

El framework de Colecciones & Streams

Las interfaces Centrales

`Collection`

- `Set` y `SortedSet`
- `List`
- `Queue`

`Map` y `SortedMap`

Las interfaces Secundarias

`Iterator` y `ListIterator`

Las implementaciones convencionales

`HashSet`, `HashMap`, `ArrayList`, `LinkedList`, `TreeSet`,
`TreeMap`, *etc.*

Interfaces Funcionales, funciones lambda

Streams

Colecciones

Composición

Una colección es un objeto que representa a un grupo de objetos. Se usa para almacenar, recuperar y manipular un conjunto de datos.

Un *framework* de colecciones permite representar y manipular colecciones de una manera unificada, independientemente de los detalles de implementación. El *frameworks* de colecciones de JAVA cuentan con:

- Interfaces: son tipos de datos abstractos que representan colecciones y que permiten manejarlas en forma independiente de su implementación. Forman jerarquías de herencia.
- Implementaciones: son implementaciones concretas de las interfaces. Son estructuras de datos.
- Algoritmos: son métodos de clase que realizan operaciones útiles (búsquedas y ordenamientos) sobre objetos que implementan alguna de las interfaces de colecciones.

Java incluye en su librería, implementaciones de las estructuras de datos más comunes. Esta parte de la librería es conocida como API de colecciones.

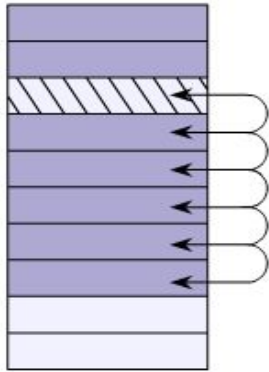
Colecciones

Tecnologías de almacenamiento

Existen cuatro tecnologías de almacenamiento básicas disponibles para almacenar objetos: arreglo, lista enganchada, árbol y tabla de hash.

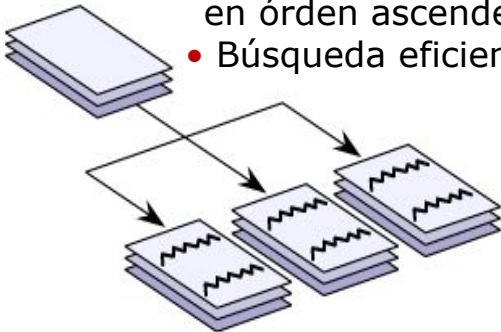
Arreglo

El acceso es muy eficiente.
Es ineficiente cuando se agrega/elimina un elemento.
Los elementos se pueden ordenar.



Arbol

- Almacenamiento de valores en orden ascendente.
- Búsqueda eficiente.



Lista Enlazada

Acceso muy ineficiente, hay que recorrer la lista.
Eficiente cuando se agrega/elimina un elemento.
Los elementos se pueden ordenar.

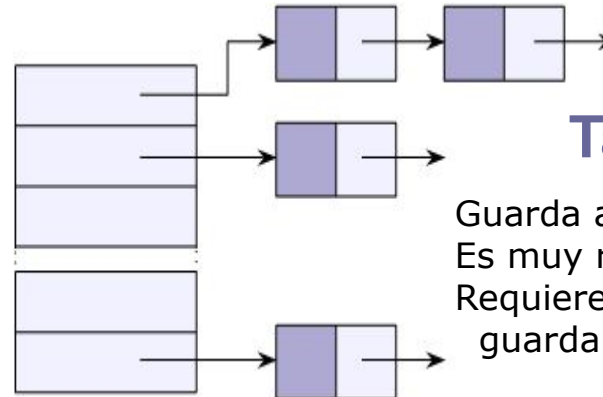
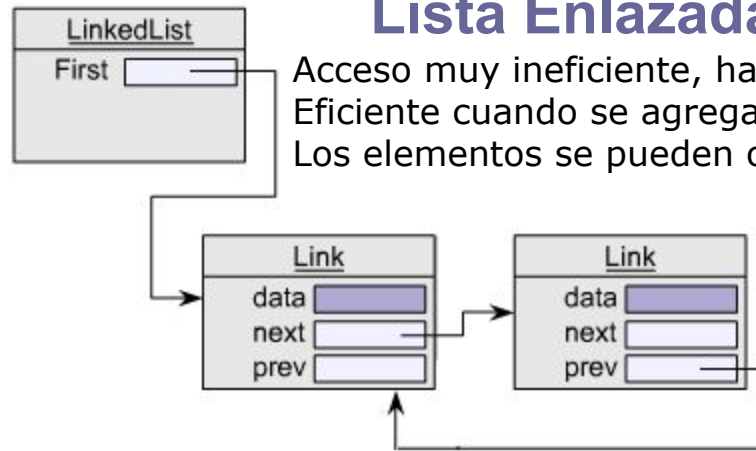


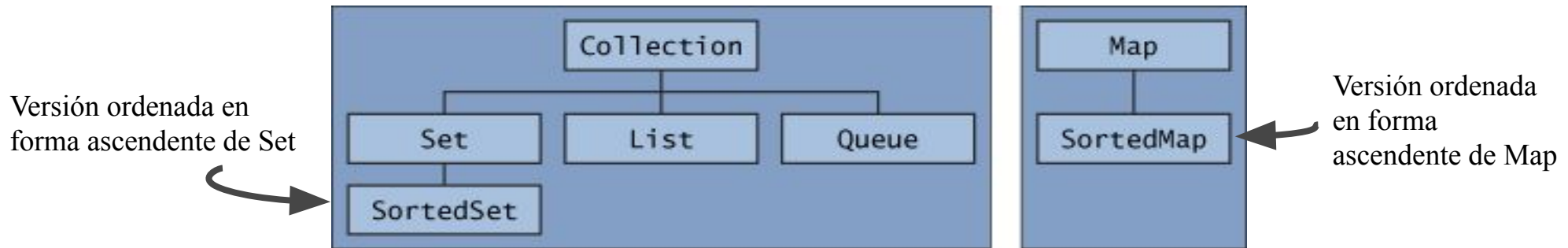
Tabla de hash

Guarda asociaciones (clave, valor).
Es muy rápido, accede por clave
Requiere memoria adicional para guardar las claves (tabla).

Colecciones

Jerarquías de Interfaces

Encapsulan distintos tipos de colecciones y son el fundamento del framework de colecciones.



