

JAVA İLE FONKSİYONEL PROGRAMLAMA : LAMBDA 'DAN STREAM GATHERERS 'LARA

ÜMİT KÖSE

- Türksat Uydu Haberleşme Kablo TV ve İşletme A.Ş - 2017 -
- Ankara Üniversitesi - Yüksek Lisans - 2018 - 2022
- Fırat Üniversitesi - Lisans - 2012 - 2016

	https://umiitkose.com		umiitkose
	umiitkose@gmail.com		umiitkose
	umiitkose.com		@umiitkose

JAVA İLE FONKSİYONEL PROGRAMLAMA : LAMBDA 'DAN STREAM GATHERERS 'LARA

Neler Konuşacağımız?

01. Giriş

05. Method Reference

02. Fonksiyonel Programlama?

06. Stream

03. Lambda

07. Stream Gatherers

04. Functional Interface (Big Four)

08. Çıkarım

Giriş

Fonksiyon : Javadaki metodları unutun 😊



- Pure Functions and referential transparency
- Immutability
- Recursion
- First-Class and higher order functions
- Functional composition
- Currying
- Partial Function Application
- Lazy Evaluation

Giriş

PURE FUNCTION

Girdilere aynı çıktıları veren metodlardır. Fonksiyonlar kendilerine parametre geçenler dışında hiçbir şey değiştiremez.

```
public int toplama(int a, int b) {  
    return a + b;  
}
```

IMMUTABILITY

Değeri değişmeyen obje yada değişkenler.

```
public record RecordUser(String name, int age) {}
```

HIGHER ORDER FUNCTION

- Bir yada daha fazla lambda ifadesini parametre olarak almak
- Bir lambda ifadesini geriye döndürmek

```
List<Integer> example = Stream.of(1, 2, 3, 4, 5)  
    .filter(i1 -> i1 > 2)  
    .map(i1 -> i1 * 2)  
    .distinct()  
    .sorted()  
    .limit(3)  
    .toList();
```

LAZY EVALUATION

Kullanılana kadar değişkenin ilk değer atamasının yapılmaması, çağrıldığı yerde başlatılması.

FIRST CLASS CITIZEN

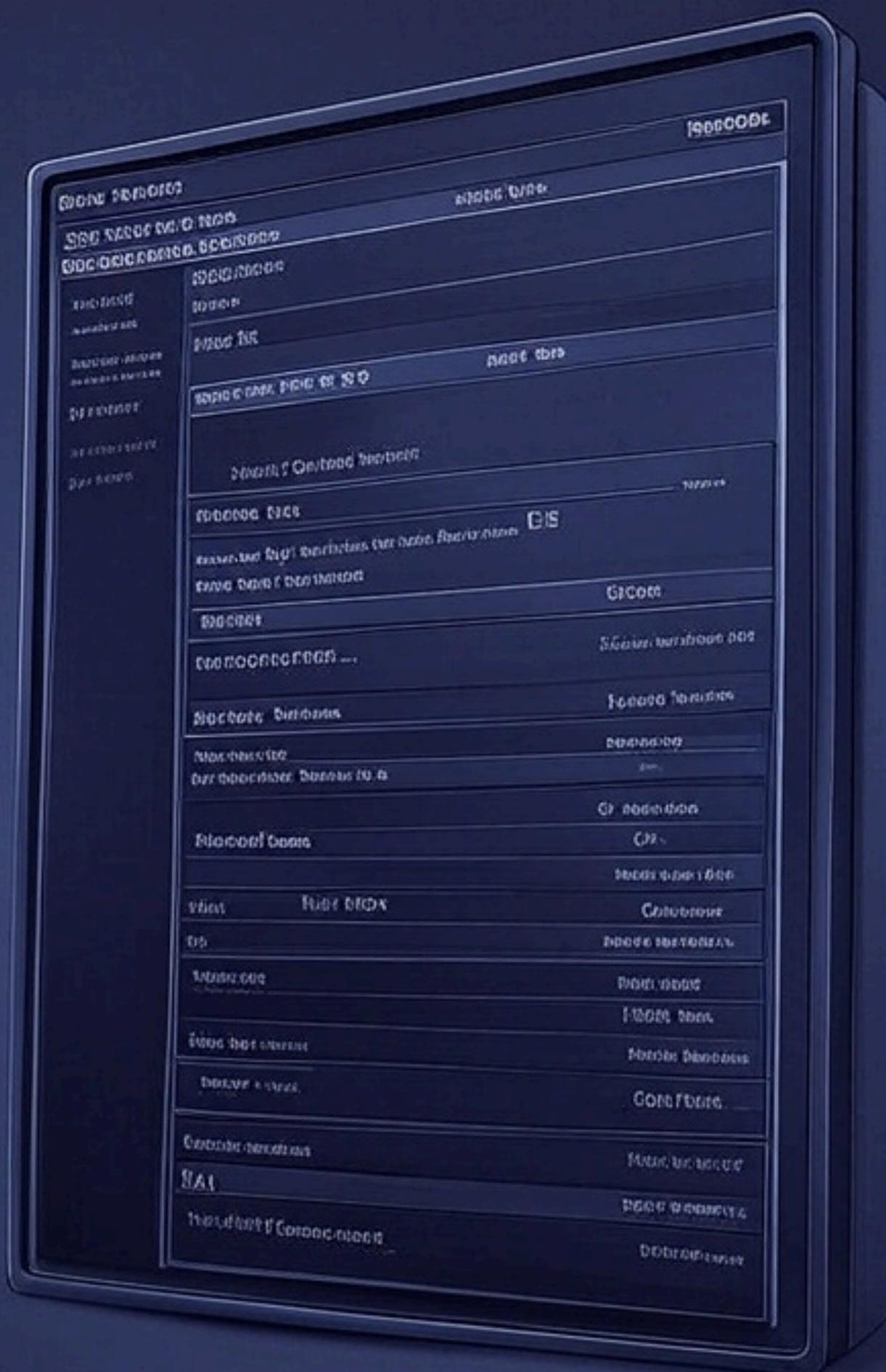
Bir değişkene atanma, metoda argüman geçirilme yada geriye dönüş olarak getirilme gibi özelliklere sahip yapılara first class citizen denir.

