BINAR

Java (Stream) Part 2

Silver - Chapter 3 - Topic 5

BINAR

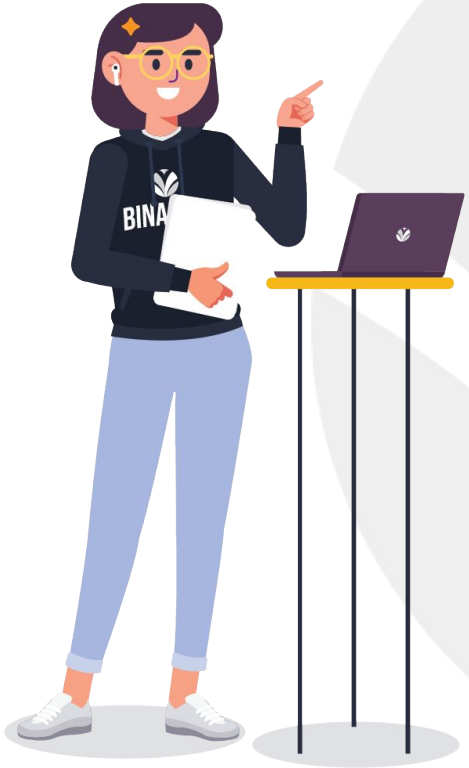
Selamat datang di Chapter 3 Topic 5 online course Back End Java dari Binar Academy!

Kita sampai di penghujung topik pada chapter ini! ✨

Wah.. menurut kamu kerasa atau nggak kerasa nih proses belajar kita selama chapter 3? Iya, soalnya kita udah mau menutup chapter ini dengan topik **Part 2.**

Khusus membahas tentang Stream, **topik ini akan mengenalkan kamu pada aliran data dan cara memanipulasi data.**

Kalau gitu, kita langsung geser aja slide-nya!



Dari sesi ini, kita bakal bahas hal-hal berikut:

- Pengantar Stream
- Cara membuat Stream
- Operasi Transformation, Filtering, Ordering. Aggregate, Check dan forEach
- Konsep Collector

Buat mengawali topic kali ini, kita bakal mulai dengan **Pengantar Stream.**

Sama seperti dalam bahasa Inggris, stream artinya adalah aliran.

Pada Java, Stream bermakna sebagai aliran data.

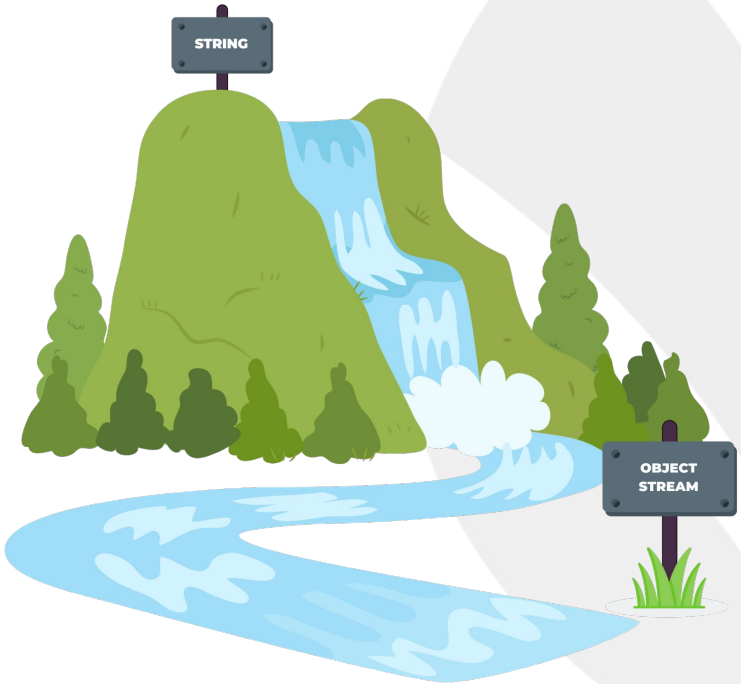
Aliran data pada Java Stream~

Java Stream merupakan fitur baru di Java yang berupa aliran data.

Sederhananya, Java Stream adalah **aliran data yang mengalir, memanipulasi data tanpa data tersebut disimpan terlebih dahulu**.

