Unidad 6. Java 8. Clases anónimas. Expresiones lambda e interfaces funcionales.

Contenido

1 Actividad inicial motivacional.

Comparativa de velocidades de manejo de datos con Java 8 funcional y paralelismo con respecto a versiones anteriores, ejecutando el siguiente ejemplo. Los alumnos probarán el ejemplo en su workspace para la unidad 4. Se introduce el concepto de procesadores multinúcleo y multithreading.

```
package actividadinicialcomparativavelocidad;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.concurrent.TimeUnit;

public class ParalelismoStreamsComparativa {


        private static List<Integer> buildIntRange() {
            List<Integer> numbers = new ArrayList<>(5);
            for (int i = 0; i < 6000 ;i++)
              numbers.add(i);
            return Collections.unmodifiableList(numbers);
          }

        public static void main(String[] args) {
            List<Integer> source = buildIntRange();


            System.out.println(" EMPEZAMOS LA ACTIVIDAD PACIENCIA ");

            long start = System.currentTimeMillis();
            for (int i = 0; i < source.size(); i++) {
              try {
                TimeUnit.MILLISECONDS.sleep(1);
              } catch (InterruptedException e) {
                e.printStackTrace();
              }
            }
            System.out.println("Modo tradicional: " +
(System.currentTimeMillis() - start) + "ms" + " MUY LENTO MUY LENTO" );
```

```
            start = System.currentTimeMillis();
            source.stream().forEach(r -> {
              try {
                TimeUnit.MILLISECONDS.sleep(1);
              } catch (Exception e) {
                e.printStackTrace();
              }
            });
            System.out.println("Stream Con procesado secuencial: " +
(System.currentTimeMillis() - start) + "ms" + " IGUAL DE LENTO IGUAL DE
LENTO");


            start = System.currentTimeMillis();
            source.parallelStream().forEach(r -> {
              try {
                TimeUnit.MILLISECONDS.sleep(1);
              } catch (Exception e) {
                e.printStackTrace();
              }
            });
            System.out.println("parallelStream : Con procesado paralelo QUE
RÁPIDO " + (System.currentTimeMillis() - start) + "ms" + " APROVECHANDO LA
POTENCIA DE MI PROCESADOR FORKJOINPOOL");



            System.out.println(" ¿COMO QUEREIS APRENDER A PROGRAMAR? ");
          }

}
}
```

2 Clases Anonimas

2.1 Clase anónima

Una **clase interna anónima** es una forma de **clase interna que se declara y crea una instancia con una sola declaración**. Como consecuencia, **no hay un nombre para la clase que pueda usarse** en otra parte del programa; Es decir, es anónimo.
Las **clases anónimas se utilizan normalmente en situaciones** en las que es necesario **poder crear una clase de ligeras** que se pasan **como parámetro**. Esto normalmente se realiza con una interfaz. Por ejemplo:
Ejemplo

```
public static Comparator<String> CASE_INSENSITIVE =
        new Comparator<String>() {
            @Override
            public int compare(String string1, String string2) {
                return
string1.toUpperCase().compareTo(string2.toUpperCase());
```

```
            }
        };
```

Por eso como hemos visto en todos los ejemplos anteriores, **podemos crear variables de tipo Interfaz**, a **partir de clases anónimas** es una **práctica muy utilizada** en programación Java en la actualidad.

2.1.1 Constructores en clases anónimas. Actividad guiada clases anónimas.

En el siguiente ejemplo podeis ver como **declaramos una clase anónima, y sobrescribimos sus métodos**. Es una **nueva manera de realizar herencia de manera dinámica**. Es **muy habitual en la programación actual**. Siempre intentamos **cambiar de manera dinámica el comportamiento de las clases**. **Las expresiones lambda son la quinta esencia** de esta **nueva tendencia** en programación.

El **objetivo de esta herencia dinámica es no tener que hacer otro fichero .java** para **una clase nueva de la que sólo queremos modificar un método**. Por eso hacemos esta **herencia dinámica y en tiempo de ejecución**
Lo que no podemos **en una clase anónima que va a heredar de otra clase de manera dinámica es declarar un constructor nuevo**, debemos usar el de la clase original, como veis en el ejemplo.

ClaseBase.java

```
public class ClaseBase {

        public ClaseBase (String nombre) {


        }

        public void metodoNombre(String nombre) {

                System.out.println("El nombre es:" + nombre);
        }


    public static void main(String[] args) {



        ClaseAnonima anon = new ClaseBase ("Marta") {
        @Override
        public void metodoNombre(String nombre) {

        System.out.println("Sobreescribiendo el método nombre anónimamente "
+ nombre);
        }
    };

        anon.metodoNombre("Marta");
```

```
    }

}
```

El **constructor implícito para nuestra subclase anónima a partir de la ClaseBase** llamará a un constructor de `ClaseBase` **que coincida con la signatura de la llamada al constructor** `ClaseBase (String)` **. Si no hay ningún constructor disponible, obtendrá un error de compilación**. Cualquier excepción lanzada por el constructor emparejado también es lanzada por el constructor implícito. El punto en el que creamos la Clase Anónima es la sobreescritura. En el momento que **sobreescribimos estamos creando una clase que hereda de la anterior**. Pero a esta clase heredada **no le podemos cambiar el constructor**, podemos usar el del padre llamando a super(), o a través del new..

