Java

Funcional
Streams

Imperativo vs Funcional

- Imperativo: en base a un objetivo, determinar los pasos a seguir
  - Uso de variables para mantener estado
  - Usar un ciclo para iterar sobre una colección ("*external iteration*")
  - Acceder a los elementos en forma secuencial
- Funcional: especificar que se desea obtener, pero no cómo
  - Foco en la inmutabilidad: no se usan variables
  - Se especifica la acción a desarrollar sobre una colección ("*internal iteration*")

Expresiones lambda

Bloque de código que se le puede pasar a un método para que sea ejecutado posteriormente.

```
btn.setOnAction(event -> System.out.println("Hello World!"));
```

```
List<String> list;
...
list.sort(Comparator.comparingInt(s -> s.length()));
```

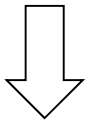
```
Collections.sort(list, (s1, s2) -> s2.length() - s1.length());
```

```
() -> { for (int i = 0; i<= 10; i++) System.out.println(i);}
```

Referencias a métodos

A veces ya existe un método que realiza exactamente la acción que queremos pasarle a otro código.

```
btn.setOnAction(event -> System.out.println(event));
```



```
btn.setOnAction(System.out::println);
```

Referencias a métodos

```
Arrays.sort(points,
        Comparator.comparing(Point::getX)
        .thenComparing(Point::getY));
```

Referencias a métodos

Además de clases y métodos de clase o instancia podemos referenciar métodos propios (*this*) o de la clase padre (*super*)

```
public class Greeter {
    public void greet(ActionEvent e) {
        System.out.println("Hello, " + e);
    }

    public class TimerGreeter extends Greeter {
        @Override
        public void greet(ActionEvent e) {
            Timer t = new Timer(1000, super::greet);
            t.start();
        }
    }
}
```

Alcance de variables

```
public static void repeatMessage(String text, int delay) {
    ActionListener listener = event -> {
        System.out.println(text);
        Toolkit.getDefaultToolkit().beep();
    };
    new Timer(delay, listener).start();
}
```

text es una variable "capturada"

```
public static void countDown(int start, int delay) {
    ActionListener listener = event -> {
        start--;
        System.out.println(start);
    };
    new Timer(delay, listener).start();
}
```

No se pueden modificar variables "capturadas"

Error local variables referenced from a lambda expression must be final or effectively final

Interfaces funcionales más comunes

Interface	Tipo de los Parámetros	Retorna	Nombre método abstracto	Otros métodos
`Runnable`	no	void	`run`	
`Supplier<T>`	no	T	`get`	
`Consumer<T>`	T	void	`accept`	`andThen`
`Function<T,R>`	T	R	`apply`	`andThen, compose, identity`
`BiFunction<T, U, R>`	T, U	R	`apply`	`andThen`

Interfaces funcionales más comunes

Interface	Tipo de los Parámetros	Retorna	Nombre método abstracto	Otros métodos
`UnaryOperator<T>`	T	T	`apply`	`compose, andThen, identity`
`BinaryOperator<T>`	T, T	T	`apply`	`andThen, maxBy, minBy`
`Predicate<T>`	T	boolean	`test`	`and, or, negate, isEqueal`
`BiPredicate<T,U>`	T, U	boolean	`test`	`and, or, negate`

Ejemplos

```
public static void main(String[] args) {
   BinaryOperator<Integer> add = (n1, n2) -> n1 + n2;
   System.out.println(add.apply(3, 4));
   System.out.println(add.apply(add.apply(4, 6), 5));
}
```

Existen interfaces especializadas para `int`, `long` y `double`

```
IntBinaryOperator add2 = (n1, n2) -> n1 + n2;
System.out.println(add2.applyAsInt(3, 4));
System.out.println(add2.applyAsInt(add.apply(4, 6), 5));
```

Ejemplos

```
public class Professor {

    String name;
    Integer age;

    public Professor(String name, Integer age) {
        this.name = name;
        this.age = age;
    }
}
```