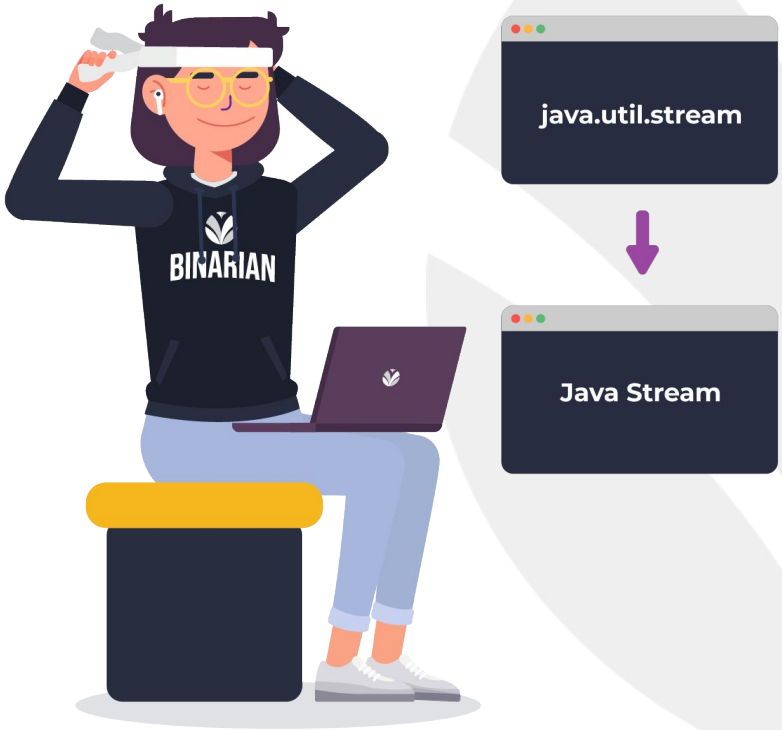
Sama kayak air yang mengalir dari atas ke bawah dan punya tujuan akhirnya, Java Stream juga punya sumber dan ada tujuannya.

Kalau di Java Stream, sumbernya berupa string yang tujuan atau hasil akhirnya adalah object stream.



Selain itu, Java Stream memakai **lambda expression** dan berhubungan juga sama Java Collection.

Dengan kata lain, Java Stream adalah implementasi dari class **java.util.stream**



“Java Stream itu kenapa penting ya?”

Karena Java Stream ini adalah fitur yang ada di Java 8.

Dengan memahami Stream, maka kamu bisa belajar teknologi Java. Terutama yang digunakan dalam membuat data mengalir kayak video dan data real time lainnya.

Stream itu sebenarnya Object lho!

Object stream bukan merupakan suatu container untuk menyimpan data.

Hal ini karena object stream nggak bakal mengubah data yang menjadi source-nya, ketika beroperasi atau mengalir.

Karena data pada Stream ini nggak disimpan, object Stream yang udah dioperasikan atau dialirkan jadi **nggak bisa dipakai secara berulang-ulang**.

Dari situ, object tersebut harus dibikin baru untuk dialirkan lagi.

Oh iya walaupun harus dibikin baru, si object Stream nggak bakal mengalir kalau nggak diperintahkan buat mengalir.

“Terus supaya kita bisa memerintahkan object Stream untuk mengalir itu kayak gimana?”

Nice question. Sekarang kita menjawab pertanyaan itu dengan belajar cara **Pembuatan Stream.**

BINAR

Coba kita perhatikan contoh di bawah ini dulu, ya!

```
Stream<String> stream1 = Stream.of("abc");

Stream<String> stream2 = Stream.empty();

String data = null;
Stream<String> stream3 = Stream.ofNullable(data);

Stream<String> stream4 = Stream.of("abc", "def", "ghi");

Stream<String> stream5 = Arrays.asList("abc", "def",
"ghi").stream();
```

Dari contoh tadi, kesimpulannya kayak gini, gengs~

Seluruh contoh dari stream tersebut merupakan pembuatan Stream dari suatu String.

- **Stream1** adalah contoh dari pembuatan object stream dengan sebuah String.
- **Stream2** adalah contoh dari pembuatan object Stream String yang kosong.

```
Stream<String> stream1 = Stream.of(“abc");

Stream<String> stream2 = Stream.empty();

String data = null;
Stream<String> stream3 = Stream.ofNullable(data);

Stream<String> stream4 = Stream.of(“abc”, “def”, “ghi”);

Stream<String> stream5 = Arrays.asList(“abc”, “def”,
“ghi”).stream();
```

- **Stream3** merupakan contoh dari pembuatan object Stream String dengan data yang isinya null. Di kasus kayak gini, object stream yang dibuat bakal jadi object yang empty.
- **Stream4** merupakan contoh stream yang menggunakan array of string.
- **Stream5** merupakan contoh stream yang menggunakan collections.