2.1.2 Clases anónimas a partir de un interfaz

Naturalmente, **lo anterior no funciona cuando se implementa una interfaz.** Cuando creas **una clase anónima desde una interfaz**, la clase superclase es `java.lang.Object` que solo **tiene un constructor sin argumentos**. Pero **igualmente podremos crear una clase anónima a partir de un interfaz**. Lo podeis observar en el siguiente Ejemplo.

Declaro el interfaz `ClaseAnonima.` Y construyo **una clase anónima a partir de la declaración del interfaz,** y en el new anónimo **sobreescribo** el método abstracto del interfaz **metodonombre**. Como veis **la clase no existe, estoy instanciando la clase y recogiéndola** en un tipo que es de tipo **ClaseAnonima, que es un interfaz.**

```
interface ClaseAnonima {


                public void metodoNombre(String nombre) ;


                }
```

El tipo de la **clase de la variable anon es el interfaz**. Pero lo que he creado **es una clase nueva, no un interfaz**. En resumen, **así se realiza la creación anónima de clases** a partir de **interfaces.**

```
ClaseAnonima anon = new ClaseAnonima() {
                @Override
                public void metodoNombre(String nombre) {

                System.out.println("Sobreescribiendo el método nombre
```

```
anónimamente " + nombre);
                }


            };
```

```
public class ClaseAnonimaInterfaz {

         interface ClaseAnonima {


                public void metodoNombre(String nombre) ;


                }


            public static void main(String[] args) {



                ClaseAnonima anon = new ClaseAnonima() {
                @Override
                public void metodoNombre(String nombre) {

                System.out.println("Sobreescribiendo el método nombre
anónimamente " + nombre);
                }


            };

                anon.metodoNombre("Marta");

            }

        }
```

2.1.3 Clase anónima lambda

Es parte del espíritu de las expresiones lambda. Sobrescribir métodos de interfaces para crear clases. Pero fijaos que aquí **no nos hace falta escribir el método otra vez**, usar la **etiqueta @Override**. Es lo que llamamos **sobreescritura implícita**. Pero tiene el mismo efecto que el caso anterior. Además, **no necesitamos hacer un new**, solo escribir la **expresión lambda.** El new se hace de manera implícita. Esto **ya lo vimos en el tema anterior**. La clase se crea de manera automática, con un solo método, el que define el interfaz funcional.

```
ClaseAnonima anon = (nombre)->
                System.out.println("Sobreescribiendo el método nombre
anónimamente " + nombre);
```

```
interface  ClaseAnonimaInterfazlambda {



         interface ClaseAnonima {


                public void metodoNombre(String nombre) ;


                }


            public static void main(String[] args) {



                ClaseAnonima anon = (nombre)->
                System.out.println("Sobreescribiendo el método nombre
anónimamente " + nombre);


                anon.metodoNombre("Marta");

            }

        }
```

2.1.4 Interfaces funcionales legacy

En **versiones anteriores a la versión 8 de Java, hay interfaces** que ya pueden ser **considerados como funcionales**. Por ejemplo, **los interfaces Runnable y Callable.** No son objetivos de este curso. **Runnable es un interfaz que se usa para la programación de hilos y Callable para la programación de Tareas**. Podemos crear objetos de estos **interfaces directamente con clases anónimas**, como hemos visto en el ejemplo anterior.

Thread thread = new Thread(() -> System.out.println("Hello From Another Thread"));
thread.start();

2.1.5 Actividad independiente clases anónimas

1. Crea un interfaz InterfazSalida con el método abstracto void imprimir(String Salida).
2. Crea una variable de clase abstracta que sobreescriba el método e

imprima:

/**********************************/

*** La salida es ***

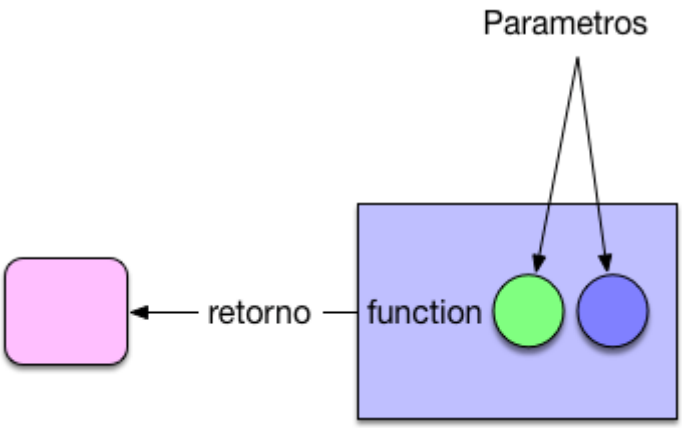
*** aquí la cadena de salida

*****

3 Actividad Guiada Interfaces funcionales predefinidos

4 Interfaces funcionales predefinidos

Vamos a ver una serie de interfaces funcionales predefinidos en java para el uso de expresiones funcionales y Streams. **Son Predicate, Consumer, Supplier, y Function**. Todos estos **interfaces funcionales los vamos a usar en conjunto para aplicar operaciones en Stream**. Vamos a describir uno a uno. Como interfaces funcionales que son realizan una sola tarea o función encapsulada, y nos permiten como ya hemos comentado la programación funcional.



