class: inverse, center, middle

# Újdonságok Java 8-tól

---

## Tematika 1.

- Default és static interfész metódusok
- Bevezetés a lambda kifejezések használatába
- Saját és beépített funkcionális interfészek. Method reference.
- `Optional` osztály használata
- Streamek
- Források, közbülső és lezáró műveletek
- Primitívek használata streamekben
- Párhuzamosság streamek esetén
- Collectors

---

## Tematika 2.

- Új típusok: `LocalDate`, `LocalTime`, `LocalDateTime`, műveletek, parse és format. Átjárás régi típusok között.
- `Period` és `Duration`
- Időzónák használata
- Collections Framework módosítások
- Comparator
- Könyvtár és fájlkezelés
- Annotációk
- Párhuzamosság
- Egyéb apróságok

---

## Források 1.

- Cay S. Horstmann: Java SE 8 for the Really Impatient
- Jeanne Boyarsky, Scott Selikoff: OCA: Oracle Certified Associate Java SE 8 Programmer I Study Guide: Exam 1Z0-808
- Jeanne Boyarsky, Scott Selikoff: OCP: Oracle Certified Professional Java SE 8 Programmer II Study Guide: Exam 1Z0-809 1st Edition

---

## Források 2.

- http://www.oracle.com/technetwork/java/javase/8-whats-new-2157071.html

- https://docs.oracle.com/javase/9/whatsnew/toc.htm#JSNEW-GUID-C23AFD78-C777-460B-8ACE-58BE5EA681F6
- http://www.oracle.com/technetwork/java/javase/10-relnote-issues-4108729.html

## Default metódusok

- Cél a visszafele kompatibilitás: amennyiben bővítjük az interfészt, ne kelljen minden implementációt módosítani
- Definiál egy absztrakt metódust is, melyet az osztályok override-olhatnak
- Ha nincs override-olva, a default implementáció használt

---

## Default metódus szabályok

- Default metódus csak interfészen belül definiálható
- A `default` kulcsszóval kell ellátni, és kötelező implementálni
- Nem látható el `static`, `final`, vagy `abstract` módosítókkal
- Láthatósági módosítója `public`, melyet nem kötelező kitenni. Más láthatósági módosító nem használható.

---

## Default metódusok leszármazásnál és implementálásnál

- Leszármazott interfész dönthet:
- Öröklődik, nem definiálja felül
- Felüldeklarálhatja (override)
- Absztrakt metódusként definiálja felül

---

## Default metódusok és a többszörös öröklődés

- Leszármazáskor vagy implementáláskor névütközés
- Override nélkül nem fordul le
- Override-dal egyértelműsíthető

---

## Static interfész metódusok

- Hasonló az osztályban definiált static metódushoz
- Különbség: nem öröklődik
- Csak az interfész nevével minősíthető

## Funkcionális programozás

- Java objektumorientált programozási nyelv
- Elemi egység az objektum
- Funkcionális programozás
  - Új programozási módszertan

- Elemi egység a függvény, ezért pl. paraméterként átadható
- Deklaratív: nem a lépéseket adjuk meg
- Nincs állapot
- Program függvények hívása és kiértékelése

---

## Funkcionális programozás Javaban

- Lambda kifejezés olyan kódblokk, mely paraméterként átadható
- Felfogható névtelen metódusnak is
- Erőssége főleg a kollekcióknál nyilvánul meg
- Párhuzamos feldolgozás erősen támogatott

---

## `Comparator` használata anonymous inner class-szal

```
trainers.sort(new Comparator<Trainer>() {
    @Override
    public int compare(Trainer trainer1, Trainer trainer2) {
        return trainer1.getName().compareTo(trainer2.getName());
    }
});
```

---

## Strategy tervezési minta

- Algoritmus részét kintről lehet megadni
- Valójában csak egy működést, egy utasítást kellene paraméterként átadni
- Objektumorientált overhead: hozzá tartozó osztály és példány
- Egyszerűsítés: anonymous inner class
- További egyszerűsítés: csak a metódustörzs átadása, lambda kifejezés

---

## `Comparator` használata lambda kifejezéssel

```
trainers.sort((trainer1, trainer2) ->
trainer1.getName().compareTo(trainer2.getName()));
```

- `trainer1` és `trainer2` paraméterrel kell meghívni egy metódust, mely visszatérési értéke a kifejezés értéke

---

## Lambda kifejezés szintaktika

- Paraméterek
  - Amennyiben egy paraméter áll, nem kötelező a zárójel
  - Több paraméter esetén kötelező a zárójel
  - Megadható a típus

- Nyíl operátor
- Törzs
  - szabványos blokk, több utasítás, utasítások pontosvesszővel lezárva, `return` utasítás
  - egy utasítás esetén a zárójel, `return` utasítás és a pontosvessző elhagyható

---

## Funkcionális interfész

- Lambda kifejezések funkcionális interfészekkel működnek
- Csak egy metódust tartalmazhatnak
- `@FunctionalInterface` annotáció, több metódus esetén nem fordul le
- `Comparator` pl. funkcionális interfész

```
trainers.sort((trainer1, trainer2) ->
trainer1.getName().compareTo(trainer2.getName()));
```

---

## Method reference

- Amennyiben a lambda kifejezés csak egy metódust hív
- Négy formája
  - Statikus metódushívás
  - Példány metódusának hívása
  - Metódus hívása paraméteren (első paraméter a példány, a többi paraméter a hívás paraméterei)
  - Konstruktor hívása

```
trainers.sort(Trainer::compareTraniersByName);
```

---

## Keresés

```
public Trainer findFirst(List<Trainer> trainers, String name) {
    for (Trainer trainer: trainers) {
        if (trainer.getName().equals(name)) {
            return trainer;
        }
    }
    throw new IllegalArgumentException("Cannot find trainer with name: " +
name);
}
```

---

## Strategy tervezési minta

- Keresés paraméterezhető

```
public Trainer findFirst(List<Trainer> trainers, Condition<Trainer>
condition) {
    for (Trainer trainer: trainers) {
        if (condition.apply(trainer)) {
            return trainer;
        }
    }
    throw new IllegalArgumentException("Cannot find trainer applied to the
condition");
}
```

---

## Saját funkcionális interfész

```
public interface Condition<T> {

    boolean apply(T t);
}
```

Használata:

```
findFirst(trainers, trainer -> trainer.getName().equals("John Doe"));
```

