Unidad 6. Java Funcional

Contenido

1 Interfaces funcionales y lambdas. Tipos complejos en Java

Recordamos de nuestro conocimiento en teoría del lenguaje **tres paradigmas de los lenguajes de programación: imperativo, funcional y declarativo.** Java ha sido tradicionalmente un lenguaje de programación imperativo o declarativo, basado en la mutabilidad, asignando valores a variables y transportando o transformando datos de una variable o estructura de datos a otra.

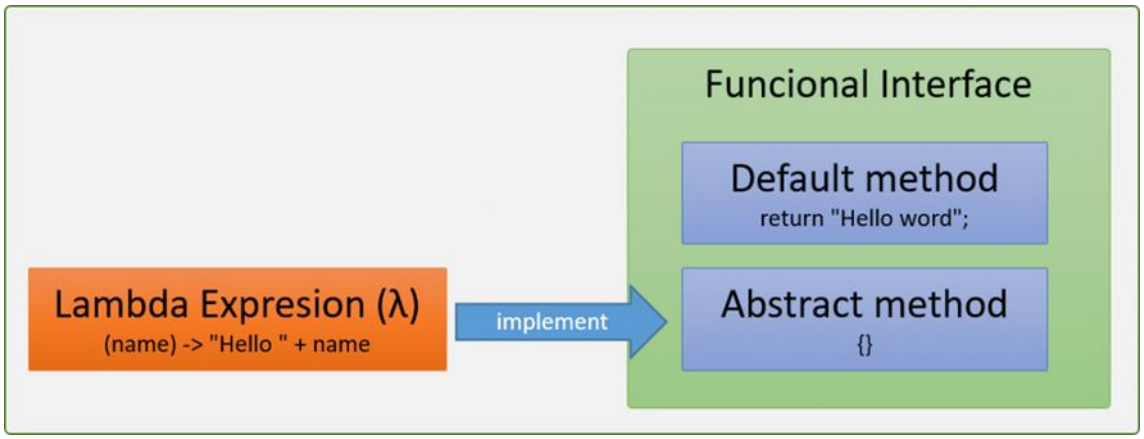
El paradigma funcional se basa en el concepto matemático de función**.** Los programas escritos **en lenguajes funcionales** consistirán en un conjunto de definiciones de funciones (entendiendo estas no como subprogramas clásicos de un lenguaje imperativo) junto con los **argumentos** sobre los que se aplican.

A partir de la versión 8 de Java, un nuevo paradigma de programación está disponible en Java. Es la **programación funcional.** Realizar código usando funciones y listas únicas o básicamente como si estuviéramos usando un lenguaje funcional como Lisp. Esta nueva forma de programar se basa en:

- **Interfaces funcionales:** interfaces que ofrecen sólo un método abstracto. Es decir, hacen un trabajo.

- **Expresiones Lambda**: Se **basan en el cálculo lambda.** La clave es la declaración de funciones **anónimas.** En el cálculo lambda, una función se puede declarar de forma anónima. **Por ejemplo, Square(x)x^2** se puede definir como $x.x^2$ o $(x) \rightarrow x^2$. De esta manera definiremos nuestras **funciones en** java. Y resolveremos nuestros algoritmos usando básicamente funciones.

El objetivo es **sobrescribir ese método abstracto proporcionado por la interfaz funcional** con una expresión **Lambda.** Si la interfaz no es funcional, no se le puede pasar una expresión lambda. Ahorramos instrucciones y líneas de código haciéndolo de esta manera. Se puede decir que la expresión lambda representa una función anónima que se contribuye a la interfaz funcional que define la función anónima. Aportan una acción real a la definición



1.1 Clases anónimas

Una **clase interna anónima** es una forma de clase interna que se declara e instancia con una **sola declaración.** Como resultado, **no hay ningún nombre para la clase que se pueda usar** en otra parte del programa; Quiero decir, es anónimo.

Las **clases anónimas se suelen usar en situaciones en las** que es necesario poder crear una **clase ligera** que se pase como parámetro **o argumento.** Esto generalmente se hace con una interfaz.

Por ejemplo, en el siguiente subapartado, crearemos una clase anónima a partir de otra clase y de una interfaz. Esta práctica está ampliamente extendida en Java hoy en día.

1.2 Clases anónimas a partir de una Clase. Constructores

La idea es lograr la herencia evitando la necesidad de agregar un archivo .java y escribir una nueva clase. A veces, necesitamos una nueva clase que modifique ligeramente la clase principal. En este escenario, no necesitamos escribir el código para una nueva clase, podemos crear la clase en tiempo de ejecución como se mostrará en el siguiente ejemplo. Creamos clases anónimas mediante la anulación de algunos de los métodos de clase principal en tiempo de ejecución.

En el siguiente programa, puede encontrar una clase denominada BaseClassParaAnonima. Contiene un constructor y el método metodoNombre. Lo que se intenta en el programa es crear un nuevo objeto de una nueva clase anónima. Para obtenerlo seguimos estos pasos.

1. Creamos una variable ltipo de la clase, BaseClassParaAnonima.

   ```
   BaseClassParaAnonima anon
   ```

2. Usamos el operador new para invocar el constructor BaseClassParaAnonima

   ```
   new BaseClassParaAnonima()
   ```

3. Llegado a este punto, Java permite sobrescribir dinámicamente el método de la clase padre metodoNombre.

```
{
                @Override
  public void metodoNombre(String name) {

                System.out.println("Clase anonima creada sobreescrita:" +
name);
        }

    };
```

Teniendo en cuenta el `BaseClassParaAnonima` resulta que el nuevo objeto creado es diferente de uno creado a partir de la clase original. La razón es que los métodos `metodoNombre` son diferentes. Como resultado, tenemos una nueva clase de objeto, pero esta nueva clase no tiene nombre, una clase de anónimo.

El método para un objeto de la clase `BaseClassParaAnonima` es

```
public void metodoNombre(String nombre) {

                System.out.println("Clase creada normal: " + nombre);
```

```
        }
```

El nuevo objeto creado tiene este método metodoNombre:

```
      @Override
        public void metodoNombre(String name) {

                    System.out.println("Clase anonima creada sobreescrita:" +
name);
                }
```

Como puedes comprobar son diferentes. Las clases anónimas están destinadas a crear clases a partir de interfaces como explicaremos en el siguiente ejemplo.

```
      BaseClassParaAnonima anon = new BaseClassParaAnonima() {
              @Override
              public void metodoNombre(String name) {

                    System.out.println("Clase anonima creada sobreescrita:" +
name);
              }


          };
```

BaseClassParaAnonima.java

```
package anonimas;
public class BaseClassParaAnonima {


            public BaseClassParaAnonima () {


            }

            public void metodoNombre(String nombre) {

                  System.out.println("Clase creada normal: " + nombre);
            }


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

```
            BaseClassParaAnonima objetoNormal = new BaseClassParaAnonima();
            objetoNormal.metodoNombre("Betty");
            BaseClassParaAnonima anon = new BaseClassParaAnonima() {
              @Override
              public void metodoNombre(String name) {
                  System.out.println("Clase anonima creada sobreescrita:" +
name);
              }

          };
            anon.metodoNombre("Mildred");
        }
    }
```

1.2.1 Clases anónimas para interfaces.

Este caso es el más habitual en la programación Java. Las últimas incorporaciones del marco de API de Java, el marco del SDK de Android (para desarrolladores de Android), están diseñadas para crear clases anónimas a partir de interfaces. El procedimiento es similar al ejemplo anterior:

a. Se ha definido una interfaz `InterfaceAnonimo` en este ejemplo.

```
interface InterfaceAnonimo {
            public void metodoNombre(String name);
    }
```

b. Se declara una variable de tipo Interfaz. Este concepto ya ha sido introducido anteriormente

```
InterfaceAnonimo anon =
```

c. Java permite llamadas a una interfaz inexistente constructores (interfaces no tienen constructor) si anulamos los métodos abstractos de esta interfaz

```
new InterfaceAnonimo() {
```

d. En tiempo de ejecución el método metodoNombre es implementado por la clase anónima.

```
@Override
                    public void metodoNombre(String name) {

                           System.out.println("Sobreescribimos el método de
manera anónima:" + name);
                    }
```

e. De nuevo, el resultado es un objeto de la nueva clase anónima creado a partir de un interfaz. Además, no tiene nombre, por tanto, es anónimo.

```
package anonimas;
public class ClasesAnonimasDesdeInterfaz {

	interface InterfaceAnonimo {

		public void metodoNombre(String name);

	}

	public static void main(String[] args) {

		InterfaceAnonimo anon = new InterfaceAnonimo() {
			@Override
                    public void metodoNombre(String name) {

                           System.out.println("Sobreescribimos el método de
manera anónima:" + name);
                    }

		};

		anon.metodoNombre("Mildred");

	}

}
```

Las clases y objetos anónimos son un concepto clave para desarrollar expresiones lambda en Java. En el siguiente capítulo vamos a presentar interfaces funcionales y expresiones lambda. Las expresiones de Lambda son una nueva forma de invalidar métodos y crear clases anónimas.

1.3 Interfaces funcionales

Como se mencionó anteriormente, una interfaz funcional es una interfaz que especifica solo un método abstracto. Antes de continuar, recuerde que **no todos los métodos de interfaz son abstractos.** A partir de JDK 8, una interfaz puede tener uno o más métodos predeterminados. **Los métodos predeterminados no son** abstractos. Tampoco lo son los métodos de **interfaz estática o privada.** Por lo tanto, un método de interfaz es abstracto sólo si no especifica una implementación.

Esto significa que una interfaz funcional puede incluir métodos predeterminados, estáticos o privados, pero en todos los casos debe **tener un único método abstracto.** Debido a que los **métodos de interfaz no predeterminados, no estáticos y no privados son implícitamente abstractos,** no es necesario usar el **modificador** abstract (aunque se puede especificarlo, si lo desea).

En definitiva, **son las herramientas para soportar expresiones lambda** con el fin de llevar a cabo este nuevo paradigma **en Java.** En el caso de las interfaces, el concepto es representar un tipo único de operación como veremos más adelante.

Fíjate en las interfaces y clases que implementan aquellas interfaces que ya han existido en versiones anteriores de Java o con un solo **método como Listener, Runnable, Callable o Comparable.** ¿Podemos crear una clase anónimas a partir de estas interfaces? Claro.

```
Runnable thread = new Thread("Nuevo Hilo ") {
      public void run(){
        System.out.println("run by: " + getName());
       }
   };
```

```
new ActionListener() {
                public void actionPerformed(ActionEvent arg0) {

                        System.out.println("Botón pulsado:");

                }
        });
```

Todavía no hemos abordado las expresiones lambda. Una de sus funciones es sobreescribir los métodos de interfaz funcional. ¿Podemos declarar o sobrescribir **estas interfaces** y objetos de forma anónima con una expresión lambda? Sí.

Hilos Runnable con una expresión lambda.

```
Runnable threadLambda = () -> System.out.println("ejecutado por " +
getName());
```

Es exactamente lo mismo que el siguiente código

```
Runnable thread = new Thread() {
      @Override
      public void run(){
        System.out.println("ejecutado por: " + getName());
      }
   };
```

Y este código realiza la misma acción. Están sobreescribiendo el método run y creando una clase anónima Runnable tipada.

