


Java8

SOFTWARE INGENIARITZA

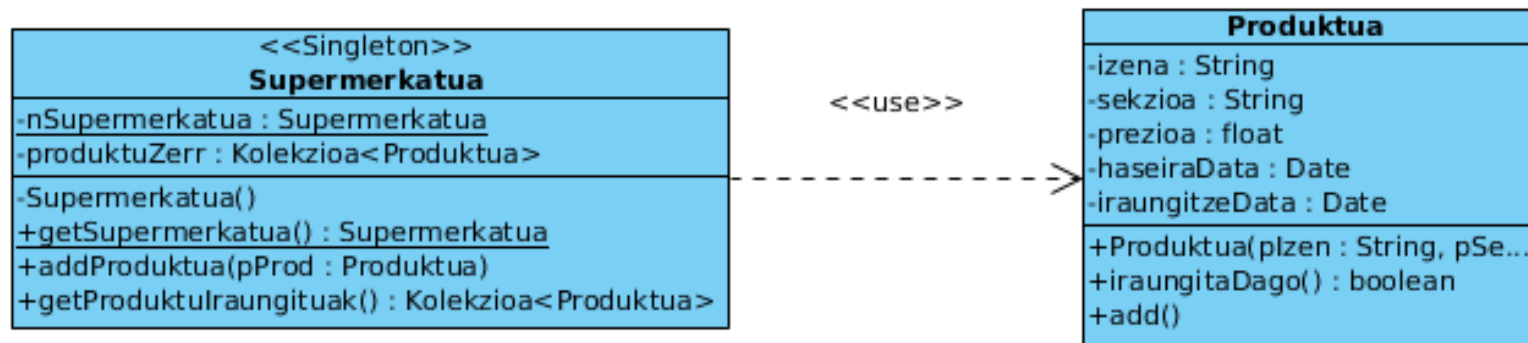


EDUKIAK

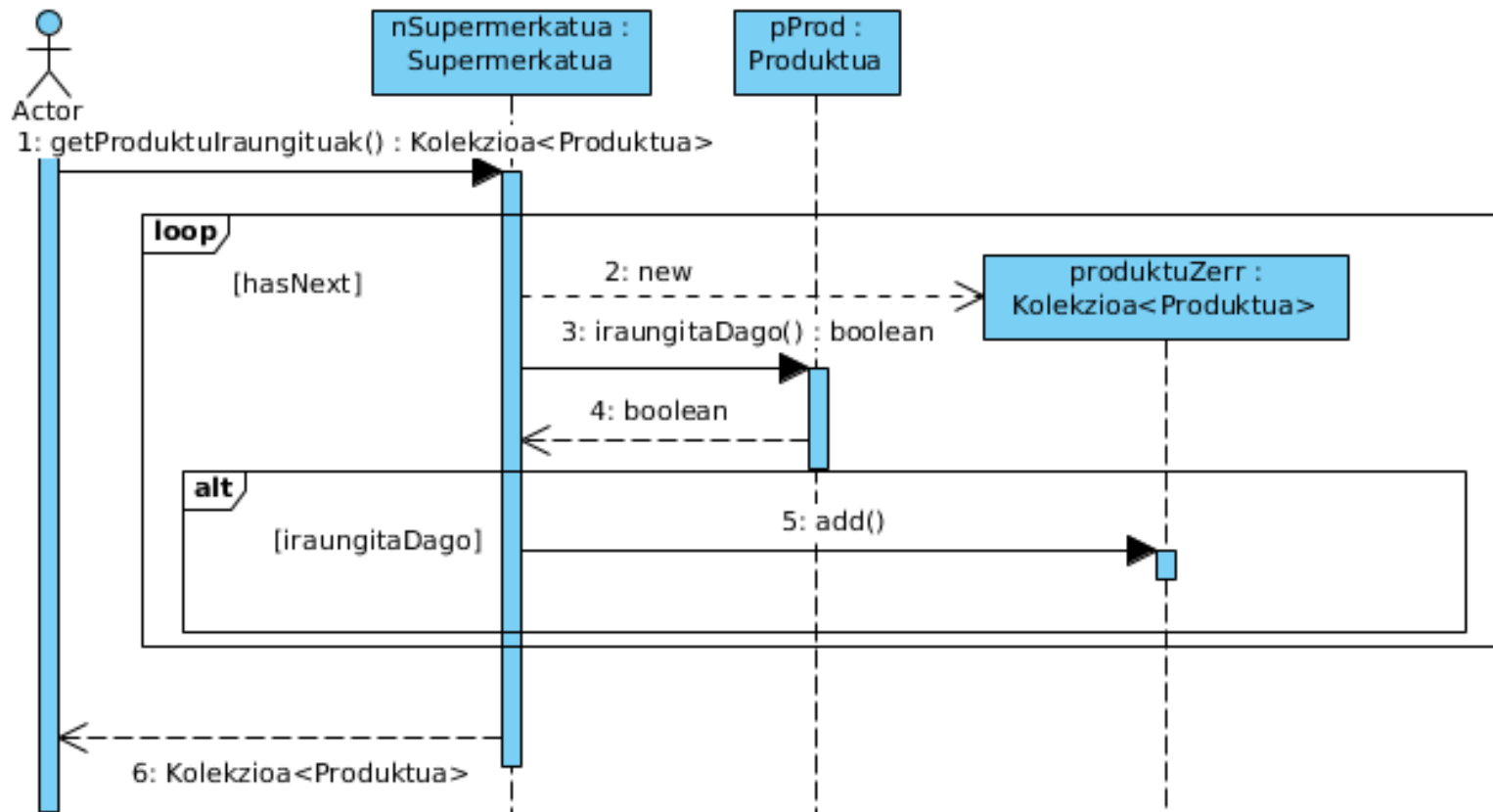
- ▶ Sarrera
 - ▶ Behaviour parametrization
 - ▶ Interfaze funtzionalak
 - ▶ Lambda espresioak
 - ▶ Stream eta agregazio operazioak
 - ▶ Interfazeak
- 

