

A Java haladó szintű lehetőségei

Jeszenszky Péter

Debreceni Egyetem, Informatikai Kar

jeszenszky.peter@inf.unideb.hu

Utolsó módosítás: 2019. március 11.

Java SE 8

- 2014. március 18-án jelent meg a JDK 8.
 - Lásd: JDK 8: *General Availability*
<http://mail.openjdk.java.net/pipermail/announce/2014-March/000166.html>
- Újdonságok:
 - *What's New in JDK 8*
<http://www.oracle.com/technetwork/java/javase/8-whats-new-2157071.html>
 - *JDK 8 Features* <http://openjdk.java.net/projects/jdk8/features>
- A legfontosabb változások, név szerint a lambda kifejezések és a streamek egy OpenJDK alprojekt, a *Project Lambda* keretében kerültek kifejlesztésre.
 - Lásd: *Project Lambda* <http://openjdk.java.net/projects/lambda/>

Tárgyalt témák

- Nem absztrakt interfész metódusok
- `java.util.Optional`
- Funkcionális interfészek
- Lambda kifejezések
- Streamek

Nem absztrakt interfész metódusok: interfészek fejlődése (1)

- **Probléma:** hogyan adhatók hozzá új metódusok egy már létező interfészhez?
 - Amikor egy új metódust adunk hozzá egy interfészhez, akkor egy implementációt kell nyújtanunk hozzá minden egyes, az interfészt implementáló osztályban.
 - Széles körben használt interfész esetén ez hatalmas mennyiségű munkát igényelhet!
- **Megoldás:** az alapértelmezett és statikus interfész metódusok úgy teszik lehetővé új metódusok hozzáadását egy interfészhez, hogy azok automatikusan rendelkezésre állnak minden implementációban.
 - Ráadásul ezen metódusok hozzáadása nem igényli a létező implementációk módosítását vagy újrafordítását.
 - Ezt **bináris kompatibilitás**nak nevezik.

Nem absztrakt interfész metódusok: interfészek fejlődése (2)

- Valós példa (Java SE):
 - Vegyünk például a `java.lang.Iterable<T>` interfészt a Java SE 7-ben:
<https://docs.oracle.com/javase/7/docs/api/java/lang/Iterable.html>

```
public interface Iterable<T> {  
    Iterator<T> iterator();  
}
```

Nem absztrakt interfész metódusok: interfészek fejlődése (3)

- Valós példa (Java SE): (folytatás)
 - A Java SE 8 a `forEach(consumer)` és `splitterator()` alapértelmezett metódusokat adta hozzá az interfészhez:

```
public interface Iterable<T> {  
  
    Iterator<T> iterator();  
  
    default void forEach(Consumer<? super T> action) {  
        Objects.requireNonNull(action);  
        for (T t : this) {  
            action.accept(t);  
        }  
    }  
  
    default Splitterator<T> splitterator() {  
        return Spliterators.splitteratorUnknownSize(iterator(), 0);  
    }  
}
```

Nem absztrakt interfész metódusok (1)

- Egy nem absztrakt interfész metódus egy, a `default`, `static` vagy `private` módosítók valamelyikével deklarált interfész metódus, melynek metódustörzse is van.
 - Az alapértelmezett metódusokat és statikus interfész metódusokat a Java SE 8 vezette be, a privát interfész metódusokat a Java SE 9.
- Implicit módon absztrakt minden olyan interfész metódus, melynek nincs `private`, `default` vagy `static` módosítója.

Nem absztrakt interfész metódusok (2)

- Az `abstract`, `default` és `static` módosítók kölcsönösen kizárják egymást interfész deklarációknál.
 - Fordítási hiba, ha egy interfész deklarációnak egynél több módosítója van ezek közül.
- Fordítási hiba, ha egy, a `private` módosítót tartalmazó interfész metódus deklaráció az `abstract` vagy `default` módosítót is tartalmazza.
- Azonban megengedett, hogy egy interfész metódus deklaráció a `private` és a `static` módosítót is tartalmazza.

Nem absztrakt interfész metódusok:

Alapértelmezett metódusok (1)

- Egy alapértelmezett metódus egy interfészben a `default` módosítóval deklarált példánymetódus.
 - Virtuális kiterjesztési metódusnak (***virtual extension method***) is nevezik.
- A metódustörzs a metódus implementációját szolgáltatja az interfészt a metódus felülírása nélkül implementáló osztályok számára.
- Fordítási hiba, ha egy alapértelmezett metódus a `java.lang.Object` osztály egy metódusát írja felül.

Nem absztrakt interfész metódusok:

Alapértelmezett metódusok (2)

- Amikor egy interfész kiterjeszt egy alapértelmezett metódust tartalmazó interfészt, akkor a következőket teheti:
 - Egyáltalán nem említi az alapértelmezett metódust, mely azt jelenti, hogy öröklí azt.
 - Újraderiniálhatja a metódust, felülírva azt.
 - Absztraktként deklarálhatja újra a metódust, mely a felülírására kényszeríti az implementáló osztályokat.
- Hasonlóan, amikor egy osztály implementál egy alapértelmezett metódust tartalmazó interfészt, akkor a következőket teheti:
 - Egyáltalán nem említi az alapértelmezett metódust, mely azt jelenti, hogy öröklí azt.
 - Újraderiniálhatja a metódust, felülírva azt.
 - Absztraktként deklarálhatja újra a metódust, mely a felülírására kényszeríti az alosztályokat. (Ez a lehetőség csak akkor adott, ha az osztály absztrakt.)

Nem absztrakt interfész metódusok: Alapértelmezett metódusok (3)

- Valós példa az OpenJDK 11-ből:
 - Lásd az alábbi interfészek és osztályok `splitter()` metódusát:
 - `java.lang.Iterable`
<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Iterable.html>
 - `java.util.Collection`
<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Collection.html>
 - `java.util.Set`
<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Set.html>
 - `java.util.HashSet`
<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/HashSet.html>

```

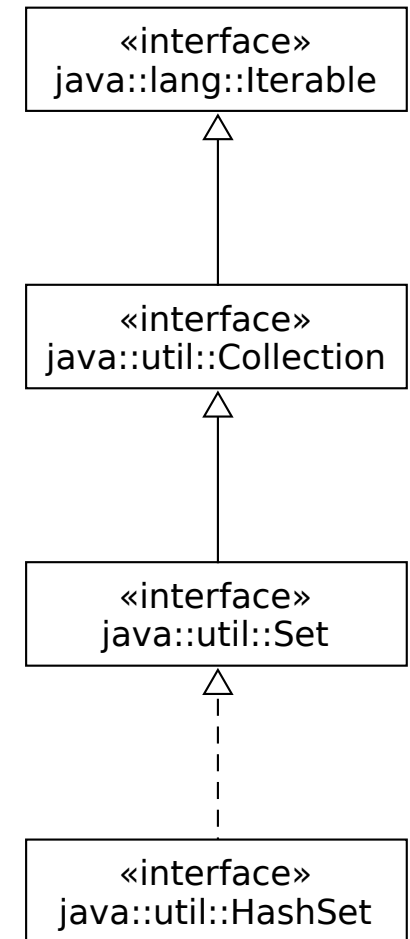
// java.lang.Iterable:
public interface Iterable<T> {
    ...
    default Spliterator<T> spliterator() {
        return Spliterators.spliteratorUnknownSize(iterator(), 0);
    }
}

// java.util.Collection:
public interface Collection<E> extends Iterable<E> {
    ...
    @Override
    default Spliterator<E> spliterator() {
        return Spliterators.spliterator(this, 0);
    }
}

// java.util.Set:
public interface Set<E> extends Collection<E> {
    ...
    @Override
    default Spliterator<E> spliterator() {
        return Spliterators.spliterator(this, Spliterator.DISTINCT);
    }
}

// java.util.HashSet:
public class HashSet<E> extends AbstractSet<E>, implements Set<E>,
    Cloneable, java.io.Serializable {
    ...
    public Spliterator<E> spliterator() {
        return new HashMap.KeySpliterator<>(map, 0, -1, 0, 0);
    }
}

```



Nem absztrakt interfész metódusok:

Alapértelmezett metódusok (5)

- Ennek nem szándékos következményeként az alapértelmezett metódusok lehetővé teszik a többszörös öröklést.

– Példa:

```
public interface A {  
    default void someMethod() {  
        System.out.println("A.someMethod() is called");  
    }  
}  
  
public interface B {  
    default void someMethod() {  
        System.out.println("B.someMethod() is called");  
    }  
}  
  
public class SomeClass implements A, B {  
    } // nem fordul le
```

Nem absztrakt interfész metódusok: Alapértelmezett metódusok (6)

- Ennek nem szándékos következményeként az alapértelmezett metódusok lehetővé teszik a többszörös öröklést.
 - Példa: (folytatás)
 - Az alábbi hibát kapjuk a SomeClass osztály fordításakor:

```
SomeClass.java:1: error: types A and B are incompatible;  
public class SomeClass implements A, B {  
    ^  
    class SomeClass inherits unrelated defaults for someMethod()  
    from types A and B  
1 error
```

Nem absztrakt interfész metódusok: Alapértelmezett metódusok (7)

- Ennek nem szándékos következményeként az alapértelmezett metódusok lehetővé teszik a többszörös öröklést.
 - Példa: (folytatás)
 - A hiba javításához az osztály újra kell, hogy definiálja a metódust:

```
public class SomeClass implements A, B {  
  
    @Override  
    public void someMethod() {  
        // implementáció adása  
    }  
  
}
```

Nem absztrakt interfész metódusok: Alapértelmezett metódusok (8)

- Ennek nem szándékos következményeként az alapértelmezett metódusok lehetővé teszik a többszörös öröklést.
 - Példa: (folytatás)
 - Az újradefiniált metódus azonban bármely deklaráló interfész alapértelmezett implementációját meghívhatja:

```
public class SomeClass implements A, B {  
  
    @Override  
    public void someMethod() {  
        A.super.someMethod();  
    }  
  
}
```

Nem absztrakt interfész metódusok: Alapértelmezett metódusok (9)

- Példák alapértelmezett metódusokra a Java SE 11-ben:
 - `java.util.Comparator`
<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Comparator.html>
 - Lásd például a `reversed()` metódust.
 - `java.lang.Iterable`
<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Iterable.html>
 - Lásd a `forEach(action)` és `splitterator()` metódusokat.
 - `java.util.Collection`
<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Collection.html>
 - Lásd például a `stream()` és `parallelStream()` metódusokat.
 - `java.util.List` <https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/List.html>
 - Lásd például a `sort(comparator)` metódust.
 - `java.util.stream.Stream`
<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/Stream.html>
 - Lásd a `dropWhile(predicate)` és `takeWhile(predicate)` metódusokat.
 - ...

