Introducción a la Programación

Tema 5. Tratamientos Secuenciales

1. Introducción

Al tratar con agregados de datos (listas, conjuntos, arrays, cadenas de caracteres, secuencias de valores y en general todos los tipos que implementen la interfaz *Iterable<E>* o *Stream<E>*) es muy común buscar información sobre cantidades agregadas: máximos o mínimos, posiciones de elementos en el agregado, etc. Estos tratamientos se repiten continuamente. Nuestro objetivo en este tema es estudiar estos tratamientos sobre agregados de datos, hacer un catálogo lo más extenso posible de los mismos y, a ser posible, implementarlos en métodos que puedan ser reutilizados.

Java 8 ya proporciona, de forma muy eficiente y clara, la mayoría de estos métodos asociados al tipos de datos *Stream<E>* pero aquí queremos desarrollar estas ideas para poderlas implemenar en lenguajes que dispongan de ese tipo de datos o simplemente para comprender su funcionamiento interno.

Dado un agregado de objetos podemos plantearnos preguntas del tipo:

- ¿cuánto vale la suma de una expresión de los objetos del agregado?
- ¿y el producto de los que verifican una propiedad?
- ¿existe algún objeto que verifique una determinada condición?
- ¿cuáles son los objetos que cumplen una determinada propiedad?
- ¿cuál es el valor más grande de los que cumplen una condición?
- ¿según un orden dado cuál es mayor elemento para una condición?
- actualiza todos los objetos que cumplen una condición
- almacena en una lista los objetos que cumplan una condición
- etc.

Cada una de esas preguntas y otras similares pueden ser respondidas con métodos que tienen una estructura común.

Ejemplo: ¿Cómo se calcula la suma de los cuadrados de los números enteros contenidos en una lista de Enteros?

Como el agregado es una lista usamos un *for* extendido para recorrerla. El código lo concretamos en un método que tomando una lista como parámetro y devuelve el resultado esperado.

```
public static Integer sumaCuadrados(List<Integer> v){
   Integer suma=0;
   for (Integer e: v){
        if(e%3==0){
             suma = suma + e*e;
        }
    }
    return suma;
}
```

Igualmente podemos preguntarnos por la suma de los cuadrados de todos los enteros que van desde un entero dado, a, hasta otro b (mayor que a) en pasos de c (positivos). Por ejemplo la secuencia 10, 12, 14, 16, 18, 20 sería especificada por a=10, b=20, c=2.

Para generar la secuencia usamos un *for* clásico. El método resultante es:

```
public static Integer sumaCuadradosIntervalo(Integer a, Integer b,
   Integer c){
   if(!(b>a && c>0)) throw new IllegalArgumentException();
   Integer suma=0;
   for (Integer e=a; e<=b; e=e+c){
         suma = suma + e*e;
   }
   return suma;
}
```

El método anterior dispara una excepción si los parámetros recibidos no cumplen lo especificado. Salvando la comprobación de la idoneidad de los parámetros ambos métodos tienen, como vemos, la misma estructura.

Veamos ahora un problema más general: sumar los cuadrados de los enteros contenidos en una lista que son múltiplos de 3.

```
public static Integer sumaCuadradosMultiplos3(List<Integer> v){

   Integer suma=0;
   for(Integer e: v){
       if(Enteros.esMultiplo(e,3)){
           suma = suma + e*e;
       }
   }
   return suma;
}
```

O en la secuencia definida por *a, b, c*. Es decir los enteros desde *a* hasta *b* con incrementos de *c*.

```
public static Integer sumaCuadradosIntervaloMultiplos3(Integer a,
   Integer b, Integer c){
```

```
    if(!(b>a && c>0))
            throw new IllegalArgumentException();
    Integer suma=0;
    for(Integer e= a; e<=b; e=e+c){
        if(Enteros.esMultiplo(e,3)){
            suma = suma + e*e;
        }
    }
    return suma;
}
```

En estos métodos aparece una estructura que se repetirá. Esta estructura tiene varios elementos a destacar: fuente de datos y mecanismo de recorrido de los datos, acumulador, filtro, transformación, criterio de parada.