Sarrera

Supermerkatu klasean *produktulraungituakEman()* kodetzeko eskatu digute. Azken horrek iraungitako produktuen zerrenda bueltatzen du.




Sekuentzia diagrama



Soluzio posible bat

```
public List<Produktua> getProduktuIraungituak() {  
    List<Produktua> iraungituak = new ArrayList<>();  
    for (Produktua produktua : produktuak) {  
        if (produktua.iraungitaDago()) {  
            iraungituak.add(produktua);  
        }  
    }  
    return iraungituak;  
}
```



Hurrengoak eskatuz gero...

Iraungitakoen zerrenda:

```
public List<Produktua> getProduktuIraungituak() {  
    List<Produktua> iraungituak = new ArrayList<>();  
    for (Produktua produktua : produktuak) {  
        if (produktua.iraungitaDago())  
            iraungituak.add(produktua);  
    }  
    return iraungituak;  
}
```

2. sekziokoen zerrenda:

```
public List<Produktua> getProduktuSekzio2(){  
    List<Produktua> sekziokoak =newArrayList<>();  
    for (Produktua produktua : produktuak){  
        if (produktua.getSekzioa().equals("2"))  
            sekziokoak.add(produktua);  
    }  
    return sekziokoak;  
}
```

12 euro baino garestiagoen zerrenda:

```
public List<Produktua> getProduktuKostu() {  
    List<Produktua>kostukoak =newArrayList<>();  
    for(Produktua produktua : produktuak) {  
        if (produktua.getPrezio() > 12)  
            kostukoak.add(produktua);  
    }  
    return kostukoak;  
}
```

Hurrengoak eskatuz gero...

Iraungitakoen zerrenda:

```
public List<Produktua> getProduktuIraungituak() {  
    List<Produktua> iraungituak = new ArrayList<>();  
    for (Produktua produktua : produktuak) {  
        if (produktua.iraungitaDago())  
            iraungituak.add(produktua);  
    }  
    return iraungituak;  
}
```

Aldaketa lerro
bakarrean, baina
hiru metodo!!!

2. sekziokoen zerrenda:

```
public List<Produktua> getProduktuSekzio2() {  
    List<Produktua> sekziokoak = new ArrayList<>();  
    for (Produktua produktua : produktuak) {  
        if (produktua.getSekzioa().equals("2"))  
            sekziokoak.add(produktua);  
    }  
    return sekziokoak;  
}
```

12 euro baino garestiagoen zerrenda:

```
public List<Produktua> getProduktuKostu() {  
    List<Produktua> kostukoak = new ArrayList<>();  
    for (Produktua produktua : produktuak) {  
        if (produktua.getPrezio() > 12)  
            kostukoak.add(produktua);  
    }  
    return kostukoak;  
}
```

Hurrengoak eskatuz gero...

Iraungitakoen zerrenda:

```
public List<Produktua> getProduktuIraungituak() {  
    List<Produktua> iraungituak = new ArrayList<>();  
    for (Produktua produktua : produktuak) {  
        if ((produktua.iraungitaDago()))  
            iraungituak.add(produktua);  
    }  
    return iraungituak;  
}
```

**Betekizunen aldaketen aurrean,
nola berrerabili antzeko kodea?**

```
public List<Produktua> getProduktuSekzio2(){  
    List<Produktua> sekziokoak =newArrayList<>();  
    for (Produktua produktua : produktuak){  
        if ((produktua.getSekzioa().equals("2")))  
            sekziokoak.add(produktua);  
    }  
    return sekziokoak;  
}
```

```
public List<Produktua> getProduktuKostu() {  
    List<Produktua>kostukoak =newArrayList<>();  
    for(Produktua produktua : produktuak) {  
        if ((produktua.getPrezio() > 12))  
            kostukoak.add(produktua);  
    }  
    return kostukoak;  
}
```


Behaviour parametrization

```
public interface Filtratu {  
    boolean test(Produktua pProd);  
}
```