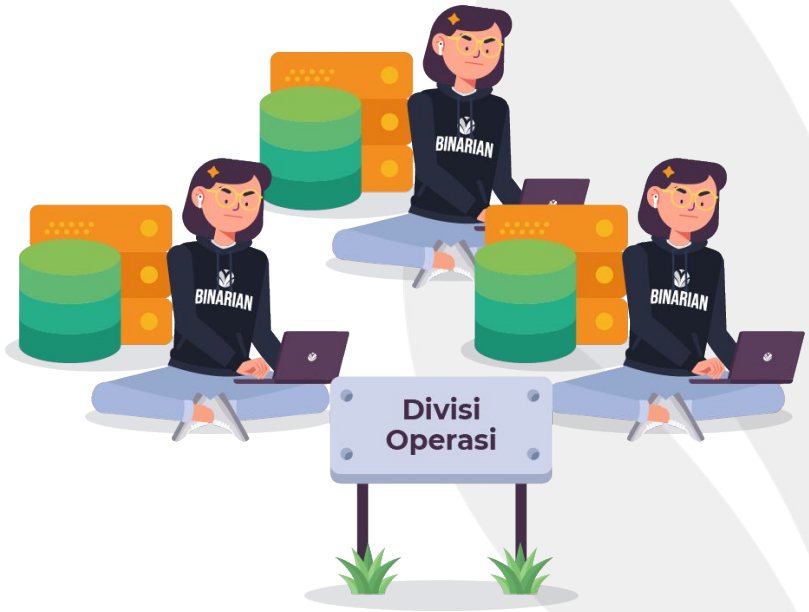
```
Stream<String> stream1 = Stream.of("abc");

Stream<String> stream2 = Stream.empty();

String data = null;
Stream<String> stream3 = Stream.ofNullable(data);

Stream<String> stream4 = Stream.of("abc", "def", "ghi");

Stream<String> stream5 = Arrays.asList("abc", "def",
"ghi").stream();
```

“Kalau Stream-nya udah dibikin, berarti langsung bisa dialirkan, dong?”

Well.. sayangnya belum.

Pas stream udah dibikin, stream tersebut belum bisa dialirkan. Melainkan **stream harus dioperasikan terlebih dahulu**.

Ngomongin pengoperasian Stream, setelah Stream udah jadi, bakal ada berbagai operasi yang bisa dipakai.



“Ada apa aja sih cara pengoperasian Stream?”

Okey deh, tanpa berlama-lama lagi, sekarang kita bakal kepoin tentang **Stream Operation.**

Sebuah stream bisa punya banyak operasi lho, gengs~

Stream yang udah bergerak dan beroperasi, nantinya bakal jadi sebuah **Pipeline.**

Pada gambar di samping ini, ada contoh dari suatu pipeline stream. Perhatikan baik-baik yaaaa~

Oh iya, operasi pada streams bisa dibagi jadi dua, yaitu **intermediate operation** dan **terminal operation**.

```
configs.stream().map(config ->
config.getConfigIdentifier().getWhite
listCode())

.collect(Collectors.toSet());
```

BINAR

Intermediate Operation

Yaitu operasi yang bakal **mengembalikan value stream pas udah beroperasi**.

Dari situ kita bisa melakukan chaining atau melanjutkan dengan operasi lain terhadap operasi ini.

