Lambdas:

static Collection<T> filter(Collection<T> c, Predicate<T> p);

Why is Java still changing?

* Stream processing
* Passing code to methods with behavior parameterization
* Parallelism and shared mutable data
* Java needs to evolve

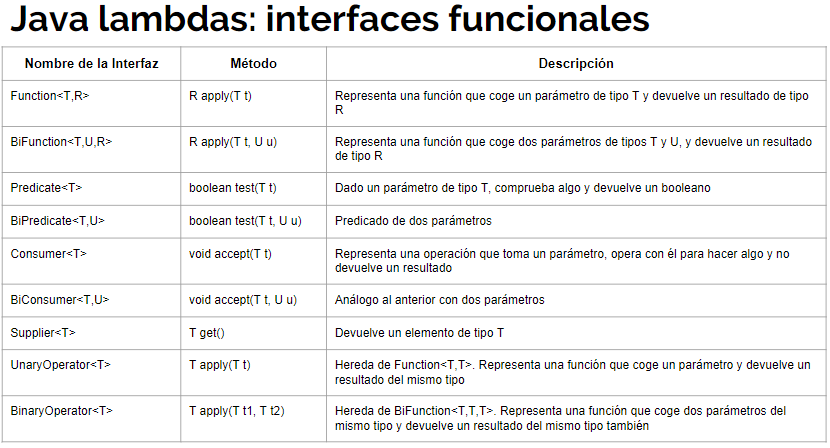
Functions in Java

* Methods and lambdas as first-class citizens
* Passing code: an example
* From passing methods to lambdas
* Java 8: the Streams API and parallelism
* 1.4 Streams vs. Collections
* 1.4.1 External vs. internal iteration and parallelism
* las expresiones lambda son instancias de interfaces funcionales.
* Una interfaz funcional es simplemente una interfaz que tiene exactamente un método abstracto (los métodos default, static y los públicos heredados de Object no cuentan)
* Java contiene en su librería múltiples casos de SAM (Single Abstract Method) interfaces:

@FunctionalInterfaceinterface Comparator<T> {

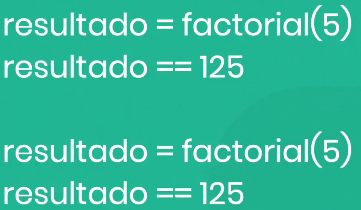
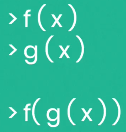
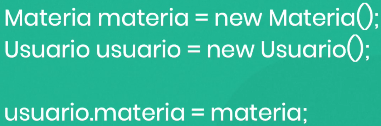
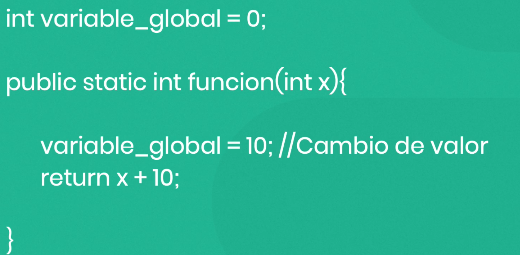
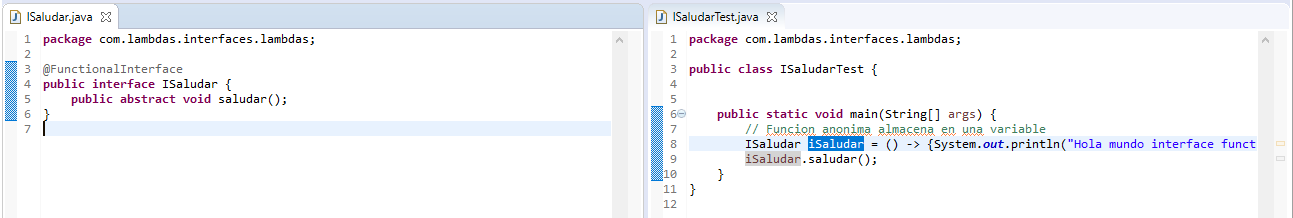
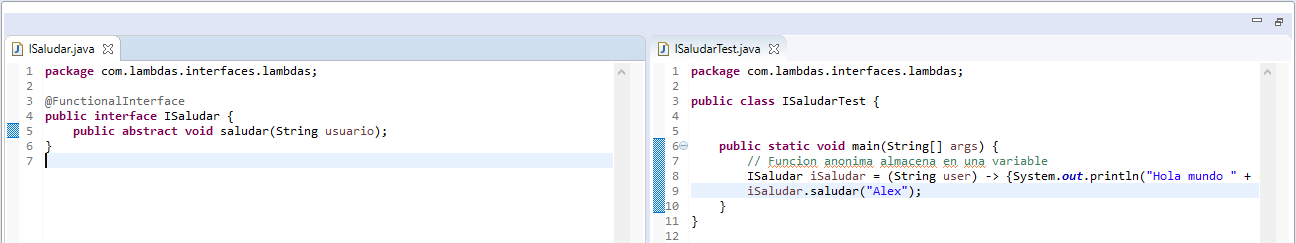