Nem absztrakt interfész metódusok: statikus interfész metódusok (1)

- Egy statikus interfész metódus egy interfészben a `static` módosítóval deklarált metódus.
- A statikus interfész metódusokat nem öröklik az alinterfészek.
- A statikus interfész metódusok hívása egy bizonyos példányra történő hivatkozás nélkül történik, az osztályok statikus metódusaihoz hasonlóan.
- Fordítási hiba egy statikus metódus törzsében a `this` vagy a `super` kulcsszó előfordulása.

Nem absztrakt interfész metódusok: statikus interfész metódusok (2)

- A statikus interfész metódusok lehetővé teszik egy interfészhez kötődő konkrét segédmetódusok hozzáadását közvetlenül magához az interfészhez.
 - A Java SE 8 előtt az ilyen segédmetódusokat kizárólag külön segédosztályokban lehetett megadni.

Nem absztrakt interfész metódusok: statikus interfész metódusok (3)

- Valós példa az OpenJDK 11-ből:

`java.util.List`

<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/List.html>

```
package java.util;
...
public interface List<E> extends Collection<E> {
    ...
    static <E> List<E> of() {
        return ImmutableList.empty();
    }

    static <E> List<E> of(E e1) {
        return new ImmutableList(e1);
    }
    ...
}
```

Nem absztrakt interfész metódusok: statikus interfész metódusok (4)

- Példák statikus interfész metódusokra a Java SE 11-ben:
 - `java.util.Comparator`
<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Comparator.html>
 - Lásd például a `naturalOrder()` és `reverseOrder()` metódusokat.
 - `java.util.List`
<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/List.html>
 - Lásd a `copyOf(collection)` és az `of(...)` metódusokat.
 - `java.util.stream.Stream`
<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/Stream.html>
 - Lásd például a `builder()`, `empty()` és `of(...)` metódusokat.
 - ...

Nem absztrakt interfész metódusok: privát interfész metódusok (1)

- Egy privát interfész metódus egy interfészben a `private` módosítóval deklarált metódus.
 - A `private` módosító kombinálható a `static` módosítóval.
- A privát interfész metódusokat nem öröklik az alinterfészek.
- Az alapértelmezett metódusok és a statikus interfész metódusok közötti kódmegosztásra szolgálnak.

Nem absztrakt interfész metódusok: privát interfész metódusok (2)

- Példa:

```
public interface Bookshelf {  
  
    List<Book> getBooks();  
  
    default List<Book> filterByPublisher(String publisher) {  
        return getBooks().stream()  
            .filter(book -> book.getPublisher().equals(publisher))  
            .collect(Collectors.toList());  
    }  
  
    default List<Book> filterByKeyword(String keyword) {  
        return getBooks().stream()  
            .filter(book -> book.getKeywords().contains(keyword))  
            .collect(Collectors.toList());  
    }  
}
```

Nem absztrakt interfész metódusok: privát interfész metódusok (3)

- Példa: az előző interfész refaktorált változata, mely egy privát interfész metódust használ

```
public interface Bookshelf {  
  
    List<Book> getBooks();  
  
    default List<Book> filterByPublisher(String publisher) {  
        return filterBy(book -> book.getPublisher().equals(publisher));  
    }  
  
    default List<Book> filterByKeyword(String keyword) {  
        return filterBy(book -> book.getKeywords().contains(keyword));  
    }  
  
    private List<Book> filterBy(Predicate<Book> predicate) {  
        return getBooks().stream()  
            .filter(predicate)  
            .collect(Collectors.toList());  
    }  
}
```

java.util.Optional (1)

- Egy konténer objektum, mely vagy tartalmaz egy nem `null` értéket, vagy nem.
- Elsődlegesen olyan metódusok visszatérési típusaként szolgál, melyeknél egyértelműen szükséges a „nincs eredmény” ábrázolása és ahol `null` használata valószínűleg hibát okoz.
 - Rákényszeríti a programozót arra, hogy foglalkozzon egy érték hiányával, így tehát segíti a `NullPointerException` kivételek elkerülését.
- Egy `Optional` típusú változó értéke soha nem szabad, hogy `null` legyen, mindig egy `Optional` példányra kell, hogy mutasson.
- Lásd: `java.util.Optional<T>`
<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Optional.html>

java.util.Optional (2)

- Statikus metódusok:

- `Optional<T> empty()`:
 - Visszaad egy üres `Optional` példányt.
- `Optional<T> of(T value)`:
 - Visszaad egy `Optional` példányt a megadott nem null értékkel.
- `Optional<T> ofNullable(T value)`:
 - Visszaad egy `Optional` példányt, mely a megadott értéket tartalmazza, ha az nem null, egyébként egy üres `Optional` példányt ad vissza.

- Példánymetódusok:

- `boolean isPresent()`:
 - Visszaadja, hogy a példány tartalmaz-e értéket.
- `T get()`:
 - Ha a példány tartalmaz értéket, akkor visszaadja azt, egyébként `NoSuchElementException` kivételt dob.
- `T orElse(T other)`:
 - Ha a példány tartalmaz értéket, akkor visszaadja azt, egyébként `other`-t adja vissza.
- ...

java.util.Optional (3)

- Primitív specializált verziók:
 - `java.util.OptionalDouble`
<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/OptionalDouble.html>
 - `java.util.OptionalInt`
<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/OptionalInt.html>
 - `java.util.OptionalLong`
<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/OptionalLong.html>

java.util.Optional (4)

- Példa:

```
Optional<Book> findBook(String isbn) {  
    // ...  
}  
  
Optional<Book> optional = findBook(isbn);  
if (optional.isPresent()) {  
    Book book = optional.get();  
    // használjuk a Book objektumot  
} else {  
    // foglalkozunk az objektum hiányával  
}
```

Funkcionális interfészek (1)

- Egy funkcionális interfész egy olyan interfész, melynek csak egy absztrakt metódusa van.
 - Egyetlen absztrakt metódusú (***Single Abstract Method*** – SAM) interfészként vagy típusként is ismert.
 - Az egyetlen absztrakt metódust a funkcionális interfész **funkcionális metódusának** nevezik.
 - Egy funkcionális interfésznek több alapértelmezett, statikus és/vagy privát metódusa is lehet.
 - Az alapértelmezett és statikus interfész metódusok a Java SE 8-ban kerültek bevezetésre, a privát interfész metódusok a Java SE 9-ben.
- Lásd:
 - *The Java Language Specification, Java SE 11 Edition – Functional Interfaces.*
<https://docs.oracle.com/javase/specs/jls/se11/html/jls-9.html#jls-9.8>

Funkcionális interfészek (2)

- A `FunctionalInterface` annotáció típus szolgál annak jelzésére, hogy egy interfész funkcionális.
 - Lásd: `java.lang.FunctionalInterface`
<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/FunctionalInterface.html>
- Fordítási hibát okoz, ha egy interfész deklaráció a `@FunctionalInterface` annotációval van megjelölve, de valójában nem funkcionális interfész.
- Mivel bizonyos interfészek esetlegesen funkcionálisak, nem szükséges vagy kívánatos minden funkcionális interfész megjelölése a `@FunctionalInterface` annotációval.

Funkcionális interfészek (3)

- Példa:

```
@FunctionalInterface
public interface Task {
    void perform();
}
```

```
@FunctionalInterface
public interface Converter<F, T> {
    T convert(F from);
}
```

Beépített funkcionális interfészek (1)

- A Java SE 8-ban sok ténylegesen funkcionális interfész meg lett jelölve a `@FunctionalInterface` annotációval.
 - Lásd: *Uses of Class `java.lang.FunctionalInterface`*
<https://docs.oracle.com/javase/8/docs/api/java/lang/class-use/FunctionalInterface.html>
 - Példák:
 - `java.io.FileFilter`
<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/io/FileFilter.html>
 - `java.lang.Runnable`
<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Runnable.html>
 - `java.util.Comparator<T>`
<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Comparator.html>
 - ...

Beépített funkcionális interfészek (2)

- A Java SE 8 ráadásul sok új funkcionális interfész is bevezetett, lásd a `java.util.function` csomagot.
 - Lásd: `java.util.function`
<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/function/package-summary.html>
 - Példák:
 - `java.util.function.Function<T, R>`
 - `java.util.function.Predicate<T>`
 - `java.util.function.Supplier<T>`
 - `java.util.function.Consumer<T>`
 - ...