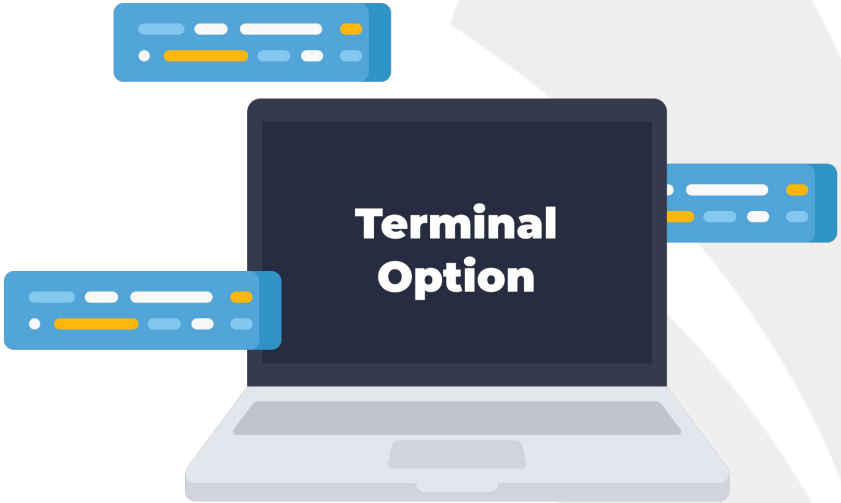
Dalam suatu pipeline, stream bisa punya lebih dari satu intermediate operation. Pas intermediate operation dipakai, stream belum mengalir.

Terminal Operation

Yaitu operasi pada stream yang bisa **men-trigger stream untuk mengalir atau berjalan, sekaligus mengakhiri sebuah stream.**

Sebuah pipeline stream bisa beroperasi tanpa intermediate operation, tapi nggak bisa beroperasi tanpa adanya terminal operation.

Ibarat lagu yang liriknya kayak gini “aku tanpamu, butiran debu~”



Ada catatan penting nih. Si terminal operation nggak akan mengembalikan value berupa stream.

“kenapa?”

Karena terminal operation dipakai untuk mengakhiri sebuah stream, jadi operasinya selalu terletak di akhir pipeline.

Kita balik lagi ke contoh pipeline tadi ya, gengs~

Pada contoh tersebut, ada satu intermediate operation dan juga terminal operation. Detailnya kayak gini, nih~

- **Intermediate operation** yang dipakai adalah map()
- **Terminal operation** yang dipakai adalah collect()

```
configs.stream().map(config ->
config.getConfigIdentifier().getWhite
listCode())

.collect(Collectors.toSet());
```

Berikut adalah perbedaan dari Intermediate Operation dengan Terminal Operation ya, sob!

Intermediate Operation	Terminal Operation
Me-return stream object	Tidak me-return stream object
Lazy, atau tidak mentrigger stream untuk berjalan	Mentrigger stream untuk berjalan
Di dalam suatu pipeline bisa terdapat lebih dari satu	Di dalam suatu pipeline hanya ada satu
Tidak mengakhiri sebuah stream	Mengakhiri sebuah stream
Bisa dilanjutkan (di-chaining) dengan operasi lain	Tidak bisa dilanjutkan (di-chaining) dengan operasi lain

Kalau yang ini adalah contoh operasi dari Intermediate Operation dan Terminal Operation~

Intermediate Operation	Terminal Operation
Contoh operasinya yaitu: • filter() • map() • distinct() • sorted() • limit() • skip()	Contoh operasinya yaitu: • forEach() • toArray() • reduce() • collect() • min() • max() • count() • anyMatch() • allMatch() • noneMatch() • findFirst() • findAny()

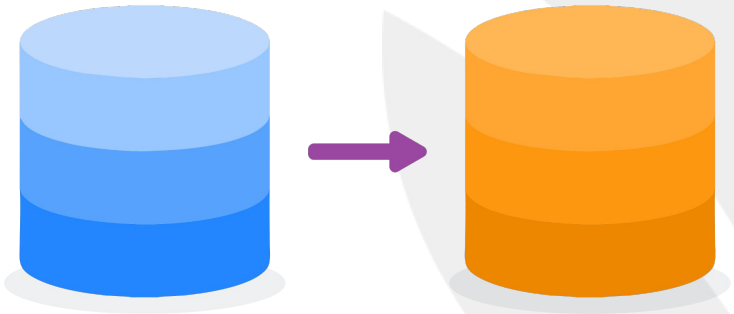
Masih berkaitan sama dua operasi, yaitu Intermediate dan Terminal Operation, kali ini kita bakal lebih detail masuk ke contoh operasi. Mencakup:

- Transformation,
- Filtering,
- Ordering, Aggregate,
- Check, dan.
- forEach.

Contoh operasi yang pertama adalah transformation~

Transformation adalah proses untuk mengubah suatu data jadi data yang lain.