---

## Már létező beépített interfész

- `java.util.function.Predicate`, `boolean test(T t)` metódussal

```
public Trainer findFirst(List<Trainer> trainers, Predicate<Trainer>
condition) {
    for (Trainer trainer: trainers) {
        if (condition.test(trainer)) {
            return trainer;
        }
    }
    throw new IllegalArgumentException("Cannot find trainer applied to the
condition");
}
```

---

## Beépített interfészek 1.

- `Supplier<T>`: `get()` metódus visszatérési típusa `T`
- `Consumer<T>`: `accept(T t)` metódus visszatérési típusa `void`
- `BiConsumer<T>`: `accept(T t, U u)` metódus visszatérési típusa `void`
- `Predicate<T>`: `test(T t)` metódus visszatérési típusa `boolean`
- `BiPredicate<T>`: `test(T t, U u)` metódus visszatérési típusa `boolean`

---

## Beépített interfészek 2.

- `Function<T, R>`: `apply(T t)` metódus visszatérési típusa `R`
- `BiFunction<T, U, R>`: `apply(T t, U u)` metódus visszatérési típusa `R`
- `UnaryOperator<T>`: `apply(T t)` metódus visszatérési típusa `T`
- `BinaryOperator<T>`: `apply(T t1, T t2)` metódus visszatérési típusa `T`

## `Optional` osztály szükségessége

- Ha az algoritmus nem ad vissza értelmezhető eredményt speciális bemenő adatokra (ismeretlen, nincs adat, stb.)
- Gyakran speciális értéket adunk vissza, vagy `null` értéket
- Rossz gyakorlat a `NullPointerException` miatt
- Kivételkezelés sem megfelelő, hiszen nem feltétlen kivételes eset

---

## `Optional` osztály

- Burkoló osztály, érték nélkül vagy értékkel (üres, vagy értéket tartalmazó doboz)
- `Optional.empty()` vagy `Optional.of(T)` factory metódusok
- `Optional.ofNullable(T)` factory metódus
- Generikussal

```
public static Optional<Double> average(int... scores) {
    if (scores.length == 0) return Optional.empty();
    int sum = 0;
    for (int score: scores) {
        sum += score;
    }
    return Optional.of((double) sum / scores.length);
}
```

---

## Érték lekérése

```
Optional<Double> opt = average(90, 100);
if (opt.isPresent()) {
    Double d = opt.get();
}
```

Ha nem tartalmaz értéket: `java.util.NoSuchElementException: No value present`

---

## `Optional` metódusai

- `ifPresent(Consumer c)`
- `orElse(T other)`
- `orElseGet(Supplier s)`
- `orElseThrow(Supplier s)`

```
avarage(90, 10)
    .ifPresent(System.out::println);

double d = avarage(90, 10)
    .orElse(Double.NaN);
```

`Comparator.comparing`

```
people.sort(Comparator.comparing(Person::getName));

people.sort(Comparator.comparing(Person::getLastName)
    .thenComparing(Person::getFirstName));

people.sort(Comparator.comparing(Person::getName,
  (s, t) -> s.toLowerCase().compareTo(t.trim().toLowerCase())));
```

`int`, `long` és `double` esetén

```
people.sort(Comparator.comparingInt(p -> p.getName().length()));
```

---

## `null`, natural order és fordított rendezés

```
people.sort(comparing(Person::getMiddleName,
  nullsFirst(naturalOrder())));

people.sort(comparing(Person::getMiddleName,
  reverseOrder()));

people.sort(comparing(Person::getName,
  Comparator.comparingInt(String::length)).reversed());
```

## Streamek

- A stream adatok egymásutánja, mint egy cső vagy futószalag
- Véges és végtelen streamek
- Az éppen feldolgozás alatt lévő elem férhető hozzá, lehet, hogy a többi még nem is állt elő
- Részei
  - Forrás (source)
  - Közbülső műveletek (intermediate operations)
  - Záró műveletek (terminal operations)

---

## Források

- `Stream<String> empty = Stream.empty();`
- `Stream<Integer> singleElement = Stream.of(1);`
- `Stream<Integer> fromArray = Stream.of(1, 2, 3);`

```
List<String> list = Arrays.asList("a", "b", "c");
Stream<String> fromList = list.stream();
Stream<String> fromListParallel = list.parallelStream();

Stream<Double> randoms = Stream.generate(Math::random);
Stream<Integer> oddNumbers = Stream.iterate(1, n -> n + 2);
```

---

## Záró műveletek

- `count()`
- `min()` és `max()`
- `findAny()` és `findFirst()`
- `allMatch()`, `anyMatch()`, `noneMatch()`
- `forEach()`
- `reduce()`
- `collect()`

---

### `count()`

```
Stream<String> s = Stream.of("monkey", "gorilla", "bonobo");
System.out.println(s.count()); // 3
```

---

### `min()` és `max()`

```
Stream<String> s = Stream.of("monkey", "ape", "bonobo");
Optional<String> min = s.min((s1, s2) -> s1.length() – s2.length());
min.ifPresent(System.out::println); // ape
```

- Üres stream esetén `Optional.empty()`

---

### `findAny()` és `findFirst()`

```
Stream<String> s = Stream.of("monkey", "gorilla", "bonobo");
Stream<String> infinite = Stream.generate(() -> "chimp");
s.findAny().ifPresent(System.out::println); // monkey
infinite.findAny().ifPresent(System.out::println); // chimp
```

- Üres stream esetén `Optional.empty()`

---

### `allMatch()`, `anyMatch()`, `noneMatch()`

```
List<String> list = Arrays.asList("monkey", "2", "chimp");
Stream<String> infinite = Stream.generate(() -> "chimp");
Predicate<String> pred = x -> Character.isLetter(x.charAt(0));
System.out.println(list.stream().anyMatch(pred)); // true
```

```
System.out.println(list.stream().allMatch(pred)); // false
System.out.println(list.stream().noneMatch(pred)); // false
System.out.println(infinite.anyMatch(pred)); // true
```

---

`forEach()`

```
Stream<String> s = Stream.of("Monkey", "Gorilla", "Bonobo");
s.forEach(System.out::print); // MonkeyGorillaBonobo
```

`reduce()`

```
BinaryOperator<Integer> op = (a, b) -> a * b;
Stream<Integer> stream = Stream.of(3, 5, 6);
System.out.println(stream.reduce(1, op, op)); // 90
```

