Capítulo 13
Interfaces funcionales de la API
Expresiones lambda
Stream

Interfaces funcionales y expresiones lambda

En las interfaces, distinguimos entre métodos por defecto, estáticos y abstractos. De todos ellos, en la definición de la clase, solamente hay que implementar los últimos. Se llaman ***interfaces funcionales*** a aquellas que tienen un solo método abstracto. Son especialmente importantes porque tienen una sintaxis alternativa que permite una implementación más sencilla.

La más conocida es la interfaz Comparator, que ya hemos usado repetidas veces y que nos va a servir de ejemplo. Vamos a ilustrar su implementación y manejo a través de la lista de clientes del capítulo anterior. Supongamos que, en determinados momentos, queremos hacer una ordenación o una búsqueda por nombres, para lo cual necesitamos un comparador basado en el atributo nombre.

A la hora de implementar una clase comparadora, podemos seguir varios caminos:

Interfaces funcionales y expresiones lambda

Primera forma. Creamos explícitamente una clase ComparaNombres, que implemente la ***interfaz Comparator***, para comparar objetos Cliente basándose en el atributo ***nombre***,

```
class ComparaNombres implementa Comparator<Cliente> {
	public int compare(Cliente c1, Cliente c2) {
		return c1.nombre.compareTo(c2.nombre);
	}
}
```

Creamos un objeto ComparaNombres y lo pasamos a la función donde se va a usar

```
Comparator<Cliente> comp = new ComparaNombres();
Collections.sort(lista, comp); //La Lista queda ordenada por nombres
```

Incluso podríamos prescindir de la variable comp, escribiendo una sola sentencia,

```
Collections.sort(lista, new ComparaNombres());
```

Interfaces funcionales y expresiones lambda

Segunda forma. Si vamos a usar el comparador una sola vez, no necesitamos implementar la clase comparadora explícitamente. Basta crear un objeto con una clase anónima,

```
Comparator<Cliente> comp = new Comparator<Cliente>() {
        public int compare(Cliente c1, Cliente c2) {
                return c1.nombre.compareTo(c2.nombre);
        }
}
Collections.sort(lista, comp);
```

O incluso, prescindiendo de la variable comp,

```
Collections.sort(lista, new Comparator<Cliente>() {
        public int compare(Cliente c1, Cliente c2) {
                return c1.nombre.compareTo(c2.nombre);
        }
} ) ;
```

Interfaces funcionales y expresiones lambda

Tercera forma (expresiones lambda). No es necesario especificar el nombre del método compare() ya que la interfaz Comparator solo tiene ese método abstracto. Por tanto, para implementar una ***interfaz funcional con una expresión lambda***, basta escribir la lista de parámetros y el cuerpo de la función abstracta separados por una flecha (->).

```
Comparator<Cliente> comp =
(Cliente a, Cliente b) -> {return a.nombre.compareTo(b.nombre);};
```

Todo lo que está a la derecha del operador de asignación es la expresión lambda del método compare() de la interfaz Comparator, implementado para comparar nombres. El nombre del método no aparece, ya que Java lo infiere del lado izquierdo, donde aparece el de la interfaz Comparator, cuyo único método abstracto es compare ().Y también infiere el tipo de los parámetros de entrada (Cliente en nuestro caso) que, por tanto, se puede omitir del lado derecho.

Interfaces funcionales y expresiones lambda

```
Comparator<Cliente> comp;
comp = (a,b) -> {return a.nombre.compareTo(b.nombre);};
```

Cuando el cuerpo de la función es una sola sentencia, también podríamos prescindir de la orden return. En general, entre las llaves podemos escribir tantas sentencias como sean necesarias. También podemos prescindir de la variable comp y colocar la expresión lambda directamente en la lista de parámetros de sort () ,

```
Collections.sort(lista,
                (a,b) -> {return a.nombre.compareTo(b.nombre);}
                ) ;
```

Java sabe que el segundo parámetro de sort() es un objeto Comparator e interpreta que el código que le pasamos corresponde al método compare().