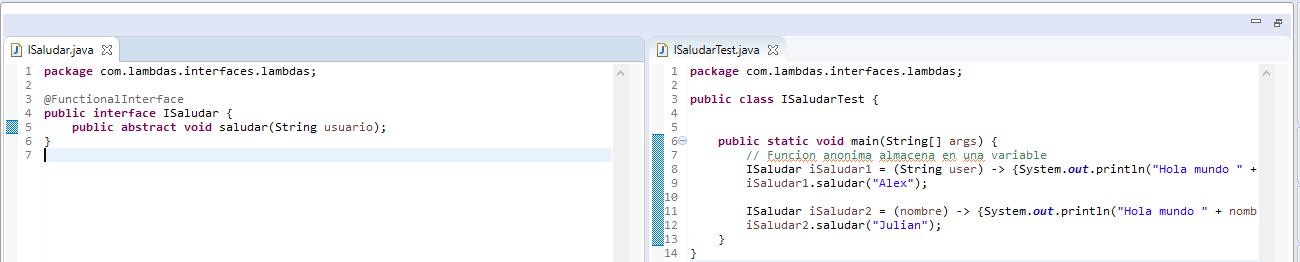
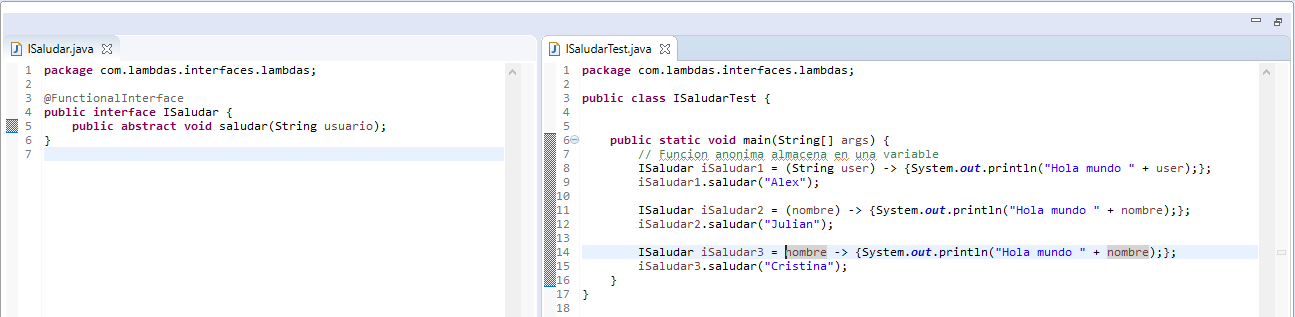
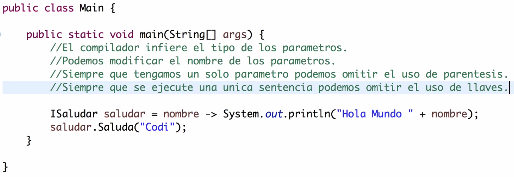
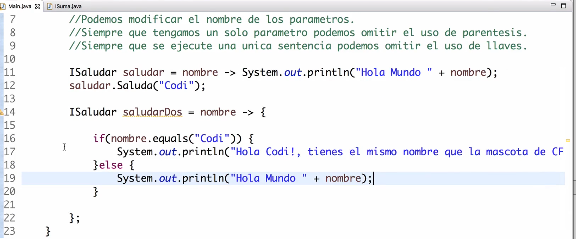
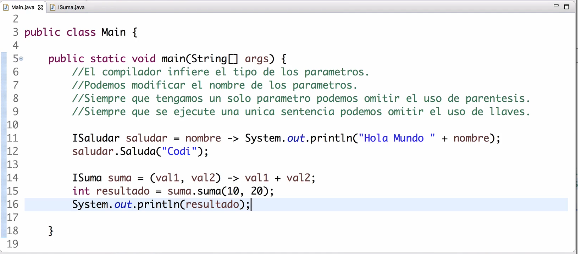
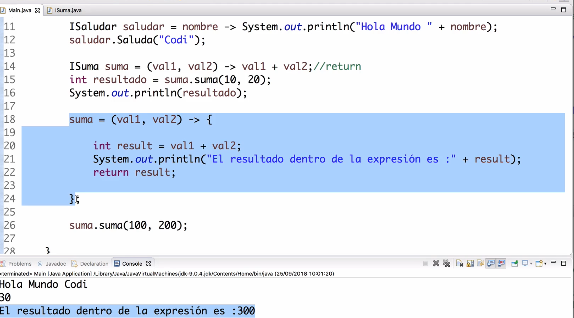
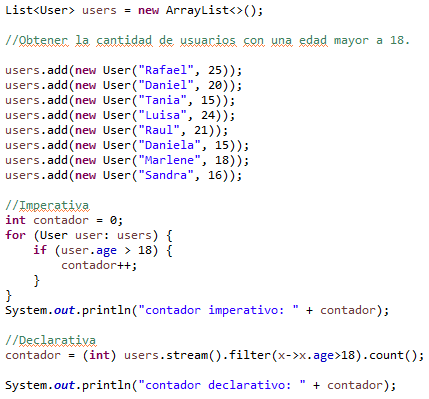
int compare (T o1, T o2);