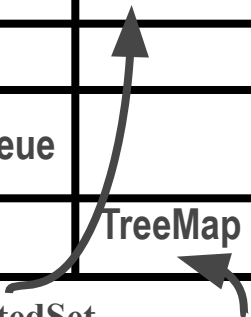
- **Collection:** representa un conjunto de objetos, llamados elementos. Es la raíz de la jerarquía de colecciones. La plataforma Java no provee una implementación directa para la interface Collection, pero si para sus subinterfaces Set, List y Queue. Permite elementos duplicados.
- **Set:** extiende Collection y no permite elementos duplicados. Modela el concepto de conjunto matemático.
- **List:** extiende Collection, es una colección que permite elementos duplicados (también llamada secuencia) y que incorpora acceso posicional mediante índices.
- **Queue:** extiende Collection proveyendo operaciones adicionales para inserción, extracción e inspección de elementos. Típicamente los elementos de una Queue están ordenados usando una estrategia FIFO (First In First Out). Se incorporó a partir de la versión jse 5.0.
- **Map:** permite tener pares de objetos que “mapean” claves con valores. No permite claves duplicadas. Cada clave mapea a lo sumo con un valor.

Colecciones

Implementaciones de las interfaces

La tabla muestra las implementaciones de propósito general que vienen con la plataforma java, las cuales siguen una convención de nombre, combinando la estructura de datos subyacente con la interface del framework:

Interfaces	Tecnologías de almacenamiento				
	Tabla de Hashing	Arreglo de tamaño variable	Árbol	Lista Encadenada	Tabla de Hashing + Lista Encadenada
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Queue		ArrayBlockingQueue		LinkedBlockingQueue	
Map	HashMap		TreeMap		LinkedHashMap



Todas las implementaciones de propósito general:

- Tienen implementado el método `toString()`, el cual retorna a la colección de una manera legible, con cada uno de los elementos separados por coma.
- Tienen por convención al menos 2 constructores: el nulo y otro con un argumento `Collection`:
`TreeSet()` y `TreeSet(Collection c)`
`LinkedHashSet()` y `LinkedHashSet(Collection c)`

Colecciones

La interface Collection

La interface **Collection** representa un conjunto de objetos de cualquier tipo. Esta interface se usa cuando se necesita trabajar con grupos de elementos de una manera muy genérica.

```
public interface Collection<E> extends Iterable<E> {  
  
    // operaciones básicas  
    int size();  
    boolean isEmpty();  
    boolean contains(Object element);  
    boolean add(E element);  
    boolean remove(Object element);  
    Iterator iterator();  
  
    // operaciones en "masa"  
    boolean containsAll(Collection<E> c);  
    boolean addAll(Collection<E> c);  
    boolean removeAll(Collection<E> c);  
    boolean retainAll(Collection<E> c);  
    void clear();  
  
    // operaciones de Arreglos  
    Object[] toArray();  
}
```

Las clases que implementan esta interface pueden iterarse con un objeto Iterator.

Este método retorna un iterador que permite recorrer la colección desde el comienzo hasta el final

Convierte la colección a un arreglo


Colecciones

La interface List

Un objeto **List** es una secuencia de elementos donde puede haber duplicados. Además de los métodos heredados de **Collection**, define métodos para recuperar y modificar valores en la lista por posición como: **E get(int index)**; **E set(int index, E element)**; **boolean add(E element)**; **void add(int index, E element)**; **E remove(int index)**.

La plataforma java provee 2 implementaciones de List, que son **ArrayList** y **LinkedList**.

```
import java.util.*;
public class DemoIterador{
    public static void main(String[] args){
        List<Integer> lista = new ArrayList<Integer>();
        lista.add(1);
        lista.add(new Integer(2));
        lista.add(190);
        lista.add(90);
        lista.add(7);
        lista.remove(new Integer(2));
        System.out.print(lista.toString());
    }
}
```



[1, 190, 90, 7]

Se podría reemplazar por **new LinkedList<Integer>()** y todo sigue funcionando

boxing/unboxing

Convierte automáticamente datos de tipo primitivo `int` a objetos de la clase `Integer`. Mejora a partir JSE 5.0

Colecciones

La interface Set

Un objeto Set es una colección que no contiene elementos duplicados. Tiene exactamente los mismos métodos que la interface **Collection**, pero agrega la restricción de no mantener duplicados.

La plataforma java provee implementaciones de propósito general para Set. Por ejemplo **HashSet** (mejor performance, almacena los datos en una tabla de hash) y **TreeSet** (más lento pero ordenados).

```
public class DemoIterador{
    public static void main(String[] args) {
        Set<String> instrumentos= new HashSet<String>();
        instrumentos.add(" Piano");
        instrumentos.add(" Saxo");
        instrumentos.add(" Violin");
        instrumentos.add(" Flauta");
        instrumentos.add(" Flauta");
        System.out.println(instrumentos.toString());
    }
}
```

La interface Set es útil para crear colecciones sin duplicados desde una colección `c` con duplicados.

salida

[Violin, Piano, Saxo, Flauta]

```
Set<String> sinDup=new TreeSet<String>(c);
```

Implementa **SortedSet**

Cambiando únicamente la instanciación por un objeto **TreeSet()**, obtenemos una colección ordenada:

salida

```
Set<String> instrumentos= new TreeSet<String>();
```

[Flauta, Piano, Saxo, Violin]

En este caso el compilador chequea que los objetos que se insertan (add()) sean **Comparables!!**

Colecciones

La interface Map

Un objeto Map mapea claves con valores. No puede contener claves duplicadas y cada clave mapea con a lo sumo un valor.

La plataforma java provee implementaciones de propósito general para **Map**. Por ejemplo, las clases **HashMap** y **TreeMap** con un comportamiento y performance análogo a las implementaciones mencionadas para la interface **Set**.

```
public interface Map <K,V> {  
    // Operaciones Básicas  
    V put(K clave, V valor);  
    V get(K clave);  
    V remove(K clave);  
    boolean containsKey(K clave);  
    boolean containsValue(V valor);  
    int size();  
    boolean isEmpty();  
    // Operaciones en "masa"  
    void putAll(Map <K,V> t);  
    void clear();  
    // Vistas  
    Set<K> keySet();  
    Collection<V> values();  
    ...  
}
```

```
public class DemoMap {  
    public static void main(String[] args) {  
        Map<String, Integer> numeros=new HashMap<String,Integer>();  
        numeros.put("uno", new Integer(1));  
        numeros.put("dos", new Integer(2));  
        numeros.put("tres", new Integer(3));  
        System.out.println(numeros.toString());  
    }  
}
```

{tres=3, uno=1, dos=2}

salida

Implementa
SortedSet0

Cambiando únicamente la instanciación por un objeto **TreeMap()**, obtenemos una colección ordenada:

salida

```
Map<String, Integer> numeros=new TreeMap<String, Integer>();
```

{dos=2, tres=3, uno=1}

En este caso el compilador chequea que los objetos que se insertan (put()) sean **Comparables!!**

Colecciones

La interface Queue

Un objeto **Queue** es una colección diseñada para mantener elementos que esperan por procesamiento. Además de las operaciones de **Collection**, provee operaciones para insertar, eliminar e inspeccionar elementos. No permite elementos nulos.