- *identity*, *accumulator*, *combiner*
- Mindig új objektumot hoz létre

---

## Három paraméteres `collect()` metódus

- Ugyanazt a módosítható objektumot módosítja
- *supplier*, *accumulator*, *combiner*

```
Stream<String> stream = Stream.of("w", "o", "l", "f");
Set<String> set = stream.collect(TreeSet::new, TreeSet::add,
    TreeSet::addAll); // TreeSet példányt ad vissza
System.out.println(set); // [f, l, o, w]
```

---

## Egy paraméteres `collect()` metódus

- `Collector` interfész, definiál un. *mutable reduction operatort*

```
Stream<String> stream = Stream.of("w", "o", "l", "f");
Set<String> set = stream.collect(Collectors.toCollection(TreeSet::new));
// TreeSet példányt ad vissza
System.out.println(set); // [f, l, o, w]
```

```
Stream<String> stream = Stream.of("w", "o", "l", "f");
Set<String> set = stream.collect(Collectors.toSet());
System.out.println(set); // [f, w, l, o]
```

## Közbülső műveletek

- `filter()`
- `distinct()`
- `limit()` és `skip()`
- `map()`
- `flatMap()`
- `sorted()`

- peek()

---

**filter()**

```
Stream<String> s = Stream.of("monkey", "gorilla", "bonobo");
s.filter(x -> x.startsWith("m")).forEach(System.out::print); //monkey
```

---

**distinct()**

```
Stream<String> s = Stream.of("duck", "duck", "duck", "goose");
s.distinct().forEach(System.out::print); // duckgoose
```

---

**limit() és skip()**

```
Stream<Integer> s = Stream.iterate(1, n -> n + 1);
s.skip(5).limit(2).forEach(System.out::print); // 67
```

---

**map()**

- Egy-egy megfeleltetés

```
Stream<String> s = Stream.of("monkey", "gorilla", "bonobo");
s.map(String::length).forEach(System.out::print); // 676
```

---

**flatMap()**

```
List<String> zero = Arrays.asList();
List<String> one = Arrays.asList("Bonobo");
List<String> two = Arrays.asList("Mama Gorilla", "Baby Gorilla");
Stream<List<String>> animals = Stream.of(zero, one, two);
animals.flatMap(l -> l.stream()).forEach(System.out::println);
// Bonobo
// Mama Gorilla
// Baby Gorilla
```

- Stream elemeihez stream rendelhető, és ezeket kombinálja
- Egy elemhez vagy nem tartozik elem, vagy egy vagy több is tartozhat

---

**sorted()**

```
Stream<String> s = Stream.of("brown-", "bear-");
s.sorted().forEach(System.out::print); // bear-brown-

Stream<String> s = Stream.of("brown bear-", "grizzly-");
s.sorted(Comparator.reverseOrder())
.forEach(System.out::print); // grizzly-brown bear-
```

---

### peek()

```
Stream<String> stream = Stream.of("black bear", "brown bear", "grizzly");
long count = stream.filter(s -> s.startsWith("g"))
.peek(System.out::println).count(); // grizzly
System.out.println(count); // 1
```

- Debug célokra

---

## Lazy kiértékelés

- A közbülső műveletek nem hajtódnak végre, míg az lezáró művelet nem hajtódik végre

## Primitív streamek

- Megvalósítható csomagoló osztályokkal és klasszikus streamekkel
- Gyakori műveletek egyszerűbben elvégezhetőek

```
IntStream.of(1, 2, 3).average().getAsDouble(); // 2.0
```

---

## Primitív streamek fajtái

- `IntStream`
- `LongStream`
- `DoubleStream`

---

## Források

```
DoubleStream empty = DoubleStream.empty();
DoubleStream varargs = DoubleStream.of(1.0, 1.1, 1.2);
DoubleStream random = DoubleStream.generate(Math::random);
DoubleStream fractions = DoubleStream.iterate(.5, d -> d / 2);

IntStream integers = IntStream.range(1, 6);
IntStream rangeClosed = IntStream.rangeClosed(1, 5);
```

---

## Streamek közötti átjárás

- `map`, `mapToObj`, `mapToInt`, `mapToLong` és `mapToDouble` metódusok
- Mögöttük saját funkcionális interfészek, hiszen a szabvány interfészeket nem lehet primitív típusokkal paraméterezni

```
Stream<String> objStream = Stream.of("penguin", "fish");
IntStream intStream = objStream.mapToInt(s -> s.length());
```

---

## További interfészek és osztályok

- Pl. a `mapToInt` metódus paramétere `ToIntFunction` funkcionális interfész
- Sok funkcionális interfésznek van primitív típussal rendelkező párja, pl `BooleanSupplier`, `DoublePredicate`, `ToDoubleBiFunction`, stb.
- Ugyanígy az `Optional` sem paraméterezhető, ezért van `OptionalInt`, `OptionalLong` és `OptionalDouble`

---

## Max, min, átlag, stb. gyűjtése

- `IntSummaryStatistics` osztály
  - Darabszám, minimum és maximum érték, összeg és átlag

```
IntStream integers = IntStream.range(1, 6);
IntSummaryStatistics stats = ints.summaryStatistics();
int max = stats.getMax(); // 6
int min = stats.getMin(); // 1
```

## Collectorok 1.

- avaraging
- counting
- `groupingBy`
- `joining`
- `maxBy` és `minBy`
- `mapping`
- `partitioningBy`

---

## Collectorok 2.

- summarizing
- summing
- `toList()`, `toSet()`
- `toCollection()`
- `toMap()`

---

## Alap collectorok

- String (`joining`)
- Matematikai (avaraging, counting, max, min, summarizing, summing)

```
Stream<String> ohMy = Stream.of("lions", "tigers", "bears");
String result = ohMy.collect(Collectors.joining(", "));
System.out.println(result); // lions, tigers, bears
```

```
Stream<String> ohMy = Stream.of("lions", "tigers", "bears");
Double result = ohMy.collect(Collectors
```

```
  .averagingInt(String::length));
System.out.println(result); // 5.333333333333333
```

---

## Collection collectorok

- Kollekció (`toList`, `toSet`, `toCollection` - kollekció példányosítása adandó át paraméterként)

```
Stream<String> ohMy = Stream.of("lions", "tigers", "bears");
Set<String> result = ohMy.filter(s -> s.startsWith("t")
    .collect(Collectors.toCollection(TreeSet::new));
// TreeSet példányt ad vissza
System.out.println(result); // [tigers]
```