Beépített funkcionális interfészek (3)

- `java.util.function.Function<T, R>`
 - Egy eredményt létrehozó egyargumentumú függvényt ábrázol.
 - Funkcionális metódusa: `R apply(T t)`.
 - Az adott argumentumra alkalmazza a függvényt.
 - Nem absztrakt metódusai:
 - `andThen(after)`: visszaad egy összetett függvényt, mely először a példány által ábrázolt függvényt alkalmazza a bemenetére, majd az `after` függvényt az eredményre.
 - `compose(before)`: visszaad egy összetett függvényt, mely először a `before` függvényt alkalmazza a bemenetére, majd a példány által ábrázolt függvényt az eredményre.
 - `identity()`: visszaad egy, mindig az argumentumát visszaadó függvényt.
 - Lásd:
<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/function/Function.html>

Beépített funkcionális interfészek (4)

- `java.util.function.Predicate<T>`
 - Egy egyargumentumú predikátumot (logikai értékű függvényt) ábrázol.
 - Funkcionális metódusa: `boolean test(T t)`.
 - Kiértékeli a predikátumot az adott argumentumra.
 - Nem absztrakt metódusai:
 - `and(other)`, `or(other)`: visszaad egy összetett predikátumot, mely a példány és az `other` predikátum logikai konjunkcióját/diszjunkcióját ábrázolja.
 - `negate()`: visszaad egy, a példány logikai negáltját ábrázoló predikátumot.
 - ...
 - Lásd:
<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/function/Predicate.html>

Beépített funkcionális interfészek (5)

- `java.util.function.Supplier<T>`
 - Egy eredményeket szolgáltató objektumot ábrázol.
 - Funkcionális metódusa: `T get()`.
 - Az eredményt szolgáltatja.
 - Nem absztrakt metódusai: nincsenek
 - Lásd:
<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/function/Supplier.html>

Beépített funkcionális interfészek (6)

- `java.util.function.Consumer<T>`
 - Egy olyan műveletet ábrázol, mely egyetlen input argumentumot vár és nem ad vissza eredményt.
 - Funkcionális metódusa: `void accept(T t)`.
 - A műveletet hajtja végre az adott argumentumon.
 - A legtöbb funkcionális interfésszel ellentétben a `Consumer` várhatóan mellékhatást fejt ki.
 - Nem absztrakt metódusai:
 - `andThen(after)`: egy összetett `Consumer`-t ad vissza, mely először a példány által ábrázolt műveletet hajtja vége, majd az `after` műveletet.
 - Lásd:
<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/function/Consumer.html>

Lambda kifejezések (1)

- Egy funkcionális interfészt implementáló névtelen belső osztály egy példányát ábrázolják nagyon tömören.
 - Egy lambda kifejezés kiértékelése egy funkcionális interfészt implementáló névtelen belső osztály egy példányát hozza létre.
- Lásd: *The Java Language Specification, Java SE 11 Edition – Lambda Expressions*
<https://docs.oracle.com/javase/specs/jls/se11/html/jls-15.html#jls-15.27>

Lambda kifejezések (2)

- Tekintsük a következő példányosítást:

```
new VmilyenFunkcionálisInterfész() {  
    @Override  
    VmilyenTípus vmilyenMetódus(paraméterek) {  
        törzs  
    }  
}
```

- Az ekvivalens lambda kifejezés:

(paraméterek) -> {*törzs*}

- Egy formális paraméterlistából és egy törzsből állnak.

Lambda kifejezések (3)

- Valós példa: szál létrehozása és indítása (Java 8 előtti és Java 8 stílus)

```
Runnable runnable = new Runnable() {  
    @Override  
    public void run() {  
        System.out.println("Hello, world!");  
    }  
};  
Thread thread = new Thread(runnable);  
thread.start();
```

```
Runnable runnable = () -> System.out.println("Hello, world!");  
Thread thread = new Thread(runnable);  
thread.start();
```

Lambda kifejezések (4)

- A lambda kifejezések névtelen függvényeket ábrázolnak.

Lambda kifejezések (5)

- Példák:

```
() -> {} // Nincs paraméter, void eredmény
() -> 42 // Nincs paraméter, kifejezés törzs
() -> null // Nincs paraméter, kifejezés törzs
() -> { return 42; } // Nincs paraméter, blokk törzs return-nel
() -> { System.gc(); } // Nincs paraméter, void blokk törzs
() -> { // Összetett blokk törzs return utasításokkal
    if (true) return 12;
    else {
        int result = 15;
        for (int i = 1; i < 10; i++)
            result *= i;
        return result;
    }
}
```

Lambda kifejezések (6)

- Példák: (folytatás)

<code>(int x) -> x + 1</code>	<code>// Egyetlen deklarált típusú paraméter</code>
<code>(int x) -> { return x + 1; }</code>	<code>// Egyetlen deklarált típusú paraméter</code>
<code>(x) -> x + 1</code>	<code>// Egy kikövetkeztetett típusú paraméter</code>
<code>x -> x + 1</code>	<code>// A zárójelek opcionálisak egyetlen</code>
	<code>// kikövetkeztetett típusú paraméternél</code>
<code>(String s) -> s.length()</code>	<code>// Egyetlen deklarált típusú paraméter</code>
<code>(Thread t) -> { t.start(); }</code>	<code>// Egyetlen deklarált típusú paraméter</code>
<code>s -> s.length()</code>	<code>// Egy kikövetkeztetett típusú paraméter</code>
<code>t -> { t.start(); }</code>	<code>// Egy kikövetkeztetett típusú paraméter</code>
<code>(int x, int y) -> x + y</code>	<code>// Több deklarált típusú paraméter</code>
<code>(x, y) -> x + y</code>	<code>// Több kikövetkeztetett típusú paraméter</code>
<code>(var x, var y) -> x + y</code>	<code>// Több kikövetkeztetett típusú paraméter</code>
<code>(x, int y) -> x + y</code>	<code>// Illegális: nem keverhető deklarált és</code>
	<code>// kikövetkeztetett típus</code>

Lambda kifejezések (7)

- Paraméterek:
 - Egy lambda kifejezés formális paramétereit, ha vannak, paraméter specifikátorok egy vesszőkkel elválasztott zárójelezett listája vagy azonosítók egy vesszőkkel elválasztott zárójelezett listája adja meg.
 - Paraméter specifikátorok egy listájában minden egyes paraméter specifikátort opcionális módosítók, egy típus (vagy var) és a paraméter nevét megadó azonosító alkotnak.
 - Azonosítók egy listájában minden egyes azonosító egy paraméter nevét adja meg.
 - Ha egy lambda kifejezésnek nincsenek formális paramétereit, akkor a -> token és a törzs előtt egy üres zárójelpár jelenik meg.
 - Ha egy lambda kifejezésnek pontosan egy formális paramétere van és azt egy azonosító adja meg, nem pedig paraméter specifikátor, akkor elhagyhatók az azonosító körül a zárójelek.

Lambda kifejezések (8)

- Paraméterek: (folytatás)
 - Egy lambda kifejezés minden egyes formális paraméterének egy kikövetkeztetett vagy deklarált típusa van.
 - Fordítási hibát okoz, ha egy lambda kifejezés deklarált típusú és kikövetkeztetett típusú formális paramétereket is deklarál.
 - A Java SE 11 vezeti be a var használatát lambda paraméterekhez, mely lehetővé teszi hozzájuk annotációk és módosítók használatát.

Lambda kifejezések (9)

- Törzs:
 - Egyetlen kifejezés vagy egy blokk.
 - Példa:

```
file -> file.isFile() && file.getName().endsWith(".java");
```

```
file -> {  
    return file.isFile() && file.getName().endsWith(".java");  
}
```

Lambda kifejezések (10)

- Törzs: (folytatás)
 - A lambda kifejezések hatáskör szempontjából nem vezetnek be új szintet.
 - A törzsben megjelenő nevek, a `this` és `super` kulcsszavak jelentése ugyanaz, mint a körülvevő szöveggörnyezetben (azt kivéve, hogy a lambda paraméterek új neveket vezetnek be).
 - A befoglaló szöveggörnyezet egy lokális változója csak akkor hivatkozható, ha `final` vagy gyakorlatilag `final`, egyébként fordítási hibát kapunk.
 - Egy változó gyakorlatilag `final` (*effectively final*), ha a kezdőértékadást követően nem kap új értéket.

Lambda kifejezések (11)

- Törzs: (folytatás)
 - Példák (nem) gyakorlatilag `final` lokális változókra:

```
void m1(int x) {  
    int y = 1;  
    foo(() -> x + y);  
    // Legális: x és y is gyakorlatilag final.  
}  
  
void m2(int x) {  
    foo(() -> x + 1);  
    x++;  
    // Illegális: x nem gyakorlatilag final (növelésre kerül).  
}
```

Lambda kifejezések (12)

- Lambda kifejezés kiértékelése nem eredményezi a kifejezés törzsének kiértékelését, a végrehajtás később történhet, amikor is a funkcionális interfész megfelelő metódusa meghívásra kerül.