La plataforma java provee una implementación de **Queue**: **PriorityQueue** (es una cola con prioridades) en el paquete `java.util` y varias en el paquete `java.util.concurrent` como **DelayQueue** y **BlockingQueue** que implementan diferentes tipos de colas, ordenadas o no, de tamaño limitado o ilimitado, etc.

```
public interface Queue<E> extends Collection<E>{
```

```
// Búsqueda
```

```
E peek();
```

```
boolean offer(E e);
```

```
E poll();
```

```
}
```

Recupera, pero no elimina la cabeza de la cola.

Inserta el elemento en la cola si es posible

Recupera y elimina la cabeza de la cola.

```
public class DemoQueue{
    public static void main(String[] args) {
        PriorityQueue<String> pQueue
```

```
        PriorityQueue<String>();
        pQueue.offer("Buenos Aires");
        pQueue.offer("Montevideo");
        pQueue.offer("La Paz");
        pQueue.offer("Santigao");
        System.out.println(pQueue.peek());
        System.out.println(pQueue.poll());
        System.out.println(pQueue.peek());
    }
}
```

PriorityQueue chequea que los objetos que se insertan sean **Comparables!!**

= new

Buenos Aires
Buenos Aires
La Paz

salida

Colecciones

Mecanismos para recorrerlas

Hay dos maneras de recorrer una colección:

1) Usando la construcción: **for-each**

```
ArrayList<Integer> lista= new ArrayList<Integer>();  
lista.add("Elemento 1");  
lista.add("Elemento 2");  
for (int elemento: lista)  
    System.out.println(elemento);  
}
```

2) Usando la interface **Iterator/ListIterator**

Un objeto **Iterator**, permite recorrer una colección y eliminar elementos selectivamente durante la recorrida. Siempre es posible obtener un **iterador** para una colección, dado que la interface **Collection** extiende la interface **iterable**.

```
package java.util;  
public interface Iterator<E> {  
    boolean hasNext();  
    E next();  
    default void remove() {  
        throw new UnsupportedOperationException("remove");  
    }  
}
```

```
package java.util;  
public interface ListIterator<E> extends Iterator<E>{  
    boolean hasPrevious();  
    E previous();  
}
```

Colecciones

Mecanismos para recorrerlas

Iterando sobre objetos Collection.

```
import java.util.*;
public class IteradoresLista {

    public static void main(String[] args) {
        ArrayList<Character> lista = new ArrayList<Character>();
        lista.add('1'); lista.add('2'); lista.add('3');
        lista.add('4'); lista.add('5'); lista.add('6');
        lista.add('7'); lista.add('8');
        char nro;
        System.out.println("Lista original: "+lista);
        Iterator<Character> it1 = lista.iterator();
        while (it1.hasNext()) {
            nro = it1.next();
            if (nro%2==0)
                it1.remove();
        }
        System.out.println("Lista procesada: "+lista);
        System.out.println("Lista procesada con for: "+lista);
        for (Character numero : lista)
            System.out.print(numero);
        System.out.print("Iteramos desde atrás: ");
        ListIterator<Character> it = lista.listIterator(lista.size());
        while (it.hasPrevious())
            System.out.print(it.previous());
    }
}
```

Si no se define al iterador de tipo <Character>, hay que castear cuando se recupera el objeto, dado que es de tipo Object y no lo puede asignar a un char.

Lista original: [1, 2, 3, 4, 5, 6, 7, 8]
Lista procesada: [1, 3, 5, 7]
Lista procesada con for: 1357
Iteramos desde atrás: 7531

Colecciones

Mecanismos para recorrerlas

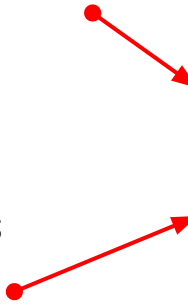
Iterando sobre objetos Map

Los objetos **Map** proveen vistas como objetos **Collection**, a partir de las cuales se puede tratar a un objeto **Map** como una colección usando los métodos **values()** y **keys()**

```
import java.util.*;

public class DemoIteradores {
    public static void main(String[] args) {
        Map<String, Alumno> tablaAlu = new HashMap<String, Alumno>();
        tablaAlu.put("7892/8", new Alumno("7892/8", "Gomez", "Juana"));
        tablaAlu.put("3321/0", new Alumno("3321/0", "Perez", "Sol"));
        tablaAlu.put("3421/7", new Alumno("3421/7", "Rusciti", "Maria"));
        tablaAlu.put("7892/2", new Alumno("7892/2", "Gomez", "Juana"));
        Collection<Alumno> listAlu = tablaAlu.values();
        for (Alumno unAlu : listAlu) {
            unAlu.setApe(unAlu.getApe().toUpperCase());
            System.out.println("Alumno:" + unAlu);
        }
        Set<String> keys = tablaAlu.keySet();
        Iterator<String> legajos = keys.iterator();
        Alumno unAlu;
        while (legajos.hasNext()) {
            String legajo = (String) legajos.next();
            unAlu = tablaAlu.get(legajo);
            System.out.println("Alumno:" + unAlu);
        }
    }
}
```

```
public class Alumno {
    private String leg;
    private String ape, nom;
    . . .
    public String getApe(String a){
        return ape;
    }
    public String toString(){
        return "Alumno:"+leg+"-"+ape+", "+nom;
    }
}
```



Alumno:Alumno:3421/7-RUSCITI, Maria
Alumno:Alumno:7892/2-GOMEZ, Juana
Alumno:Alumno:7892/8-GOMEZ, Juana
Alumno:Alumno:3321/0-PEREZ, Sol

Colecciones

Funciones Lambda

- Las **expresiones o funciones lambda** son una forma corta de escribir funciones o bloques de código que pueden ser pasados como parámetros a métodos o usar dentro de colecciones.
- Es una manera de representar un método sin tener que escribir toda la estructura del método.
- Se conocen también como funciones anónimas.