Sintaxis general expresión lambda

La sintaxis general de una expresión lambda consiste en,

(tipo1 param1, tipo2 param2, ...) -> {Cuerpo de La expresión Lambda};

Es decir,

- Una lista, entre paréntesis, de parámetros formales separados por comas. Los tipos de los parámetros se pueden omitir si Java los puede inferir del lado izquierdo en una operación de asignación. Cuando hay un solo parámetro de entrada, también se pueden omitir los paréntesis.
- Una flecha -> (guión alto - seguido de >).
- El cuerpo de la función entre llaves, que puede consistir en una sentencia o un bloque de sentencias. Si es una única sentencia y no devuelve ningún valor, las llaves se pueden omitir. Si es una única sentencia y devuelve un valor, la orden return se puede omitir, ya que Java lo devuelve automáticamente.

Con una expresión lambda, realmente estamos creando un método anónimo.

Repaso Interfaces de la API de Comparación

Las **interfaces de la API** de Java que facilitan facilitan las operaciones de comparación son:

- Comparable se usa cuando el orden natural de los objetos es lógico (ej., orden por edad).
- Comparator se usa cuando necesitas múltiples formas de ordenar sin tocar la clase original (ej., orden por nombre).

Característica	`Comparable<T>`	`Comparator<T>`
Ubicación	En la propia clase	Externa a la clase
Método	`compareTo(T o)`	`compare(T o1, T o2)`
Orden definido	**Natural** (un único criterio)	**Personalizado** (pueden ser múltiples)
Modificabilidad	Se debe modificar la clase	Se puede definir en cualquier parte

Repaso Interfaces de la API de Comparación

Ejemplo Comparación no natural por nombre de la clase persona:

```
List<Persona> lista = Arrays.asList(new Persona("Carlos", 25),
        new Persona("Ana", 25), new Persona("Beatriz", 22) );
// Comparator para ordenar por nombre
Comparator<Persona> ordenPorNombre = (p1, p2) ->
p1.getNombre().compareTo(p2.getNombre());
Collections.sort(lista, ordenPorNombre);
System.out.println(lista); // [Ana (25), Beatriz (22), Carlos (25)]
```

Además, a partir de Java8 se introdujeron otros métodos de Comparator:

- Comparator.comparing(criterio) que simplifica la lógica de la comparación.

```
// Ordenar por nombre con Comparator.comparing()
lista.sort(Comparator.comparing(Persona::getNombre));
System.out.println(lista); // [Ana (25), Beatriz (22), Carlos (25)]
```

Repaso Interfaces de la API de Comparación

- Comparator.comparing(criterio).reversed() nos facilita la ordenación descendente.

```
// Orden descendente por edad
 lista.sort(Comparator.comparing(Persona::getEdad).reversed());
 System.out.println(lista); // [Carlos (25), Ana (25), Beatriz (22)]
```

- Comparator.comparing(criterio1).thenComparing(criterio2) nos permite ordenar por criterio1, y a igualdad de criterio1 por criterio2.

```
// Ordenar por edad, y si hay empate, por nombre
 lista.sort(Comparator.comparing(Persona::getEdad)
                          .thenComparing(Persona::getNombre));
 System.out.println(lista); //[Beatriz (20), Ana (25), Carlos (25)]
```

Tipos Interfaces funcionales de la API

Las **expresiones lambda** junto con las **interfaces funcionales de la API** de Java facilitan el desarrollo de código conciso y reutilizable.
Las interfaces funcionales son aquellas que tienen un único método abstracto y se pueden usar con expresiones lambda.

Las principales interfaces funcionales son:

Interfaz	Método principal	Parámetros	Devuelve	Uso común
`Predicate<T>`	`boolean test(T t)`	1	`boolean`	Filtrar datos
`Function<T, R>`	`R apply(T t)`	1	`R`	Transformar datos
`Consumer<T>`	`void accept(T t)`	1	`void`	Realizar acciones sin devolver resultado
`Supplier<T>`	`T get()`	0	`T`	Proveer valores

