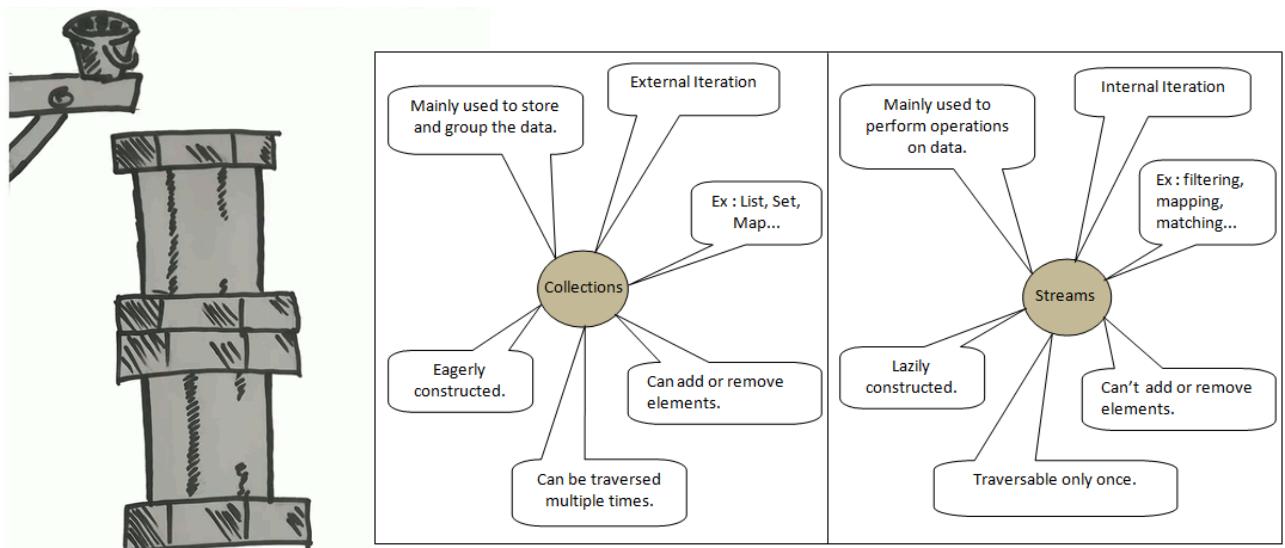


UT15.3. STREAMS

Un Stream és una **seqüència o successió de dades** (NO una col·lecció NI una estructura de dades) sobre les quals es pot fer una sèrie d'operacions, que poden anar encadenades (chaining) fins a donar un resultat final.



Estes operacions realitzades amb els Stream poden ser de dos tipus:

- **Intermèdies**: Filtren o transformen la seqüència de dades. Donen com a resultat un nou **stream** al qual li podem seguir aplicant noves operacions (operacions pendents)
- **Terminals**: Finalitzen el processament (efectúa l'operació resultant de les intermèdies). Retornen un valor o realitzen alguna acció sobre les dades.

Les col·leccions en Java ofereixen mètodes per a passar a streams dels objectes que contenen (mètodes stream i parallelStream). Així, si tenim un ArrayList anomenat "usuaris" que conté objectes de tipus "Usuari" podríem fer:

```
Stream<Usuari> fluxUsuaris = usuaris.stream();
```

La idea és crear, a partir d'una col·lecció o taula (Map), o bé explícitament, un Stream al qual s'apliquen operacions intermèdies encadenades (el que es coneix com una canonada o pipeline), obtenint un resultat final per mitjà d'una operació terminal.

Un primer avantatge que podem veure als Stream (més endavant en veurem més) és que disposa de moltes més operacions per processar les dades que les col·leccions o les taules.

Els Stream són objectes que implementen la **interfície Stream**. Per tant, la classe Stream no existeix i els objectes Stream no es poden crear amb un constructor sinó només invocant certs mètodes que retornen un Stream.

S'anomenen “operacions agregades” aquelles que operen sobre la totalitat d'un Stream, permetent l'execució en paral·lel, transparent al programador, per augmentar la velocitat del procés (imagina càculs o cerques sobre matrius).

Característiques d'un Stream

- Les operacions intermèdies retornen un Stream (encadenament).
- Les operacions intermèdies es posen en cua i són invocades en executar una operació terminal (s'invoca la composició de les intermèdies).
- Només es pot recórrer o consumir una vegada.
- Iteració interna vs. externa: ens centrem en què fer amb les dades, no en com recórrer-les.

Subtipus bàsics

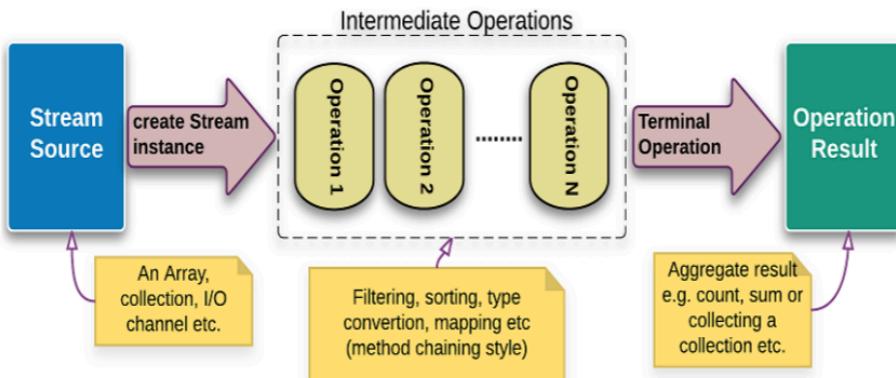
- **IntStream**: Stream d'elements “int” (ja que sabem que no es pot fer Stream<int>)
- **LongStream**: Stream d'elements “long”
- **DoubleStream**: Stream d'elements “double”

Formes de crear un Stream

Hi ha diverses maneres d'obtenir un Stream inicial, és a dir, que no procedisca d'un altre Stream. En veiem algunes:

- A partir d'una col·lecció: invocant al mètode “stream()”, definit a les classes de tipus Collection.
`Stream<T> nomStream = nomColeccio.stream();`
- A partir d'un array de tipus T[]: usant el mètode of() de la interfície Stream, amb l'array com a paràmetre d'entrada.
`Stream<T> nomStream = Stream.of(T[] array);`
- A partir d'una taula de tipus T[]: usant el mètode “stream()” de la classe Arrays, amb la taula com a paràmetre (utilitza este mètode si T representa un tipus primitiu)
`Stream<T> nomStream = Arrays.stream(T[] array);`
- Inicialitzant-lo directament: també amb el mètode “of()” però passant-li com a llista de paràmetres els valors de tipus T que l'inicialitzen
`Stream<T> nomStream = Stream.of(T v1, T v2, T v3, ...);`
- Iterant sobre un valor inicial amb el mètode “iterate()” de Stream tornarà un stream infinit, ordenat i seqüencial
`Stream<T> nomStream = Stream.iterate(T i, UnaryOperator<T> o);`
- També podem fer que no siga infinit definint la condició d'iteració amb iterate:
`Stream<T> nomStream = Stream.iterate(T i, Predicate<? super T> p, UnaryOperator<T> o);`
- A partir d'un Supplier tornarà un Stream infinit, seqüencial i no ordenat
`Stream<T> nomStream = Stream.generate(Supplier<T> s);`

Java Streams



```
List<String> localitats = new ArrayList<>();
    List.of("Petrer", "Elda", "Novelda", "Villena", "Monòver", "Castalla", "Alacant", "Elx"));
Stream<String> streamLocalitats = localitats.stream();
streamLocalitats.forEach(c -> System.out.println(c.toUpperCase()));
```

El Stream localitats **conté una còpia de totes les dades de la llista, no una referència** als originals. Per tant els canvis que es facen al Stream no es reflecteixen a la llista original que queda intacta.

També podem crear IntStream fent un rang obert o bé tancat:

```
IntStream xifresPositives = IntStream.range(1, 10);
o bé
IntStream xifresPositives = IntStream.rangeClosed(1, 9);
```

Algunes classes de l'API de java tenen mètodes que tornen Stream o algun dels seus derivats (IntStream, DoubleStream...) a destacar per exemple la classe Random:

```
// Genera un stream de 100 enters aleatoris entre 0 i 10
IntStream aleatoris = new Random().ints(100, 0, 10);
```

Un altre exemple, la classe String ens ofereix "chars()" que genera un IntStream amb tots els seus caràcters (en format enter).

```
String hola = "Hola món!";
hola.chars().forEach(System.out::println);

String hola = "Hola món!";
hola.chars().forEach(i -> System.out.println((char) i));
```

En realitat no necessitem crear les variables:

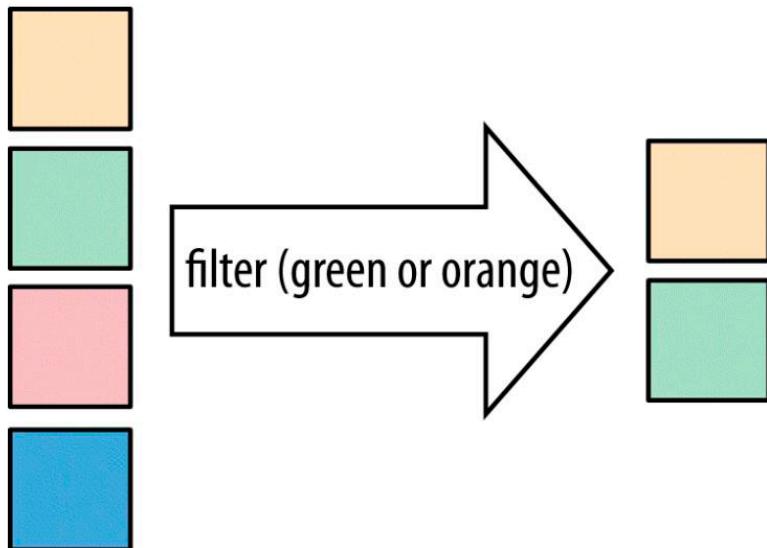
```
"Hola món!".chars().forEach(...)
```

Prova el mètode "tokens()" de la classe Scanner.