}

* Una interfaz funcional por tanto contendrá el método propio de su interfaz, así como copias abstractas de todos los métodos de la clase “Object”
* Una expresión lambda puede ser usada en cualquier sitio que se espera una instancia de una clase “Single Abstract Method” (ni siquiera de una @FunctionalInterface, que es un tipo particular de SAM)
* Las interfaces SAM que llevan la anotación @FunctionalInterface son las que Java espera o recomienda que implementes por medio de lambdas
* Java lambdas: target typing
  + Para que una expresión lambda sea asignable a una interfaz T:
    - T debería ser una interfaz funcional
    - La expresión lambda tiene que tener el mismo número de parámetros que el método abstracto de T y los tipos deben ser compatibles (en caso necesario, siempre que sea posible el compilador podrá inferir los tipos a partir del método abstracto).
    - El tipo del valor de retorno de la lambda debe ser compatible con el del método abstracto de T.
    - Si el cuerpo de la lambda lanza (throw) alguna excepción, estas deben ser compatibles con las que lanza el método abstracto de T.
* 
* Java lambdas: ¿Funciones como parámetros?

Programación funcional

Tutorial: I:\VT\Java\Java8\Curso de programación funcional en Java

* Programación funcional se enfoca en el declarativo en la que estamos haciendo y no como lo está haciendo
* Funcional: se delega el control de flujo a funciones,
* Conceptos de funciones puras:
  + Las funciones puras, son funciones las cuales dado el mismo input, siempre retornan el mismo output, además de no tener efectos secundarios.
  + 
* Composición de funciones
  + La composición de funciones es el proceso de combinar dos o más funciones, teniendo como finalidad ejecutar cada una de estas funciones en secuencia para obtener un resultado en concreto.
  + 
* Estados compartidos
  + El estado compartido es cualquier variable, objeto o espacio de memoria que exista en un ámbito compartido.
  + 
* Mutabilidad,
  + Un objeto inmutable es aque que no puede ser modificado una vez que haya sido creado.
    - Podemos ver a los objetos inmutables como constantes, una vez asignado un valor a estos no podrán ser modificados.
  + Una variable en java no es un objeto inmutable
* Efecto secundario
  + Un efecto secundario es cualquier cambio de estado en la aplicación que sea observable fuera de la función llamada.
  + 
    - No es una función pura, ya que esta posee efectos secundarios, modifica el valor de una variable externa,
* Interfaces funcionales
  + Una interface funcional se define como una interfaz que tiene uno y solo uno método abstracto, es decir que define un método que hacer pero no como,
  + A partir del método abstracto debemos definir el comportamiento a partir de una expresión lambda, a partir de una función anónima
    - Dentro de la interface se puede tener varios métodos default o métodos estáticos, pero siempre únicamente un método abstracto.
    - La anotación @FunctionalInterface define que una interface es funcional.
    - Una expresión lambda tiene dos partes, los parámetros y el cuerpo.
    - Una expresión lambda es una función anónima almacenada en una variable
    - 
    - Para utilizar la expresión lambda definida se debe llamar al método abstracto del objeto de la interface.
    - Expresiones lambdas con parámetros
    - 
      * En el apartado de parámetros de la expresión lambda se puede omitir los tipos de datos de los parámetros
      * El compilador infiere el tipo de los parámetros, no es que java es dinámico.
      * Los parámetros los podemos nombrar de la manera que nosotros deseemos, no es obligatorio que tenga el parámetro tenga el mismo nombre que el parámetro en la interface’s .
    - 
    - Cuando la expresión lambda tiene un solo parámetro, se puede omitir el uso de los paréntesis.
    - Cuando se trabaje con expresiones las cuales ejecuten una solo sentencia se puede omitir el uso de llaves.
    - 
    - 
    - Expresión lambda con un tipo de retorno en la función lambda.
    - 
    - Si la expresión lambda tiene una sola sentencia el return es implícito, si la expresión lambda va a tener mas de una sentencia se hace el uso de llaves y debe tener la sentencia return;
    - 
  + Programación imperativa o declarativa.
    - Programación imperativa
      * Es un paradigma de la programación
      * Describe como la computadora debe de realizar una tarea.
      * Describe cada uno de los pasos a seguir,
      * En la programación imperativa describe paso a paso todo lo que la computadora tiene que hacer para llegar a una solución (Ofrece una solución), un conjunto de instrucciones que deben ejecutarse para variar el estado del programa, y de esta manera hallar una solución. Con este paradigma implementar un algoritmo para desarrollar software
      * Un programa imperativo, es una fórmula que tiene pasos escritos por el programador, pasos que cambiar el estado del programa con el fin de llegar a obtener un resultado.
    - Programación declarativa
      * Es un paradigma de la programación, este es todo lo contrario a la programación imperativa.
      * En este paradigma se describe el problema en vez de ofrecer una solución.
      * La programación declarativa provee el “que se de hacer”, peso no el “cómo hacer“.
        + La programación declarativa provee el “que”, pero deja el “cómo” liberado a la implementación particular del intérprete.
        + La programación declarativa hace uso de la programación funcional ya que delega el uso de ciclos y condiciones a funciones.
    - Ejemplos
      * 
  + Api Stream
    - Trabajo con colecciones, ya sea un arreglo, una matriz, una lista, las colecciones permiten guardar diferentes tipos de datos para ser procesados.
    - Al trabajar con colecciones se pueden realizar tareas muy comunes, por ejemple buscar elementos, filtrar elementos, o agrupar elementos, estas tareas son muy comunes que a partir de la versión 8 de Java se incorpora la abstracción Stream API, lo cual permite trabajar con streams.
    - La stream son una abstracción de interface, los cuales permiten procesar información de una colección, trabajando con un alto nivel de abstracción,
    - Se puede definir a un stream como una secuencia de elementos de un origen, que admiten operaciones concatenadas, al trabajar con los streams se puede trabajar con operaciones estilo SQL, se puede filtrar, agrupar, sumar, contar, etc.
    - Los streams trabajan con las colecciones de dos maneras:
      * Estructura de proceso
        + Cuando se refiere a Estructura de proceso, significa que el stream generara un nuevo stream, de esta manera se puede concatenar la ejecución de acciones.
      * Iteración interna
        + Cuando se refiere a Iteración interna, este quiere decir que el stream internamente estará realizando varias acciones de tal manera que al obtener un resultado en concreto.
      * Tipos de métodos como trabajar con Streams
        + Metodos que trabajan con Estructura de proceso

Los métodos que devuelven nuevos Stream, es decir métodos que trabajan con la colección mediante una estructura de proceso. Que un método retorne un nuevo Stream es practico ya que con ello se puede concatenar la ejecución de diferentes métodos, por ejemplo ejecutar un filtro son un filtro ejecutado, únicamente concatenamos el siguiente filtro.

Son los métodos de devuelven un nuevo Stream.

* + - * + Métodos que trabajan con Iteración interna

El método count devuelve un valor de tipo long, este otro de tipo de método de devuelven un valor en concreto, a todos estos métodos que devuelve un valor en concreto diremos que trabajan las colecciones mediante una Iteración interna ya que toman la colección y realizan ciertos procesos para obtener como resultado un dato en concreto.

Son los métodos de devuelven un valor en concreto.