Tipos Interfaces funcionales de la API

Predicate<T>: se usa para comprobar una condición en un valor del tipo genérico T. Se utiliza como argumento de métodos allMatch, anyMatch o noneMatch, etc.

```
System.out.println("¿Hay alguna cadena vacía?");
System.out.println(palabras.anyMatch(p -> p.isEmpty()));
```

Su método abstracto es: ***boolean test(T valor)***
que devuelve true si la condición se verifica para valor y false en caso contrario.
Por ejemplo, para comprobar si un Integer es positivo, podemos definir el predicado: *Predicate<Integer> esPositivo = x -> x > 0;*
Ejemplo: esPositivo.test(5) devolverá true.
El método test(), es el único abstracto de la interfaz Predicate, pero junto a él hay tres métodos por defecto (negate, and y or).
Predicate<T> negate() devuelve un nuevo predicado que es la negación del predicado invocante.

Interfaces funcionales de la API

En nuestro caso, esPositivo.negate() nos devuelve un predicado que comprueba si un Integer es no positivo (menor o igual que 0).
Predicate<Integer> esNoPositivo = esPositivo.negate();
Ejemplo: esPositivo.negate().test(5) que dará el mismo resultado, false.

El segundo método por defecto es ***Predicate<T> and(Predicate<? super T> otro)*** que devuelve un predicado que es la conjunción del predicado invocante y del que se pasa como parámetro, de modo que test() devolverá true cuando los dos predicados sean ciertos para el valor que se le pase como parámetro. El tipo genérico de otro debe ser igual o una superclase de T.
Ejemplo: *Predicate<Integer> esPar = n -> n % 2 == 0* que comprueba si un entero es par.
Si queremos saber si el entero 6 es par y positivo:
Predicate<Integer> esPositivoYPar = esPar.and(esPositivo);

Interfaces funcionales de la API

Ejemplo: esPositivoYPar.test(6) devolverá true. Es lo mismo que:

esPar.and(esPositivo).test(6)

El tercer método es la disyunción: ***Predicate<T> or(Predicate<? Super T> otro)*** que devuelve un predicado cuyo método test() devolverá true cuando, al menos, uno de los dos predicados (invocante y otro) sean ciertos.

Predicate<Integer> esPosotivoOPar = esPar.or(esPositivo);

esPositivoOPar.test(6) //true, par y positivo

esPositivoOPar.test(-3) //false, ni par ni positivo

Function<T, V>: a partir de un argumento de tipo T, devuelve un resultado de tipo V. Se usa mucho con el método map(). Su única función abstracta es, ***V apply(T x)***

Ejemplo: Definir una función que calcula el cuadrado de un valor real

Function<Double, Double> cuadrado= x -> x*x;

System.out.println(cuadrado.apply(2.0)); //mostrará 4.0 por consola

Interfaces funcionales de la API

También podemos asignarle directamente la referencia a un método ya existente, si se da el caso. Su sintaxis es ***Clase::función***. Ejemplo:

```
System.out.println("Cuadrados de los 10 primeros enteros");
Arrays.stream(enteros).map(e -> e * e)
      .forEach(System.out::println);
```

Consumer<T>: sirve para realizar una tarea a partir de un argumento de entrada. Su método abstracto ***void accept(T t)*** recibe un valor de una clase T, con el que hace operaciones sin devolver nada.

Ejemplo: Mostrar por pantalla un saludo a distintos clientes

```
Consumer<Cliente> saludoClie =e-> System.out.println("Hola, "+ c.nombre);
```

El método accept(), recibirá como argumento un objeto Cliente y, a partir de él creará un mensaje de saludo: Cliente clie=new Cliente("123", "Jorge", 20); saludoClie.accept(clie); //se mostrará "Hola, Jorge"

Interfaces funcionales de la API

Suplier<T>: sirve para proporcionar un valor sin argumentos de entrada. Su método abstracto es ***T get()*** que hace operaciones devolviendo un objeto de tipo T.
Ejemplo: Generar y mostrar por pantalla un saludo de bienvenida
Supplier<String> mensajeBienvenida = () -> "¡Bienvenido a Java Lambda!";
El método get(), proveerá un mensaje:
System.out.println(mensajeBienvenida.get());

Referencias a métodos

A partir de la versión 8 de Java, es posible trabajar con referencias a métodos ya definidos en alguna clase.