---

### `Collectors.toMap()`

- keyMapper
- valueMapper
- mergeFunction (opcionális)
- mapSupplier (opcionális)

```
Stream<String> ohMy = Stream.of("lions", "tigers", "bears");
Map<Integer, String> map = ohMy.collect(Collectors.toMap(
  String::length,
  v -> v,
  (s1, s2) -> s1 + "," + s2,
  TreeMap::new));
// TreeMap példányt ad vissza
System.out.println(map); // // {5=lions,bears, 6=tigers}
System.out.println(map.getClass()); // class. java.util.TreeMap
```

---

### `Collectors.groupingBy()`

```
Stream<String> ohMy = Stream.of("lions", "tigers", "bears");
Map<Integer, List<String>> map = ohMy.collect(
    Collectors.groupingBy(String::length));
System.out.println(map); // {5=[lions, bears], 6=[tigers]}
```

---

### `List` helyett `Set`

```
Stream<String> ohMy = Stream.of("lions", "tigers", "bears");
Map<Integer, Set<String>> map = ohMy.collect(
    Collectors.groupingBy(String::length, Collectors.toSet()));
System.out.println(map); // {5=[lions, bears], 6=[tigers]}
```

---

## Más collector

```
Stream<String> ohMy = Stream.of("lions", "tigers", "bears");
Map<Integer, Long> map = ohMy.collect(Collectors.groupingBy(
    String::length, Collectors.counting()));
System.out.println(map); // {5=2, 6=1}
```

---

## Visszatérési típus módosítása

```
Stream<String> ohMy = Stream.of("lions", "tigers", "bears");
Map<Integer, Set<String>> map = ohMy.collect(
    Collectors.groupingBy(String::length, TreeMap::new, Collectors.toSet()));
// TreeMap példányt ad vissza
System.out.println(map); // {5=[lions, bears], 6=[tigers]}
```

---

## Partícionálás

- Csak két csoportra bontás: `true` és `false`

```
Stream<String> ohMy = Stream.of("lions", "tigers", "bears");
Map<Boolean, List<String>> map = ohMy.collect(
    Collectors.partitioningBy(s -> s.length() <= 5));
System.out.println(map); // {false=[tigers], true=[lions, bears]}
```

---

## `List` helyett `Set`

```
Stream<String> ohMy = Stream.of("lions", "tigers", "bears");
Map<Boolean, Set<String>> map = ohMy.collect(
    Collectors.partitioningBy(s -> s.length() <= 7, Collectors.toSet()));
System.out.println(map);// {false=[], true=[lions, tigers, bears]}
```

---

## Mapping

```
Stream<String> ohMy = Stream.of("lions", "tigers", "bears");
Map<Integer, Optional<Character>> map = ohMy.collect(
    Collectors.groupingBy(
        String::length,
        Collectors.mapping(s -> s.charAt(0),
            Collectors.minBy(Comparator.naturalOrder()))));
System.out.println(map); // {5=Optional[b], 6=Optional[t]}
```

## Párhuzamos stream

- Műveletek futtatása párhuzamos szálakon
- Sebesség
- Módosulhat az eredmény is (tipikusan pl. a sorrend)

- Alapértelmezetten függ a CPU (magok) számától
- Nagyobb elemszám esetén, ugyanis van overhead

---

## Párhuzamos stream létrehozása

```
List<Integer> l = Arrays.asList(1,2,3,4,5,6);
Stream<Integer> parallelStream = l.stream().parallel();

Stream<Integer> parallelStream2 = l.parallelStream();

parallelStream
  .forEach(s -> System.out.print(s+" "));
```

- `Stream.isParallel()` metódussal lehet lekérdezni
- `Stream.sequential()` metódussal egyszálúsítható

---

## Párhuzamosság megtartása

- A műveleteknek egyszálú és párhuzamos streameknél is ugyanazt az eredményt kell hozniuk, kivéve, ha explicit módon nemdeterminisztikus, pl. a `findAny()`
- Bizonyos műveletek elveszítik a párhuzamosságot (pl. `flatMap`)
- Bizonyos műveletek megtartják (pl. `Stream.concat(Stream s1, Stream s2)`, ha valamelyik párhuzamos)

---

## Külső változó módosítása

- Szinkronizációs problémák
- Szinkronizálva elvesztjük a performancia előnyt
- Mellékhatás

```
List<Integer> l = new ArrayList<>();
IntStream.range(0, 100).parallel().forEach(l::add);
// null elemek, sőt, IndexOutOfBoundsException
System.out.println(l);
```

- Érdemes csak a paraméterekkel dolgozni, vagy kizárólag olvasni

---

## Sorrend

- Sorrendezettség függ a forrástól és a közbülső műveletektől
- Sorrendezett forrás, pl. a `List`, nem sorrendezett pl. a `HashSet`
- Közbülső művelet sorrendezhet, pl. a `sorted()`
- Közbülső művelet elvesztheti a sorrendezettséget, pl. `unordered()`
- Bizonyos záró műveletek figyelmen kívül hagyhatják a sorrendet, pl `forEach()`, helyette `forEachOrdered()`

## Sorrend elvesztése

- Sorrend elvesztése nemdeterminisztikus végeredménnyel jár
- Sorrend elvesztése csak párhuzamos streameknél járhat teljesítménynövekedéssel
- Bizonyos műveletek sorrendezett párhuzamos streameknél teljesítménycsökkenéssel járhatnak (pl. `skip()`, `limit()`, `findFirst()` metódusok, plusz műveletek a szálak együttműködésére)
- Szinkronizálják a szálakat, lassabb lesz, de konzisztens eredményt ad
- `findAny()` nem ad konzisztens eredményt, cserébe gyorsabb

---

## `unordered()` metódus

- `skip()`, `limit()`, `findFirst()` metódusoknál
- Ezeknél a `unordered()` metódus teljesítménynövekedést okozhat, ha nem számít a sorrend

---

`reduce()`

```
BinaryOperator<Integer> op = (a, b) -> a * b;
Stream<Integer> stream = Stream.of(3, 5, 6);
System.out.println(stream.reduce(1, op, op)); // 90
```

- *identity*, *accumulator*, *combiner*
- Mindig új objektumot hoz létre

---

## `reduce()` metódus párhuzamos környezetben

Meg kell felelni a következő szabályoknak:

- `combiner.apply(identity, u)` értéke `u`
- accumulator asszociatív és állapotmentes: `(a op b) op c` egyenlő `a ob (b op c)` értékével
- combiner asszociatív és állapotmentes, és az identity-vel kompatibilis: `combiner.apply(u, accumulator.apply(identity, t))` egyenlő `accumulator.apply(u, t)` értékével

---

## `collect()` metódus párhuzamos környezetben

- Ugyanazok a feltételek, mint a `reduce()` metódusnál

---

## Egy paraméteres `collect()` metódus

- Mutable reduce operator

- `Collector.characteristics()` metódus, `Collector.Characteristics` enum

---

## Collector párhuzamos működés

Csak a következő feltételek együttes teljesülése esetén tud hatékonyan (párhuzamosan) működni:

- Stream párhuzamos
- A collector rendelkezik a `Collector.Characteristics.CONCURRENT` jellemzővel
- A stream nem sorrendezett, vagy a collector rendelkezik a `Collector.Characteristics.UNORDERED` jellemzővel

---

## Megfelelő collectorok

- A `Collectors.toSet()` nem rendelkezik a `CONCURRENT` jellemzővel
- Használhatóak a `Collectors.toConcurrentMap()` vagy `Collectors.groupingByConcurrent()` metódusok

### `LocalDate` és `LocalTime`

- `java.time` csomagban
- `LocalDate` csak dátumot tartalmaz idő nélkül
- `LocalTime` csak időt tartalmaz
- `LocalDateTime` időt és dátumot is tartalmaz
- Nem tartalmaznak időzónát
  - `ZonedDateTime`
- Immutable

---

## Használatuk

- `System.out.println(LocalDateTime.now());` eredménye `2017-04-05T10:27:36.986`
- Konkrét megadás `of()` metódusok használatával
- Nem lenient, kivételt dob, `DateTimeException`

```
LocalDate date = LocalDate.of(2015, Month.JANUARY, 20);
LocalDate date = LocalDate.of(2015, 1, 20);
```

---

## Műveletek

- `plusXxx()` és `minusXxx()` metódusokkal
- Láncolt hívások
- `DayOfWeek` és `Month` enumok
- Összehasonlítás az `isBefore` és `isAfter` metódusokkal

```
LocalDate date = LocalDate.of(2014, Month.JANUARY, 20);
System.out.println(date); // 2014-01-20
date = date.plusDays(2);
System.out.println(date); // 2014-01-22
date = date.plusWeeks(1);
System.out.println(date); // 2014-01-29

LocalDate.of(2014, Month.JANUARY, 20)
  .plusDays(2)
  .plusWeeks(1);
```

---

## Átjárás a típusok között

```
LocalDateTime localDateTime = LocalDateTime.now();
LocalDate localDate = localDateTime.toLocalDate();
LocalTime localTime = localDateTime.toLocalTime();
LocalDateTime newLocalDateTime = LocalDateTime.of(localDate, localTime);
```

---

## Átjárás a régi típus között

```
Date in = new Date();
LocalDateTime ldt = LocalDateTime.ofInstant(in.toInstant(),
ZoneId.systemDefault());
Date out = Date.from(ldt.atZone(ZoneId.systemDefault()).toInstant());
```

---

## Formázás és parse-olás

- `DateTimeFormatter`
- Konstanssal: `DateTimeFormatter.ISO_LOCAL_DATE`
- Lokalizált stílussal: `DateTimeFormatter.ofLocalizedDate(FormatStyle.SHORT)`
- Formátum stringgel: `DateTimeFormatter.ofPattern("MMMM dd, yyyy, hh:mm")`
- Default locale / `withLocale(Locale)`
- Használata formatter vagy a `LocalDateTime` felől
- Immutable és szálbiztos

---

## Formázás és parse-olás példa

```
DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy.MM.dd.
HH:mm");
LocalDateTime now = LocalDateTime.of(2017, Month.JANUARY, 1, 12, 0);
System.out.println(formatter.format(now));
System.out.println(now.format(formatter));
```

```
LocalDateTime start = LocalDateTime.parse("2017.01.01. 12:00", formatter);
System.out.println(start);
```

---

### `ChronoUnit`

- Enum, `YEARS`, `MONTHS`, `DAYS`, `HOURS` értékkel
- Különbség: `ChronoUnit.DAYS.between(date1, date2);`

### `Instant` osztály

- Egy pillanat az idővonalon UTC időzóna szerint
- Létrehozása `Instant.now()` metódussal
- Kiírás ISO-8601 formátumban
- Nincs naptár társítva
  - Nem működnek a műveletek hónapokkal és évekkel
- Összehasonlítható

---

### `Instant` osztály példák

```
Instant now = Instant.now(); // 2019-07-10T16:06:22.944565200Z

Instant afterNow = now.plus(1, ChronoUnit.HOURS);
afterNow.isAfter(now); // true

now.plus(1, ChronoUnit.MONTHS); // UnsupportedTemporalTypeException:
Unsupported unit: Months
```

---

## Összehasonlítás más osztályokkal

- `Instant`
  - Mindig UTC-ben definiálja a pillanatot, nincs naptár
- `LocalDateTime` osztály: dátum és idő
  - Nem tartalmaz időzónát, ISO-8601 naptár van hozzá társítva
  - Különböző pillanatokat reprezentálhat időzónánként
- `ZonedDateTime` tartalmaz időzónát is
  - Időzónát tartalmaz, ISO-8601 naptár van hozzá társítva
  - Megmondja, hogy melyik időzónában meghatározott a pillanat

---

## Konvertálások

```
// Cél időzóna meghatározása, majd időzóna információ elhagyása
LocalDateTime dateTime = LocalDateTime.ofInstant(now,
ZoneId.systemDefault());
```

```
// Átváltás ZonedDateTime példánnyá, időzóna megadásával, majd Instant példánnyá konvertálás
Instant instant = dateTime.atZone(ZoneId.systemDefault()).toInstant();
```

## `Period` osztály létrehozása

- Dátum szintű hossz (pl. 2 év, 6 hónap és 2 nap)
- `Period annually = Period.ofYears(1);`
- `Period everyYearAndAWeek = Period.of(1, 0, 7);`
- Két dátum különbségeként is létrehozható
- `Period period = Period.between(localDate1, localDate2);`