* + - Metodos de la abstracción Stream
      * users.stream():
        + Este método nos devuelve un nuevo stream, una nueva abstracción, este nuevo stream posee todos los elementos de nuestra colección, con este nuevo stream podemos ejecutar y concatenar sus diferentes métodos.
        + Existen diferentes formas de generar nuevos Streams:

Generalmente a través de una colección

Hace referencia a todos los objetos que implementan la interface List, ya sea un Arraylist, un LinkedList, etc.

Stream<User> stream1 = users.stream();

A partir de un Stream ya existente.

Stream<User> stream2 = stream1.filter(x -> x.age > 18);

A partir de un arreglo

Integer[] numbers = {1,2,3,4,5};

Stream<Integer> stream3 = Stream.*of*(numbers);

O partir de una secuencia de datos

Stream<String> stream4 = Stream.*of*("Juan","Pablo","Daniel");

* + - * users.stream().count():
        + Este método retorna la cantidad de elementos que tiene el stream.
        + El método devuelve un valor numérico long, que puede ser almacenado en una variable.
      * users.stream().filter():
        + Este método permite filtrar elementos de una colección,
        + El filtro se hará a través de un predicado, en este caso el predicado será una expresión lambda.

En el apartado de parámetros de la expresión lambda, solo vamos a colocar un solo parámetro, que hace referencia a un objeto, a un elemento de la colección.

El parámetro es del tipo de los elementos de la colección.

En el cuerpo de la expresión lambda se debe poner una expresión booleana, ya sea verdadero o false, o mejor a un poner una sentencia la cual retorne un valor boolean,

Ejemplo: users.stream().filter(x -> x.getAge() > 18)

* + - * + El método filter devolverá un nuevo Stream con los elementos filtrados a través del predicado que devuelva true en el cuerpo de la expresión lambda.

Ejemplo: users.stream().filter(x -> true)

Ejemplo: users.stream().filter(x -> false)

* + - * + El método filter devuelve una abstracción, un nuevo Stream, al retornar esta nueva abstracción (Que implementa una Interface), se puede concatenar la ejecución de diferentes métodos, por ejemplo podemos ejecutar un filtro sobre un filtro previamente ejecutado.

System.***out***.println(users.stream().filter(x->x.age>18));

java.util.stream.ReferencePipeline$2@816f27d

* + - * + La ejecución de los métodos Stream se lo hace descendente,
      * Listas
        + Obtener una lista de una abstracción Stream utilizar el método collect(), este método retornara un objeto de forma puntual, retornara algo en concreto.
        + Este método collect() recibe como argumento un collector: Collectors.toList()

collect( Collectors.toList()), y de esta forma obtenemos una lista como resultado.

* + - * Transformar valores
        + Para modificar el valor de los elementos de la colección en ese caso utilizar el método map().
        + El método map nos permite convertir objetos del Stream en objetos diferentes.
        + El método map recibe como argumeto un Function, en este caso una expresión lambda, en este argumento se debe colocar las condiciones para convertir un objeto a otro.
        + Esta expresión lamba únicamente tiene un parámetro, y el parámetro hará referencia a un elemento de la lista, en el cuerpo de la expresión lambda debemos retornar el nuevo valor, será a partir de ese nuevo valor que se construirá un nuevo stream, el método map retornara un nuevo Stream que es generado a partir de los nuevos datos retornados.

//Problematica: Obtener el cuadrado de todos los numetos en la lista

List<Integer> numeros = Arrays.*asList*(1,2,3,4,5);

List<Integer> result = numeros.stream().map(x -> x\*x).collect(Collectors.*toList*());

System.out.println(result); [1, 4, 9, 16, 25]

* + - * + Al hacer uso del método map, o cualquier método de Stream, no se modifica la colección original
      * Metodo FlatMap
        + Este método permite combinar múltiples colecciones de datos de tal forma que obtengamos como resultado final una única colección.
        + Utilizar Stream.of(…), como argumento utilizar las tres colecciones, el método of devuelve un nuevo Stream con tres elementos y cade elemento es una List<Integer>, todavía no es una lista unificada, para unificar en una sola lista utilizar el método flatMap que recibe como argumento una function una expresión lambda, esta expresión lambda tendrá un solo parámetro, parámetro que hará referencia a cada uno de los elementos de cada colección, dentro del cuerpo de la expresión lambda colocar el parámetro x.stream(), el método flatMap devuelve un nuevo Stream el cual posee todos los elementos unificados