4.1 Predicate

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

Representa el tipo de operación que definiríamos como condicional sobre el o los parámetros que recibe. **Es tarea del programador definir esa operación por medio de expresiones lambdas** o funciones anónimas.

En Java no tenemos funciones independientes. Además, los métodos no son ciudadanos de primera. (No **se pueden agregar a colecciones** ni **pasar a métodos como parámetros**). Por lo tanto, **definimos interfaces y creamos**

objetos a partir de estas interfaces. Estos objetos se pueden pasar a métodos como `Iterables.filter()`. Con las **lambdas de Java es mucho más fácil trabajar con predicados.**

4.1.1 Ejemplo Inicial

En este ejemplo **vamos a ver el uso básico de Predicate, el uso inicial por decirlo de alguna manera.** Crearemos **unos cuantos interfaces predicate y los probaremos.** Además empezaremos a introducir **el principal uso de los interfaces funcionales que no es otro que poder pasar una función como parámetro** para que sea ejecutada dentro de otra función.

Primero **veremos un interfaz sencillo.** Lo definimos para que reciba la edad, un entero. Devolverá un booleano como todos los interfaces. **Devuelve true si la variable i, la edad es mayor que 18.** Lo definimos para que devuelva false en caso contrario.

```
Predicate<Integer> adultoEuropa = (i)-> i>18;
```

Para **probarlo debemos llamar al método del interfaz test()** con un valor entero, en este caso 20, y **la llamada devolverá true.**

```
if (adultoEuropa.test(20))
                        System.out.println("La persona es adulta");
```

Muy **importante en este ejemplo es que podemos asignar una función** a un interfaz funcional, **no sólo una lambda** usando el operador ::, como es estática `EjemploPredicateInicial::EsMayor;` `nombredelaclase::función.` Si os fijais en el ejemplo

```
Predicate<Integer> adultoEuropa2 =
                                EjemploPredicateInicial::EsMayor;
```

Se asigna una función estática, podría ser también una función no estática lo veremos más adelante en el curso. Como veis la función **la puedo asignar porque recibe un Integer, y devuelve un booleano** como el Predicate definido. EL **operador :: significa enlace o puntero a una función**

```
        private static boolean EsMayor(Integer edad) {
```

```
                return edad>18;
        }
```

Cuando **pruebe el predicate con test**, se ejecutará **la función EsMayor(20)** con el parámetro recibido 20. Los **interfaces funcionales sirven para guardar funciones y lambdas en variables.**

```
                if (adultoEuropa2.test(20)) {

                        System.out.println("La persona es adulta");
                }
```

Para finalizar el ejemplo, veréis que **se puede pasar una función o una lambda como parámetros** usando interfaces funcionales. **Hemos definido la función**, `imprimirSiEsMayorDeEdad` que **recibe como segundo parámetro** un **Predicate** `Predicate<Integer> funcionEdad`.

```
private static void imprimirSiEsMayorDeEdad(Integer edad, Predicate<Integer>
funcionEdad) {
```

Puedo usar **ese Predicate dentro de la función llamando al método test** del parámetro **funcionEdad** de tipo Predicate.

```
if (funcionEdad.test(edad))
```

Como comprobareisen el ejemplo a esta función le puedo pasar:

1. Una variable de tipo Predicate

```
imprimirSiEsMayorDeEdad(20,adultoEuropa2);
```

2. Una función

```
  imprimirSiEsMayorDeEdad(20,EjemploPredicateInicial::EsMayor);
```

3. O una función lambda

```
imprimirSiEsMayorDeEdad(34, (n)-> n>=18);
```

Con este **sistema puedo pasar funciones o funciones anónimas** (expresión lambda) como parámetros. O almacenar **una función o expresión lambda** en una variable.

EjemploPredicateInicial.java

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

public class EjemploPredicateInicial {

        private static boolean EsMayor(Integer edad) {

                return edad>18;
        }


        private static void imprimirSiEsMayorDeEdad(Integer edad, Predicate<I
nteger> funcionEdad) {

                if (funcionEdad.test(edad))


                        System.out.println("La persona es mayor de edad con e
dad: " + edad);
                else

                        System.out.println("La persona es menor de edad con e
dad: " + edad);

        }

          public static void main(String[] args) {


                Predicate<Integer> adultoEuropa = (i)-> i>18;
                Predicate<Integer> adultoUSA = (i)-> i>21;
                Predicate<String> ejemploNombre = (n)-> n.startsWith("A");
```

```
                    Predicate<Integer> adultoEuropa2 =
                                        EjemploPredicateInicial::EsMayor;

                    if (adultoEuropa.test(20))
                            System.out.println("La persona es adulta");

                    if (ejemploNombre.test("Antonio"))
                            System.out.println("El nombre de la persona empiez
a por A");

                    if (adultoEuropa2.test(20)) {

                            System.out.println("La persona es adulta");
                    }


                    imprimirSiEsMayorDeEdad(20,adultoEuropa2);
                    imprimirSiEsMayorDeEdad(20,EjemploPredicateInicial::EsMayo
r);
                    imprimirSiEsMayorDeEdad(20,adultoUSA);

                    imprimirSiEsMayorDeEdad(34, (n)-> n>=18);