Las referencias a métodos se escriben: *nombre de la clase :: nombre del método* (sin paréntesis ni lista de argumentos) cuando este es estático. Si es no estático, en vez del nombre de la clase pondremos una referencia a un objeto de la clase donde está definido el método.

También se pueden usar referencias a métodos constructores. Ej: Cliente::new.

```
class Saludo {
   String nombre;
   Saludo(String nombre) {
      this.nombre = nombre;
   }
   public String toString() {
      return "Hola, " + nombre;
   }
}
```

```
Function<String, Saludo> construyeSaludo = Saludo::new;
Saludo s = construyeSaludo.apply("Julia");
System.out.println(s); //¡Hola Julia!
```

Ejemplo Interfaces funcionales de la API

```
import java.util.function.*;
public class EjemploFunciones {
    public static void main(String[] args) {
        Predicate<Integer> esPar = n -> n % 2 == 0;
        Function<Integer, String> aTexto = n -> "Número: " + n;
        Consumer<String> imprimir = System.out::println;
        Supplier<Double> generarAleatorio = () -> Math.random() * 100;

        int numero = 10;
        if (esPar.test(numero)) {
            String mensaje = aTexto.apply(numero);
            imprimir.accept(mensaje);  // "Número: 10"
        }
       System.out.println("Número aleatorio: " + generarAleatorio.get());
    }
}
```

Ejercicios Interfaces funcionales de la API

Ejercicio 1: Verificar si un número es positivo.

Ejercicio 2: Verificar si una cadena tiene más de 5 caracteres.

Ejercicio 3: Obtener la longitud de una cadena.

Ejercicio 4: Registrar un mensaje en una lista (simulación de logs).
ACLARACIÓN: Nos inventamos los mensajes.

Ejercicio 5: Proveer la fecha y hora actual y mostrarla.

Introducción Stream

Las colecciones aportan versatilidad y potencia al procesamiento y la manipulación de datos complejos. Sin embargo, para recorrerlas disponemos de los iteradores, cuyo manejo puede resultar incómodo.

A partir de Java 8, se ha introducido una serie de herramientas que permiten efectuar operaciones globales con los elementos de una colección completa, sin necesidad de recorrerlas nodo a nodo, aprovechando el procesamiento paralelo de una forma transparente al programador. En un programa, a menudo, hay trozos de código (tareas a realizar) que se pueden ir ejecutando sin esperar a que hayan terminado otras que vienen escritas antes. Ejecuciones simultáneas de dos partes del código se conocen como ejecución paralela.

También pueden encadenarse, una a continuación de otra, formando tuberías, para dar un resultado final, sin necesidad de acceder a resultados intermedios.

Interfaz Stream

Los objetos de las clases que implementan la interfaz Stream son sucesiones de objetos sobre los que se puede realizar una serie de operaciones que pueden ir encadenadas hasta dar un resultado final.
Dichas operaciones pueden ser de dos tipos:
Intermedias: dan como resultado un nuevo Stream al que se le pueden seguir aplicando nuevas operaciones.
Terminales: dan un resultado final, numérico o de otro tipo, pero no un Stream.

A partir de una colección o una tabla, o bien explícitamente, un Stream al que se aplican operaciones intermedias encadenadas (es lo que se conoce como una tubería o pipeline), obteniendo un resultado final por medio de una operación terminal.
La ventaja de crear el Stream es que dispone de muchas más operaciones para procesar los datos que las colecciones o las tablas.

Creación de los Streams

Los Stream son objetos que implementan la interfaz Stream. Por tanto, la clase Stream no existe y los objetos Stream no se pueden crear con un constructor, sino llamando a funciones implementadas para ello.