```
Runnable threadLambda = () -> System.out.println("ejecutado por: " +
getName());
```

Esas interfaces anteriores no son interfaces funcionales perse. Sin embargo, pueden comportarse como interfaces funcionales en Java. Su nombre, interfaces legacy.

Como ya lo hemos descrito, una interfaz funcional es una interfaz que solo ofrece un método abstracto. Podemos etiquetarlos con la etiqueta de interfaz funcional @FunctionalInterface. Aunque es opcional, es recomendable.

Mostramos un ejemplo de una interfaz funcional. Incluso si la interfaz incluye tres métodos, solo uno de ellos es abstracto: `double operation(double a, double b);`

Los otros dos son predeterminados default:

```
default double identity(double num) {

                return num;
        }
```

Y estático, static:

```
public static int transformaAInteger(double num) {

                return (int) num;
        }
```

Se puede apreciar que tienen cuerpo. Podemos implementar métodos en interfaces de estos dos tipos.

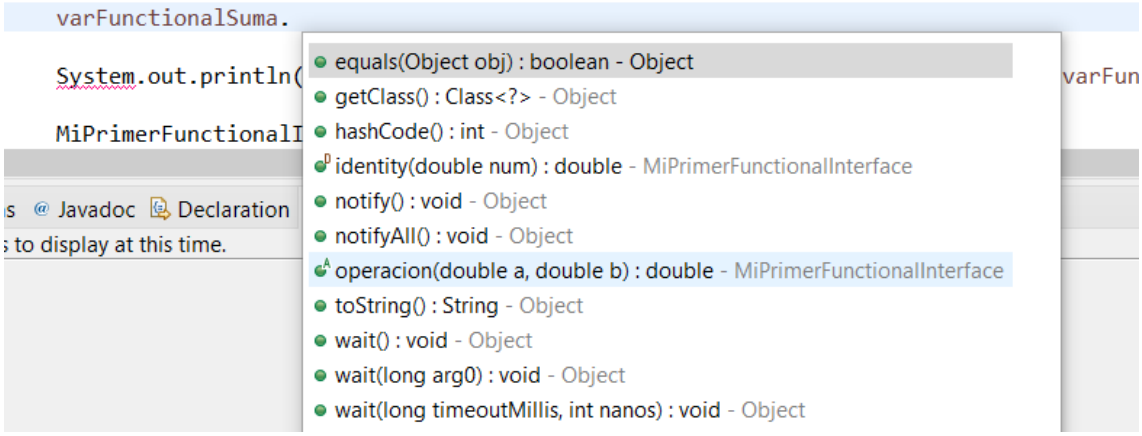
```
@FunctionalInterface
	interface MiPrimerFunctionalInterface {

		double operacion(double a, double b);

		default double identity(double num) {

			return num;
		}

		public static int transformaAInteger(double num) {

			return (int) num;
		}

	}
```

Para concluir esta exposición, cualquier clase que implemente esta interfaz solo tiene que invalidar un método, operacion.

Una vez explicada la interfaz funcional, mira cómo creamos la clase anónima, ya que seguimos el mismo patrón que anteriormente.

```
MiPrimerFunctionalInterface varFunctionalSuma = new
MiPrimerFunctionalInterface() {

			@Override
			public double operacion(double a, double b) {
				// TODO Auto-generated method stub
				return a + b;
			}
		};
```

Ahora tenemos un objeto Anónimo almacenado en el varFunctionalSuma. El objeto sigue la estructura definida por la interfaz. Hereda métodos de la clase Object y también del método predeterminado definido en MiPrimerFunctionalInterface:



Se puede llamar al método operación sobrescrito, que realizará una suma:

```
System.out.println("La operacion definida en la clase anónima es la suma:"
 + varFunctionalSuma.operacion(5, 7));
```

Lo que es nuevo es el uso de la expresión lambda. Como se mencionó anteriormente, las expresiones lambda anulan el método abstracto de una interfaz. Por lo tanto, si llamamos a la operación del método varFunctionalProduct, se puede inferir que calculará el producto de estos dos números pasados como argumentos a la operación del método.

```
MiPrimerFunctionalInterface varFunctionalProduct = (x, y) -> x * y;

            System.out.println("La operación definida en la expresion lambda
producto da como resultado:"
                        + varFunctionalProduct.operacion(5, 7));
```

No te asustes, te lo explicaremos en el apartado siguiente.

El resultado de la consola de ejecución para este programa:

```
La operación definida en la clase anónima es la suma:12.0
La operación definida en la expresión lambda producto da como resultado:35.0
```

FunctionalInterfaceEjemplo.java

```
package introduccionprogramacionfuncional;

public class FunctionalInterfaceEjemplo {
```

```
	@FunctionalInterface
	interface MiPrimerFunctionalInterface {

		double operacion(double a, double b);

		default double identity(double num) {

			return num;
		}

		public static int transformaAInteger(double num) {

			return (int) num;
		}

	}

	public static void main(String[] args) {

		MiPrimerFunctionalInterface varFunctionalSuma = new
MiPrimerFunctionalInterface() {

			@Override
			public double operacion(double a, double b) {
				// TODO Auto-generated method stub
				return a + b;
			}
		};

		System.out.println("La operacion definida en la clase anónima es
la suma:" + varFunctionalSuma.operacion(5, 7));

	MiPrimerFunctionalInterface varFunctionalProduct = (x, y) -> x * y;

		System.out.println("La operación definida en la expresion lambda
producto da como resultado:"
				+ varFunctionalProduct.operacion(5, 7));

	}

}
```

1.4 Expresiones Lambda.

La expresión lambda se basa en un elemento de sintaxis y un operador que difieren de lo que hemos visto en los temas anteriores. El operador, a veces

llamado **operador lambda** u operador **de flecha**, es **->**.

Este operador divide una expresión lambda en dos partes:

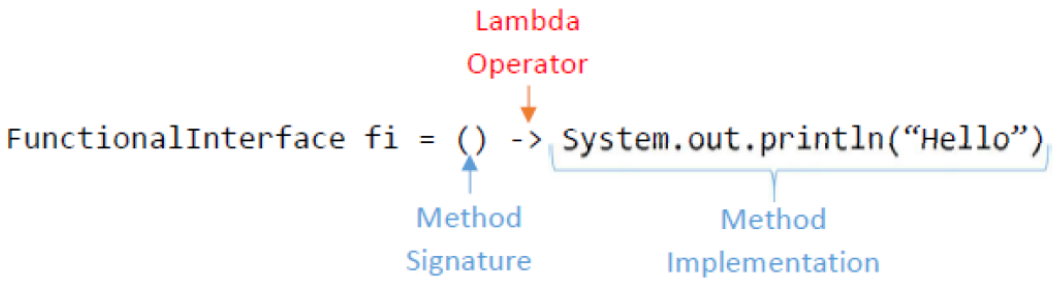
- **El lado izquierdo** especifica los argumentos requeridos por la expresión lambda.
- **En el lado derecho** está el cuerpo lambda, que especifica las acciones de la expresión lambda.

Java define dos tipos de cuerpos lambda. El primer tipo consiste **en una sola expresión, y el otro tipo consiste en** un bloque de **instrucciones** o bloque de **código.** Comenzaremos con lambdas que **definen la expresión única**.

Expresión Lambda simple

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

1. La signatura o cabecera del método
2. El operador lambda
3. Implementación del método



Volvamos al ejemplo anterior:

```
interface MiPrimerFunctionalInterface {

		double operacion(double a, double b);

		default double identity(double num) {

			return num;
		}
```

```
c varFunctionalProduct = (x,y)-> x*y;
```

Repasando la expresión lambda que implementó la operación del método, podemos concluir que la firma de expresión lambda coincide con la firma de

interfaz funcional. La aridad de expresión lambda difiere de la aridad del método de operación en la declaración de tipo. Mientras que las expresiones lambda no definen tipos, las interfaces sí los definen. Además, el método tiene un nombre, pero la expresión lambda no. Resolvamos este enigma.

Las expresiones lambda se basan en el cálculo lambda. Esta rama del cálculo se caracteriza por la definición de funciones anónimas. Tradicionalmente en matemáticas, las funciones se definen como f(x)= x+1;, name(variable)= expresión matemática. Por el contrario, lambda calculas permite funciones anónimas sin nombre. Para declararlos, el cálculo lambda proporciona dos notaciones:

a. λx.x+1
b. x->x+1

Algunos de los lenguajes de programación, incluido Java, utilizan la segunda notación para declarar sus propias funciones anónimas.

En el ejemplo anterior, la interfaz funcional `MiPrimerFunctionalInterface` actúa como una plantilla para las clases en general. Además, es una plantilla para todas las expresiones lambda que puede declarar siguiendo el patrón de interfaz. Al hacerlo, no es necesario declarar tipos en las expresiones lambda teniendo en cuenta que los tipos ya están declarados en la interfaz. El compilador infiere que los argumentos (x,y) son de doble tipo desde la operación del método abstracto, escríbalos como doble: `double operacion(double a, double b);`.

Siguiendo esta línea de pensamiento, x pares con doble a, y pares con doble b. Queda algo para resolver el dilema, y ese es el tipo de retorno. El tipo de retorno para la operación del método es double. Aunque no hay una instrucción return en las expresiones lambda, el compilador asume que la expresión lambda devuelve el resultado de la instrucción java `x*y.` La instrucción de retorno está implícita en el código. Esta coincidencia de patrones funciona para expresiones lambda de una sola línea. Como demostraremos más adelante, para las lambdas de instrucción de bloque, necesitamos agregar una instrucción return. Vamos a mostrarlo con la siguiente lambda que puede agregar al final del Ejemplo `FunctionalInterfaceEjemplo`:

Expresiones Lambda de Bloque

En este caso, es obligatorio insertar el boque de instrucciones entre corchetes y declarar explícitamente la sentencia de retorno al final de la instrucción de bloque, siempre que el método sobrescrito devuelva un resultado.

```
MiPrimerFunctionalInterface varFunctionalbloque = (x,y)->  { x=2*x ; return x*y; };
```

El sintaxis para una expresión de bloque lambda es:

(param1, param2…., paramn) -> {

Instruccion 1;
Instruccion 2
…
Instruccion n;
Return Instruccion; Sólo si la interfaz funcional define un tipo de retorno.

}

La evidencia presentada nos enseña que una expresión lambda es similar a una función. Las diferencias surgen de la naturaleza anónima de lambdas, y la declaración implícita de tipos, que se infiere de la interfaz funcional relacionada con la lambda.

Se mostrará otro ejemplo para completar esta sección. Intentaremos escribir el código para una expresión lambda que muestre en la consola solo el subconjunto de los números pares de los n primeros números.

El primer paso a dar es declarar la interfaz funcional necesaria para crear una variable adecuada para la lambda. Como señalamos, la lambda no necesita devolver un valor, ya que su trabajo es mostrar números pares, por lo que el método abstracto en la interfaz devuelve void, `void displayEvenNumbers(int n);`.

Como resultado del hecho de que necesitamos imprimir en la consola el subconjunto de los números pares, a partir de los n primeros números, los métodos requieren n como parámetro.

```
@FunctionalInterface
	interface NumerosImpares {

		void muestraNumerosImpares(int n);

	}
```

El siguiente paso es escribir el algoritmo en la expresión lambda. Declaramos una variable de interfaz NumerosImpares. Después de eso, asignamos la lambda. La lambda recibe n como argumento. El bucle for itera n veces y solo imprime el número si i%2==0, que significa el resto de dividir la variable I por

2 es cero. Recuerde que un número par es un número divisible por 2.

```
		NumerosImpares evenLambda = (n) -> {

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

				if (i%2==0)
					System.out.println("El número " + i + " es
impar.");

			}


		};
```

Puede darse cuenta de que este lambda de bloque no tiene una instrucción return ya que el método displayEvenNumber devuelve void. Y luego, para probar esta lambda, necesitamos hacer una llamada al método muestraNumerosImpares.

```
evenLambda.muestraNumerosImpares (10);
```

Ejecucion

```
El número 0 es impar.
El número 2 es impar.
El número 4 es impar.
El número 6 es impar.
El número 8 es impar.
El número 10 es impar.
```

EjemploNumerosImpares.java

```
package introduccionprogramacionfuncional;
public class EjemploNumerosImpares {

	@FunctionalInterface
	interface NumerosImpares {

		void muestraNumerosImpares(int n);

	}


	public static void main(String[] args) {


		NumerosImpares evenLambda = (n) -> {
```

```
                    for (int i=0; i<=n ; i++) {

                          if (i%2==0)
                                System.out.println("El número " + i + " es
impar.");

                    }


             };


             evenLambda.muestraNumerosImpares(10);

       }


}
```

Ejercicio

La idea de esta práctica es hacer que la interfaz funcional sea genérica y probar el algoritmo para Long y Float.

