### EJERCICIO TUTORIAL DE LAMBDA

### Proyecto

Como el objetivo de este tutorial es centrarnos en el uso de Stream y Lambdas, he creado el proyecto más sencillo que se me ha ocurrido. Un carrito de la compra que nos devuelve el número de productos

y el precio total de ellos. A continuación dejo el código:

##### CarritoDeLaCompra.java

Java

| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | import java.util.Collection;    public class CarritoDeLaCompra {    private Collection<Integer> precios;    public CarritoDeLaCompra(Collection<Integer> precios) {    this.precios = precios;  }    public int calcularPrecioTotal() {    int precioTotal = 0;    for(Integer precio : precios){    precioTotal += precio;    }  return precioTotal;  }    public int contarNumeroProductos() {    return precios.size();  }  } |
| --- | --- |

##### CarritoBuilder.java

Java

| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34 | import java.util.ArrayList;    public class CarritoBuilder {    ArrayList<Integer> precios = new ArrayList<Integer>();    public CarritoBuilder(int size){    for(int i = 0; i < size; i++){    Double random = Math.random()\*100+1;  precios.add(random.intValue());  }  }    public CarritoBuilder(int size, int value){    for(int i = 0; i < size; i++){    precios.add(value);  }  }    public CarritoDeLaCompra build(){    return new CarritoDeLaCompra(this.precios);  }    public CarritoBuilder add(Integer nuevoValor){    precios.add(nuevoValor);  return this;  }  } |
| --- | --- |

También, como buenos seguidores de la filosofía TDD que somos en [Autentia](https://www.autentia.com/), he creado los test que me permitan probar que lo que realizo es totalmente funcional.

Os dejo también el código de los test:

##### CarritoDeLaCompraTest.java

Java

| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | import com.autentia.CarritoBuilder;  import com.autentia.CarritoDeLaCompra;  import org.junit.Assert;  import org.junit.Test;    public class CarritoDeLaCompraTest {    @Test  public void shouldReturnTheCountOfAllItems() throws Exception {    CarritoBuilder builder = new CarritoBuilder(30);  CarritoDeLaCompra carritoDeLaCompra = builder.build();  Assert.assertEquals(30, carritoDeLaCompra.contarNumeroProductos());  }    @Test  public void shouldCalculateTotalPrice() throws Exception {    CarritoBuilder builder = new CarritoBuilder(60,5);  CarritoDeLaCompra carritoDeLaCompra = builder.build();  Assert.assertEquals(300, carritoDeLaCompra.calcularPrecioTotal());    }  } |
| --- | --- |

Con el objetivo de en los siguientes tutoriales realizar pruebas de rendimiento, en vez de modificar los métodos voy a crear nuevos. **AVISO: los tiempos de ejecución de los test que aparecen en esta primera parte no son representativos.**

### 4. Lambdas y Stream

Empecemos con nuestras primeras líneas de programación funcional. Para empezar, creamos un nuevo método, en mi caso el método **calcularPrecioTotalLambda()** cuyo código es el siguiente:

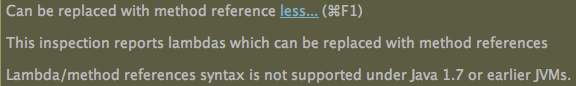
##### calcularPrecioTotalLambda()

Java

| 1  2  3  4 | public int calcularPrecioTotalLambda() {  int precioTotal = this.precios.stream().mapToInt(precio -> precio.intValue()).sum();  return precioTotal;  } |
| --- | --- |

Como podéis observar, hacemos uso del stream, el cual nos recogerá cada objeto de la Collection **precios** (*this.precios.stream()*) y lo mapeará a Integers (*mapToInt()*).

La lambda que pasamos por parámetro a *mapToInt* nos dará un warning.