Opcions de filtratge

- filter(Predicate<T>): ens permet filtrar utilitzant una condició.
- limit(n): ens permet obtenir els n primers elements.
- skip(m): ens permet obviar els m primers elements.
- distinct: descarta els elements repetits (segons marque “equals”)

Filter



- Operació intermèdia
- Ens permet eliminar aquells elements del stream que no compleixen una determinada condició
- Condició com a Predicate.
- Molt combinable amb findFirst, findAny
- Combinable amb la resta de mètodes intermedis i terminals.

```
List<Integer> valors = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
Stream<Integer> temporal = valors.stream();
temporal.filter(e -> e>3).forEach(e -> System.out.println(e));
```

```
List<Integer> valors = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
Predicate<Integer> esParell = e -> e % 2 == 0;
Consumer<Integer> imprimir = e -> System.out.println(e);

Stream<Integer> temporal = valors.stream();
temporal.filter(esParell).forEach(imprimir);
```

```

public class Aplicacio {

    public static void main(String[] args) {

        List<Integer> valors = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

        Stream<Integer> temporal = valors.stream();
        temporal
            .filter(Aplicacio::esParell)
            .filter(Aplicacio::esMajorQue5)
            .forEach(System.out::println);

    }

    public static boolean esParell(Integer e){
        return e%2==0;
    }

    public static boolean esMajorQue5(Integer e){
        return e > 5;
    }

}

```

```

List<Integer> valors = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
Function<Integer,Predicate<Integer>> majorQue = max -> e -> e > max;
Stream<Integer> temporal = valors.stream();
temporal
    .filter(Aplicacio::esParell)
    .filter(majorQue.apply(6))
    .forEach(System.out::println);

```

Limit

- Operació intermèdia.
- Ens permet limitar el nombre d'elements que contindrà el stream.
- Passem com a paràmetre el nombre d'elements que tindrà el stream resultant.

```

Stream<Integer> temporal = Stream.iterate(0, n->n+2);

temporal.limit(5).forEach(System.out::println);

```

```

Stream.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
    .filter(i -> i % 2 == 0)
    .limit(5)
    .forEach(n -> System.out.print(n + " "));

```

Skip

- Operació intermèdia.
- Este mètode pren un número N com a argument i torna una seqüència (stream) després d'eliminar els primers N elements

```
Stream.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
    .filter(i -> i % 2 == 0)
    .skip(2)
    .forEach(i -> System.out.print(i + " "));
```

Distinct

- Operació intermèdia.
- Este mètode elimina els elements duplicats i retorna el Stream sense duplicats.

```
Stream.of(2,5,6,9,3,8,1,3,8,9)
    .filter(i -> i % 2 != 0)
    .distinct()
    .forEach(i -> System.out.print(i + " "));
```

Diferències?

```
Stream.iterate(0, i -> i + 1)
    .filter(i -> i % 2 == 0)
    .skip(5)
    .limit(15)
    .forEach(System.out::println);
```

```
Stream.iterate(0, i -> i + 1)
    .filter(i -> i % 2 == 0)
    .limit(15)
    .skip(5)
    .forEach(System.out::println);
```

```
Stream.iterate(0, i -> i + 1)
    .limit(15)
    .skip(5)
    .filter(i -> i % 2 == 0)
    .forEach(System.out::println);
```

Concatenació d'Streams. Concat

El mètode `Stream.concat()` crea un flux concatenat en què els elements són tots els elements del primer flux seguits de tots els elements del segon flux. El flux resultant s'ordena si els dos fluxos d'entrada estan ordenats i és paral·lel si qualsevol flux d'entrada és paral·lel.

```
Stream<String> s1 = Stream.of("daw", "dam", "asir");
Stream<String> s2 = Stream.of("llet", "lletuga", "aigua", "sal");
Stream<String> s3 = Stream.of("verd", "blau");

Stream<String> concatenat = Stream.concat(s1, Stream.concat(s2, s3));

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

```
daw
dam
asir
llet
lletuga
aigua
sal
verd
blau
```

```
List<Integer> llista = new ArrayList<>(Arrays.asList(1,9, -3, 7));
Stream<Integer> unsEnters = Stream.of(65,9,-32,1);

Stream<Integer> totsEnters = Stream.concat(llista.stream(), unsEnters);

totsEnters.distinct().forEach(System.out::println);
```

També es poden concatenar streams de tipus derivat:

```
Stream<Integer> enters = Stream.of(9,8,-3,2);
Stream<Double> doubles = Stream.of(3.2, 0.75);

Stream<Number> numeros = Stream.concat(enters, doubles);
```

Ordenació de dades. Sorted

- Operació intermèdia.
- Ens permet ordenar els elements d'un stream.
- Recordem que per ordenar qualsevol conjunt d'objectes d'una classe necessitem en primer lloc o com a punt de partida que implemente l'interfície Comparable<T>. També es pot passar com a paràmetre un Comparator<T>

```
Stream.of(15,4,8,1,-5)
    .sorted()
    .forEach(n -> System.out.print(n + " "));
```

```
Stream.of(15,4,8,1,-5)
    .sorted((a,b) -> b - a )
    .forEach(n -> System.out.print(n + " "));
```

```
Persona p1 = new Persona("75849521X", "Pepito López", 32);
Persona p2 = new Persona("94874621T", "Laura Coves", 31);
Persona p3 = new Persona("44746921V", "Ana Cazorla", 25);

Stream.of(p1,p2,p3)
    .sorted((a,b) -> a.getDni().compareTo(b.getDni()))
    .forEach(System.out::println);
```

```
Persona p1 = new Persona("75849521X", "Pepito López", 32);
Persona p2 = new Persona("94874621T", "Laura Coves", 31);
Persona p3 = new Persona("44746921V", "Ana Cazorla", 25);

Stream.of(p1,p2,p3)
    .sorted((a,b) -> a.getEdat() - b.getEdat())
    .forEach(System.out::println);
```

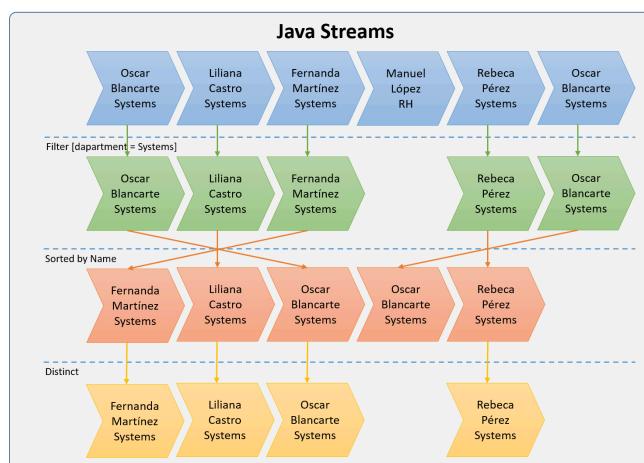
Altres maneres d'usar el mètode “sorted”

```
Stream.of(p1,p2,p3)
    .sorted(Comparator.comparingInt(Persona::getEdat))
    .forEach(System.out::println);
```

```
Stream.of(p1,p2,p3)
    .sorted(Comparator.comparing(p->p.getEdat()))
    .forEach(System.out::println);
```

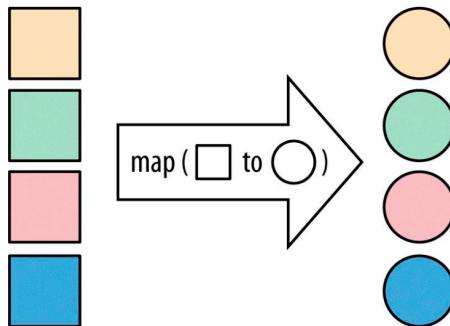
```
Function<Persona, Integer> perEdat = p -> p.getEdat();
Function<Persona, String> perDni = p -> p.getDni();

Stream.of(p1,p2,p3)
    .sorted(comparing(perEdat).thenComparing(perDni))
    .forEach(System.out::println);
```



Transformació de dades

Map



- Operació intermèdia molt utilitzada.
- Permet aplicar una transformació a una sèrie de dades.
- Retorna un stream resultat de la correspondència de cada element de la seqüència original transformat en altra dada.
- Accepta com a paràmetre una funció (Function<T, R>)

```
Stream.of(15, 4, 8, 1, -5)
    .map(n -> n*2)
    .forEach(System.out::println);
```

```
Persona p1 = new Persona("75849521X", "Pepito López", 32);
Persona p2 = new Persona("94874621T", "Laura Coves", 31);
Persona p3 = new Persona("44746921V", "Ana Cazorla", 25);
```

```
Stream.of(p1, p2, p3)
    .map(Persona::getDni)
    .forEach(System.out::println);
```

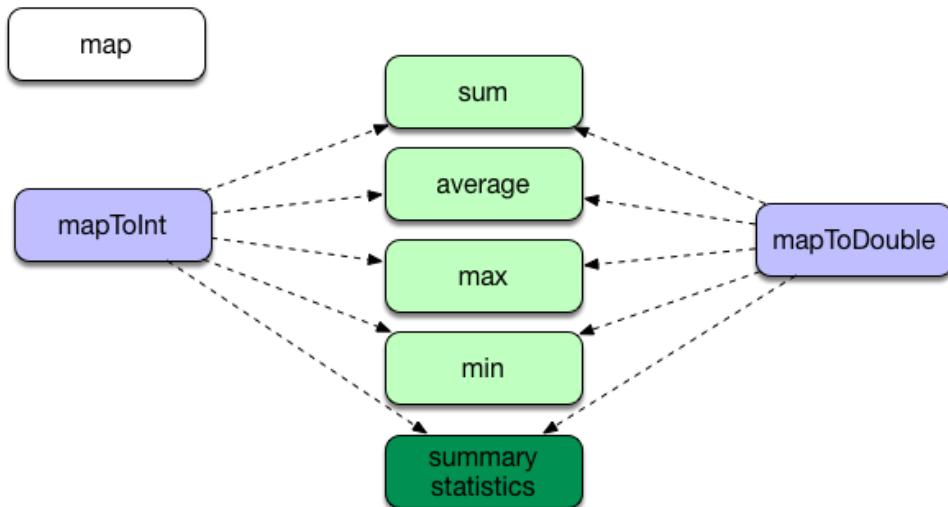
```
Stream.of(p1, p2, p3)
    .map(Persona::getDni)
    .map(StringBuilder::new)
    .map(StringBuilder::reverse)
    .forEach(System.out::println);
```

```
Stream.of(0.6, 2.5, 0.25, 0.4)
    .map(Fraccio::new)
    .map(Fraccio::simplificar)
    .forEach(System.out::println);
```

```
Stream.of(0.6, 2.5, 0.25, 0.4)
    .map(Fraccio::new)
    .map(Fraccio::simplificar)
    .sorted((a,b) -> Double.compare(a.toDouble(), b.toDouble()))
    .forEach(System.out::println);
```

MapToInt i MapToDouble

Hi ha dos mètodes addicionals orientats a treballar amb dades numèriques. Aquests mètodes són **mapToInt** i **mapToDouble**. Si canviem el nostre mètode de map a mapToInt o mapToDouble (quan sapiem que la transformació és a un enter o un double primitius) se'n obrirà la possibilitat d'accendir a mètodes addicionals orientats a estadístiques.



Els mètodes **sum**, **average**, **max** i **min** són **TERMINALS** (igual que foreach) per la qual cosa tornen un resultat i no un altre stream (fins que no troba una operació terminal realment no s'executa cap intermèdia). En el cas de “average”, “max” i “min” retornen un **Optional**.