```
@FunctionalInterface
       interface NumerosImpares<T> {

             void muestraNumerosImpares(T n);

       }
```

1.5 Interfaces predefinidos en Java

En nuestros programas anteriores, para definir una lambda necesitábamos declarar y definir una Interfaz Funcional. Para evitar eso, la API de Java ofrece interfaces funcionales predefinidas. Ahorra tiempo al desarrollador y reduce la cantidad de código para nuestros programas. En esta sección presentaremos algunas de las interfaces funcionales Java más útiles.

Revisaremos algunas de las interfaces funcionales predefinidas más útiles en Java. El paquete java.util.function contiene todas estas interfaces funcionales, esto necesitamos importarlas ya sea que las utilicemos en nuestros programas.

La documentación oficial de este paquete se encuentra en el siguiente enlace:

https://docs.oracle.com/en/java/javase/14/docs/api/java.base/java/util/function/package-summary.html

1.5.1 Composición de funciones

Uno de los puntos clave de la programación funcional o declarativa es la composición de funciones. Este concepto deriva del concepto matemático de composición. La mayoría de las interfaces funcionales predefinidas ofrecen la posibilidad de composición. Teniendo en cuenta eso, debemos explicar el concepto desde un punto de vista matemático.

Vamos a definir las siguientes funciones:

F(x)= x+1
G(x)= 2x

F(2) podría evaluarse como 3 (2+1)
Los valores de G(2) son 4 (2x2)

La composición matemática permite que el matemático combine funciones. Entonces, si componemos FoG(X) la evaluación matemática de FoG(2) debería funcionar como:

Primero evaluamos la función más interna G(x), G(2), el valor de retorno es 4. Usamos ese 4 como entrada para F(x) en la composición, luego debemos evaluar F(4) = 5. Como resultado, la composición de FoG(2) devuelve como valor a 5.

Ejercicio

¿Qué pasa si componemos las funciones en el orden opuesto GoF(x)?
¿Cuál sería el valor devuelto de GoF(3)?

1.5.2 Interfaz funcional de predicado

Esta interfaz se utiliza para aplicar operaciones de selección o filtrado. Los predicados en Java se implementan con **interfaces. Predicado<T>**es una interfaz funcional genérica que representa una función de argumento único que devuelve un valor booleano. Se encuentra en el paquete java.util.function. Contiene un **método test(T t)** que evalúa el predicado **dado** el argumento. Este es el método **para sobrescribir por la expresión lambda.**

Representa el tipo de operación que definiríamos como condicional a los parámetros que recibe. **Es deber del programador definir esa operación utilizando expresiones lambda** o funciones anónimas.

La definición de interfaz en la API de Java es Predicate<T>. Se puede inferir aquí que el predicado es una interfaz genérica. Funciona con diferentes tipos.

```
@FunctionalInterface
public interface Predicate<T>
```

Ofrece los métodos siguientes

default **Predicate**<**T**>	**and**(**Predicate**<? super **T**> other) Devuelve un predicado compuesto que representa un AND lógico de cortocircuito de este predicado y otro.
static <T> **Predicate**<T>	**isEqual**(**Object** targetRef) Devuelve un predicado que prueba si dos argumentos son iguales según**Objects.equals(Object, Object)**.
default **Predicate**<**T**>	**negate**() Devuelve un predicado que representa la negación lógica de este predicado.
default **Predicate**<**T**>	**or**(**Predicate**<? super **T**> other) Devuelve un predicado compuesto que representa un OR lógico de este predicado y el pasado como parámetro.
boolean	**test**(**T** t)

En la tabla, se señala que los únicos métodos abstractos a implementar es la prueba, debido a la naturaleza de la interfaz funcional del predicado. Los otros métodos son predeterminados y ofrecen funcionalidad adicional al predicado, como, por ejemplo, la capacidad de combinar algunas variables de predicado. La combinación de interfaces funcionales es el resultado de aplicar el concepto matemático de composición de funciones, que detallaremos más adelante.

En el siguiente fragmento de código, estamos definiendo un predicado que verifica si un número es mayor que 10. También declaramos otro que comprueba si un número es menor que 20. Finalmente, combinamos ambos con los métodos ofrecidos por la interfaz de predicados para combinar interfaces de predicados.

Como se mencionó anteriormente, la interfaz Predicate pertenece al paquete java.util.function. Para usarlo en nuestro programa necesitamos importarlo.

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

En segundo lugar, estamos definiendo una variable Predicate<Integer>mayorque10. El objetivo de la implementación es crear una lambda que pruebe si un número es mayor que 10. A partir de entonces, necesitamos declarar un predicado numérico, como Integer. En la declaración en línea del lado derecho tenemos la expresión lambda (n)-> n>10; . Recibe un número entero como parámetro y se devuelve el tipo booleano ya que estamos utilizando un operador de comparación.

Finalmente, estamos probando el predicado usando el método de prueba. El método de prueba recibe un Integer ya que hemos declarado el predicado como Integer Predicate<Integer>. Además, cuando llamamos al método de prueba lambda (n)-> n>10; porque el lambda anula el método de prueba en las interfaces funcionales de Predicates.

```
            Predicate<Integer> mayorque10 = (n)-> n>10;


            System.out.println("¿Es el siguiente número  " + numero + "
mayor que 10? "+
                        mayorque10.test(numero));
```

Resultado de la ejecución:

```
Introduce un numero entero
13
¿Es el siguiente número  13 mayor que 10? true
```

Para 13, el método de prueba devuelve true, siguiendo la implementación de lambda

(13)->13>10 y 13>10 se evalúa como verdadero.

Notas Cornell:

Comenta el siguiente código y explica cómo funciona

```
Predicate<Integer> menorque20 = (n)-> n<20;

            System.out.println("¿Es el siguiente número  " + numero + "
menor que 20? "+
                        menorque20.test(numero));
```

PredicateInterface.java

```
package introduccionprogramacionfuncional;

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

public class PredicateInterface {

      public static void main(String[] args) {


            Scanner sc = new Scanner(System.in);

            System.out.println("Introduce un numero entero");


            int numero = sc.nextInt();

            Predicate<Integer> mayorque10 = (n)-> n>10;


            System.out.println("¿Es el siguiente número  " + numero + "
mayor que 10? "+
                        mayorque10.test(numero));


            Predicate<Integer> menorque20 = (n)-> n<20;

            System.out.println("¿Es el siguiente número  " + numero + "
menor que 20? "+
                        menorque20.test(numero));



            Predicate<Integer> and = mayorque10.and(menorque20);

            System.out.println("¿Es el siguiente número  " + numero + "
mayor que 10 "+
                         " y " + " menorque20? " + and.test(numero));


            Predicate<Integer> or = mayorque10.or(menorque20);
```

```
            System.out.println("¿Es el siguiente número " + numero + " mayor
que 10 "+
                               " 0 " + " menor que 20? " + and.test(numero));

      }

}
```

Composición de Predicates

Para completar esta sección vamos a explicar la última parte del código. Estamos usando el método predeterminado y, para combinar a predicados y producir uno nuevo, Predicate<Integer>and = mayorque10. and(menorque20);. El método and devuelve un nuevo predicado que resulta de la composición lógica de los dos. Por lo tanto, si hacemos dos predicados mayorque10, que evalúa si el número es mayor que 10, y menorque20, que evalúa si un número es menor que 20, La composición and da como resultado un nuevo predicado que evalúa si el número es mayor que diez e inferior a 20.

```
Predicate<Integer> and = mayorque10.and(menorque20);

            System.out.println("¿Es el siguiente número  " + numero + "
mayor que 10 "+
                               " y " + " menor que 20? " + and.test(numero));
```

La ejecución quedaría como sigue

```
¿Es el siguiente número  13 mayor que 10  y  menor que 20? true
```

Cornell notes

Comenta este código

```
Predicate<Integer> or = mayorque10.or(menorque20);

            System.out.println("¿Es el siguiente número " + numero + " mayor
que 10 "+
                               " 0 " + " menor que 20? " + and.test(numero));

      }
```

Ejercicio

Escribe un predicado que compruebe si un número es un número primo.

El algoritmo para los números primos es:

Leer (n)
Integer i=1
boolean numeroPrimo = true

While (i<=n/2)

If (n%i==0)
numeroPrimo =false

End While

1.5.3 El interface Consumer

El Consumidor <T> está definido de representar o ser la función de plantilla **con** un argumento de tipo T **(parámetros)** que **devuelve nulo,** nada. Esta interfaz **consumirá básicamente algunos datos de entrada** y realizará una acción, es decir, la interfaz representa o define una **operación que realiza una acción y no devuelve ningún resultado. La interfaz ofrece el método abstracto aceptar ser reemplazado por la expresión lambda o las clases anónimas.**

La declaración de la interfaz del consumidor tiene la siguiente forma:

Interface Consumer<T>

Ofrece el método de aceptación antes mencionado y algunos métodos predeterminados para la composición de funciones:

`void`	**`accept`**`(`**`T`**` t)` Realiza esta operación sobre el argumento dado.
`default` **`Consumer`**`<`**`T`**`>`	**`andThen`**`(`**`Consumer`**`<? super` **`T`**`> after)` Devuelve un `consumidor` compuesto que realiza, en secuencia, esta operación seguida de la operación posterior.

Esta interfaz es el modelo para lambdas que funciona como procedimientos.

Reciben parámetros, pero no devuelven ningún valor. Los programadores utilizan este tipo de interfaces funcionales para imprimir en la consola, la interfaz gráfica o para escribir en archivos o bases de datos.

Vamos a analizar el siguiente ejemplo:

```
´
Consumer<Integer> consumer = i -> System.out.println("Consumer 1 hace una
operación " + +i*i);
```

Este consumidor ejecutará lo que está en la lambda, el cuadrado de la variable i.

```
System.out.println("Ejecutamos el primer Consumer");
      consumer.accept(7);
```

Resultado de la ejecución:

```
Ejecutamos el primer Consumer
Consumer 1 hace una operación 49
```

Construimos el segundo consumidor a partir del primer consumidor y el método **andThen()**. Este nuevo consumidor tiene ahora dos operaciones. Cuando llamamos al método accept de consumerWithAndThen,primero se ejecutará el lambda consumidor. Después de eso, ejecutará la segunda lambda, la nueva que hemos agregado. Como se ilustra en la ejecución a continuación, la nueva marca consumerWithAndThen está compuesta por los consumidores. El original más la expresión que hemos añadido como parámetro para el método **andThen**. Una vez más, estamos componiendo interfaces, dos funciones.

```
Consumer<Integer> consumerWithAndThen =
                  consumer
                  .andThen(i -> System.out.println("Consumer With and Then:
Explica la operación anterior, el cuadrado de :" + i ));
```

Ejecución:

```
Ejecutamos el consumer compuesto que ejecutará los dos uno detras de otro
Consumer 1 hace una operación 36
Consumer With and Then: Explica la operación anterior, el cuadrado de :6
```

Debemos tener en cuenta que la segunda lambda no se ejecutará hasta que la primera haya cumplido con su deber. En programación, conocemos este comportamiento como una devolución de llamada, una función que se ejecuta tan pronto como la función con la que está relacionada ha terminado sus

ejecuciones.

Finalmente, para demostrar que podemos crear clases anónimas a partir de Interfaces Funcionales, tenemos este fragmento de código. Aquí, estamos anulados el Método del Consumidor aceptar.