Cuál es la sintaxis?

sin parámetros	<code>() -> sentencia</code>
con un parámetro	<code>(parámetro) -> sentencia</code>
con más de un parámetro	<code>(parámetro1, parámetro2) -> sentencia</code>
con sentencias	<code>(parámetro1, parámetro2) -> { sentencia1; sentencia2; }</code>

Para poder utilizar expresiones **lambda** es necesario implementar **interfaces funcionales**.

Colecciones

Funciones Lambda

En Java, una **interfaz funcional** es una interfaz que contiene un único método abstracto. Una **función lambda** (->) en Java es una expresión que permite definir una implementación rápida y anónima de una **interfaz funcional**, utilizando una sintaxis compacta para representar ese único método.

Definición de una interfaz funcional propia

```
@FunctionalInterface 3 usages 2 implementations
public interface OperadorBinario {
    int operador(int num1, int num2); 2 usages 2 in
}
```

Clase concreta

```
public class Suma implements OperadorBinario {
    @Override
    public int operador(int num1, int num2) {
        return num1 + num2;
    }
    public static void main(String[] args) {
        OperadorBinario s = new Suma();
        System.out.println(s.operador(1, 2));
    }
}
```

Clase anónima

```
OperadorBinario suma = new OperadorBinario() {
    @Override 2 usages
    public int operador(int num1, int num2) {
        return num1 + num2;
    }
};
System.out.println(suma.operador(3,4));
```

Función lambda

```
OperadorBinario sum = (int num1, int num2) -> {
    return num1 + num2;
};
System.out.println(sum.operador(1,2));
```

Colecciones

Funciones Lambda de la API de Java (`java.util.function`)

Consumer: Recibe un **valor** y no retorna valor. Recibe un parámetro de tipo T y no se retorna nada.

```
@FunctionalInterface
public interface Consumer<T> {
    void accept(T t);
}
```

implementación del método
`accept(T t)`

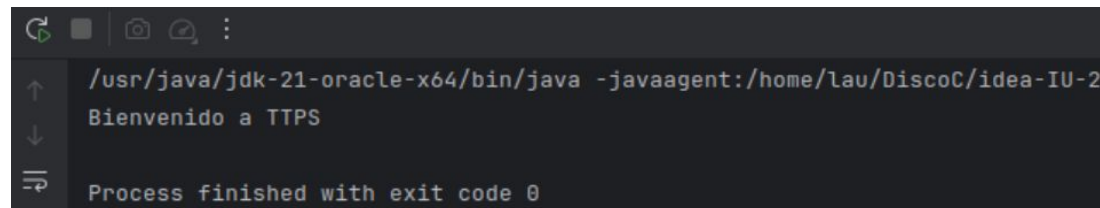
```
Consumer<String> consumer = (param) -> {
    System.out.println(param);
};
Consumer<String> consumer = (param) -> System.out.println(param);

Consumer<String> consumer = System.out::println;

consumer.accept("Bienvenido a TTPS");
```

3 formas
equivalentes

Cuando la lambda llama directamente a un método existente, sin modificar su comportamiento se puede reemplazar `->` con `::`



```
/usr/java/jdk-21-oracle-x64/bin/java -javaagent:/home/lau/DiscoC/idea-IU-2
Bienvenido a TTPS
Process finished with exit code 0
```


Colecciones

Funciones Lambda de la API de Java (java.util.function)

BiConsumer: Recibe dos **valores** y no retorna valor. Recibe dos parámetro de tipo T, U y no se retorna nada.

```
@FunctionalInterface
public interface BiConsumer<T,U> {
    void accept(T y, U u);
}
```

implementación del método
`accept(T t)`

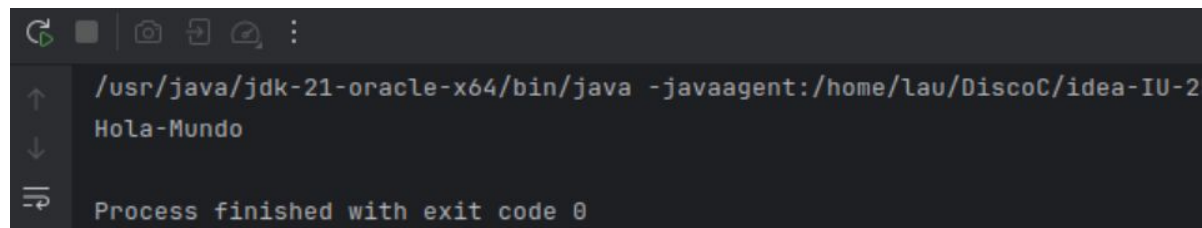
```
BiConsumer<String, String> biConsumer = (a,b) -> {
    System.out.println(a+ "-" +b);
};
```

```
BiConsumer<String, String> biConsumer = (a,b) -> System.out.println(a+ "-" +b);
```

```
biConsumer.accept("Hola", "Mundo");
```

2 formas
equivalentes

Acá no se puede usar `::` porque es válida únicamente cuando hay un parámetro.



```
/usr/java/jdk-21-oracle-x64/bin/java -javaagent:/home/lau/DiscoC/idea-IU-2
Hola-Mundo
Process finished with exit code 0
```

Colecciones

Funciones Lambda de la API de Java (java.util.function)

Supplier: No recibe **valores** y debe retornar algo.

```
@FunctionalInterface
public interface Supplier<T> {
    T get();
}
```

implementación del
método `get()`

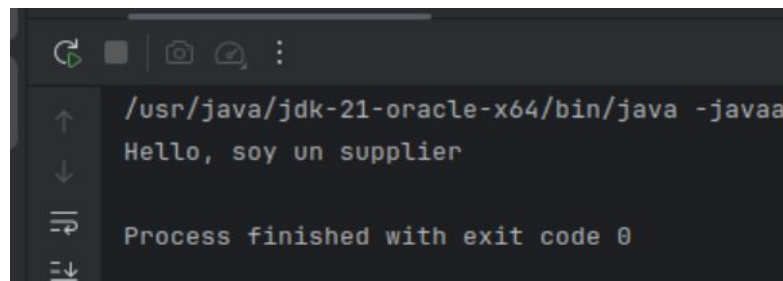
```
Supplier<String> supplier = () -> {
    return "Hello, soy un supplier";
};

//no se escribe el return con la forma corta
Supplier<String> supplier = () -> "Hello, soy un supplier";

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

2 formas
equivalentes

Acá no se puede usar `::` porque no existe parámetro de entrada.