```
Stream.of(15,4,8,1,-5)
    .mapToInt(n -> n*2)
    .max()
    .ifPresent(System.out::println);
```

```
Persona p1 = new Persona("75849521X", "Pepito López", 32);
Persona p2 = new Persona("94874621T", "Laura Coves", 31);
Persona p3 = new Persona("44746921V", "Ana Cazorla", 24);

Stream.of(p1,p2,p3)
    .mapToInt(Persona::getEdat)
    .average()
    .ifPresent(System.out::println);
```

```
System.out.println(Stream.of(p1,p2,p3)
    .mapToInt(Persona::getEdat)
    .summaryStatistics());
```

MapToObj

Una altra de les operacions intermèdies que podem fer és mapToObj sobre un IntStream / LongStream / DoubleStream. Este mètode retorna un Stream amb valor d'objecte que consta dels resultats d'aplicar la funció donada. Per exemple, imaginem que a partir d'una llista d'enters volem convertir-los a un String del seu valor en hexadecimal:

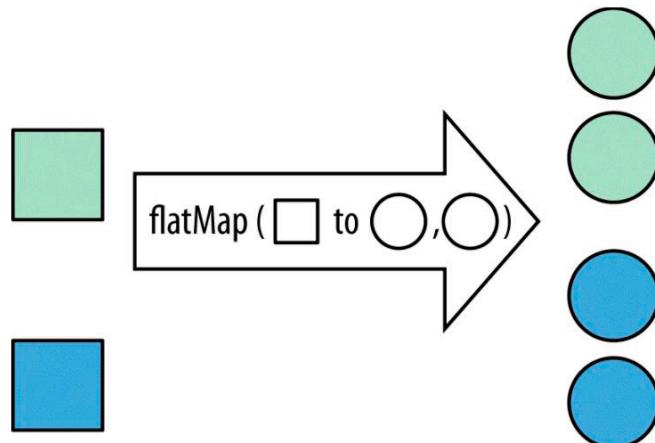
```
IntStream valors = IntStream.range(7, 19);  
  
Stream<String> valorsHexadecimal = valors  
    .mapToObj(Integer::toHexString);  
  
valorsHexadecimal.forEach(System.out::println);
```

Vegem per exemple com convertir un Stream de doubles a un d'Angles:

```
DoubleStream.of(30.5, 88.25, 45.0, 158.75)  
    .mapToObj(Angle::new)  
    .forEach(System.out::println);
```

(*) També podem aplicar mètodes map, mapToInt, mapToDouble o MapToLong...

FlatMap



Els streams sobre col·leccions d'un nivell (com List) es poden transformar (map) fàcilment. Però què passa si tenim una col·lecció que inclou dins una altra?

```
public class Turista extends Persona {  
    private List<Viatge> viatges;
```

```

Viatge v1 = new Viatge("Mallorca", 7);
Viatge v2 = new Viatge("Roma", 10);
Viatge v3 = new Viatge("París", 8);
Viatge v4 = new Viatge("Valencia", 3);
Turista t1 = new Turista(Arrays.asList(v1, v3), "78459874X", "David Tibar", 28);
Turista t2 = new Turista(Arrays.asList(v2, v4), "95415258Q", "Laura Méndez", 32);
List<Turista> turistes = new ArrayList<>(Arrays.asList(t1, t2));

```

Per a vore totes les destinacions, amb estil imperatiu “for” niuem dos bucles:

```

for(Turista t: turistes){
    for(Viatge v: t.getViatges()) {
        System.out.println(v.getDesti());
    }
}

```

► Podem observar bé per adonar-nos dels tipus de retorn dels mètodes intermedis:

```

// Intent sense flatMap
turistes.stream()                                // Stream<Turista>
    .map(t -> t.getViatges())                    // Stream<List<Viatge>>
    .forEach(System.out::println);

```

► Necessitem un mètode que unifique totes les llistes de viatges en un sol Stream:
 ► Eixa és la funcionalitat de flatMap.

Amb flatMap no només apliquem una transformació, sino que passen a un sol Stream:

```

turistes.stream()
    .map(t -> t.getViatges())
    .flatMap(lv -> lv.stream())
    .map(v -> v.getDesti())
    .forEach(System.out::println);

```

Imaginem que volem obtenir la suma de tots els dies que han estat de vacances els nostres turistes en tots els seus viatges:

```

int totalDies = turistes
    .stream()
    .map(Turista::getViatges)
    .flatMap(List::stream)
    .mapToInt(Viatge::getDies)
    .sum();

System.out.println(totalDies);

```

També tenim les versions primitives (flatMapToInt, ...):

```

int[][] numeros = {{1, 2, 2, 3, 1, 4}, {4, 2, 3, 3, 1, 1}};
Arrays
    .stream(numeros)
    .flatMapToInt(fila -> Arrays.stream(fila))
    .forEach(System.out::println);

```

Amb això podem deduir altres formes per concatenar Streams. Si recordem anteriorment hem vist l'exemple

```

Stream<String> s1 = Stream.of("daw", "dam", "asir");
Stream<String> s2 = Stream.of("llet", "lletuga", "aigua", "sal");
Stream<String> s3 = Stream.of("verd", "blau");

Stream<String> concatenat = Stream.concat(s1, Stream.concat(s2, s3));
concatenat.forEach(System.out::println);

```

Amb flatMap podem aconseguir el mateix efecte sense haver de niar funcions concat.

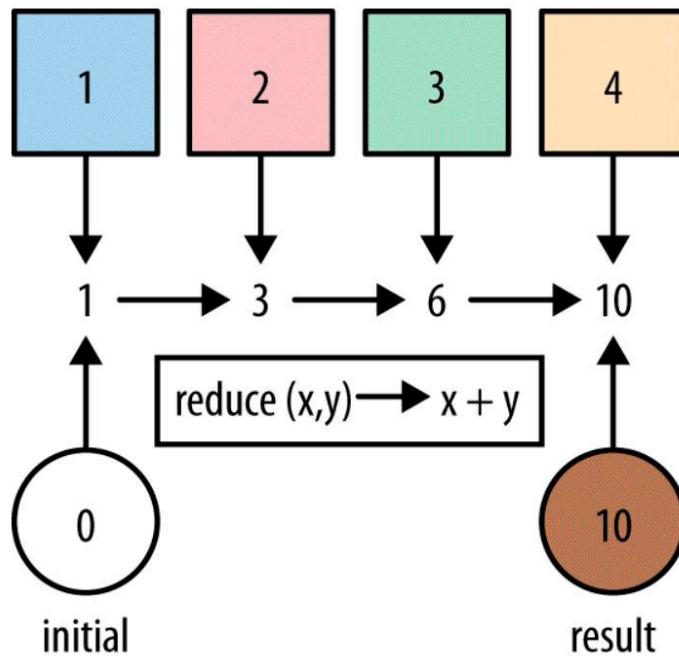
```

Stream<String> s1 = Stream.of("daw", "dam", "asir");
Stream<String> s2 = Stream.of("llet", "lletuga", "aigua", "sal");
Stream<String> s3 = Stream.of("verd", "blau");

Stream<String> concatenat = Stream.of(s1, s2, s3)
    .flatMap(Function.identity());
concatenat.forEach(System.out::println);

```

Reducció de dades. Reduce



► `reduce(BinaryOperator<T> accumulator): Optional<T>`

Realitza la reducció del Stream usant una funció associativa. Retorna un `Optional`

► `reduce(T identity, BinaryOperator<T> accumulator): T`

Realitza la reducció usant un valor inicial i una funció associativa. Retorna un valor com a resultat.

Redueix, la qual cosa ens permet obtindre un únic resultat partint d'un stream. Operació terminal.

Per a poder aconseguir el nostre resultat únic final, farem ús dels paràmetres següents:

- **Identitat (Identity):** És el valor inicial del procés. [Paràmetre opcional. Si no es proporciona este paràmetre el mètode “reduce” retorna un `Optional`]
- **Acumulador (Accumulator):** És l’encarregat de processar els valors i acumular-ne el resultat.
- **Combinador (Combiner):** El combinador és usat per combinar el resultat parcial obtingut quan la reducció és paral·lelitzada. [Paràmetre opcional]

Aquí veiem el funcionament invocant al primer constructor de “reduce” (sense Identity)

```
Optional<Integer> elMaxim = Stream.of(9,8,2,15,3).reduce(Integer::max);  
elMaxim.ifPresent(System.out::println);
```

Més habitual és utilitzar un primer paràmetre com a identitat o valor inicial de la reducció.

```
int laSuma = Stream.of(9,8,2,15,3).reduce(0, (a,b)->a+b);  
System.out.println(laSuma);
```

```
int laSuma = Stream.of(9,8,2,15,3).reduce(0, Integer::sum);  
System.out.println(laSuma);
```

Com podem imprimir la suma de dies de tots els viatges efectuats pels turistes? I la suma de dies de cada turista?

```
int reduce = turistes.stream()  
    .map(Turista::getViatges)  
    .flatMap(List::stream)  
    .map(Viatge::getDies)  
    .reduce(0, Integer::sum);  
  
System.out.println(reduce);
```

```
turistes.stream()  
    .map(Turista::getViatges)  
    .map(List::stream)  
    .map(ls -> ls.map(Viatge::getDies))  
    .map(sd -> sd.reduce(0, Integer::sum))  
    .forEach(System.out::println);
```

Analitza què imprimeix el codi següent:

```
String reduce = Stream.of("Gener", "Març", "Juny", "Agost", "Novembre")  
    .map(String::toUpperCase)  
    .reduce("", (a, m) -> a + m.charAt(a.length()));  
  
System.out.println(reduce);
```

Amb això podem deduir altres formes per concatenar Streams. Amb reduce podem aconseguir el mateix efecte sense haver de niar funcions concat:

```
Stream<String> s1 = Stream.of("daw", "dam", "asir");
Stream<String> s2 = Stream.of("llet", "lletuga", "aigua", "sal");
Stream<String> s3 = Stream.of("verd", "blau");

Stream<String> concatenat = Stream.of(s1,s2,s3)
    .reduce(Stream.empty(), Stream::concat);

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

Operacions terminals

Fins ara la majoria d'operacions vistes són intermèdies (generen un altre stream). Un exemple d'operació terminal seria “**reduce**” o “**forEach**” que ja no tornen un altre stream. També hem vist altres operacions terminals com ara **sum**, **average**, **max** i **min** (en este cas només per als IntStream o DoubleStream).

- **long count()**
Retorna un enter llarg amb el nombre d'elements que conté el *Stream*.
- **Optional<T> findFirst()**
Retorna el primer element del Stream, o un Optional buit si no hi ha cap element.
- **Optional<T> findAny()**
Retorna un element qualsevol del *Stream* o un Optional buit si no hi ha cap element.
- **boolean allMatch(predicat)**
Retorna *true* si la condició del predicat es compleix per a tots els elements del Stream, *false* per a la resta de casos.
- **boolean anyMatch(predicat)**
Retorna *true* si la condició del predicat la compleix almenys un element del Stream, *false* per a la resta de casos.
- **boolean noneMatch(predicat)**
Retorna *true* si la condició del predicat no es compleix per a cap element del Stream, *false* per a la resta de casos.

```
List<Integer> llista = new ArrayList<>(Arrays.asList(1,9, -3, 7));
Stream<Integer> unsEnters = Stream.of(65,9,-32,1);

Stream<Integer> totsEnters = Stream.concat(llista.stream(), unsEnters);

long numPositius = totsEnters.filter(i -> i>0).count();
System.out.println("Elements positius: " + numPositius);
```

```

List<Integer> llista = new ArrayList<>(Arrays.asList(1, 9, -3, 7));

Optional<Integer> primer = llista.stream()
    .filter(i -> i % 2 == 0)
    .findFirst();

primer.ifPresentOrElse(i -> System.out.println("Primer: " + i),
    () -> System.out.println("No s'ha trobat cap element"));

```

```

Arrays.asList(1, 2, -3, 8)
    .stream()
    .filter(i -> i % 2 == 0)
    .findFirst()
    .ifPresentOrElse(i -> System.out.println("Primer: " + i),
        () -> System.out.println("No s'ha trobat cap element"));

```

```

Arrays.asList(1, 2, -3, 8)
    .parallelStream()
    .filter(i -> i % 2 == 0)
    .findAny()
    .ifPresentOrElse(i -> System.out.println("Un exemple: " + i),
        () -> System.out.println("No s'ha trobat cap element"));

```

```

boolean totsPositius = Arrays.asList(1, 7, -3, 9)
    .stream()
    .allMatch(i -> i > 0);

System.out.println("Tots positius: " + totsPositius);

```

```

boolean totsImparells = Arrays.asList(1, 7, -3, 9)
    .stream()
    .allMatch(i -> i % 2 != 0);

System.out.println("Tots imparells: " + totsImparells);

```

```

boolean algunNegatiu = Arrays.asList(1, 7, -3, 9)
    .stream()
    .anyMatch(i -> i < 0);

System.out.println("Hi ha almenys un negatiu: " + algunNegatiu);

```

```

boolean senseParells = Arrays.asList(1, 7, -3, 9)
    .stream()
    .noneMatch(i -> i % 2 == 0);

System.out.println("No hi ha parells: " + senseParells);

```

```

Predicate<Integer> condicio = i -> i > 0;
condicio = condicio.and(i -> i % 2 == 0);

boolean positiuParell = Arrays.asList(1, 7, -2, 9)
    .stream()
    .anyMatch(condicio);

System.out.println("Hi ha almenys un positiu parell: " + positiuParell);

```

// EXERCICIS 1-15 .

Recol·lectors



Fins ara, les operacions realitzades amb streams han acabat amb una eixida per consola.

I si volem transformar un stream (immutable) i guardar-ne el resultat en una col·lecció (mutable)? Operació collect.

Java SE 8 introduceix **Collectors**, amb mètodes estàtics molt usuals i pràctics al paquet `java.util.stream.Collectors.*`;

Ens permet realitzar algun tipus d'operació i recol·lectar el valor en un sol (un sol valor, una sola col·lecció...).

Alguns es solapen amb operacions finals que ja hem vist; poden utilitzar-se juntament amb altres recol·lectors.

Alguns dels recol·lectors bàsics són:

- ▶ **counting**: compta el nombre d'elements.
- ▶ **minBy, maxBy**: obté el mínim o màxim segons un comparador.
- ▶ **summingInt, summingLong, summingDouble**: la suma dels elements (segons el tipus).
- ▶ **averagingInt, averagingLong, averagingDouble**: la mitjana (segons el tipus).
- ▶ **summarizingInt, summarizingLong, summarizingDouble**: els valors anteriors, agrupats en un objecte de resum d'estadístiques (segons el tipus).
- ▶ **joining**: unió dels elements en una cadena de text.

A “Collection”

- Produeixen com a resultat una de les col·leccions ja conegeudes: List i Set
- També pot produir altres estructures de dades de tipus key-value com Map.

Al següent exemple convertim el “steam” en una llista modificable:

```
Stream<Integer> dades = Stream.iterate(0, i -> i + 7)
    .filter(i -> i % 2 == 0)
    .limit(5);

List<Integer> llista = dades.collect(toList());
llista.add(99);

System.out.println(llista);
```

(Utilitza `Collectors.toList()` si no has importat la classe “`Collectors`” al teu projecte)

També podem fer-ho en una llista no modificable.

```
Stream<Integer> dades = Stream.iterate(0, i -> i + 7)
    .filter(i -> i % 2 == 0)
    .limit(5);

List<Integer> llista = dades.collect(toUnmodifiableList());
llista.add(99);

System.out.println(llista);
```

(*) A partir de Java 16 s'afegeix el mètode “`toList()`” a la classe “`Stream`” (sense fer ús de “`collect`”) que simplifica molt la nostra tasca: [Collecting Stream Items into List in Java](#)

```
List<Integer> dades = Stream.iterate(0, i -> i + 7)
    .filter(i -> i % 2 == 0)
    .toList();
```

També podem recol·lectar les dades en un Set:

```
Stream<Integer> dades = Stream.of(2,8,3,2,5,8,2)
    .filter(i->i%2==0);

Set<Integer> conjunt = dades.collect(toSet());

System.out.println(conjunt);

Set<Integer> conjunt= Stream.of(2,8,3,2,5,8,2)
    .filter(i->i%2==0)
    .collect(toUnmodifiableSet());
```

Si volem utilitzar una implementació personalitzada (`ArrayList`, `LinkedList`, `HashSet`...), necessitarem utilitzar el recol·lector **toCollection** amb una col·lecció proporcionada de la nostra elecció.

```
ArrayList<Integer> llista= Stream.of(2,8,3,2,5,8,2)
    .filter(i->i%2==0)
    .collect(toCollection(ArrayList::new));
```

```
LinkedList<Integer> llista= Stream.of(2,8,3,2,5,8,2)
    .filter(i->i%2==0)
    .collect(toCollection(LinkedList::new));
```

A “Map”

El recol·lector “toMap” es pot utilitzar per a recopilar elements Stream en una instància de “map”. Per a fer-ho, necessitem proporcionar dos funcions:

```
keyMapper  
valueMapper
```

Usarem keyMapper per a extreure una clau de mapa d'un element Stream i valueMapper per extreure un valor associat amb una clau determinada.

Per exemple si volem guardar els elements en un “map” que tinga cadenes com a claus i les seues longituds com a valors:

```
Stream<String> paraules = Stream.of("aigua", "llet", "lletuga", "peix");  
Map<String, Integer> mides = paraules  
| | .collect(toMap(Function.identity(), String::length));  
  
System.out.println(mides);
```

També podem, a partir d'un Stream de persones generar un “map” la clau del qual siga el dni i el seu valor siga el seu nom:

```
Persona p1 = new Persona("75849521X", "Pepito López", 32);  
Persona p2 = new Persona("94874621T", "Laura Coves", 31);  
Persona p3 = new Persona("44746921V", "Ana Cazorla", 24);  
  
Map<String, String> dniNom = Stream.of(p1,p2,p3)  
| | .collect(toMap(p -> p.getDni(), p-> p.getNom()));
```

toMap NO evalua si els valors clau són iguals. Si troba claus duplicades, llança immediatament una IllegalStateException.

En estos casos amb col·lisió de claus, hauríem d'utilitzar toMap amb un tercer paràmetre que indica què fer en cas de col·lisions:

```
Persona p1 = new Persona("75849521X", "Pepito López", 32);  
Persona p2 = new Persona("94874621T", "Laura Coves", 31);  
Persona p3 = new Persona("75849521X", "Ana Cazorla", 24);  
  
Map<String, String> dniNom = Stream.of(p1,p2,p3)  
| | .collect(toMap(p -> p.getDni(), p-> p.getNom(), (persona, personaB) -> persona));
```

Recol·lecta i llavors...