- **Fuente de datos:** Es el origen de los datos. Puede ser una variable de un tipo agregado de datos (Set, List, …), un agregado virtual de datos (los enteros pares entre los valores a y b, …), una secuencia descrita por un primer elemento y el siguiente de uno dado, etc. El tipo del agregado será Ag y asumimos que los elementos que lo componen son de tipo *E*.
- **Mecanismo de generación de los datos:** La forma concreta de genera uno tras otro los datos del agregado. Los elementos en una lista se pueden recorrer hacia arriba o hacia abajo. Una posibilidad es generar los índices de la lista con un for y posteriormente obtener los contenidos de las casillas. Los enteros pares entre a y b pueden ser generados mediante un for, etc.
- **Acumulador**: Es una variable que *acumula* el resultado calculado hasta ese momento. Un acumulador es una variable que es de un tipo *A* que hay que especificar. Necesita un valor inicial y una función *acumula(a,e)* capaz de acumular el elemento *e* en el acumulador a. Dependiendo que el tipo *A* sea mutable o inmutable la función acumula tendrá un diseño u otro. Un acumulador concreto es *suma.* En ese caso *A* es un tipo numérico, se inicializa a cero y la función acumula es de la forma *a = a+e*. Algunos acumuladores tienen, además, un *criterio de parada*. Es una expresión lógica, que podemos denominar *salir(a),* que depende del valor del acumulador y nos indica si el actual valor del acumulador ya podemos saber el valor acumulado final y por lo tanto salir del bucle de la iteración. Veremos ejemplos más abajo
- **Filtro**: es una expresión lógica sin efectos laterales que nos sirve para escoger los elementos del agregado sobre los que vamos a hacer el cálculo. Como nombre general el filtro lo concretaremos en una expresión lógica que llamaremos *filtra(e).* En este caso es la expresión *Enteros.esMultiplo(e,3).*
- **Transformación**: Es una expresión que a partir de cada objeto del agregado calcula otro valor que es el que queremos acumular. Como nombre general la transformación la concretaremos en una expresión que llamaremos *transforma(e)* que puede devolver un tipo *T* distinto a *E.* En este caso la función es *e*e*.

- **Resultado**: Una función que a partir del valor del acumulador devuelve el resultado buscado. Como nombre general la concretaremos en una función que llamaremos *resultado(a)* que devolverá un valor del tipo *R*.

Con estas ideas los esquemas iterativos cuyo objetivo es acumular un valor a partir de un agregado tienen la siguiente forma:

```
R esquemaSecuencial(Ag ag) {
        A a = valorinicial;
        Para cada e en ag {
                if (filtra(e)) {
                        T e1 = transforma(e);
                        a = acumula(a,e1);
                }
                if(salir(a)) break;
        }
        return resultado(a);
}
```

En el esquema anterior vemos entonces varios elementos: el acumulador, la fuente de datos y el mecanismo concreto para ir recorriendo los datos de la fuente de datos.

Como vemos el acumulador es una variable de tipo A que tiene un conjunto de operaciones asociadas:

- acumula(a,e)
- salir(a)
- resultado(a)
- valorinicial()

Dependiendo de que el tipo *A* sea mutable o inmutable las funciones anteriores pueden tomar una forma u otra. La funciones anteriores pueden considerar separadas unas de otras, si estamos programando en un lenguaje como C o considerarlas asociadas al tipo A si usamos un lenguaje como Java u otro orientado a objeto.

El mecanismo de recorrido de los datos debe proporcionar los datos uno tras otro. Abajo veremos ejemplos y posibles abstracciones de este mecanismo. Una abstracción de este mecanismo es el tipo *Iterable<E>* ofrecido por Java y otros lenguajes. Este tipo ofrece dos operaciones básicas:

- E siguienteElemento(e)
- boolean esElUltimoElmento(e)

Frente al esquema secuencial anterior podemos pensar en esquema paralelo de acumular los datos de un agregado. Para ello necesitamos que el mecanismo de generar los datos se dote

de la operación adicional de partir los datos disponibles en dos partes y el acumulador de la operación *A combina(a1,a2)* capaz de combinar dos valores del acumulador.

El tipo *Stream<E>* en java ofrece la abstracción adecuada para el procesamiento secuencial o paralelo de agregados de datos. Combinado con un acumulador concreto permite diseñar de una forma genérica los algoritmos para acumular valores a partir de un agregado de datos ya sea en forma secuencial o paralela.

En este capítulo nos concentraremos en los esquemas secuenciales y los detalles de su implementación en diferentes tipos de lenguajes.

En los que sigue vamos a usar el tipo *Optional<T>* proporcionado en el API de Java 8. Las instancias de este tipo son contenedores que pueden contener un valor o alternativamente un *null*. Si un valor está presente, *isPresent ()* devolverá *true* y *get ()* devolverá el valor. Si se intenta conseguir el valor y el contenedor está vacío disparará la excepción *NoSuchElementException*. Tiene dos métodos de factoría:

- *static <T> Optional<T> empty():* que construye un contenedor vacío.
- *static <T> Optional<T> of(T e):* que construye un contenedor no vacío con el elemento *e* que debe ser distinto de *null*.