Ejemplos

Creamos un método que dada una lista de profesores, aplique un `Consumer` (procedimiento) a cada profesor.

```
public static void process(List<Professor> list,
Consumer<Professor> consumer) {
    for (Professor e : list) {
        consumer.accept(e);
    }
}
```

Ejemplos

```
public static void main(String[] args) {
   List<Professor> professors = Arrays.asList(
           new Professor("Juan", 40),
           new Professor("Sabrina",50),
           new Professor("Andrea",30)
   );

   process(professors, e -> System.out.println(e.name));
   process(professors, e -> { e.age += 1; });
   process(professors, e -> {
       e.name = e.name.toUpperCase();
   });
   process(professors, e -> System.out.println(e.name + ": " + e.age));
}
```

```
Juan
Sabrina
Andrea
```

```
JUAN: 41
SABRINA: 51
ANDREA: 31
```

Ejemplos

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

   Function<Integer, Integer> minus10 = n -> (n - 10);
   Function<Integer, Integer> mult2 = t -> (t * 2);

   System.out.println(minus10.apply(100));

   System.out.println(minus10.andThen(mult2).apply(60));

   System.out.println(minus10.compose(mult2).apply(60));

}
```

```
90
100
110
```

Streams

- Son objetos que implementan la interface Stream
- Permiten realizar tareas utilizando programación funcional
- Operan sobre colecciones pero no generan una nuevo espacio (no pueden ser reutilizados)
- Sobre un stream se realiza una "operación intermedia" que genera un nuevo stream
- Las operaciones intermedias son lazy: no se realizan hasta que se invoca una operación terminal
- Existen Streams especializados: IntStream, DoubleStream, etc.

Streams

- Operaciones intermedias más comunes
  - `filter`
  - `distinct`
  - `limit`
  - `map`
  - `sorted`
- Operaciones terminales más comunes
  - `foreach`
  - Operaciones de reducción: `average, count, max, min, reduce`
  - Obtener una colección: `collect, toArray`
  - Búsqueda: `findFirst, findAny, anyMatch, allMatch`

Ejemplos: IntStream

```
int[] values = {3, 10, 6, 1, 4, 8, 2, 5, 9, 7};

System.out.print("Original values: ");

// IntStream.of(values) invoca a Arrays.stream(values)
IntStream.of(values).forEach(e -> System.out.printf("%d ", e));
System.out.println();

System.out.printf("%nCount: %d%n", IntStream.of(values).count());
System.out.printf("Min: %d%n", IntStream.of(values).min().getAsInt());
System.out.printf("Max: %d%n", IntStream.of(values).max().getAsInt());
System.out.printf("Sum: %d%n", IntStream.of(values).sum());
System.out.printf("Average: %.2f%n",
IntStream.of(values).average().getAsDouble());
```

Ejemplos: IntStream

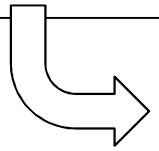
```
System.out.printf("Sum via reduce method: %d%n",
         IntStream.of(values).reduce(0, (x, y) -> x + y));

System.out.printf("Sum of squares via reduce method: %d%n",
               IntStream.of(values).reduce(0, (x, y) -> x + y * y));

System.out.printf("Product via reduce method: %d%n",
               IntStream.of(values).reduce(1, (x, y) -> x * y));

System.out.printf("Even values displayed in sorted order: ");

IntStream.of(values).filter(v -> v % 2 == 0).sorted()
    .forEach(e -> System.out.printf("%d ", e));
System.out.println();
```

Ejemplos: IntStream

```
System.out.printf(
       "Odd values multiplied by 10 displayed in sorted order: ");
IntStream.of(values)
     .filter(value -> value % 2 != 0)
     .map(value -> value * 10)
     .sorted().forEach(value -> System.out.printf("%d ", value));
System.out.println();

System.out.printf("%nSum of integers from 1 to 9: %d%n",
IntStream.range(1, 10).sum());


System.out.printf("Sum of integers from 1 to 10: %d%n",
IntStream.rangeClosed(1, 10).sum());
```

Ejemplos