Interfazea

```
public class Sekziokoak  
    implements Filtratu{  
    @Override  
    public boolean test(Produktua pProd) {  
        return pProd.getSekzio().equals("2");  
    }  
}
```

Interfazearen 3. implementazioa

```
public class Iraungitakoak  
    implements Filtratu{  
    @Override  
    public boolean test(Produktua pProd) {  
        return pProd.iraungitaDago();  
    }  
}
```

Interfazearen 1. implementazioa

```
public class Kostukoak  
    implements Filtratu{  
    @Override  
    public boolean test(Produktua pProd) {  
        return pProd.getPrezio()>12;  
    }  
}
```

Interfazearen 2. implementazioa

Aldatzen dena parametro legez. Interfazeak!!!

Behaviour parametrization

```
public List<Produktua> filtratuProd(Filtratu pFiltro,
{
    List<Produktua> filtratuak= new ArrayList<>();
    for (Produktua produktua : produktuZerr) {
        if (pFiltro.test(produktua))
            filtratuak.add(produktua);
    }
    return filtratuak;
}
```

Supermerkatua Klasea

```
public interface Filtratu {
    boolean test(Produktua pProd);
}
```

Interfazea

```
public class Iraungitakoak implements Filtratu
{...}
public class Sekziokoak implements Filtratu
{...}
public class Kostukoak implements Filtratu
{...}
```

Interfazearen implementazioak

```
List<Produktua> aIraungi=superM.filtratuProd(new Iraungitakoak());
List<Produktua> aSekzio =superM.filtratuProd(new Sekziokoak());
List<Produktua> aKostu  =superM.filtratuProd(new Kostukoak());
```

MAIN

Aldatzen dena parametro legez. Interfazeak!!!

Behaviour parametrization

```
public List<Produktua> filtratuProd(Filtratu pFiltro,
{
    List<Produktua> filtratuak= new ArrayList<>();
    for (Produktua produktua : produktuZerr) {
        if (pFiltro.test(produktua))
            filtratuak.add(produktua);
    }
    return filtratuak;
}
```

Supermerkatua Klasea

```
public interface Filtratu {
    boolean test(Produktua pProd);
}
```

Interfazea

```
public class Iraungitakoak implements Filtratu
{...}
public class Sekziokoak implements Filtratu
{...}
public class Kostukoak implements Filtratu
{...}
```

Interfazearen implementazioak

```
List<Produktua> aIraungi=superM.filtratuProd(new Iraungitakoak());
List<Produktua> aSekzio=superM.filtratuProd(new Filtratu() {
```

Klase anonimoa →

MAIN

```
@Override
public boolean test(Produktua pProd) {
    return pProd.getSekzio().equals("2");
}}
```

Aldatzen dena parametro legez. Interfazeak!!!

Interfaze funtzionalak

- ▶ **Java8**-tik aurrera, **aurredefinitutako** interfazeak dira:
 - **Metodo abstraktu bakarra**
- ▶ Funtzioak/baldintzak irudikatzen dituzte: **portaerak**
- ▶ Definitzerakoan, `@FunctionalInterface` jarri

Predicate

```
@FunctionalInterface
public interface Predicate <T>{
    boolean test (T t) ;
}
```

Consumer

```
@FunctionalInterface
public interface Consumer <T>{
    void accept (T t) ;
}
```

Supplier

```
@FunctionalInterface
public interface Supplier <T>{
    T get () ;
}
```

Function

```
@FunctionalInterface
public interface Function <T,R>{
    R apply (T t) ;
}
```

Interfaze funtzionalak

```
public List<Produktua> filtratuProd (Predicate<Produktua> pPredicate) {  
    List<Produktua> filtratuak= new ArrayList<>();  
    for (Produktua produktua : produktuZerr) {  
        if (pPredicate.test(produktua))  
            filtratuak.add(produktua);  
    }  
    return filtratuak;  
}
```

Supermerkatua

Implementazioak

```
public class Iraungitakoak implements Predicate<Produktua>{...}  
  
public class Sekziokoak implements Predicate<Produktua>{...}  
  
public class Kostukoak implements Predicate<Produktua>{...}
```

```
public interface Predicate <T>{  
    boolean test (T t) ;  
}
```

```
List<Produktua> aIraungi = superM.filtratuProd(new Iraungitakoak());  
List<Produktua> aSekzio  = superM.filtratuProd(new Sekziokoak());  
List<Produktua> aKostu   = superM.filtratuProd(new Kostukoak());
```

MAIN

**Baina, oraindik inplementazioa egin behar!
Klase berri bat edo klase anonimoa...**

Lambda espresioak

- ▶ Nola erabili interfaze funtzionalak?
- ▶ Orain arte

```
public class Sekziokoak implements Predicate<Pertsona>{  
    boolean test(Produktua pProduktua) {  
        return pProduktua.getSekzio().equals("2");  
    }  
}
```

- ▶ Luzea eta neketsua

Lambda espresioak

```
public class Sekziokoak implements Predicate<Pertsona> {  
    boolean test(Produktua pProduktua) {  
        return pProduktua.getSekzio().equals("2");  
    }  
}
```

Askoz konpaktuagoa!!!

