Actualizado: **20 de septiembre de 2022**

Apunte sobre el uso de colecciones en Java, lambda expresiones y Streams

El objetivo de este apunte es ofrecer una primera visión sobre los temas mencionados. Está pensado como una guía introductoria para aquellos alumnos que no estén familiarizados con el uso de las colecciones en Java y presenta una variedad de ejemplos que serán útiles para el desarrollo de los ejercicios de la cursada.

Colecciones

Java provee un framework de colecciones muy rico conformado por un conjunto de interfaces y clases que las implementan. En este apunte, solamente vamos a describir una de ellas: **ArrayList**.

Una instancia de la clase java.util.ArrayList constituye una colección lineal ordenada, que crece dinámicamente y puede contener elementos duplicados. Ofrece acceso posicional en tiempo constante. Está indexada comenzando desde cero; es decir que el primer elemento de la lista se encuentra en esa posición.

Permite agregar y eliminar elementos. Cada vez que se agrega un elemento, la lista lo ubica en la última posición; no obstante esto, pueden agregarse elementos en otras posiciones, lo cual genera el desplazamiento de una posición en sentido creciente de todos los elementos que se encuentren ubicados a partir del lugar en el que se agrega el nuevo. Pueden eliminarse elementos de cualquier posición de la lista. Cuando esto ocurre, la lista genera un desplazamiento en sentido contrario al anteriormente descrito con el fin de no dejar espacios vacíos (por supuesto que este desplazamiento no ocurre cuando se elimina el último elemento de la lista).

Definición y Creación

Java permite incluir en el tipo de las colecciones, el tipo de los elementos que estas contienen. Este tipo podrá ser una clase o una interfaz, pero no un tipo primitivo (para esto último deberían utilizarse wrappers).

Por ejemplo, un ArrayList que contendrá instancias de la clase Alumno se define y crea de la siguiente manera:

```
List<Alumno> alumnos = new ArrayList<Alumno>();
```

Este tipado permitirá al compilador chequear que no se agregue a la lista ningún elemento que no esté tipado con la clase Alumno o alguna de sus subclases, lo cual resulta en un uso más seguro.

Notar que en este caso, la variable se declara utilizando la interface List permitiendo desacoplar la lista de una implementación particular.

En general, el tipo de los elementos de la colección se especifica entre "<" y ">", de la siguiente manera:

```
List<T> items;
```

en donde "T" indica el nombre de una clase o interfaz. Por lo tanto la declaración anterior indica que "items" será una instancia de ArrayList que contendrá objetos de tipo "T".

Protocolo de uso

Entre los mensajes más utilizados de ArrayList se encuentran los siguientes:

add(<T> object)
Agrega el objeto "object" , de tipo "T", a la última posición de la lista.

add(int index, <T> object)
Agrega el objeto "object", de tipo "T", en la posición indexada con index (recordar que la primera posición es la cero). Si index no denota la última posición sino una intermedia, se realiza un corrimiento de los elementos existentes para hacer lugar al nuevo elemento.

get(int index)
Devuelve el elemento que se encuentra en la posición indicada por index.
Esto NO elimina el elemento de la lista.

size()
Devuelve la cantidad de elementos que contiene la lista. Este número siempre superará en uno al índice de la posición del último elemento (puesto que el índice de las listas comienza en cero).

contains(<T> object)
Devuelve true si la lista contiene el elemento "object", y false en caso contrario.

remove(int index)
Elimina el elemento contenido en la posición indicada por index. Esto genera un

corrimiento de los elementos ubicados en las posiciones superiores, de forma de no dejar lugar vacío.

clear()
Elimina todos los elementos de la lista, dejándola vacía.

isEmpty()
Devuelve true si la lista está vacía, y false en caso contrario.

Expresiones Lambda

A partir de Java 8, se incorporó el soporte a expresiones lambda. Las expresiones lambda son **funciones anónimas** que no pertenecen a ninguna clase y son utilizadas porque necesitamos utilizar una funcionalidad una única vez. Normalmente, creamos expresiones lambda con el mero propósito de **enviarla como parámetro** a una función de alto orden (se entiende como función de alto orden a una función que recibe como parámetro a otra función).

El potencial de las expresiones lambda es enorme, y se utiliza en distintos escenarios a lo largo de una aplicación.