```
Professor[] aux = new Professor[]{new Professor("Juan Días", 30),
        new Professor("Ana García", 28),
        new Professor("Maria Santillar", 28),
        new Professor("Justo Quintana", 80),
        new Professor("John Watkins", 56),
        new Professor("Mary Sinclair", 56)};

List<Professor> professors = Arrays.asList(aux);

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



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

Ejemplos

```
Predicate<Professor> minus50 = e -> e.getAge() < 50;

List<Professor> list = professors.stream()
      .filter(minus50)
      .sorted(Comparator.comparing(Professor::getAge)
              .thenComparing(Professor::getName))
      .collect(Collectors.toList());
```

Podemos calcular cuál será el resultado, pero no en qué orden se ejecutarán las operaciones intermedias.

¿Son equivalentes?

```
professors.stream()
        .map(Professor::getAge)
        .distinct()
        .sorted()
        .forEach(e -> System.out.println(e));
```

```
professors.stream()
        .map(Professor::getAge)
        .sorted()
        .distinct()
        .forEach(e -> System.out.println(e));
```

¿Son equivalentes?

```
professors.stream()
        .map(Professor::getAge)
        .distinct()
        .sorted()
        .forEach(e -> System.out.println(e));
```

```
professors.stream()
        .distinct()
        .map(Professor::getAge)
        .sorted()
        .forEach(e -> System.out.println(e));
```

Ejemplos

```
Map<Integer, List<Professor>> groupByAge =
       professors.stream()
.collect(Collectors.groupingBy(Professor::getAge));

groupByAge.forEach(
       (age, profs) -> {
         System.out.printf("%d : ",age);
         profs.forEach(p -> System.out.printf("%s ",
                                        p.getName()));
         System.out.println();
       }
);
```

Ejemplos

Promedio de edad de los docentes, descartando los que ya han cumplido 65 años.

```
double average = professors
        .stream()
        .filter(p -> p.getAge() < 65)
        .mapToInt(Professor::getAge)
        .average()
        .getAsDouble();
```

Operaciones terminales

Obtener una lista con los docentes que ya cumplieron 65. Además imprimir el promedio de edad de los mismos.

```
Stream<Professor> stream = professors.stream()
          .filter(p -> p.getAge() >= 65);

List<Professor> up65 = stream.collect(Collectors.toList());
double avg = stream.mapToInt(Professor::getAge)
          .average().getAsDouble();
```

java.lang.IllegalStateException: stream has already been operated upon or closed

Operaciones terminales

Obtener una lista con los docentes que ya cumplieron 65. Además imprimir el promedio de edad de los mismos.

```
Supplier<Stream<Professor>> stream = () ->
professors.stream().filter(p -> p.getAge() >= 65);

List<Professor> up65 = stream.get()
                .collect(Collectors.toList());

double avg = stream.get()
       .mapToInt(Professor::getAge)
       .average()
       .getAsDouble();
```

Streams infinitos. Substreams

Stream provee dos métodos de clase para generar streams infinitos

```
Stream<String> echos = Stream.generate(() -> "Echo");

Stream<Double> randoms = Stream.generate(Math::random);
```

```
Stream<Integer> ints = Stream.iterate(0, n -> n + 1)
```

```
Stream<Double> randoms = Stream.generate(Math::random)
    .limit(100).skip(50);
```

Ejercicios

```
public class Professor {

    String name;
    Integer age;
        char gender;          // 'F' o 'M'
    public Professor(String name, Integer age, char gender) {
        this.name = name;
        this.age = age;
        this.gender = gender;
    }
}
```

Ejercicios

1. Contar cuántos docentes hay de cada género
2. Obtener una lista de los profesores en edad de jubilarse, donde primero aparezcan las mujeres y luegos los hombres, y dentro de cada género ordenados por edad.
3. Obtener el promedio de edad de los educadores por género

Etc.

Para más ejemplos y operaciones sobre *streams* ver
https://stackify.com/streams-guide-java-8/

Para las novedades sobre *streams* en Java 9 y 10

https://www.baeldung.com/java9-stream-collectors

https://www.logicbig.com/tutorials/core-java-tutorial/java-10-changes/collectors-changes.html