Fonksiyonel Programlama

Fonksiyonel programlama, programları "ne yapılacağını" tanımlayan, yan etkileri olmayan saf fonksiyonlar kullanarak oluşturma yöntemidir.

WIE NEDEN?

- Declarative (Ne) style & imperative (Nasıl) style



Java -> Fonksiyonel Programlama

- **Functional Interface, Stream, Lambda - Java 8**

Java 8'e fonksiyonel programlamayı kazandıran birçok özellik gelmiştir. Sınıfların çoğunda bu özelliklerle ilgili geliştirmeler yapılmıştır.

Mart 2014

Mart 2014

- **Method References - Java 8**

Java 8 sürümüyle method reference kavramı kodun daha okunabilirliğini sağlamak için gelmiştir.

- **dropWhile & takeWhile - Java 9**
- **listOf - Java 16**

Belirli sürümlerde Stream API, Koleksiyonlar başta olmak üzere bir çok sınıfı fonksiyonel programlamaya faydalayan metodlar eklenmiştir.

Mart 2017

Mart 2021

- **Stream Gatherers - Java 24**

Streamlerde intermediate işlemleri özelleştirebilmemiz için Java 22 sürümüyle başlayıp, 24 sürümüyle release olan Stream Gatherers tanıtılmıştır.

Mart 2025

- **record - Java 16**

Javaya immutable nesneler oluşturabilmemiz için record sınıfları eklenmiştir.

The Java Version Almanac

Systematic collection of information about the history and the future of Java.

 javaalmanac.io



Java

$\lambda \rightarrow$
expression

Alonzo Church - 1941

Lambda

İsim vermeden, kısa süreli kullanım amacıyla tasarlanmış fonksiyonlardır.

- Lambda ifadeleri Anonymous Classların kısaltması olarak düşünebilirsiniz.
- Lambda, bir fonksiyonun kısa ve isimsiz bir şekilde tanımlanmasını sağlar.
- Tek bir soyut metoda sahip olmalıdır.

