

## REFERENCIAS A MÉTODOS (TAMBIÉN LLAMADO MÉTODOS DE REFERENCIA)

Referencias a métodos es una sintaxis que utiliza el operador `::` y generamos una sintaxis todavía más concisa que si la escribimos con su expresión lambda equivalente. No obstante, se puede usar en menos situaciones que la sintaxis de expresiones lambda.

La idea general es que cuando el cuerpo de un expresión lambda solo tiene una sentencia y esa sentencia es una llamada a un método, el compilador puede examinar la definición de ese método y por el contexto deducir o asumir los parámetros de la expresión lambda con lo que nos basta con indicar el nombre del método. Y para indicar que queremos utilizar este recurso debemos usar el operador `::`:

Aunque parezca de locos, los términos "Métodos de Referencia" y "Referencia a métodos" son equivalentes.

Seguimos "casi" textualmente:

<http://www.oracle.com/technetwork/es/articles/java/expresiones-lambda-a-pi-stream-java-2633852-esa.html>

### Métodos de Referencia.

Cuando la expresión lambda se compone de una sola sentencia e invoca algún método existente por medio de su nombre, existe la posibilidad de escribirla usando métodos de referencia, con lo cual se logra un código más compacto y fácil de leer. Existen tres tipos de métodos de referencia y uno adicional para constructores:

#### **Métodos Estáticos**

Cuando el método invocado es estático, la forma de escribir la expresión lambda usando métodos de referencia es la siguiente: **NombreClase::métodoEstático**, donde **NombreClase** es el nombre de la clase que contiene el método y **métodoEstático** es el nombre del método estático a invocar.

#### **Ejemplo:**

Vamos a usar un método static `sum()` de `Integer`. La clase `Integer` desde `JDK 8` tiene novedades para soportar la programación funcional. Una de ellas es que hay un método `sum()` static, de forma que `Integer.sum(3,2)` devolverá `5`. No parece en principio un método especialmente útil, pero se incorporó para usarlo en ciertos contextos de programación funcional que veremos más adelante. Este método lo vamos a usar como

cuerpo de una **expresión lambda** para **implementar** un **interface funcional** **BinaryOperator** que obliga a implementar el método

***apply(T,T):T***, es decir, definimos una función que recibe dos parámetros del mismo tipo y retorna un resultado del mismo tipo de sus parámetros:

```
BinaryOperator<Integer> sumar = (a,b) -> Integer.sum(a,b);
```

como el cuerpo de la expresión lambda tiene una **única instrucción** y dicha instrucción consiste en **invocar a un método static** podemos **sustituir la expresión lambda por la sintaxis referencia a método**:

```
BinaryOperator<Integer> sumar = Integer::sum;
```

```
import java.util.function.BinaryOperator;
class App{
    public static void main(String[] args){
        BinaryOperator<Integer> sum1 = (a,b) -> Integer.sum(a,b);
        BinaryOperator<Integer> sum2= Integer::sum;
        System.out.println(sum1.apply(2,3));
        System.out.println(sum2.apply(2,3));
    }
}
```

Nos fijamos que al escribir `BinaryOperator<Integer> sum2=Integer::sum;` el compilador deduce que:

1. Queremos una expresión lambda con una única sentencia que es la llamada al método **Integer.Sum()**.
2. Observando la parte izquierda de la expresión

```
BinaryOperator<Integer> sum2
```

El compilador deduce que la expresión lambda tiene que implementar el método ***apply(Integer,Integer):Integer***

3. Por lo tanto, el cuerpo de `apply()` será:
  - una llamada al método `Integer.sum`
  - el método **Integer.sum** necesita dos **Integer** y al no especificarlos asume que son los parámetros de la llamada al `apply()`

Fíjate en cambio, que si nos salimos del caso concreto en el que hay un mapeo o correspondencia automática entre los argumentos de `apply` y los del método de su cuerpo, no habrá equivalencia entre lambda y referencia a métodos, por ejemplo, ahora `sum1` y `sum2` no son equivalentes:

```
import java.util.function.BinaryOperator;
class App{
    public static void main(String[] args){
        BinaryOperator<Integer> sum1 = (a,b) -> Integer.sum(a+10,b+20);
        BinaryOperator<Integer> sum2=Integer::sum;
        System.out.println(sum1.apply(2,3));
    }
}
```

```

        System.out.println(sum2.apply(2,3));
    }
}

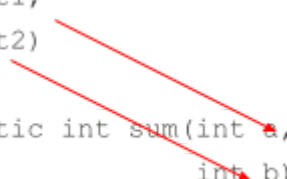
```

por lo tanto, la referencia a métodos establece una correspondencia concreta entre argumentos de `apply()` y argumentos de `sum()`,

```

T apply(T t1,
        T t2)
public static int sum(int a,
                     int b)