`p -> p.getSekzio().equals("2")`

Lambda espresioak

```
public class Sekziokoak implements Predicate<Pertsona>{  
    boolean test(Produktua pProduktua) {  
        return pProduktua.getSekzio().equals("2");  
    }  
}
```

Sarrera parametroa

p -> p.getSekzio().equals("2")

Lambda espresioak

```
public class Sekziokoak implements Predicate<Pertsona>{  
    boolean test(Produktua pProduktua) {  
        return pProduktua.getSekzio().equals("2");  
    }  
}
```

implementazioa

```
p -> p.getSekzio().equals("2")
```

Lambda espresioak

- ▶ Interfaze funtzionalak inplementatu, klaserik sortu barik

```
p -> p.getSekzio().equals("2")  
p -> p.iraungitaDago()  
p -> p.getPrezio() > 12
```

- ▶ Parametroak egitekoekin erlazionatzen dituzte

Lambda espresioak

► Sintaxia

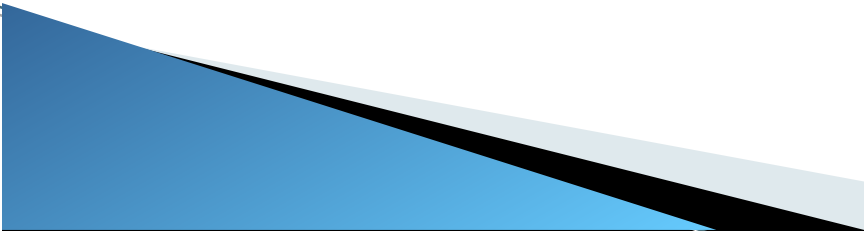
`(parametroak) -> gorputza`

- *Parametroak*: interaze funtzionalaren metodo abstraktuaren parametro zerrenda

```
p          -> p.getSekzio().equals("2")  
( p , pr) -> {p.getPrezio() > pr}
```

- *Gorputza*: instrukzio blokea edo espresioa - **giltza artean**

```
p          -> p.getSekzio().equals("2")  
( p , pr) -> {p.getPrezio() > pr}
```



Lambda espresioak

```
public List<Produktua> filtratuProd (Predicate<Produktua> pPredicate) {  
    List<Produktua> filtratuak= new ArrayList<>();  
    for (Produktua produktua : produktuZerr) {  
        if (pPredicate.test(produktua))  
            filtratuak.add(produktua);  
    }  
    return filtratuak;  
}
```

```
public interface Predicate <T>{  
    boolean test (T t) ;  
}
```

Supermerkatua

```
List<Produktua> aIraungi=superM.filtratuProd( (p) -> p.iraungitaDago() );  
List<Produktua> aSekzio =superM.filtratuProd( (p) -> p.getSekzioa().equals("2") );  
List<Produktua> aKostu  =superM.filtratuProd( (p) -> p.getPrezio() > 12 );
```

MAIN

Implementazioa (portaera) parametro legez pasatu, lambda espresio bidez

Metodo erreferentziak

- ▶ Klase batek interfaze funtzional baten sinadura daukan metodoa badu, metodoaren erreferentzia parametro bezala pasa daiteke.
- ▶ Sintaxia:

`Klasea::metodoa`

`Objektua::metodoa`

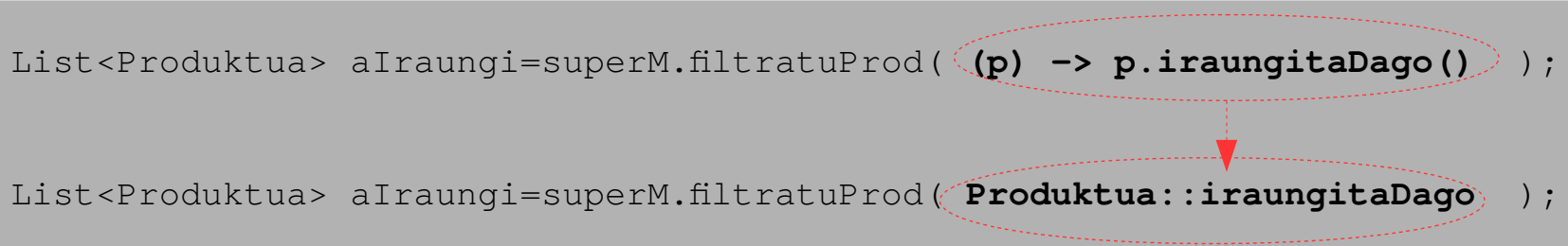
- ▶ Adibidea:

```
produktuak (comparing (Produktua::getPrezioa) ) ;
```

