Java Currying ,es una de las técnicas más habituales cuanto trabajamos con Java 8 y la programación funcional. Sin embargo siempre es un concepto difícil de explicar cuando estamos empezando . Vamos a intentar acercarnos a él a través de algunos ejemplos muy sencillos. Para ello nos vamos a centrar en un sencillo bucle for que imprime una tabla de multiplicar en pantalla.

```
package com.arquitecturajava;

public class Principal1 {

    public static void main(String[] args) {

        int numero = 5;

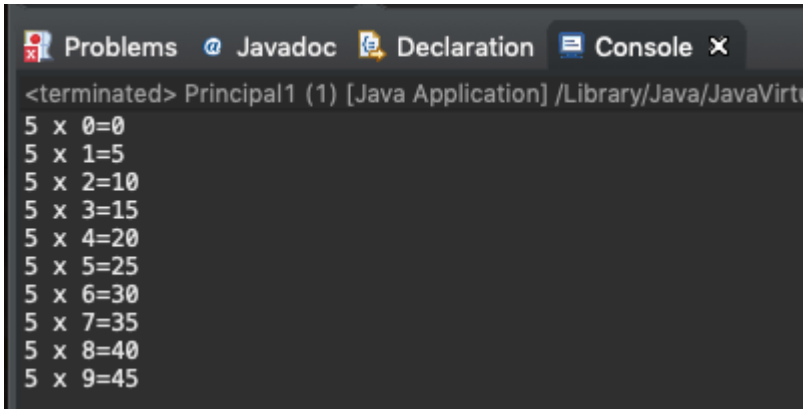
        for (int i = 0; i < 10; i++) {

            System.out.format("%d x %d=%d \n", numero, i, numero * i);
        }

    }

}
```

Se trata de un ejemplo que simplemente nos imprime una tabla de multiplicar en la consola no tiene mucho más.



Si nosotros queremos imprimir otra tabla de multiplicar como puede ser la tabla de multiplicar del 7 la operación es parecida:

```
package com.arquitecturajava;

public class Principal2 {

    public static void main(String[] args) {

        int numero = 5;

        for (int i = 0; i < 10; i++) {

            System.out.format("%d x %d=%d \n", numero, i, numero * i);
        }

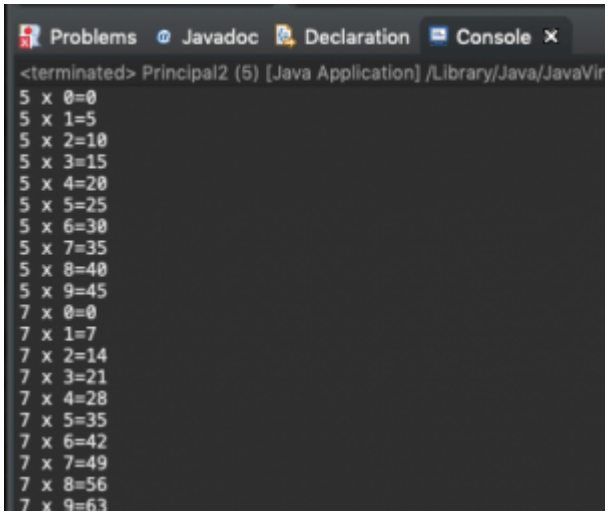
        numero = 7;

        for (int i = 0; i < 10; i++) {

            System.out.format("%d x %d=%d \n", numero, i, numero * i);
        }
    }
```

```
}
```

El resultado serán dos tablas de multiplicar:



Es evidente que el código es poco reutilizable lo más lógico es crear una función que parametrice el número:

```
package com.arquitecturajava;

public class Principal3 {

    public static void main(String[] args) {

        int numero = 5;

        tablaMultiplicar(numero);

        numero = 7;

        tablaMultiplicar(numero);

    }
```

```
    private static void tablaMultiplicar(int numero) {
        for (int i = 0; i < 10; i++) {

            System.out.format("%d x %d=%d \n", numero, i, numero * i);
        }
    }

}
```

El resultado es idéntico:



El concepto de Java Currying

Es aquí donde las cosas pueden comenzar a complicarse , imaginémonos que la tabla de multiplicar se puede imprimir en varios formatos, es decir soporta otro formato diferente a la hora de presentar la información en pantalla. Vamos a verlo:

```
package com.arquitecturajava;

public class Principal3 {

    public static void main(String[] args) {
```

```
        int numero = 5;

        tablaMultiplicar(numero);

        tablaMultiplicarTexto(numero);
    }

    private static void tablaMultiplicar(int numero) {
        for (int i = 0; i < 10; i++) {

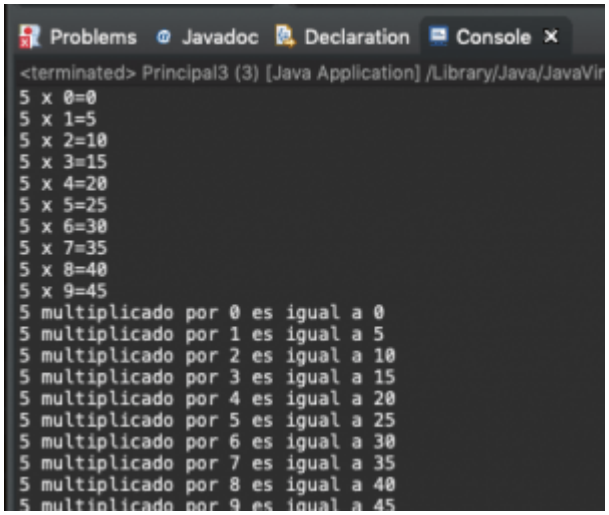
            System.out.format("%d x %d=%d \n", numero, i, numero * i);
        }
    }

    private static void tablaMultiplicarTexto(int numero) {
        for (int i = 0; i < 10; i++) {

            System.out.format("%d multiplicado por %d es igual a %d
\n", numero, i, numero * i);
        }
    }

}
```

El resultado es :



¿Es el código correcto? . En principio si que lo parece tenemos dos funciones una que nos muestra una multiplicación con un formato sencillo y otra que nos muestra una multiplicación con un formato más complejo. Todo ok, lamentablemente si pensamos un poco más a detalle en la funcionalidad que estamos implementando nos daremos cuenta que el bucle for esta repetido , ya que ambas funciones lo comparten . No parece nada grave pero sí que es cierto que no es lo “correcto”. ¿Qué podemos hacer para solventar este problema? . Podemos apoyarnos en programación funcional y generar una nueva función que reciba el formato como parámetro. Claro al pasar el formato como una variable tendremos que decidir qué tipo de variable es. En nuestro caso va a ser una función de tipo BiConsumer . Es decir recibe 2 parámetros e imprime un resultado veamoslo:

```
package com.arquitecturajava;

import java.util.function.BiConsumer;

public class Principal4 {

    public static void main(String[] args) {

        int numero = 5;
```

```
        tablaMultiplicarFormato(numero, (n, i) - >
System.out.format("%d x %d=%d \n", n, i, n * i));

        numero = 7;

        tablaMultiplicarFormato(numero, (n, i) - >
System.out.format("%d multiplicado por  %d  es igual a %d \n", n, i, n
* i));

    }

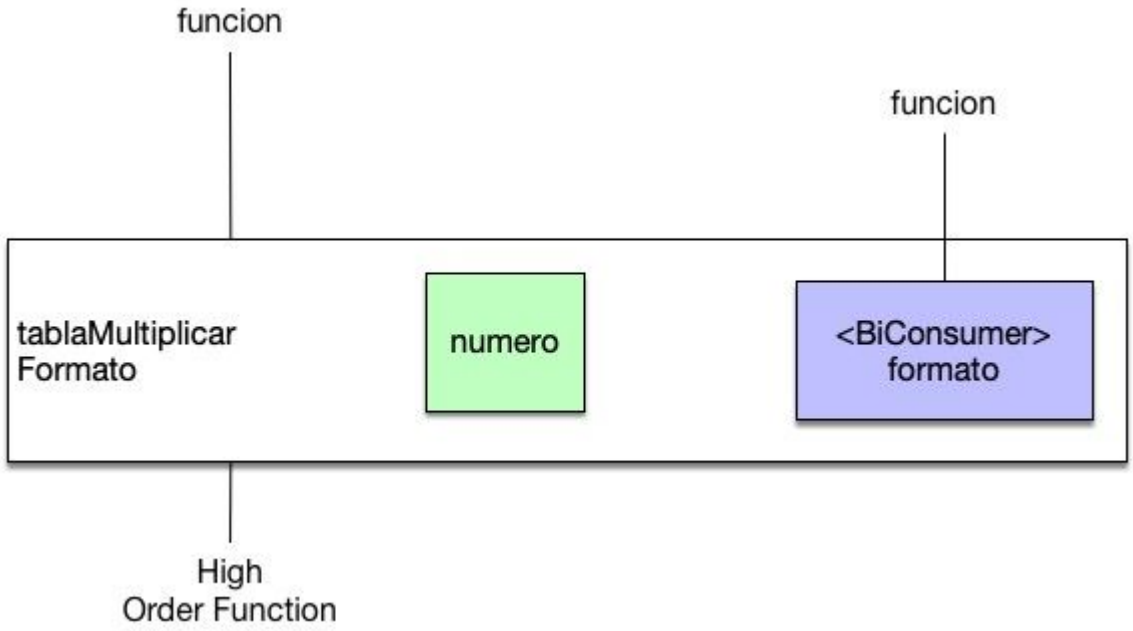
    private static void tablaMultiplicarFormato(int numero, BiConsumer
< Integer, Integer> formato) {

        for (int i = 0; i < 10; i++) {

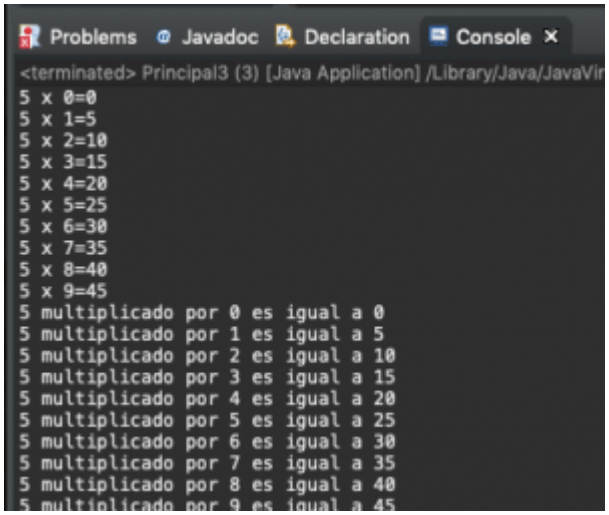
            formato.accept(numero, i);
        }
    }

}
```

Hemos avanzado un paso más y al usar programación funcional y el concepto de High Order Function (una función que recibe como parámetro otra función). Hemos conseguido optimizar el código de forma clara para nuestro pequeño ejemplo.



El resultado es idéntico:



¿Podemos estar contentos con esto? bueno en principio sí , suele ser suficiente .Sin embargo puede haber situaciones que nos generen una duda razonable ,vamos a verlas.

```
package com.arquitecturajava;

import java.util.function.BiConsumer;
```

```
public class Principal4b {

    public static void main(String[] args) {

        tablaMultiplicarFormato(5, (n, i) -> System.out.format("%d x
%d=%d \n", n, i, n * i));

        tablaMultiplicarFormato(5, (n, i) -> System.out.format("%d *
%d=%d \n", n, i, n * i));

        tablaMultiplicarFormato(5, (n, i) -> System.out.format("%d
multiplica por %d=%d \n", n, i, n * i));

        tablaMultiplicarFormato(5, (n, i) -> System.out.format("%d
multiplicado por  %d  es igual a %d \n", n, i, n * i));

    }

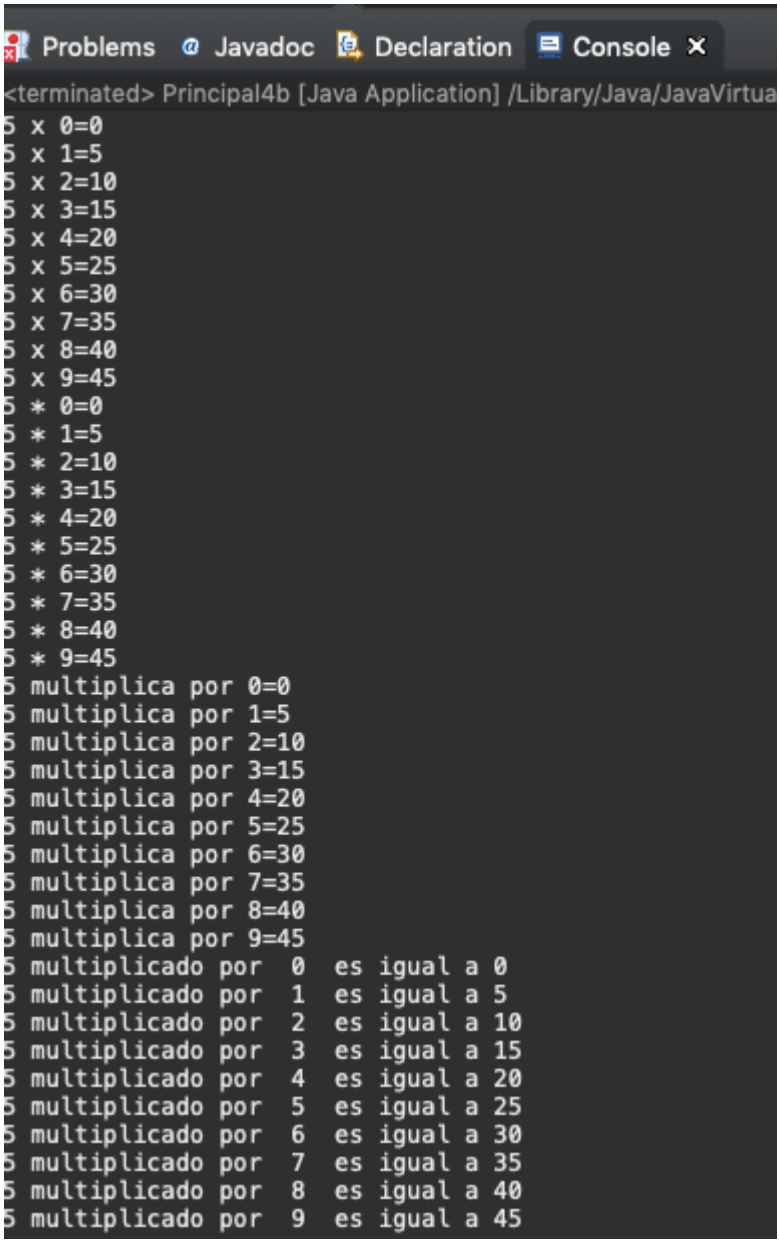
    private static void tablaMultiplicarFormato(int numero, BiConsumer
<Integer,Integer> formato) {

        for (int i = 0; i <10; i++) {

            formato.accept(numero, i);
        }
    }

}
```

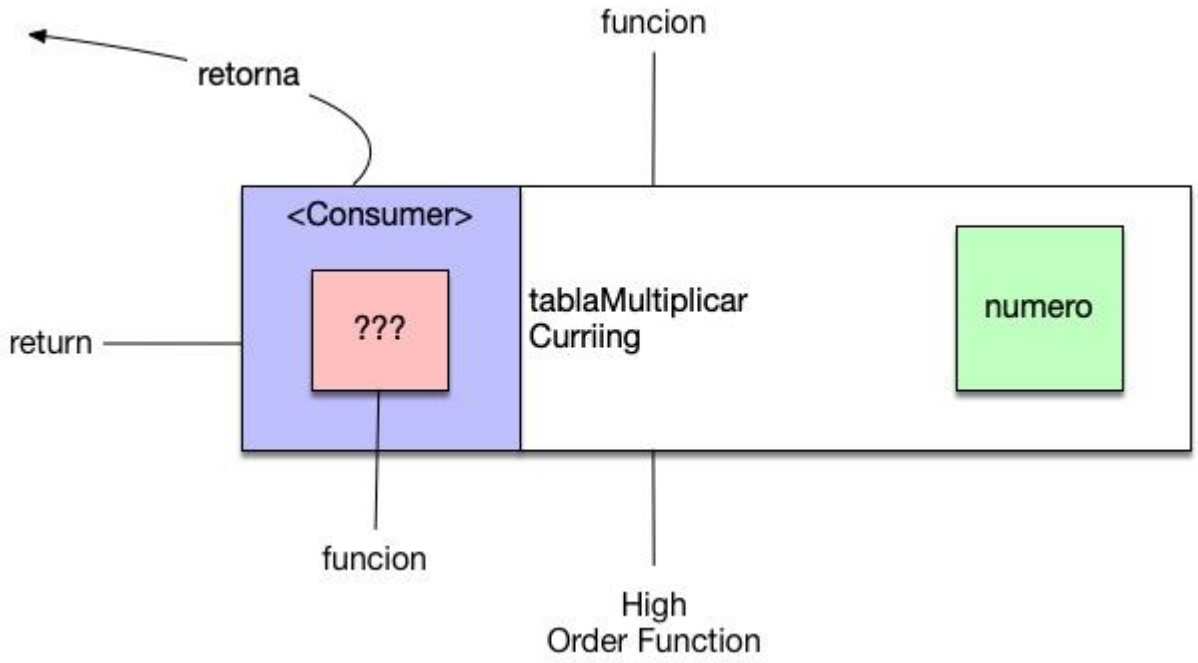
En este caso nos hemos construido un ejemplo de código que nos imprime la tabla de multiplicar del 5 varias veces. Para ello le aplicamos varios formatos a la función de multiplicar con formato y todo funciona.



Java Currying a detalle

El programa se comporta de la forma adecuada así que … no tenemos ningún problema , sin embargo no es difícil darse cuenta que estamos continuamente pasando como parámetro el número “5” y eso es repetir código de una forma o de otra. La solución clásica este problema sería construir una función que sea tablaMultiplicarFormato5 … pero es evidente que se trata de una chapuza ya que tendríamos cientos de funciones una por cada número que deseemos no repetir. ¿Qué otra solución puede ser razonable? Es aquí donde el

concepto de Java Currying viene a rescatarnos. Podemos diseñar una función que devuelva otra función que se genera dinámicamente a partir de un parámetro que la pasamos .



Esto puede parecer un poco extraño en un primer momento y cuesta entenderlo. Seguimos trabajando en una situación en la que contamos con un high order function pero en este caso como parámetro de retorno. Nuestra función ya no recibe una función como parámetro sino que la devuelve. Vamos a verlo un código.

```
ackage com.arquitecturajava;

import java.util.function.BiConsumer;
import java.util.function.Consumer;

public class Principal5b {

    public static void main(String[] args) {

        int numero = 5;
```

```
        Consumer <BiConsumer <Integer, Integer >>
        tablaMultiplicar5 = tablaMultiplicarCurrying(numero);

        tablaMultiplicar5.accept((n, i) - > System.out.format("%d x
%d=%d \n", n, i, n * i));

        tablaMultiplicar5.accept((n, i) - > System.out.format("%d *
%d=%d \n", n, i, n * i));

        tablaMultiplicar5.accept((n, i) - > System.out.format("%d
multiplica por %d=%d \n", n, i, n * i));

        tablaMultiplicar5.accept((n, i) - > System.out.format("%d
multiplicado por  %d  es igual a %d \n", n, i, n * i));

    }

    static Consumer <BiConsumer<Integer, Integer>>
tablaMultiplicarCurrying(int numero) {

        return (formateador) - > {

            for (int i = 0; i < 10; i++) {

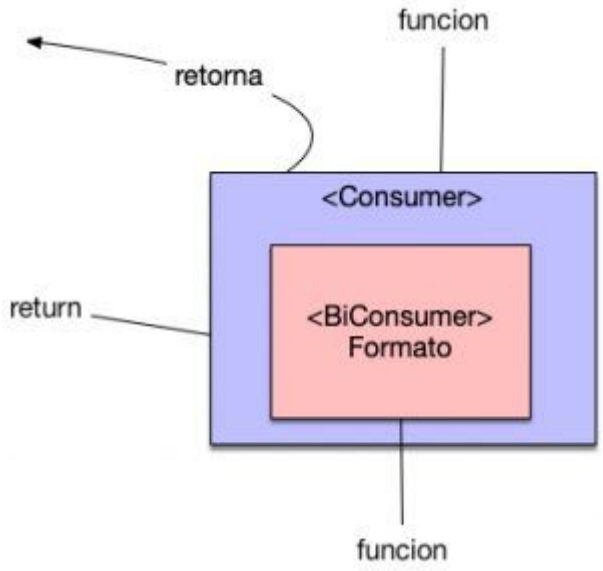
                formateador.accept(numero, i);
            };

        };

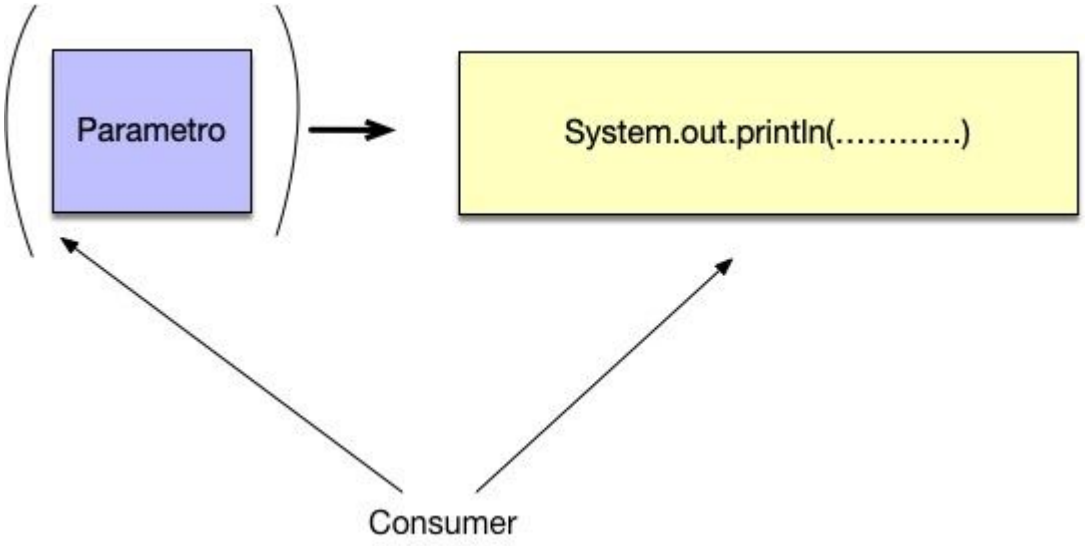
    }
```

}

Lo que en este código es difícil de entender es qué es lo que realmente esta devolviendo la función tablaMultiplicarCurrying.

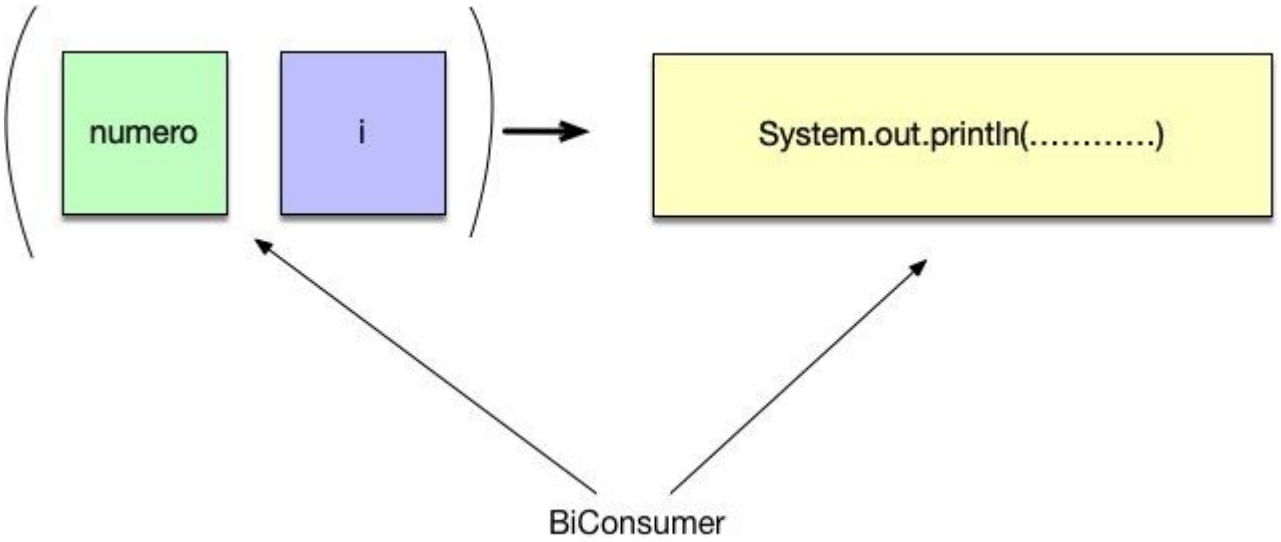


Parece claro que devuelve un Consumer , es decir una función consumidora. Estas funciones tienen siempre la misma firma reciben uno o varios parámetros y ejecutan una funcionalidad y no devuelven nada.

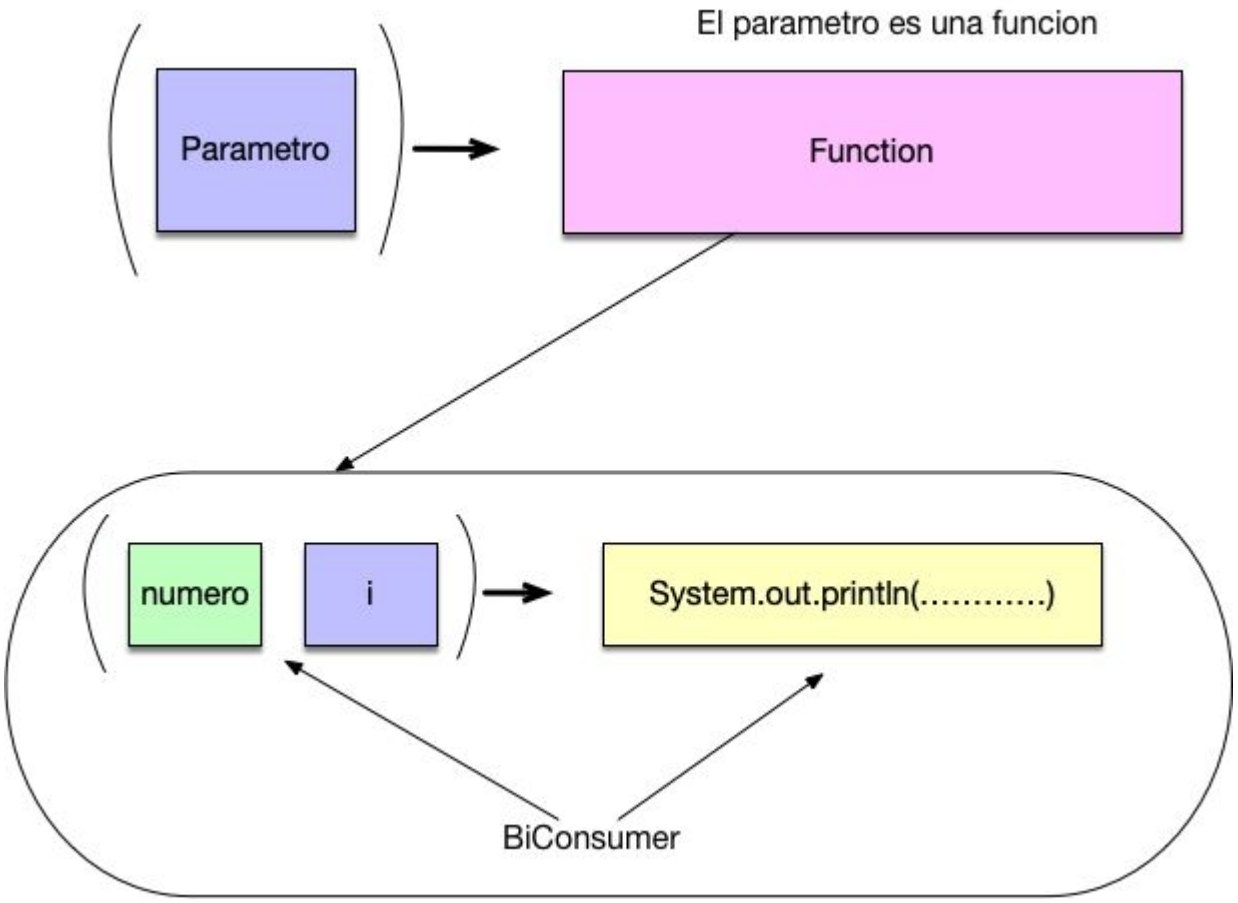


Sin embargo en nuestro caso es algo más complejo ya que se trata de una función que

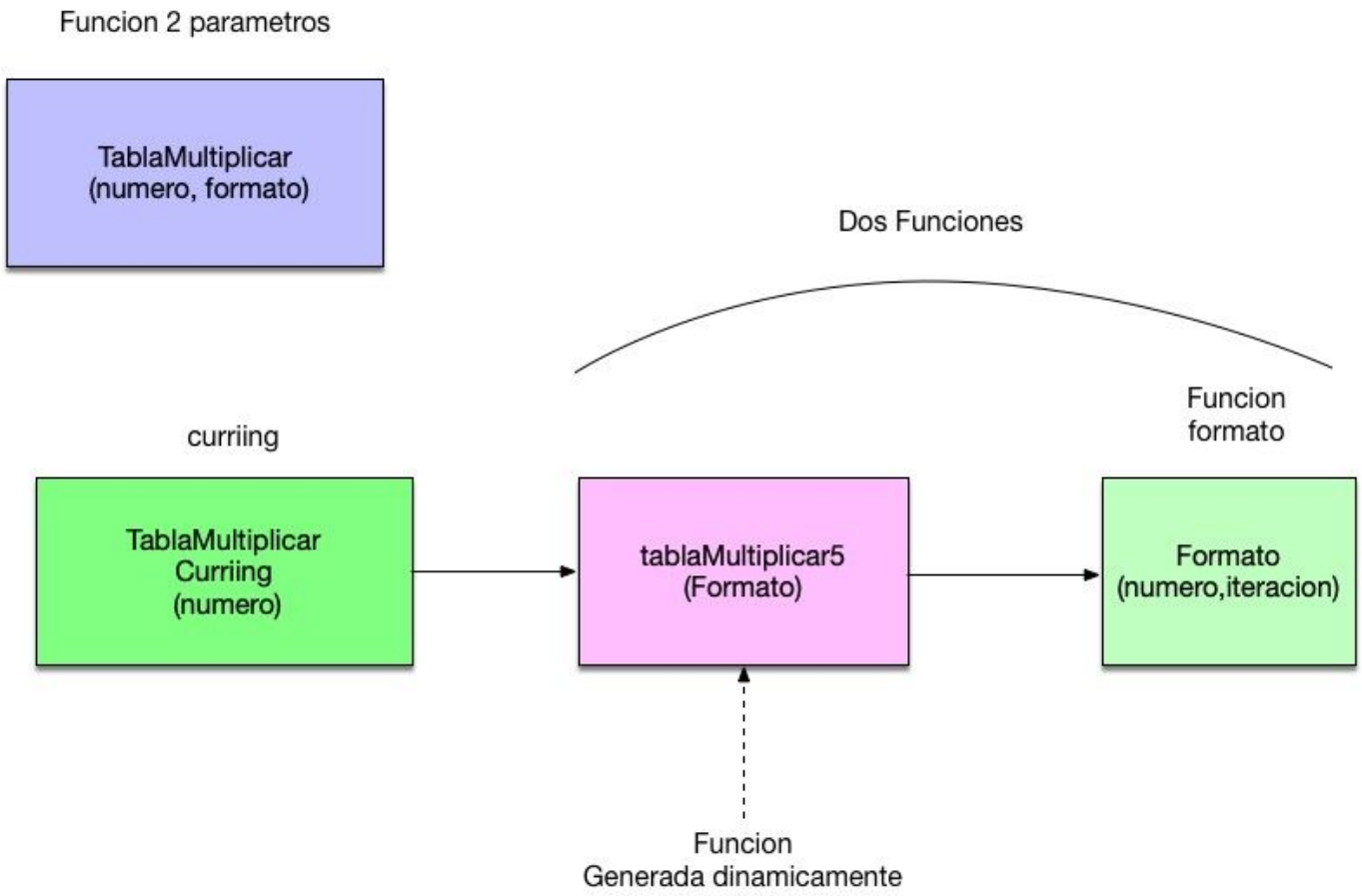
devuelve un Consumer y este consumer tiene como parámetro un BiConsumer.



Así pues la estructura es como se muestra a continuación:



¿Qué es lo que esta haciendo todo este código? . Este código lo que hace es construir una función que recibe un parámetro con ese parámetro de forma dinámica se construye otra función parametrizada a un valor concreto que tendrá sus propios parámetros y funcionara de forma totalmente individual.



Este es el concepto de Java Currying , cuando una función que tiene varios parámetros se desglosa en n funciones. De esta forma podemos generar funciones dinámicas que nos sean útiles a la hora de reutilizar nuestro código.

Otros artículos relacionados

1. Java 8 Lambda Expressions (I)
2. Java 8 Lambda y forEach (II)
3. El concepto de Java constructor reference
4. Java Optional Stream y reference methods
5. interfaces funcionales