



Academia Tehnică Militară "Ferdinand I"
Facultatea de Sisteme Informaticе și Securitate Cibernetică

PARADIGME DE PROGRAMARE (ÎN JAVA)

CURS 6 - PROGRAMARE FUNCȚIONALĂ

COL(R) TRAIAN NICULA

TEMATICĂ DE CURS

- Introducere în paradigme de programare (1)
- Programare orientată pe obiecte (OOP) (3)
- **Programare funcțională și asincronă (2/3)**
- Programare reactivă (1)
- Servicii web REST (2)
- Quarkus Framework (4)

CUPRINS CURS

- Colectarea datelor cu stream-uri
- Procesare paralelă și performanță
- Metode implicite
- Optional ca alternativă la null
- Gândire funcțională
- Bibliografie

COLECTAREA DATELOR CU STREAM-URI

- Am învățat în cursul precedent că **stream-urile ne ajută să procesăm colecțiile** în **mod declarativ**, similar operațiilor cu baze de date
- Am folosit **collect** ca **operație terminală** pentru a **combina toate elementele stream-ului** într-o listă
- Invocarea metodei **collect** pe un stream **declanșează o operație de reducere** pe elementele stream-ului (parametrizată cu un Collector)
- În acest curs vom descoperi cum **putem folosi collect** ca o operație de reducere pentru a **obține rezultat summarizat**
- **Collectors** sunt utili pentru că furnizează un **mecanism flexibil și concis de definire a criteriilor pentru colectarea datelor**



COLECTAREA DATELOR CU STREAM-URI

- Exemplu: gruparea tranzacțiilor după valută
- *Soluția imperativă*

```

Map<Currency, List<Transaction>> transactionsByCurrencies =
    new HashMap<>();
for (Transaction transaction : transactions) {
    Currency currency = transaction.getCurrency();
    List<Transaction> transactionsForCurrency =
        transactionsByCurrencies.get(currency);
    if (transactionsForCurrency == null) {
        transactionsForCurrency = new ArrayList<>();
        transactionsByCurrencies
            .put(currency, transactionsForCurrency);
    }
    transactionsForCurrency.add(transaction);
}

```

Create the Map where the grouped transaction will be accumulated.

Extract the Transaction's currency.

Iterate the List of Transactions.

If there's no entry in the grouping Map for this currency, create it.

Add the currently traversed Transaction to the List of Transactions with the same currency.

- *Soluția funcțională declarativă*

```

Map<Currency, List<Transaction>> transactionsByCurrencies =
    transactions.stream().collect(groupingBy(Transaction::getCurrency));

```



COLECTAREA DATELOR CU STREAM-URI

COLECTORI PREDEFINIȚI

- Colectorii predefiniți pot fi creați cu metodele clasei **Collectors** și oferă 3 funcționalități importante:
 - **Reducerea și sumarea elementelor** la o singură valoare – calcularea sumei totale a valorii tranzacționate
 - **Gruparea elementelor** – exemplul anterior
 - **Partiționarea elementelor** – caz special de grupare care folosește un Predicate ca funcție de grupare
- **Reducerea și sumarea**
- **Collectors**, parametrii metodei **collect**, pot fi folosiți pentru:
 - A **reorganiza** elementele stream-ului **într-o colecție**
 - Combinarea elementelor **într-un rezultat final**
 - Sau tipuri complexe precum un **map multinivel**

COLECTAREA DATELOR CU STREAM-URI

COLECTORI PREDEFINIȚI

- Exemplu: **calcularea persoanelor** din lista population:

```
long howMany = population.stream().collect(Collectors.counting());  
System.out.println(howMany);
```

- Același lucru se poate realiza cu operația **count**, dar colectorul **counting** are avantajul că poate fi **combinat cu alți colectori**
- Dacă importăm static clasa **Collectors**

```
import static java.util.stream.Collectors.*;
```

- Putem utiliza **counting()** în loc de **Collectors.counting()**



COLECTAREA DATELOR CU STREAM-URI

COLECTORI PREDEFINIȚI

- **Găsirea valorii maxime și minime dintr-un stream de valori**
- Se folosesc colectorii **Collectors.maxBy** și **Collectors.minBy** care primesc un **Comparator** ca argument

```
Comparator<Person> personAgeComp = Comparator.comparingInt(Person::getAge);
Optional<Person> oldest = population.stream().collect(maxBy(personAgeComp));
System.out.println("oldest: " + oldest.get().getName());
Optional<Person> youngest = population.stream().collect(minBy(personAgeComp));
System.out.println("youngest: " + youngest.get().getName());
```