            }

}
```

4.1.2 Ejemplo 2. Para quien quiera saber más con listas.

En este ejemplo **vamos a usar el interfaz funcional como base para crear una clase**. **No es el uso habitual** sólo quiero que lo veáis para que **entendáis completamente que un interfaz funcional** se usa exactamente igual que uno normal. También introducimos **un nuevo concepto que extenderemos en el siguiente tema**, la **composición de funciones** o interfaces funcionales.
Si os fijaos **creo una clase MayorQueCinco que implementa el interfaz Predicate** `class MayorQueCinco<E> implements Predicate<Integer>`. Como es un **interfaz con un solo método abstracto**, sólo tengo que **sobreescribir el método abstracto del interfaz test.**

```
    @Override
    public boolean test(Integer v) {
```

Y luego **declarar la clase, y hacer un new** para obtener un **objeto de la clase MayorQueCinco.** Y usarla **variable llamando a test**

```
      MayorQueCinco <Integer> mqc = new MayorQueCinco <> ();
```

```
if (mqc.test(n))
```

Como **hemos dicho antes puedo crear un interfaz Predicate combinando varios interfaces Predicate.** Tenemos el segundo interfaz menor que nueve, mqn definido en nuestro programa.

```
      Predicate <Integer> mqn = (i) -> i<9;
```

Podemos combinar **mayorquecinco mqc** y **menorquenueve mqn**, con **dos funciones default que nos ofrece el interfaz**, Predicate, **and y or.** En el ejemplo**, creamos una variable interfaz nueva** llamando al método and, `mAnd= mqn.and(mqc);`
mAnd será equivalente a aplicar un **and lógico** sobre los dos interfaces, `(i) -> i>5` && `(i) -> i<9`. Es decir mAnd **devolverá verdadero cuando reciba un número entre 5 y 9**.

```
Predicate <Integer> mAnd= mqn.and(mqc);
```

Ejecutar el programa y comprobarlo

```
Escriba un número:
6
el número 6 es mayor que cinco
el número 6 es mayor que 5 y menor que nueve
```

EjemploPredicate2.java

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

class MayorQueCinco<E> implements Predicate<Integer> {

    @Override
    public boolean test(Integer v) {

        Integer five = 5;

        return v > five;
    }
}




public class EjemploPredicate2 {



  public static void main(String[] args) {


          Scanner sc = new Scanner(System.in);

      MayorQueCinco <Integer> mqc = new MayorQueCinco <> ();


      System.out.println("Escriba un número:");
      int n= sc.nextInt();
```

```
      if (mqc.test(n))
        System.out.println("el número " + n +" es mayor que cinco");
      else
          System.out.println("el número " + n +" es menor que cinco");

      Predicate <Integer> mqn = (i) -> i<9;
                Predicate <Integer> mAnd= mqn.and(mqc);

                if (mqn.test(n))
                        System.out.println("el número " + n +" es mayor que 5
y menor que nueve");
                else
                        System.out.println("el número " + n +" es menor que 5
y menor que nueve");
            }




}
```

4.1.3 Notas Cornell. Comentario de código

Comenta las líneas señaladas con el marcado del ejemplo anterior y di que hacen. Apóyate en los apuntes para realizar la tarea.

4.1.4 Ejemplo Predicate con listas. Actividad de ampliación

En este ejemplo vamos a **analizar el uso de un interfaz predicate**. Como veis es un interfaz que **devuelve un booleano después de aplicar una operación lógica** sobre un objeto. El ejemplo es orientativo **para que veáis como vamos a usar los interfaces funcionales en el futuro.**

```
Predicate<Persona> edad = (p) -> p.getEdad() >= 35l;
```

EjemploPredicate.java

```
import java.time.LocalDate;
```

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



public class EjemploPredicate {


        /**
         * @param args
         */
        public static void main(String[] args) {
List<Persona> listaPersonas = new ArrayList<>();


                listaPersonas.add(new Persona("12345678A", "Pepe", "Perez", L
ocalDate.of(1990, 1, 2)));
                listaPersonas.add(new Persona("23456789B", "Juan", "Martínez"
, LocalDate.of(1981, 2, 3)));
                listaPersonas.add(new Persona("34567890C", "Ana", "Ramírez",
LocalDate.of(1972, 3, 4)));
                listaPersonas.add(new Persona("45678901D", "María", "López",
LocalDate.of(1993, 4, 5)));



                listaPersonas
                        .stream()
                        .filter((p) -> p.getEdad() >= 35l)
                        .forEach(System.out::println);



                Predicate<Persona> edad = (p) -> p.getEdad() >= 35l;
                Predicate<Persona> nombre = (p) -> p.getApellidos().contains(
"e");
                Predicate<Persona> complejo = edad.or(nombre);


                System.out.println("");
                listaPersonas
                        .stream()
                        .filter(complejo)
```

```
                .forEach(System.out::println);
```

Tenemos **tres métodos útiles en el interfaz predicate**. Métodos útiles para constuir predicados complejos (*and, or, negate*…).

Mirad **como he creado un predicado más complejo** a partir **de dos predicados simples con el or**. Nos servirá igualmente **para modificar condiciones en tiempo** de ejecución con el predicado edad y el predicado nombre