```
Consumer<Integer> consumer3 = new Consumer<Integer> () {

		@Override
		public void accept(Integer t) {
			// TODO Auto-generated method stub

			System.out.println("Consumer anónimo");

		}



	};
```

ConsumerInterface.java

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

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

	Consumer<Integer> consumer = i -> System.out.println("Consumer 1 hace
una operación " + +i*i);
	Consumer<Integer> consumerWithAndThen =
			consumer
			.andThen(i -> System.out.println("Consumer With and Then:
Explica la operación anterior, el cuadrado de :" + i ));

	System.out.println("Ejecutamos el primer Consumer");
	consumer.accept(7);
	System.out.println("Ejecutamos el consumer compuesto que ejecutará los
dos uno detras de otro");
	consumerWithAndThen.accept(6);



	Consumer<Integer> consumer3 = new Consumer<Integer> () {

		@Override
		public void accept(Integer t) {
			// TODO Auto-generated method stub

			System.out.println("Consumer anónimo");

		}



	};
```

```
        }
}
```

Ejercicio

A. Crear un consumer que implemente un menú que ofrezca estas opciones:

1. Convertir de Farehheit a Celsius
2. Convertir de Celsius a Fahrenheit
3. Convertir de libras a kilogramos
4. Convertir de Kilogramas to libras
5. Convertir de Kw to caballos
6. Convertir de caballos a Kw

Cada opción leerá un valor de la consola y lo convertirá en la medida de destino. Los resultados se mostrarán en la consola.

B. Implementando un bucle anidado, crear un consumer que muestre la lista de los n primeros números primos, n pasado como parámetro.

1.5.4 Interface Supplier

La interfaz Supplier realiza la función opuesta a la de los consumidores. Permite anular con una lambda que no recibe ningún parámetro y devuelve un valor. La definición de proveedor en Java es:

```
@FunctionalInterface
public interface Supplier<T>
```

El método a anular es obtener en el estado en la siguiente tabla:

Modificador y Tipo	Método y descripción
T	get() Obtiene un resultado.

Presta atención al método get. Devuelve un tipo genérico T y no declara ningún argumento. Java no ofrece composición de interfaces funcionales para Supplier. El siguiente ejemplo aborda la descripción del proveedor. Es una interfaz funcional notablemente simple. Los proveedores tienen como objetivo ofrecer valores a los programadores, siguiendo diferentes esquemas. Estos valores pueden ser números en una serie, pueden ser datos de un archivo o una base de datos.

Hemos creado dos objetos de proveedores en el ejemplo.

El primero lee un nombre de la consola y devuelve este nombre:

```
Supplier<String> supNombre = ()->{

                        i=5;
                        System.out.println("¿Puedes introducir tu nombre,
por favor?");
                        return sc.nextLine();


            };
```

Observa cómo el lado izquierdo de la lambda no tiene ningún parámetro en su definición`()->`

El segundo proveedor devuelve un número aleatorio del 0 al veinte. Utilizamos la clase Random de java.util para generar el número aleatorio. El método que llamamos es nextInt(20) que genera un int aleatorio desde el rango de 0 y 20.

Si revisamos el ejemplo, hemos creado el objeto pero no lo hemos asignado a una variable. En su lugar, estamos llamando al método del objeto directamente. Es una práctica común en Java funcional.

```
new Random().nextInt(20);
```

Este programa ilustra una característica bastante importante de las lambdas. Repasemos la variable Scanner sc. Siempre que desee hacer uso de la variable dentro de la lambda, sc debe declararse final. Esto resulta de la especificación

lambda, y lambdas es un objeto, dado que a los lambdas no se les permite modificar variables que no están en su ámbito. Como puede ver, sc está fuera del bloque lambda, fuera de su alcance.

El compilador no dejará usar la variable sc dentro de la lambda a menos que se defina utilizando el modificador final, lo que la hace inmutable, lo que significa que no se puede modificar. El propósito de esta característica es evitar efectos secundarios e introducir inmutabilidad en nuestras lambdas. Estas dos características son características deseadas de la programación funcional, que presentaremos en el próximo capítulo. Por el contrario, puede modificar las propiedades del objeto contenedor lambda siguiendo la especificación orientada a objetos de Java. Resaltado en amarillo, la propiedad que se puede modificar. En azul la variable local que debe declararse final para ser utilizada en el bloque lambda.

Por otro lado estamos usando la propiedad i como estática. Es la única manera de poder usarla en el cuerpo de la lambda y poder modificarla. Con el modificador final no la podríamos modificar.

```
private static  int i=0;


Supplier<String> supNombre = ()->{

                        i=5;
                        System.out.println("¿Puedes introducir tu nombre,
por favor?");
                        return sc.nextLine();


            };
```

SupplierInterface.java

```
package introduccionprogramacionfuncional;
import java.util.Random;
import java.util.Scanner;
import java.util.function.Supplier;

public class SupplierInterface {

private static  int i=0;
      public static void main(String[] args) {

            final Scanner sc = new Scanner(System.in);

            Supplier<String> supNombre = ()->{

                        i=5;
```

```
                        System.out.println("¿Puedes introducir tu nombre,
por favor?");
                        return sc.nextLine();


            };

            System.out.println("Nombre leido por la consola: "
+supNombre.get());

            Supplier<Integer> supRandom = () -> new Random().nextInt(20);

                  System.out.println("Genera un número aleatorio: "
+supRandom.get());
      }

}
```

Ejercicio

C. Según el ejemplo, puede crear un proveedor que devuelva un valor aleatorio entero de 0 a 100. Utilice el proveedor en un bucle para mostrar 50 números aleatorios.

   ¿Podrías hacerlo con el tipo Double?

Nota: las siguientes interfaces son similares a las funciones debido al hecho de que reciben parámetros y devuelven resultados. Estudiaremos tres de ellos aunque Java ofrece más de ellos: UnaryOperator, Function, BiFunction.

1.5.5 La interface funcional Function

Este propósito predefinido de la interfaz de función es definir una plantilla para lambdas que reciben un solo parámetro y devuelven un valor. Junto con el

BiFunction y el UnaryOperator muestran un gran uso ya que la API de Java sigue este patrón para muchas de sus clases, y los programadores también codifican este tipo de lambdas.

La declaración Java para la interfaz de función tiene esta estructura:

Interface Function<T,R>

La interfaz proporciona el siguiente método, teniendo en cuenta que el método que sobreescribe la lambda es apply.

default <V> Function<T, V>	andThen(Function<? super R,? extends V> after)	Devuelve una función compuesta que primero aplica esta función a su entrada y, a continuación, aplica la función after al resultado.
R	apply(T t)	Aplica esta función al argumento dado.
default <V> Function<V, R>	compose(Function<? super V,? extends T> before)	Devuelve una función compuesta que primero aplica la función before a su entrada y, a continuación, aplica esta función al resultado.
static <T> Function<T, T>	identity()	Devuelve la función identidad f(x)=x

El método que debemos sobreescribir con la expresión lambda es R apply (T t). Si miramos más de cerca la declaración de interfaz, define dos tipos genéricos <T, R>. T es el parámetro de entrada Type. R es el valor devuelto. En este caso, para declarar una variable Function Typed es necesario introducir a los tipos incluidos en el operador diamante, es decir, Function<Integer, Double>; En este escenario, el método apply recibiría un parámetro Integer y devolvería un valor Double.

Demos un ejemplo.

```
Function<Integer,Double> raizCuadrada = (n)-> Math.sqrt(n);
```

Esta variable `raizCuadrada` almacena una lambda que sobreescribe el método Function apply. Siguiendo la declaración de interfaz Function<Integer,Double>, podemos decir que la lambda recibe un Integer y devuelve un Double. Además, el código lambda devuelve la raíz cuadrada del número pasado.

To ejecutar este código lambda que llamamos el método apply pasando un entero, 25.

```
System.out.println( " El resultado de la raiz cuadrada es " +
raizCuadrada.apply(25));
```

The result is a Double, 5.0

```
El resultado de la raiz cuadrada es 5.0
```

Esta es la versión más simple ya que hemos hecho uso del método apply. Hay otros dos métodos predeterminados interesantes para la interfaz de función, andThen, y compose. Están destinados a llevar a cabo la composición de funciones para interfaces de funciones.

La variable function1 recibe un String y devuelve la versión en mayúsculas de esta cadena.

```
Function <String,String> function1 = (s-> s.toUpperCase() + "  Function 1 to
uppercase.");

            System.out.println( " valor de function1  " +
function1.apply(stringEjemplo));
```

En la ejecución:

```
valor de function1   MI STRING COMO PARAMETRO   Function 1 pasa a mayusculas.
```

La variable function2 recibe un String y devuelve la misma cadena, pero los espacios en blanco exteriores se han borrado.

```
            Function <String,String> function2 = (s->s.trim() + " Function 2
elimina blancos.");
            System.out.println( " valor de function2  " +
function2.apply(stringEjemplo));
```

En la ejecución:

```
valor de function2  Mi String como parametro Function 2 elimina blancos.
```

La función de interfaz3 combina las dos interfaces anteriores. Ya hemos introducido el método andThen para otras interfaces. Se sabe que el andThen funciona como una devolución de llamada. Una vez que la función1 devuelve un resultado, este resultado va como parámetro a la función2. Hasta que la primera función no termina, la segunda no se ejecuta. Además, el resultado, el resultado de la función1 es la entrada de la función2. Este es el resultado que la function3.apply(ejemploString) devolverá**, en este orden.**

```
Function <String,String> function3 = function1.andThen(function2);
            System.out.println(      "      function3      value      "      +
function3.apply(stringExample));
```

En la salida del ejemplo puede ver cómo se ha evaluado primero la Función 1.

```
valor de function3  MI STRING COMO PARAMETRO   Function 1 pasa a mayusculas.
Function 2 elimina blancos.
```

Y por último, el método de composición. Ya hemos descrito la composición de la función. En la composición de dos funciones matemáticas, FoG(x), la primera en ser evaluada es G(x), la función más interna. La salida de G(x) será la entrada de F(x) la función externa. El método compose funciona siguiendo este patrón. Como resultado, en el siguiente ejemplo, function2 se ejecutará primero. El resultado de function2, será la entrada de function1. Este es el resultado que la functionn4.apply(stringEjemplo) **devolverá, en este orden.**

```
      Function <String,String> function4= function1.compose(function2);
            System.out.println( " valor de function4 " +
function4.apply(stringEjemplo) );
```

En la salida de ejemplo puede ver cómo se ha evaluado primero la Función 2.

```
valor de function4 MI STRING COMO PARAMETRO FUNCTION 2 ELIMINA BLANCOS.
Function 1 pasa a mayusculas.
```

FunctionInterface.java

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

public class FunctionInterface {


      public static void main(String[] args) {

            String stringEjemplo=" Mi String como parametro ";

            Scanner sc = new Scanner(System.in);


            Function<Integer,Double> raizCuadrada = (n)-> Math.sqrt(n);

            System.out.println( " El resultado de la raiz cuadrada es " +
raizCuadrada.apply(25));
```

```
            Function <String,String> function1 = (s-> s.toUpperCase() + "
Function 1 pasa a mayusculas.");

            System.out.println( " valor de function1  " +
function1.apply(stringEjemplo));

            Function <String,String> function2 = (s->s.trim() + " Function 2
elimina blancos.");
            System.out.println( " valor de function2  " +
function2.apply(stringEjemplo));

            Function <String,String> function3 =
function1.andThen(function2);
            System.out.println( " valor de function3  " +
function3.apply(stringEjemplo));

                  Function <String,String> function4=
function1.compose(function2);
            System.out.println( " valor de function4 " +
function4.apply(stringEjemplo) + "function 1 pasa a mayusculas");



      }

}
```

1.5.6 The UnaryOperator Interface

El operador unario es casi idéntico a la interfaz de función. La única diferencia destacable es que devuelve el mismo Tipo que recibe como parámetro dado que se extiende desde la interfaz Función. En consecuencia, no es necesario definir el Tipo dos veces.

La estructura del operador unario es:

```
public interface UnaryOperator<T>
extends Function<T,T>
```

Y los métodos ofrecidos por esta interfaz son los mismos que la interfaz de función proporciona, apply, andThen, compose e identity, . Se podría afirmar que es una especialización de la primera.

En el siguiente ejemplo, se define el interfaz factorial como Long, donde Long es el Tipo recibido como parámetro y también el tipo devuelto.