```
/usr/java/jdk-21-oracle-x64/bin/java -javaa
Hello, soy un supplier

Process finished with exit code 0
```

Colecciones

Funciones Lambda de la API de Java (java.util.function)

Function: Represents a función que acepta un argumento y produce un resultado. Su único método funcional es `apply()`.

```
@FunctionalInterface
public interface Function<T, R> {
    R apply(T t);
}
```

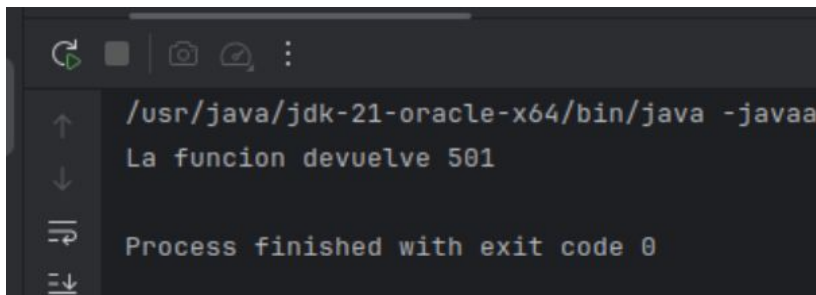
implementación del método
`apply(T t)`

```
Function<Integer, String> function = (num) -> {
    return "La funcion devuelve " + (num+500);
};
Function<Integer, String> function1 = (num) -> "La función devuelve " + (num+500);

System.out.println(function.apply(1));
```

2 formas
equivalentes

Acá no se puede usar `::` porque es válida únicamente cuando hay un parámetro.



```
/usr/java/jdk-21-oracle-x64/bin/java -javaa
La funcion devuelve 501
Process finished with exit code 0
```

Colecciones

Funciones Lambda de la API de Java (java.util.function)

BiFunction: Representa una función que acepta dos argumentos y produce un resultado. Su único método funcional es `apply()`.

```
@FunctionalInterface
public interface BiFunction<T, U, R> {
    R apply(T t, U u);
}
```

implementación del método
`apply(T,U)`

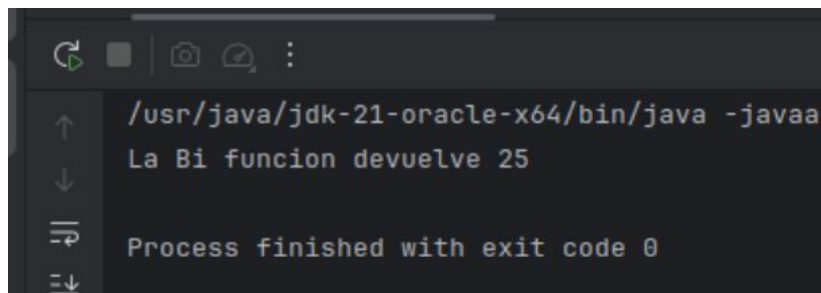
```
BiFunction<Integer, Integer, String> function = (a, b) -> {
    return "La Bi funcion devuelve " + (a+b);
};
```

```
Function<Integer, Integer, String> function = (a, b) -> "La suma es " + (a+b);
```

```
System.out.println(function.apply(10, 15));
```

2 formas
equivalentes

Acá no se puede usar `::` porque es válida únicamente cuando hay un parámetro.



```
/usr/java/jdk-21-oracle-x64/bin/java -javaa
La Bi funcion devuelve 25
Process finished with exit code 0
```

Colecciones

Funciones Lambda de la API de Java (java.util.function)

Predicate: Represents a función que recibe un valor y devuelve un booleano.

```
@FunctionalInterface
public interface Predicate<T> {
    boolean test(T t);
}
```

implementación del método
test(T)

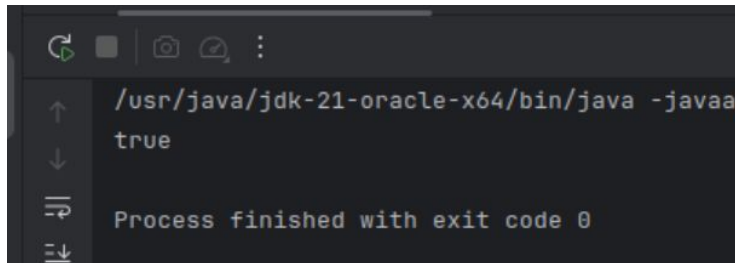
```
Predicate<String> predicate = (str) -> {
    return str.length()>5;
};
```

```
Predicate<String> predicate = (str) -> str.length()>5;
```

```
System.out.println(predicate.test("Este texto tiene más de 5 caracteres"));
```

2 formas
equivalentes

No se puede usar `::` porque no se hace referencia a un método existente sino que se tiene una expresión de comparación: `str.length()>5`



```
/usr/java/jdk-21-oracle-x64/bin/java -javaa
true
Process finished with exit code 0
```

Colecciones

Funciones Lambda de la API de Java (java.util.function)

BiPredicate: Represents a función que recibe dos valores y devuelve un booleano.

```
@FunctionalInterface
public interface BiPredicate<T, U> {
    boolean test(T t, U u);
}
```

implementación del método
`test(T, U)`

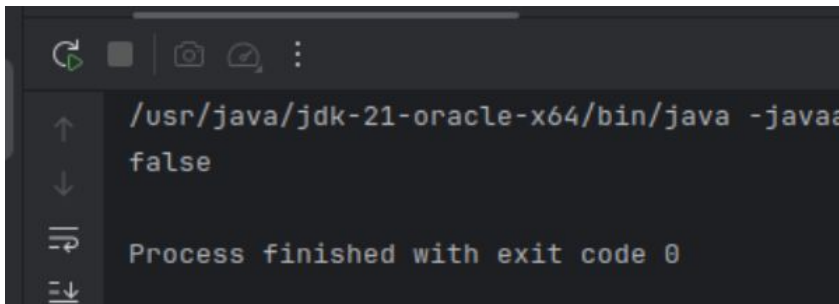
```
BiPredicate<Integer, Integer> biPredicate = (a, b) -> {
    return (a>b);
};

BiPredicate<Integer, Integer> biPredicate1 = (a, b) -> (a>b);

System.out.println(biPredicate1.test(4,6));
```

2 formas
equivalentes

No se puede usar `::` porque no se hace referenci a un método existente sino que se tiene una expresión de comparación:
`a>b`



```
/usr/java/jdk-21-oracle-x64/bin/java -javaa
false
Process finished with exit code 0
```

Colecciones

Funciones Lambda de la API de Java (`java.util.function`)

UnaryOperator: Represents a función que recibe un valor y devuelve otro valor del mismo tipo.