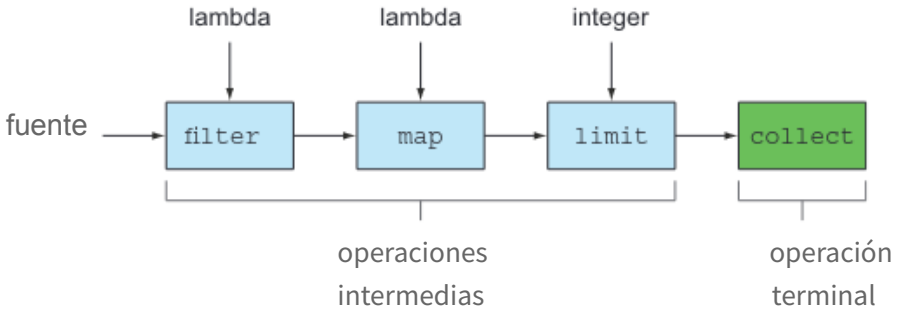
La sintaxis de una expresión lambda es:

```
(parámetros, separados, por, coma) -> { cuerpo lambda }
```

1. El operador lambda (->) separa la declaración de parámetros de la declaración del cuerpo de la función.
2. Parámetros:
    - Cuando se tiene un solo parámetro no es necesario utilizar los paréntesis.
    - Cuando no se tienen parámetros, o cuando se tienen dos o más, es necesario utilizar paréntesis. Si hay más de un parámetro, es necesario separarlos con coma.
3. Cuerpo de lambda:
    - Si el cuerpo de la expresión lambda tiene una única línea no es necesario utilizar las llaves y no necesitan especificar la cláusula return en el caso que retorne algún valor.
    - Si el cuerpo de la expresión lambda tiene más de una línea es necesario utilizar las llaves y es necesario incluir la cláusula return en el caso que retorne algún valor .

API Stream

Streams es una API de Java que nos va a facilitar el manejo de colecciones. Esto nos permite hacer operaciones como buscar, filtrar, sumar elementos de una colección de una forma sencilla. La API Stream define una amplia variedad de operaciones con la intención de permitir su encadenamiento, de la siguiente forma.



En la figura anterior puede verse un ejemplo de su uso. A partir de la fuente, en nuestro caso será una colección, se aplicarán una serie de operaciones para obtener el resultado deseado. Vamos a utilizar streams junto con las expresiones lambda para operar con las colecciones de objetos.

Las operaciones definidas en la API se pueden clasificar en **dos categorías**: operaciones intermedias y operaciones terminales. Las **operaciones intermedias** retornan un stream para permitir el encadenamiento con otras operaciones. Las **operaciones terminales** permiten retornar un tipo diferente a Stream, como una Lista, un Integer o void.

Nota: si no se quiere retornar un stream, siempre se debe utilizar una operación terminal.

Solución tradicional vs solución utilizando streams y lambdas

Por ejemplo, si quisiéramos obtener los alumnos inscriptos a partir del 2020.

Solución tradicional:

```
List<Alumno> alumnosInscriptosDespues2020 =  new ArrayList<Alumno>();
for(Alumno alumno: alumnos){
    if(alumno.getAñoIngreso() > 2020){
          alumnosInscriptosDespues2020.add(alumno);
    }
}
```

Solución con Streams y expresiones lambda:

```
List<Alumno> alumnosInscriptosDespues2020 = alumnos.stream()
             .filter(alumno -> alumno.getAñoIngreso() > 2020)
             .collect(Collectors.toList());
```

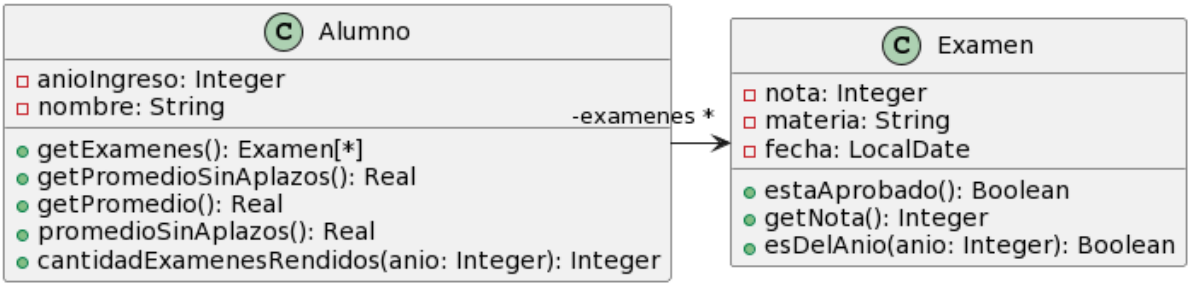
En este apunte se presentan ejemplos de las operaciones de Streams indicadas en las siguientes tablas. Podrá ver cómo ordenar una colección, sumarla, encontrar el objeto que maximiza o minimiza un criterio, encontrar el primer objeto que cumple con un predicado, calcular el promedio de una colección, filtrar una colección o ejecutar una misma operación para cada uno de los objetos de la colección.

Operaciones intermedias
filter
map
limit
sorted

Operaciones terminales
count | sum
average
findAny | findFirst
collect
anyMatch | allMatch | noneMatch
min | max

Ejemplos prácticos:

Vamos a ver en detalle algunas de las operaciones de Streams siguiendo un ejemplo. Se define la clase Alumno donde cada alumno conocerá los exámenes que ha rendido.