CollectingAndThen és un recol·lector especial que ens permet realitzar una altra acció en un resultat, immediatament després què finalitza la recopilació.

```
List<Integer> llista = Stream.iterate(0, i -> i + 7)
    .filter(i -> i % 2 == 0)
    .limit(5)
    .collect(Collectors.collectingAndThen(toList(), Collections::unmodifiableList));

int longitud = Stream.iterate(0, i -> i + 7)
    .filter(i -> i % 2 == 0)
    .limit(5)
    .collect(Collectors.collectingAndThen(toList(), List::size));
```

què mostrarà per pantalla?

```
List<String> productes = Stream.of("llet", "aigua", "peix", "lletuga")
    .collect(Collectors.collectingAndThen(toList(), l -> l.subList(0, l.size() - 1)));

System.out.println(productes);
```

joining

El recol·lector “joining” es pot utilitzar per unir elements Stream<String>.

Podem unir-los fent:

```
String productes= Stream.of("llet", "aigua", "peix", "lletuga")
    .collect(Collectors.joining(", "));

System.out.println(productes);
```

També podem concatenar un String a principi i final de la cadena:

```
String productes= Stream.of("llet", "aigua", "peix", "lletuga")
    .collect(Collectors.joining(", ", "Productes: ", "..."));

System.out.println(productes);
```

Inclús podem mostrar una llista d'enters separats per comes.

```
String valors = IntStream.of(3, 5, 8, 9, 1, 9)
    .mapToObj(String::valueOf)
    .collect(Collectors.joining(", "));

System.out.println(valors);
```

counting

Counting és un recol·lector simple que permet comptar tots els elements de l'Stream. Ara podem escriure:

```
Long elements = Stream.of("llet", "aigua", "peix", "lletuga")
    .collect(counting());

System.out.println(elements);
```

summarizingDouble/Long/Int.

```
DoubleSummaryStatistics longituds = Stream.of("llet", "aigua", "peix", "lletuga")
    .collect(summarizingDouble(String::length));

System.out.println(longituds);

- Aplicacio (run) X
run:
DoubleSummaryStatistics{count=4, sum=20,000000, min=4,000000, average=5,000000, max=7,000000}
```

```
Persona p1 = new Persona("75849521X", "Pepito López", 32);
Persona p2 = new Persona("94874621T", "Laura Coves", 31);
Persona p3 = new Persona("44746921V", "Ana Cazorla", 24);

IntSummaryStatistics edats = Stream.of(p1,p2,p3)
    .collect(summarizingInt(Persona::getEdat));

System.out.println(edats);

- Aplicacio >
- Aplicacio (run) X
run:
IntSummaryStatistics{count=3, sum=87, min=24, average=29,000000, max=32}
```

Podem obtindre només la mitjana o només la suma dels valors:

```
Double mitjanaEdats = Stream.of(p1,p2,p3)
    .collect(averagingInt(Persona::getEdat));

int sumaEdats = Stream.of(p1,p2,p3)
    .collect(summingInt(Persona::getEdat));

double mitjanalLletres = Stream.of(p1,p2,p3)
    .collect(averagingInt(p -> p.getNom().length()));
```

maxBy / minBy

Els recollents MaxBy/MinBy retornen l'element més gran o més xicotet d'un Stream segons una instància de Comparator proporcionada.

```
Optional<Integer> max = Stream.of(2,8,3,2,5,8,2)
    .collect(maxBy(Comparator.naturalOrder()));
```

Obtenim la mínima persona si l'ordre és per edat:

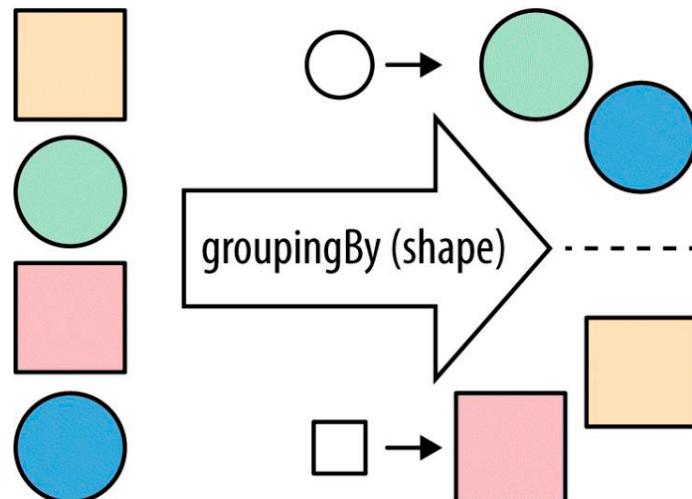
```
Optional<Persona> min = Stream.of(p1,p2,p3)
    .collect(minBy((una, otra) -> una.getEdat() - otra.getEdat()));
```

Una altra manera:

```
Optional<Persona> min = Stream.of(p1,p2,p3)
    .collect(minBy(Comparator.comparingInt(Persona::getEdat)));
```

Podem veure que **retorna un “Optional”** què serà “empty” si no troba cap valor.

groupingBy



El recollent GroupingBy s'utilitza per a agrupar objectes per alguna propietat i després emmagatzemar els resultats en una instància de Map.

Per exemple podem agrupar-los per longitud de cadena:

```
Map<Integer, Set<Persona>> agrupatsPerLletra = Stream.of(p1,p2,p3)
    .collect(groupingBy(p->p.getNom().length(), toSet()));
```

Es poden utilitzar recollents bàsics, per a realitzar algun càlcul

```
Map<Integer, Long> agrupatsPerLletraQuants = Stream.of(p1,p2,p3)
    .collect(groupingBy(p->p.getNom().length(), counting()));
```

```
Map<Integer, Double> agrupatsPerLletraMitjanaEdat = Stream.of(p1,p2,p3)
    .collect(groupingBy(p->p.getNom().length(),
        averagingInt(Persona::getEdat)));
```

Es poden crear diversos nivells d'agrupament:

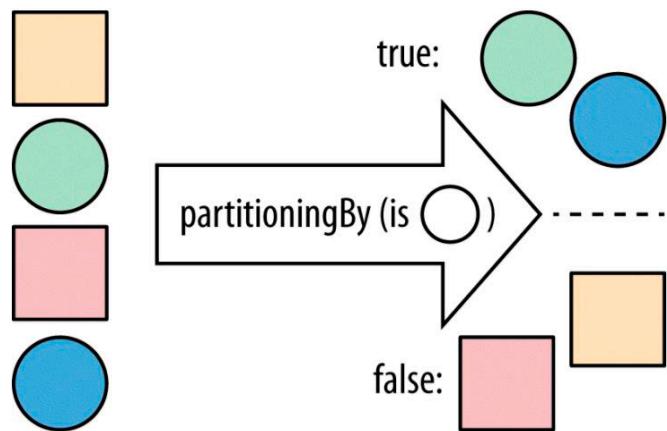
```
Persona p1 = new Persona("75849521X", "Pepito López", 32);
Persona p2 = new Persona("94874621T", "Laura Coves", 31);
Persona p3 = new Persona("44746921V", "Ana Cazorla", 24);
Persona p4 = new Persona("77846512N", "Lorenzo Map", 31);

Map<Integer, Map<Integer, Set<Persona>>> perLletresIEdat = Stream
    .of(p1,p2,p3, p4)
    .collect(groupingBy(Persona::getLength,
        groupingBy(Persona::getEdat, toSet())));
```

Si ometem el toSet suposarà que els elements del grup van en una llista:

```
Map<Integer, Map<Integer, List<Persona>>> perLletresIEdat = Stream
    .of(p1,p2,p3, p4)
    .collect(groupingBy(Persona::getLength,
        groupingBy(Persona::getEdat)));
```

partitioningBy



PartitioningBy és un cas especialitzat de groupingBy que accepta una instància de Predicate i després recopila elements de l'Stream en una instància de Map que emmagatzema valors booleans com a claus i col·leccions com a valors. En la clau "true", podem trobar com a valor una col·lecció d'elements que coincideixen amb el Predicat donat, i en la clau "false" podem trobar com a valor una col·lecció d'elements que no coincideixen amb el Predicat donat.

```
Map<Boolean, List<Persona>> majorsDe30 = Stream.of(p1,p2,p3,p4)
    .collect(partitioningBy(p -> p.getEdat() > 30));
```

```
Map<Boolean, List<Persona>> comencenPer7 = Stream.of(p1,p2,p3,p4)
    .collect(partitioningBy(p -> p.getDni().startsWith("7")));
```

```
Function<Integer, Predicate<Persona>> majorDe =
    i -> p -> p.getEdat() > i;

Map<Boolean, List<Persona>> majorsDe = Stream.of(p1,p2,p3,p4)
    .collect(partitioningBy(majorDe.apply(31)));
```

mapping

De vegades pot passar que volem transformar els elements abans d'agrupar-los. Per això tenim la funció “mapping”.

Abans hem pogut agrupar persones segons la seu edat:

```
Map<Integer, List<Persona>> perEdat = Stream.of(p1,p2,p3,p4)
    .collect(groupingBy(Persona::getEdat));
```

Ens pot interessar agrupar per edat, però no fer grups de persones sinó simplement el seu nom. Per això podem fer:

```
Map<Integer, List<String>> perEdat = Stream.of(p1,p2,p3,p4)
    .collect(groupingBy(Persona::getEdat, mapping(Persona::getNom, toList())));
```

Recorda que amb **CollectingAndThen** primer es recol·lecta, i posteriorment s'aplica una funció al resultat, mentre que amb **mapping** es fa just al contrari, apliquem una funció a cada element abans de recol·lectar al grup.