```



si no nos interesa esta correspondencia “por defecto”, no podemos usar referencia a métodos

Por si causa confusión usar un método static del API, añadimos al tutorial de oracle un ejemplo con un método static escrito por nosotros

```

import java.util.function.BinaryOperator;

class App {

    static int restar(int x, int y){
        return x -y;
    }

    public static void main(String[] args) {
        BinaryOperator<Integer> resta1 = (a, b) -> a-b;
        BinaryOperator<Integer> resta2 = (a, b) -> restar(a,b); // restar(a,b) es idem a App.restar(a,b)
        BinaryOperator<Integer> resta3 = App::restar;

        System.out.println(resta1.apply(3, 2));
        System.out.println(resta2.apply(4, 2));
        System.out.println(resta3.apply(5, 2));
    }
}

```

### **Métodos de instancia de un objeto(para métodos no static)**

Cuando contamos con una referencia a un objeto y deseamos invocar alguno de sus métodos de instancia dentro de la expresión lambda, la forma en la que la escribiríamos usando métodos de referencia es la siguiente: **RefObjeto::métodoInstancia**, donde **RefObjeto** es la referencia al objeto y **métodoInstancia** es el nombre del método a invocar. Por ejemplo, la clase `java.lang.System` tiene una referencia a un objeto de tipo

`java.io.PrintStream` denominada `out`, usaremos esa referencia en el siguiente ejemplo. Es una referencia static y por tanto el nombre de la referencia será `System.out`

Primero veamos cómo se escribiría usando una expresión lambda:

```
Consumer<Integer> print = (a) -> System.out.println(a);
```

Y ahora usando métodos de referencia:

```
Consumer<Integer> print = System.out::println;
```

Nótese que **la referencia al objeto la tenemos en `System.out`** e invocamos su método de instancia `+println(int):void`

*Podemos probarlo ...*

```
import java.util.function.Consumer;

public class App{
    public static void main(String[] args){
        Consumer<Integer> print = a -> System.out.println(a);
        print.accept(3);

        Consumer<Integer> print2 =System.out::println;
        print2.accept(456);
    }
}
```

Vamos a ponernos en el pellejo del compilador. Nos llega `System.out::println` lo cual quiere decir que el compilador **debe inferir el parámetro de `println()`**. Si observo la parte izquierda de la expresión veo que **se trata de un `Consumer`**, por lo tanto la expresión realmente implementa el método `accept(T t):void`, lo único lógico por tanto es que `println()` imprima `t`. Pero `T` es genérico, y en nuestra instrucción es un `Integer`, por lo tanto estaremos imprimiendo un `Integer` y estamos invocando a la versión de `println(Object o)`

```
println(Object x)
Prints an Object and then terminate the line.
```

La **correspondencia entre el parámetro de método del interface funcional y del `println()` tiene que ser exacta**, así por ejemplo observamos que

```
System.out.println(a+x);
```

no se puede escribir con referencia a métodos

```
import java.util.function.Consumer;

public class App{
    public static void main(String[] args){
        int x=3;
        Consumer<Integer> print = a -> System.out.println(a+x);
        print.accept(3);

        Consumer<Integer> print2 =System.out::println;
        print2.accept(3);
    }
}
```

De nuevo, añadimos al tutorial de oracle un ejemplo en el que usa una referencia *op* de una clase definida por nosotros.

```
import java.util.function.BinaryOperator;
class Operacion{
    int restar(int x, int y){
        return x -y;
    }
}
class App {

    public static void main(String[] args) {
        Operacion op= new Operacion();
        BinaryOperator<Integer> resta1 = (a, b) -> a-b;
        BinaryOperator<Integer> resta2 = (a, b) -> op.restar(a,b);
        BinaryOperator<Integer> resta3 = op::restar;

        System.out.println(resta1.apply(3, 2));
        System.out.println(resta2.apply(4, 2));
        System.out.println(resta3.apply(5, 2));
    }
}
```

Observa que el método `restar()` que usamos es coherente con `apply()` de `BinaryOperator`, es decir, se le pasan dos parámetros del mismo y devuelve un valor del mismo tipo. Si por ejemplo, `restar()` tuvieran 3 parámetros no se podría aplica a `BinaryOperator`

### **EJERCICIO U8\_B4\_E1:**

Basándose en `Consumer<String>`, escribe un "hola mundo " utilizando el operador `::`

### **Métodos de instancia de algún tipo**

Este caso es parecido al anterior, pero se diferencia en que no contamos con una referencia a un objeto (como `System.out` o como `op` de ejemplos anteriores) , **sólo conocemos su tipo** y podríamos escribir la expresión lambda de la siguiente forma: **Tipo::métodoInstancia**, donde **Tipo** es la clase y **métodoInstancia** es el nombre del método de instancia a invocar.