Metodo erreferentziak

```
List<Produktua> aIraungi=superM.filtratuProd( (p) -> p.iraungitaDago() );
```

```
List<Produktua> aIraungi=superM.filtratuProd( Produktua::iraungitaDago );
```

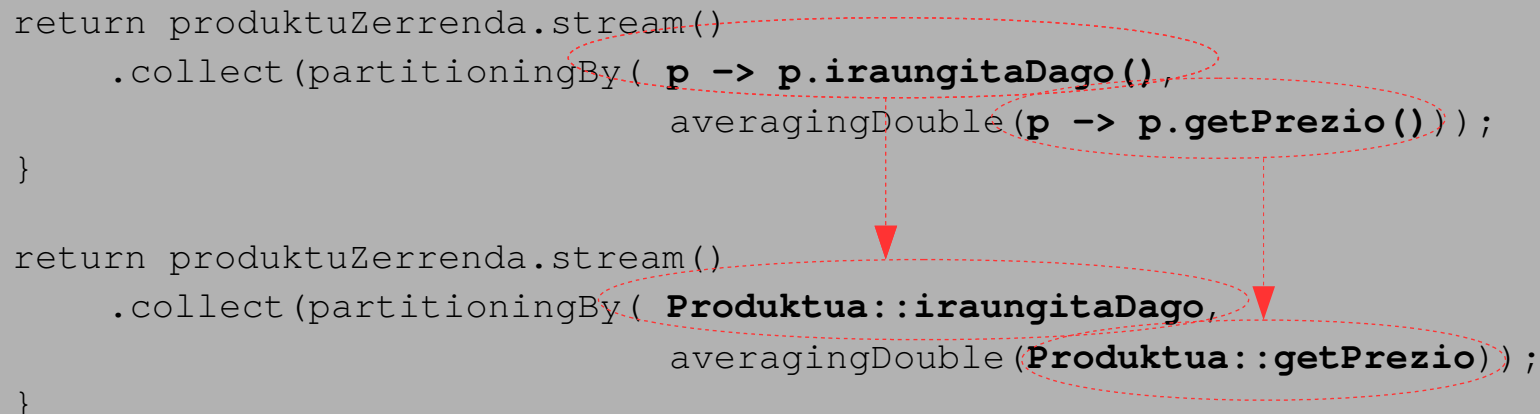


```
Consumer<String> cons = p -> System.out.println(p);
```

```
Consumer<String> cons = System.out::println;
```



```
return produktuZerrenda.stream()  
    .collect(partitioningBy( p -> p.iraungitaDago(),  
                             averagingDouble(p -> p.getPrezio()) ));  
}  
  
return produktuZerrenda.stream()  
    .collect(partitioningBy( Produktua::iraungitaDago,  
                             averagingDouble(Produktua::getPrezio) ));  
}
```



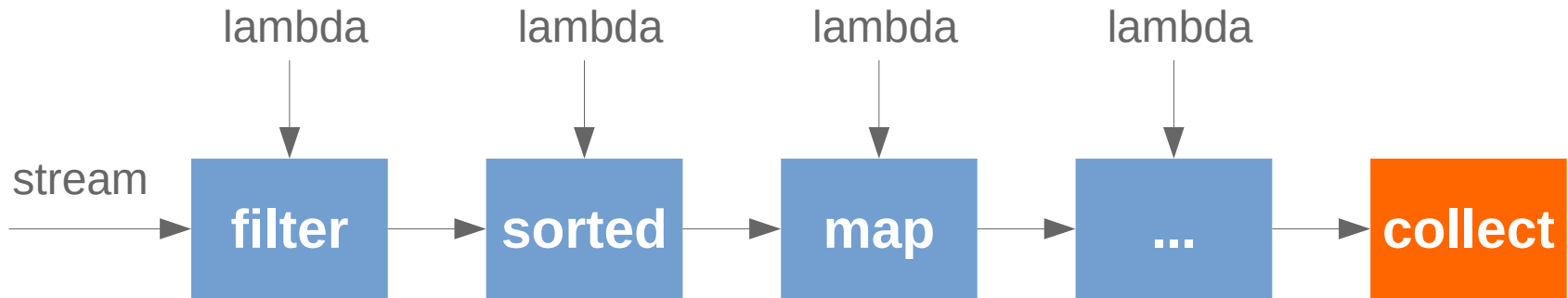
Stream eta agregazio operazioak

- ▶ Java-ren 8. bertsioiko nobedadeak:
 - Algoritmo arruntaren inplementazioa
 - Filtraketa
 - Map
 - ForEach
 - Batura
 -
 - Lambda espresioen erabilpena
 - Barne iterazioak *Stream*-en bidez

Stream eta agregazio operazioak

- ▶ Nola dabilta berrietasun horiek?
 - *Pipeline*: operazio kateaketa (datu fluxu sekuentzia)
 - Datu fluxuak
 - `stream()` : Sekuentziala
 - `parallelStream()` : Konkurrentea
 - *Barne iterazioak*
 - *Operazioak*:
 - Bitartekoak: `map`, `sorted`, `filter` ...
 - Amaierakoak: `collect`, `sum`, `forEach` ...