El tipo *Optional<T>* tiene otros métodos que pueden consultarse en el API.

2. Fuentes de datos y mecanismos para la generación de los datos del agregado

Algunas de fuentes de datos que veremos son:

- Secuencias numéricas (de enteros o reales).
- Tablas de una o varias dimensiones
- Listas, conjuntos, …
- El tipo Iterable
- El tipo Stream

2.1 Array

El recorrido de los elementos de un array o cualquier otro agregado indexable sigue el esquema

```
E[] ag = ...;
       ...
for (int i = 0; i < ag.length; i++) {
       T e = ag[i];
       ...
}
...
```

2.2 Arrays de dos dimensiones

Un array de dos dimensiones puede ser recorrido en la forma:

```
E[][] ag = ;
…
for(i = 0; i < ag.length ; i++){
       for(j =0; j < ag[i].length; j++){
              E e = ag[i][j];
              ...
       }
}
```

2.3 Secuencias Numéricas

Por ejemplo la secuencia de *0* a *n-1* de 1 en 1. O la secuencia aritmética de *a* hasta *b* con razón de *c* (siendo *b > a* y *c > 0*). O la secuencia aritmética de *a* hasta *b* con razón de *c* (siendo *a > b* y *c < 0*). O la secuencia geométrica de *a* hasta *b* con razón de *c* (siendo *b > a* y *c > 1*).

```
…
for(i = a; i < b; i = i+c ){     // secuencia aritmética con a <= b, c > 0
   …
}
for(i = a; i > b; i = i-c ){     // secuencia aritmética con a >= b, c > 0
   …
}
for(i = a; i < b; i = i*c ){     // secuencia geométrica con a <= b, c > 1
   …
}
```

2.4 Secuencias de pares

Generación secuencias de pares de enteros *i, j* con *i* desde *a1* hasta *b1* (sin incluir) y pasos de *c1* (*a1 <= b1, c1 > 0*) y *j* desde *a2* hasta *b2* (sin incluir) y pasos de *c2* (*a2 <= b2, c2 > 0*). De la misma forma podríamos generar tripletas *i, j, k*.

```
…
for(i = a1; i < b1; i = i+c1){
   for(j = a2; j < b2; j= j+c2){
          …
   }
}
```

2.5 El tipo Iterable

Cuando un agregado de datos implementa el interface iterable el recorrido en Java puede hacerse de la forma:

```
Iterable<E> ag = ...;
...
for (E e : ag) {
          ...
}
```

Tipos que implementan *Iterable<E>* son *List<E>*, *Set<E>, SortedSet<E>* y muchos más.

2.6 Iterable anidado

Cuando un agregado de datos implementa el interface iterable y cada uno de los elementos tiene una propiedad *P* que también es Iterable entonces el recorrido en Java puede hacerse de la forma:

```
Iterable<T> ag = ...;
...
for (T t : ag) {
   for(E e: t.getP()){
          ...
   }
}
```

Este esquema de generación puede generalizarse a anidamientos mayores

3. Acumuladores

Es conveniente conocer un catálogo de acumuladores de uso muy general. Para cada acumulador se indicará su objetivo, el tipo de la variable de acumulador y los detalles de las funciones que tiene asociado.

También indicaremos la función adecuada para combinar los resultados de dos acumuladores del mismo tipo. Esta función es adecuada para implementar acumuladores que puedan llevar a cabo tratamientos sobre secuencias en paralelo. Si solamente se van a llevar a cabo tratamientos secuenciales la función combina no es necesaria.

3 El catálogo de acumuladores

3.1 Suma (sum)

- Objetivo: Suma de los valores del agregado cuyos elementos deben ser de tipo numérico.
- Tipo del acumulador: *Double, Integer* u otro tipo numérico.
- Valor Inicial: *Cero*.
- Expresión acumuladora: *a = a +e.*
- Combina(a1,a2): *a1+a2.*
- Salir(a): *Falso*
- Resultado(a): a

3.2 Producto (multiply)

- Objetivo: Multiplicar de los valores del agregado cuyos elementos deben ser de tipo numérico.
- Tipo del acumulador: *Double, Integer* u otro tipo numérico.
- Valor Inicial: *Uno*.
- Expresión acumuladora: *a = a*e.*
- Combina(a1,a2): *a1*a2.*
- Salir(a): *Falso*
- Resultado(a): a