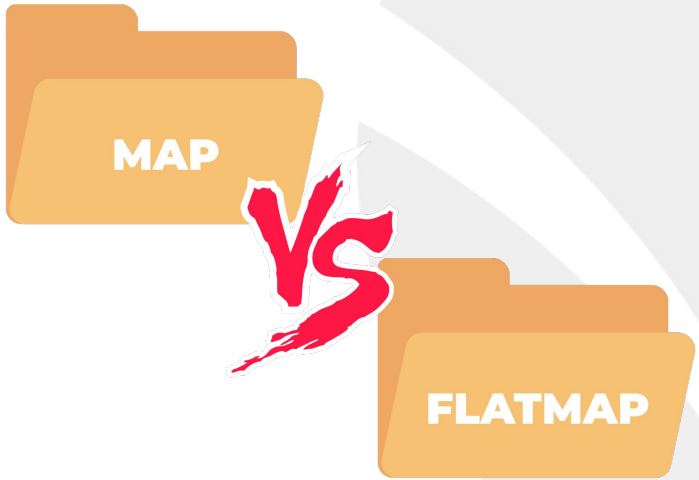
- Intermediate operation yang dipakai untuk melakukan transformation, yaitu **map()** dan **flatMap()**
- Map dan flat map sama-sama punya parameter yang berupa sebuah **Lambda**



“Emang bedanya map() sama flatMap() itu apa sih?”

Bedanya ada di returnnya, gengs.

- Map punya return stream sedangkan flatMap punya return stream dari stream yang baru.
- Map nggak mengubah struktur dari suatu stream, sedangkan flatmap mengubah struktur dari suatu stream.



Berikut adalah contoh dari transformation yang memakai map!

```
List<ConfigReadModel> configs =
configRepository.findByFieldCode(updateValue.getFieldCode());
List<String> whitelistCode = configs.stream().map(config ->
config.getConfigIdentifier().getWhitelistCode())
                            .collect(Collectors.toSet())
```

Pada contoh di atas, transformation dilakukan untuk mendapatkan List yang isinya String dari source yang memakai List dari object ConfigReadModel.

Dari contoh tersebut kira-kira apa lagi ya yang bisa kita temukan?

Jadi, pada contoh tersebut, kita juga bisa menemukan penggunaan map yang mengeksekusi Lambda function.

Lambda tersebut mengeksekusi:

config.getConfigIdentifier().getWhitelistCode()

pada setiap elemen dari List ConfigReadModel.

Pas udah mendapatkan value yang diinginkan pakai map, terminal operation bakal dipakai buat menghimpun value tersebut.

Terminal operation yang digunakan adalah collect().

Sedangkan operasi collect() dipakai buat menghimpun suatu stream value untuk menjadi suatu collections.

BINAR

Berikut adalah contoh dari suatu transformation yang memakai flatMap():

```
List<String> list1 = Arrays.asList("a","b","c");
      List<String> list2 = Arrays.asList("d", "e", "f");

      List<List<String>> listOfLists = Arrays.asList(list1, list2);

      List<String> listOfString = listOfLists.stream()
              .flatMap(p -> p.stream())
              .collect(Collectors.toList());
      System.out.println(listOfString)
```

Output dari contoh tersebut adalah **[a, b, c, d, e, f]**

Dari contoh tersebut, kita bisa lihat kalau source dari stream tersebut merupakan list dari list.

Kalau dibentuk kayak array, jadinya kayak gini:

[[“a”,”b”,”c”] , [“d”,”e”,”f”]]

Tapi kalau pakai flatmap(), kita bisa menggabungkan keduanya jadi gini:

[“a”, “b”, “c”, “d”, “e”, “f”]

Contoh operasi stream yang kedua adalah Filter~

Eits, ini bukan filter buat bikin wajah kamu jadi lebih glowing, ya!

Filter ini merupakan operasi dari intermediate yang bisa dipakai untuk **mem-filter atau menyeleksi suatu data.**

Misalnya, ada suatu class bernama player yang bakal menyeleksi player yang punya point lebih dari 100.

```
public class Player {
    private String name;
    private int points;
}
```

Nantinya kita bakal menyeleksi player dengan point lebih dari 1000 kayak gini:

```
List<Player> playersWithMoreThan1000Points = players
  .stream()
  .filter(p -> p.getPoints() > 1000)
  .collect(Collectors.toLst());
```

Berarti list playersWithMoreThan1000Points merupakan player dengan point di atas 1000

Operasi stream yang ketiga adalah Ordering~

Ordering adalah operasi yang berkaitan dengan pengurutan**.** Dalam melakukan ordering ini, kita harus menggunakan sort.