Lambda kifejezések (13)

- Fordítási hibát okoz, ha egy lambda kifejezés értékadáستól, hívástól vagy típuskényszerítéstől eltérő bármely más szövegkörnyezetben fordul elő egy programban.
 - Az `x -> x + 1` lambda kifejezés megengedett előfordulásai például a következők:
 - `IntFunction f = x -> x + 1;`
 - `(IntFunction) x -> x + 1`
 - `IntStream.of(1, 2, 3).map(x -> x + 1)`

Lambda kifejezések (14)

- Példa:

- Lásd:

`java.util.function.IntBinaryOperator`

<https://docs.oracle.com/javase/8/docs/api/java/util/function/IntBinaryOperator.html>

```
IntBinaryOperator addition = (a, b) -> a + b;  
IntBinaryOperator subtraction = (a, b) -> a - b;  
System.out.println(addition.applyAsInt(40, 2));           // 42  
System.out.println(subtraction.applyAsInt(10, 20));        // -10
```

Metódus referenciák (1)

- Egy metódus referencia arra szolgál, hogy egy metódushívásra hivatkozzunk anélkül, hogy ténylegesen hívás történne.
 - Bizonyos formájú metódus referencia kifejezések lehetővé teszik az osztálpéldány létrehozás vagy tömb létrehozás úgy történő kezelését is, mintha azok metódushívások lennének.
- Egy metódus referencia kifejezés kiértékelése egy funkcionális interfésztípus egy példányát hozza létre.
 - Ez nem eredményezi a megfelelő metódus végrehajtását, a végrehajtás később történhet, amikor is a funkcionális interfész megfelelő metódusa meghívásra kerül.
- Lásd: *The Java Language Specification, Java SE 11 Edition – Method Reference Expressions*
<https://docs.oracle.com/javase/specs/jls/se11/html/jls-15.html#jls-15.13>

Metódus referenciák (2)

- Fordítási hibát okoz, ha egy metódus referencia kifejezés értékadástól, hívástól vagy típuskényszerítéstől eltérő bármely más szövegkörnyezetben fordul elő egy programban.
 - A `System.out::println` metódus referencia kifejezés megengedett előfordulásai például a következők:
 - `Consumer<Object> consumer = System.out::println;`
 - `(Consumer<Object>) System.out::println`
 - `Stream.of(1, 2, 3).forEach(System.out::println)`

Metódus referenciák (3)

- Amikor egy típus több metódusának ugyanaz a neve vagy egy osztálynak egynél több konstruktora van, akkor a megfelelő metódus vagy konstruktor kiválasztása a metódus referencia kifejezés által megcélzott funkcionális interfésztípuson alapul.
- Ha egy metódus referencia egy példánymetódusra hivatkozik, akkor az implicit lambda kifejezésnek egy extra paramétere van ahhoz képest, amikor egy statikus metódusra hivatkozik.
 - Az extra paraméter azt a példányt határozza meg, melyen a metódus meghívásra kerül.

Metódus referenciák (4)

- Példák:
 - Referencia egy statikus metódusra:
 - `System::currentTimeMillis`
 - `java.time.LocalDate::parse`
 - Referencia egy bizonyos objektum egy példánymetódusára:
 - `"Hello, world!":length`
 - `System.out::println`

Metódus referenciák (5)

- Példák: (folytatás)
 - Referencia egy bizonyos típus egy tetszőleges objektumának egy példánymetódusára:
 - `String::length`
 - `String::startsWith`
 - `List::size`
 - `List<String>::size`
 - Referencia egy konstruktorra:
 - `int[]::new`
 - `ArrayList::new`
 - `ArrayList<String>::new`

Metódus referenciák (6)

- Nem lehetséges egy bizonyos szignatúra előírása, mint például `Arrays::sort(int[])`. Ehelyett a funkcionális interfész szolgáltat a kiválasztási algoritmus bemenetéül szolgáló argumentum típusokat.

Metódus referenciák (7)

- Példa:
 - Referencia egy statikus metódusra:
 - Lásd: `java.util.function.LongSupplier`
<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/function/LongSupplier.html>

```
LongSupplier supplier = System::currentTimeMillis;  
System.out.println(supplier.getAsLong()); // 1549705345645  
  
Function<String, LocalDate> parser = LocalDate::parse;  
LocalDate localDate = parser.apply("2019-02-09")
```

Metódus referenciák (8)

- Példa:
 - Hivatkozás egy bizonyos objektum egy példánymetódusára:
 - Lásd: `java.util.function.IntSupplier`
<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/function/IntSupplier.html>

```
IntSupplier supplier = "Hello, world!":length;  
System.out.println(supplier.getAsInt());    // 13  
  
Consumer<Object> consumer = System.out::println;  
consumer.accept("Hello, world!");           // Hello, world!  
consumer.accept(42);                         // 42  
consumer.accept(LocalDate.now());           // 2019-02-09
```

Metódus referenciák (9)

- Példa:

- Hivatkozás egy bizonyos típus egy tetszőleges objektumának egy példánymetódusára:

- Lásd:

- `java.util.function.ToIntFunction`

<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/function/ToIntFunction.html>

- `java.util.function.BiPredicate`

<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/function/BiPredicate.html>

```
ToIntFunction<String> length = String::length;  
System.out.println(length.applyAsInt("Hello, world!"));    // 13  
  
BiPredicate<String, String> startsWith = String::startsWith;  
System.out.println(startsWith.test("Hello, world!", "He")); // true
```

Metódus referenciák (10)

- Példa:
 - Referencia konstruktorra:

```
IntFunction<int[]> intArrayFactory = int[]::new;  
int[] a = intArrayFactory.apply(5);
```

```
Supplier<ArrayList<String>> arrayListFactory1 =  
    ArrayList<String>::new; // alapértelmezett konstruktor  
var list1 = arrayListFactory1.get();  
list1.add("Hello, world!");
```

```
IntFunction<ArrayList<String>> arrayListFactory2 =  
    ArrayList<String>::new; //egyargumentumú konstruktor  
var list2 = arrayListFactory2.apply(5)  
list2.add("Hello, world!");
```

Metódus referenciák (11)

- Gyakorlati példa:

```
String[] fruits = new String[] {  
    "plum",  
    "PEACH",  
    "apricot",  
    "BANANA",  
    "APPLE",  
    "pear"  
};  
Arrays.sort(fruits, String::compareToIgnoreCase);  
System.out.println(Arrays.toString(fruits));  
// [APPLE, apricot, BANANA, PEACH, pear, plum]
```

Streamek

- Egy stream elemek egy sorozata, melyen műveletek végezhetők.
- A `java.util.stream.Stream<T>` interfész ábrázol egy streamet.
 - Lásd:
<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/Stream.html>
- Lásd a `java.util.stream` csomagot is.
<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/package-summary.html>

Streamek: eltérések a kollekcióktól

- A streamek több tekintetben is különböznek a kollekcióktól:
 - **Nincs mögöttük tárhely:** egy stream nem egy elemeket tároló adatszerkezet, hanem elemeket továbbít egy forrásból.
 - **Funkcionális természetűek:** egy művelet egy streamen egy eredményt hoz létre, de nem módosítja a stream forrását.
 - **Nem feltétlenül korlátosak:** míg a kollekciók véges méretűek, a streamek nem szükségszerűen; a rövidzár műveletek lehetővé tehetik számítások véges időben történő befejezését végtelen streameken.
 - **Fogyaszthatók:** egy stream elemei csupán egyszer kerülnek meglátogatásra a stream élete során, pont úgy, mint egy iterátornál.

Streamek: létrehozás (1)

- Streamek többféle módon hozhatók létre, például:
 - **Kollekciókból:** a `stream()` és `parallelStream()` metódusokkal.
 - **Tömbökből:** a `java.util.Arrays` osztály `stream(...)` statikus gyártó metódusaival.
 - **Egyedi objektumokból:**
 - A `java.util.Stream` interfész `of(T t)` és `of(T... values)` statikus gyártó metódusaival.
 - A `java.util.Stream` interfész `builder()` metódusa által visszaadott `Stream.Builder` objektummal.

Streamek: létrehozás (2)

- Számos Java SE osztály biztosít metódusokat streamek szerzéséhez, például:
 - **java.io.BufferedReader**: a `lines()` metódus sorok egy streamjét adja vissza.
 - **java.lang.String**: a `lines()` metódus sorok egy streamjét adja vissza.
 - ...
- Lásd: *Uses of Interface java.util.stream.Stream*
<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/class-use/Stream.html>

Streamek: létrehozás (3)

- Példa:

```
// Stream létrehozása kollekcióból:
```

```
import java.util.List;
```

```
...
```

```
List<String> list = List.of("apple", "banana", "orange", "pear");
```

```
Stream<String> stream = list.stream();
```

```
// Stream létrehozása tömbből:
```

```
import java.util.Arrays;
```

```
...
```

```
Stream<String> stream = Arrays.stream(new String[] {  
    "apple",  
    "banana",  
    "orange",  
    "pear"  
});
```

Streamek: létrehozás (4)

- Példa: (folytatás)

```
// Stream létrehozása egyedi objektumokból:
```

```
Stream<String> stream = Stream.of("apple", "banana", "orange", "pear");
```

```
// Stream létrehozása egyedi objektumokból Stream.Builder-rel:
```

```
Stream.Builder<String> builder = Stream.builder();
```

```
Stream<String> stream = builder.add("apple")
```

```
                                .add("banana")
```

```
                                .add("orange")
```

```
                                .add("pear")
```

```
                                .build();
```

Streamek: létrehozás (5)

- Példa: (folytatás)

```
// Stream létrehozása sztringből:  
Stream<String> stream = "Hello,\nworld!".lines();
```

```
// Stream létrehozása szövegállományból:  
import java.nio.file.Files;  
import java.nio.file.Path;  
...  
Stream<String> stream = Files.lines(Path.of("lines.txt"));
```

Streamek: műveletek (1)

- A stream műveletek két típusba sorolhatók:
 - A **köztes műveletek** (*intermediate operations*) egy új streamet adnak vissza.
 - Példák: `filter()`, `map()`, `sorted()`
 - A **terminális műveletek** (*terminal operations*) egy streamtől különböző eredményt hoznak létre vagy mellékhataást eredményeznek.
 - Tehát `void` vagy nem stream visszatérési típusúak.
 - Példák: `count()`, `max()`, `forEach()`

Streamek: műveletek (2)

- A köztes műveletek további két típusba sorolhatók:
 - Az **állapotmentes műveletek** (*stateless operations*) nem őriznek a korábban látott elemekből állapotot, amikor egy új elemet dolgoznak fel, tehát minden egyes elem a többi elemen történő műveletektől függetlenül dolgozható fel.
 - Példák: `filter()`, `map()`
 - Az **állapotőrző műveletek** (*stateful operations*) felhasználhatnak a korábban látott elemekből állapotot, amikor új elemeket dolgoznak fel, eredmény létrehozása előtt szükséges lehet számukra a teljes bemenet feldolgozása.
 - Példák: `distinct()`, `sorted()`

Streamek: műveletek (3)

- Bizonyos műveletek **rövidzár műveletnek** (***short-circuiting operation***) vannak jelölve:
 - Egy köztes művelet rövidzár, ha végtelen bemenetből is létrehozhat eredményként egy véges streamet.
 - Példák: `limit()`, `takeWhile()`
 - Egy terminális művelet rövidzár, ha végtelen bemenet esetén is befejezheti működését véges időben.
 - Példák: `anyMatch()`, `findFirst()`
- Ha egy csővezetékben egy rövidzár művelet van, az csupán szükséges, de nem elégséges feltétele annak, hogy egy végtelen bemenet feldolgozása véges időben fejeződjön be.