3.3 Contador (count)

- Objetivo: Contar los elementos del.
- Tipo del acumulador: Integer.
- Valor Inicial: Cero.
- Expresión acumuladora: a = a+1.
- Combina(a1,a2): *a1+a2.*
- Salir(a): Falso
- Resultado(a): a

3.4 Acumula Lista (toList)

- Objetivo: Devuelve en una Lista los elementos del agregado.
- Tipo del acumulador*: List<E>*
- Valor Inicial: Lista Vacía.
- Expresión acumuladora: a.*add(e)*
- Combina(a1,a2): *e= copy(a1);e.addAll (a2); return e.*
- Salir(a): Falso
- Resultado(a): a

3.5 Acumula Conjunto (toSet)

- Objetivo: Devuelve en un Conjunto los elementos del agregado.
- Tipo del acumulador: Set<E>
- Valor Inicial: Conjunto Vacía.
- Expresión acumuladora: a.add(e)
- Combina(a1,a2): *e= copy(a1);e.addAll(a2); return e.*
- Salir(a): Falso
- Resultado(a): a

De forma similar tendríamos el acumula conjunto ordenado u otros similares

3.6 Acumula Array (toArray)

- Objetivo: Devuelve un para *array*, integer con los elementos del agregado colocados en las casillas del array y el entero el número de elementos del agregado.
- Tipo del acumulador: *(E[] a, i)*
- Valor Inicial: (new E[m], 0). Con m suficiente para asegurar *m* mayor que el tamaño del agregado.
- Expresión acumuladora: (a[i] = e, i++)
- Combina(a1,i1,a2,i2): *(e,v)= copy(a1,i1);(e,v).addAll(a2,i2); return (e,v).*
- Salir(a): Falso
- Resultado(a,i): (a,i)

Las funciones *copy, addAll* son fácilmente implementables.

3.7 Para Todo (all)

- Objetivo: Decide si todos los elementos cumplen una propiedad *p(e).*
- Tipo del acumulador: *Boolean*
- Valor Inicial: true
- Expresión acumuladora:

```
a = p(e);
```

- Combina(a1,a2): *a1 && a2*
- Salir(a): !a
- Resultado(a): a

3.8 Existe (any)

- Objetivo: Decide si alguno de los elementos cumplen la propiedad p(e).
- Tipo del acumulador: Boolean
- Valor Inicial: false
- Expresión acumuladora:

```
a = p(e);
```

- Combina(a1,a2): *a1 || a2*
- Resultado(a): a

3.9 Busca Primero (first)

- Objetivo: buscar, si existe, el primer objeto que cumple una propiedad *p(e).* Si no existe dispara la excepción *NoSuchElementException.*
- Tipo del acumulador: *E*
- Valor Inicial: *null*
- Expresión acumuladora:

```
if(p(e)) a = e;
```

- Combina(a1,a2): *c(a1,a2)*
- Salir(a): *a !=null*
- Resultado(a):

```
(a == null)? Optional.empty(): Optional.of(a)
```

La función *c(a1,i1,a2,a2)* vendría dada por:

```
if(a1 == null && a2 == null) return null;
if(a1 != null) return a1;
if(a2 != null) return a2;
```

3.10 Posición del Primero (index)

- Objetivo: buscar, si existe, la posición del primer objeto que cumple una propiedad p(e). Si no existe devuelve -1.
- Tipo del acumulador: *(a,i): (Integer, Integer)*
- Valor Inicial: *(-1,0)*
- Expresión acumuladora:

```
if(p(e)) a=i;
i++;
```

- Combina(a1,i1,a2,a2): *c(*a1,i1,a2,a2)
- Salir(a): *a != -1*
- Resultado(a): a

La función *c(a1,i1,a2,a2)* vendría dada por:

```
if(a1 == -1 && a2 ==-1) return -1;
if(a1 != -1) return a1;
if(a2 != -1) return a2+i1;
```

3.11 Máximo (max)

- Objetivo: Calcula el máximo de los elementos del agregado. El máximo se calcula con respecto a un orden dado representado por el *Comparator<E> ord*. Si el agregado es vacío devuelve la excepción *NoSuchElementException.*
- Tipo del acumulador: *E*
- Valor Inicial: *null*

- Expresión acumuladora:

```
if(a == null || ord.compare(e,a) > 0) a = e;
```

- Combina(a1,a2): *max(a1,a2)*
- Salir(a): *false*
- Resultado(a):

```
(a == null)? Optional.empty(): Optional.of(a)
```