Formas de crear un Stream

- A partir de una colección: llamando al método stream(), definido en las clases de tipo Collection,
  Stream<T> nombreStream = nombreColeccion.stream();
- A partir de una tabla: llamando al método of() , de la clase Stream, con la tabla como parámetro
  Stream<T> nombreStream = Stream.of(T[] tabla);
- A partir de una tabla: usando el método stream(), de la clase Arrays, con la tabla como parámetro
  Stream<T> nombreStream = Arrays.stream(T[] tabla);

Creación de los Streams

- Inicializándolo directamente: también con el método of () de Stream, pero pasándole como lista de parámetros los valores que lo inicializan
  Stream<T> nombreStream = Stream.of(T val1, T val2, ...)

Ejemplo:

```
List<String> lista = new ArrayList<>();
lista.add("dado");
lista.add("arte");
lista.add("bola");
Stream<String> streamCad = lista.stream();
```

El Stream *streamCad* contiene una copia de todos los datos de la lista, no una refe rencia a los originales. Por tanto, los cambios que se hagan en el Stream no se van a reflejar en la lista original, que permanecerá intacta.

Utilidades de los Streams

El uso de Streams en Java permite procesar datos de manera más eficiente y legible, sin necesidad de recorrer manualmente las colecciones con bucles. Son ideales para realizar operaciones como filtrar, transformar, reducir, ordenar, contar y agrupar datos de forma declarativa y moderna.

Las operaciones sobre Stream son agregadas y se inspiran en las operaciones globales de las colecciones, ya que operan sobre la totalidad del Stream. Muchas de ellas hacen uso de interfaces funcionales de la API, se han diseñado para trabajar con expresiones lambda.

Resumen utilidades Streams

Principales utilidades de los Streams

Filtrado de datos (filter)

Permiten seleccionar elementos que cumplan una condición específica.

Ejemplo: Obtener números pares de una lista (mostrándolos por consola).

```
List<Integer> numeros = Arrays.asList(1, 2, 3, 4, 5, 6);
//solo mostrándolos por consola
numeros.stream()
        .filter(n -> n % 2 == 0)
        .forEach(System.out::println);

//creando una lista que luego mostramos
List<Integer> pares = numeros.stream()
        .filter(n -> n % 2 == 0)
        .collect(Collectors.toList());
System.out.println(pares); // [2, 4, 6]
```

Resumen utilidades Streams

Transformación de datos (map)

Sirven para modificar los elementos de una colección.

Ejemplo: Convertir una lista de nombres a mayúsculas.

```
List<String> nombres = Arrays.asList("juan", "maria", "ana");
//solo mostrándolos por consola
nombres.stream()
        .map(String::toUpperCase)
        .forEach(System.out::println);

//creando una lista que luego mostramos
List<String> mayusculas = nombres.stream()
        .map(String::toUpperCase)
        .collect(Collectors.toList());
System.out.println(mayusculas); // [JUAN, MARIA, ANA]
```

Resumen utilidades Streams

Reducción de datos (reduce)

Permiten obtener un único resultado a partir de una colección.

Ejemplo: Sumar todos los números de una lista.

```
List<Integer> numeros = Arrays.asList(1, 2, 3, 4, 5);
int suma = numeros.stream()
        .reduce(0, Integer::sum);
System.out.println(suma); // 15
```

Contar elementos (count)

Podemos contar cuántos elementos cumplen una condición.

Ejemplo: Contar cuántos nombres tienen más de 4 caracteres.

```
List<String> nombres = Arrays.asList("Ana", "Pedro", "Luis", "Beatriz");
long cantidad = nombres.stream()
        .filter(n -> n.length() > 4)
        .count();
System.out.println(cantidad); // 2 (Pedro y Beatriz)
```

Resumen utilidades Streams

Ordenación de datos (sorted)

Se pueden ordenar elementos de la colección de forma ascendente o descendente.

Ejemplo: Ordenar una lista de números de mayor a menor.

```
List<Integer> numeros = Arrays.asList(5, 3, 8, 1);
List<Integer> ordenados = numeros.stream()
        .sorted(Comparator.reverseOrder())
        .collect(Collectors.toList());
System.out.println(ordenados); // [8, 5, 3, 1]
```

Eliminación de duplicados (distinct)

Permiten eliminar valores repetidos en una colección.