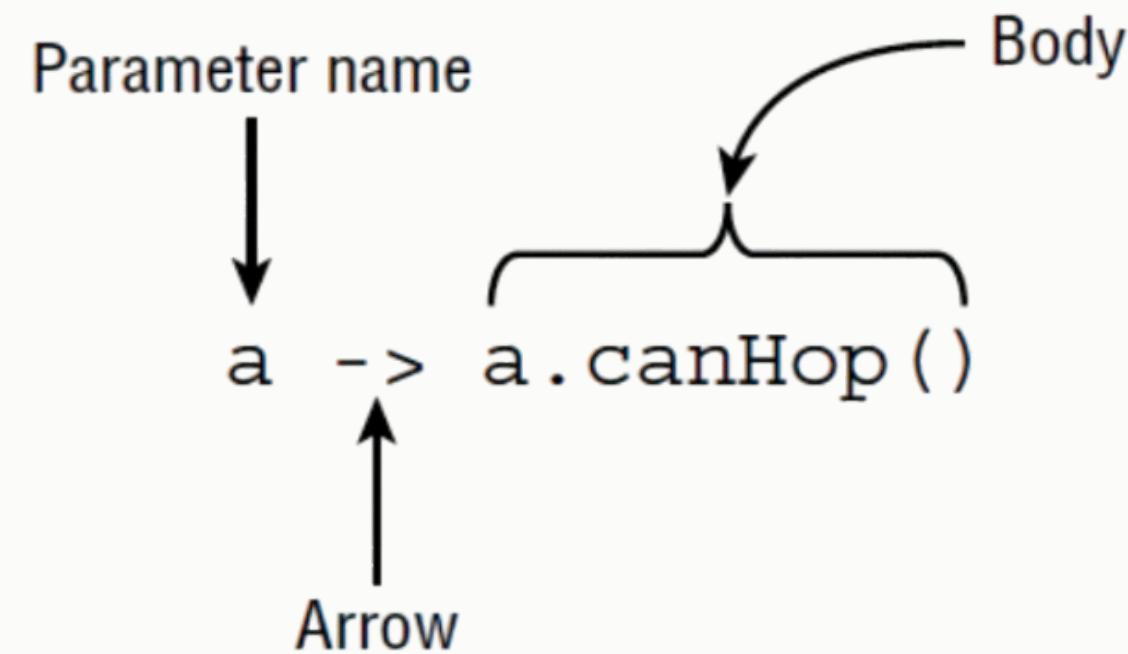
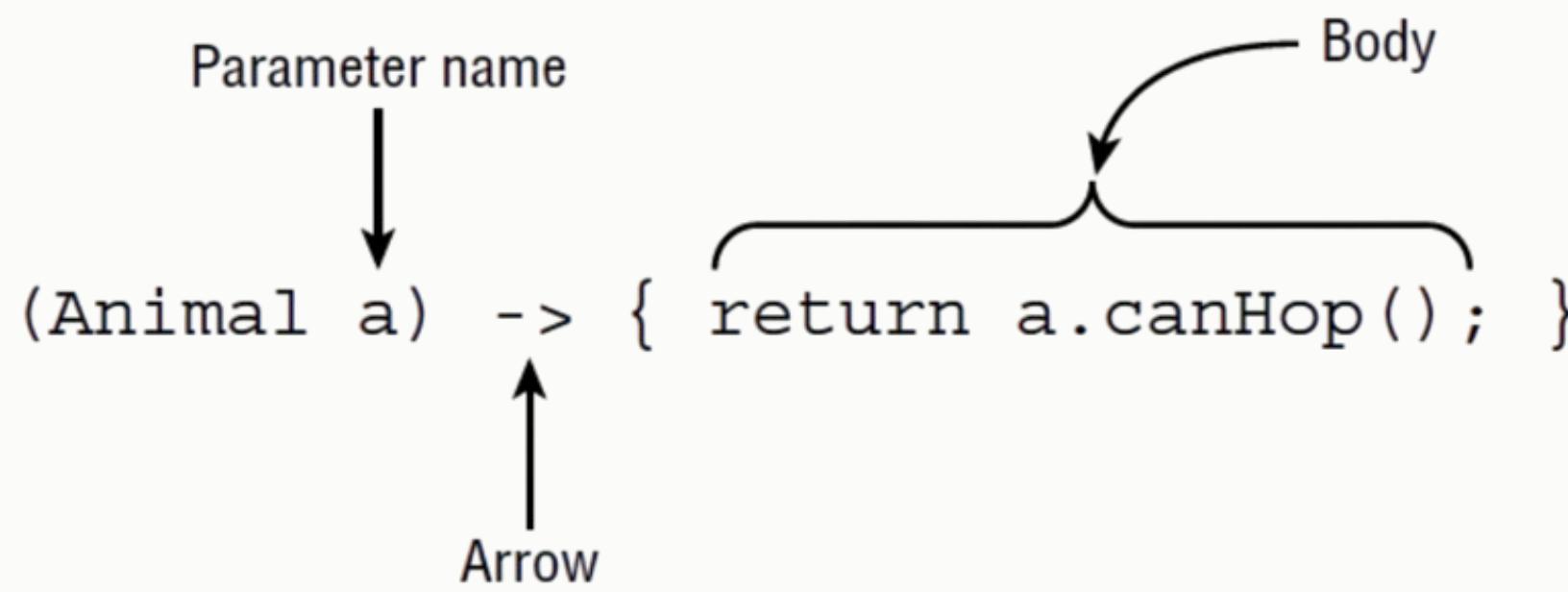


```
Runnable runnable = new Runnable() {  
    @Override  
    public void run() {  
        System.out.println("Merhaba Dünya!");  
    }  
};
```



```
Runnable runnable = () -> System.out.println("Merhaba Dünya!");
```