Streamek: sorrend

- A forrástól és a köztes műveletektől függően a streameknek lehet egy meghatározott sorrendje (***encounter order***), melyben az elemeket a műveletek számára szolgáltatják.
 - Bizonyos stream források (mint például a `java.util.List` vagy a tömbök) lényegükből fakadóan rendezettek, míg mások (például a `java.util.HashSet`) nem.
 - Néhány köztes művelet, mint például a `sorted()`, egy sorrendet írhat elő egy egyébként rendezetlen streamhez, mások pedig rendezetlenné tehetnek egy rendezett streamet, mint például a `BaseStream.unordered()`.
 - Néhány terminális művelet továbbá figyelmen kívül hagyhatja a sorrendet, mint például a `forEach()` párhuzamos stream csővezetékeknél.

Streamek: viselkedési paraméterek kívánatos jellemzői (1)

- A legtöbb stream művelet a felhasználó által meghatározott viselkedést leíró paramétereket vár.
- Ezek a viselkedési paraméterek mindig egy funkcionális interfész példányai és rendszerint lambda kifejezések vagy metódus referenciák.

Streamek: viselkedési paraméterek kívánatos jellemzői (2)

- A viselkedési paraméterek kívánatos jellemzői:
 - **Interferencia-mentesség (*non-interference*):**
 - A viselkedési paraméterek nem szabad, hogy módosítsák a stream adatforrását.
 - **Állapotmentesség (*statelessness*):**
 - Egy lambda kifejezés (vagy a megfelelő funkcionális interfészt implementáló más objektum) állapotörző, ha eredménye olyan állapottól függ, mely a stream csővezeték végrehajtása során megváltozhat.
 - A stream csővezeték eredménye nemdeterminisztikus vagy rossz lehet, ha a stream műveletek viselkedési paraméterei állapotörzők.

Streamek: viselkedési paraméterek kívánatos jellemzői (3)

- Példa az interferencia-mentességi követelmény megsértésére:
 - Az alábbi kód `java.util.ConcurrentModificationException` kivételt dob:

```
List<String> list = new ArrayList<>(List.of("apple", "banana"));  
list.stream()  
    .peek(s -> list.add("peach"))  
    .forEach(System.out::println); // kivételt dob
```

Streamek: viselkedési paraméterek kívánatos jellemzői (4)

- Példa az állapotmentességi követelmény megsértésére:

```
for (int i = 0; i < 5; i++) {  
    Set<Integer> seen = new HashSet<>();  
    int sum = IntStream.of(1, 2, 1, 3, 2, 1, 4)  
        .parallel()  
        .map(n -> seen.add(n) ? n : 0)  
        .sum();  
    System.out.println(sum);  
}  
// 14  
// 11  
// 12  
// 10  
// 10
```

Streamek: viselkedési paraméterek kívánatos jellemzői (5)

- Példa az állapotmentességi követelmény megsértésére: (folytatás)
 - Ahhoz, hogy mindig a helyes eredményt kapjuk, egy szinkronizált (szálbiztos) halmaz szükséges, azonban ez aláássa a párhuzamosságot!

```
for (int i = 0; i < 5; i++) {  
    Set<Integer> seen = Collections.synchronizedSet(new HashSet<>());  
    int sum = IntStream.of(1, 2, 1, 3, 2, 1, 4)  
        .parallel()  
        .map(n -> seen.add(n) ? n : 0)  
        .sum();  
    System.out.println(sum);  
}  
// 10  
// 10  
// 10  
// 10  
// 10
```

Streamek: viselkedési paraméterek kívánatos jellemzői (6)

- Példa az állapotmentességi követelmény megsértésére: (folytatás)
 - A helyes eredmény valójában állapotőrző lambda kifejezés használata nélkül, a párhuzamosságot maximálisan kihasználó módon is megkapható.

```
for (int i = 0; i < 5; i++) {  
    int sum = IntStream.of(1, 2, 1, 3, 2, 1, 4)  
        .parallel()  
        .distinct()  
        .sum();  
    System.out.println(sum);  
}  
// 10  
// 10  
// 10  
// 10  
// 10
```

Streamek: viselkedési paraméterek kívánatos jellemzői (7)

- Kerülendő a viselkedési paramétereknél a mellékhatás.
 - A `forEach()` és `forEachOrdered()` terminális műveletek kivételével a viselkedési paraméterek mellékhatásai nem mindig kerülnek végrehajtásra.
 - Megengedett egy implementáció számára a számítás optimalizálásához műveletek (vagy teljes szakaszok) kihagyása a csővezetékbeől – ennél fogva viselkedési paraméterek végrehajtásának kihagyása –, ha ez nem befolyásolja az eredményt.

Streamek: viselkedési paraméterek kívánatos jellemzői (8)

- Kerülendő a viselkedési paramétereknél a mellékhatás. (folytatás)
 - Példa: a `count()` terminális művelet
 - Egy implementáció dönthet úgy, hogy nem hajtja végre a stream csővezetékét, ha az elemek számát közvetlenül a stream forrásból is meg tudja határozni.
 - A `map()` és `peek()` köztes műveletek nem kerülnek végrehajtásra az alábbi csővezetékben!

```
List.of("apple", "banana", "peach")  
    .stream()  
    .map(String::toUpperCase)  
    .peek(System.out::println)  
    .count(); // eredmény: 3
```

Streamek: csővezetékek (1)

- Stream műveletek egy csővezetékké láncolhatók össze.
- Egy stream csővezeték egy forrásból áll, melyet nulla vagy több köztes művelet és egy terminális művelet követ.

Streamek: csővezetékek (2)

- A köztes műveletek mindig lusta kiértékelésűek.
 - Egy köztes művelet végrehajtása ténylegesen nem eredményez semmiféle műveletvégzést.
 - A csővezeték forrásának bejárása nem kezdődik el, míg a terminális művelet végrehajtásra nem kerül.
- Majdnem minden esetben mohó kiértékelésűek a terminális műveletek.
 - Egy terminális művelet végrehajtása indítja el az adatforrás bejárását, a csővezeték feldolgozása a visszatérés előtt fejeződik be.

Streamek: csővezetékek (3)

- Példa:

```
Stream.of("banana", "apple", "pear", "orange")  
  .filter(s -> s.length() > 4)    // köztes művelet  
  .map(String::toUpperCase)        // köztes művelet  
  .sorted()                        // köztes művelet  
  .forEach(System.out::println); // terminális művelet  
// APPLE  
// BANANA  
// ORANGE
```

Streamek: csővezetékek (4)

- A terminális művelet végrehajtása során a csővezeték elhasználdódik és nem többé használható.
 - Példa:

```
Stream<String> stream = Stream.of("apple", "banana", "orange", "pear")  
    .filter(s -> s.length() == 6);  
stream.anyMatch(s -> s.endsWith("e")); // eredmény: true  
stream.count();                       // eredmény: IllegalStateException
```

Streamek: csővezetékek (5)

- A terminális művelet végrehajtása során a csővezeték elhasználdik és nem többé használható.
 - Ha újra be kell járni ugyanazt az adatforrást, akkor egy új streamet kell szerezni az adatforrástól.
- Példa:

```
Supplier<Stream<String>> supplier =  
    () -> Stream.of("apple", "banana", "orange", "pear")  
               .filter(s -> s.length() == 6);  
supplier.get().anyMatch(s -> s.endsWith("e"));    // eredmény: true  
supplier.get().count();                          // eredmény: 2
```

Streamek: csővezetékek (6)

- A köztes műveletek lusta kiértékelése lehetőséget kínál optimalizálásra.
 - Köztes műveletek egy láncra végrehajtható az adatokon egyszer végighaladva minimális közbülső állapottal.

Streamek: csővezetékek (7)

- Kizárólag állapotmentes köztes műveleteket tartalmazó csővezetékek feldolgozhatók egyetlen menetben, szekvenciálisan és párhuzamosan is, minimális puffereléssel.

Streamek: primitív streamek (1)

- A következő specializált stream interfészek állnak rendelkezésre primitív típusú értékek sorozatainak feldolgozásához:
 - `java.util.stream.IntStream`
<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/IntStream.html>
 - `java.util.stream.DoubleStream`
<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/DoubleStream.html>
 - `java.util.stream.LongStream`
<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/LongStream.html>

Streamek: primitív streamek (2)

- A műveleteik olyan specializált funkcionális interfészek példányait várják paraméterként, mint például az `IntConsumer`, `IntFunction`, `IntPredicate` vagy `IntSupplier`.
- Az objektum streamekhez képest további terminális műveleteket biztosítanak, mint például az `average()` és `sum()`.