Dari contoh ordering di samping, kalau kita nggak mendefine kriteria sortingnya, maka outputnya bakal diurutkan dari **abjad yang paling awal ke yang paling akhir** kayak gini:

- Abas
- Abi
- Abu

```
List<String> list =
Arrays.asList("Abu", "Abi",
"Abas");

List<String>
sortedList =
list.stream().sorted().collect(
Collectors.toList());


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

Tapi, kalau mau melakukan sorting pakai kriteria yang bermacam-macam, ya kita bisa menggunakan parameter **Comparator** di method **sorted()**.

Misalnya, kita mau mengurutkan ke arah sebaliknya, maka outputnya jadi kayak gini:

- Abu
- Abi
- Abas

```
List<String> list =
Arrays.asList("Abu", "Abi",
"Abas");

            List<String>
sortedList = list.stream()

.sorted(Comparator.reverseOrder
())

.collect(Collectors.toList());

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

“Terus, gimana kalau kita mau melakukan ordering di suatu collection of objects?”

Okey, kalau gitu kita coba pake contoh class Player lagi ya! Pantengin terus slide setelah ini~

```
public class Player {
    private String name;
    private int points;
}
```

Misalnya, kita bakal mengurutkan point dari yang paling besar ke yang paling kecil.

Gini, nantinya sortedList bakal menghasilkan Player dengan point paling besar sampai ke paling kecil.

Kalau mau mengubah dari yang paling kecil ke yang paling besar, kita bisa menghilangkan method reversed()-nya. Sederhana kan?

```
List<Player> sortedList =
users.stream()

.sorted(Comparator.comparingInt(Player:
:getPoints).reversed())
  .collect(Collectors.toList());
```

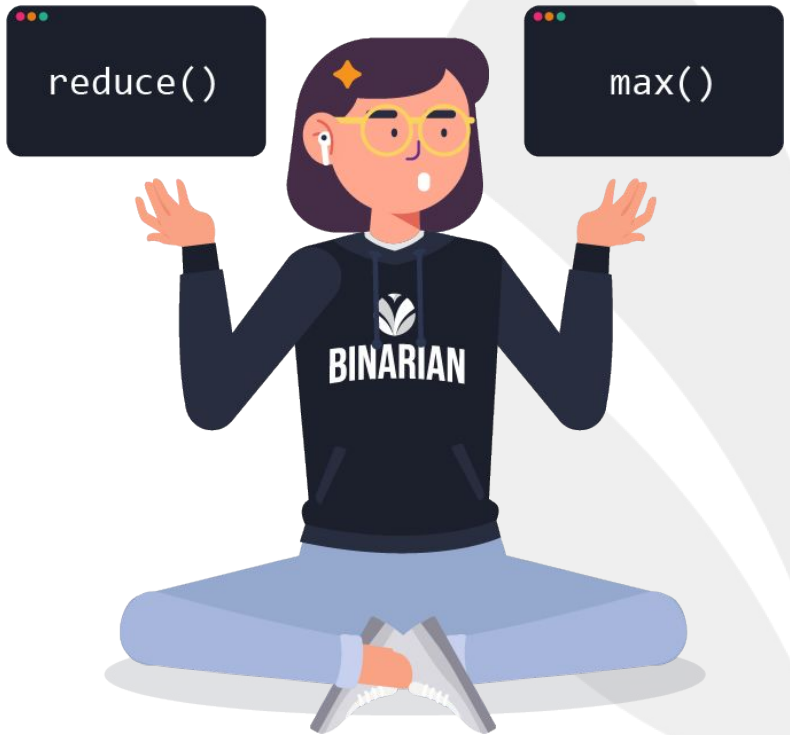
```
sortedList.forEach(System.out::println);
```

Lanjut! Di operasi stream yang keempat ada Aggregate

Aggregate merupakan operasi untuk melakukan suatu **penghitungan agregat dari suatu stream.** Method yang digunakan yaitu:

- **reduce()**
- **max()**

Kita bahas satu-satu ya!



- **reduce()** merupakan operasi yang dipakai dalam melakukan akumulasi.

  Contohnya kayak yang di samping ya, gengs!

```
int sum = 0;
for (int i : numbers) {
    sum += i;
  }
```

Menjadi

```
int sum = Arrays.stream(numbers).reduce(0, Integer::sum);
```

- **max()** merupakan operasi yang dipakai untuk mencari nilai maksimum.

```
Optional<Integer> maxOptional = numList.stream().max(Comparator.comparing(Integer::intValue));
  maxOptional.ifPresent(e -> System.out.println("Max: " + e));
```

Oh iya, ada juga operasi min() buat mencari nilai minimum, lho.