Lambda Syntax



```
() -> true;
```

```
x -> x.startsWith("test");
```

```
(String x) -> x.startsWith("test");
```

```
(x, y) -> { return x.startsWith("test"); }
```

```
(String x, String y) -> x.startsWith("test");
```

```
x, y -> x.startsWith("fish")
```

```
x -> { x.startsWith("camel"); }
```

```
x -> { return x.startsWith("giraffe") }
```

```
String x -> x.endsWith("eagle")
```

Functional Interface

Javada arayüzün tek bir metodu olması gerektiğini belirtmek için kullanılan anotasyondur.

- Opsiyoneldir
- Eğer arayüze birden fazla metot eklenirse ilgili fonksiyonda compiler hatası verilecektir.

```
@FunctionalInterface
public interface FunctionalInterfaceExample {
    // Abstract method
    void execute();

    // Default method
    default void defaultMethod() {
        System.out.println("Default method in FunctionalInterfaceExample");
    }

    // Static method
    static void staticMethod() {
        System.out.println("Static method in FunctionalInterfaceExample");
    }
}
```

Functional Interface (Big Four)

CONSUMER

Parametreleri alıp, işleyip geriye değer döndürmez.

1

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

FUNCTION

Parametre alır ve geriye sonuç döndürür.

3

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

PREDICATE

Argüman alır ve geriye boolean bir değer döndürür.

2

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

SUPPLIER

Argüman almayıp geriye değer döndürür.

4

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

Functional Interface (Diğerleri)

java.util.function paketi altında 43 tane functional interface bulunmaktadır.

Functional interfaces	Return type	Single abstract method	# of parameters
DoubleSupplier IntSupplier LongSupplier	double int long	getAsDouble getAsInt getAsLong	0
DoubleConsumer IntConsumer LongConsumer	void	accept	1 (double) 1 (int) 1 (long)
DoublePredicate IntPredicate LongPredicate	boolean	test	1 (double) 1 (int) 1 (long)
DoubleFunction IntFunction LongFunction	R	apply	1 (double) 1 (int) 1 (long)
DoubleUnaryOperator IntUnaryOperator LongUnaryOperator	double int long	applyAsDouble applyAsInt applyAsLong	1 (double) 1 (int) 1 (long)
DoubleBinaryOperator IntBinaryOperator LongBinaryOperator	double int long	applyAsDouble applyAsInt applyAsLong	2 (double, double) 2 (int, int) 2 (long, long)

Method Reference

Method references daha kolay bir şekilde kodu okumanın diğer bir yoludur. Lambdalar gibi yeni syntaxı kullanmak zaman alabilir.

- static methods
- Instance methods on a parameter to be determined at runtime

```
Converter converter = Math::round;
Converter converter1 = (x) -> Math.round(x);
```

```
StringParameterChecker methodRef = String::isEmpty;
StringParameterChecker methodRef2 = (String s) -> s.isEmpty();
```

- Instance methods on a particular object
- Constructors

```
var str = "";
StringChecker lambda = (s) -> str.startsWith(s);
StringChecker methodReference = str::startsWith;
```

```
Supplier<String> newString = String::new;
Supplier<String> newString2 = () -> new String();
```

Method Reference

Tip	Önceki Kısım	Sonraki Kısım	Örnek
static methods	Sınıf	Method	Math::random
instance methods on a particular object	Değişken	Method	str::startsWith
instance methods on a parameter	Sınıf	Method	String::isEmpty
Constructor	Sınıf	new	String::new

Streams

Koleksiyonlardaki map-reduce dönüşümleri gibi, eleman akışlarında fonksiyonel programlama paradigmasını destekleyen sınıflardır.