[](https://www.adictosaltrabajo.com/wp-content/uploads/2016/05/Captura-de-pantalla-2016-05-10-a-las-15.26.05.png)

Si pulsamos el atajo **[Alt + Enter]** la solución será sustituir la lambda por una referencia a método. De tal manera que el código quedaría así:

##### calcularPrecioTotalRefMethod()

Java

| 1  2  3  4  5  6 | public int calcularPrecioTotalLambda() {  int precioTotal = this.precios.stream()  .mapToInt(Integer::intValue)  .sum();  return precioTotal;  } |
| --- | --- |

Las dos formas hacen lo mismo: sacar de cada Integer su valor. La diferencia es la llamada que hacemos. En el primer caso (**i -> i.intValue()**) llamamos al método intValue de cada Integer.

En el segundo caso (**Integer::intValue**) hacemos uso del método a través de una referencia (incluidas en Java 8). Si tuviera que decantarme por una forma, quizás la más clara sea

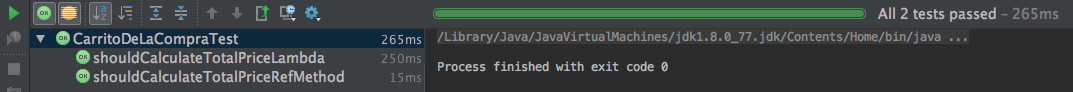
**i -> i.intValue()**, pero sin duda la más correcta es **Integer::intValue**.

Ahora creamos los test de estos dos nuevos métodos y vemos que todo funciona:

##### CarritoDeLaCompraTest.java

Java

| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | @Test  public void shouldCalculateTotalPriceLambda() throws Exception {    CarritoBuilder builder = new CarritoBuilder(60, 5);  CarritoDeLaCompra carritoDeLaCompra = builder.build();  Assert.assertEquals(300, carritoDeLaCompra.calcularPrecioTotalLambda());  }    @Test  public void shouldCalculateTotalPriceRefMethod() throws Exception {    CarritoBuilder builder = new CarritoBuilder(60,5);  CarritoDeLaCompra carritoDeLaCompra = builder.build();  Assert.assertEquals(300, carritoDeLaCompra.calcularPrecioTotalRefMethod());  } |
| --- | --- |



### 5. Filter

Hasta aquí ha sido un primer contacto muy básico con los Streams, las Lambdas y las Referencias a métodos. Pretendía dar a entender las similitudes entre un bucle for y un stream para ahora

entrar más en detalle en las posibilidades que nos ofrece Java 8.

Obviamente con las opciones que tenemos ahora en *CarritoDeLaCompra* poco podríamos hacer, por lo que, debemos extender funcionalidades. La primera va a ser un detector de descuentos

que comprueba si algún precio es mayor o igual que el que pasamos por parámetro y, por cada coincidencia, descuenta un 5% :

##### calcularDescuentoTotal()

Java

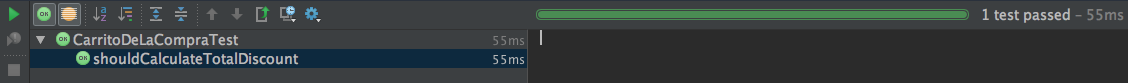
| 1  2  3  4  5  6  7  8  9  10  11 | public long calcularDescuentoTotal(int cantidadConDescuento){    long descuentoTotal = 0;    for(Integer precio : precios){  if(precio >= cantidadConDescuento){  descuentoTotal += (cantidadConDescuento\*5)/100;  }  }  return descuentoTotal;  } |
| --- | --- |

El método de testeo para este nuevo método:

##### Test

Java

| 1  2  3  4  5  6  7  8 | @Test  public void shouldCalculateTotalDiscount() throws Exception {    CarritoBuilder builder = new CarritoBuilder(20,100);  CarritoDeLaCompra carritoDeLaCompra = builder.build();  Assert.assertEquals(100, carritoDeLaCompra.calcularDescuentoTotal(100));    } |
| --- | --- |

[](https://www.adictosaltrabajo.com/wp-content/uploads/2016/05/Captura-de-pantalla-2016-05-12-a-las-19.17.38.png)

Ahora, vamos a hacer un nuevo método que use Stream y Lambda:

##### calcularDescuentoTotalLambda()

Java

| 1  2  3  4  5  6  7  8  9  10  11  12 | public long calcularDescuentoTotalLambda(int cantidadConDescuento){    long descuentoTotal = 0;    Long numeroDeDescuentos = this.precios.stream()  .filter(precio -> precio.intValue() >= cantidadConDescuento)  .count();    descuentoTotal = (cantidadConDescuento\*5/100)\*numeroDeDescuentos;    return descuentoTotal;  } |
| --- | --- |

Volvemos a tener un ejemplo para comparar entre prog. imperativa y prog. funcional, ahora tratemos de entenderlo. Como podéis observar, he añadido un nuevo método llamado **filter()**.