Rezultat:

oldest: Mihai

youngest: Marius

- Codul afișează numele persoanelor cu vârstă maximă și minimă din populație
- **Optional** este folosit pentru cazul **nu există nicio persoană** în stream



COLECTAREA DATELOR CU STREAM-URI

COLECTORI PREDEFINIȚI

- Clasa **Collectors** furnizează o metodă pentru sumare, **Collectors.summingInt**, care mapează un obiect la int ce urmează a fi sumat
- **Collectors.summingLong** și **Collectors.summingDouble** se comportă în mod similar pentru tipurile primitive corespunzătoare
- În afară de sumare există **posibilitatea de a calcula media** folosind metodele **Collectors.averagingInt**, **averagingLong** și **averagingDouble**

```
int totalAge = population.stream().collect(summingInt(Person::getAge));
System.out.println("Total age: " + totalAge);
double avgAge = population.stream().collect(averagingInt(Person::getAge));
System.out.println("Average age: " + avgAge);
```

Rezultat:
 Total age: 278
 Average age: 30.889

- Pentru generarea de statistici

```
IntSummaryStatistics populationStats = population.stream().collect(summarizingInt(Person::getAge));
System.out.println(populationStats);
```

Rezultat:

IntSummaryStatistics{count=9, sum=278, min=5, average=30.888889, max=55}



COLECTAREA DATELOR CU STREAM-URI

COLECTORI PREDEFINIȚI

- Colectorul returnat de **metoda *joining*** concatenează într-un singur String toate sirurile de caractere rezultate prin invocarea metodei *toString* pe toate obiectele din stream

```
String allNames = population.stream().map(Person::getName).collect(joining());  
System.out.println(allNames);
```

- Rezultat: *IonMariaVasileVioletaMariusAlexiaGeorgeCristinaMihai*
- Se observă că rezultatul nu este util, dar există o altă **versiune supraîncărcată a metodei *joining*** care acceptă un String de delimitare între elementele consecutive

```
allNames = population.stream().map(Person::getName).collect(joining(", "));  
System.out.println(allNames);
```

- Rezultat: *Ion, Maria, Vasile, Violeta, Marius, Alexia, George, Cristina, Mihai*



COLECTAREA DATELOR CU STREAM-URI

COLECTORI PREDEFINIȚI

- Colectorii discutați până acum sunt **versiuni specializate**, convenabile, ale **procesului de reducere** definit de metoda **reducing**
- Putem calcula suma vârstei persoanelor folosind **reducing**

```
int totalAge = population.stream().collect(reducing(0, Person::getAge, (i, j) -> i + j));  
System.out.println(totalAge);
```

- Rezultat: 278
- **Metoda reducing** primește 3 argumente:
 - **Valoare de pornire** a operației de reducere - 0 în exemplu
 - **Funcția folosită pentru a transforma obiectul** Persoană în întreg – Person::getAge
 - **BinaryOperator** care agregă 2 elemente într-o singură valoare – suma în acest caz

```
Optional<Person> maxAge = population.stream().collect(reducing(  
    (p1, p2) -> p1.getAge() > p2.getAge() ? p1 : p2));  
System.out.println(maxAge.get());
```

Rezultat: Mihai



COLECTAREA DATELOR CU STREAM-URI

COLECTORI PREDEFINIȚI

- Diferența dintre **collect** și **reduce**
 - **Metoda reduce** se folosește pentru a **combina 2 valori și a rezulta una nouă**, este o **reducere imutabilă**
 - **Metoda collect** poate opera pe un **container mutabil** și în același timp poate **lucra în paralel**
- Exemplu anterior poate fi simplificat și mai mult

```
int totalAge = population.stream()
    .collect(reducing(0,    // initial value
                      Person::getAge, // transformation function
                      Integer::sum)); // aggregation function

System.out.println(totalAge);
```

- Rezultat: 278
- Stream API permite ca aceeași operație să fie executată în mai multe feluri



COLECTAREA DATELOR CU STREAM-URI

GRUPAREA

- O operație comună în lucrul cu bazele de date o reprezintă **gruparea elementelor în seturi**
- Pentru gruparea elementelor utilizăm metoda **Collectors.groupingBy**

```
Map<Person.Gender, List<Person>> personByGender = population.stream()
    .collect(groupingBy(Person::getGender));
System.out.println(personByGender);
```