Ejemplo: En la siguiente expresión lambda no se usan referencias que no sean las de los parámetros como `System.out` u `op` de ejemplos anteriores. Es cierto que `s` es una referencia pero es un parámetro. De `s` sabemos que es de tipo `String`

```
s->s.toUpperCase()
```

```
import java.util.function.Function;
class App {
    public static void main(String[] args) {
        //Function<String,String> mayusculador=s->s.toUpperCase();
        Function<String,String> mayusculador=String::toUpperCase;
        System.out.println(mayusculador.apply("hola"));
    }
}
```

El siguiente ejemplo define un `java.lang.Comparator` que nos permitirá comparar cadenas sin importar si son mayúsculas/minúsculas.

Primero veamos cómo se escribiría usando una **expresión lambda**:

```
Comparator<String> upper = (a, b) -> a.compareToIgnoreCase(b);
```

cuando decimos que **no tenemos una referencia a un objeto** nos referimos que en la instrucción **no hay una referencia a un objeto que no sea un parámetro** ya que los parámetros también pueden ser referencias pero nos referimos a una referencia que no es parámetro.

Y ahora usando **métodos de referencia**:

```
Comparator<String> upper = String::compareToIgnoreCase;
```

Nótese que en este caso no contamos con la referencia a un objeto como tal, pero sabemos que queremos comparar objetos de tipo String y con eso es suficiente para que podamos escribir nuestra expresión lambda usando métodos de instancia de algún tipo.

Observa que el compilador:

1. al ver el operador `::` sabe que va a tener que inferir cosas basándose en el método `compareToIgnoreCase`
2. En este caso el método no es static, ni aprecia referencia como ocurría con `out` en `System.out::println`. Lo que se aprecia es su tipo es String `String::compareToIgnoreCase`, se examina la definición de este método en la clase String y se aprecia que precisa dos parámetros de tipo String que se harán corresponder con los de llamada a `compare()`

un ejemplo para ejecutar

```
import java.util.Comparator;
class App {
    public static void main(String[] args) {
        Comparator<String> upper = (a, b) -> a.compareToIgnoreCase(b);
```

```

        System.out.println(upper.compare("ana", "Bea"));
        Comparator<String> upper2 = String::compareToIgnoreCase;
        System.out.println(upper2.compare("ana", "Bea"));
    }
}

```

**ojo:** Al ver el nombre de la clase antes de los :: podríamos pensar que estamos ante el primer tipo de método referencia (referencia a método static) pero compareToIgnoreCase() NO es un método static, si no un método instancia, por eso estamos ante otro caso distinto de "método referencia".

**EJERCICIO U8\_B4\_E2:** Añade al siguiente ejemplo un "imprimir3" de forma que la forma de implementar el interfaz funcional sea con el operador ::

```

import java.util.function.Function;

public class App{
    public static void main(String[] args){
        imprimir(new Function<String, String>(){
            @Override
            public String apply(String s){
                return s.toLowerCase();
            }
        }, "imprimir1:STRING TO LOWERCASE");

        imprimir(s -> s.toLowerCase(), "imprimir2: STRING TO LOWERCASE");

        //añadir impresion con tipo::
    }

    public static void imprimir(Function<String, String> function, String s){
        System.out.println(function.apply(s));
    }
}

```

## **REFERENCIA A MÉTODOS CONSTRUCTORES**

Nos apartamos en este caso de los ejemplos de la url de oracle mencionada al principio del boletín

Para el caso de constructores podemos escribir expresiones lambda como métodos de referencia de la siguiente forma: **Clase::new**, dónde **Clase** es la clase que deseamos instanciar y **new** es la palabra reservada que indica que el método que queremos referenciar es un constructor.

Ejemplo:

Queremos referenciar a un constructor que le pasamos dos enteros y nos devuelve un objeto de la clase Punto. Necesitamos pues usar dos parámetros y un valor de retorno y por lo tanto usamos el interface funcional **BiFunction**

```

import java.util.function.BiFunction;

class Punto {
    private int x;
    private int y;

    public Punto(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }
}

public class App{
    public static void main(String[] args) {
        //BiFunction<Integer, Integer, Punto> constructor = (x, y) -> new Punto(x, y);
        // Referencia al constructor de la clase Punto usando BiFunction
        BiFunction<Integer, Integer, Punto> constructor = Punto::new;

        // Crear un objeto Punto con las coordenadas (3, 4)
        Punto punto = constructor.apply(3, 4);
        System.out.println("Coordenadas: (" + punto.getX() + ", " + punto.getY() + ")");
    }
}

```

### **EJERCICIO U8\_B4\_E3:**

Añade al ejemplo anterior, a la clase Punto un constructor copia y al main la creación de un punto copia. Observa que lógicamente, para usar el constructor copia precisaré un objeto Function. Y observa como aunque Punto tenga varios constructores el compilador por el contexto utiliza el apropiado.