Dicho método recibe como parámetro un [predicado](https://www.adictosaltrabajo.com/tutoriales/expresiones-lambda-con-java-8/#034). Para entenderlo, no hay más que traducirlo

al español: *filtro*.

Nos va a servir de filtro con la condición que nosotros le pongamos, aplicando el siguiente método únicamente sobre los valores que la cumplan. En mi caso, la

condición es que cada valor debe ser mayor o igual a la cantidad pasada por parámetro para que se cuente. Al final, tendremos la cuenta de todos los precios que tienen descuento.

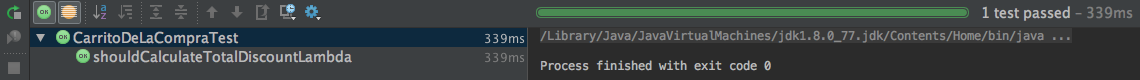
Una vez contados aplicamos unas matemáticas básicas para sacar el descuento total.

Ahora creamos el test y lo ejecutamos para comprobar si es correcto.

##### shouldCalculateTotalDiscountLambda()

Java

| 1  2  3  4  5  6  7  8 | @Test  public void shouldCalculateTotalDiscountLambda() throws Exception {    CarritoBuilder builder = new CarritoBuilder(20,100);  CarritoDeLaCompra carritoDeLaCompra = builder.build();  Assert.assertEquals(100, carritoDeLaCompra.calcularDescuentoTotalLambda(100));    } |
| --- | --- |

[](https://www.adictosaltrabajo.com/wp-content/uploads/2016/05/Captura-de-pantalla-2016-05-12-a-las-19.38.17.png)

### 6. anyMatch()

Para no repetir números muy extensos, he creado dos constantes que usaré a lo largo de los próximos test.

Java

| 1  2 | private final Long TOTAL\_SIZE = 20000000L;  private final Long NUMBER\_ADD = 1000000L; |
| --- | --- |

Tambien he añadido un método a CarritoBuilder llamado addMultiple() que me permita añadir mucho valores para dejar el número negativo en mitad del array completo.

##### addMultiple()

Java

| 1  2  3  4  5  6  7  8  9 | public CarritoBuilder addMultiple(int size, int value){    for(int i = 0; i < size; i++){    precios.add(value);  }    return this;  } |
| --- | --- |

Vamos a seguir añadiendo funcionalidades a nuestro carrito. El nuevo método que he creado si detecta un valor erróneo devuelve true. Los valores erróneos serán

todos aquellos valores que lleguen como un *número negativo*. A continuación muestro el código del nuevo método:

##### detectarError()

Java

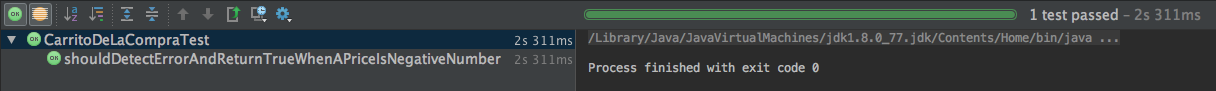
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | public boolean detectarError() {    boolean negativeFind = false;    for (Long precio : precios) {    if (precio < 0) {    negativeFind = true;  }  }    return negativeFind;  } |
| --- | --- |

Como es evidente, ahora necesitamos la clase de test para este método:

##### shouldDetectErrorAndReturnTrueWhenAPriceIsNegative()

Java

| 1  2  3  4  5  6  7  8  9  10 | @Test  public void shouldDetectErrorAnThrowRuntimeExceptionWhenAPriceIsNegative(){    CarritoBuilder builder = new CarritoBuilder(TOTAL\_SIZE,NUMBER\_ADD);  builder.add(-1L);  builder.addMultiple(TOTAL\_SIZE,NUMBER\_ADD);  CarritoDeLaCompra carritoDeLaCompra = builder.build();  Assert.assertTrue(carritoDeLaCompra.detectarError());    } |
| --- | --- |



A continuación dejo el código del ejemplo usando un nuevo método: *anyMatch()*

##### detectarErrorAnyMatch()

Java

| 1  2  3  4 | public boolean detectarErrorAnyMatch() {    return this.precios.stream().anyMatch(precio -> precio.intValue() < 0);  } |
| --- | --- |

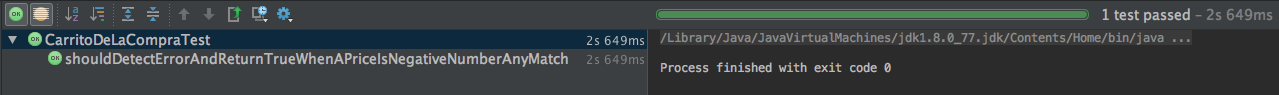
**anyMatch()**, al igual que **filter()**, recibe un [predicado](https://www.adictosaltrabajo.com/tutoriales/expresiones-lambda-con-java-8/#034). En este caso devuelve true si encuentra *algún* precio **negativo**.

Creamos el test y lo ejecutamos para comprobar que funciona correctamente:

##### shouldDetectErrorAndReturnTrueWhenAPriceIsNegativeAnyMatch()

Java

| 1  2  3  4  5  6  7  8  9  10 | @Test  public void shouldDetectErrorAndReturnTrueWhenAPriceIsNegativeAnyMatch(){    CarritoBuilder builder = new CarritoBuilder(TOTAL\_SIZE,NUMBER\_ADD);  builder.add(-1L);  builder.addMultiple(TOTAL\_SIZE,NUMBER\_ADD);  CarritoDeLaCompra carritoDeLaCompra = builder.build();  Assert.assertTrue(carritoDeLaCompra.detectarErrorAnyMatch());    } |
| --- | --- |



### 7. findAny() e isPresent()

Para estos nuevos métodos voy a usar el mismo ejemplo de antes. Más adelante veremos si realmente existen diferencias más allá de la sintaxis de cada forma. Tomamos como referencia

el detectarError() y realizamos el siguiente método:

Java

| 1  2  3  4  5  6 | public boolean detectarErrorFindAny() {    return this.precios.stream().filter(precio -> precio.intValue() < 0)  .findAny()  .isPresent();  } |
| --- | --- |

En este ejemplo hacemos uso de los métodos **findAny()** e **isPresent()**. Si traducimos al español quedaría como «…y encuentra alguno. ¿Está presente?». Es decir, findAny() nos devuelve

un [Optional](https://www.adictosaltrabajo.com/tutoriales/optional-java-8/) cuando se cumple la condición de filter(). Este Optional tiene como método isPresent() el cual si encuentra una coincidencia devolverá **true**. Hay que aclarar que esta forma recorre **todo** el stream.

Habría una forma de evitar que recorra todo el stream y es usando el método findFirst(), es decir, «encuentra el primero».

##### shouldDetectErrorAndReturnTrueWhenAPriceIsNegativeNumberFindAny()

Java

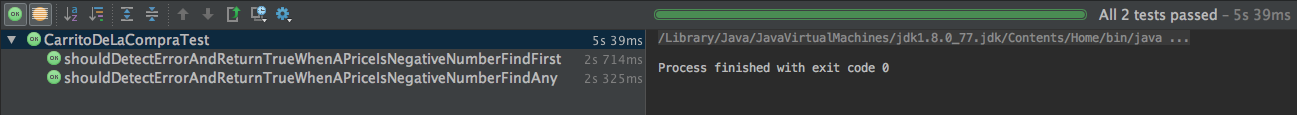
| 1  2  3  4  5  6  7  8  9  10 | @Test  public void shouldDetectErrorAndReturnTrueWhenAPriceIsNegativeNumberFindAny(){    CarritoBuilder builder = new CarritoBuilder(TOTAL\_SIZE,NUMBER\_ADD);  builder.add(-1L);  builder.addMultiple(TOTAL\_SIZE,NUMBER\_ADD);  CarritoDeLaCompra carritoDeLaCompra = builder.build();  Assert.assertTrue(carritoDeLaCompra.detectarErrorFindAny());    } |
| --- | --- |