```
Predicate<Persona> complejo = edad.or(nombre);
```

```
Predicate<Persona> edad = (p) -> p.getEdad() >= 35l;
Predicate<Persona> nombre = (p) ->
p.getApellidos().contains("e");
Predicate<Persona> complejo = edad.or(nombre);
```

4.1.5 Actividad independiente predicate

1. Realizar una **expresión Lambda y un Predicate** que nos indique si un número es **divisible por 7** o no.

2. Realizar una **expresión Lambda y un Predicate** que nos indique si un número es **divisible por 7** o por 4 usando el anterior.

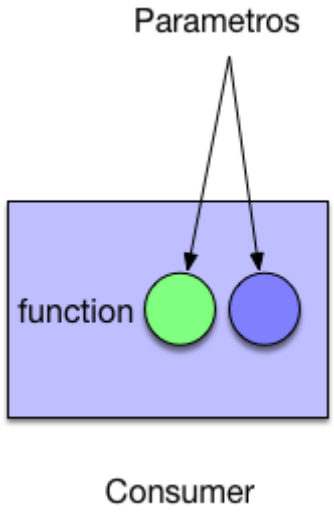
3. Realizar una expresión Lambda y un Predicate para números primos donde el predicado nos indique si el número es primo. **Usar una Lambda de bloque** para hacerlo

4.2 Interfaz predefinido Consumer

El interfaz funcional Consumer <T> pretende representar una función con

un argumento (parámetros) de tipo T que **no retorna nada**. Este **interfaz básicamente va a consumir unos datos de entrada** y va a **realizar una acción**. Es decir, el interfaz **representa o define una operación que realiza una acción y no devuelve resultado**. Es un interfaz predefinido en java con el método abstracto accept para ser implementado por la interfaz lambda. Ofrece el método andThen **default** para **realizar composición de interfaces Consumer.**

accept: ejecuta la acción definida por la expresión lambda o función anónima.
andThen: adicionalmente **podemos definir otra función anónima** que se realizará después de **que la función del accept sea realizada**, al estilo de una **función Callable**, funciones que se ejecutan **después de realizar una determinada acción**.
Para **ejecutar el Interfaz Consumer hay que llamar al método accept**. La lambda o función que le asignamos lo que hace es **sobreescribir el método Accept** del **interfaz consumer**. Como **ya hemos dicho los interfaces funcionales** ofrecen **un solo método abstracto a sobreescribir** con la expresión Lambda. Para **consumers** es **el método Accept.** Para usar el interfaz con una variable, hay que llamar a ese método.




```
Consumer.Accept(i);
```

4.2.1 Ejemplo 1

En este ejemplo podéis ver como primero para cada elemento aplicamos el **accept** que es **escribir el cuadrado del número y luego en el AndThen**, cuando termina el accept **explicamos la operación decimos que hemos imprimido, además del cuadrado del número.**
Sobreescribimos el método get del interfaz con una expresión lambda. Es lo mismo que hemos hecho antes pero directamente..

```
Consumer<Integer> consumer = i -> System.out.println("Consumer normal "+i*i);
```

```
Consumer.Accept(6);
                //Usando expresiones lambda
```

Salida:
Consumer normal 36

```
consumerWithAndThen.accept(7);
```

Consumer normal 36
Consumer With and Then: (hemos imprimido el cuadrado de:6)

Consumer **ejecutará la primera expresión lambda declarada en el. ConsumerWithAndThen**, ejecutará la primero expresionlambda del Consumer inicial, y luego la segunda declarada con el AndThen

Hemos **construido el segundo consumer a partir del primero** componiendo los dos interfaces Consumer. El **resultado es un interfaz consumer, en el que se ejecutarán las dos lambdas definidas**. Al usar **andThen, la primera lambda se ejecuta antes** que la segunda. Podemos usar otro método compose, y el resultado es el contrario, se ejecuta **primero la lambda posterior, la más interna**. Están basados en el concepto matemático de composición de funciones.

```
Consumer<Integer> consumerWithAndThen = consumer.andThen(i -> System.out.prin
tln("Consumer With and Then: (hemos imprimido el cuadrado de:" + i + ")"));
```

Si ejecutáis el programa el primer Consumer produciría este resultado

```
Consumer.Accept(6);
```

```
Consumer normal 36
```

El **segundo Consumer produciría un resultado** como el del primero, obtener el cuadrado, más la **segunda lambda, imprimir que operación hemos hecho**.

```
consumerWithAndThen.accept(7);
```

```
Consumer normal 49
Consumer With and Then: (hemos imprimido el cuadrado de:7)
```

EjemploConsumer.java

```
import java.util.Arrays;
```

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


public class EjemploConsumer {


	/**
	 * @param args
	 */
	public static void main(String[] args) {



Consumer<Integer> consumer = i -> System.out.println("Consumer normal "+i*i);
		Consumer<Integer> consumerWithAndThen = consumer.andThen(i -> System.out.println("Consumer With and Then: (hemos imprimido el cuadrado de:" + i + ")"));
		Consumer.Accept(6);
		//Usando expresiones lambda
		consumerWithAndThen.accept(7);


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

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

				System.out.println("Consumer normal "+t*t);

			}


		};
	}

}
```

Nota: Clases anónimas,

Quiero que os fijeis bien en **el ejemplo en la variable consumer3.** Es exactamente igual que la **variable consumer**. La diferencia **es la manera de implementarlo**. En el **consumer usamos una expresión lambda**. En **la variable consumer3**, estamos usando **una función anónima**. El resultado es el mismo, con dos técnicas diferentes. En amarillo os subrayo **la función anónima** que estamos utilizando o **expresión lambda de bloque**. Observar que para **el new la funciona anónima** es **un poco diferente** a la que **hemos hablado antes en el tema**. No se pone la flecha de operador.