```
UnaryOperator<Long> factorial
```

UnaryOperatorExample.java

```
package PredefinedInterfaces;

import java.util.function.UnaryOperator;

package introduccionprogramacionfuncional;

import java.util.function.UnaryOperator;

public class UnaryOperatorEjemplo {


public static void main(String[] args) {

            UnaryOperator<Long> factorial = (n) -> {

                  Long res= 1L;
                  for (int i = 1; i<=n ;i++) {


                        res= res*i;
                  }

                  return res;
            };




            System.out.println("The factorial de 5 es: " +
factorial.apply(5L));

}

}
```

Activitidad

Agregue al ejemplo anterior un operador unario Long potenciaDeDos que calcula la potencia del número. Después de eso, combínalo con el factorial para obtener un nuevo UnaryOperator:

a. Usando el método andThen

b. Con el método compose

Mostrar los diferentes resultados en la consola.

1.5.7 Interfaz funcional Bifunction

El interface Bifunction es prácticamente igual que el interfaz Function sólo que recibe dos tipos genéricos como parámetros de entrada en vez de uno, y un tercer parámetro R que indica el tipo de salida.

La estructura del interface Bifunction es:

```
@FunctionalInterface
public interface BiFunction<T,U,R>
```

T y U son los tipos de los parámetros de entrada. R es el valor retornado

El método sobreescrito por la expression lambda es R apply(T t. U u). En total, esta interfaz ofrece estos dos métodos.: andThen y apply.

default <V> BiFunction<T, U,V>	andThen (Function<? super R,? extends V> after)	Returns a composed function that first applies this function to its input, and then applies the after function to the result.
R	apply (T t, U u)	Applies this function to the given arguments.

Un ejemplo de Interface Bifunction

El siguiente interfaz recibe dos enteros como parámetro y devuelve un Doble.

```
BiFunction<Integer, Integer, Double> floatDiv
```

```
package PredefinedInterfaces;

package introduccionprogramacionfuncional;
import java.util.function.BiFunction;

public class BifunctionInterfaceEjemplo {
```

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

		BiFunction<Integer, Integer, Double> floatDiv = (a,b) ->
(double) a/b;



		BiFunction <String, String, String > concatMayusculas = (s1,s2)-
> (s1 +s2).toUpperCase();


		System.out.println ("Division decimal de 5 y 7:" +
floatDiv.apply(5, 7));


		System.out.println ("concatMayusculas interface:" +
concatMayusculas.apply("String 1", "String 2"));




	}

}
```

1.5.8 Interfaces funcionales para tipos primitivos

Veamos los diferentes tipos de especialización de interfaces funcionales predefinidas que se ven en el tema anterior. Para casi todos los tipos primitivos en Java tenemos su especialización en Interfaz Funcional. Si presta atención a los ejemplos anteriores de interfaces funcionales, todos ellos reciben tipos genéricos. Los tipos genéricos no pueden ser tipos primitivos. Para solucionar este problema, Java ofrece interfaces funcionales que devuelven o reciben tipos primitivos, evitando la necesidad de pasarlos como tipos genéricos, lo que no está permitido.

Dado que un tipo primitivo no puede ser un argumento de **tipo genérico**, existen versiones de la interfaz Function para los tipos primitivos más utilizados **double, int, long y sus** combinaciones en tipos de argumento y tipo devuelto:

1. ***IntFunction***, ***LongFunction***, ***DoubleFunction***- ***Los argumentos*** **son del tipo especificado,**el tipo devuelto está parametrizado.
1. ***ToIntFunction***, ***ToLongFunction***, ***ToDoubleFunction:*** Los tipos de retorno son de tipo **especificado,** los argumentos están parametrizados.
1. ***DoubleToIntFunction***, ***DoubleToLongFunction***, ***IntToDoubleFunction***, ***IntToLongFunction***, ***LongToIntFunction***, ***LongToDoubleFunction*** - Tienen un **argumento** y un tipo de retorno definidos como tipos primitivos, según lo especificado por sus nombres.

En el siguiente ejemplo utilizamos algunas de estas especializaciones.

La interfaz IntFunction, recibe un tipo primitivo int y devuelve un tipo Genérico, String en este caso.

```
IntFunction<String> convertdeIntaString = (i)-> String.valueOf(i);
```

La interfaz toIntFunction, recibe un Generic Type, String, y devuelve un tipo primitivo int.

```
ToIntFunction<String> convertdeStringaInt = (s -> Integer.valueOf(s));
```

Observe cómo siempre definimos el tipo Genérico en el diamante operador. El tipo primitivo se infiere del nombre de la interfaz funcional.

En el último ejemplo no hay ningún tipo genérico pasado como parámetro. Tanto int como doble son tipos primitivos.

```
IntToDoubleFunction convertdeIntaDouble = (i -> Double.valueOf(i));
```

Con todo, Java incluye estas especializaciones para permitir al programador utilizar tipos primitivos en sus expresiones lambda.

PrimitiveTypeSpecialization.java

```
package PredefinedInterfaces;

package introduccionprogramacionfuncional;
import java.util.Scanner;
import java.util.function.IntFunction;
import java.util.function.IntToDoubleFunction;
import java.util.function.ToIntFunction;

public class PrimitiveTypeSpecialization  {
```

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


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

		IntFunction<String> convertdeIntaString = (i)->
String.valueOf(i);

		ToIntFunction<String> convertdeStringaInt = (s ->
Integer.valueOf(s));

		IntToDoubleFunction convertdeIntaDouble = (i ->
Double.valueOf(i));

		String cadena = convertdeIntaString.apply(numero1);

		System.out.println("Convertir de tipo primitivo int  " + numero1
+ " a String " + cadena);

		int numero2 = convertdeStringaInt.applyAsInt(cadena);

			System.out.println("Convertir de String  " + cadena + " a
tipo primitivo in " + numero2);

		Double numerodecimal =
convertdeIntaDouble.applyAsDouble(numero1);


		System.out.println("Convertir de primitivo int " + numero1 + " a
primitivo double " + numerodecimal);

	}

}
```

2 Programación funcional

2.1 Categorización de paradigmas de programación

Por supuesto, la programación funcional no es el único estilo de programación en la práctica. En términos generales, los estilos de programación se pueden clasificar en paradigmas de programación imperativos y declarativos:

El **enfoque imperativo** define un programa como una secuencia de declaraciones que cambian el estado del programa hasta que alcanza el estado final. La programación procedimental es un tipo de programación imperativa en la que construimos programas utilizando procedimientos o subrutinas. Uno de los paradigmas de programación populares conocidos como programación orientada a objetos (POO) extiende los conceptos de programación procedimental.

En contraste, el **enfoque declarativo** expresa la lógica de un cálculo sin describir su flujo de control en términos de una secuencia de declaraciones. En pocas palabras, el enfoque del enfoque declarativo es definir lo que el programa tiene que lograr en lugar de como debería lograrlo. La programación funcional es un subconjunto de los lenguajes de programación declarativos.

2.2 Programación funcional

La programación funcional es un paradigma de programación en el que tratamos de unir todo en el estilo de funciones matemáticas puras. Es un tipo declarativo de estilo de programación. Su enfoque principal está en "qué resolver" en contraste con un estilo imperativo donde el enfoque principal es "cómo resolver". Utiliza expresiones en lugar de declaraciones. Una expresión se evalúa para producir un valor, mientras que una instrucción se ejecuta para asignar variables. Esas funciones tienen algunas características especiales que se analizan a continuación.

La programación funcional se basa en el cálculo lambda:

El cálculo lambda es un marco desarrollado por Alonzo Church para estudiar cálculos con funciones. Se puede llamar como el lenguaje de programación más pequeño del mundo. Da la definición de lo que es computable. Cualquier cosa que pueda ser calculada por el cálculo lambda es computable. Es equivalente a la máquina de Turing en su capacidad de computar. Proporciona un marco teórico para describir las funciones y su evaluación. Forma la base de casi todos los lenguajes de programación funcionales actuales.

Nota: Alan Turing fue un estudiante de la Alonzo Church y creó la máquina Turing que sentó las bases del estilo de programación imperativo.

Lenguajes de programación que soportan programación funcional: Haskell, JavaScript, Scala, Erlang, Lisp, ML, Clojure, OCaml, Common Lisp, Racket.

Los lenguajes de programación funcionales se clasifican en dos grupos, es decir,

- Lenguajes funcionales puros: estos tipos de lenguajes funcionales solo admiten los paradigmas funcionales. Por ejemplo − Haskell.

- Lenguajes funcionales impuros: estos tipos de lenguajes funcionales admiten los paradigmas funcionales y la programación de estilo imperativo. Por ejemplo − LISP.

2.3 Programación Funcional. Ventajas

- Código libre de errores − La programación funcional no admite el estado, por lo que no hay resultados de efectos secundarios y podemos escribir códigos sin errores.

- Programación paralela eficiente: los lenguajes de programación funcionales NO tienen estado mutable, por lo que no hay problemas de cambio de estado. Se puede programar "Funciones" para que funcionen en paralelo como "instrucciones". Tales códigos admiten una fácil reutilización y estabilidad.

- Eficiencia: los programas funcionales consisten en unidades independientes que pueden ejecutarse simultáneamente. Como resultado, tales programas son más eficientes.

- Soporta funciones anidadas: la programación funcional admite funciones anidadas.

- Evaluación Lazy − La programación funcional admite construcciones funcionales perezosas como listas perezosas, mapas perezosos, etc. La evaluación Lazy o por necesidad es una estrategia de evaluación que retrasa el cálculo de una expresión hasta que su valor sea necesario, y que también evita repetir la evaluación en caso de ser necesaria en posteriores ocasiones. Esta compartición del cálculo puede reducir el tiempo de ejecución de ciertas funciones de forma exponencial, comparado con otros tipos de evaluación.

Basándose en la idea de la programación declarativa, la programación funcional busca resolver problemas de algoritmos utilizando llamadas a funciones. El siguiente ejemplo tratará de ilustrar las diferencias sobre la programación imperativa y declarativa:

Una vez que hemos leído el número al que queremos calcular la tercera potencia, las soluciones imperativas se basan en la declaración de variables y la asignación para resolver el problema.

```
		//solución imperativa
		double result = numero*numero*numero;
		System.out.println("El cubo de "+ numero + " es " + result);
```

Por otro lado, la programación funcional trata de evitar la declaración y asignación de variables, que es la principal causa de bugs y errores en nuestros programas. Como se muestra en el ejemplo, en una línea resolvemos el problema con llamadas a funciones o métodos.

```
		//Solución funcional

System.out.println("El cubo de "+ numero + " es " +
ProgramacionFuncionalEjemplo.potenciaCubo(numero));
```

Al mismo tiempo, los métodos en programación funcional intentan omitir la asignación de valores a la variable tanto como sea posible.

```
public static double potenciaCubo(double x) {

		return x*x*x;
	}
```

ProgramacionFuncionalEjemplo.java

```
package paradigmaprogramacionfuncional;
import java.util.Scanner;

public class ProgramacionFuncionalEjemplo {

	public static double potenciaCubo(double x) {

		return x*x*x;
	}

	public static void main(String[] args) {

		Scanner sc = new Scanner(System.in);


		System.out.println("Escribir un número para calcular el cubo");
```

```
            double numero = sc.nextInt();


            //Imperative solution
            double result = numero*numero*numero;
            System.out.println("El cubo de "+ numero + " es " + result);



            //Declarative and functional solution
System.out.println("El cubo de "+ numero + " es " +
ProgramacionFuncionalEjemplo.potenciaCubo(numero));

      }

}
```

2.4 Programación funcional Java 8

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 Java repasaremos **los conceptos básicos de la programación funcional, y** qué partes de **ella son factibles en Java.**

2.4.1 Conceptos básicos de la programación funcional

La programación funcional incluye los siguientes conceptos clave:

1. Las funciones son 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. Sin estado
2. Sin efectos secundarios.
3. Variables inmutables
4. Favorecer la recursión 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, podemos beneficiarnos 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 programación funcional** son para uso diario de **los programadores de Java.**

2.5 Las funciones son objetos de primera clase / miembros de primera clase del lenguaje

Una de las principales ventajas que Java 8 y la programación funcional aporta a la mesa es que las funciones se han convertido en objetos de primera clase o miembros del lenguaje a través de las interfaces funcionales. Algo que es bastante común en algunos de los lenguajes de programación como JavaScript, que es que las funciones se pueden almacenar en variables o se pueden pasar como parámetros, era imposible en versiones anteriores de Java.

En este extracto de Javascript estamos pasando la función myCallback como parámetro a la función addContact.

