

## TEMA 6. Estructuras de datos avanzadas y P00



# Estructuras de datos avanzadas y POO. Interfaces funcionales y Streams.

Interfaces funcionales.

- Poder cambiar el código (método) a ejecutar en tiempo de ejecución.
- Basadas en expresiones/funciones lambda.
  - Funciones sin nombre ni clase asociada.
  - Puntero a función.
  - Funciones anónimas.
- Vídeo UPM sobre programación funcional.
  - <https://www.youtube.com/watch?v=8LVdhVMNQSw>
- Ámpliamente utilizada en los lenguajes actuales: Python, Java, Javascript, Typescript, C#, C++

# Estructuras de datos avanzadas y POO. Colecciones en Java.

Sintáxis.

(parámetros) → {cuerpo-lambda}

- `() -> System.out.println("Hello Lambda")`
- `x -> x + 10`
- `(int x, int y) -> { return x + y; }`
- `(String x, String y) -> x.length() - y.length()`
- `(String x) -> {  
    listA.add(x);  
    listB.remove(x);  
    return listB.size();  
}`

# Estructuras de datos avanzadas y POO. Colecciones en Java.

Ejemplo.

```
public class funcional1 {  
    public interface ICalculadora<T> {  
        public T Suma(T op1, T op2);  
        /*public T Resta (T op1, T op2);  
        public T Multiplicacion (T op1, T op2);  
        public T Division (T op1, T op2);*/  
    }  
  
    public static void main(String[] args) {  
        System.out.println(new ICalculadora<Integer>() {  
            @Override  
            public Integer Suma(Integer op1, Integer op2) {  
                return op1.intValue() + op2.intValue();  
            }  
        }.Suma(5, 6).toString());  
        ICalculadora<String> c= (String op1, String op2)->{  
            return op1.concat(op2);  
        };  
        System.out.println(c.Suma("Hola ", "mundo"));  
    }  
}
```

# Estructuras de datos avanzadas y POO. Colecciones en Java.

Interfaces funcionales.

- Interfaces predefinidas para funciones lambda.
- Usas en otras clases, en especial colecciones.
- Paquete `java.util.function`.
- Interfaz con un solo método abstracto, aunque pueden implementar otros no abstractos.
- La concrección del método se realiza en tiempo de ejecución, con una función lambda.
- Características.
  - Son interfaces , como su nombre indica.
  - Tienen un único método abstracto.
  - Se pueden introducir métodos por defecto.
  - Se pueden introducir métodos estáticos.
  - Se indican con la anotación `@FunctionalInterface`.

# Estructuras de datos avanzadas y POO. Colecciones en Java.

Interfaces funcionales.

- Consumidores. **Consumer** <T>. Acepta un valor y no devuelve nada  $(x) \rightarrow \{.. \}$ , también existe la versión biconsumidora  $(x,y) \rightarrow \{.. \}$

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

- Se pueden concatenar consumidores con `andThen`.

```
public static void main(String[] args) {
    ArrayList<Integer> lista = new ArrayList<Integer>();
    for (int i = 0; i < 20; i++) {
        lista.add((int) (Math.random() * 10000));
    }
    Consumer<Integer> c = (Integer i) -> System.out.print(i);
    Consumer concatenado=c.andThen( j-> System.out.println("->" + (j+1)));
    for (Integer in : lista) {
        concatenado.accept(in);
    }
}
```

# Estructuras de datos avanzadas y POO. Colecciones en Java.

Interfaces funcionales.

- Proveedores. **Supplier** <T>. No acepta parámetros, pero devuelve uno ()→{ return x;}. Util para crear factorias de objetos.

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

- Se pueden concatenar consumidores con andthen.

```
public class Factoria {
    private static HashMap<String,Supplier<Persona>> clases;
    static{
        clases= new HashMap<>();
        clases.put("persona",Persona::new);
        clases.put("estudiante", Estudiante::new);
    }
    public static Persona get(Supplier<? extends Persona> s){
        return s.get();
    }
}
```

```
public static Persona create(String nombre) {
    if(Factoria.clases.get(nombre)!=null)
        return Factoria.clases.get(nombre).get();
    else
        return null;
}

public static void main(String[] args) {
    Estudiante e=(Estudiante)
    Factoria.create("estudiante");
    System.out.println(e.getClass().getName());
}
}
```

# Estructuras de datos avanzadas y POO. Colecciones en Java.

Interfaces funcionales.

- Funciones. **Supplier <T>**.  $(x) \rightarrow \{ \text{return } y; \}$ . También existen las bifunciones  $(x,y) \rightarrow \{ \text{return } z; \}$ . Se aplica el algoritmo con `apply`, además se pueden concatenar con `andThen` y realizar composiciones con `compose`.