La función *max* vendría dada por:

```
if(a1 == null && a2 == null) return null;
if(a1 != null && a2 == null) return a1;
if(a1 == null && a2 != null) return a2;
return ord.compare(a2,a1) > 0 ? a2 : a1;
```

3.12 Mínimo (min)

Objetivo: Calcula el mínimo de los elementos del agregado. El máximo se calcula con respecto a un orden dado representado por el *Comparator<E> ord*. Si el agregado es vacío devuelve la excepción *NoSuchElementException.*

- Tipo del acumulador: *E*
- Valor Inicial: *null*
- Expresión acumuladora:

```
if(a == null || ord.compare(e,a) < 0) a = e;
```

- Combina(a1,a2): *min(a1,a2)*
- Salir(a): *false*
- Resultado(a):

```
(a == null) ? Optional.empty() : Optional.of(a)
```

La función *min* vendría dada por:

```
if(a1 == null && a2 == null) return null;
if(a1 != null && a2 == null) return a1;
if(a1 == null && a2 != null) return a2;
return ord.compare(a2,a1) < 0 ? a2 : a1;
```

3.13 Reducir (reduce)

- Objetivo: Acumular los elementos del agregado dados un valor inicial *e0* (que se devolverá si el agregado está vacío) y un operador binario *b(e1,e2).* Este acumulador es una generalización de otros vistos anteriormente como suma, producto
- Tipo del acumulador: E
- Valor Inicial: *e0*
- Expresión acumuladora:

```
a = b(a,e);
```

- Combina(a1,a2): *b(a1,a2)*
- Salir(a): false
- Resultado(a): a

3.14 Agrupar (groupBy)

- Objetivo: Agrupar los elementos del agregado en grupos tras la aplicación de una función *f(e).* Dos elementos estarán en el mismo grupo si al aplicarle la función f resulta el mismo valor. Asumimos que la función f tiene la signatura *T f(E e).*
- Tipo del acumulador: *Map<T,List<E>>*
- Valor Inicial: *new HashMap<T,List<E>>*
- Expresión acumuladora:

```
T v = f(e);
List<E> ls;
if(a.contains(v)){
       ls = a.get(v);
} else {
       ls = new ArrayList<E>();
       a.put(v,ls);
}
ls.add(e);
```

- Combina(a1,a2): *e = copy(a1); e.putAll(a2); return e;*
- Salir(a): false
- Resultado(a): a

El acumulador anterior tiene otras variantes al sustituir el tipo del acumulador por

- *Map<T,Set<E>>*
- *Map<T,SortedSet<E>>*

3.15 Partir (partitioningBy)

- Objetivo: Dividir los elementos del agregado en dos grupos: los que cumplen y los que no cumple un predicado *p(e).*
- Tipo del acumulador: *Map<Boolean,List<E>>*
- Valor Inicial:

```
a = new HashMap<Boolean,List<E>>;
a.put(true, new ArrayList<E>());
a.put(false, new ArrayList<E>());
```

- Expresión acumuladora:

```
Boolean v = p(e);
List<E> ls; = a.get(v);
ls.add(e);
```

- Combina(a1,a2): *e = copy(a1); e.putAll(a2); return e;*
- Salir(a): false
- Resultado(a): a

El acumulador anterior tiene otras variantes al sustituir el tipo del acumulador por

- *Map<Boolean,Set<E>>*
- *Map<Boolean,SortedSet<E>>*

3.16 Concatenar (joining)

- Objetivo: Acumular los elementos de un agregado que cuyos elementos se supone de tipo *String* en una única cadena de caracteres dados un prefijo *pr*, un sufijo *sf* y un separador *sp*.
  El acumulador tiene una variable adicional para detectar el primer elemento.
- Tipo del acumulador: *(a,b) :(String,boolean)*
- Valor Inicial: (*"",true)*
- Expresión acumuladora: (*a,b) = b? (e,false): (a+sp+e,false)*
- *Combina(a1,a2): a1+a2;*
- Salir(a): false
- Resultado(a): *pr+a+sf*

3.17 Resumen Estadístico(summarizing)

- Objetivo: Suponiendo que el tipo E es un tipo numérico este acumulador calcula información estadística relevante como el número de elementos, la suma, la suma de los cuadrados, el máximo, el mínimo, etc. A partir de esa información básica se puede deducir otra derivada como la media, la varianza, la desviación típica, etc. El acumulador se compone de las propiedades *(n,s,s2,mx,mn)* que acumulan el número de elementos, la suma, la suma de los cuadrados, el máximo y el mínimo.
- Tipo del acumulador: (Integer, E, E, E, E)
- Valor Inicial:

```
a = (0,0,0,null,null);
```