Source

Her bir stream, var olan veri kaynağından yeni bir Stream objesi oluşturarak Stream pipeline oluşturulur.

Intermediate

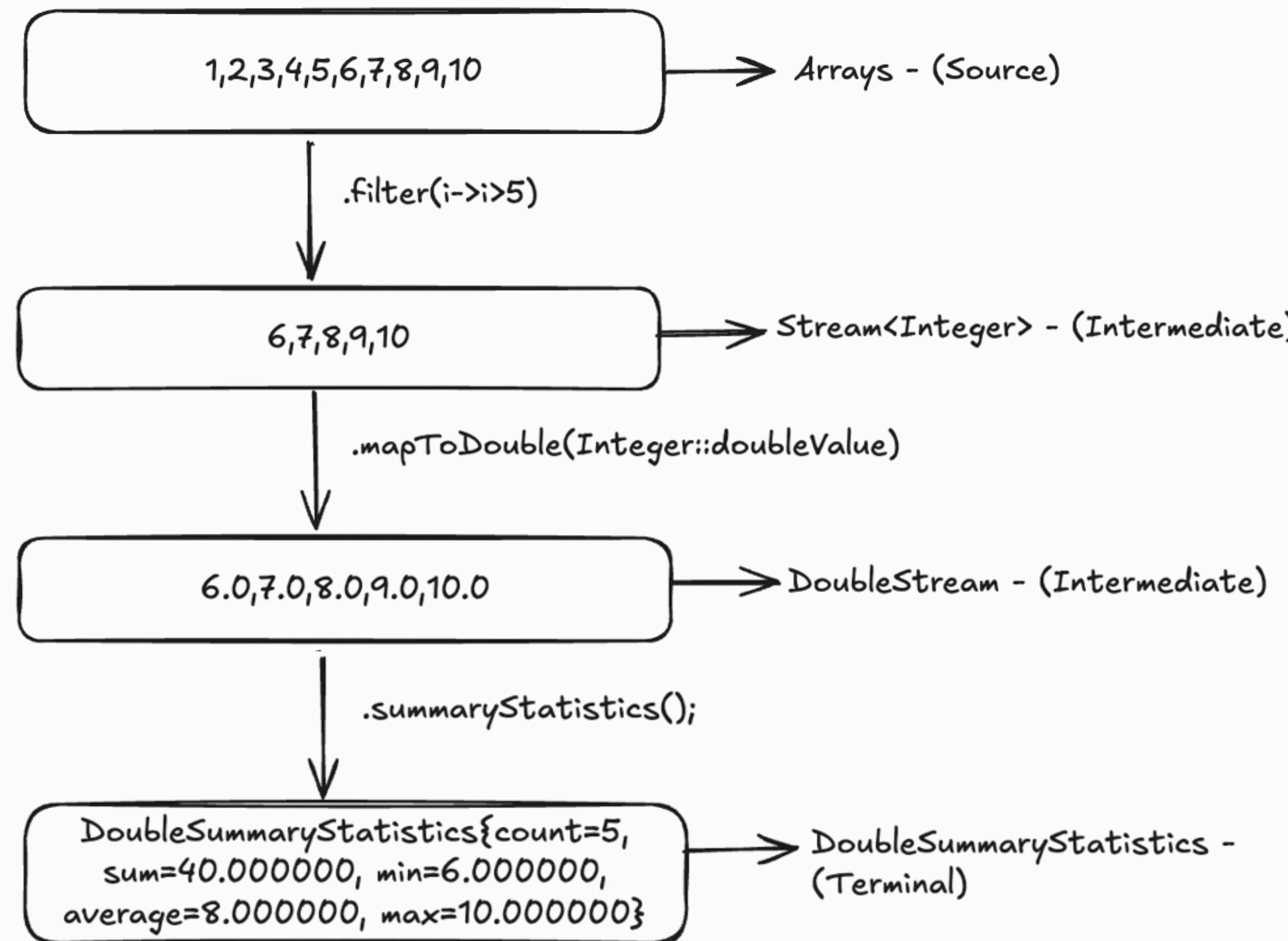
Ara işlemler yeni bir stream döndürür. Bunlar her zaman lazydir. filter() gibi bir ara işlemi yürütmek aslında herhangi bir filtreleme gerçekleştirmez. bunun yerine, geçildiğinde verilen öngörüyle eşleşen başlangıç akışının öğelerini içeren yeni bir akış oluşturur.