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

    default <V> Function<V, R> compose(Function<? super V, ? extends T> before) {
        Objects.requireNonNull(before);
        return (V v) -> apply(before.apply(v));
    }

    default <V> Function<T, V> andThen(Function<? super R, ? extends V> after) {
        Objects.requireNonNull(after);
        return (T t) -> after.apply(apply(t));
    }

    static <T> Function<T, T> identity() {
        return t -> t;
    }
}
```

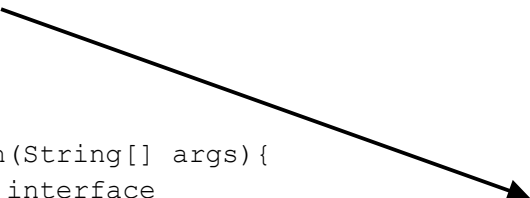


# Estructuras de datos avanzadas y POO. Colecciones en Java.

Interfaces funcionales.

- PredicadosFunciones. **Predicate** <T>. (x)→{ return boolean;}. Se pueden combinar con **and** y **negative** con otros predicados. Para ejecutar el predicado se llama al método **test**

```
import java.util.function.Predicate;
public class PredicateInterfaceExample {
    static Boolean checkAge(int age){
        if(age>17)
            return true;
        else return false;
    }
    public static void main(String[] args){
        // Using Predicate interface
        Predicate<Integer> predicate = PredicateInterfaceExample::checkAge;
        // Calling Predicate method
        boolean result = predicate.test(25);
        System.out.println(result);
    }
}
```



# Estructuras de datos avanzadas y POO. Colecciones en Java.

Interfaces funcionales.

- Operadores. **UnaryOperator <T>**.  $(x) \rightarrow \{ \text{return } x; \}$ . Extiende de Function. Se puede usar además el operador **then**, **apply** y **compose**. Existe también el operador binario.
- Listado de interfaces disponibles:
- 

<https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html>

# Estructuras de datos avanzadas y POO. Colecciones en Java.

## Interfaces funcionales.

Interface	Description
<b>BiConsumer</b> <T,U>	Represents an operation that accepts two input arguments and returns no result.
<b>BiFunction</b> <T,U,R>	Represents a function that accepts two arguments and produces a result.
<b>BinaryOperator</b> <T>	Represents an operation upon two operands of the same type, producing a result of the same type.
<b>BiPredicate</b> <T,U>	Represents a predicate (boolean-valued function) of two arguments.
<b>BooleanSupplier</b>	Represents a supplier of boolean-valued results.
<b>Consumer</b> <T>	Represents an operation that accepts a single input argument and returns no result.
<b>DoubleBinaryOperator</b>	Represents an operation upon two double-valued operands and producing a double-valued result.
<b>DoubleConsumer</b>	Represents an operation that accepts a single double-valued argument and returns no result.
<b>DoubleFunction</b> <R>	Represents a function that accepts a double-valued argument and produces a result.
<b>DoublePredicate</b>	Represents a predicate (boolean-valued function) of one double-valued argument.
<b>DoubleSupplier</b>	Represents a supplier of double-valued results.
<b>DoubleToIntFunction</b>	Represents a function that accepts a double-valued argument and produces an int-valued result.
<b>DoubleToLongFunction</b>	Represents a function that accepts a double-valued argument and produces a long-valued result.
<b>DoubleUnaryOperator</b>	Represents an operation on a single double-valued operand that produces a double-valued result.

# Estructuras de datos avanzadas y POO. Interfaces funcionales y Streams.

## Streams.

- Las operaciones sobre las colecciones son comunes, independiente del programa.
- Posible personalizar su comportamiento con interfaces funcionales.
- Ejemplo:

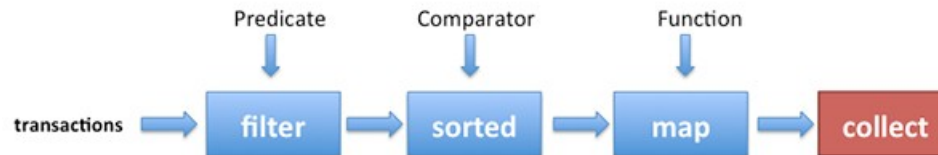
```
private void moveBullets() {  
    for (Bullet b : this.bullets) {  
        b.move();  
    }  
    this.bullets.removeIf(b -> (b.getPosicion().getX() <= b.getInc() || !b.isLive() || b.hascollided()));  
}
```

- ¿A qué método de la interfaz funcional llamara removeIf para evaluar?
- Pensar en cómo sería el código anterior sin usar interfaces funcionales.

# Estructuras de datos avanzadas y POO. Interfaces funcionales y Streams.

## Streams.

- Se define en la interfaz Stream, se conoce también como flujos de datos.
- Se aplica a colecciones: Arrays, List, Map, Queue, Set...y otros elementos como ficheros.
- Proporciona un conjunto de operaciones sobre el stream.
- Las operaciones reciben interfaces funcionales.
- Se pueden encadenar operaciones.



# Estructuras de datos avanzadas y POO. Interfaces funcionales y Streams.

## Streams.

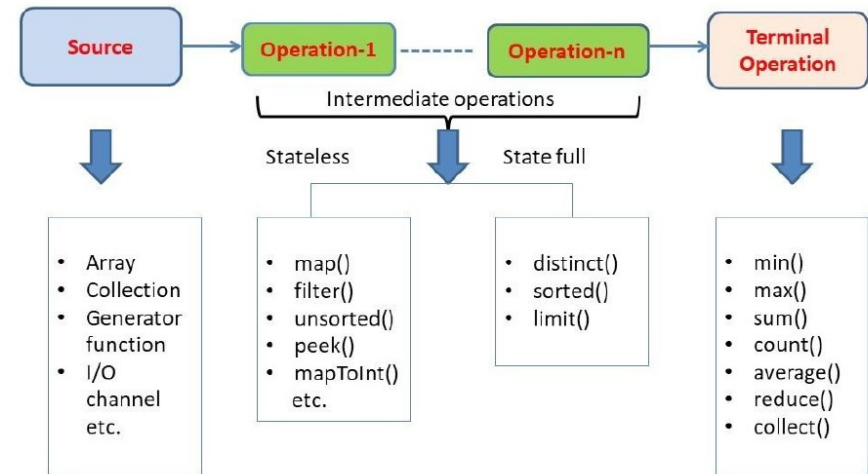
Existen más de 20 operadores.

Se tienen:

- Operaciones finales para obtener un número, una cadena, un objeto o una colección denominados colectores.
- Operaciones intermedias, que pueden concatenar otras operaciones.
- Con estado (statefull) y sin estado (stateless)

Es posible ejecutar las opciones en paralelo con `parallelStream`.

