# 18. Streamek

## Ismétlés: Lambda kifejezések

A lambda kifejezések a Java 8-as változatától kezdve használhatók, és sokszor lényegesen lerövidítik a szükséges kódot.

Egy lambda kifejezés tulajdonképpen egy egyszerűsített metódus.

Vegyük például a következő metódust:

int plusz(int a, int b) {  
 return a + b;  
}

Ez lambda kifejezéssel így írható le:

(a, b) -> a + b

Figyeld meg, hogy:

* ennek a metódusnak nincs neve,
* a paramétereket zárójelek között kell megadni, utána nyíllal,
* a nyíl után a visszaadott értéket adjuk meg return nélkül,
* a visszaadott értéknek és a paramétereknek nincs megadva a típusa (de megadható).

A lambda kifejezéseket akkor fogjuk használni, amikor paraméterként kell megadni egy metódust, így nem lesz szükség a névre.

A típusokat a fordító fogja kitalálni a helyzetnek megfelelően (nem csak egészekkel működik!)

a és b helyett tetszőleges neveket használhatunk.

Ha a metódus több soros, a lambda kifejezésekben is használjuk a kapcsos zárójeleket és a return utasítást! Például:

(p, s) -> {  
 int kulonbseg = p.length() – s.length();  
 return kulonbseg;  
}

Mi lesz ennek a metódusnak a visszadott értéke?

Ha bonyolult a metódus, akkor inkább írjuk meg külön, és a lambda kifejezésből hívjuk meg!

Ha a lambda kifejezés csak egy metódushívásból áll, akkor még tömörebben írhatjuk (metódus referencia). Például:

(String s) -> s.toUpperCase()

helyett ezt írhatjuk:

String::toUpperCase

Két paraméter esetén:

(String p, String s) -> p.sompareTo(s)

helyett ezt:

String::compareTo

Statikus metódus esetén:

(double x, double y) -> Double.compare(x, y)

helyett ezt:

Double::compare

Új objektum létrehozásához használjuk a new kulcsszót! Például:

() -> new Szo()

helyett írd ezt:

Szo::new

Tömb létrehozása metódus referenciával:

String[]::new

ami ezt jelenti:

(n) -> new String[n]

## Streamek

A streamek megkönnyítik az adatok feldolgozását, küönösen sok adat esetén. A részleteket a stream könyvtárra bízhatjuk, amely kihagyja a felesleges műveleteket, és el tudja osztani a feldolgozást több processzormag között. Ráadásul gyakran érthetőbb is lesz a program.

Nézzünk egy példát! Tegyük fel, hogy egy listában meg akarjuk számolni a tíz karakternél hosszabb szavakat!

Ezt eddig így csináltuk:

ArrayList<String> szavak = … ;  
int db = 0;  
for (String szo : szavak)  
 if (szo.length() > 10)  
 db++;

Ha az adatoklat stream-ként (adatfolyamként) kezeljük, akkor ezt így írhatjuk:

Stream<String> szavak = … ;  
int db = szavak  
 .filter(s -> s.length() > 10)  
 .count();

A filter metódus szűri a szavakat, a count pedig megszámolja őket.

A stream műveleteket nem kötelező külön sorba írni, de így áttekinthetőbb a program.

A streamek hasonlítanak a kollekciókra, de vannak lényeges különbségek:

* A stream nem tárolja az adatokat, csak feldolgozza. Az adatok máshonnan (listából, fájlból, adatbázisból, …) jönnek.
* A stream nem változtatható meg. A feldolgozás eredménye általában egy új stream vagy egy érték.
* Lusta feldolgozás (lazy processing): csak akkor végzi el a műveleteket, ha feltétlenül kell. Például ha az első öt hosszú szót keressük, csak addig nézi a szavak hosszát, amíg ötöt talál, a többi szót nem viszgálja meg.

## Streamek előállítása

A legegyszerűbb módszer a Stream.of metódus használata. Például:

Stream<String> szavak = Stream.of("Tercsi", "Fercsi", "Kata", "Klára");

Stream<Integer> szamok = Stream.of(3, 1, 4, 1, 5, 9);

Bármilyen kollekcióból is előállítható stream a .stream() metódussal. Például:

ArrayList<String> szolista = new ArrayList<>();  
Stream<String> szavak = szolista.stream();

Ha nagyon sok adat van, a feldolgozás elosztható a processzorok között a .parallel() metódussal, de vigyázni kell, mert ilyenkor a stream elemeit nem sorrendben dolgozza fel!

Például:

szavak.parallel();

Nyisd meg a *StreamDemo1* projektet, és hasonlítsd össze a stream használatát a ciklussal!

## Az eredmények összegyűjtése

Ha a feldolgozás eredménye egy stream, akkor azt tömbbé vagy listává alakíthatjuk.