Terminal

Stream.forEach veya IntStream.sum gibi terminal işlemleri, bir sonuç veya yan etki üretmek için akışı tarayabilir. Terminal işlemi gerçekleştirildikten sonra, stream pipeline tüketilmiş kabul edilir ve artık kullanılamaz;.

```
DoubleSummaryStatistics doubleSummaryStatistics = Stream.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10) } → Source  
    .filter(i -> i > 5) } → Intermediate  
    .mapToDouble(Integer::doubleValue)  
    .summaryStatistics(); } → Terminal
```

Streams Çalışması



Source

Java bir çok veri kaynağını stream olarak kullanmamıza izin verir.

```
List<Integer> list = List.of(1, 2, 3, 4, 5);  
var streamList = list.stream();
```

```
String content = "Hello, World!";  
var chars = content.chars(); //IntStream  
var lines = content.lines(); //Stream<String>
```

```
Pattern pattern = Pattern.compile("\\s+");  
var stringStream = pattern.splitAsStream(content);
```

```
var stream = Stream.of("one", "two", "three");
```

Source

```
IntStream range = IntStream.range(1, 10);
```

Sonsuz streamlerde oluşturabiliriz.

```
var stream1 = Stream.iterate(0, n -> n + 1);
```

```
var stream2 = Stream.generate(() -> "Hello");
```

```
Stream.iterate(0, n -> n + 1)
    .limit(10) //limit olmasaydı sonsuz döngü olurdu.
    .forEach(System.out::println);
```

Veri Örneği

```
public record TopMovies(String title, String englishTitle, String genre, double rating, int year) {  
  
    @Override  
    public String toString() {  
        return String.format("%s (%s) - %s - %.1f - %d",  
            title, englishTitle, genre, rating, year);  
    }  
  
    new TopMovies("Esaretin Bedeli", "The Shawshank Redemption", "Dram", 9.3, 1994),  
    new TopMovies("Baba", "The Godfather", "Suç, Dram", 9.2, 1972),  
    new TopMovies("Kara Şövalye", "The Dark Knight", "Aksiyon, Suç, Dram", 9.0, 2008),  
    new TopMovies("Baba 2", "The Godfather Part II", "Suç, Dram", 9.0, 1974),  
    new TopMovies("12 Öfkeli Adam", "12 Angry Men", "Dram", 9.0, 1957),  
    new TopMovies("Schindler'in Listesi", "Schindler's List", "Biyografi, Dram, Tarih",  
9.0, 1993))  
}
```

Intermediate İşlemler

Filtreleme İşlemleri:

```
List<TopMovies> filteredMovies = topMoviesList.stream()
    .filter(movie -> movie.rating() > 8.0)
    .toList();
```

```
List<TopMovies> distinctMovies = topMoviesList.stream()
    .distinct()
    .toList();
```

Sıralama İşlemleri:

```
List<TopMovies> sortedMovies = topMoviesList.stream()
    .sorted(Comparator.comparingDouble(TopMovies::rating))
    .toList();
```

```
List<TopMovies> sortedMoviesReverse = topMoviesList.stream()
    .sorted(Comparator.comparingDouble(TopMovies::rating).reversed())
    .toList();
```

Intermediate İşlemler

takeWhile & dropWhile İşlemleri:

```
List<TopMovies> takeWhileMovies = topMoviesList.stream()
    .takeWhile(movie -> movie.rating() > 8.0)
    .toList();
```

```
List<TopMovies> dropWhileMovies = topMoviesList.stream()
    .dropWhile(movie -> movie.rating() > 8.0)
    .toList();
```

limit & skip İşlemleri:

```
List<TopMovies> limitedMovies = topMoviesList.stream()
    .limit(10)
    .toList();
```

```
List<TopMovies> skippedMovies = topMoviesList.stream()
    .skip(3)
    .toList();
```

Intermediate İşlemler

mapping İşlemleri:

```
List<String> movieTitles = topMoviesList.stream( )
    .map(TopMovies::title)
    .toList();
```

```
List<Integer> moviesLength = topMoviesList.stream( )
    .map(TopMovies::title)
    .map(String::length)
    .toList();
```

flatMap İşlemleri:

```
List<String> movieGenres = Stream.of(Stream.of("Action", "Drama"), Stream.of("Comedy", "Horror"))
    .flatMap(genre -> genre)
    .toList();
```