```
function addContact(id, callback) {
   callback();
}

function  myCallback() {
   alert('Hello World');
}


addContact (myCallback);
```

En el paradigma de programación funcional, las funciones son **objetos de primera clase en el lenguaje.** Esto significa que puede crear una **"instancia" de** una función, al igual que una referencia variable a esa instancia de función, así como 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 **llega Java son las** interfaces funcionales.

Vamos a explicarlo con un ejemplo sencillo. En nuestra clase

FunctionPrimeraClaseEjemplo, **tenemos una** formula de parámetro de interfaz funcional que aplica la **fórmula recibida, llamando al método apply. Observe que para ejecutar la función necesitamos llamar al método apply,** formula.apply(x); .

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


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

También tenemos dos funciones formulaCuadrado y formulaCubo. Estas dos funciones de firma Doble x, y tipo devuelto Doble, coinciden con la firma de la Interfaz Funcional `Function<Double,Double> formula`.

```
public Double formulaCuadrado(Double x) {
        return x*x;
    }
    public Double formulaCubo(Double x) {
        return x*x*x;
    }
```

Podemos crear un objeto de tipo `FuncionPrimeraClaseEjemplo`.

```
FuncionPrimeraClaseEjemplo objeto = new FuncionPrimeraClaseEjemplo();
```

Después de eso, podemos pasar al método functionFormula una función de este objeto u otro objeto de una clase diferente, object::formulaCuadrado. Para ello, utilizamos el operador :: , que permite a los programadores hacer

referencia a las funciones.

En este caso, hacemos referencia a la función a través del objeto, y el sintax para esta expresión es:

object_variable_name::nombreMetodo

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

Por lo tanto, los interfaces funcionales proporcionan la oportunidad de pasar funciones como parámetro y también almacenarlas en una variable.

```
Function<Double,Double> formulaVar = objeto::formulaCuadrado;
```

Además, podemos pasar como parámetro una expresión lambda, lo que finalmente es una función anónima.

```
result = object.functionFormula(numero, x->x*x*x*x);
```

Finalmente, podemos pasar como parámetro un método estático, un método de clase. La sintaxis para los métodos estáticos es ligeramente diferente ya que usamos el nombre de la clase para hacer referencia a un método estático:

nombreClase::nombreMetodo

```
FunctionFirstClassExample:: formulaRaizCuadrada
```

```
result = objeto.funcionFormula(numero,
FuncionPrimeraClaseEjemplo::formulaRaizCuadrada);
```

FunctionFirstClassExample.java

```
package paradigmaprogramacionfuncional;

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


public class FuncionPrimeraClaseEjemplo{


	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 Double formulaRaizCuadrada(Double x) {
            return Math.sqrt(x);


        }
        public static void main(String[] args) {
            Scanner sc = new Scanner(System.in);
            System.out.println("Introduzca un número en la consola");
            Double numero = sc.nextDouble();
            FuncionPrimeraClaseEjemplo objeto = new
FuncionPrimeraClaseEjemplo();


            Double result = objeto.funcionFormula(numero,
objeto::formulaCuadrado);
            System.out.println("Calculamos el cuadrado del número  " +
result);
            Function<Double,Double> formulaVar = objeto::formulaCuadrado;
            result = objeto.funcionFormula(numero, objeto::formulaCubo);


            System.out.println("Calculamos el cubo del número: " + result);
            result = objeto.funcionFormula(numero, x->x*x*x*x);
            System.out.println("Calculamos la cuarta potencia del número: "
+ result);
            result = objeto.funcionFormula(numero,
FuncionPrimeraClaseEjemplo::formulaRaizCuadrada);
```

```
        System.out.println("Calculamos la raiz cuadrada del número: " +
result);

        }

}
```

Un ejemplo de ejecución:

```
25
Calculamos el cuadrado del número  625.0
Calculamos el cubo del número: 15625.0
Calculamos la cuarta potencia del número: 390625.0
Calculamos la raiz cuadrada del número: 5.0
```

2.6 Funciones puras, sin estado.

En programación, una función pura es una función que tiene las siguientes propiedades:

1. La función siempre devuelve el mismo valor para las mismas entradas.
2. La evaluación de la función no tiene efectos secundarios. Los efectos secundarios se refieren al cambio de otros atributos del programa no contenidos en la función, como el cambio de valores de variables globales o el uso de flujos de E/S (sin mutación de variables estáticas locales, variables no locales, argumentos de referencia mutables o flujos de entrada/salida).

Efectivamente, el valor devuelto de una función pura se basa solo en sus entradas y no tiene otras dependencias o efectos en el programa general.

Las funciones puras son conceptualmente similares a las funciones matemáticas. Para cualquier entrada dada, una función pura debe devolver exactamente un valor posible.

Sin embargo, al igual que una función matemática, se le permite devolver ese mismo valor para otras entradas. Además, al igual que una función matemática, su salida está determinada únicamente por sus entradas y no por ningún valor almacenado en algún otro estado global.

Un ejemplo:

```
public class PuraFuncionClass{

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

Cada vez que llame a la función de suma devolverá el mismo valor, es decir, la evaluación de suma (5,7) será siempre 12, sin importar cuántas veces llame a esta función.

Observe cómo el valor devuelto de la función sum() **depende únicamente de los parámetros de entrada.** También tenga en cuenta que **el método sum() no tiene** efectos secundarios, lo que significa **que no modifica ningún estado (variables) fuera de la función** o clase.

Lo contrario se puede encontrar en el siguiente ejemplo **de una función no pura:**

```
public class NoPuraFuncionExample{
   private int value = 1;

   public int sum(int nextValue) {
      this.value += nextValue;
      return this.value;
   }
```

Aquí, el efecto es el contrario. La primera vez que llame a esta función sum(3) devolvería 4. Siempre que después de esta llamada el valor de la propiedad cambie a 4, la segunda vez que llame a sum(3), el resultado devuelto de esta llamada de método sería 7. Al final, el valor devuelto de la función depende del valor de la propiedad del objeto, que cambia en cada llamada.

2.6.1 Función de orden superior

La idea detrás de las funciones de alto orden representa el empleo de funciones como argumentos. Para que una función sea una función de alto orden, debe cumplir la siguiente condición: Cualquier parámetro en la firma de la función debe ser una función, y el tipo de retorno debe ser una función del mismo modo. Ergo, el producto final es una función que manipula funciones y genera una

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 una **interfaz funcional.** Se describe en el siguiente **ejemplo de una función de orden superior en Java. Es más fácil de lo que parece y se lleva mucho tiempo usando en otros lenguajes como JavaScript o Python.**

public Supplier<Double> funcionOrdenSuperior(Supplier<Double> numero, Function<Double,Double> funcion)

El método funcionOrdenSuperior toma como parámetro una **interface Consumer** que nos da un **número, una interfaz de función** y devuelve una **interfaz Supplier.** Lo que estamos haciendo básicamente es permitir que las expresiones o funciones lambda **se pasen como parámetro** y devuelvan una función o expresión lambda como **parámetro.**

```
public Supplier<Double> funcionOrdenSuperior(Supplier<Double> number,
Function<Double,Double> function) {

                return ( ()->function.apply(number.get()));

        }
```

Los parámetros de la firma funcionOrdenSuperior son interfaces funcionales, lambdas o funciones. Asimismo, el tipo devuelto es un Supplier<Double> otra función. Se muestra en la instrucción return de este ejemplo, cómo estamos devolviendo una función, que es una lambda en este caso.

```
package introductionfunctionalprogrammin;

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

```
public class FuncionOrdenSuperiorEjemplo {


            public Supplier<Double> funcionOrdenSuperior(Supplier<Double>
number, Function<Double,Double> function) {


                  return ( ()->function.apply(number.get()));

            }


              public static void main(String[] args) {


                    Scanner miScanner = new Scanner(System.in);

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

                    Double numero1 = miScanner.nextDouble();

                    FuncionOrdenSuperiorEjemplo highOrder = new
FuncionOrdenSuperiorEjemplo() ;


                    Supplier<Double> supplierNumerico=
highOrder.funcionOrdenSuperior(()-> numero1 ,(x)->x*x);

                    System.out.println(" Función de orden superior que
retorna un supplier " + supplierNumerico.get());


                    System.out.println(" Función de orden superior que
retorna un supplier directamente"
                    +  highOrder.funcionOrdenSuperior(()-> numero1 ,(x)-
>x*x).get());

              }

      }
```

Debido a que este es un concepto absolutamente importante en la programación funcional, vamos a introducir otro ejemplo. Además, emplearemos algunos de los conceptos que hemos aprendido, como la composición de funciones y el operador ::.

El método componeFormulaSeleccionadaYRedondea recibe un argumento Function<Double,Double>. También devuelve una Función<Double,Doble>

```
public Function<Double,Double>
componeFormulaSeleccionadayRedondea(Function<Double,Double> function) {
```

```
            return function.andThen(FuncionOrdenSuperiorEjemplo2::redondea);

      }
```

El cálculo que realiza consiste en combinar la fórmula pasada como parámetro, una función, con la función redondear, mostrada en amarillo resaltado. Finalmente, llamamos a esta función en el programa principal.

```
Function<Double,Double> miFormula =
objectHighOrder.componeFormulaSeleccionadayRedondea(FuncionOrdenSuperiorEjemplo2::cuadrado);
```

La función cuadrada es un método estático que se adapta a la definición función<doble, doble>; Por lo tanto, podemos pasarlo como un parámetro

```
public static Double cuadrado(Double a) {


            return a*a;
      }
```

Como resultado, tenemos una función que calcula el cuadrado y redondea el resultado en la variable de tipo interfaz funcional Function<Double,Double> miFormula.

Podemos emplear esta función en cualquier lugar de nuestro código, e infinitas veces. Es una función pura también; devolverá el mismo valor en cualquier llamada. Para invocar la función, utilizamos el método apply myFormula.apply(24.2).

```
System.out.println("La función obtenida de componer finalmente calcula el
cuadrado" +
              " y redondea el resultado del cuadrado: " +
miFormula.apply(24.2));


       }
```

El resultado de la ejecución es

```
función obtenida de componer finalmente calcula el cuadrado y redondea el
resultado del cuadrado: 586.0
```

FuncionOrdenSuperiorEjemplo2.java

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

public class FuncionOrdenSuperiorEjemplo2 {


	public Function<Double,Double>
componeFormulaSeleccionadayRedondea(Function<Double,Double> function) {

		return function.andThen(FuncionOrdenSuperiorEjemplo2::redondea);


	}


	public static Double  raizCuadrada(Double a) {


		return Math.sqrt(a);
	}

public static Double cuadrado(Double a) {


		return a*a;
	}


public static double redondea(Double a) {


		return (double) Math.round(a);


}



	 public static void main(String[] args) {

		FuncionOrdenSuperiorEjemplo2 objectHighOrder = new
FuncionOrdenSuperiorEjemplo2();


		Function<Double,Double> miFormula =

objectHighOrder.componeFormulaSeleccionadayRedondea(FuncionOrdenSuperiorEjemp
lo2::cuadrado);


		System.out.println("La función obtenida de componer finalmente
calcula el cuadrado" +
		" y redondea el resultado del cuadrado: " +
miFormula.apply(24.2));
```

```
        }
}
```

Ejercicio

Añadir al ejemplo anterior miFormula2 de manera que el método resultado realice la raíz cuadrada y luego redondee, usando el método raizCuadrada.

```
Function<Double,Double> miFormula2
```

Añadir al ejemplo anterior miFormula3 de manera que el método resultado realice el cubo y luego redondee, pasando una función lambda que calcule el cubo.

```
Function<Double,Double> miFormula3
```