Vegem un exemple on agrupem per edat i comptem quants elements hi ha en cada grup:

```
Map<Integer, Long> perEdat = Stream.of(p1,p2,p3,p4)
    .collect(groupingBy(Persona::getEdat, counting()));
```

El problema que se'n pot plantejar és que volem desar el nombre d'ocurrències en format Integer, i no Long. Per això farem ús de **collectingAndThen**

```
Map<Integer, Integer> perEdat = Stream.of(p1,p2,p3,p4)
    .collect(groupingBy(Persona::getEdat,
        collectingAndThen(counting(), Long::intValue)));
```

filtering

Igual que amb mapping, de vegades pot passar que desitgem filtrar els elements abans de recol·lectar. Per això tenim la funció “filtering”. Abans hem agrupat per edat, “mapetjant” les persones a noms. Si a més volem filtrar els que tenen una longitud menor de 12 caràcters:

```
Map<Integer, List<String>> perEdat = Stream.of(p1,p2,p3,p4)
    .collect(groupingBy(Persona::getEdat,
        mapping(Persona::getNom,
            filtering(nom -> nom.length() < 12, toList()))));
```

teeing

A partir de Java 12 tenim un recol·lector integrat que s'encarrega d'executar dos recol·lectors i combinar-lo. Tot el que hem de fer és proporcionar els dos recol·lectors i la funció combinadora.

```

int sumaMinMaxEdat = Stream.of(p1,p2,p3,p4)
    .map(Persona::getEdat)
    .collect(teeing(
        minBy(Integer::compareTo),
        maxBy(Integer::compareTo),
        (min, max) -> min.get() + max.get()));

```

toArray

Hi ha vegades que no ens interessa recollir les dades en una col·lecció sinó en un array clàssic. Per això disposem de la funció `toArray`, que es pot utilitzar de dos maneres

<code>Object[] toArray()</code>	<code><A> A[] toArray(IntFunction<A[]> generator)</code>	Returns an array containing the elements of this stream.
		Returns an array containing the elements of this stream, using the provided generator function to allocate the returned array, as well as any additional arrays that might be required for a partitioned execution or for resizing.

La primera i més senzilla simplement tornarà un array (de tipus `Object` o de tipus primitiu), amb els elements que conté l'stream:

```
int[] valors = IntStream.of(3,8,-3,16).toArray();
```

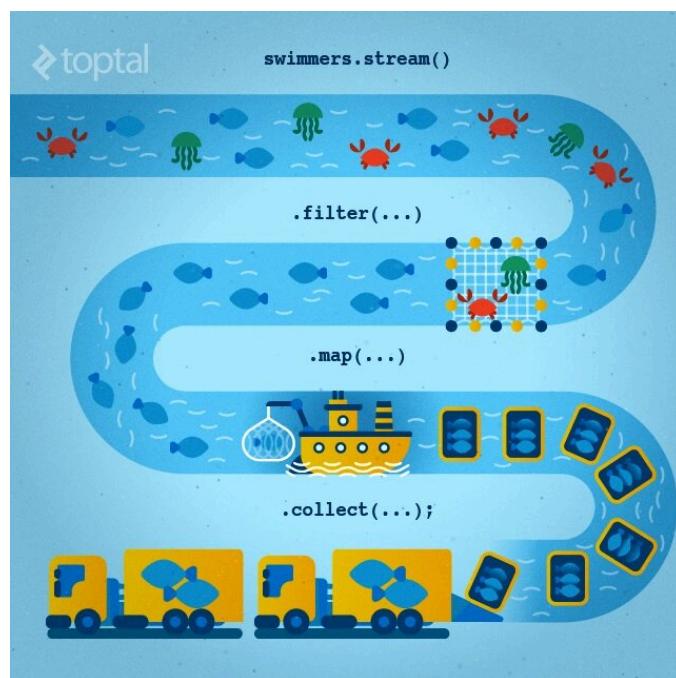
```
double[] valors = DoubleStream.of(3.75, 8.0 , -3, 16.25).toArray();
```

Si el que volem és recollir les dades en un array de tipus de dades complex (un objecte o array) llavors hem d'usar el segon mètode passant-li la funció generadora:

```
String[] arrayStrings = Stream.of("llet", "lletuga", "aigua", "sal").toArray(String[]::new);
```

Este generador també podria ser una funció lambda:

```
String[] arrayStrings = Stream.of("llet", "lletuga", "aigua", "sal").toArray(mida -> new String[mida]);
```



Combinant tot això...

Per a obtindre una llista amb els noms d'aquelles persones majors de 30 anys:

```
List<String> nomsDelsMajorsDe30 = Stream.of(p1,p2,p3,p4)
    .filter(p -> p.getEdat() > 30)
    .map(Persona::getNom)
    .map(String::toUpperCase)
    // .forEach(nom -> nomsDelsMajorsDe30.add(nom)); NOOOOO!!!!
    .collect(toList());
```

Obtindre un “Map” que tinga com a clau el DNI i com a valor l'edat d'aquelles persones majors de 30 anys.

```
Map<String, Integer> dnisIEdatosDelsMajorsDe30 = Stream.of(p1,p2,p3,p4)
    .filter(p -> p.getEdat() > 30)
    .collect(toMap(Persona::getDni, Persona::getEdat));
```

Volem recol·lectar les persones per la màxima edat possible (una o cap persona) i que només recol·lecte el nom.

```
String resultat = Stream.of(p1,p2,p3,p4)
    .collect(collectingAndThen(
        maxBy(comparing(Persona::getEdat)), //Recordem que maxBy retorna Optional
        p -> p.map(Persona::getNom).orElse(" --- ")));
```

Què obtindrem en este cas?

```
Map<Integer, List<String>> resultat = Stream.of(p1,p2,p3,p4)
    .collect(groupingBy(Persona::getEdat,
        mapping(p -> p.getNom().toUpperCase(),
            flatMapping(nom -> Stream.of(nom.split(" ")),
                filtering( lletra -> !lletra.equals(" "), toList())))));
```

I en este cas?

```
Map<Integer, String> resultat = Stream.of(p1,p2,p3,p4)
    .collect(groupingBy(Persona::getEdat,
        mapping(p -> p.getNom().toUpperCase(),
            flatMapping(nom -> Stream.of(nom.split(" ")),
                filtering( lletra -> !lletra.equals(" "), collectingAndThen(toSet(),
                    l -> String.join(" ", l)))))));
```

A partir d'una matriu de 3 columnes i N files volem crear per a cada fila de la matriu un nou triangle. Després volem transformar cada triangle en un double que conginga l'àrea d'eixe triangle, i retornar una llista de Doubles amb les àrees calculades de tots els triangles.

```
double[][] costats = {  
    {10, 7, 5.27},  
    {10, 19.06, 20},  
    {15.64, 16, 20}  
};  
  
List<Double> arees = Stream.of(costats)  
    .map(r -> new Triangle(r[0], r[1], r[2]))  
    .map(Triangle::getArea)  
    .collect(Collectors.toList());
```

I si tinguerem els costats en un array unidimensional?

```
double[] costats = {10, 7, 5.27, 10, 19.06, 20, 15.64, 16, 20};  
  
List<Double> arees = IntStream.range(0, costats.length/3)  
    .mapToObj(i -> new Triangle(costats[3*i], costats[3*i + 1], costats[3*i + 2]))  
    .map(Triangle::getArea)  
    .collect(toList());
```

Convertir el codi següent a l'estil funcional:

```
double[] preus = {2.54, 9.25, 1.23, 0.90, 9.24, 5.05};  
  
double max = 0;  
for (double preu : preus) {  
    if (max < preu) {  
        max = preu;  
    }  
}
```

Solució 1:

```
final double max = Arrays.stream(preus).reduce(0, Math::max);
```

Solució 2:

```
final double max = Arrays.stream(preus).max().getAsDouble();
```

Solució 3

```
final double max = Arrays.stream(preus)  
    .boxed()  
    .sorted(Comparator.reverseOrder())  
    .findFirst()  
    .get();
```

Solució 4

```
final double max = Arrays.stream(preus)
    .boxed()
    .collect(maxBy(naturalOrder()))
    .get();
```

Solució 5:

```
final double max = Arrays.stream(preus)
    .boxed()
    .collect(collectingAndThen(toList(), Collections::max));
```

Converteix la funció següent a l'estil funcional:

```
private static boolean isPrime(final int n){
    for(int i=2; i<n; i++){
        if(n % i == 0) return false;
    }

    return n > 1;
}
```

Solució 1:

```
public static boolean isPrime(final int n) {
    return n > 1
        && IntStream.range(2, n).noneMatch(i -> n % i == 0);
}
```

Solució 2:

```
public static boolean isPrime(final int n) {
    IntPredicate esDivisible = divisor -> n % divisor == 0;
    return n > 1 && IntStream.range(2, n).noneMatch(esDivisible);
}
```

Desenvolupa un mètode que a partir d'una llista d'Integers i un selector (predicat) ens retorna la suma de tots aquells elements que compleixen amb este selector.

```
public static int totalValors(List<Integer> llista, Predicate<Integer> selector) {
    return llista.stream()
        .filter(selector)
        .reduce(0, Integer::sum);
}

public static void main(String[] args) {
    List<Integer> valors = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
    System.out.println(totalValors(valors, e -> e > 8));
    System.out.println(totalValors(valors, Programa::isPrime));
}
```

```

//mala pràctica
final List<Integer> aleatoris = new ArrayList<>();
Stream.iterate(0, i -> i+1)
    .limit(10)
    .forEach(x -> aleatoris.add(new Random().nextInt(10)));

final List<Integer> aleatoris2 = new ArrayList<>();
Stream.iterate(0, i -> i+1)
    .map(x -> new Random().nextInt(10))
    .distinct()
    .limit(10)
    .forEach(aleatoris2::add);

//bona pràctica
final List<Integer> aleatoris3;
aleatoris3 = Stream.iterate(0, i -> i+1)
    .limit(10)
    .map(x -> new Random().nextInt(10))
    .collect(toList());

final List<Integer> aleatoris4;
aleatoris4 = Stream.iterate(0, i -> i+1)
    .map(x -> new Random().nextInt(10))
    .distinct()
    .limit(10)
    .collect(toList());

System.out.println(aleatoris);

```

// EXERCICIS 16 - 20

Extensió d'operacions

Podem crear noves operacions terminals utilitzant el mètode de fàbrica **java.util.stream.Collector.of(...)**; o amb l'ús dels col·lectors predefinits en **Collectors**, creant operacions terminals definides per l'usuari i reutilitzables.