---

## `Period` osztály használata

- `parse()` metódussal, `toString()` eredménye
- `P2Y`, `P3M`, `P4W`, `P5D`, `P1Y2M3D`
- Normalizálás (`normalized()` metódus) 12 hónapot évvé vált
- Immutable és szálbiztos

---

## Műveletek dátumokon

- `LocalDate` és `LocalDateTime` `plus()` és `minus()` metódusaival
  - `LocalDateTime.now().plus(Period.ofYears(1))`
- `Period` `addTo()` és `subtractFrom()` metódusaival
  - ```
    LocalDateTime localDateTime = (LocalDateTime)
    Period.ofYears(1).addTo(LocalDateTime.now());
    ```

---

## Műveletek `Period` példányon

- `plusXxx` és `minusXxx` metódusok
- `plus()` metódussal másik `Period` példány adható hozzá

---

## `Duration` osztály létrehozása

- Idő szintű hossz (pl. 2 óra és 30 perc)
- `Duration duration = Duration.ofHours(3);`
- Két dátum különbségeként is létrehozható
- `Duration duration = Duration.between(localDateTime1, localDateTime2);`

---

## `Duration` osztály használata

- `parse()` metódussal, `toString()` eredménye
- `PT20.345S`, `PT15M`, `P2D`
    ```

- Mindig normalizálva tárolva
- `System.out.println(Duration.parse("PT125M")); // PT2H5M`
- Immutable és szálbiztos

---

## Műveletek dátumokon

- `LocalDate`, `LocalTime` és `LocalDateTime` `plus()` és `minus()` metódusaival
  - `LocalDateTime.now().add(Duration.parse("PT2M"))`
- `Duration` `addTo()` és `subtractFrom()` metódusaival
  - `Duration.parse("PT2M").addTo(LocalDateTime.now());`
  - `(LocalDateTime)` cast

---

## Műveletek `Duration` példányokon

- `plusXxx` és `minusXxx` metódusok
- `plus()` metódussal másik `Duration` példány adható hozzá

## Időzónák

- `ZoneId` osztály
- Rendelkezésre álló időzónák: `ZoneId.getAvailableZoneIds()`
- Rendszer időzónája: `ZoneId.systemDefault()`, pl.: `Europe/Prague`
- Létrehozás: `ZoneId zoneId = ZoneId.of("Europe/Budapest");`

---

### `ZonedDateTime` osztály

- Dátum, idő és időzóna
- `ZonedDateTime.of(LocalDateTime, ZoneId)` metódus
- `toString()`, pl.: `2017-01-02T12:00+01:00[Europe/Prague]`
- Átváltás másik időzónába

```
ZonedDateTime localZonedDateTime =
  ZonedDateTime.now();
// 2017-04-05T13:46:26.372+02:00[Europe/Prague]
ZonedDateTime utcZonedDateTime =
  zonedDateTime.withZoneSameInstant(ZoneId.of("UTC"));
// 2017-04-05T11:46:26.372Z[UTC]
```

---

## Nyári időszámítás

- Nyári időszámítás, daylight saving, DST
- Helyi időt egy órával későbbre állítják az adott időzóna idejéhez képest (2017-ben március 26-án 2:00-kor 3:00-ra)

---

## Nyári időszámítás példa

```
ZonedDateTime zonedDateTime =
  ZonedDateTime.of(LocalDateTime.of(2017, Month.MARCH, 26, 1, 59),
    ZoneId.of("Europe/Budapest"));
  // 2017-03-26T01:59+01:00[Europe/Budapest]
zonedDateTime =
  zonedDateTime.plus(Duration.ofMinutes(1));
  // 2017-03-26T03:00+02:00[Europe/Budapest]

ZonedDateTime start =
  ZonedDateTime.of(LocalDateTime.of(2017, Month.MARCH, 26, 1, 00),
    ZoneId.of("Europe/Budapest"));
  // 2017-03-26T01:00+01:00[Europe/Budapest]
ZonedDateTime end =
  ZonedDateTime.of(LocalDateTime.of(2017, Month.MARCH, 26, 6, 00),
    ZoneId.of("Europe/Budapest"));
  // 2017-03-26T06:00+02:00[Europe/Budapest]
Duration duration = Duration.between(start, end); // PT4H
```

## `Clock` osztály

- Meg lehet adni egy órát, mely alapján kerülnek létrehozásra a dátum és idő típusú példányok
- Használható pl. teszteléshez

```
Clock clock = Clock.fixed(
        LocalDateTime.of(2019, 1, 1, 0, 0,
0).atZone(ZoneId.systemDefault()).toInstant(),
        ZoneId.systemDefault());

LocalDateTime nowWithClock = LocalDateTime.now(clock); // mindig 2019-01-
01T00:00
```

---

## Különböző órák

- `Clock.fixed()`: rögzített óra, mindig ugyanazt az időt mutatja
- `Clock.offset()`: az aktuális időhöz képest mindig egy fix eltolással mutatja az időt (elállított óra)
- `Clock.systemXXX()`: rendszeróra
- `Clock.tickXXX()`: olyan óra, mely csak a megadott időközönként lép előre, pl. csak percenként, addig ugyanazt az időt mutatja

## `Collection.removeIf` metódus

```
boolean removeIf(Predicate<? super E> filter)
```

```
List<String> names = new ArrayList<>(Arrays.asList("John", "Harry"));
names.removeIf(n -> n.startsWith("J"));
```

---

### `List.replaceAll` metódus

```
void replaceAll(UnaryOperator<E> o)
```

```
List<Integer> list = Arrays.asList(1, 2, 3);
list.replaceAll(x -> x * 2);
System.out.println(list); // [2, 4, 6]
```

---

### `List.sort` metódus

```
default void sort(Comparator<? super E> c)
```

---

### `Collection.forEach` metódus

```
List<String> cats = Arrays.asList("Annie", "Ripley");
cats.forEach(System.out::println);
```

---

### `Iterator.forEachRemaining`

```
Iterator<String> i = Arrays.asList("Annie", "Ripley").iterator();
i.forEachRemaining(System.out::println);
```

---

### `Map.putIfAbsent` metódus

```
Map<String, String> favorites = new HashMap<>();
favorites.put("Tom", "Tram");
favorites.put("Jenny", "Tram");
favorites.putIfAbsent("Tom", "Bus Tour"); // {Tom=Tram, Jenny=Tram}
```

---

### `Map.merge` metódus

- Ha nincs benne az adott kulccsal érték (vagy `null`), beleteszi, ha benne van, összefésüli