```
@FunctionalInterface
public interface UnaryOperator<T> extends Function<T,T>{
    boolean apply(T t, U u);
}
```

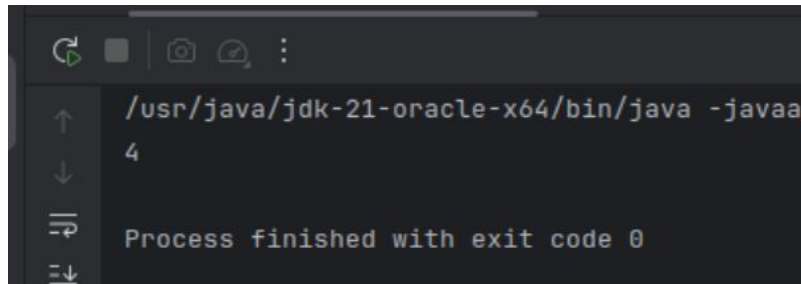
implementación del método
`apply(T, U)`

```
UnaryOperator<Integer> unary = (number) -> {
    return number * number;
};
UnaryOperator<Integer> unary = (number) -> number * number;

System.out.println(unary.apply(2));
```

2 formas
equivalentes

No se puede usar `::` porque no se hace referenci a un método existente sino que se tiene una expresión aritmética: `a+b`



```
/usr/java/jdk-21-oracle-x64/bin/java -javaa
4
Process finished with exit code 0
```

Colecciones

Funciones Lambda de la API de Java (`java.util.function`)

BinaryOperator: Represents a función que recibe dos valores del mismo tipo y devuelve un valor del mismo tipo.

```
@FunctionalInterface
public interface BinaryOperator<T> extends BiFunction<T,T,T>{
    boolean apply(T t, U u);
}
```

implementación del método
`apply(T, U)`

No se pone `BinaryOperator<Integer, Integer, Integer>` porque es redundante.

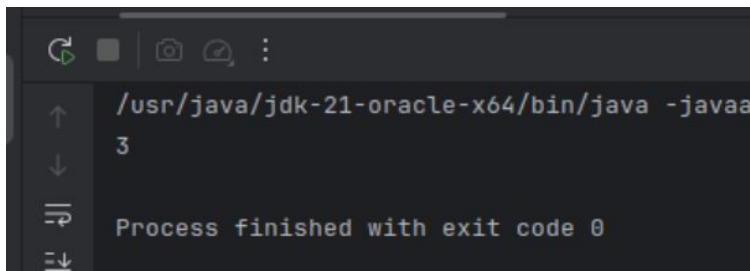
```
BinaryOperator<Integer> binaryOperator = (a, b) ->{
    return a+b;
};
```

```
BinaryOperator<Integer> binaryOperator = (a, b) -> a+b;
```

```
System.out.println(binaryOperator.apply(1,2));
```

2 formas
equivalentes

No se puede usar `::` porque no se hace referenci a un método existente sino que se tiene una expresión aritmética: `a+b`



```
/usr/java/jdk-21-oracle-x64/bin/java -javaa
3
Process finished with exit code 0
```


Colecciones

Funciones Lambda - Síntesis

Nombre	Entrada	Salida	método funcional
Consumer<T>	T	-	<code>void accept(T)</code>
BiConsumer<T,U>	T, U	-	<code>void accept(T,U)</code>
Supplier<T>	-	T	<code>T get()</code>
Function<T,R>	T	R	<code>R apply(T)</code>
BiFuncion<T, U, R>	T, U	R	<code>R apply(T, U)</code>
Predicate<T>	T	boolean	<code>boolean test(T)</code>
Predicate<T>	T	boolean	<code>boolean test(T)</code>
BiPredicate<T, U>	T, U	boolean	<code>boolean test(T, U)</code>
UnaryOperator<T>	T	T	<code>T apply(T)</code>
BinaryOperator<T>	T, T	T	<code>T apply(T, T)</code>

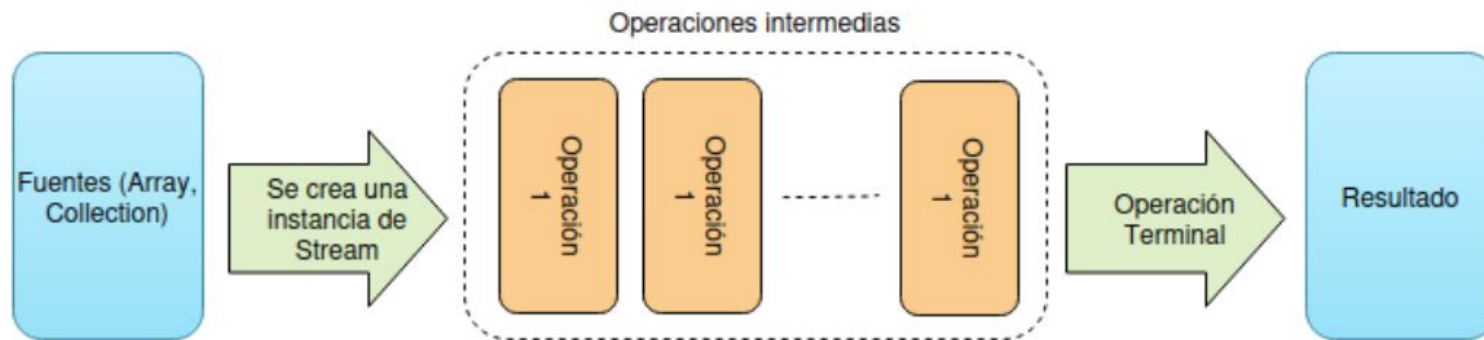
Streams

Qué son?

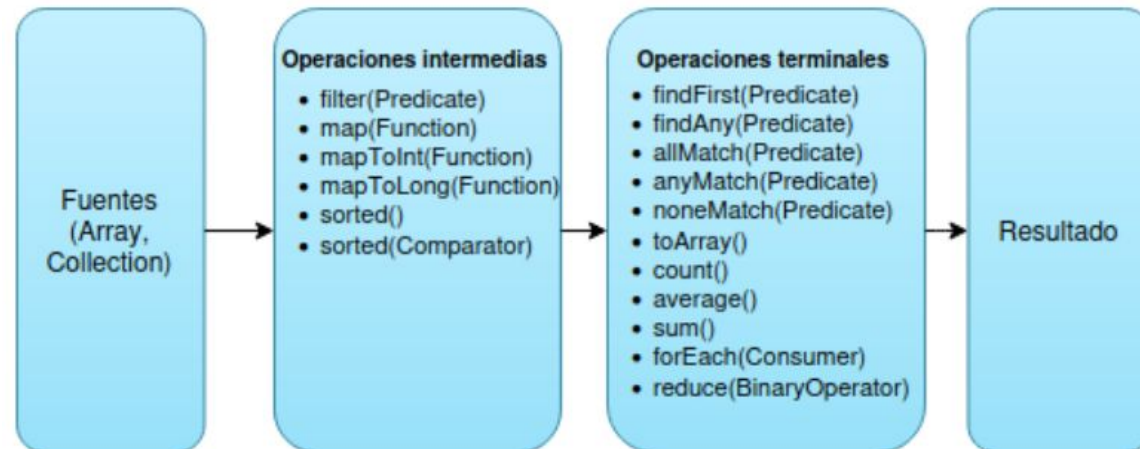
Los streams en JAVA se utilizan para procesar secuencias de elementos. La clase principal es `Stream<T>` y se encuentra en el paquete `java.util.stream`.

Un stream es un flujo de datos sobre el que se pueden hacer operaciones encadenadas. Las operaciones pueden ser intermedias o terminales.

Los streams introducen el paradigma funcional en Java.



Las operaciones intermedias como `filter`, `map` o `sort` devuelven un stream, por lo que podemos encadenar múltiples operaciones intermedias.



Las operaciones terminales como `forEach` o `reduce` son void o devuelven un resultado que no es un stream.

Streams

Creación

Pueden crearse a partir de diferentes fuentes de datos como colecciones y arreglos mediante los métodos `stream()` y `of()`.

A partir de un arreglo	<pre>String[] arr = {"a", "b", "c"}; Stream<String> stream = Arrays.stream(arr);</pre>
A partir de una Collection	<pre>List<String> list = new LinkedList(); list.add("a"); list.add("b"); list.add("c"); Stream<String> stream = list.stream();</pre>
A partir de elementos	<pre>Stream<String> stream = Stream.of("a", "b", "c");</pre>
A partir de un archivo de texto	<pre>Stream<String> stream = Files.lines(Path.get("bandas.txt"));</pre>

Streams

Operación forEach

forEach

`forEach` es una operación final de los streams en Java. El `forEach` de la interfaz `Stream` se utiliza para realizar una acción en cada elemento de un Stream. Recibe como argumento una interfaz `Consumer`.