Terminal İşlemler

count & min & max:

```
long count = topMoviesList.stream()
    .filter(movie -> movie.rating() > 8.0)
    .count();
```

```
TopMovies minMovie = topMoviesList.stream()
    .min(Comparator.comparingDouble(TopMovies::rating))
    .orElse(null);
```

```
TopMovies maxMovie = topMoviesList.stream()
    .max(Comparator.comparingDouble(TopMovies::rating))
    .orElse(null);
```

forEach:

```
topMoviesList.stream()
    .forEach(movie -> System.out.println("Movie: " + movie.title()));
```

Terminal İşlemler

reduce

```
topMoviesList.stream().reduce((movie1, movie2) -> {
    if (movie1.rating() > movie2.rating()) {
        return movie1;
    } else {
        return movie2;
    }
}).ifPresent(movie -> System.out.println("Highest rated movie: " + movie.title()));
```

anyMatch:

```
boolean b = topMoviesList.stream()
    .anyMatch(movie -> movie.rating() > 8.0);
```

collect

```
Map<Double, List<String>> collect = topMoviesList.stream()
    .collect(
        Collectors.groupingBy(
            TopMovies::rating,
            Collectors.mapping(TopMovies::title, Collectors.toList())
        )
    );
System.out.println(collect);
```

Stream Gatherers

- **Kısaca Nedir?**

Java 22 ile tanıştığımız, Java 24 ile hayatımıza giren Stream Gatherers, yazılan Stream işlemleri için esneklik, özelleştirme ve verimli kullanım kazandırmayı amaçlayan, intermediate operasyonlar için esnekler sunan geliştirme.

- **Amaç**

- Streamleri daha fazla özelleştirecek, bu API için kullanımını artttırmak
- Stream ile ilgili kullanımlarda kodun yeniden kullanılabilirliğini artttırmak.
- Mümkin olduğunda sonsuz büyüklükteki akışları işlemek için özel ara işlemlere izin vermek.
- Karmaşık Stream operasyonlarının anlaşılmasını ve uygulanmasını basitleştirmek.

Stream Gatherers - ?

$\langle A \rangle$ – the potentially mutable state type of the gatherer operation (often hidden as an implementation detail)

```
public interface Gatherer<T, A, R> { ... }
```

$\langle T \rangle$ – the type of input elements to the gatherer operation

$\langle R \rangle$ – the type of output elements from the gatherer operation

Stream Gatherers - ?

interface Gatherer<T,A,R>

Supplier<A> initializer()

Opsiyoneldir, State sağlar. Eğer state ihtiyaç varsa oluşturma işlemi burada gerçekleştirilir.

Integrator<A,T,R> integrator()

Her giriş elemanı integrator ile uygulanır. Downstream true döndüğü sürece işlemeye devam edilir.

BinaryOperator<A> combiner()

Her giriş elemanı integrator ile uygulanır. Downstream true döndüğü sürece işlemeye devam edilir.

BiConsumer<A, Downstream<R>> finisher()

Daha fazla giriş elemanı olmadığında, bu fonksiyon state ile çağırılır ve Downstream final eylemi uygular.

Gatherer<T,A,R> andThen()

iki gatheri birleştiren metottur.

Downstream

interface Downstream<T>

boolean push(T element)

Gatherer işlemine devam etmesi için kullanılır.

default boolean isRejecting(){ return false;}

false dönüşü ile hesaplanacak olan verilerin stream pipeline aşağıya gönderilmemesi sağlanır.

Map Örneği

Stream'de map olmadığını düşünün.

```
var result = text.lines()
    // .map(String::length)
    .gather(map(String::length))
    .toList();
```

```
<T, R> Gatherer<T, ?, R> map(Function<T, R> mapper){
    Gatherer.Integrator<Void, T, R> integrator =
        (_ , element, downstream) ->
            downstream.push(mapper.apply(element));
    return Gatherer.of(integrator);
}
```

Filter Örneği

Stream'de filter olmadığını düşünün.

```
Stream.of(1, 2, 3, 4, 5, 6)
        .gather(filter(i -> i % 2 == 0))
        .forEach(System.out::println);
```

```
<T> Gatherer<T, ?, T> filter(Predicate<T> filter) {
    Gatherer.Integrator<Void, T, T> integrator = (_, value, downstream) -> {
        if (filter.test(value)) {
            return downstream.push(value);
        }
        return true;
    };
    return Gatherer.of(integrator);
}
```