```
Map<String, String> letters = new HashMap<>();
String one = letters.merge("a", "1", String::concat); // {a=1}
  // one = 1
String two = letters.merge("a", "2", String::concat); // {a=12}
  // two = 12
```

---

### `Map.compute` metódus

- Előző érték figyelembe vételével kiszámolja az új értéket

```
Map<String, String> letters = new HashMap<>();
String one = letters.compute("a", (k, v) -> (v == null) ? '1' :
v.concat('1')); // {a=1}
  // one = 1
String two = letters.compute("a", (k, v) -> (v == null) ? '1' :
v.concat('1')); // {a=11}
  // two = 11
```

---

### `Map.computeIfPresent` metódus

```
Map<String, Integer> counts = new HashMap<>();
counts.put("Jenny", 1);
Integer jenny = counts.computeIfPresent("Jenny", (k, v) -> v + 1); //
{Jenny=2}
  // jenny = 2
Integer sam = counts.computeIfPresent("Sam", (k, v) -> v + 1); // {Jenny=2}
  // sam = null
```

---

### `Map.computeIfAbsent` metódus

```
Map<String, Integer> counts = new HashMap<>();
counts.put("Jenny", 15);
Integer jenny = counts.computeIfAbsent("Jenny", k -> 1); // {Jenny=15}
  // jenny = 15
Integer sam = counts.computeIfAbsent("Sam", k -> 1); // {Jenny=15, Sam=1}
  // sam = 1
```

---

### `Map.replace`

Két paraméteres (csak ha benne van):

```
Map<String, String> favorites = new HashMap<>();
favorites.put("Tom", "Tram");
favorites.put("Jenny", "Tram"); // {Tom=Tram, Jenny=Tram}
String tom = favorites.replace("Tom", "Bus Tour");
  // {Tom=Bus Tour, Jenny=Tram}; tom = Tram
String henry = favorites.replace("Henry", "Tram");
  // {Tom=Bus Tour, Jenny=Tram}; henry = null
```

Három paraméteres (ha benne van, és az az értéke):

```
Map<String, String> favorites = new HashMap<>();
favorites.put("Tom", "Tram"); // {Tom=Tram}
boolean hiking = favorites.replace("Tom", "Bus Tour", "Hiking");
  // {Tom=Tram}; hiking = false
boolean busTour = favorites.replace("Tom", "Tram", "Bus Tour");
  // {Tom=Bus Tour}; busTour = true
```

## `Map.remove`

Két paraméteres (csak ha az az értéke):

```
Map<String, String> favorites = new HashMap<>();
favorites.put("Tom", "Tram"); // {Tom=Tram}
boolean busTour = favorites.remove("Tom", "Bus Tour"); // {Tom=Tram}
  // busTour = false
favorites.remove("Tom", "Tram"); // {}
  // busTour = true
```

## `Collections` osztály

- `NavigableSet` és `NavigableMap` támogató metódusok `unmodifiable|synchronized|checked|empty)Navigable(Set|Map)`
- Debugging célokból checked wrapperek, ahol nincs deklarálva típus paraméter
- Új `checkedQueue` metódus
- `emptySorted(Set|Map)` metódusok

## Sorok beolvasása

```
try (Stream<String> lines = Files.lines(path)) {
    Optional<String> passwordEntry
        = lines.filter(s -> s.contains("password")).findFirst();
}
```

Alapértelmezett UTF-8 karakterkódolás

## `close()` láncolása

```
try (Stream<String> filteredLines
    = Files.lines(path).filter(s -> s.contains("password"))) {
            Optional<String> passwordEntry = filteredLines.findFirst();
}
```

## `onClose()` handler

```
try (Stream<String> filteredLines
    = Files.lines(path).onClose(() -> System.out.println("Closing"))
        .filter(s -> s.contains("password"))) { ... }
```

## `BufferedReader lines()` metódusa

```
try (BufferedReader reader
    = new BufferedReader(new InputStreamReader(url.openStream()))) {
            Stream<String> lines = reader.lines();
}
```

---

## Könyvtár bejárás

```
try (Stream<Path> entries = Files.list(pathToDirectory)) {
    ...
}
```

Nem lép be az alkönyvtárakba

---

## Rekurzív könyvtár bejárás

```
try (Stream<Path> entries = Files.walk(pathToRoot)) {
    // depth-first bejárás
}
```

- `Files.walk(pathToRoot, depth)` megadható maximum mélység
- `FileVisitOption.FOLLOW_LINKS` szimbólikus linkek követése

---

## Base64

```
Base64.Encoder encoder = Base64.getEncoder();
String original = username + ":" + password;
String encoded =
encoder.encodeToString(original.getBytes(StandardCharsets.UTF_8));
```

---

## Base64 streameknél

```
Path originalPath = ..., encodedPath = ...;
Base64.Encoder encoder = Base64.getMimeEncoder();
try (OutputStream output = Files.newOutputStream(encodedPath)) {
    Files.copy(originalPath, encoder.wrap(output));
}
```

```
Path encodedPath = ..., decodedPath = ...;
Base64.Decoder decoder = Base64.getMimeDecoder();
try (InputStream input = Files.newInputStream(encodedPath)) {
    Files.copy(decoder.wrap(input), decodedPath);
}
```

## Repeating annotation

```
@Repeatable(Schedules.class)
public @interface Schedule {
 String cron() default "";
}

public @interface Schedules {
     Schedule[] value();
}

@Schedule(cron = "0 0 1 1 *")
@Schedule(cron = "0 0 * * *")
public void doPeriodicCleanup() { ... }
```

---

## Type annotation

- Példányosításnál

```
new @Interned MyObject();
```

- Típuskényszerítésnél

```
myString = (@NonNull String) str;
```

---

## Type annotation

- Interfész implementáláskor

```
class UnmodifiableList<T> implements
    @Readonly List<@Readonly T> { ... }
```

- Metódusban kivételek deklarációjánál

```
void monitorTemperature() throws
    @Critical TemperatureException { ... }
```

---

## Type annotation használata

- Pluginelhető típus ellenőrzés
- Java nem tartalmaz ilyen keretrendszert
- Még erősebb típusosság
- Checker Framework

## Atomic osztályok