```
int sum = widgets.stream()  
    .filter(w -> w.getColor() == RED)  
    .mapToInt(w -> w.getWeight())  
    .sum();
```



# Estructuras de datos avanzadas y POO. Interfaces funcionales y Streams.

## Streams.

Nombre	Definición	Descripción
Distinct	Stream<T> distinct()	Devuelve un stream con los elementos que sean diferentes en el stream.
Filter	Stream<T> filter(Predicate<? super T> predicate)	Devuelve un stream con los elementos que sean ciertos para la interfaz de tipo Predicado que se le pasa
FlatMap	<R> Stream<R> flatMap (Function<? super T,? extends Stream<? extends R>> mapper)	Aplana un flujo, une diferentes flujos de datos en uno solo, por ejemplo procesar una lista cuyos elementos a su vez contienen una colección que se ha de procesar.
Limit	Stream<T> limit(long maxSize)	Un nuevo flujo con un tamaño máximo
Map	<R> Stream<R> map(Function<? super T,? extends R> mapper)	Devuelve un nuevo flujo con elementos a los que se les ha aplicado la interfaz funcional indicada.
Peek	Stream<T> peek(Consumer<? super T> action)	Recibe una interfaz consumidor, es útil para depuración.
Skip	Stream<T> skip(long n)	Elimina los n primeros elementos del flujo de datos.
Sorted	Stream<T> sorted()	Devuelve un flujo ordenado.

Nombre	Definición	Descripción
allMatch	boolean allMatch(Predicate<? super T> predicate)	Cierto si todos los elementos del stream cumplen con el predicado
anyMatch	boolean anyMatch(Predicate<? super T> predicate)	Cierto si alguno de los elementos cumple con el predicado
Count	long count()	Devuelve el número de elementos del stream
NoneMatch	boolean noneMatch(Predicate<? super T> predicate)	Cierto si ningún elemento cumple con el predicado
reduce	T reduce(T identity, BinaryOperator<T> accumulator)	Reduce una colección a un único objeto de tipo T aplicando una interfaz funcional de tipo función con 2 parámetros de tipo T.
Max	Optional<T> max(Comparator<? super T> comparator)	Devuelve el elemento máximo, aplicando un comparador
Min	Optional<T> min(Comparator<? super T> comparator)	Igual que el anterior, pero obtiene el mínimo
FindAny	Optional<T> findAny()	Devuelve un elemento del flujo.
FindFirst	Optional<T> findFirst()	Devuelve el primer elemento del stream
Collect	<R,A> R collect(Collector<? super T,A,R> collector)	Se realiza una selección de los elementos del stream, donde R es el tipo del resultado, A es el acumulador y T es el tipo.

# Estructuras de datos avanzadas y POO. Interfaces funcionales y Streams.

## Streams.

```
List<String> list = people.stream().map(Person::getName).collect(Collectors.toList());
```

```
Set<String> set = people.stream().map(Person::getName).collect(Collectors.toCollection(TreeSet::new));
```

```
String joined = things.stream().map(Object::toString).collect(Collectors.joining(", "));
```

```
int total = employees.stream().collect(Collectors.summingInt(Employee::getSalary));
```

```
Map<Department, List<Employee>> byDept =  
employees.stream().collect(Collectors.groupingBy(Employee::getDepartment));
```

```
Map<Department, Integer> totalByDept= employees.stream().collect(Collectors.groupingBy(Employee::getDepartment,  
Collectors.summingInt(Employee::getSalary)));
```

```
Map<Boolean, List<Student>> passingFailing =students.stream()  
    .collect(Collectors.partitioningBy(s -> s.getGrade() >= PASS_THRESHOLD));
```