Streamek: primitív streamek (3)

- Példa:

```
IntStream.of(1, 2, 3, 4)
    .map(i -> i * i)
    .average()
    .ifPresent(System.out::println); // 7.5
```

```
IntStream.range(1, 5)
    .map(i -> i * i)
    .average()
    .ifPresent(System.out::println); // 7.5
```

```
IntStream.rangeClosed(1, 3)
    .mapToObj(i -> "x" + i)
    .forEach(System.out::println);
// x1
// x2
// x3
```

Streamek: primitív streamek (4)

- Példa: (folytatás)

```
Stream.of("banana", "fig", "mango")  
    .mapToInt(String::length)    // egy IntStream-et ad vissza  
    .forEach(System.out::println);  
// 6  
// 3  
// 4
```

```
Stream.of("A1", "B2", "C3", "D4")  
    .map(s -> s.substring(1))  
    .mapToInt(Integer::parseInt)    // egy IntStream-et ad vissza  
    .max()  
    .ifPresent(System.out::println); // 4
```

```
DoubleStream.of(1.2, 2.3, 3.4, 4.5)  
    .mapToLong(Math::round)    // egy LongStream-et ad vissza  
    .sum();    // eredmény: 11
```

Streamek: csővezeték végrehajtás (1)

- A csővezetékek végrehajtása vertikálisan nem pedig horizontálisan történik!
 - Példa:

```
Stream.of("apple", "banana", "orange", "pear")
    .filter(s -> {
        System.out.println("filter: " + s);
        return true;
    })
    .forEach(s -> System.out.println("forEach: " + s));
```

```
// Output:
filter: apple
forEach: apple
filter: banana
forEach: banana
filter: orange
forEach: orange
filter: pear
forEach: pear
```

Streamek: csővezeték végrehajtás (2)

- A vertikális végrehajtás csökkentheti az elemeken végrehajtandó műveletek számát.
 - Példa:

```
Stream.of("apple", "banana", "orange", "pear")
    .map(s -> {
        System.out.println("map: " + s);
        return s.toUpperCase();
    })
    .anyMatch(s -> {
        System.out.println("anyMatch: " + s);
        return s.startsWith("B");
    });
```

```
// Output:
map: apple
anyMatch: APPLE
map: banana
anyMatch: BANANA
```

Streamek: csővezeték végrehajtás (3)

- A műveletek sorrendje jelentős hatással lehet a teljesítményre!
 - Példa: vessük össze a kimenetet a következő példa kimenetével!

<pre>Stream.of("apple", "banana", "orange", "pear") .map(s -> { System.out.println("map: " + s); return s.toUpperCase(); }) .filter(s -> { System.out.println("filter: " + s); return s.startsWith("A"); }) .forEach(s -> System.out.println("forEach: " + s));</pre>	<pre>// Output: map: apple filter: APPLE forEach: APPLE map: banana filter: BANANA map: orange filter: ORANGE map: pear filter: PEAR</pre>
--	--

Streamek: csővezeték végrehajtás (4)

- A műveletek sorrendje jelentős hatással lehet a teljesítményre!
 - Példa:

```
Stream.of("apple", "banana", "orange", "pear")
    .filter(s -> {
        System.out.println("filter: " + s);
        return s.startsWith("a");
    })
    .map(s -> {
        System.out.println("map: " + s);
        return s.toUpperCase();
    })
    .forEach(s -> System.out.println("forEach: " + s));
```

```
// Output:
filter: apple
map: apple
forEach: APPLE
filter: banana
filter: orange
filter: pear
```

Streamek: csővezeték végrehajtás (5)

- A műveletek sorrendje jelentős hatással lehet a teljesítményre!
 - Példa: a `sorted()` egy állapotörző köztes művelet, mely a teljes bemenetet elfogyaszthatja, mielőtt eredményt adna. Vessük össze a kimenetet a következő példa kimenetével!

```
Stream.of("pear", "banana", "apple", "orange")
    .sorted((s1, s2) -> {
        System.out.printf("sorted: %s, %s\n", s1, s2);
        return s1.compareTo(s2);
    })
    .filter(s -> {
        System.out.println("filter: " + s);
        return s.startsWith("a");
    })
    .map(s -> {
        System.out.println("map: " + s);
        return s.toUpperCase();
    })
    .forEach(s -> System.out.println("forEach: " + s));
```

```
// Output:
sorted: banana, pear
sorted: apple, banana
sorted: orange, apple
sorted: orange, banana
sorted: orange, pear
filter: apple
map: apple
forEach: APPLE
filter: banana
filter: orange
filter: pear
```

Streamek: csővezeték végrehajtás (6)

- A műveletek sorrendje jelentős hatással lehet a teljesítményre!
 - Példa: itt a `filter()` művelet egyetlen elemre redukálja a streamet és így egyáltalán nem történik rendezés.

```
Stream.of("pear", "banana", "apple", "orange")
    .filter(s -> {
        System.out.println("filter: " + s);
        return s.startsWith("a");
    })
    .sorted((s1, s2) -> {
        System.out.printf("sorted: %s, %s\n", s1, s2);
        return s1.compareTo(s2);
    })
    .map(s -> {
        System.out.println("map: " + s);
        return s.toUpperCase();
    })
    .forEach(s -> System.out.println("forEach: " + s));
```

```
// Output:
filter: pear
filter: banana
filter: apple
filter: orange
map: apple
forEach: APPLE
```

Streamek: redukciós műveletek (1)

- Egy redukciós művelet egy terminális művelet, mely egy input elemsorozatból egyetlen összesítő eredményt képez egy egyesítő művelet ismételt alkalmazásával.
 - Példák: számok összegének vagy maximumának meghatározása, elemek összegyűjtése egy listába.
- A stream osztályok általános célú redukciós műveletei a `reduce()` és a `collect()`, de specializált redukciós műveleteik is vannak, mint például a `sum()`, `max()` vagy `count()`.

Streamek: redukciós műveletek (2)

- Az ilyen műveletek könnyedén valósíthatók meg egyszerű szekvenciális ciklusokként, mint például:

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

Streamek: redukciós műveletek (3)

- A streamek azonban egy sokkal absztraktabb és tömörebb kifejezési formát használnak a számítás leírására, mely lehetővé teszi a párhuzamosítást.
 - Például az előbbi szekvenciális ciklus az alábbi módon írható:

```
int sum = numbers.stream().reduce(0, (a, b) -> a + b);
```

vagy

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

Streamek: redukciós műveletek: reduce (1)

- A `reduce ( )` egy terminális művelet, mely elemek egy sorozatát egyetlen elemre redukálja.

Streamek: redukciós műveletek: reduce (2)

- Terminológia:
 - **Egységelem** (*identity*): a redukció kezdőértéke, egyben alapértelmezett értéke arra az esetre, ha nincsenek input elemek.
 - **Akkumulátor** (*accumulator*): egy függvény, mely két paramétert kap, nevezetesen a redukció egy részleges eredményét és a következő elemet, melyekből egy új részleges eredményt állít elő.
 - **Egyesítő** (*combiner*): egy kétparaméteres függvény, mely két részleges eredményt kap és azokat egy új részleges eredménnyé egyesíti.
 - Az egyesítő párhuzamos redukcióknál szükséges, ahol a bemenet felosztásra kerül és minden egyes partícióhoz egy részleges eredmény kerül kiszámításra, majd a részleges eredmények egyesítése révén kerül előállításra a végső eredmény.

Streamek: redukciós műveletek: reduce (3)

- Az akkumulátor és az egyesítő is egy asszociatív, interferencia mentes és állapotmentes függvény kell, hogy legyen.

Streamek: redukciós műveletek: reduce (4)

- A `reduce()` műveletnek a következő három formája van:
 - **`reduce(akkumulátor)`**:
[https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/Stream.html#reduce\(java.util.function.BinaryOperator\)](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/Stream.html#reduce(java.util.function.BinaryOperator))
 - Egy akkumulátor (`BinaryOperator`) alkalmazásával végez redukciót.
 - **`reduce(egységelem, akkumulátor)`**:
[https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/Stream.html#reduce\(T,java.util.function.BinaryOperator\)](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/Stream.html#reduce(T,java.util.function.BinaryOperator))
 - Egy egységelem és egy akkumulátor (`BinaryOperator`) alkalmazásával végez redukciót.
 - **`reduce(egységelem, akkumulátor, egyesítő)`**:
[https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/Stream.html#reduce\(U,java.util.function.BiFunction,java.util.function.BinaryOperator\)](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/Stream.html#reduce(U,java.util.function.BiFunction,java.util.function.BinaryOperator))
 - Egy egységelem, egy akkumulátor (`BiFunction`) és egy egyesítő (`BinaryOperator`) alkalmazásával végez redukciót.
 - Sok ezt a formát használó redukció sokkal egyszerűbben adható meg `map` és `reduce` műveletek explicit kombinálásával.

Streamek: redukciós műveletek: reduce (5)

- Példák:

```
IntStream.of(1, 3, 1, 5, 2, 3)
    .reduce((a, b) -> a + b)
    .ifPresent(System.out::println); // 15
```

```
IntStream.of(1, 3, 1, 5, 2, 3)
    .reduce(Integer::sum)
    .ifPresent(System.out::println); // 15
```

```
IntStream.of(1, 3, 1, 5, 2, 3)
    .reduce((a, b) -> a * b)
    .ifPresent(System.out::println); // 90
```

```
IntStream.of(1, 3, 1, 5, 2, 3)
    .reduce(Integer::max)
    .ifPresent(System.out::println); // 5
```

Streamek: redukciós műveletek: reduce (6)

- Példák: (folytatás)

```
Stream.of("banana", "kiwi", "pineapple", "mango")  
    .reduce((x, y) -> x.length() > y.length() ? x : y)  
    .ifPresent(System.out::println); // pineapple
```

```
Stream.of("banana", "kiwi", "pineapple", "mango")  
    .reduce(String::concat)  
    .ifPresent(System.out::println);  
// bananakiwipineapplemango
```

```
Stream.of("banana", "kiwi", "pineapple", "mango")  
    .reduce((x, y) -> x + "|" + y)  
    .ifPresent(System.out::println);  
// banana|kiwi|pineapple|mango
```

Streamek: redukciós műveletek: reduce (7)

- Példák: (folytatás)

```
Integer sum1 = IntStream.of(1, 3, 1, 5, 2, 3)
    .reduce(0, (a, b) -> a + b);    // eredmény: 15
```

```
Integer sum2 = IntStream.of(1, 3, 1, 5, 2, 3)
    .reduce(0, Integer::sum);        // eredmény: 15
```

```
Integer prod = IntStream.of(1, 3, 1, 5, 2, 3)
    .reduce(1, (a, b) -> a * b);    // eredmény: 90
```

```
Integer max = IntStream.of(1, 3, 1, 5, 2, 3)
    .reduce(Integer.MIN_VALUE, Integer::max); // eredmény: 5
```

Streamek: redukciós műveletek: reduce (8)

- A teljesítménnyel kapcsolatos kérdés:
 - Egy akkumulátor függvény jellemzően egy új értéket ad vissza a stream minden egyes elemének feldolgozásakor. Ennek jelentős hatása lehet a teljesítményre!
 - Példa:
 - Itt a `concat ( )` függvény a stream minden egyes eleméhez egy új `String` objektumot hoz létre.

```
Stream<String> strings;  
...  
String concatenated = strings.reduce("", String::concat)
```

Streamek: redukciós műveletek:

`collect (1)`

- Egy módosítható redukciós művelet egy módosítható eredmény konténerbe (mint például egy kollekció vagy egy `StringBuilder`) gyűjti össze az input elemeket a stream elemeinek feldolgozásakor.
- A módosítható redukciós művelet neve `collect()`.