- Rezultat: {**MALE**=[Ion, Vasile, Marius, George, Mihai], **FEMALE**=[Maria, Violeta, Alexia, Cristina]}
- **Person::getGender** poartă numele de **funcție de clasificare** deoarece este **folosită pentru separarea elementelor în grupuri diferite**
- **Rezultatul** operației de grupare este **tipul Map** având ca și **cheie** valoarea **returnată de funcția de clasificare** iar ca **valoare** o **listă cu elementele care satisfac criteriul de clasificare**



COLECTAREA DATELOR CU STREAM-URI

GRUPAREA

- Se poate realiza **gruparea multinivel** folosind o versiune supraîncărcată a metodei **Collectors.groupingBy** care acceptă **2 argumente: primul este funcția de clasificare iar al 2-lea este tot un colector de același tip**

```
Map<Person.Gender, Map<Stages, List<Person>>> peopleByGenderStages = population.stream()
.collect(
    groupingBy(Person::getGender,
        // Second level classification
        groupingBy(p -> {
            if(p.getAge() < 12)
                return Stages.CHILD;
            else if (p.getAge() >= 12 && p.getAge() < 18)
                return Stages.TEENAGER;
            else
                return Stages.ADULT;
        })
    );
System.out.println(peopleByGenderStages);
```

```
package atm.paradigms.tests;

public enum Stages {
    CHILD, TEENAGER, ADULT
}
```

{**MALE**={ADULT=[Ion, Vasile, George, Mihai], CHILD=[Marius]}, **FEMALE**={ADULT=[Maria, Violeta, Alexia, Cristina]}}



COLECTAREA DATELOR CU STREAM-URI

GRUPAREA

- În exemplu anterior am văzut că este posibil să trecem un al 2-lea colector **groupingBy** pentru realizarea grupării multinivel
- Operația de **grupare multinivel** poate fi extinsă la un număr oricât de **mare de nivele**
- **groupingBy acceptă ca al 2-lea argument orice tip de colector**
- Putem calcula numărul de persoane în funcție de gen

```
Map<Person.Gender, Long> genderCount = population.stream()
.collect(groupingBy(Person::getGender, counting()));
System.out.println(genderCount);
```

- Rezultat: {FEMALE=4, MALE=5}



COLECTAREA DATELOR CU STREAM-URI

GRUPAREA

- Pentru a găsi **cea mai în vîrstă persoană** în funcție de gen folosim:

```
Map<Person.Gender, Optional<Person>> oldestByGender = population.stream()
.collect(groupingBy(Person::getGender,
                    maxBy(comparingInt(Person::getAge)))
));
System.out.println(oldestByGender);
```

Rezultat: {**MALE=Optional[Mihai]**, **FEMALE=Optional[Violeta]**}

- Optional** este tipul returnat de colectorul **maxBy**, care nu este util în acest caz deoarece cheia este adăugată numai dacă găsește elemente
- Putem evita **Optional** cu metoda **Collectors.collectingAndThen**

```
Map<Person.Gender, Person> oldestByGender = population.stream()
.collect(groupingBy(Person::getGender,
                    collectingAndThen(
                        maxBy(comparingInt(Person::getAge)),
                        Optional::get
                    )
));
System.out.println(oldestByGender);
```

Rezultat:
{FEMALE=Violeta, MALE=Mihai}



COLECTAREA DATELOR CU STREAM-URI

GRUPAREA

- Putem **suma vîrstele persoanelor grupate în funcție de gen** folosind metoda *summingInt*

```
Map<Person.Gender, Integer> totalAgeByGender = population.stream()
.collect(groupingBy(Person::getGender,
                    summingInt(Person::getAge)));
System.out.println(totalAgeByGender);
```

- Rezultat: {*FEMALE*=119, *MALE*=159}
- Pentru a **transforma elementele** înainte de folosim metoda *mapping*
- mapping* are **2 argumente: funcția de transformare și colectorul** care acumulează rezultatul

```
Map<Person.Gender, Set<Stages>> humanStagebyGender = population.stream()
.collect(groupingBy(Person::getGender,
                    mapping(p -> {
                        if(p.getAge() < 12) return Stages.CHILD;
                        else if (p.getAge() >= 12 && p.getAge() < 18) return Stages.TEENAGER;
                        else return Stages.ADULT;
                    }),
                    toSet()))
                );
```

Rezultat: {*MALE*=[*ADULT*, *CHILD*], *FEMALE*=[*ADULT*]}

COLECTAREA DATELOR CU STREAM-URI

PARTIȚIONAREA

- **Partiționarea** este un **caz special de grupare**, care **folosește un Predicate** (returnează boolean) ca funcție de partiționare
- **Map** rezultat va avea **cheie tipul Boolean** rezultând 2 grupuri diferite, unul pentru **true** și unul pentru **false**

```
Map<Boolean, List<Person>> partitioned = population.stream()
.collect(partitioningBy(Person::isEmployed));
List<Person> employed = partitioned.get(true);
System.out.println(employed);
```