Stream eta agregazio operazioak



Stream eta agregazio operazioak

```
public List<Produktua> getProduktuIraungituak() {  
    List<Produktua> iraungituak = new ArrayList<>();  
    for (Produktua produktua : produktuak) {  
        if (produktua.iraungitaDago())  
            iraungituak.add(produktua);  
    }  
    return iraungituak;  
}
```

JAVA7



```
Public List<Produktua> getProduktuIraungituak() {  
    return produktuZerrenda.stream()           //barne iterazioa  
        .filter(p->p.iraungitaDago()) //filtroa  
        .collect(toList()); //zerrenda batean multzokatu  
}
```

JAVA8

Stream eta agregazio operazioak

- ▶ Barne iterazioa:

- Sekuentziala

```
produktuZerrenda.stream()           //sekuentzialki iteratu  
    .filter( p -> p.iraungitaDago()) //filtroa  
    .collect(toList()); //zerrenda batean multzokatu
```

- Paraleloa

```
produktuZerrenda.parallelStream() //paraleloan iteratu  
    .filter( p -> p.iraungitaDago()) //filtroa  
    .collect(toList()); //zerrenda batean multzokatu
```

Stream eta agregazio operazioak

► stream VS parallelStream



parallelStream-ek datuen fluxua prozesadoreak beste zatitan banatzen du. Elementuen prozesaketaren ordena aldatu egin daiteke.

Stream eta agregazio operazioak

► Bitarteko operazioak: fluxu berria sortu

OP	Argumentua	Buelta	Helburua
<code>filter</code>	<code>Predicate<T></code>	<code>Stream<T></code>	Predikatua betetzen duten elementuen fluxua bueltatu.
<code>map</code>	<code>Function<T, R></code>	<code>Stream<R></code>	Fluxuko elementu bakoitzari funtzio bat aplikatu, eta emaitza fluxu berri batean bueltatu. Tipo primitiboentzat aldaerak daude (<code>mapToInt</code> edo <code>mapToDouble</code>)
<code>sorted</code>	<code>Comparator<T></code>	<code>Stream<T></code>	Fluxu bateko elementuak baldintza batzuen arabera ordenatu eta emaitza fluxu berri batean bueltatu.
<code>distinct</code>		<code>Stream<T></code>	Fluxu berria bueltatu, errepikatu gabeko elementuez osatutakoa

Stream eta agregazio operazioak

► Bitarteko operazioak:

```
Public double getIraungituenPrezioTotala() {  
    return produktuZerrenda.stream()  
        .filter(p->p.iraungitaDago()) //filtroa  
        .mapToDouble(p->p.getPrezio()) //mapaketa  
        .sum();                       //batuketa  
}
```

```
Public List<Produktua> getZerrendaPreziozOrdenatuta() {  
    return produktuZerrenda.stream()  
        .mapToDouble(p->p.getPrezio()) //mapaketa  
        .sorted((i1,i2)-> i2.compareTo(i1)) //konparaketa  
        .collect(toList());             //bilketa  
}
```

Stream eta agregazio operazioak

► Amaierako operazioak: prozesua ejekutatu

OP	Argumentua	Buelta	Helburua
<code>forEach</code>	<code>Consumer<T></code>	<code>void</code>	Fluxuko elementu bakoitza kontsumitu, definitutako lambda aplikatuz.
<code>count</code>		<code>long</code>	Fluxuko elementu kopurua bueltatu.
<code>collect</code>	<code>Collector<T, A, R></code>	<code>R</code>	Fluxua erreduzitu zerrenda mapa edo balio oso bat sortzeko, definitutako rekolekzio metodoaren arabera.
<code>anyMatch</code>	<code>Predicate<T></code>	<code>boolean</code>	Fluxuko elementuetako batek predikatua betetzen badu, <code>true</code> bueltatu.
<code>allMatch</code>	<code>Predicate<T></code>	<code>boolean</code>	Fluxuko elementu orok predikatua betetzen badute, <code>true</code> bueltatu.

Stream eta agregazio operazioak

- ▶ **Amaierako operazioak:** zenbakidun fluxuak
(`IntStream` edo `DoubleStream`)

OP	Arg.	Buelta	Helburua
<code>sum</code>		<code>int</code> edo <code>double</code>	Fluxuko elementuen batuketa bueltatu.
<code>average</code>		<code>OptionalDouble</code>	Fluxuko elementuen batazbestekoa bueltatu.
<code>summaryStatistics</code>		<code>IntSummaryStatistics</code> , <code>DoubleSummaryStatistics</code>	Fluxuko elementuen estatistikak bueltatzen ditu

Stream eta agregazio operazioak

- Bilketa metodoak: modu estatikoan inportatzea komeni. `java.util.stream.Collectors` klasea.