A tömbbé alakításhoz a .toArray() metódus használható. Például:

String[] eredmeny = szavak.toArray(String[]::new);

A listává alakításhoz használhatjuk a .toList() metódust. Ennek szüksége van egy collectorra, amely meghatározza, hogyan kerüljenek a stream értékei a listába. A Collectors osztály tartalmaz ilyeneket. Például:

ArrayList<String> eredmeny = szavak.collect(Collectors.toList());

Az eredmények összegyűjtése után a stream tovább már nem használható!

## Streamek átalakítása

A .map() metódus egy metódust kap, és ezt a stream minden elemére alkalmazza. Az eredmény egy új stream lesz. Például:

Stream<String> szavak = Stream.of("Egy", "Kettő", "Három");  
Stream<String> kisbetus = szavak.map(s -> s.toLowerCase());

A kisbetus streamben minden szó kisbetűs lesz.

A .filter() metódus is egy metódust kap, amely egy logikai értéket ad vissza. Azok az elemek kerülenk az új streambe, amelyekre a visszaadott érték true. Például:

Stream<String> szavak = Stream.of("Egy", "Kettő", "Három");  
Stream<String> otbetus = szavak.filter(s -> s.length() == 5);

Melyik szavak kerülnek az otbetus stream-be?

A .limit(n) metódus a stream első n elemét veszi. Ha n-nél kevesebb elem van, akkor az összes elemet.

A .skip(n) metódus ezzel szemben kihagyja az első n elemet.

A .distinct() metódus kihagyja az ismétlődő elemeket. Például:

Stream<Integer> szamok = Stream.of(1, 2, 3, 2, 1, 4);  
Stream<Integer> kulonbozo = szamok.distinct();

A kulonbozo streambe ez kerül: 1, 2, 3, 4

A .sorted() metódus egy rendezett streamet állít elő.

## Az Optional típus

A Java-ban gyakori, hogy ha egy metódusnak nincs eredménye, akkor null értéket ad vissza. Ez azonban veszélyes, mert ha a programunk megpróbálja meghívni az eredmény egy metódusát, akkor NullPointerException kivételt kapunk, és a program leáll.

Ennek elkerülésére a stream könyvtár más megközelítést alkalmaz. Amikor egy lekérdezésnek lehet, hogy nincs eredménye, akkor Optional<T> típusú értéket ad vissza, ahol T az eredmény típusa, ha van.

Nézzünk egy példát! Tegyük fel, hogy az első 10 karakternél hosszabb szót keressük egy streamben a .findFirst() metódussal! Előfordulhat, hogy nincs egyetlen ilyen szó sem a streamben.

Ezért a .findFirst() metódus Optional<String> típusú értéket ad vissza:

Optional<String> opt = szavak  
 .filter(s -> s.length() > 10)  
 .findFirst();

Az eredmény (opt) egy objektum, amely vagy egy értéket tárol, vagy egy jelzést, hogy nincs érték.

Az .orElse() metódus segítségével kiolvashatjuk az értéket, vagy ha nincs, akkor megadhatunk egy alapértelmezett értéket:

int hossz = opt.orElse("").length();

Egy másik lehetőség az .ifPresent() metódus használata, amely akkor ad igaz értéket, ha van eredmény. Ekkor az értéket a .get() metódusssal olvassuk ki:

if (opt.ifPresent())  
 System.out.println("Hossz: " + opt.get());

Ha nekünk kell olyan metódust írni, amely Optional<T> típust ad vissza, akkor a következő lehetőségeink vannak:

* Ha van eredmény, akkor: return Optional.of(eredmény);
* Ha nincs eredmény, akkor: return Optional.empty();

## Lezáró műveletek

A lezáró műveletek (terminal operations) olyan műveletek, amelyek elindítják a számításokat, és egy értéket eredményeznek. Ezután a stream már nem használható.

Például ilyen lezáró művelet a .findFirst() és a .findAny(). Ez utóbbi keres egy találatot, nem feltétlenül az elsőt.

Tegyük fel. hogy egy y-ra végződő hosszú szót keresünk, mindegy hogy melyiket:

String y = szavak  
 .parallel()  
 .filter(s -> s.length() > 10)  
 .filter(s -> s.endsWith("y"))  
 .findAny()  
 .orElse("");

Magyarázd el a fenti kód működését! Hogyan lehetne a két filtert összevonni?

Ilyen művelet a .max() és a .min() is, amelyeknek meg kell adni comparator-t is. Például a leghosszabb string keresése a szavakban:

Optional<String> lh = szavak.max((p,s) ->  
 p.length()‑s.length());

Miért Optional típusú az eredmény?

A .toArray() és a .collect() metódusok is lezárják a feldolgozást.