Stream juga bisa melakukan operasi check dengan terminal operation

Operasi check ini dipakai untuk melakukan pencocokan stream yang punya return boolean.

Beberapa operasi tersebut antara lain:

- **anyMatch()** yang dipakai buat mencari kalau ada yang match.
- **allMatch()** dipakai kalau mau mencocokkan keseluruhan stream noneMatch kalau nggak ada yang cocok.

Penasaran sama contohnya? Berikut adalah salah satu contohnya, gengs~

```
List<String> words = Arrays.asList("abc", "def");
boolean anyMatch = words.stream.anyMatch(word → word.length() > 3);
```

Dari contoh tersebut, maka value dari anyMatch adalah false karena nggak ada yang cocok.

Hayooo, masih ingat sama penggunaan forEach di topic sebelumnya nggak?

forEach() di topic ini adalah operasi yang mirip sama forEach pada Lambda di topic 4.

Jadi, kalau pakai method forEach(), kita bisa melakukan perulangan. Sekarang kamu jadi ingat, kan?

forEach ini adalah terminal operation. Jadi pada suatu pipeline, forEach bisa melakukan intermediate operation terlebih dahulu sebelum melakukan operasi forEach.

Perhatikan contoh penggunaan forEach() pada stream di samping, ya!

```
List<String> words = Arrays.asList("abc", "def");
words.stream().forEach(System.out::print);
```

Masuk ke pembahasan materi terakhir di topik ini, kita bakal ngomongin tentang **konsep Collector.**

Pada Collector ini ada berbagai method yang bisa kamu pakai pada Stream.

“Jadi, Apa itu Collectors?”

Collectors adalah suatu class yang isinya bermacam-macam method yang bisa dipakai untuk **memanipulasi output dari stream.**

Buat pakai class ini, kita harus menggunakan **terminal operation collect()** pada pipeline.

“Emangnya tujuan memanipulasi stream itu buat apa?”

Kalau pakai collectors, kita bisa memanipulasi stream supaya bisa menghasilkan collection kayak list dan set.

Selain itu, kita juga bisa menggunakannya buat mencari nilai rata-rata dan masih banyak lagiiii~

Oh iya! di collector ini, ada sekitar **43 macam method** yang bisa kita pakai, lho. Keren, kan?!

Berikut adalah contoh dari penggunaan collectors untuk menghasilkan data set.

Yap! Jadi, dicontoh ini kita bakal mengambil student yang punya umur di atas 18.

```
class Student{
   int id;
   String name;
   int age;
   public Student(int id, String name, int age) {
    this.id = id;
    this.name = name;
    this.age = age;
   }
}
List<Student> studentlist = new ArrayList<Student>();
    studentlist.add(new Student(11,“Bobon",22));
    studentlist.add(new Student(22,“Bibin",18));
    studentlist.add(new Student(33,“Beben",22));
    Set<Student> students = studentlist.stream()
                        .filter(n→ n.id>18)
                        .collect(Collectors.toSet());
```

Kalau yang ini adalah contoh dari penggunaan collectors dalam menghitung suatu rata-rata.

Jadi, kita bakal menghitung rata-rata dari umur para student

Simple kan? Simple dong~

```
class Student{
   int id;
   String name;
   int age;
   public Student(int id, String name, int age) {
    this.id = id;
    this.name = name;
    this.age = age;
   }
}
Double avgAge = studentlist.stream()
        .collect(Collectors.averagingInt(s->s.age));
    System.out.println("Average Age of Students is: "+avgAge);
```

Misi terakhir di topic 5~

Silakan **implementasikan Stream pada project Java yang kamu buat di pertemuan sebelumnya**!

Latihan ini dilakukan di kelas dan silakan diskusikan hasil jawaban kamu dengan teman sekelas dan fasilitator.

Selamat mencoba, yaa~

Wah, nggak terasa udah sampai di penghujung materi Java 8 - Stream!

Di materi ini, kita udah mengenal banyak contoh operasi seperti Transformation, Filtering, Ordering, Aggregate, Check, dan forEach.

Boleh dong sob ceritain, dari contoh operasi tersebut, mana yang paling challenging ketika kamu pelajari? Alasannya kenapa?

