;
public enum Operation {
    PLUS("+", (x, y) -> x + y),
    MINUS("-", (x, y) -> x - y),
    TIMES("*", (x, y) -> x * y),
    DIVIDE("/", (x, y) -> x / y);
    private final String symbol;
    private final DoubleBinaryOperator op;

    Operation(String symbol, DoubleBinaryOperator op) {
        this.symbol = symbol;
        this.op = op;
    }

    public String toString() {
        return symbol;
    }

    public double apply(double x, double y) {
        return op.applyAsDouble(x, y);
    }
}
```

```
@FunctionalInterface
public interface DoubleBinaryOperator {
    double applyAsDouble(double left, double right);
}
```

Es una interface funcional del paquete **java.util.function** que representa a una función que toma 2 valores double como argumento y retorna un valor double como resultado.

Tipo enumerativo basado en lambdas

Es posible implementar el **comportamiento específico de cada constante enum** pasándole al constructor una expresión Lambda con dicho comportamiento.

El constructor guarda la expresión Lambda en la variable de instancia **op.**

El método **apply()** de **Operation** se usa para invocar al Lambda. No están más las sobreescrituras del método **apply()**.

Resumen

Las **expresiones lambdas** se utilizan para implementar **interfaces funcionales o interfaces con un único método abstracto**.

Los **lamdbas** son la mejor manera de representar **funciones pequeñas**.

Las expresiones lambdas **eliminan** la necesidad de usar **clases anónimas para interfaces función**.

El **tipo de la expresión Lambda**, los **tipos de los parámetros** y el **tipo del valor de retorno** no están necesariamente presentes en el código. El compilador usa **inferencia de tipos** para deducir los tipos del contexto.

Las **expresiones Lambdas** son esencialmente **objetos**.

Las **expresiones Lambdas no tienen estado**.

Una línea de código es ideal para un lambda y 3 es un máximo razonable.