- Rezultat: [Ion, Violeta, Alexia, George, Mihai]
- Același rezultat se putea obține cu:

```
List<Person> employed = population.stream().filter(Person::isEmployed).collect(toList());
System.out.println(employed);
```



COLECTAREA DATELOR CU STREAM-URI

PARTIȚIONAREA

- **Partiționarea** are avantajul că **păstrează ambele liste** din stream-ul de elemente corespunzătoare *true* sau *false*
- La fel ca **groupingBy** există o metodă supraîncărcată **partitioningBy** care primește **2 argumente: funcția de partiționare și un colector**

```
Map<Boolean, Map<Person.Gender, List<Person>>> employedPeopleByGender =  
population.stream().collect(  
    partitioningBy(Person::isEmployed,  
                  groupingBy(Person::getGender))  
)  
System.out.println(employedPeopleByGender);
```

- Rezultat: `{false={MALE=[Vasile, Marius], FEMALE=[Maria, Cristina]}, true={MALE=[Ion, George, Mihai], FEMALE=[Violeta, Alexia]}`

COLECTAREA DATELOR CU STREAM-URI

PARTIȚIONAREA

- Exemplu: găsirea **persoanelor cu vârstă cea mai mare** din grupurile de **angajați și neangajați**

```
Map<Boolean, Person> olderPartitionedByEmployed =  
population.stream().collect(  
    partitioningBy(Person::isEmployed,  
        collectingAndThen(  
            maxBy(comparingInt(Person::getAge)),  
            Optional::get)  
    )  
);  
System.out.println(olderPartitionedByEmployed);
```

- Rezultat: {*false*=Vasile, *true*=Mihai}



COLECTAREA DATELOR CU STREAM-URI

[Cuprins](#)

PARTIȚIONAREA

- Partiționarea numerelor în prime și neprime

```
package atm.paradigms.tests;
import java.util.*;
import java.util.stream.IntStream;
import static java.util.stream.Collectors.*;

public class PartitioningByTest {
    public static void main(String[] args) {
        Map<Boolean, List<Integer>> primes = partitionPrime(29);
        System.out.println(primes);
    }
    private static boolean isPrime(int candidate){
        // factors less than or equal with the square root
        int root = (int) Math.sqrt(candidate);
        return IntStream.rangeClosed(2, root)
            .noneMatch(i -> candidate % i == 0);
    }
    public static Map<Boolean, List<Integer>> partitionPrime(int n){
        return IntStream.rangeClosed(2, n).boxed()
            .collect(partitioningBy(c -> isPrime(c)));
    }
}
```

Rezultat:

```
{false=[4, 6, 8, 9, 10, 12, 14, 15, 16,
18, 20, 21, 22, 24, 25, 26, 27, 28],
true=[2, 3, 5, 7, 11, 13, 17, 19, 23,
29]}
```

Metoda **isPrime** verifică potențialii divizori ai numărului **candidate**. Cu **noneMatch** se asigură că nici unul nu se împarte exact.

Metoda **partitioningBy** împarte numerele din interval în prime și neprime.

COLECTAREA DATELOR CU STREAM-URI

METODELE STATICHE ALE CLASEI COLLECTORS

Metodă	Tip returnat	Descriere
toList	List<T>	Adună elementele stream-ului într-o listă
<code>List<Person> people = populationStream.collect(toList());</code>		
toSet	Set<T>	Adună elementele stream-ului într-un Set, eliminând duplicatele
<code>Set<Person> people = populationStream.collect(toSet());</code>		
toCollection	Collection<T>	Adună elementele stream-ului într-o colecție creată de Supplier-ul furnizat ca argument
<code>Collection<Person> people = populationStream.collect(toCollection(), ArrayList::new)</code>		
counting	long	Numără elementele din stream
<code>long peopleCount = populationStream.collect(counting());</code>		
summingInt	Integer	Suma valorilor proprietății Integer a elementelor din stream
<code>int totalAge = populationStream.collect(summingInt(Person::getAge));</code>		
averagingInt	Double	Media valorilor proprietății Integer a elementelor din stream
<code>double avgAge = populationStream.collect(averagingInt(Person::getAge));</code>		