A .forEach() lezáró metódus nem állít elő értéket, hanem alkalmaz egy metódust minden elemre. Például:

szavak.forEach(s -> System.out.println(s));

Vigyázat! Párhuzamos feldolgozás esetén nem sorrendben lesznek az elemek!

Végül van három logikai értéket visszaadó lezáró művelet, amelyek azt vizsgálják, hogy minden elem/valamelyik elem/egyik elem sem felel meg egy feltételnek. Ezek az .allMatch(), az .anyMatch(), illetve a .noneMatch().

Például a következő kódrészlet azt vizsgálja, hogy a szavak stream minden eleme tartalmaz-e e betűt:

boolean eredmeny = szavak.allMatch(s -> s.contains("e"));

Hogyan vizsgálnád meg, hogy van-e olyan szó, amely x-et tartalmaz?

## Primitív típusú streamek

Van három speciális stream, amely a hatékonyság érdekében primitív típusú adatokkal dolgozik: az IntStream, a Longstream és a DoubleStream. (A többi primitív típushoz nincs ilyen.) Ezek kezelése hasonlít egymásra.

Előállításuk a normál streamekhez hasonlóan történhet. Például:

IntStream szamok = IntStream.of(3, 1, 4, 2, 5, 8);

int[] ertekek = {…};  
egeszek = IntStream.of(ertekek);

Ha a és b közötti egyesével növekvő egészeket szeretnénk (a benne van, b már nem), akkor ezt írjuk:

IntStream is = IntStream.range(a, b);

A kockadobásokat így modellezhetjük:

Random vg = new Random();  
IntStream dobasok = vg.ints(1,7);

Normál streamből a .mapToInt() metódussal állíthatunk elő IntStream-et:

IntStream hossz = szavak.mapToInt(s –> s.length());

Az IntStream-en végzett map művelet egy másik IntStream-et eredményez. Ha azt szeretnénk, hogy az eredmény objektumokból álló stream legyen, akkor a mapToObj metódust kell használni:

String szo = "Vakáció";  
Stream<String> indexek = IntStream.range(0, szo.length())  
 .mapToObj(i -> szo.substring(0, i+1));

(Eredmény: "V", "Va", "Vak", … "Vakáció")

A primitív típusú streameknek a normál streamek metódusain kívül van még négy metódusuk: sum, average, max és min. Például az alábbi kód összeadja a szavak hosszait a streamben:

int osszesHossz = szavak  
 .mapToInt(s -> s.length())  
 .sum();

A max és a min metódusoknál nem kell megadni comparator-t, és opcionális eredményt adnak.

Az average metódus is opcionális eredményt ad, mert a stream lehet üres is. A szavak átlagos hosszát így számíthatjuk ki:

double atlag = szavak  
 .mapToInt(s -> s.length())  
 .average()  
 .orElse(0);

## Az eredmények csoportosítása

Ha csoportosítani szeretnénk a stream elemeit, akkor a stream collect metódusát kell meghívnunk így:

stream.collect(Collectors.groupingBy(metódus), *művelet*)

A metódus minden elemhez meghatároz egy kulcsot. Azok az elemek tartoznak egy csoportba, amelyeknek ugyanaz a kulcsuk. Minden kulcshoz tartozik egy lista ezekből az elemekből.

A csoportosítás is lezáró művelet, amelynek eredménye egy Map.

Például:

Map<String, Long> darabSzamok = szavak  
 .collect(Collectors.groupingBy(  
 s -> s.substring(0, 1),  
 Collectors.counting()  
 ));

Ez a kód a szavak első karaktere alapján csoportosít, majd megszámolja, hány elem tartozik az egyes csoportokhoz.

A létrehozott Map kulcsa a kezdőbetű lesz, a hozzá tartozó értékek pedig a csoporthoz tartozó elemek számai.

A csoportokon belüli összeadáshoz a summingInt, a summingLong és a summingDouble metódusok használhatók. Ezeknek meg kell adni egy metódust, amely megadja, hogy mit kell összeadni.

A csoportokon belüli átlagokhoz az averageInt, az averageLong és az averageDouble metódusok használhatók az összegzéshez hasonlóan.

A csoportokon belüli legkisebb vagy legnagyobb elem megtalálásához a minBy vagy a maxBy metódusokat kell használni, és megadni nekik egy comparator-t.

Például így kereshetjük meg kezdőbetűnként a leghosszabb szót:

Map<String, Optional<String>> leghosszabbak = szavak  
 .collect(Collectors.groupingBy(  
 s -> s.substring(0, 1),  
 Collectors.maxBy((p,s) -> p.length()-s.length()))  
 ));

Nyisd meg a *GroupDemo* projektet, és futtasd le! Figyeld meg, hogyan csoportosítottunk!