2.6.2 Sin estado

Como **se mencionó al principio de estos capítulos,** una regla del paradigma de programación funcional es no tener estado**.** "Ningún estado" o "apátrida" generalmente es entendido por cualquier estado fuera de la función. Una función **puede tener variables locales** que contienen el estado temporal **internamente,** pero la función no debe hacer referencia a **ninguna variable miembro de la clase u objeto** al que pertenece la función. No debe **tener** acceso a **valores externos.**

Ejemplo de una función que no hace uso de ningún estado externo a ella.

```
public class Calculator{
   public int sum(int a, int b) {
      return a + b;
   }
}
```

Por el contrario, en la **siguiente versión** del **método sum** el **problema** es que el método depende del estado de **la calculadora** en un momento determinado, **ya que su cálculo radicaba en el valor initVal.**

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

public setInitVal(int val) {
```

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

Este tipo de prácticas de programación pueden conducir a errores y errores en sucódigo. Sin embargo, dado que Java es un lenguaje de programación orientado a objetos, tenemos que confiar en el estado de un objeto para hacer algunos cálculos. El camino a seguir debe ser tratar de encontrar un equilibrio o equilibrio entre la programación funcional y la orientada a objetos. Por lo tanto, en algún contexto, cuando es necesario, aplicamos conceptos del Diseño Orientado a Objetos. Debería ser obligatorio en cualquier escenario de programación donde necesitemos representar datos o comportamiento, es decir, hacer abstracciones.

2.6.3 Sin efectos secundarios

Otra regla en el paradigma de la programación funcional es la de no tener efectos secundarios. 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.*

Se deduce del ejemplo anterior. Siempre que mi estado no cambie, es poco probable que ocurra **un resultado no deseado** en las ejecuciones de funciones. Es una regla en el paradigma de la programación funcional. Esto significa **que una función no puede cambiar ningún estado** fuera de la función. **Cambiar el estado del objeto externo en una función se** conoce como efecto **secundario.**

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

```
public class Calculator{
   public int sum(int a, int b) {
     return a + b;
   }
}
```

El valor devuelto de la suma de llamadas (5,7) sería siempre 12. En consecuencia, es un método de programación funcional correcto. No reproduce ni causa errores.

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

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

En este segundo escenario, dado que las funciones de suma se basan en el valor de la propiedad initVal, el estado del objeto podría llevar al programa a errores indeseables.

2.6.4 Objetos inmutables

La definición que puedes ver en Wikipedia establece que un objeto inmutable es aquel cuyo estado no se puede modificar después de que se crea. O en otras palabras, alguna variable que no puede cambiar su valor.

Un objeto inmutable es un objeto cuyo estado no se puede modificar después de crearlo. En aplicaciones multihilo eso es una gran cosa. Permite que un hilo actúe sobre los datos representados por objetos inmutables sin preocuparse por lo que otros hilos están haciendo.

Es muy común en aplicaciones de tamaño grande y mediano que múltiples programas hagan referencia a la misma variable. Este es el concepto introducido anteriormente, multithreading o multiprocesamiento. Hasta cierto punto, estamos tratando de evitar el efecto secundario de algunos procesos o hilos que podrían modificar el mismo objeto en un período de tiempo.

En Java, es bastante simple crear un objeto inmutable. La forma de proceder es declarar una clase con propiedades privadas, getters y no setters. Los valores de propiedad se fijan o se asignan en el constructor. Una vez creado el objeto, es imposible modificar su estado.

En el siguiente programa, la clase `ObjetoInmutablePersona` posee propiedades privadas. Además, no se declara ningún método set para estas propiedades. Una vez que el programa crea el objeto, queda claro que no es posible modificar sus propiedades.

```
            ObjetoInmutablePersona person = new
ObjetoInmutablePersona("John","Doe", 40);
```

Sin embargo, las propiedades se pueden evaluar desde getters, y eso da como resultado que el programa tenga acceso completo al estado del objeto.

```
System.out.println("Nombre completo de la persona:"+ person.getNombre() + " "
+ person.getApellidos());
```

InmutableObjectPerson.java

```
package paradigmaprogramacionfuncional;

public class ObjetoInmutablePersona {

	private String nombre="";
	private String apellidos="";
	private int edad=0;


	public ObjetoInmutablePersona() {

	}

	public ObjetoInmutablePersona(String nombre, String apellidos, int
edad) {
		super();
		this.nombre = nombre;
		this.apellidos = apellidos;
		this.edad = edad;
	}

	public String getNombre() {
		return nombre;
	}

	public String getApellidos() {
		return apellidos;
	}

	public int getEdad() {
		return edad;
	}




	@Override
	public String toString() {
		return "ObjetoInmutablePersona [nombre=" + nombre + ",
lastName=" + apellidos + ", age=" + edad + "]";
	}
```

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


                ObjetoInmutablePersona person = new
ObjetoInmutablePersona("John","Doe", 40);

                System.out.println("Nombre completo de la persona:"+
person.getNombre() + " " + person.getApellidos());



        }
}
```

2.6.5 Favorecer la recursividad sobre los bucles

Una cuarta regla en el paradigma de programación funcional es favorecer la recursividad sobre el bucle. La recursión utiliza llamadas a funciones 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. Nos vemos más adelante en el curso.

El principio de recursividad es la capacidad de resolver problemas de computación con una función que se llama a sí misma. En la siguiente sección, analizaremos ampliamente la recursividad.

3 Recursividad

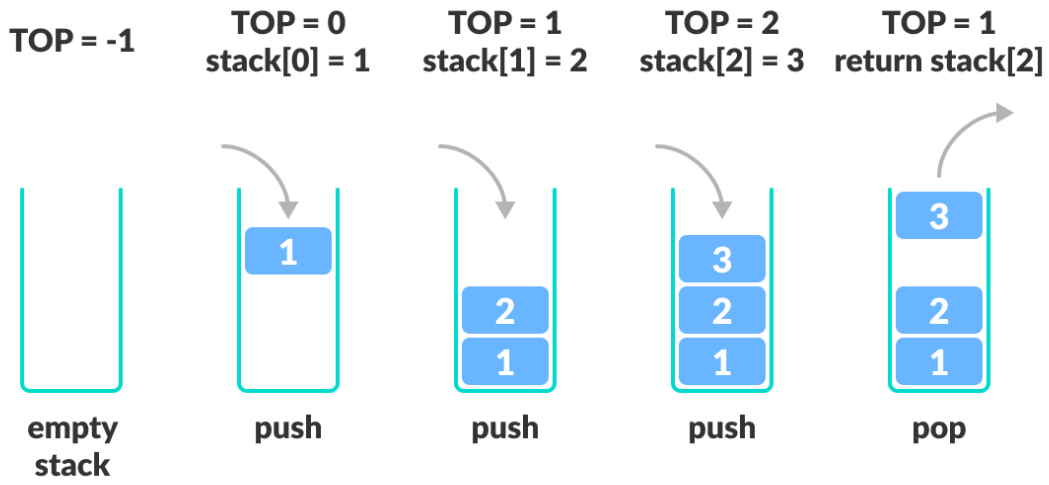
Una de las **herramientas de programación que se hace necesaria en la programación funcional** es el concepto de **recursividad**. En general, la recursividad es el proceso de **definir algo en términos de sí mismo y es algo similar a una definición circular**. El componente clave de un método recursivo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

3.1 La pila de ejecución

3.1.1 ¿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.**



3.2 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 locales**, **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, el **stack, la pila, se vaciará y la pila vacía será destruida** por la máquina virtual java, justo antes de la terminación del programa o hilo de ejecución.

(1)	(2)	(3)	(4)	(5)	(6)	(7)
*(Runtime Stack for Main Thread)						(*Empty Stack)
			second_func()			
		first_func()	first_func()	first_func()		
	main()	main()	main()	main()	main()	

```
/**
 *
 */
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");
        }

}
```

3.3 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**.

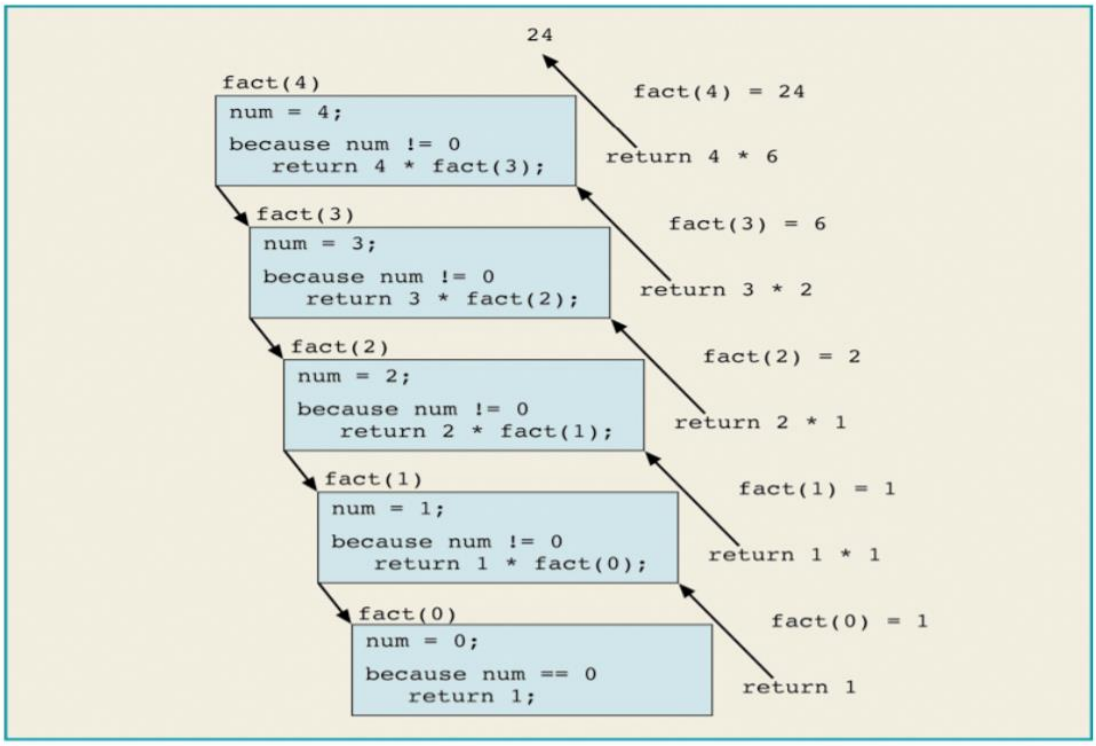
```
public static Long factorial(long n)
        {
           if (n <= 1) // base case
              return 1;
           else
              return n*factorial(n-1);
        }
```

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*factorial(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

```
package paradigmaprogramacionfuncional;

import java.util.Scanner;

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 millón 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?

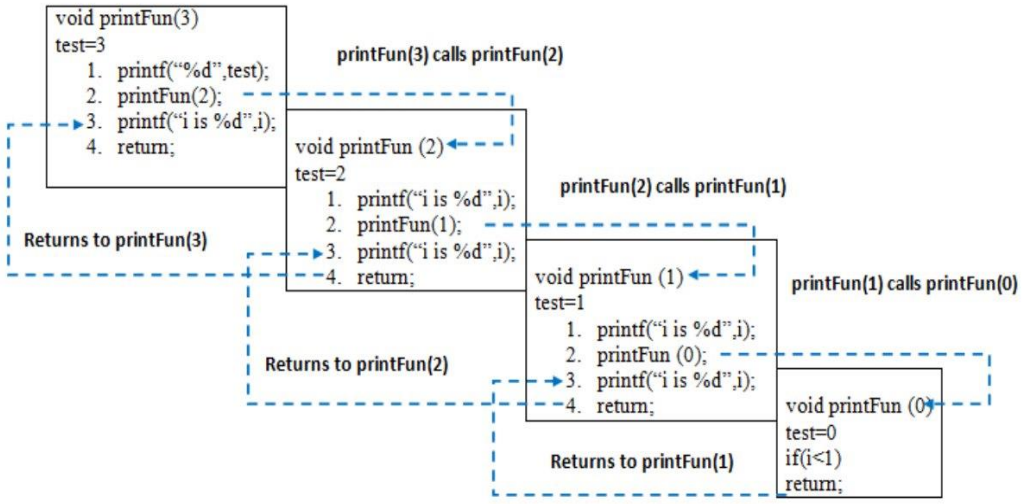
Cuando se llama a cualquier función desde main(), se le **asigna memoria en la pila. Una función recursiva se llama a sí misma**, la **memoria de la función llamada se asigna encima de la memoria asignada a la función de llamada** y **se crea una copia diferente de las variables locales para cada llamada 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.