COLECTAREA DATELOR CU STREAM-URI

METODELE STATICHE ALE CLASEI COLLECTORS

Metodă	Tip returnat	Descriere
summarizingInt	IntSummaryStatistics	Colectează statistici privind proprietatea de tip Integer a elementelor din stream precum minim, maxim, total și media
<code>populationStream.collect(summarizingInt(Person::getAge));</code>		
joining	String	Concatenează String-urile rezultate prin aplicarea metodei map pe fiecare element din stream
<code>String names = populationStream.map(Person::getName).collect(joining(", "));</code>		
maxBy	Optional<T>	Un optional în jurul elementului maxim din stream determinat de comparatorul dat sau Optional.empty() dacă stream-ul este gol
<code>Optional<Person> maxAge = populationStream.collect(maxBy(comparingInt(Person::getAge)));</code>		
minBy	Optional<T>	Un optional în jurul elementului minim din stream determinat de comparatorul dat sau Optional.empty() dacă stream-ul este gol
<code>Optional<Person> minAge = populationStream.collect(minBy(comparingInt(Person::getAge)));</code>		



COLECTAREA DATELOR CU STREAM-URI

METODELE STATICHE ALE CLASEI COLLECTORS

Metodă	Tip returnat	Descriere
reducing	Tipul produs de operația de reducere	Reduce stream-ul la o singură valoare pornind de la o valoare initială folosită acumulator pe care o combină iterativ cu fiecare element din stream folosind un BinaryOperator
<code>int totalAge = populationStream.collect(reducing(0, Person::getAge, Integer::sum));</code>		
collectingAndThen	Tipul produs de operația de transformare	Aplică o funcție de transformare pe rezultatul returnat de un alt colector
<code>int totalAge = populationStream.collect(collectingAndThen(toList(), List::size));</code>		
groupingBy	Map<K, List<T>>	Grupează elementele dintr-un stream funcție de valoarea proprietăților lor și folosește acele valori ca și chei în Map-ul rezultat
<code>Map<Person.Gender, List<Person>> peopleByGender = populationStream.collect(groupingBy(Person::getGender));</code>		
partitioningBy	Map<Boolean, List<T>>	Partiționează elementele din stream pe baza rezultatului în urma aplicării unui Predicate pe fiecare element
<code>Map<Boolean, List<Person>> partitionByEmployed = populationStream.collect(partitioningBy(Person::isEmployed));</code>		



PROCESARE PARALELĂ ȘI PERFORMANCE

- Înainte de Java 7 procesarea paralelă a colecțiilor era complicată:
 - Era necesară **spargerea explicită** a structurii de date în secțiuni
 - **Fiecare secțiune** era **atribuită unui fir de execuție** diferit
 - Era necesară **sincronizarea acestora** pentru a evita condiții de concurență, se aștepta **finalizarea tuturor firelor de execuție** apoi se **combinau rezultatele parțiale** în rezultatul final
- **Java 7 a introdus** un mecanism denumit **fork/join** care permite:
 - **fork** - împărțirea unei sarcini paralelizate în subsarcinii ce se execută pe thread-uri separate din **thread pool** (ForkJoinPool). Implementează interfața **ExecutorService**
 - **join** - rezultatul final se obține prin combinarea rezultatelor fiecărei subsarcini
- **Stream API** oferă **posibilitatea de a executa operații paralele** fără mult efort și într-o manieră consistentă
- Putem **converti o colecție într-un stream paralel** prin invocarea metodei **parallelStream** pe acesta



PROCESARE PARALELĂ ȘI PERFORMANCE

- Un **stream paralel** este un stream care **este spart în secțiuni** care se procesează pe fire de execuție diferite
- **Procesarea este împărțită pe toate nucleele procesorului** astfel încât toate lucrează
- Exemplu: să se scrie o metodă care ia ca argument un număr n și returnează suma numerelor de la 1 la argumentul dat

```
public static long iterativeSum(long n){  
    long result = 0;  
    for (long i = 1L; i <= n; i++){  
        result += i;  
    }  
    return result;  
}
```

Rezolvare iterativă în stil Java 7

```
public static long sequentialSum(long n){  
    return Stream.iterate(1L, i -> i + 1)  
        .limit(n)  
        .reduce(0L, Long::sum);  
}
```