- Expresión acumuladora:

```
(n,s,r,mx,mn) = (n+1,s+e,r+e^2,max(mx,e),min(mn,e);
```

- Combina(a1,a2): *(n1+n2,s1+s2,r1+r2,max(mx1,mx2),min(mn1,mn2);*

- Salir(a): false
- Resultado(a): a

Las funciones *max* y *min* tendrían la implementación usada en los correspondientes acumuladores

```
max(e1,e2) == e1 == null || e > mx ? e : mx;
min(mn,e)== mn == null || e < mn ? e : mn;
```

3.18 Algoritmos para grupos consecutivos de elementos:

Podemos diseñar también esquemas secuenciales para grupos consecutivos de elementos. Son generalizaciones de los anteriores. Solo presentamos la generalización de Para Todo, Existe y Contar, para grupos de dos elementos consecutivos, sin recordar todos los detalles de los mismos. La generalización a más elementos consecutivos puede hacerse fácilmente. Igual podemos decir del resto de los esquemas:

Contar Dos:

- Objetivo: Contar los pares de elementos consecutivos que cumplen la propiedad binaria p(e1,e2).
- Tipo del acumulador: (a,n): (E,Integer).
- Valor Inicial: (null,0).
- Expresión acumuladora:

```
if(a ==null){
       a = e;
} else if(p(a,e)){
       n++;
       a = e;
}
```

- Salir(a,n): Falso
- Resultado(a,n): n

Para Todo Dos:

- Objetivo: Decidir si todos los pares de elementos consecutivos cumplen la propiedad binaria *p(e1,e2).*
- Tipo del acumulador: *(a,r): (E,boolean).*
- Valor Inicial: *(null,true).*
- Expresión acumuladora:

```
if(a ==null){
       a = e;
} else if(!p(a,e)){
       r = false;
```

```
        a = e;
}
```

- Salir(a,r): !r
- Resultado(a,r): r

Existe Dos:

- Objetivo: Decidir si alguno de los pares de elementos consecutivos cumplen la propiedad binaria *p(e1,e2).*
- Tipo del acumulador: *(a,r): (E,boolean).*
- Valor Inicial: *(null,false).*
- Expresión acumuladora:

```
if(a ==null){
       a = e;
} else if(p(a,e)){
       r = true;
       a = e;
}
```

- Salir(a,r): r
- Resultado(a,r): r

Casos particulares de los esquemas anteriores son:

- Comprobar si un iterable está ordenado o estrictamente ordenado. En este caso en el esquema *Para Todo Dos* la expresión que *p(e1,e2)* sea verdadera se debe cumplir *e1 <= e2*.
- Comprobar si existen dos elementos iguales consecutivos.

4. Ejemplos y Metodología de trabajo

Los esquemas anteriores se repiten muy frecuentemente. Para poder usarlos debemos identificar en primer lugar cada uno de sus elementos:

- ¿Cuál que hace de fuente de datos?
- ¿Cuál es el mecanismo de generar los datos?
- ¿Cuál es el filtro si lo hay?
- ¿Cuál es la transformación a usar?
- ¿Cuál de los acumuladores usar?

Para los ejemplos siguientes suponemos implementados los tipos propuestos como ejercicios en el tema anterior.

Ejemplo:

Si queremos diseñar un método *static* de la clase Personas, tomando como parámetro una lista de Persona y una edad que debe ser positiva, devuelva como resultado la suma de las edades de las personas contenidas en la lista que sean mayores que la edad dada. No hay garantías de que los objetos en la lista sean distintos de *null*. En este caso tenemos:

- Origen de datos: la lista
- Generación de datos: la lista es iterable
- Filtro: la persona debe ser distinta de null.
- Transformación: calcula la edad de una persona
- Esquema a usar: suma

El código resultante, donde puede verse el código concreto del Filtro y de la Expresión, es:

```
public static Integer sumaEdades(List<Persona> lista, Integer ed){
   Integer suma = 0;
   for (Persona p : lista) {
      if(p != null){
          Integer e = p. e.getEdad();
          if (e > ed) {
                       suma = suma + e;
          }
      }
   }
   return suma;
}
```

Los siguientes ejemplos ilustran algunos de los esquemas secuenciales. Todos los métodos los podemos ubicar en la clase Racionales. Vamos viendo el enunciado, el código y algunos comentarios.