BINAR

Saatnya Quiz

1

Berikut ini adalah pernyataan yang salah mengenai stream, yaitu...

A. Sebuah stream akan beroperasi pada saat suatu stream dideklarasi
B. Sebuah stream bisa bernilai empty
C. Sebuah stream bisa digunakan untuk mengkonversi suatu data ke data lain

1

A. Sebuah stream akan beroperasi pada saat suatu stream dideklarasi

Stream bersifat lazy, atau tidak akan bergerak sampai ada yang mentrigger stream tersebut untuk beroperasi

2

Operasi pada suatu stream yang dapat men-trigger suatu stream agar dapat beroperasi adalah....

A. Intermediate operation
B. Terminal operation
C. Arithmetic operation

2

B. Terminal operation

Terminal operation merupakan suatu operasi yang digunakan untuk mentrigger stream agar berjalan

BINAR

Saatnya Quiz

1

Berikut ini adalah pernyataan yang salah mengenai stream, yaitu...

A. Sebuah stream akan beroperasi pada saat suatu stream dideklarasi
B. Sebuah stream bisa bernilai empty
C. Sebuah stream bisa digunakan untuk mengkonversi suatu data ke data lain

BINAR

1

A. Sebuah stream akan beroperasi pada saat suatu stream dideklarasi

Stream bersifat lazy, atau tidak akan bergerak sampai ada yang mentrigger stream tersebut untuk beroperasi

2

Operasi pada suatu stream yang dapat men-trigger suatu stream agar dapat beroperasi adalah....

A. Intermediate operation
B. Terminal operation
C. Arithmetic operation

2

B. Terminal operation

Terminal operation merupakan suatu operasi yang digunakan untuk mentrigger stream agar berjalan

3

Berikut pernyataan yang salah mengenai pipeline, kecuali....

A. Sebuah pipeline bisa terdiri dari satu intermediate operation dan banyak terminal operation
B. Sebuah pipeline diperbolehkan memiliki banyak intermediate operation
C. Sebuah pipeline bisa tereksekusi tanpa adanya terminal operation

BINAR

Saatnya Quiz

1

Berikut ini adalah pernyataan yang salah mengenai stream, yaitu...

A. Sebuah stream akan beroperasi pada saat suatu stream dideklarasi
B. Sebuah stream bisa bernilai empty
C. Sebuah stream bisa digunakan untuk mengkonversi suatu data ke data lain

BINAR

1

A. Sebuah stream akan beroperasi pada saat suatu stream dideklarasi

Stream bersifat lazy, atau tidak akan bergerak sampai ada yang mentrigger stream tersebut untuk beroperasi

2

Operasi pada suatu stream yang dapat men-trigger suatu stream agar dapat beroperasi adalah....

A. Intermediate operation
B. Terminal operation
C. Arithmetic operation

2

B. Terminal operation

Terminal operation merupakan suatu operasi yang digunakan untuk mentrigger stream agar berjalan

3

B. Sebuah pipeline diperbolehkan memiliki banyak intermediate operation

Sebuah pipeline bisa memiliki banyak intermediate operation namun hanya memiliki satu intermediate operation

4

Yang merupakan contoh dari terminal operation adalah.....

A. filter()
B. map()
C. forEach()

3

B. Sebuah pipeline diperbolehkan memiliki banyak intermediate operation

Sebuah pipeline bisa memiliki banyak intermediate operation namun hanya memiliki satu intermediate operation

4

Yang merupakan contoh dari terminal operation adalah.....

A. filter()
B. map()
C. forEach()

4

C. forEach()

forEach() merupakan satu contoh operasi dari map

5

Berikut pernyataan yang benar pada terminal operation, yaitu...

A. Terminal operation tidak dapat di-chaining dengan operation lain

B. Terminal operation dapat men-chaining intermediate operation

C. Terminal operation bukan merupakan operasi terakhir pada suatu pipeline

5

B. Terminal operation dapat men-chaining intermediate operation

Terminal operation bisa men-chaining intermediate operation namun tidak bisa men-chaining terminal operation

Nah, selesai sudah pembahasan kita di Chapter 3 🥳

Ngomong-ngomong, nih. Pada Chapter 4 nanti, kita akan mulai masuk ke level GOLD, lho~

Kita bakal ngobrolin tentang Spring Framework sampai Database Operation. So, see you soon~