```
Empleado p1 = new Empleado("juan", "sanchez", 20);
Empleado p2 = new Empleado("ana", "gomez", 12);
Empleado p3 = new Empleado("pedro", "gutierrez", 40);
List<Empleado> lista = Arrays.asList(p1,p2,p3);
Stream<Empleado> stream = lista.stream();

//Lista de empleados con for each tradicional
for (Empleado e:lista ) {
    System.out.println(e);
}

//Lista de empleados con foreach de Stream
stream.forEach((Empleado e) -> System.out.println(e));
```

Consumer

Se crea un `Stream` a partir de un `List`

Se imprimen los datos con `foreach` tradicional y el `foreach` de Stream

```
Consumer<Empleado> consumer=new Consumer<Empleado>(){
    @Override
    public void accept(Empleado e) {
        System.out.println(e);
    }
};
```

```
// Equivalente a stream.forEach(...)
for (Empleado e : listaWrapper) {
    consumer.accept(e);
}
```

Stream a partir de la lista

Streams

Operación filter

filter

A **filter** es una operación intermedia de los streams en Java. Toma un `Stream<T>` y devuelve otro `Stream<T>`. Recibe como argumento una interfaz `Predicate<T>` (una función que devuelve true o false para cada elemento). Solo los elementos que cumplen la condición (true) pasan al siguiente paso del pipeline.

```
Empleado p1 = new Empleado("juan", "sanchez", 20);
Empleado p2 = new Empleado("ana", "gomez", 12);
Empleado p3 = new Empleado("pedro", "gutierrez", 40);
List<Empleado> lista = Arrays.asList(p1, p2, p3);
Stream<Empleado> stream = lista.stream();
```

```
                Predicate
            stream.filter(e -> e.getEdad() > 18)
                  .forEach(System.out::println);
```

Se crea un `Stream` a partir de un `List`

Siempre devuelve una nueva lista, no modifica la lista original.

Es lazy, no hace nada hasta que se ejecuta una operación terminal.

Podría agregar en la condición:

`&& e.getApellido().startsWith("G")`

Esta es la clase que representa nuestro lambda. Lo hace el compilador internamente.

```
Predicate<Empleado> mayorEdad = new Predicate<Empleado>() {
    @Override
    public void test(Empleado e) {
        return e.getEdad() > 18;
    }
};
```

Para cada uno de los elementos, si test da true pasan al `forEach`.

Streams

Operación map

map

Un **map** es una operación intermedia de los streams en Java. Un **map** transforma los elementos de un stream en otros elementos. Recibe una **Function<T,R>** (T:tipo de entrada, R:tipo de salida). Devuelve un nuevo **Stream<R>** con los elementos transformados, **map** es como un convertidor de los elementos del stream.

```
Empleado p1 = new Empleado("juan", "sanchez", 20);
Empleado p2 = new Empleado("ana", "gomez", 12);
Empleado p3 = new Empleado("pedro", "gutierrez", 40);
List<Empleado> lista = Arrays.asList(p1, p2, p3);
Stream<Empleado> stream = lista.stream();

stream().filter(e -> e.getEdad() > 18)
    .map(Empleado e -> e.getNombre().toUpperCase() +
        " " + e.getApellido().toUpperCase())
        Function
    .forEach(System.out::println);
```

Se crea un **Stream** a partir de un **List**

Siempre devuelve una nueva lista, no modifica la lista original.

Esta es la clase que representa nuestro lambda. Lo hace el compilador internamente.

```
Function<Empleado, String> nomApeUpper = new Function<Empleado, String>() {
    @Override
    public String apply(Empleado e) {
        return e.getNombre().toUpperCase() + " " + e.getApellido().toUpperCase();
    }
};
```

Streams

Operación sorted

sorted

sorted es una operación intermedia de Stream que ordena los elementos. Devuelve un nuevo stream ordenado, no modifica la colección original. Se puede usar de dos formas:

- **sorted()** sin parámetros → usa el orden natural de los elementos (Comparable).
- **sorted(Comparator)** → ordena según un criterio definido por un Comparator.