Ejemplo: Obtener una lista sin duplicados.

```
List<Integer> numeros = Arrays.asList(1, 2, 2, 3, 4, 4, 5);
List<Integer> sinDuplicados = numeros.stream()
        .distinct()
        .collect(Collectors.toList());
System.out.println(sinDuplicados); // [1, 2, 3, 4, 5]
```

Resumen utilidades Streams

Agrupación de datos (groupingBy)

Podemos agrupar elementos de una colección según un criterio.

Ejemplo: Agrupar palabras por su longitud.

```
import java.util.*;
import java.util.stream.Collectors;

public class Ejemplo {
    public static void main(String[] args) {
        List<String> palabras = Arrays.asList("casa", "coche", "sol", "mar", "programa");

        Map<Integer, List<String>> agrupadas = palabras.stream()
                .collect(Collectors.groupingBy(String::length));

        System.out.println(agrupadas); //{3=[sol, mar], 4=[casa, coche], 12=[programa]}
    }
}
```

Detalle Interfaz Stream

Filtrar los elementos de un Stream

Stream<T> filter(Predicate<? Super T> pred)

Invocado desde el Stream original, se le pasa un predicado que se aplicará a todos los elementos del Stream. Solo aquellos que devuelvan true formarán parte del nuevo Stream devuelto por el método.

Ejemplo: Obtener, a partir de streamCad, un nuevo Stream con los elementos que empiezan por "a"

```
Stream<String> streamA = streamCad.filter(s -> s.startsWith("a"));
```

Siempre que queramos que se ejecute una determinada acción «para cada» elemento de un Stream, usaremos el método forEach

void forEach(Consumer<? Super T> tarea)

```
streamA.forEach(s -> System.out.println(s)); //se mostrará "arte" y "asa"
```

o usando referencia a métodos

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

Detalle Interfaz Stream

Para mostrar por pantalla los elementos que empiezan por «a», podríamos haber encadenado las dos operaciones para formar lo que se llama una tubería

```
lista.stream().filter(s -> s.startsWith("a")).forEach(System.out::println);
```

Las tuberías se suelen escribir una operación por línea:

```
lista.stream ()
        .filter(s -> s.startsWith("a"))
        .forEach(System.out::println);
```

Podemos obtener un Stream, a partir de una tabla, con el método estático of():

```
Cliente[] tClie= {new Cliente("111", "Marta", 20),
                  new Cliente("115", "Jorge", 21),
                  new Cliente("112", "Carlos", 18),
                  new Cliente("211", "Ana" , 19)};
```

y, a partir de ella, obtendremos un Stream. por cualquiera de los métodos,

```
Stream<Cliente> streamClie = Stream.of(tClie);
Stream<Cliente> streamClie = Arrays.stream(tClie); //otra manera
```

Detalle Interfaz Stream

Como los Stream no son reutilizables, lo habitual es escribir las tuberías completas, incluyendo la lista o la tabla iniciales cada vez.

Ordenar un Stream

Stream<T> sorted()

devuelve un nuevo Stream con los elementos ordenados según su orden natural.

Ejemplo:

```
Arrays.stream(tClie)
        . sorted()
        .forEach(System.out::println);
```

Podemos ordenar por un criterio alternativo, definiendo el comparador:

```
Comparator<Cliente> comp = (x, y) -> x .nombre.compareTo(y.nombre);
```

Ejemplo: Ordenar los clientes por nombre en vez de por DNI

```
Arrays.stream(tClie)
        .sorted((x, y) -> x.nombre.compareTo(y.nombre))
        .forEach(System.out::println);
```

Detalle Interfaz Stream

Ejemplo: Obtener un Stream con los DNI de los clientes en el orden del original, transforma un objeto de tipo Cliente en su dni, de tipo String.

```
Arrays.stream(tClie)
        .map(c -> c.dni)
        .forEach(System.out::println);
```

El método terminal ***long count()*** devuelve el número de elementos

Ejemplo:

```
long n = Arrays.stream(tClie)
        .sorted((c1,c2) -> c1.nombre.compareTo(c2.nombre))
        .filter(c ->e.edad < 20)
        .count();
```

devuelve 2, el número de clientes de menos de 20 años de edad.