Ejemplo

Dado un *array* de *String* cada una de las cuales representa un racional, obtener una lista de racionales.

```
public static List<Racional> getLista (String[] ag){
   List<Racional> lr = new ArrayList();
   for (String s : ag) {
          Racional e =  Racionales.create(s);
          lr.add(e);
   }
   return lr;
}
```

Ahora hemos usado el esquema selecciona lista, no hay filtro y *Racionales.create(e)* es la transformación que construye objetos de tipo Racional.

Ejemplo

Implementar el método *getFotos* en la clase Bibliotecas que dada una Biblioteca y dos fechas, la segunda mayor que la primera, devuelva todas las fotos distintas que fueron realizadas entre esas dos fechas.

```
public static Set<Foto> getFotos (Biblioteca b, Fecha f1, Fecha f2){
   if(!(f1.compareTo(f2) <=0)){
      throw new IllegalArgumentException();
   }
   Set<String> sr = new HashSet();
   for (Album  a : b) {
        for(Foto f: a.getFotos()){
            Fecha ff = f.getFecha();
            if(ff.compareTo(f1) > 0 && ff.compareTo(f2) < 0){
                  sr.add(f);
            }
        }
   }
   return sr;
}
```

Ahora hemos usado el esquema acumula conjunto, el filtro es que la fecha esté comprendida estrictamente entre las dos proporcionadas. Suponemos que todas las fotos son distintas de *null*. El origen de datos tiene una lista de álbumes y cada álbum una lista de fotos. Las listas son iterables.

Ejemplo

Dado un *array* de *Racional* cada una de las cuales representa un racional, obtener cuantos hay mayores que cero.

```
public static Integer getNumeroDePositivos (Racional[] ag){
   Integer num = 0;
   Racional cero = Racionales.create();
   for (int i = 0; i < ag.length ; i++) {
        Racional e = ag[i];
        if(e!=null && e.compareTo(cero)){
                num++;
        }
   }
   return num;
}
```

Ahora hemos usado el esquema contar, el filtro es que el racional sea distinto de *null* y positivo.

Ejemplo

Dada una lista de Viaje calcular cuantas ciudades distintas pueden visitarse si hacen todos los viajes de la lista que sean posibles para un grupo formado por un número dado de personas.

```
public static Integer getCiudadesDistintas (List<Viaje> lista, Integer n){
   if(n <= 0){
      throw new IllegalArgumentException();
   }
   Set<String> sr = new HashSet();
   for (Viaje e : ag) {
        if(e.getMinimoDePersonas() >= n){
            for(String s: e.getCiudadesVisitadas(){
                  sr.add(s);
            }
        }
   }
   return sr.size();
}
```

Ahora hemos usado el esquema acumula conjunto, el filtro es que el grupo sea mayor o igual al entero dado. El número de objetos distintos es el cardinal del conjunto construido. Podemos observar que el origen de datos está compuesto de una lista cada uno de cuyos objetos tiene una propiedad que es otra lista. También es interesante constatar que el filtro depende de la variable de tipo Vuelo y no de la de tipo *String*. Esta es la razón para poner el *if* entre los dos *for*. Si el fitro hubiera dependido de la ciudad, de tipo String, entonces iría después del segundo *for*.

Ejemplo

Dada una lista de Polígonos diseñar un método *static* que nos devuelva el que tenga un área mayor de entre los que tengan un número de lados dado.

```
public static Poligono getMayorArea (List<Poligono> lista, int n,
          Comparator<Poligono> ord){
   Poligono elMax = null;
   for (Poligono p : lista) {
          if (p!=null && p.getNumeroDeVertices() == n) {
                if(elMax == null && ord.compare(p,elMax) > 1){
                      elmax = e;
                }
          }
   }
   if(elMax == null) throw new NoSuchElementException();
   return elMax;
}
```

5. Conceptos aprendidos

- Operaciones habituales con agregados de datos
- Elementos comunes a las operaciones habituales: fuente de datos, generación de los datos, acumulador, filtro y transformación
- Acumuladores comunes

6. Ejercicios Propuestos

1. Dada una lista de números enteros, que se toma como parámetro, implementa un método *static* en la clase *Enteros* para cada una de las siguientes cuestiones:
    a) ¿Todos los elementos de la lista son impares?
    b) ¿Existe alguno que sea impar y primo? Supóngase implementado previamente el método *esPrimo*.
    c) ¿Cuántos hay?
    d) ¿Cuál es el valor de suma de los que sean primos? ¿Y del producto de los que sean impares?
    e) ¿Hay alguno que divida a un número n (siendo n un parámetro del método)?
    f) Buscar todos los enteros pares que están en la lista de entrada (que el método devuelva el resultado en otra lista)

2. Dado un *Set<List<Integer>>* que se toma como parámetro, implementa un método static en la clase *Enteros* para cada una de las siguientes cuestiones:
    a) ¿Cuánto vale la suma de todos los elementos que son impares?
    b) ¿Cuál es el entero más pequeño? ¿y el más grande?
    c) ¿Existe algún elemento que sea primo?
    d) ¿Existe algún elemento que sea primo y que ocupe la primera posición de cada lista del conjunto?
    e) Devolver un *List<Integer>* con todos aquellos enteros que ocupan la primera posición en cada lista del conjunto.