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


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

                                System.out.println("Consumer normal "+t*t);


                        }



                };
```

Como estáis viendo **estamos sobreescribiendo el método accept de forma explícita**. En **consumer lo hacemos de forma implícita** con la **expresión lambda** que es más cómoda. Siempre que **podemos usamos expresiones lambda**. Sino podemos porque el **código es muy complejo, usamos funciones anónimas**. Se puede hacer con todos los interfaces funcionales predefinidos que vamos a ver a continuación.

4.2.2 Ejemplo 2. Con listas para quien quiera saber más. Actividad de ampliación

En **este ejemplo vamos a ver como el método forEach** de la clase Stream es el típico ejemplo de Consumer. Vamos a reutilizarlo con un interfaz consumer. De esta manera podemos aplicarlos las veces que queramos en nuestro código.

EjemploConsumer2.java

```
import java.util.ArrayList;
```

```
import java.util.LinkedList;
import java.util.List;
import java.util.function.Consumer; public class EjemploConsumer2 {
    public static void main(String args[])
    {

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



        mostrar.accept(10);



        Consumer<List<Integer> > modificarLista = list ->
        {
            for (int i = 0; i < list.size(); i++)
                list.set(i, 2 * list.get(i));
        };



        Consumer<List<Integer> >
            mostrarLista = lista -> lista.stream().forEach(a -> System.out.pr
int(a + " "));

        List<Integer> lista = new ArrayList<Integer>();
        lista.add(2);
        lista.add(1);
        lista.add(3);



        modificarLista.accept(lista);


        mostrarLista.accept(lista);
    }
}
```

4.2.3 Actividad independiente Consumer

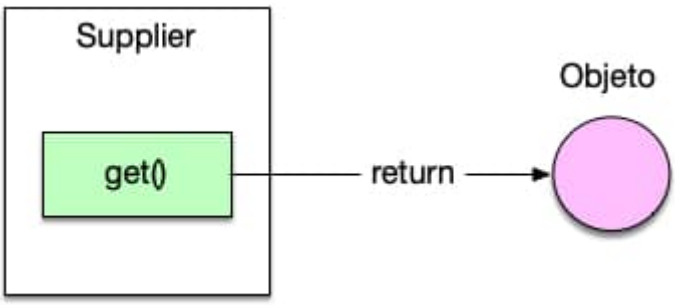
1. Realizar una interfaz consumer que recoja números, modifique el número con la función lambda x -> x^3 +2 y lo imprima por pantalla con la siguiente salida:

```
3 da 29 despues de aplicar el consumer
```

2. Realizar una interfaz **Consumer** que se base en el anterior con el método **andThen** Escriba operación realizada.

4.3 Interfaz Supplier

Supplier<T> **representa una función sin argumentos** que devuelve un valor de tipo genérico T. Este **interfaz como su nombre indica** es para **las operaciones que van a proporcionar un resultado de salida**. El concepto de Java Supplier Interface de ***java*.util.function.*Supplier*** . Es uno de los **conceptos de programación funcional** que a veces **más cuesta entender**. No **tanto por su complejidad sino** por su **posible utilidad**. Este **interface es uno de los más sencillos** ya que solo **tiene un método que nos devuelve un valor**, el método get . Un **Supplier es un proveedor** y por tanto **nos devuelve un valor** o un objeto sin que nosotros le pasemos ningún parámetro.



El **interfaz Supplier te obliga a sobreescribir el método get**. Para usarlo desde una variable y que haga el trabajo hay que usar el método get.

4.3.1 Ejemplo 1. Supplier random.

En el ejemplo **podeis apreciar como los dos supplier van proprocionando** su **método get después de sobreescribir el get del supplier** con una expresión lambda. El **primero para decimales**, el **segundo para enteros** del cero al dos.

Creamos el supplier de tipo Integer, sobre una variable supplier. Recordar que cuando escribimos la expresión lambda, reescribimos el método get.

```
        Supplier<Integer> supplier = () -> new Random().nextInt(3);
```

Para usar el supplier llamamos a su método get desde la variable.

```
        System.out.println(supplier.get());
```

SupplierAleatorio1.java

```
import java.util.Random;
import java.util.function.Supplier;
public class SupplierAleatorio1 {
    public static void main(String args[])
    {

        Supplier<Double> valorAleatorio = () ->Math.random();

        System.out.println(valorAleatorio.get());

        Supplier<Integer> supplier = () -> new Random().nextInt(3);

        System.out.println(supplier.get());


    }

}
```

4.3.2 Ejemplo Supplier 2

En **este caso vamos a trabajar con cadenas en nuestro siguiente ejemplo.** Es **otro ejemplo sencillo donde podéis ver como generamos cadenas** con el interfaz **supplier y expresiones lambda.**

```
Supplier<String> supplier3 = () -> { return new String("Hola Mundo!"); };
```

```
                String valor = supplier3.get();
                System.out.println("Obtención de un valor nuevo: " + valor);
```

Ya sabéis que la **expresión lambda sobreescribe el método get, que es el método abtracto para supplier.**

SupplierString2.java

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



public class SupplierString2 {

    public static void main(String args[])
    {

Supplier<String> supplier3 = () -> { return new String("Hola Mundo!"); };

                String valor = supplier3.get();
                System.out.println("Obtención de un valor nuevo: " + valor);


                  Supplier<Date> supplierFecha= () -> { return new Date(); };
                  Date fechaActual = supplierFecha.get();
                  System.out.println("fechaActual->" + fechaActual);







    }


}
```

4.3.3 Actividad independiente supplier

1. Construir **un supplier y su expresión lambda** que nos proporcione **números enteros aleatorios entre el 1 y el 5**.
2. **Construir un supplier y su expresión lambda** que nos **proporcione el día**. Usar el tipo Date() y su método getDay() para obtener el día.