Estos primeros cinco ejemplos son sobre las operaciones intermedias. Notar que siempre se aplica la operación terminal collect, solamente para dejar funcionando el ejemplo, pero podría aplicarse otra operación intermedia y luego la operación final.

filter

Permite filtrar los elementos de un stream según el predicado que recibe como parámetro.
Por ejemplo, queremos la lista de los alumnos que ingresaron en un año en particular

```
List<Alumno> ingresantesEnAnio2020 = alumnos.stream()
              .filter(alumno -> alumno.getAnioIngreso() == 2020)
              .collect(Collectors.toList());
```

map | mapToDouble | mapToInt

Genera un stream, de igual longitud que el original, reemplazando cada elemento con el resultado de aplicar la función que recibe como parámetro sobre cada elemento del stream original.
Ejemplo: se quiere obtener la lista de los nombres de los alumnos.

```
List<String> nombresDeAlumnos = alumnos.stream()
              .map(alumno -> alumno.getNombre())
              .collect(Collectors.toList());
```

Cuando trabajamos con tipos primitivos debemos considerarlo de la siguiente manera:

```
examenes.stream()
     .mapToInt(examen -> examen.getNota())

examenes.stream()
     .mapToDouble(alumno -> alumno.getPromedio())
```

limit

Trunca el stream dejando los primeros elementos.
Ejemplo: se quiere una cantidad fija de los alumnos

```
List<Alumno> primeros3alumnos = alumnos.stream()
              .limit(cantidad)
              .collect(Collectors.toList());
```

sorted

Genera un stream con los elementos ordenados.
Ejemplo: se quieren ordenar los alumnos por promedio en forma descendente.

```
List<Alumno> alumnosOrdenadosPorPromedioDesc = alumnos.stream()
         .sorted((a1, a2)->Double.compare(a2.getPromedio(), a1.getPromedio()))
         .collect(Collectors.toList());
```

count

Retorna la cantidad de elementos en el stream. El tipo de retorno es long. Debe hacerse la conversión a int de ser necesario.

Por ejemplo: se quiere obtener la cantidad de alumnos con un promedio mayor a cierta nota.

```
int cantidadDeAlumnosConPromediosMayorA7 = (int) alumnos.stream()
          .filter(alumno-> alumno.getPromedio() >= 7 )
          .count();
```

sum

Retorna un número que es la suma de los elementos del stream. Este debe contener números (DoubleStream, IntStream...)
Ejemplo: se quiere retornar la cantidad de exámenes tomados en un año dado

```
int totalExamenesTomadosEn2020 = alumnos.stream()
              .mapToInt(alumno -> alumno.cantidadExamenesRendidos(2020))
              .sum();
      }
```

average

Retorna un Optional con el promedio de los elementos de un stream. El objeto Optional no tiene valor si el Stream está vacío. El stream debe ser de números (DoubleStream, IntStream...).
Por ejemplo: se quiere obtener el promedio de un alumno

```
double getPromedio = examenes.stream()
              .mapToDouble(e -> e.getNota())
              .average().orElse(0);
      }
```

findFirst | findAny

Retorna un Optional con alguno de los elementos de un stream. Hay que manejar el caso de que la lista esté vacía.
Por ejemplo: se quiere obtener el primer alumno cuyo nombre comience con una letra.

```
Alumno primerAlumnoConLetraM = alumnos.stream()
                .filter(alumno -> alumno.getNombre().startsWith("M"))
                .findFirst().orElse(null);
```

collect

Lo utilizaremos para convertir los elementos de un stream en una lista. Ya vimos varios ejemplos en los casos de las operaciones intermedias.

max | min

Retorna un Optional con el elemento máximo del stream de acuerdo al comparador indicado como parámetro. Retornará null si no hay elementos en la variable alumnos.

```
Alumno mejorPromedio = alumnos.stream()
       .max((a1, a2)->Double.compare(a1.getPromedio(), a2.getPromedio())).orElse(null);
```

Ejemplos Ejecutables en Java

Para poder observar el funcionamiento concreto de todo lo descripto hasta aquí, se proveen ejemplos ejecutables en Java. Para permitirle a usted no sólo observar la ejecución de estas operaciones, sino proveerle una base que le posibilite experimentar otras operaciones y escenarios, los ejemplos se han implementado en forma de Tests de Unidad. En sus pruebas ud podrá generar nuevos tests o modificar los existentes. Por ejemplo: ¿qué ocurre si intento obtener un elemento de una posición mayor a la última?, o ¿qué ocurre si intento agregar un elemento en una posición mayor a la última?

El código Java entregado tiene tres clases:

Alumno: es una clase cuyas instancias se utilizarán como elementos de la lista.

Examen: Cada alumno tiene asociado los exámenes que ha rendido.

CollectionTest: es la clase que define los Tests de Unidad. Para esto define un escenario básico de prueba (en el método setUp()) que se utilizará para todos los tests, y un conjunto de tests que ejecuta distintas operaciones de la lista y se asegura que los resultados sean los esperados.
Tanto en el método setUp() como en los métodos de tests se podrá observar el funcionamiento de la lista descripto anteriormente.

El código con el proyecto Maven con los ejemplos puede descargarlo desde AQUÍ.