Gatherers

Gatherer interface uygulayan Gatherers sınıfı içerisinde kullanışlı bir kaç method bulundurur. Bunları inceleyecek olursak;

Gatherers.fold()

Soldan sağa bir reduce işlemi uygular.

```
Stream.of(1, 2, 3, 4, 5, 6, 7)
    .gather(Gatherers.fold(() -> 0, (a, b) -> a + b))
    .findFirst().ifPresent(System.out::println);
//Çıktı -> 28
```

Gatherers.scan()

Soldan sağa doğru her bir elemanı tarar.

```
List<String> list = Stream.of(1, 2, 3)
    .gather(Gatherers.scan(() -> "", (a, b) -> a + b))
    .toList();
//[1, 12, 123]
```

Gatherers

Gatherer interface uygulayan Gatherers sınıfı içerisinde kullanışlı bir kaç özellik bulundurur. Bunları inceleyecek olursak;

Gatherers.windowFixed()

```
Stream.of(1,2,3,4,5,6,7)
    .gather(Gatherers.windowFixed(3))
    .forEach(System.out::print);
//[1, 2, 3][4, 5, 6][7]
```

Gatherers.windowSliding()

```
Stream.of(1, 2, 3, 4, 5, 6)
    .gather(Gatherers.windowSliding(3))
    .forEach(i -> System.out.print(i + " "));
//[1, 2, 3] [2, 3, 4] [3, 4, 5] [4, 5, 6]
```

FUNCTIONAL APPROACH TO JAVA

- Java her ne kadar fonksiyonel programlamayı desteklese de, kurallar olarak nesne yönelimli programlamayı tercih etmektedir.
- Java da FP matematiksel fonksiyonlarla declarative yaklaşımla problemleri çözer, OOP ise bildiğimiz encapsulation, polymorphism ve abstraction yöntemleriyle geliştirmeleri yapar. Nesneler ile ilgilenir
- Javada immutability oldukça önemlidir. Zaten çokça kullandığımız sınıflarda bu davranışı görebiliyoruz. Bizde OOP veya FP tarafında olsun bu özelliği çok net şekilde projelerimizde kullanmalıyız.
- Hiçbir zaman sadece fonksiyonel programlama herseye yeter diyemeyiz. Nesne Merkezli Programlama ile bir çok problem çözülsse dahi birlikte iyi yanlarının kullanılmasıyla daha anlaşılır, test edilebilir ve tekrar kullanılabilir kodlar geliştirmeyi düşünebiliriz.

Don't be a functional programmer.

Don't be an object-oriented programmer.

Be a better programmer.

Brian Goetz

- Kendi functional interface ‘inizi yazmadan önce Java içerisinde ihtiyacınızı karşılayan fonksiyonel interface olup olmadığını kontrol edin. `@FunctionalInterface` annotation fonksiyonunuza korur.
- Immutable nesne oluşturmak için gelen record sınıfları kullanalım. Mutable nesnelerimizde immutable yapabileceğimiz değişkenleri yapalım.
- Metot referansları kullanarak kodlarımızı daha okunabilir ve anlaşılır yazabiliriz.
- Lambda ifadelerini olabildiğince sade yazarak anlaşılabilirliği artıralım.
- Streamleri tekrar kullanamazsınız, unutmayın.
- Streamler lazy evaluation’dır, terminal işlemi olmadan başlamazlar.
- intermediate işlemlerini özelleştirme ihtiyacınızda Stream Gatherers inceleyin, kullanın.

Any fool can write code that a computer can understand. Good programmers write code that humans can understand.

-Martin Fowler, Refactoring: Improving the Design Of Existing Code

FUNCTIONAL PROGRAMMING

LAMBDA
EXPRESSIONS

$(x) \rightarrow x * x$

METHOD
REFERENCES

`String::length`

FUNCTIONAL
INTERFACES

Predicate

Function

Consumer

STREAM API

map

filter

reduce



IMMUTABLE
PURE
CLARITIVE
BURE
APPE
DECLURATIVE
NO
SIDE
EFFECTS

IMMUTABLE
PURE
CLARITIVE
BURE
FUNCTIONS
APPE
NO SIDE EFFECTS