Rezolvare secvențială folosind Stream API



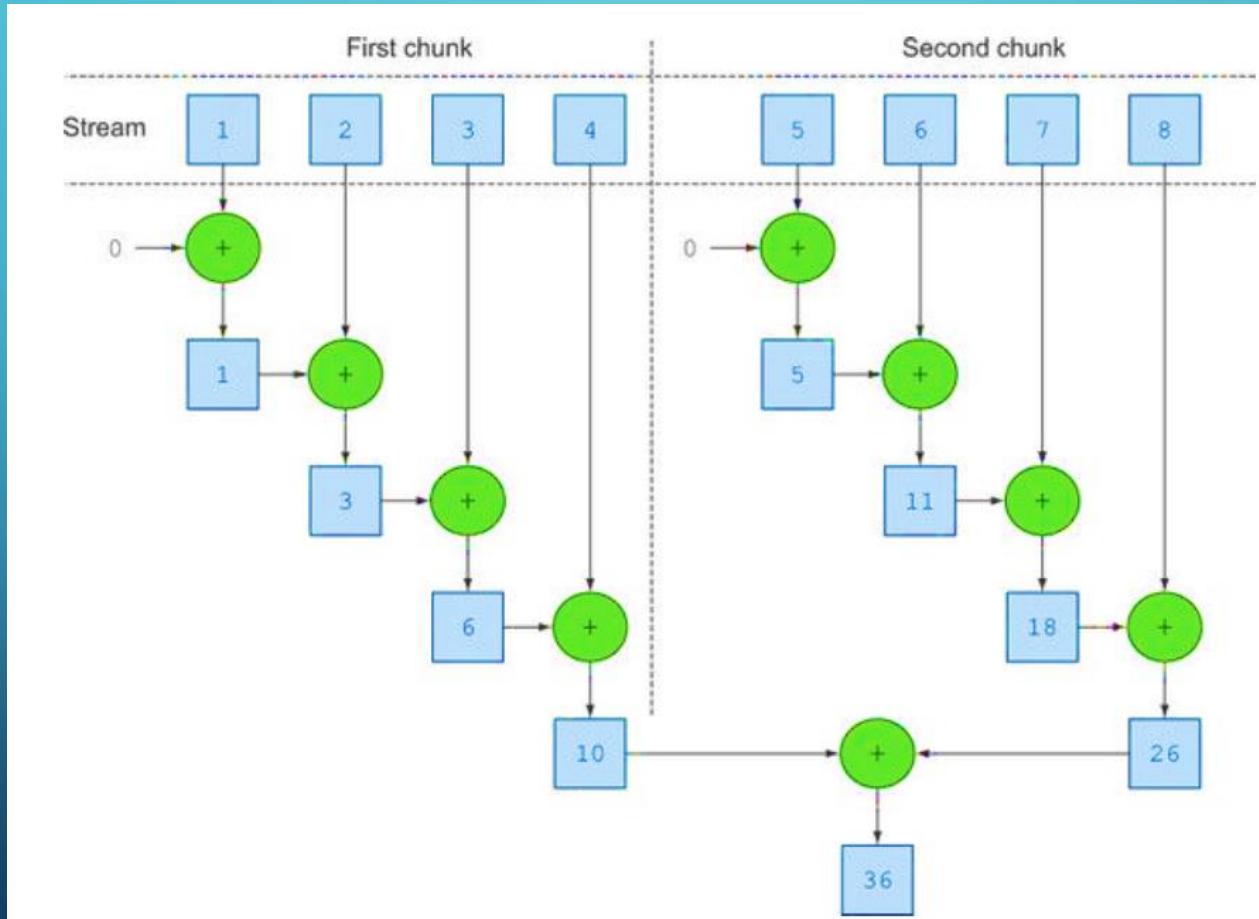
PROCESARE PARALELĂ ȘI PERFORMANȚĂ

- **Exemplu** este bun **candidat pentru paralelizare** mai ales pentru numere mari
- Posibile întrebări pentru soluția Java 7:
 - Trebuie sincronizată variabila rezultat?
 - De câte fire de execuție este nevoie?
 - Cum se efectuează generarea numerelor?
 - Cum se face adunarea la final?
- Problema este simplă cu stream-uri paralele
- **Exemplul secvențial de reducere** se paralelizează cu metoda **parallel**

```
public static long parallelSum(long n){  
    return Stream.iterate(1L, i -> i+ 1)  
        .limit(n)  
        .parallel() // turn to parallel stream  
        .reduce(0L, Long::sum);  
}
```

PROCESARE PARALELĂ ȘI PERFORMANȚĂ

- Stream API face intern împărțirea în porțiuni, operația de reducere poate lucra în paralel iar la sfârșit asamblează rezultatul final



PROCESARE PARALELĂ ȘI PERFORMANȚĂ

- Există metoda **sequential** care convertește un stream paralel într-un stream secvențial

- Ultima invocare a metodelor **parallel** sau **sequential** afectează întreg pipeline-ul

- Astfel:

stream.parallel()

.filter(...)