4.3.4 Ejemplo Supplier con factorías. Difícil. Actividad de ampliación. No Obligatorio

Este interface por lo **tanto no esta tan orientado al trabajo con Java Streams** ya que estos siempre pasan un parámetro en cada iteración sobre el stream. El caso de la interface Supplier es diferente. Vamos **a ver un ejemplo con el que podamos entender un poco mejor este concepto.** Para ello vamos a usar **un ejemplo clásico de Factorias.** Supongamos que tenemos las clases Figura, Rectángulo y Circulo. Estas clases **se pueden relacionar por herencia ya que Figura** sería la **clase abstracta y las otras sus clases hijas**.

EjemploSupplierConFactoria.java

```
public class EjemploSupplierConFactoria {




public static void main(String[] args) {

        Figura fRectangulo=FactoriaGeometrica.crearFigura("Rectangulo");
        Figura fCirculo=FactoriaGeometrica.crearFigura("Circulo");

        System.out.println("Area del rectangulo: " + fRectangulo.area());
        System.out.println("Area del círculo: " + fCirculo.area());



                }
               }
```

FactoriaGeometrica.java

```
public class FactoriaGeometrica {
```

```
  private static HashMap<String,Supplier<Figura>> mapa= new HashMap<>();

  static {

    mapa.put("Rectangulo", Rectangulo::new);
    mapa.put("Circulo", Circulo::new);

  }

  public static Figura crearFigura(String tipo) {

      if (mapa.get(tipo)!=null) {

        return mapa.get(tipo).get();
      }else {
        return null;
      }

  }
}
```

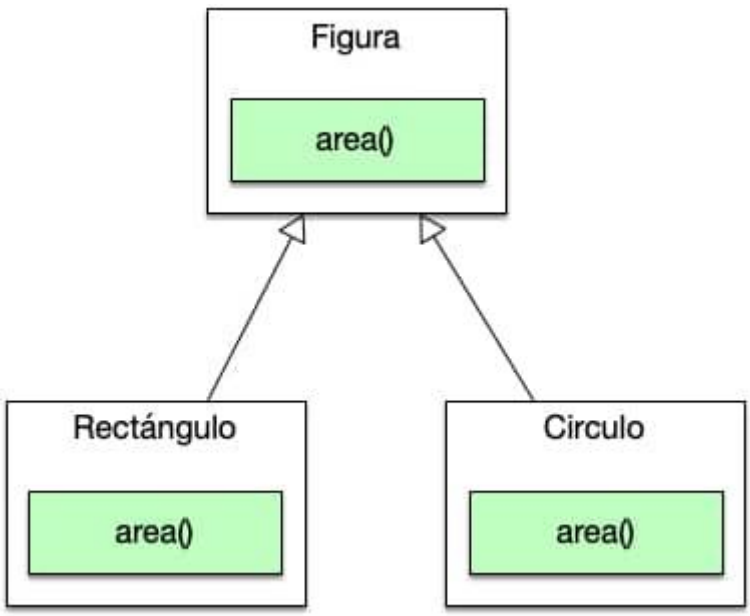


Figura.java

```
public interface Figura {
           public double area();
         }
```

```
public class Rectangulo  implements Figura{
           private int lado1;
           private int lado2;
           public Rectangulo() {
             this(2,2);
           }
           public int getLado1() {
             return lado1;
           }
           public void setLado1(int lado1) {
             this.lado1 = lado1;
           }
           public int getLado2() {
             return lado2;
           }
           public Rectangulo(int lado1, int lado2) {
             super();
             this.lado1 = lado1;
             this.lado2 = lado2;
           }
           public void setLado2(int lado2) {
             this.lado2 = lado2;
           }
           @Override
           public double area() {
             // TODO Auto-generated method stub
             return this.lado1*this.lado2;
           }
         }
```

Circulo.java

```
public class Circulo implements Figura {
           private int radio;
           public Circulo() {
             this(2);
           }
           public Circulo(int radio) {
             super();
             this.radio = radio;
           }
           public int getRadio() {
             return radio;
           }
           public void setRadio(int radio) {
             this.radio = radio;
           }
           @Override
           public double area() {
             // TODO Auto-generated method stub
             return 2 * Math.PI * radio;
           }
         }
```

En este caso estamos **usando como Supplier un método de referencia** a un constructor (**Rectangulo::new**) o (**Circulo::new**) . De esta forma la factoría funcionará de una forma similar, pero **apoyándose en el concepto de Supplier y en un HashMap** que nos **aporta versatilidad** a la hora de **ir añadiendo nuevas implementaciones.** De esta forma **hemos usado una referencia de Supplier** para apuntar **con métodos de referencia** a **cada uno de los constructores**. Recordemos que **un constructor por defecto no recibe nada** y devuelve **una instancia de la clase** . Ese es **el concepto de Supplier**.