3. Dada una lista de números racionales, que se toma como parámetro, implementa un método *static* en la clase Racionales para resolver cada una de las siguientes cuestiones:
    a) ¿Cuál es el mayor? ¿y el menor?
    b) ¿Todos los elementos de la lista tienen valor mayor que 1?
    c) ¿Cuál es el valor de suma, en valor real, de los que sean menores que 1, en valor real?
    d) ¿Y del producto, en valor real, de los que tengan el denominador mayor que 3, en valor real?
    e) Realiza los dos apartados anteriores devolviendo un valor racional
    f) Cuantos racionales distintos hay?

4. Dado un *List<Set<Racional>>* que se toma como parámetro, implementa un método *static* en la clase Racionales para resolver cada una de las siguientes cuestiones:
    a) ¿Cuál es el conjunto que tiene más elementos?
    b) ¿Existe algún conjunto con un número impar de elementos?
    c) Devolver un racional cuyo numerador sea la suma de los numeradores y cuyo denominador el producto de los denominadores.

5. Dado un *Set<List<Integer>>* que se toma como parámetro, implementa un método *static* en la clase Enteros para resolver cada una de las siguientes cuestiones:
   a) ¿Cuál es la lista de mayor tamaño?
   b) ¿Existe alguna lista que contenga algún entero con valor negativo?
   c) ¿Todas las listas son de tamaño mayor que 2?
   d) ¿Cuánto suma la lista de todos los números enteros?

6. En una biblioteca de fotos digitales, con los tipos definidos en el capítulo anterior, implemente los siguientes métodos :

   a) El método *getTamaño* de la clase Albumes, que devuelve el tamaño total de las fotos de un álbum, que toma como parámetro, como la suma de los tamaños de tod*as las fo*tos del álbum.
   b) Un método estático de la clase Bibliotecas que devuelva un álbum con las fotos de una biblioteca que tienen un descriptor dado y han sido retocadas. El nombre del álbum será el propio descriptor.

7. Usando los tipos del capítulo anterior implemente:

   a) Un método llamado *beneficioTotal* que tome como argumentos un Huerto y un *Integer*, y devuelva el beneficio total del huerto de aquellas cosechas cuyo tiempo de recolección sea menor al entero recibido.
   b) Un método llamado *aumentaBeneficios*, que reciba un Huerto, un *Set <String>* y un *Double* y aumente en esa cantidad el beneficio de aquellas cosechas cuyo nombre esté contenido en el conjunto.

8. Implemente un método estático en la clase *LectoresElectrónicos* para responder a cada una de las siguientes preguntas.
   a) Un método que construya un Set<Libro> con todos los libros de un *LectorElectrónico<Libro>* que contengan el descriptor “programación” y tengan una valoración igual o superior a 4.

9. Implemente los siguientes métodos en la clase Viajes:

   a) Un método que devuelva la calificación media de los clientes para todos los viajes en el catálogo. Parámetros de entrada: catálogo de viajes.
   b) Un método que diga si todos los viajes del catálogo son de un operador determinado. Parámetros de entrada: catálogo de viajes y nombre del operador.

10. Dadas las siguientes interfaces:

```
public interface Destino extends Comparable<Destino>{
      Double getDistancia();
      String getNombre();
}
public interface Vuelo {
      Destino getDestino();
      Double getPrecio();
      Integer getNumPlazas();
      Integer getNumPasajeros();
```

```
        Integer getCodigo();
        Fecha getFechaSalida();
        Persona getPiloto();
}
public interface CompañiaAerea {
        Integer getCodigo();
        String getNombre();
        Vuelo[] getVuelos();
}
```

Implemente los siguientes métodos en la clase Compañias:

a) Un método que dada una compañía aérea devuelva el nombre del piloto del primer vuelo (de salida más pronto) con plazas libres con destino París.
b) Construya un método que dada una compañía aérea devuelva el código del vuelo que más distancia recorre de entre los que salen hoy.