.sequential()

.map(...)

.parallel()

.reduce();

- se va execuța paralel

- Numărul implicit de fire de execuție este dat de proprietatea de sistem `java.util.concurrent.ForkJoinPool.common.parallelism` fiind **egal cu numărul de nuclee fizice**

PROCESARE PARALELĂ ȘI PERFORMANȚĂ

- Ne așteptăm ca **paralelizarea să se aibă o performanță mai bună decât procesarea secvențială**
- În ingineria software **regula de bază este măsurarea performanței**
- Exemplu de **funcție pentru măsurarea performanței**:
 - primește ca **argumente o funcție și un long**
 - **aplică funcția de 10 ori pe valoarea numerică trecută ca argument, înregistrează timpul pentru fiecare execuție și returnează pe cel mai rapid**

```
public static long measurePerformance(Function<Long, Long> adder, long n){  
    long fastest = Long.MAX_VALUE;  
    for (int i = 0; i < 10; i++){  
        long start = System.nanoTime();  
        long sum = adder.apply(n);  
        long duration = (System.nanoTime() - start)/1_000_000;  
        System.out.println("Result: " + sum);  
        if (duration < fastest) fastest = duration;  
    }  
    return fastest;  
}
```



PROCESARE PARALELĂ ȘI PERFORMANȚĂ

- Măsurăm performanța metodelor implementate

```
System.out.println("Sequential sum: "
    + measurePerformance(Parallelization::sequentialSum, 10_000_000) + " ms");
```

Rezultat: Sequential sum: 8 ms

```
System.out.println("Iterative sum: "
    + measurePerformance(Parallelization::iterativeSum, 10_000_000) + " ms");
```

Rezultat: Iterative sum: 0 ms

```
System.out.println("Parallel sum: "
    + measurePerformance(Parallelization::parallelSum, 10_000_000) + " ms");
```

Rezultat: Parallel sum: 35 ms

Rezultatele nu sunt cele așteptate din 2 cauze:

- **iterate** generează obiecte **boxed** care trebuie să fie **unboxed** pentru adunare
- **iterate** este dificil de împărțit în secțiuni independente deoarece **datele nu sunt disponibile în momentul procesării**



PROCESARE PARALELĂ ȘI PERFORMANȚĂ

- Pentru a beneficia de **avantajele paralelizării** utilizăm metoda **LongStream.rangeClosed** care are 2 avantaje:
 - lucrează cu tipul primitiv long**
 - produce un domeniu de numere ce pot fi ușor împărțite în porțiuni independente**

```
public static long rangedSum(long n){  
    return LongStream.rangeClosed(1, n)  
        .reduce(0L, Long::sum);  
}
```

Rezultat: *Ranged sum: 4 ms*

```
public static long parallelRangedSum(long n){  
    return LongStream.rangeClosed(1, n)  
        .parallel()  
        .reduce(0L, Long::sum);  
}
```

Rezultat: *Parallel ranged sum: 1 ms*

PROCESARE PARALELĂ ȘI PERFORMANȚĂ

- Recomandări pentru **folosirea eficientă a paralelizării**:
- **Măsurarea performanței** – un stream paralel nu este întotdeauna mai eficient decât unul secvențial
- **Atenție la boxing/unboxing** – poate influența semnificativ performanța. Se folosesc **stream-uri de primitive** ori de câte ori este posibil (**IntStream**, **LongStream**, **DoubleStream**)
- **Unele operații au performanțe mai proaste pe stream-uri paralele – limit și findFirst** care se bazează pe ordinea elementelor nu sunt recomandate pentru paralelizare
- Pentru **cantități mici de date paralelizarea nu este recomandată**
- **Structura datelor sursă pentru stream** – **ArrayList** poate fi spart mai ușor decât **LinkedList**

PROCESARE PARALELĂ ȘI PERFORMANȚĂ

- **Costul de calcul** pe pipeline este $N * Q$, unde N este numărul de elemente din stream iar Q este costul aproximativ de procesare. Cu cât **valoarea lui Q este mai mare cu atât performanța crește prin paralelizare**
- **Evaluarea costului de agregare** a datelor în operația terminală – combinarea rezultatelor parțiale poate elimina avantajele paralelizării
- Surse pentru stream-uri și eficiența decompoziției

Sursă	Decompoziție
ArrayList, IntStream.range	Excelentă
HashSet, TreeSet	Bună
LinkedList, Stream.iterate	Slabă