Streamek: redukációs műveletek: collect (2)

- Terminológia:
 - **Ellátó** (*supplier*): egy új módosítható eredmény konténert létrehozó függvény.
 - Párhuzamos végrehajtásnál a függvény többször is meghívható, minden egyes alkalommal egy új értéket kell, hogy visszaadjon.
 - **Akkumulátor** (*accumulator*): egy konténerbe egy elemet helyező függvény.
 - **Egyesítő** (*combiner*): egy függvény, mely két részleges eredmény konténert fésül össze úgy, hogy a második konténer elemeit az első konténerbe helyezi.
 - **Befejező** (*finisher*): egy függvény, mely egy végső transzformációt végez egy eredmény konténeren.
 - **Gyűjtő** (*collector*): egy ellátó, egy akkumulátor, egy egyesítő és egy opcionális befejező alkotja.

Streamek: redukciós műveletek: collect (3)

- Az akkumulátor és az egyesítő is egy asszociatív, interferencia mentes és állapotmentes függvény kell, hogy legyen.

Streamek: redukciós műveletek: collect (4)

- A `collect()` műveletnek a következő két formája van:
 - **`collect(ellátó, akkumulátor, egyesítő)`**:
[https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/Stream.html#collect\(java.util.function.Supplier,java.util.function.BiConsumer,java.util.function.BiConsumer\)](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/Stream.html#collect(java.util.function.Supplier,java.util.function.BiConsumer,java.util.function.BiConsumer))
 - Az adott ellátó (`Supplier`), akkumulátor (`BiConsumer`) és egyesítő (`BiConsumer`) alkalmazásával végez módosítható redukciót.
 - **`collect(gyűjtő)`**:
[https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/Stream.html#collect\(java.util.stream.Collector\)](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/Stream.html#collect(java.util.stream.Collector))
 - Egy gyűjtő (`Collector`) alkalmazásával végez módosítható redukciót.
 - A `java.util.stream.Collectors` osztály sok gyakori módosítható redukció megvalósítását adja.
 - Lásd:
<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/Collectors.html>

Streamek: redukciós műveletek: collect (5)

- Példa:

```
List<String> list = IntStream.range(0, 10)
    .mapToObj(Integer::toString)
    .collect(ArrayList::new,           // ellátó
              ArrayList::add,         // akkumulátor
              ArrayList::addAll       // egyesítő
    );
```

mely röviden az alábbi módon írható:

```
List<String> list = IntStream.range(0, 10)
    .mapToObj(Integer::toString)
    .collect(Collectors.toList());
```

Streamek: redukciós műveletek: collect (6)

- Példa:

```
String s = Stream.of("banana", "kiwi", "pineapple", "mango")
    .collect(Collector.of(
        () -> new StringJoiner(", ", "[", "]"), // ellátó
        StringJoiner::add,                       // akkumulátor
        StringJoiner::merge,                     // egyesítő
        StringJoiner::toString                   // befejező
    )); // eredmény: "[banana, kiwi, pineapple, mango]"
```

mely röviden az alábbi módon írható:

```
String s = Stream.of("banana", "kiwi", "pineapple", "mango")
    .collect(Collectors.joining(", ", "[", "]"));
// eredmény: "[banana, kiwi, pineapple, mango]"
```

Streamek: redukciós műveletek: collect (7)

- Példa:

```
List<String> list = List.of("apple", "banana", "apple", "pear",  
    "apple", "apple", "banana");  
Map<String, Long> distribution = list.stream()  
    .collect(Collectors.groupingBy(Function.identity(),  
                                   Collectors.counting()))  
    );  
distribution.forEach(  
    (value, freq) -> System.out.printf("%s: %d\n", value, freq)  
);  
// banana: 2  
// apple: 4  
// pear: 1
```

Streamek: redukciós műveletek: összetettebb példák (1)

- Tekintsük az alábbi osztályt:

```
import java.time.Year;

public class LegoSet {
    String number;
    Year year;
    int pieces;

    public LegoSet(String number, Year year, int pieces) {
        this.number = number;
        this.year = year;
        this.pieces = pieces;
    }

    public String getNumber() { return number; }

    public Year getYear() { return year; }

    public int getPieces() { return pieces; }

    public String toString() { return number; }
}
```

Streamek: redukciós műveletek: összetettebb példák (2)

- Tegyük fel, hogy adott az alábbi lista:

```
List<LegoSet> legoSets = List.of(  
    new LegoSet("60073", Year.of(2015), 233), // Service Truck  
    new LegoSet("60080", Year.of(2015), 586), // Spaceport  
    new LegoSet("75211", Year.of(2018), 519), // Imperial TIE Fighter  
    new LegoSet("21034", Year.of(2017), 468) // London  
);
```

Streamek: redukciós műveletek: összetettebb példák (3)

- Példák:

```
// Az 500-nál több építőelemből álló nagy Lego készletek lekérdezése:  
List<LegoSet> bigLegoSets = legoSets.stream()  
    .filter(legoSet -> legoSet.getPieces() > 500)  
    .collect(Collectors.toList());  
System.out.println(bigLegoSets);  
// [60080, 75211]
```

Streamek: redukációs műveletek: összetettebb példák (4)

- Példák:

```
// Az építőelemek száma összesen:  
int totalPieces = legoSets.stream()  
    .collect(Collectors.summingInt(LegoSet::getPieces));  
System.out.println(totalPieces); // 1806
```

```
// Statisztikák az építőelemek számáról:  
IntSummaryStatistics piecesSummary = legoSets.stream()  
    .collect(Collectors.summarizingInt(LegoSet::getPieces));  
System.out.println(piecesSummary);  
// IntSummaryStatistics{count=4, sum=1806, min=233,  
// average=451,500000, max=586}
```

Streamek: redukciós műveletek: összetettebb példák (5)

- Példák:

```
// A Lego készletek csoportosítása évenként:  
Map<Year, List<LegoSet>> legoSetsByYear = legoSets  
    .stream()  
    .collect(Collectors.groupingBy(LegoSet::getYear));  
System.out.println(legoSetsByYear);  
// {2017=[21034], 2018=[75211], 2015=[60073, 60080]}
```

```
// A Lego készletek számának meghatározása évenként:  
Map<Year, Long> numberOfLegoSetsByYear = legoSets  
    .stream()  
    .collect(Collectors.groupingBy(LegoSet::getYear,  
                                   Collectors.counting()))  
    );  
System.out.println(numberOfLegoSetsByYear);  
// {2017=1, 2018=1, 2015=2}
```

Streamek: redukciós műveletek: összetettebb példák (6)

- Példa:

```
// A legtöbb építőelemből álló Lego készlet lekérdezése:  
legoSets.stream()  
    .collect(Collectors.maxBy(  
        Comparator.comparing(LegoSet::getPieces)))  
    .ifPresent(System.out::println);  
// 60080
```

Streamek: redukciós műveletek: összetettebb példák (7)

- Példák:

```
// A Lego készletek felosztása egy predikátum által:  
Map<Boolean, List<LegoSet>> map = legoSets.stream()  
    .collect(Collectors.partitioningBy(  
        legoSet -> legoSet.getPieces() > 500));  
System.out.println(map);  
// {false=[60073, 21034], true=[60080, 75211]}
```

```
// A Lego készletek felosztása egy predikátum által és a Lego  
// készletek számának meghatározása az egyes részeken:  
Map<Boolean, Long> map = legoSets.stream()  
    .collect(Collectors.partitioningBy(  
        legoSet -> legoSet.getPieces() > 500,  
        Collectors.counting()));  
// {false=2, true=2}
```

Streamek: párhuzamosság (1)

- Minden stream művelet végrehajtható szekvenciálisan és párhuzamosan is.
- A JDK stream megvalósításai szekvenciális streameket hoznak létre, hacsak nem kérünk kifejezetten párhuzamosságot.
 - Például a `java.util.Collection` interfész `stream()` és `parallelStream()` metódusai szekvenciális illetve párhuzamos stream-eket hoznak létre.
- A nemdeterminisztikusnak jelölt műveletek (mint például a `findAny()`) kivételével egy számítás eredményét nem befolyásolja az, hogy szekvenciálisan vagy párhuzamosan történik a stream végrehajtása.

Streamek: párhuzamosság (2)

- Az alábbi módszerek valamelyikével kapható egy párhuzamos stream:
 - Egy kollekció `parallelStream()` metódusának meghívásával.
 - Egy már létező szekvenciális stream `parallel()` metódusának meghívásával.