4.4 Interfaz funcional Function

Function<T, R> representa una función con **un argumento de tipo T** que

retorna un **resultado de tipo R**. Como su nombre **indica este interfaz funcional va a realizar la labor de una función**. Recogerá **un objeto o valor de entrada y generará una salida**. Es muy **utilizado al igual que Predicate** en las **operaciones de Streams**. El método abstracto a **sobreescribir es Apply()**. Podemos **usar igual que en los interfaces anteriores** el **AndThen** para construir interfaces más complejos.

```
Function <String,String> transformaNombreAndThen= transformaNombre.andThen((name) -> name + " Mas Apellidos");
```

Ejecutaría el primer interfaz Función y además la segunda expresión lambda del AndThen

Fijaos igualmente que **también podemos combinar el uso de dos interfaces**. Que la salida de uno **sea la entrada de otro**.

```
        System.out.println("Transformar nombre and then:" +
                transformaNombreAndThen.apply(transformaNombre.apply(nombre)));
```

4.4.1 Ejemplo 1 interfaz funcional

Como podéis **ver en el siguiente interfaz funcional transformamos un nombre** añadiéndole "Mis Apellidos" **aplicando la función Apply**. Observar bien la ejecución, al

```
import java.util.function.Function;
public class Ejemplo1InterfazFuncion {



    public static void main(String[] args) {
Function <String,String> transformaNombre = (name) -> name + " Mis Apellidos";
        Function <String,String> transformaNombreAndThen= transformaNombre.andThen((name) -> name + " Mas Apellidos");

        Scanner sc = new Scanner(System.in);
        String  nombre;
        System.out.print("Introduzca un nombre: ");
        nombre = sc.nextLine();
```

```
        System.out.println("Transformar nombre:" + transformaNombre.apply("no
mbre"));



        System.out.println("Transformar nombre and then:" +
             transformaNombreAndThen.apply(transformaNombre.apply(nombre)));


}


}
```

4.4.2 Ejemplo 2. Creando una clase que implementa el interfaz Function

En la siguiente clase **SizeOf implementa el interfaz funcional Sizeof**. Como entrada en el interfaz funcional Function tenemos un **String** y **devolvemos un entero con la longitud de la cadena.**

SizeOf.java

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


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


   public static void main(String[] args) {


            SizeOf       sizeOf = new SizeOf();


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


            System.out.println(" Tamaño cadena " + r1.toString());


            Integer r2 = new SizeOf().apply("hola java 9 10 11 y 12");
            System.out.println(" Tamaño cadena " + r2.toString());
```

```
            }
}
```

4.4.3 Ejemplo 3. Interfaz Function con Listas. Actividad de ampliación

En el **siguiente ejemplo con Streams modificamos los nombres de una lista** de personas con un **interfaz función.** El método **peek de Stream** cambia el nombre en cada objeto persona de la lista.

EjemploInterfazFuncionEnStream.java

```
import java.util.function.Function;
public class EjemploInterfazFuncionEnStream {

    public static void main(String[] args) {

        List<Persona> lista=Stream.generate(Persona::new)
                .limit(100)
                .peek((p)->p.setNombre("Asisea"))
                .collect(Collectors.toList());



        for (Persona p: lista) {

            System.out.println(p.getNombre());
        }

    }
}
```

Persona.java

```
public class Persona {

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

```
        private LocalDate fechaNacimiento;


        public Persona() { }


        public Persona(String dni, String nombre, String apellidos, LocalDate
fechaNacimiento) {
                super();
                this.dni = dni;
                this.nombre = nombre;
                this.apellidos = apellidos;
                this.fechaNacimiento = fechaNacimiento;
        }


        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 LocalDate getFechaNacimiento() {
                return fechaNacimiento;
        }

        public void setFechaNacimiento(LocalDate fechaNacimiento) {
                this.fechaNacimiento = fechaNacimiento;
        }

        public long getEdad() {
                return Period.between(fechaNacimiento, LocalDate.now()).get(C
hronoUnit.YEARS);
        }



        }

        }

}
```

4.4.4 Actividad independiente Interfaz funcional

1. Realizar **un interfaz funcional y su expresión lambda** que aplique una función de transformación a una cadena. Le añadirá " lambda expresions" a la cadena

2. **Realizar un interfaz funcional** y **su expresión lambda** de **bloque** que **aplique una función de transformación** a un número. El **número en cuestión se transformará en su mínimo común múltiplo**.

3. **Realizar un interfaz funcional** y **su expresión lambda** de **bloque** que **aplique una función de transformación** a un número. El **número en cuestión se transformará en su factorial**.

4. **Realizar un interfaz funcional** y **su expresión lambda** de **bloque que nos calcule los n primeros números primos y los muestre por pantalla.**

5. **Realizar un interfaz funcional** y **su expresión lambda** de **bloque que nos calcule la suma de los n primeros números primos**

5 Actividad guiada interfaz comparable

5.1 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 );


}
```

5.1.1 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());


        }



        }


}
```

5.2 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);
    }

}
```

5.3 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 );
```

Sobreescribiendo 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 );
            }


        };
```

```
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**.

5.3.1 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**.

6 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**.

7 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