//Problematica: Obtener una lista unificada con Stream.flatMap

List<Integer> numeros1 = Arrays.*asList*(1,2);

List<Integer> numeros2 = Arrays.*asList*(3,4);

List<Integer> numeros3 = Arrays.*asList*(5,6);

List<Integer> result = Stream.*of*(numeros1, numeros2, numeros3)

.flatMap(x->x.stream())

.collect(Collectors.*toList*());

System.***out***.println(result);

// [1, 2, 3, 4, 5, 6]

* + - * Metodo forEach
        + Para realizar operaciones sobre los elementos de una colección, sobre los elementos de un Stream, debemos utilizar el método forEach() que permite trabajar de forma individual con cada uno de los elementos de la colección.
        + El método forEach recibe como argumento una acción Consumer<?>, una expresión lambda, esta expresión lambda tendrá un solo parámetro, parámetro que hará referencia a cada uno de los elementos de cada colección, dentro del cuerpo de la expresión lambda se debe poner las sentencias que se ejecuten para dicho elemento, ya que permite realizar ciertas acciones sobre cada uno de los elementos de la colección.
        + Si la expresión lambda ejecuta varias expresiones debe trabajar con llaves x->{}, si ejecuta una solo sentencia no es obligatorio utilizar las llaves {}.

List<Integer> numeros = Arrays.*asList*(1,2,3,4,5);

numeros.stream()

.map(x->x\*x)

.forEach(x-> System.***out***.print(" " + x));

* + - * Encontrar elementos
        + Para saber si un elemento se encuentra en el Stream se puede utilizar el método anyMatch.
        + anyMatch recibe como parámetro un Predicate, recibe como argumento una expresión lambda, esta expresión lambda tendrá un solo parámetro, parámetro que hará referencia a cada uno de los elementos de cada colección, dentro del cuerpo de la expresión lambda se debe poner una expresión booleana, ya sea true o false, o mejor poner una sentencia la cual retorne un valor boolean.
        + anyMatch devuelve verdadero o false dependiendo si por lo menos si uno de los elementos retorna true en el cuerpo de la expresión lambda, en caso contrario retorna false.

List<Integer> calificaciones = Arrays.*asList*(10,10,9,6,6,6,7,8,9,10);

**boolean** band1 = calificaciones.stream()

.filter(x->x<6)

.count() > 0;

System.***out***.println(band1);

**boolean** band2 = calificaciones.stream().anyMatch(x->x<5);

System.***out***.println(band2);

* + - * Buscar y obtener
        + Para buscar y obtener un elemento del Stream se puede utilizar el método findAny()
        + Para encontrar un elemento se puede hacer uso del método filter() para reducir el universo de datos, con el predicado a evaluar, y obtener los elementos que cumplan con el criterio de búsqueda, para obtener uno de los elementos del filtro podemos aplicar findAny() para devuelva uno de los elementos del resultado del filtro(), sin importar su orden, sin importar que sea el primero, el ultimo, el del medio, se puede hacer uso del método findAny().
        + El método findAny() retorna una abstracción Optional<?> que posee un solo elemento Optional<?>, para obtener el dato en concreto podemos utilizar el método get() de Optional<?> que retornara el elemento previamente obtenido.

User user = users.stream().filter(x->x.age==25).findAny().get();

**if** (user != **null**) {

System.***out***.println("contador declarativo: " + user.getUsername());

}

* + - * + Si se quiere devolver el primer elemento de la colección se puede usar el método findFirst().
        + El método Optional.get() obliga a devolver un valor, este método lanza una excepción “java.util.NoSuchElementException” cuando no se encuentre un elemento solicitado, sin embargo si ninguno de los elementos cumple el filtro, no existe valor alguno a devolver, es por eso que se lanza el error, porque en la mayoría de los casos no vamos a conocer si existe el elemento en la colección.
        + Para controlar el error de Optional.get() se puede utilizar el método orElse() en vez de get(). Ese método orElse() nos retorna un elemento de la colección si exite, en caso contrario se puede retornar un valor por default del tipo de objeto de la colección, ese elemento default se lo define en el argumento del método orElse(.), este método orElse() es una forma más segura de obtener un elemento ya que este método no dispara ningún tipo de excepción.

User userDefault = **new** User("Sin nombre", 0);

//Declarativa

User user = users.stream()

.filter(x->x.age==30)

.findAny()

//.get()

.orElse(userDefault);

**if** (user != **null**) {

System.***out***.println("contador declarativo: " + user.getUsername());

}