```
public static AtomicLong largest = new AtomicLong();
largest.set(Math.max(largest.get(), observed)); // Nem szálbiztos
```

helyett

```
largest.updateAndGet(x -> Math.max(x, observed));
```

vagy

```
largest.accumulateAndGet(observed, Math::max);
```

---

## `LongAdder` és `LongAccumulator` osztályok

- `AtomicLong` helyett
- Túl sok szál okozta teljesítménycsökkenés esetén

```
LongAccumulator adder = new LongAccumulator(Long::sum, 0);
adder.accumulate(value);
```

- `Short`, `Integer`, `Long`, `Float` és `Double` osztályoknak statikus `sum`, `max` és `min` metódusok

---

## `ConcurrentHashMap` újdonságok

- `putIfAbsent`, `compute`, `computeIfAbsent`, `merge` metódusok szálbiztosak
- Három fajta művelet:
- `search`: keresés
- `reduce`: összes elem alapján egy értéket állít elő
- `forEach`: minden elemre végrehajtja

---

## Metódusok formái

- Négy verziója:
- `xxxKeys`: csak a kulcsokon
- `xxxValues`: csak az értékeken
- `xxx`: kulcs és érték párokon
- `xxxEntries`: `Map.Entry` objektumokon
- `threshold` paraméter: mennyinél indítson új szálat

---

## `search` metódusok

```
String result = map.search(threshold, (k, v) -> v > 1000 ? k : null);
```

---

## `forEach` metódusok

```
map.forEach(threshold,
  (k, v) -> System.out.println(k + " -> " + v));
```

```
map.forEach(threshold,
  (k, v) -> k + " -> " + v, // Transformer
  System.out::println); // Consumer

map.forEach(threshold,
  (k, v) -> v > 1000 ? k + " -> " + v : null, // Szűri a null elemeket
    System.out::println);
```

---

## `reduce` metódusok

```
Long sum = map.reduceValues(threshold, Long::sum);

Integer maxlength = map.reduceKeys(threshold,
  String::length, // Transformer
  Integer::max); // Accumulator

Long count = map.reduceValues(threshold,
  v -> v > 1000 ? 1L : null, // Null értékeket szűri
  Long::sum);
```

`ToInt`, `ToLong` és `ToDouble` posztfixekkel

---

## `Set` view

- Nincs concurrent hash set, ezért csak egy view

```
Set<String> words = ConcurrentHashMap.<String>newKeySet();

Set<String> words = map.keySet(1L);
```

---

## `Arrays` párhuzamos műveletek

- `parallelSort` metódus, rendezés
- `parallelSetAll` metódus, elemek előállítása index alapján

```
Arrays.parallelSetAll(values, i -> i % 10);
```

- `Arrays.parallelPrefix` kombinálás az előző elemekkel

```
int[] i = {1, 1, 2, 2, 3, 3};
Arrays.parallelPrefix(i, (x, y) -> x + y ); // i = [1, 2, 4, 6, 9, 12]
```

### `CompletableFuture`

- Composition pipeline
- Creation: `CompletableFuture.supplyAsync()` vagy `runAssync()`
- Közbülső műveletek: `thenApply()` vagy `thenApplySync()`
- Lezárás: `thenAccept()`

```
CompletableFuture<List<String>> links
  = CompletableFuture.supplyAsync(() -> blockingReadPage(url))
    .thenApply(Parser::getLinks);

CompletableFuture<Void> links
  = CompletableFuture.supplyAsync(() -> blockingReadPage(url))
    .thenApply(Parser::getLinks)
    .thenAccept(System.out::println);
```

---

## `CompletableFuture` metódusok

- `thenApply`: `T -> U`, visszatérési értéken egy metódust hív
- `thenCompose`: `T -> CompletableFuture<U>`, visszatérési értéken egy metódust hív, aminek a visszatérési értéke `CompletableFuture`
- `handle`: `(T, Throwable) -> U`, kivételt is kezel
- `thenAccept`: `T -> void`
- `whenComplete`: `(T, Throwable) -> void`
- `thenRun`: `Runnable` paraméterrel

---

## `CompletableFuture` összefűzés

- `thenCombine`: mindkettőt lefuttatja, az eredményeket kombinálja
- `thenAcceptBoth`: mindkettőt lefuttatja, az eredményeket feldolgozza
- `runAfterBoth`: mindkettőt lefuttatja, majd a paraméterként átadott `Runnable` példányt futtatja
- `applyToEither`: amelyik előbb végez, annak eredményét alakítja át
- `acceptEither`: amelyik előbb végez, annak eredményét dolgozza fel
- `runAfterEither`, amelyik előbb végez, az után indít `Runnable` példányt
- `static allOf`: mindkettőt lefuttatja
- `static anyOf`: visszatér, ahogy lefut az előbb végző

## Paraméter nevek reflectionnel

```
Person getEmployee(@PathParam("dept") Long dept, @QueryParam("id") Long id)
```

helyett

```
Person getEmployee(@PathParam Long dept, @QueryParam Long id)
```

- Új osztállyal: `java.lang.reflect.Parameter`
- Fordítás a `-parameters` kapcsolóval

---

## `Null` ellenőrzés

```
stream.anyMatch(Object::isNull)
```

```
stream.filter(Object::nonNull)
```

---

## Lazy naplózó üzenetek

Késői kiértékelés

```
logger.finest(() -> "x: " + x + ", y:" + y);
```

```
this.directions = Objects.requireNonNull(directions,
    () -> "directions for " + this.goal + " must not be null");
```

Ha a `directions` értéke `null`, `NullPointerException` a megadott üzenettel

---

## Reguláris kifejezések

```
(?<city>[\p{L} ]+),\s*(?<state>[A-Z]{2})
```

```
Matcher matcher = pattern.matcher(input);
if (matcher.matches()) {
    String city = matcher.group("city");
}
```

```
Stream<String> words = Pattern.compile("[\\P{L}]+").splitAsStream(contents);
```

```
Stream<String> acronyms = words.filter(Pattern.compile("[A-
Z]{2,}").asPredicate());
```

---

## `String` műveletek

- `join` metódus

```
String joined = String.join("/", "usr", "local", "bin"); // "usr/local/bin"
```

---

## Matematikai műveletek

- `add | subtract | multiply | increment | decrement | negate )Exact` metódusok `int` és `long` paraméterekkel
- Túlcsordulás helyett kivételt dobnak