```
Empleado p1 = new Empleado("juan", "sanchez", 20);
Empleado p2 = new Empleado("ana", "gomez", 12);
Empleado p3 = new Empleado("pedro", "gutierrez", 40);
List<Empleado> lista = Arrays.asList(p1, p2, p3);
Stream<Empleado> stream = lista.stream();
```

Comparator

```
stream.sorted(Comparator.comparing(Empleado e -> e.getEdad))
    .forEach(System.out::println);
```

Se crea un **Stream** a partir de un **List**

La colección no cambia.

Se pueden usar todos los getters

Esta es la clase que representa nuestro lambda. Lo hace el compilador internamente.

```
Comparator<Empleado> compPorEdad = new Comparator<Empleado>(){
    @Override
    public int compare(Empleado e1, Empleado e2) {
        return Integer.compare(e1.getEdad(), e2.getEdad());
    }
};
```

```
lista.stream()
    .sorted(compPorEdad)
    .forEach(System.out::println);
};
```

Streams

Operación toList

toList()

`toList()` es una operación final que recoge los elementos de un stream en una lista (List).

A partir de la versión Java 19, ya no se usa, directamente se pone el `toList()`

```
Empleado p1 = new Empleado("Juan", "sanchez", 20);
Empleado p2 = new Empleado("Ana", "gomez", 12);
Empleado p3 = new Empleado("Pedro", "gutierrez", 40);
List<Empleado> lista = Arrays.asList(p1, p2, p3);
Stream<Empleado> stream = lista.stream();
List<String> list = lista.stream()
    .map(e -> e.getApellido().toUpperCase() + ", " + e.getNombre())
    .toList();
System.out.println(list);
```

```
//antes de Java 16
list = lista.stream()
    .map(e -> e.getApellido().toUpperCase() + ", " + e.getNombre())
    .collect(Collectors.toList());
System.out.println(list);
```

```
/usr/lib/jvm/jdk-21.0.4-oracle-x64/bin/java -ja
[SANCHEZ, Juan, GOMEZ, Ana, GUTIERREZ, Pedro]
[SANCHEZ, Juan, GOMEZ, Ana, GUTIERREZ, Pedro]
```

Resumen:

- `stream.toList()` → lista inmutable, más conciso (Java 16+).
- `stream.collect(Collectors.toList())` → lista modificable, más flexible.

Streams

Operación distinct

distinct

distinct es una operación intermedia de Stream que elimina elementos duplicados. Devuelve un nuevo stream sin elementos duplicados, no modifica la colección original. Se basa en los métodos **equals()** y **hashCode()** de los objetos.

```
Empleado p1 = new Empleado("juan", "sanchez", 20);
Empleado p2 = new Empleado("ana", "gomez", 12);
Empleado p3 = new Empleado("ana", "gomez", 12);
Empleado p4 = new Empleado("pedro", "gutierrez", 40);
Empleado p5 = new Empleado("pedro", "vegas", 40);
List<Empleado> lista = Arrays.asList(p1, p2, p3, p4, p5);
Stream<Empleado> stream = lista.stream();
lista.stream()
    .distinct()
    .forEach(e -> System.out.println(e));
```

En este ejemplo devuelve:

```
/usr/lib/jvm/jdk-21.0.4-oracle-x64
juan sanchez, 20 años
ana gomez, 12 años
pedro gutierrez, 40 años
pedro vegas, 40 años
```

```
lista.stream()
    .map(e -> e.getNombre())
    .distinct()
    .forEach(e -> System.out.println(e));
```

Trampita para quedarnos con nombres diferentes

```
/usr/lib/jvm/jdk-21.0.4-oracle
juan
ana
pedro
```

distinct() no recibe lambdas.

Es uno de esos métodos de **Stream** que **ya traen la lógica definida** y no es necesario que se pase un **Predicate**, **Comparator** o **Consumer**.

Streams

Operación match

AnyMatch & NoneMatch & allMatch

anyMatch evalúa si existe al menos un elemento del stream que satisface el predicado que se recibe como parámetro y en ese caso retorna true, caso contrario false.

```
Empleado p1 = new Empleado("Juan", "Sanchez", 20);
Empleado p2 = new Empleado("Pedro", "Gomez", 12);
Empleado p3 = new Empleado("Ana", "Gomez", 42);
Empleado p4 = new Empleado("Lucia", "Gutierrez", 12);
List<Empleado> lista = Arrays.asList(p1, p2, p3, p4);

Stream<Empleado> stream = lista.stream();
System.out.println("AnyMatch");
boolean res = lista.stream().anyMatch((empl) -> empl.getEdad() < 18);
System.out.println(res);

System.out.println("None Match");
res = lista.stream().noneMatch((empl) -> empl.getEdad() < 18);
System.out.println(res);

System.out.println("All Match");
res = lista.stream().allMatch((empl) -> empl.getEdad() < 18);
System.out.println(res);
```

Resumen:

- Usar **anyMatch** si solo
- Usar **filter + findFirst** para encontrar el primer valor.


Streams

Operación find

findFirst & findAny

FindFirst evalúa si existe al menos un elemento del stream que satisface el predicado que se recibe como parámetro y en ese caso retorna true, caso contrario false.

```
Empleado p1 = new Empleado("Juan", "Sanchez", 20);
Empleado p2 = new Empleado("Pedro", "Gomez", 12);
Empleado p3 = new Empleado("Ana", "Gomez", 42);
Empleado p4 = new Empleado("Lucia", "Gutierrez", 40);
List<Empleado> lista = Arrays.asList(p1, p2, p3, p4);
Stream<Empleado> stream = lista.stream();
//Quiero que me devuelva el empleado Ana Gomez si existe
Optional<Empleado> o = lista.stream()
    .filter(empl -> "Gomez".equals(empl.getApellido()))
    .findFirst();
if (o.isPresent())
    System.out.println("La edad del 1er empleado Gomez es:" + o.get().getEdad());
else
    System.out.println("No se encontró el empleado");
```



```
/usr/lib/jvm/jdk-21.0.4-oracle-x64/bin
La edad del 1er empleado Gomez es:12
```

Resumen:

- Usar **anyMatch** si solo se quiere saber si está (true/false).
- Usar **filter + findFirst/findAny** si además se desea recuperar el valor.