##### shouldDetectErrorAndReturnTrueWhenAPriceIsNegativeNumberFindFirst()

Java

| 1  2  3  4  5  6  7  8  9  10 | @Test  public void shouldDetectErrorAndReturnTrueWhenAPriceIsNegativeNumberFindFirst(){    CarritoBuilder builder = new CarritoBuilder(TOTAL\_SIZE,NUMBER\_ADD);  builder.add(-1L);  builder.addMultiple(TOTAL\_SIZE,NUMBER\_ADD);  CarritoDeLaCompra carritoDeLaCompra = builder.build();  Assert.assertTrue(carritoDeLaCompra.detectarErrorFindFirst());    } |
| --- | --- |

Ejecutamos los test para comprobar su correcto funcionamiento.



Existen [muchas otras funcionalidades](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html) pero el objetivo de este post no es explicarlas todas.

### 8. parallelStreams()

En este último punto, voy a hacer uso de *parallelStreams*. Para conocer los parallelStream os recomiendo dos enlaces:

* [Documentación oficial de Oracle](https://docs.oracle.com/javase/tutorial/collections/streams/parallelism.html)
* [Vídeo](https://www.youtube.com/watch?v=r-eF5r2pN-M) de [David Gómez](https://www.adictosaltrabajo.com/author/dgomezg/)

Cogiendo los tres ejemplos de detectar precios nulos, he creado los siguientes métodos usando un parallelStream:

##### detectarErrorXXXParallel()

Java

| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | public boolean detectarErrorAnyMatchParallel() {  return this.precios.parallelStream().anyMatch(precio -> precio.intValue() < 0);  }    public boolean detectarErrorFindAnyParallel() {  return this.precios.parallelStream().filter(precio -> precio.intValue() < 0)  .findAny()  .isPresent();  }    public boolean detectarErrorFindFirstParallel() {    return this.precios.parallelStream().filter(precio -> precio.intValue() < 0)  .findFirst()  .isPresent();  } |
| --- | --- |

##### shouldDetectErrorAnThrowRuntimeExceptionWhenAPriceIsNegativeXXXParallel()

Java

| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | @Test  public void shouldDetectErrorAndReturnTrueWhenAPriceIsNegativeAnyMatchParallel(){    CarritoBuilder builder = new CarritoBuilder(TOTAL\_SIZE,NUMBER\_ADD);  builder.add(-1L);  builder.addMultiple(TOTAL\_SIZE,NUMBER\_ADD);  CarritoDeLaCompra carritoDeLaCompra = builder.build();  carritoDeLaCompra.detectarErrorAnyMatchParallel();    }    @Test  public void shouldDetectErrorAndReturnTrueWhenAPriceIsNegativeNumberFindAnyParallel(){    CarritoBuilder builder = new CarritoBuilder(TOTAL\_SIZE,NUMBER\_ADD);  builder.add(-1L);  builder.addMultiple(TOTAL\_SIZE,NUMBER\_ADD);  CarritoDeLaCompra carritoDeLaCompra = builder.build();  Assert.assertTrue(carritoDeLaCompra.detectarErrorFindAnyParallel());    }    @Test  public void shouldDetectErrorAndReturnTrueWhenAPriceIsNegativeNumberFindFirstParallel(){    CarritoBuilder builder = new CarritoBuilder(TOTAL\_SIZE,NUMBER\_ADD);  builder.add(-1L);  builder.addMultiple(TOTAL\_SIZE,NUMBER\_ADD);  CarritoDeLaCompra carritoDeLaCompra = builder.build();  Assert.assertTrue(carritoDeLaCompra.detectarErrorFindFirstParallel());    } |
| --- | --- |

Lanzamos los test por separado:

