Unidad 7. Java 8. Programación funcional

Contenido

1 Actividad inicial

Se le pregunta a los alumnos si la siguiente afirmación es correcta:

¿Se puede construir cualquier algoritmo repetitivo resolviéndolo con simples llamadas a funciones?

Se les pide resolver un recorrido de números del 1 al 100 si usar bucles

2 Introducción

En el tema anterior vimos lo que eran las interfaces funcionales y las expresiones lambda. Ya vimos que la usamos en conjunto y nuestra misión será aplicarlas a listas con un innumerable tipo de datos, objetos. Estamos hablando de listas de cientos de miles de objetos que se manejan en nuestros programas en la actualidad, debido al uso masivo de estadísticas y el Big Data.

Partiendo de esta base recordamos lo que era un interfaz funcional. Como se ha dicho, una interfaz funcional es una interfaz que especifica sólo un método abstracto. Antes de continuar, recuerde que **no todos los métodos de interfaz son abstractos**. A partir de JDK 8, es posible que una interfaz tenga uno o más métodos default (predeterminados). **Los métodos default no son abstractos**. Tampoco lo son los métodos de interfaz estáticos o privados. Por lo tanto, un método de interfaz es abstracto sólo si no especifica una implementación.

En los **interfaces funcionales sólo podemos declarar un método abstracto**. Sólo pueden llevar a cabo una acción. Pero podemos declarar **el método equals para ser sobreescrito y el String, ambos serían abstractos** también. Son métodos que se permiten añadir a los interfaces funcionales para luego usarlos para comparación, ordenación , mapeo, etc., de listas. Igualmente p**odríamos añadir un método por defecto con la etiqueta default,** y **métodos estáticos**. Podéis verlo en el ejemplo.

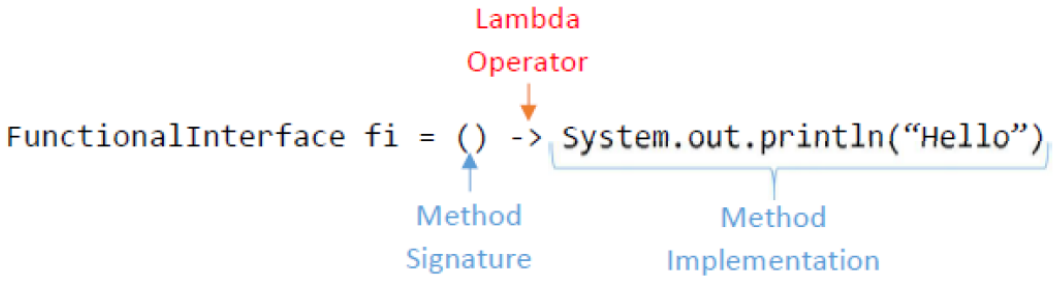
```
@FunctionalInterface
public interface InterfazFuncional {

	void print(String str); //Abstractor

	boolean equals(Object o);

	String toString();

	default void defecto() {
		System.out.println("Método con implementación por
defecto");
```

```
        }

        static void estatico() {
                System.out.println("Método estático");
        }

}
```

Una expresión lambda se basa en un elemento de sintaxis y un operador que difieren de lo que ha visto en los temas anteriores. El operador, a veces denominado **operador lambda** u **operador de flecha**, es **->**.
Divide una expresión lambda en dos partes:
El lado izquierdo especifica los parámetros requeridos por la expresión lambda.
En el lado derecho está el cuerpo lambda, que especifica las acciones de la expresión lambda.
Java define dos tipos de cuerpos lambda. Un tipo **consiste en una sola expresión, y el otro tipo consiste** en un **bloque de código**. Comenzaremos con lambdas que definen una sola expresión.

En la siguiente figura podéis ver las diferentes partes de una expresión lambda:

- La signatura del método
- El operador lambda
- La implementación del método



Visteis bastantes ejemplos en el tema anterior. Por ejemplo la lambda que calcula el inverso de un número. (n) -> 1.0 / n.

En este tema vamos a seguir contando más interfaces funcionales. Variaciones del interfaz predefinido Function. Es el objetivo de este documento.

3 Principios de la programación funcional

. La programación funcional en Java no ha sido fácil históricamente, e incluso **había varios aspectos de la programación funcional** que **ni siquiera eran realmente posibles** en Java. En **Java 8, Oracle hizo un esfuerzo para facilitar la programación funcional**, y este esfuerzo **tuvo éxito hasta cierto punto**. En estos apuntes **de programación funcional de Java** vamos a ir **a través de los conceptos básicos** de la **programación funcional**, y qué **partes de ella son factibles en Java**.

Conceptos básicos de programación funcional
La programación funcional contiene los siguientes conceptos clave:

1. Funciones como objetos de primera clase
2. Funciones puras
3. Funciones de orden superior

La programación funcional pura también **tiene un conjunto de reglas** a seguir:

1. Ningún estado
2. Sin efectos secundarios
3. Variables inmutables
4. Favorecer la recursividad sobre el bucle

Estos conceptos y reglas se explicarán en el resto de esta unidad

Incluso si no sigue todas estas reglas todo el tiempo, **nos podemos beneficiar de las ideas de programación funcional** en **nuestros programas**. Desafortunadamente, la **programación funcional no es la herramienta adecuada** para **todos los problemas**. Especialmente la idea de "sin efectos secundarios" **hace que sea difícil**, por ejemplo, **escribir en una base de datos** (que es un efecto secundario). Pero **la API Stream de java y otras cualidades de la programación funcional** son de **uso diario por los programadores java.**

1.1 *Funciones como objetos de primera clase. Actividad guiada funciones como miembros de primera clase*

En el paradigma de programación funcional, las **funciones son objetos de primera clase en el lenguaje**. Esto significa **que se puede crear una "instancia" de una función**, al igual que **una referencia de variable a esa instancia de función**, al igual que una **referencia** a un String, un HashMap o cualquier otro objeto. Las funciones también se **pueden pasar como parámetros a otras funciones**.
En Java, los **métodos no son objetos de primera clase**. Lo más cerca **que obtenemos son los interfaces funcionales en Java.**

Vamos a **explicarlo con un sencillo ejemplo**. En nuestra **clase PrimeraClase tenemos una función que aplica la formula recibida** en un **interfaz function sobre un número**.

```
public Double funcionFormula(Double x, Function<Double,Double>
formula) {


        return formula.apply(x);
}
```

Dos **funciones con la formula del cuadrado** y **el cubo.**

```
public Double formulaCuadrado(Double x) {

                return x*x;

        }

        public Double formulaCubo(Double x) {

                return x*x*x;

        }
```

En nuestra **función Main del programa principal** creamos un **objeto de tipo ClasePrincipal**, y vamos a llamar al método de esa clase que recibe un interfaz funcional como parámetro para explicar que es función como miembro de primera clase.

```
PrimeraClase objeto = new PrimeraClase();
```

Én el tema anterior veíamos un ejemplo de como **pasar una función estática** como parámetro usando el operador dos puntos a un interfaz funcional, escribíamos Nombredelaclase::nombredelafuncion . Hacemos **lo mismo ahora pero con un objeto**. Pasamos un **método del objeto como parámetro. Usamos para pasar es método como parámetro el operador ::** que **significa enlace o puntero a una función**. Como tengo un objeto de la clase paso la función como nombredelobjeto::nombredelmetodo. `objeto::formulaCubo`. De esta **manera en Java las funciones son miembros** de **primera clase**, se **pueden almacenar en variables** y se **pueden pasar como parámetros**.

```
Double resultado = objeto.funcionFormula(numero,
objeto::formulaCuadrado);

                System.out.println("Aplicamos la formula del cuadrado
con resultado " + resultado);

resultado = objeto.funcionFormula(numero, objeto::formulaCubo);
```

Como una **expresión lambda es una función anónima**, pues también es un **miembro de primera clase y lo podemos poner como parámetro** de **interfaz**

funcional, como **veis en el ejemplo.**

```
resultado = objeto.funcionFormula(numero, x->x*x*x*x);
```

PrimeraClase.java

```
import java.util.Scanner;
import java.util.function.Function;

public class PrimeraClase {


	public Double funcionFormula(Double x, Function<Double,Double>
formula) {



		return formula.apply(x);
	}


	public Double formulaCuadrado(Double x) {

		return x*x;

	}

	public Double formulaCubo(Double x) {

		return x*x*x;

	}

	public static void main(String[] args) {

		Scanner sc = new Scanner(System.in);

		System.out.println("Escribe un numero por pantalla");

		Double numero = sc.nextDouble();
		PrimeraClase objeto = new PrimeraClase();



		Double resultado = objeto.funcionFormula(numero,
objeto::formulaCuadrado);

		System.out.println("Aplicamos la formula del cuadrado
con resultado " + resultado);

		resultado = objeto.funcionFormula(numero,
objeto::formulaCubo);

		System.out.println("Aplicamos la formula del cubo con
resultado " + resultado);

		resultado = objeto.funcionFormula(numero, x->x*x*x*x);


		System.out.println("Aplicamos la formula de potencia 4
```

```
con resultado " + resultado);

        }

}
```

3.1.1 Actividad independiente funciones como miembros de primera clase.

1. Se crea una clase ClaseImpuestos con tres métodos:
    a. Float ImpuestosExtranjerosVisitantes(Float importe): calcula los impuestos con una retención del 2% pero sin iva.
    b. Float ImpuestosNacionales(Float importe): calcula los impuestos con un iva del 21%.
    c. float ImpuestosResidentesExtranjeros(Float importe):: le resta un 5% al IVA.
2. Se crea un método float ImpuestosGenericos (Function<Float,Float> formulaImpuestos). Recibe una funcion como parámetro, interfaz funcional de tipo Function que será ejecutado dentro).
3. Se crea un programa principal que pregunta por el importe, el tipo de cliente, ExtranjeroVisitante, ExtranjeroResidente y Nacional, y calcula el importe total llamado al método ImpuestosGenericos con la formula adecuada.

```
Cual es el importe a pagar: 90

Introduzca un numero entre 1 y 3 para:
   1. Nacional
   2. Extranjero residente
   3. Visitante
1
El importe es: 90
Los impuestos: 18,9
El total es: 108,8
```

1.2 *Funciones puras*

Una función es una función pura si:

1. La **ejecución de la función no tiene efectos secundarios**.

2. El valor **devuelto de la función depende únicamente de los parámetros de entrada** pasados a la función.
3. Aquí tenéis **un ejemplo de una función pura** (método) en Java:

```
public class ObjetoConFuncionPura{

   public int suma(int a, int b) {
      return a + b;
   }
}
```

Observe cómo el valor devuelto de la función suma() **solo depende de los parámetros de entrada**. Observe también **que la suma() no tiene efectos secundarios**, lo que significa **que no modifica ningún estado (variables) fuera de la función** o la clase.

Lo opuesto lo encontrareis en el siguiente **ejemplo de una función no pura**:

```
public class ObjetoConUnaFuncionNoPura{
   private int valor = 0;

   public int suma(int siguienteValor) {
      this. valor += siguienteValor;
      return this. valor;
   }
}
```

Observa **cómo el método add()** utiliza una **variable miembro para calcular su valor devuelto**, y **también modifica el estado de la variable** miembro valor, por lo que tiene un efecto secundario, modifica el estado de una clase.

1.3 *Funciones de orden superior*

Una **función es una función de orden superior** si se **cumple al menos una de las siguientes condiciones**:

- La **función** toma **una o más funciones como parámetros.**
- La **función devuelve otra función** como resultado.

En Java, **lo más cerca que podemos llegar a una función de orden superior** es una función (método) que **toma una o más Interfaces funcionales como parámetros** y **devuelve un interfaz funcional**. Aquí **teneis un ejemplo de una función de orden superior en Java:**

En este **ejemplo más sencillo podeis probar una función de orden superior**.

```
public Supplier<Double> funcionOrdenSuperiorFunction(Supplier<Double> numero,
Function<Double,Double> funcion)
```

La **funcionOrdenSuperiorFunction** toma como **parámetro un interfaz Supplier** que **nos da un número, un interfaz Function** y devuelve **un Interfaz supplier**. Lo que estamos haciendo básicamente, es permitir pasar como parámetro **expresiones lambda o funciones y devolver una función o expresión lambda como parámetro.**

Fijaos en la ejecución

```
  Supplier<Double> numero=
funcionOrdenSuperior.funcionOrdenSuperiorFunction(()-> numero1 ,(x)->x*x);
```

A función de orden superior le paso **dos parámetros, el** primero un **Supplier como expresión lambda, ()-> numero 1. El segundo un interfaz Function como expresión lambda** `,(x)->x*x`.

Y que **devuelve la función, un supplier**. Otra expresión lambda o función. Podríamos **devolver una función aquí usando el operador ::.**

```
return ( ()->funcion.apply(numero.get()));
```

FuncionOrdenSuperior.java

Ejemplo de función de orden superior

```
import java.util.Scanner;
import java.util.function.Consumer;
import java.util.function.Function;
import java.util.function.Supplier;

public class FuncionOrdenSuperior {

        public Supplier<Double> funcionOrdenSuperiorFunction(Supplier<Double>
numero, Function<Double,Double> funcion) {


                return ( ()->funcion.apply(numero.get()));

        }



          public static void main(String[] args) {


                  Scanner miScanner = new Scanner(System.in);

                  System.out.println("Escriba un número decimal");

                  Double numero1 = miScanner.nextDouble();

                  FuncionOrdenSuperior funcionOrdenSuperior = new
```

```
FuncionOrdenSuperior() ;


                    Supplier<Double> numero=
funcionOrdenSuperior.funcionOrdenSuperiorFunction(()-> numero1 ,(x)->x*x);

                    System.out.println("Resultado funcion
funcionOrdenSuperiorFunction que calcula el cuadrado " + numero.get());




          }

}
```

3.2 Sin Estado

Como **se mencionó al principio de estos apuntes**, una regla **del paradigma de programación funcional es no tener ningún estado**. Por "ningún estado" normalmente se entiende por ningún estado externo a la función. Una f**unción puede tener variables locales** que contienen **el estado temporal internamente**, pero la función **no puede hacer referencia a ninguna variable miembro de la clase u objeto** al que pertenece la función. No **debe acceder a valores externos**.

Ejemplo de una función que no utiliza ningún estado externo:

```
public class Calculadora{
   public int suma(int a, int b) {
      return a + b;
   }
}
```

Al contrario, en **este método nos encontramos con el problema** de que **dependemos del estado.**

```
public class Calculadora {
   private int initVal = 5;

public setInitVal(int val) {

initVal= val;
}
   public int suma(int a) {
      return initVal + a;
   }
}
```

Esta **función infringe claramente la regla sin estado**.

1.4 *Sin efectos secundarios*

Se deduce del ejemplo anterior. Si **nadie puede modificar mi estado**, nadie **puede producir un resultado indeseado** en mi ejecución.Es otra regla en el paradigma de programación funcional. Esto **significa que una función no puede cambiar ningún estado** fuera de la función. **Cambiar el estado fuera de una función** se **conoce como un efecto secundario.**

El **estado fuera de una función hace referencia** tanto a las **variables miembro de la clase u objeto de la función**, como a las **variables miembro dentro de los parámetros a las funciones**, o al **estado en sistemas externos** como sistemas **de archivos o bases de datos.**

```
public class Calculadora{
   public int suma(int a, int b) {
      return a + b;
   }
}
```

suma (5,7) siempre va a ser 12 en una programación funcional correcta. No transmitimos errores.

```
public class Calculadora {
   private int initVal = 5;

public setInitVal(int val) {
initVal= val;
}
   public int suma(int a) {
      return initVal + a;
   }
}
```

En este segundo caso,
Suma(7) va a ser 12
Pero si modifico el estado
setInitVal (8)
suma(7) va a ser 15, puedo transmitir errores.

3.3 Variables inmutables

Una tercera regla en el paradigma de programación funcional **es la de las variables inmutables**. **Variables inmutables hacen que sea más fácil evitar efectos secundarios**. Vimos ya que **era una variable u objeto inmutable** en **los apuntes de Diseño de patrones. Podeis encontrar ejemplos en esos apuntes**. Podéis ver en esos apuntes las ventajas de **los objetos inmutables**. En estas variables **no podemos modificar su valor**.

3.4 Favorecer la recursividad sobre el bucle

Una cuarta regla en el paradigma de programación **funcional es favorecer la recursividad sobre el bucle.** La recursividad utiliza llamadas de función para lograr bucles, por lo que el código se vuelve más funcional.

Otra alternativa a los bucles es la API de Java Streams. Esta **API está inspirada funcionalmente.** La veremos más adelante en el curso.

En el tema siguiente introduciremos recursividad, ver:
Recursividad

3.5 Interfaces Funcional

Una interfaz funcional en Java es una interfaz que sólo tiene un método abstracto. Por un método abstracto se **entiende sólo un método que no se implementa**. Una **interfaz puede tener varios métodos, por ejemplo, métodos predeterminados y métodos estáticos**, **ambos con implementaciones**, pero mientras la interfaz solo tenga un método que no se implemente, la interfaz se considera una interfaz funcional.

Un ejemplo de un interfaz funcional sería:

```
public interface MyInterface {
    public void run();
}
```

Otro ejemplo de una interfaz funcional con un método predeterminado y un método estático también. Lo vimos en el tema anterior.

```
public interface MyInterface2 {
    public void run();

    public default void doIt() {
        System.out.println("doing it");
    }

    public static void doItStatically() {
        System.out.println("doing it statically");
    }
}
```

Observad los dos métodos con implementaciones. Esto **sigue siendo una interfaz funcional, porque sólo run() no se implementa (abstracto).** Sin embargo, **si hubiera más métodos sin implementación**, la **interfaz ya no sería una interfaz funcional** y, por lo tanto, **no podría implementarse mediante una expresión lambda Java**.

4 Composición en interfaces funcionales

Este concepto lo introducimos en el tema anterior, pero ahora lo vamos a desarrollar más. Ya vimos que en los interfaces predefinidos **podemos usar andThen,** para añadir una función detrás de otra. En el caso de los interfaces funcionales **Predicate podíamos crear predicados más complejos sobreescribiendo** los métodos **and y or del interfaz tipo Predicate.**

Vamos a **añadir un nuevo método a sobreescribir, el compose**, para los **interfaces funcionales de tipo Function**.

4.1 Ejemplo de composición funcional de Java

A modo de repaso del tema anterior, para empezar, os indico un **ejemplo de composición funcional Java**. Aquí hay una sola función compuesta por otras dos funciones:

```
Predicate<String> startsWithA = (text) -> text.startsWith("A");
Predicate<String> endsWithX   = (text) -> text.endsWith("x");

Predicate<String> startsWithAAndEndsWithX =
        startsWithA.and(endsWithX);

String  input  = "A hardworking person must relax";
boolean result = startsWithAAndEndsWithX.test(input);
System.out.println(result);
```

Este ejemplo **de composición funcional crea primero dos implementaciones de predicado en forma de dos expresiones lambda.** El primer predicado devuelve true si el **String** que se le pasa como parámetro comienza con una mayúscula a (A). El segundo predicado devuelve true si la cadena que se le pasa termina con una minúscula x . Tenga en cuenta que la interfaz **Predicate** contiene un único método no implementado denominado test() que devuelve un valor booleano. Es este método que implementan las expresiones lambda.

Después **de crear las dos funciones básicas**, se compone **un tercer predicado, que llama a los métodos test()** de **las dos primeras funciones**. Esta tercera función devuelve true si ambas funciones básicas devuelven true y **false en caso contrario**.

Por último, en este ejemplo **se llama a la función compuesta y se imprime el resultado**. Puesto **que el texto comienza con una mayúscula(A) y** termina con **una x minúscula, la función compuesta devolverá true** cuando se llama con la cadena **"Una persona trabajadora debe relajarse".**

4.2 Soporte de composición funcional Java

En el ejemplo de la sección anterior se muestra cómo componer una nueva función a partir de otras dos funciones. Varias de las **interfaces funcionales en Java ya tiene soporte para la composición funcional integrada en ellas**. El soporte de **composición funcional viene en forma de métodos predeterminados y estáticos** en las interfaces funcionales. Para obtener más información sobre los métodos predeterminados y estáticos en interfaces, teneis la página de Oracle, los creadores de java.
Aquí teneis la documentación para cada clase **del paquete java.uti.Function** donde tenemos todas las **clases relativas a interfaces funcionales y soporte de expresiones lambda.**

Java.util.Function nos proporciona los interfaces funcionales predefinidos como **objetivo contenedor de las expresiones lambda** y **referencias a funciones.**
https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html

4.3 Composición de interface predicate. Practica guiada

La interfaz **Predicate (java.util.function.Predicate)** contiene algunos métodos que ayudan a componer nuevas instancias de **Predicate** de otras instancias de **Predicate**. Cubriremos algunos de estos métodos en las siguientes secciones.

4.3.1 and() y or()

El método de Predicate **and()** es un método predeterminado. El método and() se utiliza **para combinar otras dos funciones de Predicate** de la misma manera que mostramos en el tema anterior **para la composición funcional** de Java. A continuación se muestra **un ejemplo de composición funcional** con el método Predicate and():
Este ejemplo de composición de predicado compone un nuevo Predicado de otras dos instancias de Predicate utilizando **el método and()** y el metodo or() **a partir de instancias básicas de Predicate.**
Para el **Predicado compuesto devolverá true desde su método test()** si las dos **instancias de Predicate de las que se compuso también devuelven true**. En otras palabras, si divisiblePorDos y divisiblePorTres dos devuelven true.
Ya sabeis que para **el or,** con que uno delos Predicate simples devuelva verdadero, el Predicate compuesto será verdadero

EjemploComposicionPredicateAnd.java

```
import java.util.Scanner;
import java.util.function.Predicate;

public class EjemploComposicionPredicateAnd {
```

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

        Predicate<Integer> divisiblePorDos = (i) -> i%2==0 ;
        Predicate<Integer> divisiblePorTres   = (i) -> i%3==0 ;

        Predicate<Integer> composicionAnd =
divisiblePorDos.and(divisiblePorTres);
        Predicate<Integer> composicionOr =
divisiblePorDos.or(divisiblePorTres);

        Scanner miScanner = new Scanner(System.in);


    System.out.println("Escriba un número entero");
    Integer numero= miScanner.nextInt();

    if (divisiblePorDos.test(numero))
        System.out.println("el numero" +numero + " es divisible por dos"
);
    if (divisiblePorTres.test(numero))
        System.out.println("el  numero"  +  numero  +  "  es  divisible  por
tres" );


    if (composicionAnd.test(numero))
        System.out.println("el numero" +numero + " es divisible por dos
y por tres" );

    if (composicionOr.test(numero))
        System.out.println("el numero" + numero + " es divisible por dos
o por tres" );


        }


}
```

4.3.2 or()

El método **Predicate or()** **se utiliza para combinar una instancia de Predicate con otra,** para componer una tercera instancia de Predicate. **El predicado compuesto devolverá true si cualquiera de las instancias de Predicate del quevse compone de devuelve true**, cuando se llama a sus métodos test() con el mismo parámetro de entrada que el predicado compuesto. Aquí tenes un ejemplo de composición funcional Java con el Predicate or():
Compongo dos predicados con el or, el que me devuelve si un número es divisible por dos y el que me devuelve si un número es divisible por 3.

```
Predicate<Integer> composicionOr =
divisiblePorDos.or(divisiblePorTres);
```

EjemploComposicionPredicateOr.java

```
import java.util.Scanner;
import java.util.function.Predicate;

public class EjemploComposicionPredicateOr {




        public static void main(String[] args) {

        Predicate<Integer> divisiblePorDos = (i) -> i%2==0 ;
        Predicate<Integer> divisiblePorTres   = (i) -> i%3==0 ;

        Predicate<Integer> composicionAnd =
divisiblePorDos.and(divisiblePorTres);
        Predicate<Integer> composicionOr =
divisiblePorDos.or(divisiblePorTres);

        Scanner miScanner = new Scanner(System.in);


    System.out.println("Escriba un número entero");
    Integer numero= miScanner.nextInt();

    if (divisiblePorDos.test(numero))
        System.out.println("el numero" +numero + " es divisible por
dos" );
    if (divisiblePorTres.test(numero))
        System.out.println("el numero" + numero + " es divisible por
tres" );


    if (composicionOr.test(numero))
        System.out.println("el numero" + numero + " es divisible por
dos o por tres" );


        }
```

1 Este ejemplo **de composición funcional Predicate or() crea primero dos instancias básicas de Predicate**. En segundo lugar, **el ejemplo crea un tercer predicado compuesto a partir de los dos primeros, llamando al método or() en el primer predicado y pasando el segundo predicado** como parámetro **al método or().**

La **salida de ejecutar el ejemplo anterior será true** porque **la primera de las dos instancias de Predicate utilizadas en el predicado compuesto devolverá true** cuando se llama con la cadena "Una persona trabajadora debe relajarse a veces”.

4.3.3 Practica independiente

1. Crearemos un interfaz predicate divisiblePorDos que nos diga si un número es par.
2. Crearemos un interfaz predicate divisiblePorCinco que nos diga si un número es divisible por 5
3. Crearemos un interfaz predicate divisiblePorDiez componiendo los dos anteriores

4.4 Composición en el Interfaz Function. Practica Guiada composición de interfaz Function

La Function interfaz Java Function (java.util.function.Function) **también contiene algunos métodos que se pueden utilizar** para **componer nuevas instancias** de Function a partir de las existentes. Vamos a ver algunos de estos métodos en esta sección.

4.4.1 compose()

El método Java Function compose() compone **una nueva instancia de Function** de **la instancia de Function en la que se llama** y la **instancia de Function se pasa como parámetro** al método.
La función devuelta por compose() llamará **primero a la función pasada como parámetro** para compose(), y **luego llamará a la función** sobre **la que se llamó compose()**. Esto es más fácil de entender con un ejemplo, así que aquí hay un ejemplo de composición() de la función Java:
En **el ejemplo puedes ver como al componer compongo usando el multiplicar**, `multiplicar.compose(`sumar); y **sumar va dentro**. Como **sumar es más interior se va a hacer primero**. Siempre en programación funcional las operaciones más interiores se hacen primero

```
Function<Integer, Integer> multiplicar = (value) -> value * 2;
                Function<Integer, Integer> sumar      = (value) ->
value + 3;

                Function<Integer, Integer> sumarYMultiplicar =
multiplicar.compose(sumar);

                Integer result1 = sumarYMultiplicar.apply(numero1);
```

EjemploComposicionFunctionSumas.java

```
import java.util.Scanner;
import java.util.function.Function;

public class EjemploComposicionFunctionSumas {
```

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

                Integer numero1;
                Scanner miScanner = new Scanner(System.in);


            System.out.println("Escriba un número entero");
            numero1 = miScanner.nextInt();



                Function<Integer, Integer> multiplicar = (value) ->
value * 2;
                Function<Integer, Integer> sumar      = (value) ->
value + 3;

                Function<Integer, Integer> sumarYMultiplicar =
multiplicar.compose(sumar);

                Integer result1 = sumarYMultiplicar.apply(3);


                System.out.println("El resultado de sumar 2 y
multiplicar 3 a " + numero1.toString() + " es " + result1);

        }


}
```

Cuando se llama con el valor 3, recogido por pantalla, la función compuesta primero llamará a la función add y, a continuación, a la función multiply. El cálculo resultante será (3 + 3) * 2 y el resultado será 12.

4.4.2 andThen()

El método Función **andThen()** **funciona de manera opuesta al método compose().** Una función compuesta con andThen() **primero llamará a la función que inicial** y cuando termina, a continuación, **llamará a la Function pasada como parámetro** al método andThen(). Aquí está un ejemplo de función Java yThen():

EjemploComposicionAndThen.java

```
import java.util.Scanner;
import java.util.function.Function;

public class EjemploComposicionAndThen {


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

```
                Integer numero1;
                Scanner miScanner = new Scanner(System.in);


            System.out.println("Escriba un número entero");
            numero1 = miScanner.nextInt();



                Function<Integer, Integer> multiplicar = (value) ->
value * 2;
                Function<Integer, Integer> sumar      = (value) ->
value + 3;

                Function<Integer, Integer> MultiplicarYSumar =
multiplicar.andThen(sumar);

                Integer result1 = MultiplicarYSumar.apply(numero1);


                System.out.println("El resultado de multiplicar 3 y
sumar 2 a " + numero1.toString() + " es " + result1);

        }


}
```

En este **ejemplo se crea primero una función de multiplicación** y una **función add**. A continuación, se **llama al método andThen() en la función multiply** para componer una **nueva función**, pasando la función add como parámetro a andThen().
Llamar **a la función compuesta por andThen**() con el valor 3 dará como resultado el siguiente **cálculo 3 * 2 + 3** y el resultado será **9.**
Nota: Como se mencionó al principio, AndThen() funciona de manera opuesta de compose(). Por lo tanto, llamar a **a.andThen(b)** es realmente lo mismo que llamar a **b.compose(a) .**

4.4.3 Practica Independiente composición Interfaz Function

1. Creamos un interfaz function Cuadrado con una expresión lambda que calcule el cuadrado de un número
2. Creamos un interfaz function Raiz con con una expresión lambda que calcule la raíz cuadrada de un número
3. Creamos un interfaz function Identidad1 componiendo Cuadrado y Raiz con andThen.
4. Creamos un interfaz function Identidad3 componiendo Cuadrado y Raiz con compose.

Probamos los interfaces para el numero 16.

5 Interface Function y variaciones

Ya vimos en el tema anterior el interfaz predefinido Function **Function<T, R> representa una función** con **un argumento de tipo T** que retorna un **resultado de tipo R**. Como su nombre **indica este interfaz funcional va a realizar la labor de una función**. Recogerá **un objeto o valor de entrada y generará una salida**. Es muy **utilizado al igual que Predicate** en las **operaciones de Streams**. El método abstracto a **sobreescribir es Apply()**.
Vimos un ejemplo en **SizeOf.java**

```
public class SizeOf implements Function<String,Integer>{
    public Integer apply(String s){
        return s.length();
    }
```

```
    SizeOf      sizeOf = new SizeOf();


            Integer r1 = sizeOf.apply("hola java 8");
```

Vamos a ver más ejemplos o variaciones del interfaz Function.

5.1 Interfaz UnaryOperator en Java

La interfaz UnaryOperator<T> es una parte de la API java.util.function que se ha **introducido desde** Java 8, para implementar la programación funcional en Java. Representa **una función que toma un argumento y opera en él**. Sin embargo, lo que lo **distingue de una función normal es que tanto su argumento como el tipo de valor devuelto son los mismos.**

De ahí esta interfaz **funcional que toma un solo tipo genérico a declarar** :-T: denota el tipo genérico del argumento de entrada a la operación
Por lo tanto, **UnaryOperator<T> sobrecarga el tipo Function<T, T>.** Por lo tanto, **hereda los siguientes métodos de la interfaz de función**, ya vistos con anterioridad:

- apply(T t)
- andThen(Function<? super R, ? extends V> after)
- compose(Function<? super V, ? extends T> before)

La **expresión lambda asignada a un objeto de tipo UnaryOperator** se utiliza

para **definir o sobreescribir el accept()** que finalmente **operará sobre el argumento indicado**, como hemos hecho hasta ahora.
Introduce un método nuevo estático, identity, que devuelve el valor pasado como parámetro. **Es la operación matemática identidad**
1. **identity()**
Este método devuelve un UnaryOperator que toma un valor y lo devuelve. El UnaryOperator devuelto no realiza ninguna operación en su único valor. Como veis el método es estático

Syntaxis:

static UnaryOperator identity()

Parámetros: este método no toma ningún parámetro.

Returns: Un UnaryOperator que toma un valor y lo devuelve.

Lo podéis ver en el ejemplo siguiente, la operación identidad, va a devolver el mismo valor que hemos pasado como parámetro. **true**.

import java.util.function.UnaryOperator;

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

        UnaryOperator<Boolean>
            op = UnaryOperator.identity();

        System.out.println("devuelvo lo recibido como parámetro:" + op.apply(
true));
    }
}
```

En el siguiente ejemplo de operador Unario**, podeis ver que siempre devolvemos el mismo tipo que pasamos como parámetro.**

EjemploOperadorUnario.java

```
import java.util.function.UnaryOperator;
import java.util.Scanner;
import java.util.function.Function;


public class EjemploOperadorUnario {
```

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

                        Double numero1;
                             Scanner miScanner = new Scanner(System.in);


                         System.out.println("Escriba un número decimal");
                         numero1 = miScanner.nextDouble();

                    UnaryOperator<Double> operator1 = i -> Math.sqrt(i)+7;
                    UnaryOperator<Double> operator2 = i -> i*7 +2;

                    UnaryOperator<String> doblar = i -> i + " " + i;

                    // Using andThen() method
                    Double a = operator1.andThen(operator2).apply(5.2);
                    System.out.println(a);
                    Double b = operator1.compose(operator2).apply(5.3);


                    System.out.println(b);


                    String s = doblar.apply("Doblo la cadena");
                     System.out.println(s);

                }


}
```

5.2 Interfaz Bifunction. Practica guiada

```
@FunctionalInterface
```

`public interface` **`BiFunction<T,U,R>`**

Representa una **función que acepta dos argumentos y genera un resultado**. Esta es la **especialización con dos parámetros** de `Function`.
Se trata de una interfaz funcional cuyo método funcional es apply(Object, Object).
La expresión lambda **asignada a un objeto de tipo BiFunction se utiliza para definir su apply()** que **finalmente aplica la función dada en los argumentos**.
La principal ventaja de usar un BiFunction es que nos permite utilizar 2 argumentos de entrada mientras que en Function sólo podemos tener 1 argumento de entrada. Podeis ver en el ejemplo como nuestra BiFunction y nuestra expresión lambda **tiene dos parámetros de entrada y uno de salida**

```
bifuncion = (cad1,cad2)-> cad1+cad2;
```

EjemploBiFuncion.java.

```
import java.util.Scanner;
import java.util.function.BiFunction;

public class EjemploBifunction {


        public static void main(String[] args) {

        String cadena1, cadena2;


        Scanner miScanner = new Scanner(System.in);


    System.out.println("Escriba una cadena");
    cadena1 = miScanner.nextLine();

    System.out.println("Escriba otra cadena");
    cadena2 = miScanner.nextLine();



    BiFunction<String,String,String> bifuncion = (cad1,cad2)-> cad1+cad2;
```

```
    System.out.println("Concatenamos las dos cadenas con una bifuncion: " + b
ifuncion.apply(cadena1, cadena2));



}



}
```

Os dejo este otro ejemplo con una *BiFunction de orden superior*. Fijaos que le pasamos a la **función de orden superior**, **dos parámetros en forma de Supplier**, y **un Function.** Como ya comentamos, usamos **los interfaces funcionales** para poder pasar como parámetro **expresiones lambda en java.**

Detalles en este Ejemplo. Para **convertir de Integer a Double** usamos el **método estático de Double**. **Double.valueof(num1).**
Fijaos como ahora usamos **un BiFunction pasamos dos números como Suppliers**, además **de la Bifunction**. Es la **única manera de pasar funciones como parámetros en Java**. Y es **la única manera de pasar expresiones lambda como parámetros**. La función devuelve un itnerfaz Supplier.

```
public Supplier<Double> FuncionPuraBifunction(Supplier<Integer> num1, Supplier<Integer> num2, BiFunction<Integer, Integer, Double> funcion)
```

Observad la **llamada a la función**. Pasamos **tres expresiones** lambda **.()->numero1, ()->numero2** , son dos funciones que devuelven dos **números** sobre los que **queremos aplicar la función**, y `(num1,num2)->Double.valueOf(num1)/Double.valueOf(num2)` **es la función** que va a converitr los dos Integer en Decimal, y nos devuelve la división decimal.
()->numero1 es un Supplier, función **que no recibe parámetros** y que **devuelve un número.**

```
ejemplo.FuncionPuraBifunction(()->numero1, ()->numero2, (num1,num2)->Double.v
alueOf(num1)/Double.valueOf(num2));
```

```
import java.util.Scanner;

import java.util.function.BiFunction;

import java.util.function.Supplier;
```

```
public class EjemploBifunctionPura {




        public Supplier<Double> FuncionPuraBifunction(Supplier<Integer> num1,
Supplier<Integer> num2, BiFunction<Integer, Integer, Double> funcion) {



                return (()-> funcion.apply(num1.get(),num2.get()));



        }

        public static void main(String[] args) {

        Integer numero1, numero2;

        EjemploBifunctionPura ejemplo=  new EjemploBifunctionPura();
        Scanner miScanner = new Scanner(System.in);



    System.out.println("Escriba un número entero");
    numero1 = miScanner.nextInt();

    System.out.println("Escriba otro número entero");
    numero2 = miScanner.nextInt();

    Supplier<Double> resultado =

                ejemplo.FuncionPuraBifunction(()->numero1, ()->numero2, (num1
,num2)->Double.valueOf(num1)/Double.valueOf(num2));

    System.out.println("El resultado es un decimal" + resultado.get());
```

```
        }


}
```

BiFunction Pura

5.2.1 Practica independiente BiFunction

1. Crear una clase PracticaIndependienteBifunction
2. Crear el métodos Double Potencia(Double numero, Double exponente) que calcule la potencia del número a partir del exponente
3. Crear el método Double Producto (Double numero1, Double numero2) que devuelve el producto de ambos.

4. Crear un método Double ejecutaBifunction(Double numero1, Double numero 2, BiFunction<Double, Double, Double> formula) que devuelva como valor el resultado la ejecución del parámetro BuFunction formula con los parámetros numero1, numero2.

5. Crear un programa principal que recoja los dos números y devuelva los resultado para potencia y producto usando el método ejecutaBifunction

5.3 Especialización de Functions primitivos

Vamos a **ver distintos tipo de especialización de interfaces funcionales** predefinidos vistos en el tema anterior. Para casi todos **los tipos primitivos en Java tenemos su especialización.**
Puesto que **un tipo primitivo no puede ser un argumento de tipo genérico**, hay versiones de la interfaz Function para los tipos primitivos más utilizados **double, int, long y sus combinaciones** en tipos de argumento y tipo devuelto:

- ***IntFunction, LongFunction, DoubleFunction: los* argumentos son de tipo especificado**, el tipo de valor devuelto se parametriza.
- ***ToIntFunction, ToLongFunction, ToDoubleFunction:** los* **tipos de valor devuelto es de tipo especificado**, los argumentos se parametrizan.
- ***DoubleToIntFunction, DoubleToLongFunction, IntToDoubleFunction, IntToLongFunction, LongToIntFunction, LongToDoubleFunction*** : tienen **un argumento y tipo de valor devuelto definidos** como **tipos primitivos**, según lo especificado por sus nombres.

No hay una interfaz funcional lista para usar para, por ejemplo, **una función que toma un short y devuelve un byte**, pero **nada te impide escribir el tuyo propio**, como vimos el otro día podemos escribir nuestros propios interfaces funcionales. Al lof them are specialization of the Function functional interface. Básicamente son especializaciones del tipo Function.
IntFunction<Double> sería el equivalente a Function<Integer, Double>, pero el **tipo de entrada es int**, un tipo primitivo, el de salida Double.
DoubleToIntFunction es equivalente a Function a Function<Double,Integer> pero el tipo de salida **es primitivo int y el de entrada double**.
ToIntFunction<Double>: es equivalente a Function<Double,Integer>, otra vez lo mismo el **tipo de salida es primitivo**.

Note: Used if we want to use primitive types in our functional interfaces.

In the following example you will understand **specializations.** They allow us to work *with basic* types, **but we are broken by the functional purity of our interfaces.** You **can use it to perform conversions to basic types** through functional interfaces. As you have noticed, we are using, the java class version Legacy of the basic types. **Double, Integer, Long, Boolean** are classes that work exactly like the associated basic types, **double, long, boolean**.

Look at the next declarations :

`IntFunction<String> convertimosIntString,` receives a String returns a basic int type.
`ToIntFunction<String> convertimosString` recoge un tipo básico int, dentro del interfaz y devuelve un a String
IntToDoubleFunction convertimosIntADouble convierte de Double a tipo básico int.

```
import java.util.Scanner;
import java.util.function.IntFunction;
import java.util.function.IntToDoubleFunction;
import java.util.function.ToIntFunction;


public class EjemploIntFunction {




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

```
                  Scanner miScanner = new Scanner(System.in);
                  System.out.println("Escriba un número entero");
                  int numero1 = miScanner.nextInt();

                IntFunction<String> convertimosIntString = (i)-> String.value
Of(i);

                ToIntFunction<String> convertimosStringInt = (s -> Integer.va
lueOf(s));

                IntToDoubleFunction convertimosIntADouble = (i -> Double.valu
eOf(i));

                String cadena = convertimosIntString.apply(numero1);

                System.out.println("Convertimos " + numero1 + " a cadena " +
cadena);

                int numero2 = convertimosStringInt.applyAsInt(cadena);

                System.out.println("Convertimos cadena " + cadena + " a numer
o " + numero2);

                Double numerodecimal = convertimosIntADouble.applyAsDouble(nu
mero1);



                System.out.println("Convertimos numero entero" + numero1 + "
a numero decimal " + numerodecimal);

        }

}
```

6 Aplicación práctica de la programación funcional.

Hasta **ahora hemos visto conceptos teóricos de programación funcional**, la creación **de interfaces funcionales, uso de los interfaces predefinidos y expresiones Lambda**. Pero la cuestión es. **¿Para que Java introduce todos estos conceptos?** Vamos a intentar desgranar por partes el porque de toda esta nueva. Hay una serie de características de java que debemos entender.

6.1 Encadenamiento de llamadas a objetos. Practica guiada

Para **encadenar una instrucción en la que tengamos una serie de llamadas** para realizar un algoritmo, **vamos a necesitar una serie de componentes** para el diseño de esta nueva manera funcional de programar.

- Lo primero **es una clase con métodos estáticos** que genere objetos. Os pongo el ejemplo clásico de **double numero = Math.random();** que nos devuelve un número desde un método estático.
- Lo segundo **una clase que tengan métodos realicen cambios sobre objetos** de ella misma y **devuelve el propio objeto**. Opcionalmente para encapsular el código **podríamos definir interfaces** para que el usuario no conozca como realmente está construida esa clase. Lo vamos a ver en el siguiente ejemplo. Esta nueva clase tipo, de la que estamos creando objetos tendrá al menos dos tipos de métodos:

    1. **Métodos que modifiquen su estado** y **devuelvan el objeto modificado**, ya **no usamos getters y setters**. Fijaos en el siguiente. Es **un método que cambia el dni de la persona y se devuelve a si mismo**. Hay tres más en la clase Persona**. Identificarlos**.

```
public Persona cambiaDni(String dni) {

                        this.dni=dni;
                        return this;

                }
```

    2. **Métodos finales que realicen acciones sobre el objeto**, pero **no devuelven el objeto**. Estos métodos **pueden insertar en base de datos, o escribir en fichero o mostrar el objeto en pantalla**. El método a**ccionMostrarPorPantalla**, como veis no devuelve un objeto Persona. lo **escribe** en la consola. Con el método toString

pasa lo mismo. Sólo podemos usarlo al final de nuestra encadenamiento de llamadas, porque ya no devuelve un objeto.

```
public void accionMostrarPorPantalla() {

                System.out.println(this.toString());
        }
```

Construimos este tipo de diseños para poder realizar este tipo de sentencias en el que todo **el rato llamamos a funciones** y **nos acercamos más a la programación funcional**, con funciones más puras. La idea es escribir todos **nuestros programas así, y evitar asignaciones, bucles y condicionales**.

```
   CreaPersonas.getPersonas().
                    cambiaDni("3342432342")
                    .cambiaNombre("Julian")
                    .cambiaApellidos("Lopez")
                    .accionMostrarPorPantalla();
```

1. **CreaPersonas.getPersonas()** devuelve un objeto de la clase Persona, es un método estático de la clase CreaPersonas que crea personas. Si os fijais no jhe declarado ninguna variable de tipo CreaPersonas. Estoy usándola estáticamente
2. **cambiaDni("3342432342")** cambia el DNI pero devuelve el mismo objeto de la clase Persona, por eso `puedo seguir poniendo, un punto y llamando a otro método de persona.
3. **cambiaNombre("Julian")** Recoge el objeto Persona devuelto por cambiaDni y lo cambiaNombre("Julian")
4. Lo mismo con **cambiaApellidos**("Lopez")
5. El último método `accionMostrarPorPantalla` ya oevuelve ningún objeto de tipo persona, por esto es de tipo final. Va a realizar una acción sobre el objeto persona. Ya no podemos seguir llamando a métodos. Es final

El **objetivo es dejar de utilizar variables porque la asignación** de variables produce **más fallos en los programas**. Es lo que se definir como mantener la inmutabilidad. En **una sóla línea con llamadas a funciones hemos realizado un programa**, que cambia un objeto y lo muestra por pantalla **sin declarar ninguna variable**. Vamos **cambiando el estado siempre del mismo objeto**, sin pasar por **varias variables**, varias asignaciones. Produce **menos errores en programación.** Si os dais cuenta**, no hemos usado ninguna variable.**
Este ejemplo sigue el patrón de diseño llamado en cascada. Los métodos van devolviendo el mismo objeto mientras hacen modificaciones sobre el mismo.

```
           class Persona {

                private String dni;
                private String nombre;
                private String apellidos;


                public Persona() { }


                public Persona cambiaDni(String dni) {

                        this.dni=dni;
                        return this;

                }


                public Persona cambiaNombre(String Nombre) {

                        this.nombre=Nombre;
                        return this;

                }


        public Persona cambiaApellidos(String Apellidos) {

                this.apellidos=Apellidos;
                return this;

        }


        public void accionMostrarPorPantalla() {

                System.out.println(this.toString());
        }

        public String toString() {
                return "Persona [dni=" + dni + ", nombre=" + nombre + ",
apellidos=" + apellidos + "]";

        }

           }


         public class CreaPersonas {

                   public CreaPersonas() {


                   }


                   public static Persona getPersonas() {

                            return new Persona() ;
```

```
        }


    public static void main(String[] args) {


        CreaPersonas.getPersonas().
        cambiaDni("3342432342")
        .cambiaNombre("Julian")
        .cambiaApellidos("Lopez")
        .accionMostrarPorPantalla();

    }
    }
```

Podemos **usar expresiones lambdas en algunos métodos de nuestras clases** o incluso que **algún método de nuestra clase reciba un interfaz funcional**. Los métodos base de **nuestras clases como getters, setters, o métodos que realizan un cambio sencillo de estado, no merecería la pena convertirlos a funcionales**. Pero métodos que expresen comportamientos **como por ejemplo un calculo de sueldo si que podríamos hacer que fueran funcionales con expresiones lambda**. En cualquier caso se puede convirtiendo todos **los métodos de tu clase en funciones de orden superior** y realizar una programación funcional pura en Java, pero no es práctico. Es **más practico realizar modelos mixtos**.

Vamos a utilizar expresiones lambda para **crear clases de las que necesitemos sobreescribir un solo método** y para **realizar acciones**. También, **para encapsular o enmascarar nuestro código**. De manera que las clases que **usen nuestras librerías no tengan una idea total** de **cómo funcionan**. Es más seguro. Hilos, eventos, tareas, diferentes clases que vais a usar en este curso **pueden ser fácilmente sobreescritas con expresiones lambda**.

Estos interfaces **funcionales también van a permitirnos cambiar el comportamiento, o métodos de nuestra función de manera dinámica**. Pero sobre todo vamos a poder utilizarlos para lo que más nos interesa, **operaciones de filtrado , ordenado, mapeo, reducción y estadisticas** sobre **arrays ,colecciones de objetos y Strings**.

6.1.1 Practica independiente encadenamiento de llamadas alumnos

1. Se creará la clase alumno que contendrá las propiedades nombre, apellidos, dni y teléfono.
2. Se crearán métodos set para cada propiedad que permitan ser encadenados en cascada

6.2 Cambiando el comportamiento de nuestras clases con lambdas

Vamos a **introducir en este ejemplo como cambiar el comportamiento de nuestras clases de manera dinámica**. En el ejemplo siguiente hemos forzado a la clase Persona implemente **Comparable**. Como ya sabéis nos obliga a implementar el método **compareTo**. Además hemos **sobreescrito compareTo**, para que **reciba un interfaz funcional Function.** De esta manera **pasando una expresión lambda, podemos implementar nuestro método, de manera dinámica**.

El otro gran objetivo de las expresiones lambda es el de **poder pasar funciones como parámetros.** Recordad cuando **hablábamos de funciones de primera clase**. Otros lenguajes **como javascript, C++, Python** te dan esa posibilidad. Java no la tenía.

Vamos a aplicar **el concepto anterior de funciones de orden superior** para **añadir al método compareTo** un interfaz funcional como parámetro.
Fijaos que estoy usando un interfaz funcional Function como parámetro del método equals:

```
@Override
        public int compareTo(Persona2 o) {
                // TODO Auto-generated method stub
                return 0;
        }
```

```
        public int compareTo(Function<Persona2,Integer> compara) {
                // TODO Auto-generated method stub
                return compara.apply(this);
        }
```

Tengo **dos implementaciones del método**. Ya **sabéis que la aplicación de polimorfismo nos permite tener varias implementaciones de un método** si **cambiamos los parámetros del método**. El siguiente **paso es pasarle la expresión lambda para que en tiempo de ejecución me modifique el comportamiento de mi clase**. Hemos convertido al método compareTo en una función de nivel superior, y modificado el comportamiento de mi clase.. Os recuerdo que comparteTo devuelve 1, 0 o -1, mayor, igual o menor.

Primero comparamos por nombre

```
p1.compareTo((p)->p.getNombre().compareTo(p2.getNombre()))==1)
```

La expresión lambda `(p)->p.getNombre().compareTo(p2.getNombre()),` usa el método compareTo de String para compara nombre de **p1.getNombre(),** a través del parámetro p que es p1, **p.getNombre(),**con **p2.getNombre().** Podéis ver que el tipo String, implementa Comparable igualmente.

Luego **comparamos por Dni.**

```
p1.compareTo((p)->p.getDni().compareTo(p2.getDni()))==1)
```

Vamos a ver el método compareTo sobreescrito. Lo único que hace **es ejecutar la expresión lambda que le paso como parámetro**. El **código a ejecutarse** en el método es la **expresión lambda en si.** Por tanto, **si cambio la expresión lambda** le **cambio el comportamiento al método**.

```
return compara.apply(this);
```

```
public int compareTo(Function<Persona2,Integer> compara) {
                // TODO Auto-generated method stub
                return compara.apply(this);
        }
```

```
import java.util.function.Function;

class Persona2 implements Comparable<Persona2> {

                private String dni;
                private String nombre;
                private String apellidos;


                public Persona2() { }


                public Persona2 cambiaDni(String dni) {

                        this.dni=dni;
                        return this;

                }


        public Persona2 cambiaNombre(String Nombre) {

                        this.nombre=Nombre;
                        return this;

                }


        public Persona2 cambiaApellidos(String Apellidos) {

                this.apellidos=Apellidos;
                return this;

        }


        public void accionMostrarPorPantalla() {

                System.out.println(this.toString());
        }
```

```
	public String toString() {
		return "Persona [dni=" + dni + ", nombre=" + nombre + ",
apellidos=" + apellidos + "]";

	}



	public int compareTo(Function<Persona2,Integer> compara) {
		// TODO Auto-generated method stub
		return compara.apply(this);
	}


	@Override
	public int compareTo(Persona2 o) {
		// TODO Auto-generated method stub
		return 0;
	}

	public String getDni() {
		return dni;
	}


	public void setDni(String dni) {
		this.dni = dni;
	}


	public String getNombre() {
		return nombre;
	}


	public void setNombre(String nombre) {
		this.nombre = nombre;
	}


	public String getApellidos() {
		return apellidos;
	}


	public void setApellidos(String apellidos) {
		this.apellidos = apellidos;
	}




   }


 public class CreaPersonasFuncionalCompare {
```

```
                    public CreaPersonasFuncionalCompare() {


                    }


                    public static Persona2 getPersonas() {

                            return new Persona2() ;

                    }



            public static void main(String[] args) {




                    Persona2 p1 =
CreaPersonasFuncionalCompare.getPersonas().
                                        cambiaDni("3342432")
                                        .cambiaNombre("Julian")
                                        .cambiaApellidos("Lopez");

                    Persona2 p2 =
CreaPersonasFuncionalCompare.getPersonas().
                                        cambiaDni("3342436L")
                                        .cambiaNombre("Luis")
                                        .cambiaApellidos("marquez");



System.out.println("Ahora cambiamos el comportamiento para ordenar por
Nombre");

 if ( p1.compareTo((p)->p.getNombre().compareTo(p2.getNombre()))==1)  {


System.out.println(" hemos cambiado el comportamiento de el Compare con una
expresion lambda\n "
                                             + p1.toString() + " Es mayor que "+
p2.toString());


                    }
else   if ( p1.compareTo((p)->p.getNombre().compareTo(p2.getNombre()))==0)  {


System.out.println(" hemos cambiado el comportamiento de el Compare con una
expresion lambda\n "
        + p1.toString() + " Es igual que "+ p2.toString());




                    }
```

```
else {


System.out.println(" hemos cambiado el comportamiento de el Compare con una
expresion lambda\n "
                + p1.toString() + " Es menor que "+ p2.toString());

                   }


                   System.out.println("Ahora cambiamos el comportamiento
para ordenar por DNI");


        if ( p1.compareTo((p)->p.getDni().compareTo(p2.getDni()))==1)  {

System.out.println(" hemos cambiado el comportamiento de el Compare con una
expresion lambda\n "
                + " para que compare por dni" +
         p1.toString() + " Es mayor que "+ p2.toString());


                   }
else   if ( p1.compareTo((p)->p.getDni().compareTo(p2.getDni()))==0) {


        System.out.println(" hemos cambiado el comportamiento de el Compare
con una expresion lambda\n "+
                           " para que compare por dni" +
p1.toString() + " Es igual que "+ p2.toString());

                   }
else {

System.out.println(" hemos cambiado el comportamiento de el Compare con una
expresion lambda\n "+
                                          " para que compare por dni" +
        p1.toString() + " Es menor que "+ p2.toString());
                   }


           }


         }
```

6.2.1 Practica independiente cambiando comportamiento alumnos

1. La clase alumnos implementará el interfaz Comparable.
2. Se modificará el método comparable para que reciba un Predicate que compare dos objetos alumnos
3. En el programa principal:
    a. Se crean dos alumnos
    b. Se comparan con nuestra nueva versión con compareTo por

DNI

c. Se comparan con nuestra nueva versión con compareTo por nombre

7 Recursividad

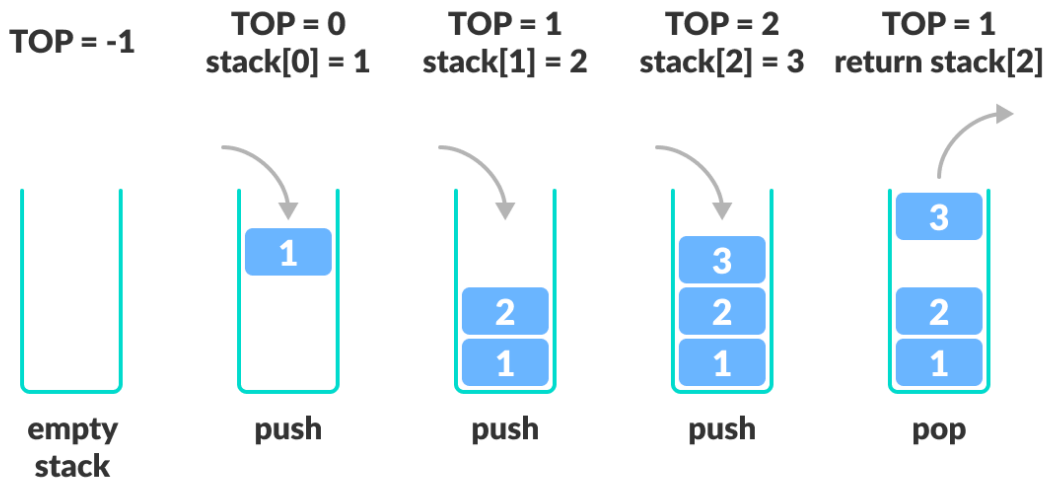
Una de las **herramientas de programación que se hace necesaria en la programación funcional** es el concepto de **recursividad**. En general, la recursividad es el proceso de **definir algo en términos de sí mismo y es algo similar a una definición circular**. El componente clave de un método recursivo es una declaración que ejecuta una llamada a sí mismo. La recursividad es un poderoso mecanismo de control.

El **proceso en el que una función se llama directa o indirectamente** se denomina **recursividad y la función correspondiente se llama función recursiva.** Usando un **algoritmo recursivo**, ciertos problemas **se pueden resolver con bastante facilidad.** Ejemplos de tales problemas son **Torres de Hanoi (TOH), recorridos** de **árboles**, y **algoritmos de búsqueda como quickshort**.

7.1 La pila de ejecución

7.1.1 ¿Qué es una pila?

Una **pila es una estructura de datos en java** que va colocando elementos uno encima de otros. Cuando hacemos **una operación push(), colocamos un elemento arriba**. Cuando **hacemos un pop(),** sacamos el elemento que **está arriba del todo.**



7.2 La Pila de ejecución en java

Cada vez **que llamamos a una función o método en uno de nuestros programas** desde una **clase o programa principal ese método va a tomar el control de la ejecución** en nuestro programa. Para **guardar su ejecución y el estado de todas sus variables y parámetros locale**s, **Java implementa**, al igual que otros compiladores, **una pila de ejecución o Runtime Stack** de tamaño limitado, para cada programa.

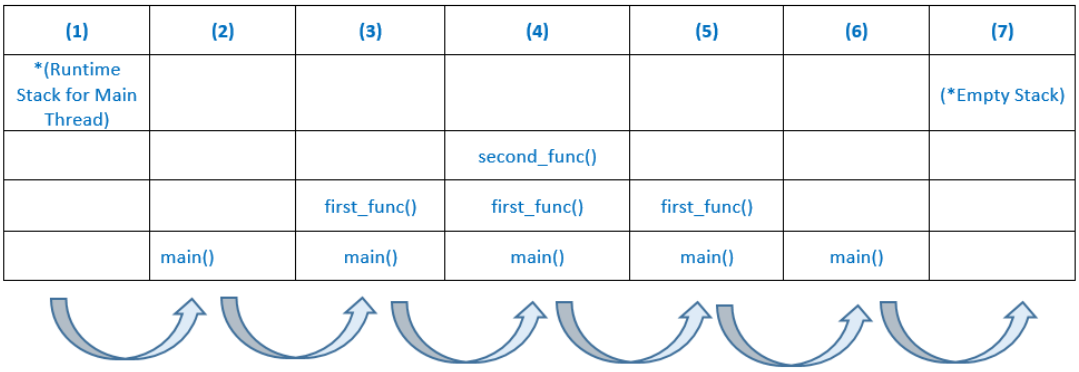
Para cada **proceso la máquina virtual java, JVM (Java Virtual Machine**) **creará una pila en tiempo de ejecución**. Todos y **cada uno de los métodos se insertarán en la pila**. Cada entrada **se denomina Registro de activación de marco de pila**.

Cada vez **que llamamos a un método o función, ese método o función Java crea una entrada en la pila**. Este mecanismo **se usa para saber en que ejecución** y **quien tiene el control de la ejecución de nuestro programa** en cada momento. Para **cada llamada** se **guarda en memoria una copia** de **sus variables locales y parámetros.**

En el ejemplo siguiente se ve de manera bastante clara. El primer elemento en ser colocado en la pila de ejecución es mi función main() del programa principal, es quien toma el control el (1**). Main() llama a first_func(), se hace una operación push en la pila** de **first_func()** y se coloca en la parte superior de la pila. El **método first_func() llama a second_func()** que toma el **control de la ejecución** y se coloca en la parte de arriba de la pila. Como veis el método que está en **la posicion superior de la pila es quien se esta ejecutando en cada momento,** tiene el control de la ejecución. **Cuando second_func() termina se hace un pop de second_func(),** se saca de la pila. El control vuelve a first_func(). La misma operación se hace cuando first_func() termina, se saca de la pila. Finalmente main() recupera el control. Termina de ejecutarse, y sale

de la pila.

Una vez finalizada, **Stack Frame se vaciará y la pila vacía será destruida** por la máquina virtual java, justo antes de la terminación del programa o hilo de ejecución.




```
/**
 *
 */
package com.test.java;

/**

 */
public class MecanismoEjecucionPilaJava {

	/**
	 * @param args
	 */
	public static void main(String[] args) {
		// TODO Auto-generated method stub
		first_func();
	}

	public static void first_func() {
		second_func();
	}

	public static void second_func() {
		System.out.println("Hello World");
	}

}
```

7.3 Recursividad. Practica guiada recursividad

¿Cuál es condición base en recursividad?

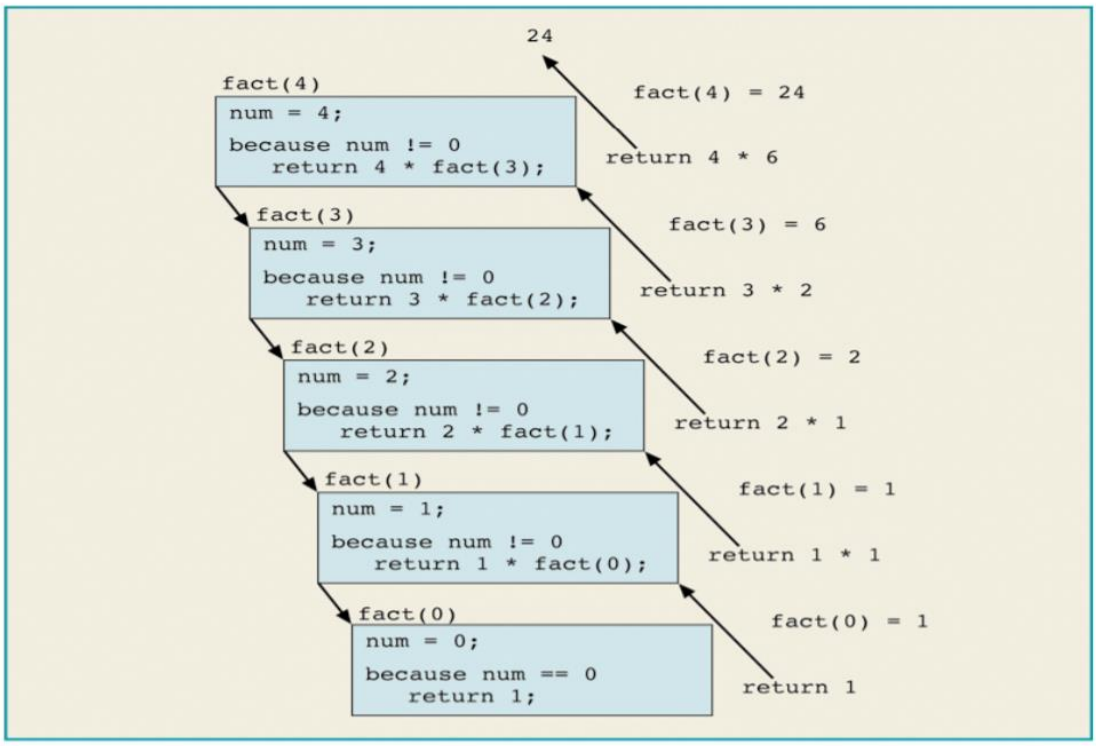
En el **programa recursivo**, se proporciona **la solución al caso base** y la **solución del problema más grande** se expresa en **términos de problemas más pequeños**.

```
public static Long factorial(long n)
	{
	    if (n <= 1) // base case
	        return 1;
	    else
	        return n*factorial(n-1);
	}
```

En el ejemplo anterior, **se define el caso base para n <= 1** y se puede resolver el problema **para un número mayor** con la **división del problema a números más pequeños** hasta que se alcance el caso base, factorial(1) que devueve 1.

En la solución anterior si **la aplicamos para el factorial(4**) la función nos devolvería **4*factoria(3).** Ahora **factorial(4) espera a que factorial(3) se resuelva**. Estamos en la **ejecución de factorial(3)**. La función **factorial(3) devuelve 3*factorial(2**). En este punto, factorial(3) espera a que se resuelva factorial(2), **factorial(2)** nos devolvería **2*factorial(1).** Es el mismo caso que el anterior, **factorial(2) debe esperar** a que **factorial(1) termine para devolver el resultado**.

En este punto hemos llegado **al caso base, factorial(1).** Cuando la llamada a factorial(1) termine, automáticamente **devolverá un 1**. Ahora tenemos que hacer el **recorrido inverso**. Factorial(2) esta esperando a que factorial(1) termine, cuando termina, **factorial(2) devolverá 2*1**. Ahora, **factorial(3) puede resolverse** recibe el resultado de factorial 2, un 2 y puede continuar, devolviendo el resultado **3*2**. Y finalmente, **factorial(4) ya tiene le resultado de factorial 3**, que es 6 y **termina su ejecución devolviendo 4*6, 24** Es un **enfoque diferente para resolver el problema**, que tradicionalmente resolveríamos con un bucle.



Factorial.java

```
public class Factorial {


	public static Long factorial(long n)
	{
	    if (n <= 1) // caso base
	        return Long.valueOf(1);
	    else
	        return n*factorial(n-1);
	}


	public static void main(String[] args) {


	    Scanner miScanner = new Scanner(System.in);


	    System.out.println("Escriba un número entero");
```

```
            Long numero1 = miScanner.nextLong();


            Long resultado = Factorial.factorial(numero1);




            System.out.println("El resultado del factorial de " +
numero1 + " es: "+ resultado);



        }


}
```

¿Por qué se produce un error de desbordamiento de pila en la recursividad? Si **el caso base no se alcanza o no está definido**, puede surgir el problema de **desbordamiento de pila.** Tomemos un ejemplo para entender esto. Si no hay caso base, no podemos **hacer la vuelta o marcha atrás**. Volver de **factorial(1)** y la **función** se quedaría **infinitamente llamándose a sí misma**, hasta que el **desborde el tamaño de la pila de ejecución java**. Si la **pila tiene un millón de entradas disponibles**, en la **llamada un millon y uno a factorial**, la pila se desborda. Se produce **la excepción stackoverflow** en Java.

```
Long factorial(Long n)
{
    // Puede causar overflow
    //
    if (n == 100)
        return Long.valueOf(1);


    else
        return n*fact(n-1);
}
```

Si se **llama a fact(10), llamará a fact(9), fact(8), fact(7**) y así sucesivamente, pero el **número nunca llegará a 100**. Por lo tanto, **el caso base no se alcanza**.

Si estas funciones agotan la memoria en la pila, **se producirá un error de desbordamiento de pila.**

¿Cuál es la diferencia entre la recursividad directa e indirecta?

Una función se denomina recursiva directa si llama a la misma función. Una función se denomina recursiva indirecta `indirectaRecFun1` si llama **a otra función nueva** `indirectaRecFun2`. La función `indirectaRecFun2` volverá a llamar a `indirectaRecFun1`. Y así **hasta que alguna de ellas alcance** el **caso base** como en una partida de ping pong, **ida y vuelta de una función a otra.** La diferencia entre recursividad directa e indirecta se ha ilustrado en el Cuadro 1.

Recursividad directa

```
void directaRecFun()
{
    // Some code....

    directaRecFun();

    // Some code...
}
```

Recursividad Indirecta

```
void indirectaRecFun1()
{
    // codigo...

    indirectaRecFun2();

    // codigo...
}

void indirectaRecFun2()
```

```
{
    // código...

    indirectaRecFun1();

    // código
}
```

Como veis en la recursividad indirecta, tenemos dos funciones. Una se llama a la otra a la vez que se hace el problema más pequeño hasta llegar al caso base en una de ellas.

¿Cuál es la diferencia entre la recursividad de cola y la de cabecera?

Una función recursiva es recursiva de cola cuando la **llamada recursiva es la última instrucción ejecutada por la función**.

Por ejemplo, en el ejemplo **anterior lo último que hace factorial es llamada recursiva**:

return n*factorial(n-1). De cola

El ejemplo siguiente la recursividad es de cabecera. Porque **se hace la llamada recursiva**, y después **se ejecutan más instrucciones en nuestro código**.

```
            // Instruccion 2
            printFuncion(test - 1);

            System.out.printf("%d ", test);
            return;
```

¿Cómo se asigna la memoria a diferentes llamadas de función en recursividad?

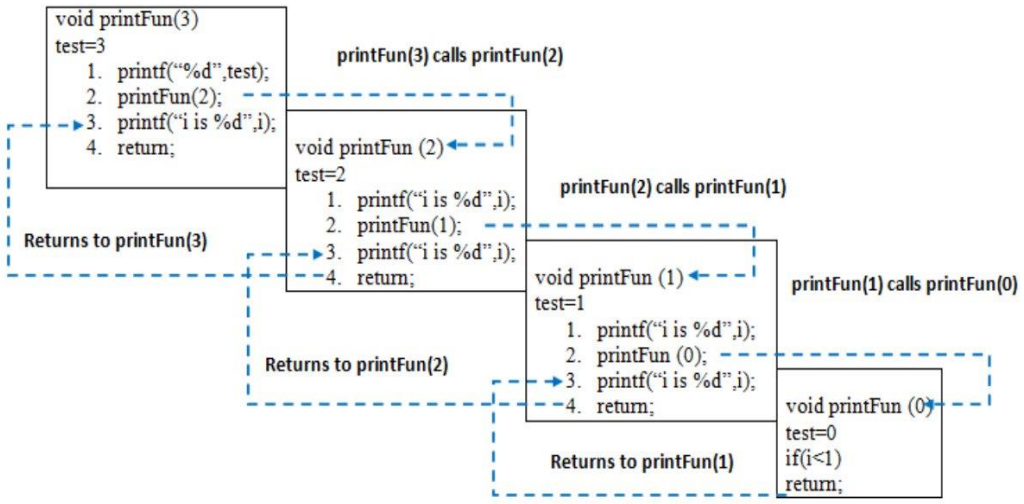
Cuando se llama a cualquier función desde main(), se le **asigna memoria en la pila**. **Una función recursiva se llama a sí misma**, la **memoria de la función llamada se asigna encima de la memoria asignada a la función de llamada** y **se crea una copia diferente de las variables locales para cada llamada de función**. Cuando se alcanza el caso base, **la función devuelve su valor a la función por la que se llama** y se desea asignar memoria y el proceso continúa.

Tomemos el ejemplo de cómo funciona la recursividad tomando una función simple.

```
public class ImprimirRecursivo {
    static void printFun(int test)
    {
        if (test < 1)
            return;

        else {
            System.out.printf("%d ", test);



            printFun(test - 1);

            System.out.printf("%d ", test);
            return;
        }
    }

    public static void main(String[] args)
    {
        int test = 3;
        printFuncion(test);
    }
}
```

Cuando se llama a **printFun(3)** desde main(), la memoria se asigna a **printFun(3)** y una la variable local test se inicializa en 3 y la instrucción 1 a 4 se inserta en la pila como se muestra en el diagrama siguiente. Primero imprime '3'. En la instrucción 2, se llama a **printFun(2)** y se asigna memoria a **printFun(2)** y variable local test se inicializa en 2 y la instrucción 1 a 4 se inserta en la pila. De forma similar,

printFun(2) llama a **printFun(1)** y **printFun(1)** llama a **printFun(0)**. **printFun(0)** va a la instrucción if y vuelve a **printFun(1)**. Las instrucciones restantes de **printFun(1)** se ejecutan y vuelve a **printFun(2)** y así sucesivamente. En la salida, se imprime el valor de 3 a 1 y, a continuación, se imprimen de 1 a 3. La pila de memoria se muestra en la imagen siguiente.



¿Cuáles son las desventajas de la programación recursiva sobre la programación iterativa?

Tenga en cuenta que **tanto los programas recursivos como los iterativos tienen la misma capacidad de resolución de problemas**, es decir, **cada programa recursivo se puede escribir de forma iterativa y viceversa** también es cierto. **El programa recursivo tiene mayores requisitos de espacio** que **el programa iterativo**, ya que **todas las funciones permanecerán en la pila** hasta que **se alcance el caso base**. También **tiene mayores requisitos de tiempo** debido a las llamadas de función se meten y sacan de la pila. **Se pierde tiempo** en esto, y **se ocupa más memoria**.

¿Cuáles son las ventajas de la programación recursiva sobre la programación iterativa?

La **recursividad proporciona una forma limpia y sencilla de escribir código**. Algunos problemas **son inherentemente recursivos** como **recorridos de árboles, Torre de Hanoi,etc.** Para estos problemas, **se prefiere escribir código recursivo**. Podemos escribir **estos códigos también iterativamente** con la ayuda de **una estructura de datos de pila**.

7.3.1 Ejercicios. Practica independiente recursividad

1. Calcular la serie de fibonnaci de manera recursiva.

   La sucesión de Fibonacci es conocida desde hace miles de años, pero fue Fibonacci (Leonardo de Pisa) quien la dio a conocer al utilizarla para resolver un problema.
   El **primer** y **segundo** término de la sucesión son

$$a_0 = 0$$

$$a_1 = 1$$

Los siguientes términos se obtienen sumando los dos términos que les preceden:

El **tercer término** de la sucesión es

$$a_2 = a_0 + a_1 =$$

$$= 0 + 1 = 1$$

El **cuarto término** es

$$a_3 = a_1 + a_2 =$$

$$= 1 + 1 = 2$$

El **quinto término** es

$$a_4 = a_2 + a_3 =$$

$$= 1 + 2 = 3$$

El **sexto término** es

$$a_5 = a_3 + a_4 =$$

$$= 2 + 3 = 5$$

El $(n+1)$**(n+1)-ésimo término** es.

2. **Para calcular el máximo común divisor de dos números** enteros puedo aplicar **el algoritmo de Euclides**, que consiste **en ir restando el más pequeño del más grande** hasta que **queden dos números iguales**, que serán el máximo común divisor de los dos números. Si comenzamos con el par de números 412 y 184, tendríamos:

412	228	44	44	44	44	44	36	28	20	12	8	**4**
184	184	184	140	96	52	8	8	8	8	8	4	**4**

Es decir, m.c.d.(412, 184)=4

Nota: Todavía es pronto para introducir **el tema de recursividad con expresiones lambda** lo **haremos en temas posteriores**, porque es un poco más complejo.