```
package paradigmaprogramacionfuncional;
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;
        printFun(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**.

3.4 Ejercicios.

1. Calcular la serie de Fibonacci 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 $(n+1)$**-ésimo término** es.

$$a_n = a_{n-2} + a_{n-1}$$

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.

4 Composición en interfaces funcionales avanzada

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 sobrescribiendo** los métodos **and y or del interfaz tipo Predicate.**

Vamos a **añadir un nuevo método a sobreescribir, el método 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, tienes la página de Oracle, los creadores de java.

Aquí tenéis 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 sabéis que, para **el or,** con que uno de los 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 que se compone de devuelve true**, cuando se llama a sus métodos test() con el mismo parámetro de entrada que el predicado compuesto. En este punto podéis ver 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 sumar y, a continuación, a la función multiplicar. 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 andThen():

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 **sobrescribir 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** 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**, podéis 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 resultados para potencia y producto usando el método ejecutaBifunction

5.3 Especialización de Functions primitivos

Vamos a **ver distintos tipos 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: Son usados cuando queremos usar tipos primitivos en nuestros interfaces funcionales.

En el siguiente ejemplo entenderás las **especializaciones.** Nos permiten trabajar *con tipos básicos*, **pero nos rompe la pureza funcional de nuestras interfaces.** Puede **usarlo para realizar conversiones a tipos básicos a través** de interfaces funcionales. Como habrás notado, estamos utilizando, la versión de clase java Legacy de los tipos básicos. **Double, Integer, Long, Boolean** son clases que funcionan exactamente como los tipos básicos asociados, **double, long, boolean.**

Look at the next declarations :

`IntFunction<String> convertimosIntString,` recibe un String y devuelve un tipo básico int.

`ToIntFunction<String> convertimosString` recoge un tipo básico int, dentro del interfaz y devuelve un 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 qué Java introduce todos estos conceptos?** Vamos a intentar desgranar por partes el porqué 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 nuestro 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 sola 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 sobrescribir 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 sobrescritas 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 estadísticas 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 sí.** 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);
        }
```

CreaPersonasFuncionalCompare.java

```
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 Actividad guiada interfaz comparable

7.1 Interfaz Comparable

Comparable<T> de java.lang está disponible desde la versión 2.0 de java. El interfaz comparable va a definir el método compareTo(objeto) y compare en la implementación debemos devolver 1 si el objeto es mayor que el pasado como parámetro. Se devuelve 0 si son iguales, y -1 si el objeto pasado como parámetro es mayor.

```
public interface Comparable<T> {
    public int compareTo(T otro);
}
```

En el ejemplo podéis ver como implementamos el interfaz comparator en la clase futbolista. Sobrescribimos el método compareTo para comparar un futbolista por calidad.

```
public int compareTo(Futbolista o) {
        // TODO Auto-generated method stub
```

```
         return (this.calidad > o.calidad)  ? 1 : ((this.calidad == o.calidad
) ? 0: -1 );


}
```

7.1.1 Ejemplo 1 Comparable

Futbolista.java

```
public class Futbolista implements Comparable<Futbolista> {
private String nombre;
private String posicion;
private int calidad;

public Futbolista(String nombre, String posicion, int calidad) {
         this.nombre=nombre;
         this.posicion=posicion;

         this.calidad=calidad;

}




public String getNombre() {
         return nombre;
}




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

```
public String getPosicion() {
        return posicion;
}



public void setPosicion(String posicion) {
        this.posicion = posicion;
}



public int getCalidad() {
        return calidad;
}



public void setCalidad(int calidad) {
        this.calidad = calidad;
}



public int compareTo(Futbolista o) {
        // TODO Auto-generated method stub



         return (this.calidad > o.calidad)  ? 1 : ((this.calidad == o.calidad
) ? 0: -1 );



}

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

```
Futbolista f1 = new Futbolista("Peri", "Delantero",6);

         Futbolista f2 = new Futbolista("Jari", "Defensa",7);



        if (f1.compareTo(f2)>0) {

            System.out.println("futbolista " + f1.getNombre() + " es mejor qu
e futbolista " + f2.getNombre());

        }

          else if (f1.compareTo(f2)==0) {

             System.out.println(f1.getNombre()  +  " es igual de bueno que "
+ f2.getNombre());

        } else if (f1.compareTo(f2)==-1) {

           System.out.println(f1.getNombre()  +  " es peor que " + f2.getNombr
e());

        }



        }

}
```

7.2 Actividad independiente interfaz Comparable más ejercicio.

Hemos **declarado este interfaz Comparable para artículo**. Vuestra misión será, **crear un función main** para **ejecutar el programa**. Crear dos artículos y compararlos. Escribid por pantalla que articulo es mayor.

Articulo

```
class Articulo implements Comparable<Articulo> {

    public String codArticulo;

    public String descripcion;

    public int cantidad;


    @Override

    public int compareTo(Articulo o) {
        return codArticulo.compareTo(o.codArticulo);
    }

}
```

7.3 Actividad guiada Interfaz Comparator

Comparator<T> , **donde T es el tipo a comparar**, es el interfaz **encargado de permitirnos el poder comparar 2 elementos** en una colección. Por tanto pudiera parecer que es igual a la interfaz **Comparable** (recordemos que esta interfaz nos obligaba a implementar el método **compareTo** (Object o)) que **hemos visto anteriormente en el paquete java.lang**. Aunque es cierto que es similar, **estas dos interfaces en absoluto son iguales.**

Mientras que **Comparable** nos obliga a implementar el método **compareTo** (Object o), la interfaz **Comparator** nos obliga a implementar el método **compare** (Object o1, Object o2). **Comparator** es un interfaz incorporado para trabajar con **Stream** en su método

El **primer método nos sirvió para implementar** un **método de comparación** en una clase de ejemplo a la que **denominamos Futbolista**. La implementación de este método **se hace para indicar el orden natural** de los elementos de esa clase.

Podemos **ordenar con este interfaz objetos con otro criterio**. Por ejemplo, para la clase **Futbolista podemos querer ordenarla por posición**. Pues que el orden natural definido no nos sirve y debemos de recurrir a la interfaz **Comparator** para implementar el **método compare** (Object o1, Object o2) **definiendo el método deseado**.

Podéis añadir este código a la función main del ejemplo anterior. De esta manera ordenamos por nombre y por posición con el método sort de la clase Stream.

Importante aquí que os deis cuenta que podemos **construir Comparator, o sobreescribiendo o con un expresión lambda**, porque **Comparator es un interfaz legacy, aunque no es funcional, es anterior a java 8**, como **sólo declara un método abstracto**, puede **sobreescribirse con lambdas**, y ser

tratado como un interfaz funcional más.

Expresión lambda

```
Comparator<Futbolista2> comparador2 =
                (fut1,fut2)-> (fut1.getCalidad() > fut2.getCalidad())  ? 1
                                    : ((fut1.getCalidad() == fut2.getCalidad())
? 0: -1 );
```

Sobrescribiendo y creándolo con **clase anónima.**

```
        Comparator<Futbolista2> comparador = new Comparator<Futbolista2> () {
            @Override
            public int compare(Futbolista2 o1, Futbolista2 o2) {
                return (o1.getCalidad() > o2.getCalidad())  ? 1 : ((o1.getCa
lidad() == o2.getCalidad()) ? 0: -1 );
            }


        };
```

Futbolista2.java

```
import java.util.Comparator;


/**
 *
 * @author carlo
 */


public class Futbolista2  {



private String nombre;
```

```
private String posicion;
private int calidad;

public Futbolista2(String nombre, String posicion, int calidad) {
        this.nombre=nombre;
        this.posicion=posicion;

        this.calidad=calidad;

}

public String getNombre() {
        return nombre;
}
public void setNombre(String nombre) {
        this.nombre = nombre;
}
public String getPosicion() {
        return posicion;
}
public void setPosicion(String posicion) {
        this.posicion = posicion;
}
public int getCalidad() {
        return calidad;
}
public void setCalidad(int calidad) {
        this.calidad = calidad;
}

 public static void main(String[] args) {


        Futbolista2 f1 = new Futbolista2("Peri", "Delantero",6);

        Futbolista2 f2 = new Futbolista2("Jari", "Defensa",7);
```

```
Comparator<Futbolista2> comparador2 =
                (fut1,fut2)-> (fut1.getCalidad() > fut2.getCalidad())  ? 1
                                    : ((fut1.getCalidad() == fut2.getCalidad())
? 0: -1 );



        Comparator<Futbolista2> comparador = new Comparator<Futbolista2> () {
            @Override
            public int compare(Futbolista2 o1, Futbolista2 o2) {
                return (o1.getCalidad() > o2.getCalidad())  ? 1 : ((o1.getCa
lidad() == o2.getCalidad()) ? 0: -1 );
            }



        };

        if (comparador.compare(f1,f2)>0) {

            System.out.println("futbolista " + f1.getNombre() + " es mejor qu
e futbolista " + f2.getNombre());

        }

         else if (comparador.compare(f1,f2)==0) {

             System.out.println(f1.getNombre()  +  " es igual de bueno que "
+ f2.getNombre());

        } else if (comparador.compare(f1,f2)==-1) {

             System.out.println(f1.getNombre()  +  " es peor que " + f2.getNo
mbre());

        }

         }
```

```
}
```

El **interfaz Comparator también nos da la posibilidad** de definir el método **AndThen,** para realizar **una acción posterior** como hemos visto en el interfaz Consumer o el Function **para combinar comparaciones**, a **partir de Java 8**, **permite composición**.

7.3.1 Actividad independiente interfaz comparator

1. Insertar **tres Artículos del ejemplo 2** del interfaz comparable y **ordenarlos por descripción,** modificando el comparador de Comparable.
2. Insertar un **nuevo interfaz Comparator** para ordenar **articulo por cantidad**.
3. Crear **un interfaz Comparator para futbolista** que lo ordene **por nombre**.

8 Actividades de refuerzo

1. **Realizar un interfaz funcional PruebaFactorial** con **la función abstracta factorial**. Realizar la **expresión lambda para el cálculo del factorial** y mostrar los resultados por pantalla.
2. **Realizar un interfaz funcional EsPrimo** de **tipo predicate**. Realizar la **expresión lambda para comprobar que el número pasado es primo** y mostrar los resultados por pantalla.
3. **Realizar un interfaz funcional SumaN** de **tipo function que calcule la suma de los n primeros números impares. Recibirá de parámetro n.**
4. **Realizar un interfaz funcional** función de tipo función y su expresión lambda **que calcule el elemento n de la serie**

   1 , 1 +1/2, 1 + 1/3 +1/4, 1 + 1/5 …. 1+ 1/n.

5. Realizar **una interfaz de tipo función y su expresión lambda** que **calcule la suma de la serie anterior**.

9 Bibliografía y referencias web

Referencias web

Tutoriales de Java Jacob Jenkov

http://tutorials.jenkov.com/

Tutoriales de Java arquitectura Java

https://www.arquitecturajava.com/

Repositorio Github Venkat Subramanian

https://github.com/venkats

Tutoriales Java geeksforgeeks

https://www.geeksforgeeks.org/

Tutoriales Java Baeldung

https://www.baeldung.com/

Bibliografía

Functional Programming in Java: Harnessing the Power Of Java 8 Lambda Expressions, Venkat Subramanian, The Pragmatic Programmers, 2014

Programación, Alfonso Jímenez Pérez, Francisco Manuel Pérez Montes, Paraninfo, 1ª edición, 2021