Detalle Interfaz Stream

Ejemplo: crear ahora un Stream de enteros inicializándolo de forma explícita:

```
Stream<Integer> streamEnteros = Stream.of(4, 3, 7, 1, 0, 8, 9, 3, 5, 4, 2, 1, 4, 6, 8, 1, 0, 2, 3);
```

Podemos ***eliminar los elementos repetidos*** (obteniendo un nuevo Stream):

Stream<T> distinct()

```
Stream.of(4, 3, 7, 1, 0, 8, 9, 3, 5, 4, 2, 1, 4, 6, 8, 1, 0, 2, 3)
    .distinct()
    .forEach(x -> System.out.print(x +" ")); //mostrará por pantalla  4 3 7 1 0 8 9 5 2 6
```

Podemos ***concatenar dos Stream:***

static Stream<T> concat(Stream<? extends T> prim, Stream<? extends T> seg)

devuelve un nuevo Stream con los elementos del 2º a continuación de los del 1º:

```
Stream<Integer> streamNuevo = Stream.of(-1, -6, -3, -3);
Stream.concat(streamEnteros, streamNuevo)
    .distinct()
    .forEach(x -> System.out.print( x + " ")); //mostrará 4 3 7 1 0 8 9 5 2 6 -1 -6 -3
```

Detalle Interfaz Stream

Podemos **crear una tabla con los elementos del Stream**:

0bject[] toArray()

Ejemplo: Crear una tabla con los pares sin repetir de un stream

```
0bject[] tObject = Stream.of(-1, -6, -3, -3)
        .distinct()
        .filter(x -> x % 2 == 0)
        .toArray();
```

Podemos ***agrupar elementos de un Stream en una colección:***

```
List<Integer> listaimpares = Stream.of(2, 5, 1, 4, -6, -3, -3)
        .filter(x -> x % 2 != 0)
        .collect(Collectors.toList());
Set<Integer> conjuntoimpares = Stream.of(5, 1, 2, 6, 3, 9, 4, 1, 7, 3, 5)
        .filter(x -> x % 2 != 0)
        //.collect(Collectors.toSet())
        .collect(Collectors.toCollection(TreeSet::new));  //mantiene el orden
```

Detalle Interfaz Stream

Podemos **crear un mapa a partir de un Stream**:

Ejemplo: Crear un mapa de los DNI (claves) sobre los nombres (valores) de los clientes, usaremos Collectors.toMap()

```
Map<String, String> mapaClientes = Stream.of(tClie)
        .collect(Collectors.toMap(e -> e.dni, e -> e.nombre));
// Obtenemos el mapa {111=Marta, 211=Ana, 112=Carlos, 115=Jorge}
```

Con **Collectors.averagingint()** podemos calcular el promedio de las edades,

```
double edadMedia = Stream.of(tClie)
        .collect(Collectors.averagingInt(c -> c.edad));
```

o una estadística general de las edades, **IntSummaryStatistics** es una clase capaz de calcular diversos parámetros estadísticos (count, sum, max, average,...):

```
IntSummaryStatisties sumarioEdad =
        streamClie.collect(Collectors.summarizingInt(c -> c.edad));
System.out.println(sumarioEdad);
```

Ejemplo uso Interfaz Stream

Ejemplo sobre Gestión de Empleados

Necesitamos almacenar los datos de nuestra plantilla de empleados: nombre, edad, salario y departamento. Y creamos algunos empleados de ejemplo:

Ana (30 años, TI, 2500.0€)
Carlos (45 años, Marketing, 4000.0€)
Beatriz (22 años, TI, 1800.0€)
David (38 años, Ventas, 3200.0€)

📌 Requerimientos

1. Filtrar empleados con salario superior a 3000€ (Predicate).
2. Transformar el nombre de los empleados a mayúsculas (Function).
3. Imprimir los empleados con un icono previo "📌 " (Consumer).
4. Generar un empleado aleatorio (Supplier).
5. Ordenar empleados por diferentes criterios (Comparable, Comparator). El orden natural lo establecemos por salario. Y el orden alternativo por nombre.