OP	Argumentua	Buelta	Helburua
<code>toList</code>		<code>int</code>	Fluxu bateko elementuak biltzen dituen kolektorea bueltatu.
<code>partitioningBy</code>	<code>Predicate<T></code>	<code>Map<boolean, D></code>	Predikatu baten arabera, elementuak (erredukzioa aplikatuz) biltzen dituen kolektorea bueltatu.
<code>groupingBy</code>	<code>Function<T></code>	<code>Map<K, D></code>	Sailkapen baten arabera, elementuak (erredukzioa aplikatuz) biltzen dituen kolektorea bueltatu.

Stream eta agregazio operazioak

- ▶ Bilketa metodoak:

```
Public Map<Boolean,List<Produktu>> getIraungiEziraungiZerr(){  
    return produktuZerrenda.stream()  
        .collect(partitioningBy(p->p.iraungitaDago()));  
}
```

Stream eta agregazio operazioak

- Bilduma metodoak:

```
Public Map<String,List<Produktu>> getSekzioZerr(){  
    return produktuZerrenda.stream()  
        .collect(groupingBy(p->p.getSekzio()));  
}
```

Stream eta agregazio operazioak

► **Optional**-ak:

- Motibazioa:
 - Zein da sekuentzi huts baten batazbestekoa?
 - Ezein elementuk bilaketa irizpiderik bete ezean, zer bueltatu?
- **Optional<T>** : Balio bat enkapsulatzeko datu mota, existitzen baldin bada.
 - Metodoak ditu:
 - hutsik dagoen jakiteko: `isPresent`
 - balioa eskatzeko: `get`
 - defektuzko balioa hutsik badago: `orElse`
 - Tipo primitiboentzako implementazioak (`OptionalDouble...`)

Interfazeak

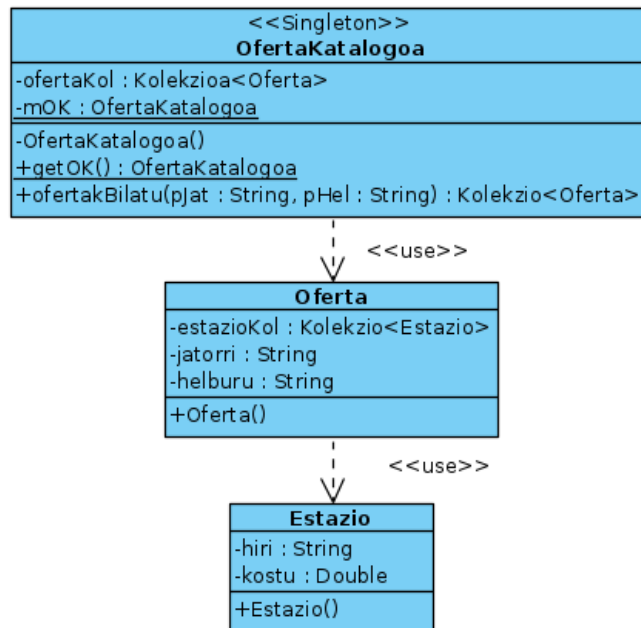
- ▶ Java8-n *defektuzko inplementazio* bat gehitu daiteke
 - `implements` egiten duten klaseek ez dute defektuzko inplementazioekin ezer egin behar

```
public interface DoIt{  
    void doSomething(int i, double x);  
    default void defektuzkoMetodoa() {  
        System.out.println("Defektuzko metodoa naiz!");  
    }  
}
```

- ▶ Interfazeetan *metodo estatikoak* definitu daitezke
 - Ezin dira deitu `implements` egiten duten klaseetatik, interfazearen izenetik baizik

GENERIZITATEA

BIDAIONDO bidaien agentziak bidaien erreserbak kudeatzeko aplikazio bat dauka. Hurrengo irudiak aplikazioaren klase diagrama adierazten du.



1. **OfertaKatalogo-ko** `printHelburuPosibleak(String pJat)` kodetu, jatorri baten helburu posible oro inprimatzeko.
2. `printHelburuPosibleak2(String pJat)` kodetu, aurrekoaren moduan egin, baina helburuak errepikatu barik.
3. `List<Oferta> getJatorrizOrdenatutakoOfertak()` kodetu, jatorriz alfabetikoki ordenatutako ofertak bueltatzeko.
4. `List<Oferta> getJatorrizHelburuzOrdenatutakoOfertak()` kodetu, lehenik jatorriz eta ondoren helburuz ordenatutako ofertak bueltatzeko.
5. `List<Oferta> getEstaziodunOfertak(String pHiri)` kodetu, estazioa hiri jakin batetan duen oferten kolekzioa bueltatzeko.
6. `printOfertakJatorriHelburu(String pJat, String pHel)` kodetu, jatorri eta helburu jakin bateko ofertak prezioz ordenatuta pantailaratzeko.
7. `Map<String, Oferta> getOfertaMinEstazioJatorri()` kodetu, helburu posible bakoitzerako estazio gutxien dituen oferten mapa bueltatzeko.
8. `Map<String, Integer> getMinEstazioJatorri()` kodetu, aurrekoaren moduan, baina estazio kopurua bueltatuz mapan.