De la mateixa manera podem crear noves operaciones intermèdies utilitzant els mètodes de fàbrica **java.util.stream.Gatherer.of(...)** i **java.util.stream.Gatherer.ofSequential(...)** o utilitzar els recopiladors predefinits en **Gatherers** (esta funció encara es troba en fase PREVIEW a la versió JDK22 de 2024)

Streams infinits

Com hem vist hi ha dues maneres de generar streams infinits, mitjançant els mètodes “generate” i “iterate”.

```
static <T> Stream<T> generate(Supplier<T> s)
    Returns an infinite sequential unordered stream where each element is generated by the provided Supplier.

static <T> Stream<T> iterate(T seed, UnaryOperator<T> f)
    Returns an infinite sequential ordered Stream produced by iterative application of a function f to an initial element seed,
    producing a Stream consisting of seed, f(seed), f(f(seed)), etc.
```

El mètode iterate necessita un primer paràmetre com a seed (llavor) o element inicial i un segon paràmetre que serà un operador unari (una sola dada d'entrada del mateix tipus que seed al qual aplicarem alguna operació)

```
String valors = Stream.iterate(3, i -> i+2)
    .map(String::valueOf)
    .limit(15)
    .collect(joining(", "));

System.out.println(valors);
```

```
// 5 primers valors a partir del 12 que siguen divisibles entre 19
String valors = Stream.iterate(12, i -> i+1)
    .filter(i -> i % 19 == 0)
    .map(String::valueOf)
    .limit(5)
    .collect(joining(", "));
```

D'altra banda la funció generate només necessita un paràmetre que serà de tipus Supplier (procediment generador):

```
Supplier<Integer> proveidor = () -> ++staticInt;

List<Integer> valors = Stream.generate(proveidor)
    .limit(15)
    .collect(toList());

System.out.println(valors);
```

El proveïdor pot ser qualsevol generador que a partir de zero paràmetres d'entrada retorne un valor d'exitida (el cas més utilitzat són els aleatoris)

```
// Obté 10 valors enters aleatoris entre 0 i 100,
// Després els transforma en binari i els guarda en una llista.
List<String> valors = Stream.generate(() -> new Random().nextInt(100))
    .limit(10)
    .map(Integer::toBinaryString)
    .collect(toList());
```

Avaluació mandrosa

Quan executem un càlcul una funció o una expressió, ho podem fer de dos maneres. La primera és de forma “ansiosa” (**eager**) el que suposa executar-la tan prompte com s’obté esta expressió en particular. Per contra, l’avaluació “mandrosa” (**lazy**) suposa posposar esta execució fins que puguem fer-la més endavant o fins al punt que ja no necessitem l’execució i la podem evitar. Per tant, un dels beneficis de ser mandrosos és precisament que podem ser més eficients.

Analitzem el següent codi:

```
public static int computar(int n) {
    System.out.println("Computant...");
    return n;
}

public static void main(String[] args) {
    int x = 4;
    int temp = computar(x);

    if (x > 5 && temp > 5) {
        System.out.println("Resultat");
    } else {
        System.out.println("Sense resultat");
    }
}
```

Java per defecte duu a terme una **avaluació ansiosa**, per la qual cosa executa el mètode “computar”, encara que després mai arriba a necessitar este valor (ja que a l’if s’incompleix l’operand esquerre i mai evalua el dret).

Això parteix de la concepció imperativa i orientada a objectes, fet que fa al llenguatge “temer” per un possible efecte secundari per la qual cosa no pot posposar l’execució.

Ara fixa't en el següent exemple:

```
public static int computar(int n) {
    System.out.println("Computant...");
    return n;
}

public static void main(String[] args) {
    int x = 4;
    Supplier<Integer> temp = () -> computar(x);
    // No estem guardant el resultat de executar la funció
    // Estem guardant un procediment que retorna la funció

    if (x > 5 && temp.get() > 5) {
        System.out.println("Resultat");
    } else {
        System.out.println("Sense resultat");
    }
}
```

En este exemple NO s'executa el mètode “computar” perquè no cal. No obstant això, si assignem per exemple un 10 a la “x” apreciem que sí que s'executarà el mètode. Acabem de crear un **codi mandrós**.

Veurem un exemple amb col·leccions.

Versió tradicional:

```
List<Integer> valors = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

// Trobar el doble del primer nombre parell major que 3
int resultat = 0;
for (int e : valors) {
    if (e > 3 && e % 2 == 0) {
        resultat = e * 2;
        break;
}
System.out.println(resultat);
```

Este codi presenta diversos problemes. Podria ser que en la col·lecció no tinguerem números més grans que 3, o no tinguerem parells o fins i tot que tinguerem una col·lecció buida. En tots estos casos el resultat seria 0, cosa que no és correcta. És un codi molt “familiar”, però molt poc declaratiu. Eliminem la mutabilitat i l'estil imperatiu.

Versió funcional:

```
List<Integer> valors = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

// Trobar el doble del primer nombre parell major que 3
valors.stream()
    .filter(e -> e > 3)
    .filter(e -> e % 2 == 0)
    .map(e -> e * 2)
    .findFirst()
    .ifPresentOrElse(System.out::println,
        () -> System.out.println("Cap"));

// Anem a comparar el nombre d'iteracions!
```

Per tant, el següent codi no executa cap càlcul en no tenir operacions terminals:

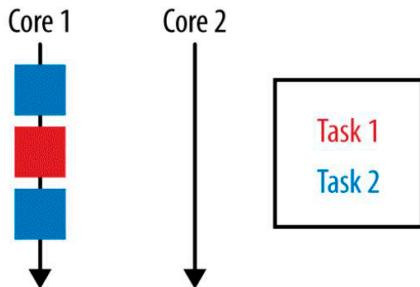
```
valors.stream()
    .filter(e -> e > 3)
    .filter(e -> e % 2 == 0)
    .map(e -> e * 2);
```

Tot això ens permet utilitzar streams infinitos ja que no es farà cap càlcul a priori. A continuació tenim un mètode al qual passem un enter “k” i un altre enter “n”. Haurà de retornar la suma total del doble dels primers “n” nombres parells començant per “k” i l'arrel quadrada de la qual (de cada número) siga més gran que 20.

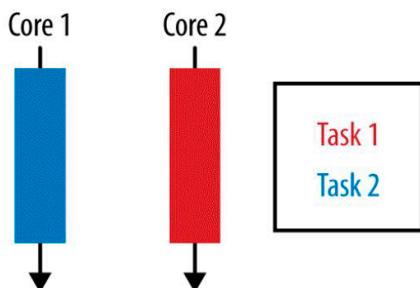
```
public static int computar(int k, int n) {
    return Stream.iterate(k, e -> e + 1)
        .filter(e -> e % 2 == 0)
        .filter(e -> Math.sqrt(e) > 20)
        .mapToInt(e -> e * 2)
        .limit(n)
        .sum();
}
```

Paral·lelitzar

Concurrent but not Parallel



Parallel and Concurrent

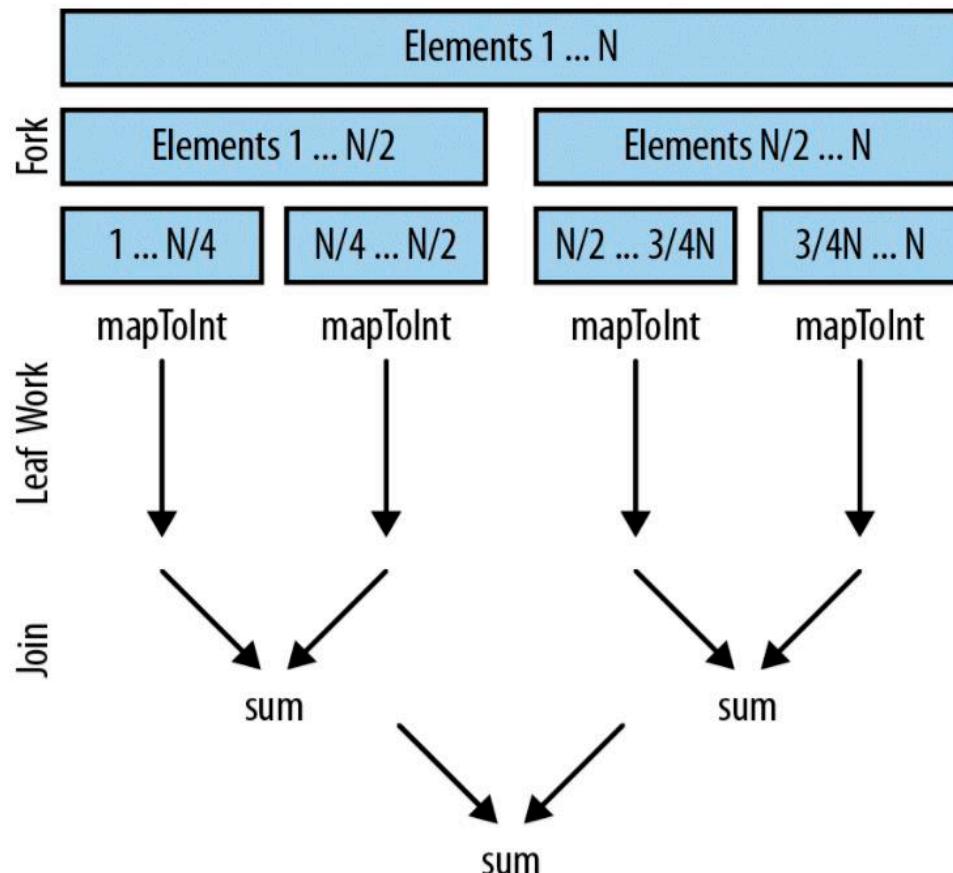


L'objectiu del paral·lelisme és reduir el temps d'execució d'una tasca específica descomponent-la en components més xicotets que es duguen a terme en paral·lel. El paral·lelisme funciona molt bé quan volem aplicar la mateixa operació sobre un gran conjunt de dades.