Streamek: párhuzamosság (3)

- A párhuzamos streamek egy megosztott szálkészletet használnak, mely a `ForkJoinPool.commonPool()` statikus metódushívásával kapható meg.
 - Lásd: `java.util.concurrent.ForkJoinPool`
<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/ForkJoinPool.html>
 - A készlet mérete a `java.util.concurrent.ForkJoinPool.common.parallelism` rendszertulajdonság révén szabályozható.
 - Az alábbi kódsor szemlélteti, hogyan kapható meg a készlet mérete:

```
System.out.println("Pool size: " +  
    ForkJoinPool.commonPool().getParallelism());  
// Pool size: 7
```

Streamek: párhuzamosság (4)

- Példa: egy szekvenciális stream

```
Stream.of("apple", "banana", "orange", "pear")
    .map(
        s -> {
            System.out.printf("map      %-6s %s\n", s,
                Thread.currentThread().getName());
            return s.toUpperCase();
        }
    ).forEach(
        s -> System.out.printf("forEach %-6s %s\n", s,
            Thread.currentThread().getName())
    );
// map      apple  main
// forEach APPLE  main
// map      banana main
// forEach BANANA main
// map      orange main
// forEach ORANGE main
// map      pear   main
// forEach PEAR   main
```

Streamek: párhuzamosság (5)

- Example: az előző példa párhuzamos verziója

```
Stream.of("apple", "banana", "orange", "pear")
    .parallel()
    .map(
        s -> {
            System.out.printf("map      %-6s %s\n", s,
                Thread.currentThread().getName());
            return s.toUpperCase();
        }
    ).forEach(
        s -> System.out.printf("forEach %-6s %s\n", s,
            Thread.currentThread().getName())
    );
// map      orange main
// map      apple  ForkJoinPool.commonPool-worker-7
// forEach APPLE  ForkJoinPool.commonPool-worker-7
// map      banana ForkJoinPool.commonPool-worker-3
// map      pear   ForkJoinPool.commonPool-worker-5
// forEach PEAR   ForkJoinPool.commonPool-worker-5
// forEach BANANA ForkJoinPool.commonPool-worker-3
// forEach ORANGE main
```

Streamek: párhuzamosság (6)

- Példa: gonosz számok
 - Egy gonosz szám egy olyan nemnegatív egész, melynek bináris alakja páros számú 1-est tartalmaz.
 - Lásd: https://en.wikipedia.org/wiki/Evil_number
 - Számoljuk meg a gonosz számokat egy szekvenciális és egy párhuzamos streammel és vessük össze a teljesítményüket!

Streamek: párhuzamosság (7)

- Példa: gonosz számok (folytatás)
 - A gonosz számok számolása szekvenciális streammel:

```
import java.util.concurrent.TimeUnit;
import java.util.stream.IntStream;

...
long startTime = System.nanoTime();
long count = IntStream.range(0, 1_000_000_000)
    .filter(n -> Integer.bitCount(n) % 2 == 0)
    .count();
long endTime = System.nanoTime();
System.out.printf("%d\nTime elapsed: %dms\n", count,
    TimeUnit.NANOSECONDS.toMillis(endTime - startTime));
// 500000000
// Time elapsed: 966ms
```

Streamek: párhuzamosság (8)

- Példa: gonosz számok (folytatás)
 - A gonosz számok számolása párhuzamos streammel:

```
import java.util.concurrent.TimeUnit;
import java.util.stream.IntStream;

...
long startTime = System.nanoTime();
long count = IntStream.range(0, 1_000_000_000)
    .parallel()
    .filter(n -> Integer.bitCount(n) % 2 == 0)
    .count();
long endTime = System.nanoTime();
System.out.printf("%d\nTime elapsed: %dms\n", count,
    TimeUnit.NANOSECONDS.toMillis(endTime - startTime));
// 5000000000
// Time elapsed: 313ms
```

Streamek: párhuzamosság (9)

- Példa: gonosz számok (folytatás)
 - A párhuzamos stream felülmúlja a szekvenciális számok számolásakor (**313ms / 966ms** = 0,32).
 - Számolás helyett gyűjtsük inkább össze a számokat egy listába!
 - Ekkor a szekvenciális stream nyeri a versenyt (**1737ms / 5549ms** = 0,31)!
 - A párhuzamos verzió végrehajtásához ráadásul a -Xmx4g parancssori opciót is meg kell adni!

Streamek: párhuzamosság (10)

- Példa: gonosz számok (folytatás)
 - A gonosz számok összegyűjtése egy listába szekvenciális streammel:

```
import java.util.concurrent.TimeUnit;
...
long startTime = System.nanoTime();
List<Integer> result = IntStream.range(0, 100_000_000)
    .filter(n -> Integer.bitCount(n) % 2 == 0)
    .boxed()
    .collect(Collectors.toList());
long endTime = System.nanoTime();
System.out.printf("%d\nTime elapsed: %dms\n", result.size(),
    TimeUnit.NANOSECONDS.toMillis(endTime - startTime));
// 5000000000
// Time elapsed: 1737ms
```

Streamek: párhuzamosság (11)

- Példa: gonosz számok (folytatás)
 - A gonosz számok összegyűjtése egy listába párhuzamos streammel:

```
import java.util.concurrent.TimeUnit;
...
long startTime = System.nanoTime();
List<Integer> result = IntStream.range(0, 100_000_000)
    .parallel()
    .filter(n -> Integer.bitCount(n) % 2 == 0)
    .boxed()
    .collect(Collectors.toList());
long endTime = System.nanoTime();
System.out.printf("%d\nTime elapsed: %dms\n", result.size(),
    TimeUnit.NANOSECONDS.toMillis(endTime - startTime));
// 5000000000
// Time elapsed: 5549ms
```

Streamek: párhuzamosság (12)

- Példa: gonosz számok (folytatás)
 - A gonosz számok számolása hagyományos for ciklussal (gyorsabb a szekvenciális streamnél):

```
long count = 0;
long startTime = System.nanoTime();
for (int i = 0; i < 1_000_000_000; i++) {
    if (Integer.bitCount(i) % 2 == 0) {
        count++;
    }
}
long endTime = System.nanoTime();
System.out.printf("%d\nTime elapsed: %dms\n", count,
    TimeUnit.NANOSECONDS.toMillis(endTime - startTime));
// 500000000
// Time elapsed: 799ms
```

Streamek: párhuzamosság (13)

- Példa: gonosz számok (folytatás)
 - A gonosz számok összegyűjtése egy listába hagyományos for ciklussal:

```
ArrayList<Integer> result = new ArrayList<>();
long startTime = System.nanoTime();
for (int i = 0; i < 100_000_000; i++) {
    if (Integer.bitCount(i) % 2 == 0) {
        result.add(i);
    }
}
long endTime = System.nanoTime();
System.out.printf("%d\nTime elapsed: %dms\n", result.size(),
    TimeUnit.NANOSECONDS.toMillis(endTime - startTime));
// 5000000000
// Time elapsed: 1515ms
```

Végtelen streamek (1)

- Példa: véletlen számok egy streamjének előállítása
 - A `java.util.Random` osztály statikus `doubles()` metódusa véletlen számok egy streamjét adja vissza.
 - Lásd:
[https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Random.html#doubles\(\)](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Random.html#doubles())

```
new Random().doubles()  
    .limit(5)  
    .forEach(System.out::println);  
// 0.6236419197176003  
// 0.12212206445812179  
// 0.355836701340987  
// 0.6088477599798222  
// 0.4282955926878853
```

Végtelen streamek (2)

- Példa: véletlen számok egy streamjének előállítása addig, amíg egy adott számot nem kapunk

```
new Random().ints(0, 100)
    .takeWhile(n -> n != 0)
    .forEach(System.out::println)
// 49
// 53
// 6
// 48
// 80
```

Végtelen streamek (3)

- Példa: a következő 5 gonosz szám előállítás a milliomodiktól kezdve

```
IntStream.iterate(0, n -> n + 1)
    .filter(n -> Integer.bitCount(n) % 2 == 0)
    .skip(1_000_000)
    .limit(5)
    .forEach(System.out::println);
// 2000001
// 2000002
// 2000004
// 2000007
// 2000008
```

Streamek: zárszó (1)

- Példa:
 - Forrás: Brian Goetz. *An introduction to the java.util.stream library*. May 9, 2016.
<https://developer.ibm.com/articles/j-java-streams-1-brian-goetz/>

Streams: zárshó (2)

```
Set<Seller> sellers = new HashSet<>();
for (Txn t : txns) {
    if (t.getBuyer().getAge() >= 65)
        sellers.add(t.getSeller());
}
List<Seller> sorted = new ArrayList<>(sellers);
Collections.sort(sorted, new Comparator<Seller>() {
    public int compare(Seller a, Seller b) {
        return a.getName().compareTo(b.getName());
    }
});
for (Seller s : sorted)
    System.out.println(s.getName());
```

```
txns.stream()
    .filter(t -> t.getBuyer().getAge() >= 65)
    .map(Txn::getSeller)
    .distinct()
    .sorted(comparing(Seller::getName))
    .map(Seller::getName)
    .forEach(System.out::println);
```

További ajánlott irodalom (1)

- Nem absztrakt interfész metódusok:
 - Brian Goetz. *Interface evolution via virtual extension methods*. June 2011.
<https://cr.openjdk.java.net/~briangoetz/lambda/Defender%20Methods%20v4.pdf>
 - *The Java Tutorials – Trail: Learning the Java Language – Lesson: Interfaces and Inheritance*.
<https://docs.oracle.com/javase/tutorial/java/land/>

További ajánlott irodalom (2)

- Streamek:
 - Brian Goetz. *Java Streams – Explore the `java.util.stream` library*. 2016.
<https://developer.ibm.com/series/java-streams/>
 - *The Java Tutorials – Trail: Learning the Java Language – Lesson: Aggregate Operations*.
<https://docs.oracle.com/javase/tutorial/collections/sstreams/>