Interface Stream<T>

Modifier and Type	Method and Description
boolean	allMatch(Predicate<? super T> predicate) Returns whether all elements of this stream match the provided predicate.
boolean	anyMatch(Predicate<? super T> predicate) Returns whether any elements of this stream match the provided predicate.
OptionalDouble	average() Returns an OptionalDouble describing the arithmetic mean of elements of this stream, or an empty optional if this stream is empty.
<R,A> R	collect(Collector<? super T,A,R> collector) Performs a mutable reduction operation on the elements of this stream using a Collector.
<R> R	collect(Supplier<R> supplier, BiConsumer<R,? super T> accumulator, BiConsumer<R,R> combiner) Performs a mutable reduction operation on the elements of this stream.
long	count() Returns the count of elements in this stream.
Stream<T>	distinct() Returns a stream consisting of the distinct elements (according to Object.equals(Object)) of this stream.
Stream<T>	filter(Predicate<? super T> predicate) Returns a stream consisting of the elements of this stream that match the given predicate.
void	forEach(Consumer<? super T> action) Performs an action for each element of this stream.
<R> Stream<R>	map(Function<? super T,? extends R> mapper) Returns a stream consisting of the results of applying the given function to the elements of this stream.
DoubleStream	mapToDouble(ToDoubleFunction<? super T> mapper) Returns a DoubleStream consisting of the results of applying the given function to the elements of this stream.
IntStream	mapToInt(ToIntFunction<? super T> mapper) Returns an IntStream consisting of the results of applying the given function to the elements of this stream.
LongStream	mapToLong(ToLongFunction<? super T> mapper) Returns a LongStream consisting of the results of applying the given function to the elements of this stream.
Stream<T>	sorted(Comparator<? super T> comparator) Returns a stream consisting of the elements of this stream, sorted according to the provided Comparator.
int	sum() Returns the sum of elements in this stream.
IntSummaryStatistics	summaryStatistics() Returns an IntSummaryStatistics describing various summary data about the elements of this stream.

Class Collectors

Modifier and Type	Method and Description
<code>static <T,A,R,RR> Collector<T,A,RR></code>	<code>collectingAndThen(Collector<T,A,R> downstream, Function<R,RR> finisher)</code> Adapts a Collector to perform an additional finishing transformation.
<code>static <T,K> Collector<T,?,Map<K,List<T>>></code>	<code>groupingBy(Function<? super T,? extends K> classifier)</code> Returns a Collector implementing a "group by" operation on input elements of type T, grouping elements according to a classification function, and returning the results in a Map.
<code>static <T,K,A,D> Collector<T,?,Map<K,D>></code>	<code>groupingBy(Function<? super T,? extends K> classifier, Collector<? super T,A,D> downstream)</code> Returns a Collector implementing a cascaded "group by" operation on input elements of type T, grouping elements according to a classification function, and then performing a reduction operation on the values associated with a given key using the specified downstream Collector.
<code>static <T> Collector<T,?,Optional<T>></code>	<code>maxBy(Comparator<? super T> comparator)</code> Returns a Collector that produces the maximal element according to a given Comparator, described as an Optional<T>.
<code>static <T> Collector<T,?,Optional<T>></code>	<code>minBy(Comparator<? super T> comparator)</code> Returns a Collector that produces the minimal element according to a given Comparator, described as an Optional<T>.
<code>static<T> Collector<T,?,Map<Boolean,List<T>>></code>	<code>partitioningBy(Predicate<? super T> predicate)</code> Returns a Collector which partitions the input elements according to a Predicate, and organizes them into a Map<Boolean, List<T>>.
<code>static <T,D,A> Collector<T,?,Map<Boolean,D>></code>	<code>partitioningBy(Predicate<? super T> predicate, Collector<? super T,A,D> downstream)</code> Returns a Collector which partitions the input elements according to a Predicate, reduces the values in each partition according to another Collector, and organizes them into a Map<Boolean, D> whose values are the result of the downstream reduction.
<code>static <T> Collector<T,?,List<T>></code>	<code>toList()</code> Returns a Collector that accumulates the input elements into a new List.

Interface Comparator<T>

Modifier and Type	Method and Description
<code>int</code>	<code>compare(T o1, T o2)</code> Compares its two arguments for order.
<code>static <T,U extends Comparable<? super U>> Comparator<T></code>	<code>comparing(Function<? super T,? extends U> keyExtractor)</code> Accepts a function that extracts a Comparable sort key from a type T, and returns a Comparator<T> that compares by that sort key.
<code>static <T,U> Comparator<T></code>	<code>comparing(Function<? super T,? extends U> keyExtractor, Comparator<? super U> keyComparator)</code> Accepts a function that extracts a sort key from a type T, and returns a Comparator<T> that compares by that sort key using the specified Comparator.
<code>default Comparator<T></code>	<code>thenComparing(Comparator<? super T> other)</code> Returns a lexicographic-order comparator with another comparator.