Si tenim el següent mètode:

```
public static int sumarEnterers(List<Integer> valors) {  
    return valors.parallelStream()  
        .mapToInt(Function.identity())  
        .sum();  
}
```

Java farà el següent:



Vegem una comparació de temps:

```
long start = System.currentTimeMillis();
IntStream.range(1, 1000000).forEach(i -> { i });
long end = System.currentTimeMillis();
System.out.println("Seqüencial: " + (end-start));

start = System.currentTimeMillis();
IntStream.range(1, 1000000).parallel().forEach(i -> { i });
end = System.currentTimeMillis();
System.out.println("Paral·lel: " + (end-start));
```

Seqüencial: **74**

Paral·lel: **11**

Volem trobar la mitjana aritmètica dels divisibles entre 3 des del 10 al 1000000

```
System.out.println(  
    IntStream.range(10, 1000000)  
        .parallel()  
        .filter(i -> i % 3 == 0)  
        .average()  
) ;
```

Si treballem amb paral·lelisme desconeixem l'ordre en què s'executaran els resultats parciaus:

```
Stream.of(1, 2, 3, 4, 5, 6)  
    .parallel()  
    .forEach(System.out::println);
```

```
rur  
4  
6  
5  
2  
3  
1
```

Podem indicar “per a cada element ordenat” (forEachOrdered) de manera que abans de mostrar el resultat col·locarà cada element en la posició que li correspon.

```
Stream.of(1, 2, 3, 4, 5, 6)  
    .parallel()  
    .forEachOrdered(System.out::println);
```

També hem de tenir en compte el paral·lelisme a l'hora de retornar el primer element (findFirst) o algun element que complisca les condicions (findAny):

```
Persona p1 = new Persona("78457415X", "Pere", 23);  
Persona p2 = new Persona("12487482T", "Ana", 22);  
Persona p3 = new Persona("87456984D", "Laura", 19);  
Persona p4 = new Persona("27316372V", "Pere", 35);  
Persona p5 = new Persona("44514996G", "Marc", 18);  
  
System.out.println(  
    Stream.of(p1,p2,p3,p4,p5)  
        .filter(p -> p.getEdat() > 20)  
        .map(Persona::getNom)  
        .map(String::toUpperCase)  
        .findFirst()  
        .orElse("NO TROBAT")  
);
```

El resultat sempre serà “PERE”. Què passarà si en comptes de findFirst usem findAny?

Si fem el stream paral·lel, el resultat de findFirst seguirà sent PERE en tots els casos.

```
Persona p1 = new Persona("78457415X", "Pere", 23);
Persona p2 = new Persona("12487482T", "Ana", 22);
Persona p3 = new Persona("87456984D", "Laura", 19);
Persona p4 = new Persona("27316372V", "Pere", 35);
Persona p5 = new Persona("44514996G", "Marc", 18);

System.out.println(
    Stream.of(p1, p2, p3, p4, p5)
        .parallel()
        .filter(p -> p.getEdat() > 20)
        .map(Persona::getNom)
        .map(String::toUpperCase)
        .findFirst()
        .orElse("NO TROBAT")
);
```

No obstant això, si usem “findAny” amb un stream paral·lel el resultat ja no serà predictable:

```
Persona p1 = new Persona("78457415X", "Pere", 13);
Persona p2 = new Persona("12487482T", "Ana", 22);
Persona p3 = new Persona("87456984D", "Laura", 19);
Persona p4 = new Persona("27316372V", "Pere", 35);
Persona p5 = new Persona("44514996G", "Marc", 18);

System.out.println(
    Stream.of(p1, p2, p3, p4, p5)
        .parallel()
        .filter(p -> p.getEdat() > 20)
        .map(Persona::getNom)
        .map(String::toUpperCase)
        .findAny()
        .orElse("NO TROBAT")
);
```

En eixe últim cas “findAny” pot retornar Ana o Pere dependent del fil d’execució que el trobe abans (treballen en paral·lel).

Recordem el funcionament de la reducció:

```
Persona p1 = new Persona("78457415X", "Pere", 13);
Persona p2 = new Persona("12487482T", "Ana", 22);
Persona p3 = new Persona("87456984D", "Laura", 19);
Persona p4 = new Persona("27316372V", "Pere", 35);
Persona p5 = new Persona("44514996G", "Marc", 18);
Persona p6 = new Persona("87445792J", "Raquel", 23);

int sumaEdats = Stream.of(p1, p2, p3, p4, p5, p6)
    .map(Persona::getEdat)
    .reduce(0, Integer::sum);

System.out.println(sumaEdats);
```

Si tenim un stream paral·lel recordem que es realitzaran les operacions parcials de forma paral·lela, i finalment es posaran en comú mitjançant una funció unidora o “combiner”. Esta funció “combiner” l'hem d'indicar com a tercer paràmetre del mètode “reduce”:

```
int sumaEdats = Stream.of(p1, p2, p3, p4, p5, p6)
    .parallel()
    .map(Persona::getEdat)
    .reduce(0, Integer::sum, Integer::sum);
```

Alguns exemples més per acabar:

```
int resultat = Stream.iterate(1, v -> v+1)
    .limit(100)
    .parallel()
    .reduce(0, Integer::sum, Integer::sum);
```

```
String resultat = Stream.iterate(1, v -> v+1)
    .limit(100)
    .parallel()
    .map(String::valueOf)
    .reduce("0", (p,v) -> String.valueOf(Integer.valueOf(p) + Integer.valueOf(v)));

System.out.println(resultat);
```

```
String resultat = Stream.iterate(1, v -> v+1)
    .limit(100)
    .parallel()
    .map(String::valueOf)
    .reduce("0", (p,v) -> String.valueOf(Integer.valueOf(p) + Integer.valueOf(v)), (p,v) -> p + " " + v);

System.out.println(resultat);
```

// EXERCICIS FINALS

Altres llenguatges

Per exemple en Kotlin, les col·leccions com les llistes o els conjunts ja disposen de mètodes com “filter”, “map”...

```
val numbers = listOf(1, 2, 3, 4, 5)

val evenSquares = numbers
    .filter { it % 2 == 0 }
    .map { it * it }

println(evenSquares) // Output: [4, 16]
```

```
val numbers = listOf(6, 2, 3, 9, 4, 6, 8, 7, 5, 1)

// sum all the numbers in the list using reduce
val sum = numbers.reduce { acc, num -> acc + num }
println("Sum of all numbers: $sum")

// get the three largest numbers using sorted and limit
val largestThree = numbers.sorted().reversed().take(3)
println("Three largest numbers: $largestThree")

// get the distinct numbers using distinct
val distinctNumbers = numbers.distinct()
println("Distinct numbers: $distinctNumbers")
```

```
data class Person(val name: String, val age: Int)

fun main() {
    val people = listOf(
        Person("Alice", 25),
        Person("Bob", 30),
        Person("Charlie", 35),
        Person("David", 20),
        Person("Emily", 28),
        Person("Frank", 25)
    )

    // Exemple de filtrar per edat
    val filteredPeople = people.filter { it.age > 25 }

    // Exemple de reduir les edats de les persones a una sola suma
    val totalAge = people.map { it.age }.reduce { acc, age -> acc + age }

    // Exemple de comprovar si hi ha alguna persona amb edat superior a 30
    val isAnyoneOver30 = people.any { it.age > 30 }

    // Exemple de comprovar si totes les persones tenen més de 18 anys
    val areAllOver18 = people.all { it.age > 18 }

    // Exemple de transformar la llista de persones a una llista de noms
    val names = people.map { it.name }

    // Exemple de tallar la llista de persones a partir del segon element
    val slicedPeople = people.slice(1..3)

    // Exemple de ordenar la llista de persones per edat de menor a major
    val sortedPeople = people.sortedBy { it.age }

    // Exemple de eliminar duplicats de la llista de persones
    val distinctPeople = people.distinctBy { it.age }
```

Si es vol treballar de forma mandra també disposa de “Sequence”:

```
val list = listOf("foo", "bar", "baz", "hello", "world")
val sequence = list.asSequence()
    .filter { it.length > 3 }
    .map { it.toUpperCase() }
    .toList()
```

En JavaScript passa el mateix:

```
// Crear un array de números
const numbers = [1, 2, 3, 4, 5];

// Utilitzar reduce per sumar tots els números de l'array
const sum = numbers.reduce((acc, val) => acc + val, 0);
console.log(sum); // output: 15

// Utilitzar filter per obtenir només els números parells de l'array
const evenNumbers = numbers.filter(val => val % 2 === 0);
console.log(evenNumbers); // output: [2, 4]

// Utilitzar some per comprovar si algun número de l'array compleix una condició
const anyMatch = numbers.some(val => val > 3);
console.log(anyMatch); // output: true

// Utilitzar every per comprovar si tots els números de l'array compleixen una condició
const allMatch = numbers.every(val => val > 0);
console.log(allMatch); // output: true

// Utilitzar map per transformar els números de l'array en strings
const strings = numbers.map(val => `Number ${val}`);
console.log(strings); // output: ["Number 1", "Number 2", "Number 3", "Number 4", "Number 5"]

// Utilitzar slice per obtenir només els primers n elements de l'array
const limit = numbers.slice(0, 3);
console.log(limit); // output: [1, 2, 3]

// Utilitzar sort per ordenar l'array
const unsortedNumbers = [5, 4, 1, 3, 2];
const sortedNumbers = unsortedNumbers.sort((a, b) => a - b);
console.log(sortedNumbers); // output: [1, 2, 3, 4, 5]

// Utilitzar new Set per obtenir els elements únics de l'array
const duplicates = [1, 2, 3, 1, 2];
const uniqueNumbers = [...new Set(duplicates)];
console.log(uniqueNumbers); // output: [1, 2, 3]
```

PER ACABAR. COSES A TINDRE EN COMpte:

JAVA Streams

```
stringLists.stream()  
    .map(str -> str.toUpperCase())  
    .collect(Collectors.toList());
```

Intermediate

filter
distinct
map
flatMap

Terminal

reduce
collect
toArray
count
max
min

findAny
findFirst
forEach
allMatch
anyMatch

Only use a Stream once

```
List<Beer> beers = getBeers();  
Stream<Beer> beerStream = beers.stream();  
  
beerStream.forEach(b ->System.out.println(b.getName())); //1  
beerStream.forEach(b ->System.out.println(b.getAlcohol())); //2
```

Line 2 will give:

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

Be careful with returning a Stream

```
public Stream<Beer> getMeMyBeers()

public void execute() {
    getMeMyBeers() //don't know if it is consumed yet!!!!
    ...
}
```

Consume the stream

```
List<Beer> beers = List.of(new Beer("Heineken", 5.2), new Beer("Delirium Tremens", 9.0), new Beer("Amstel", 5.1));

beers.stream()
    .limit(10)
    .map(i -> i.getAlcohol())
    .peek(i -> {
        if (i > 7.0)
            throw new RuntimeException();
    });
}
```

Usar peek és perillós. Només hem d'utilitzar-lo per a proves.