METODE IMPLICITE

- **Interfața în Java grupează metode înrudite**
- **Orice clasă care implementează interfața trebuie să implementeze toate metodele acesteia**
- Apar **probleme dacă interfața trebuie actualizată** prin adăugarea de metode noi
- Pentru a rezolva problema **Java 8** oferă posibilitatea de a **introduce metode cu cod de implementare**
- Exist 2 mecanisme:
 - **Metode statice** în interfețe
 - **Metode implicite** (default methods) – care furnizează o implementare pentru metode
- **Clasele care implementează o interfață moștenesc automat implementările implicite ale metodelor**



METODE IMPLICITE

- Metoda **sort** a interfeței **List** este introdusă în Java 8 și are definiția:

```
default void sort(Comparator<? super E> c){  
    Collections.sort(this, c);  
}
```

- Cuvântul cheie **default** marchează metoda ca implicită

```
List<Integer> numbers = Arrays.asList(3, 5, 7, 2, 1, 4);  
numbers.sort(Comparator.naturalOrder());  
System.out.println(numbers);
```

- Rezultat: [1, 2, 3, 4, 5, 7]
- Lista este sortată chemând metoda **sort** pe ea
- **Comparator.naturalOrder** este o metodă statică a interfeței **Comparator** care returnează un obiect de tip **Comparator** cu sortare în ordine naturală

METODE IMPLICITE

- Metodele **implicite** creează **posibilitatea de a actualiza interfețele** fără a provoca modificări pentru implementările existente
- Până la Java 8 metodele **utilitare statice** erau **implementate în clase companion**
- *Collections* este o clasă companion cu metode pentru lucrul cu obiecte de tip *Collection*
- În implementările post Java 8 metodele statice pot fi **mutate în interfețe**
- Pentru a demonstra utilitatea metodelor implicite vom dezvolta un exemplu cu o bibliotecă pentru desenat geometrii
- Biblioteca are o interfață **Resizable** cu metodele pe care o geometrie trebuie să le suporte: `setHeight`, `setWidth`, `getHeight`, `getWidth`, `setAbsoluteSize`
- Biblioteca implementează și clasele **Square** și **Rectangle**



METODE IMPLICITE

```
package atm.paradigms.shapes;

public interface Resizable {
    int getWidth();
    int getHeight();
    void setWidth(int width);
    void setHeight(int height);
    void setAbsoluteSize(int width, int height);
    void draw();
}
```

```
package atm.paradigms.shapes;

import java.util.*;

public class ShapesTest {
    public static void main(String[] args) {
        List<Resizable> shapes =
            Arrays.asList(new Square(), new Rectangle(),
                          new Ellipse());
        Utils.paint(shapes);
    }
}
```

```
package atm.paradigms.shapes;

import java.util.List;

public class Utils {
    public static void paint(List<Resizable> l){
        l.forEach( r -> {
            r.setAbsoluteSize(50, 50);
            r.draw();
        });
    }
}
```

Clasele care implementează interfața **Resizable** implementează și metodele acesteia. Metoda statică **paint** face uz de aceste metode.

Instanțe ale claselor **Square**, **Rectangle** și **Ellipse** care implementează interfața **Resizable** sunt adăugate într-o listă. Lista se trimită ca argument metodei statice **paint**.



METODE IMPLICITE

- **Versiunea 2 a bibliotecii introduce metoda `setRelativeSize`**
- **Codul anterior nu mai compilează** dar încă rulează
- Pentru a păstra compatibilitatea transformăm metoda `setRelativeSize` în **metodă implicită**

```
package atm.paradigms.shapes;

public interface Resizable {
    int getWidth();
    int getHeight();
    void setWidth(int width);
    void setHeight(int height);
    void setAbsoluteSize(int width, int height);
    void draw();
    default void setRelativeSize(int wFactor, int hFactor){
        setAbsoluteSize(getWidth()/wFactor, getHeight()/hFactor);
    };
}
```

- **Codul client anterior compilează acum fără modificări**

METODE IMPLICITE

- Introducerea metodelor **implicite** în Java 8 creează **posibilitatea moștenirii a mai multor metode cu aceeași semnătură**

```
package atm.paradigms.shapes;

public interface A {
    default void hello(){
        System.out.println("Hello from A");
    }
}
```

```
package atm.paradigms.shapes;

public interface B extends A{
    default void hello(){
        System.out.println("Hello from B");
    }
}
```

```
package atm.paradigms.shapes;

public class C implements A, B {
    public static void main(String[] args) {
        new C().hello();
    }
}
```

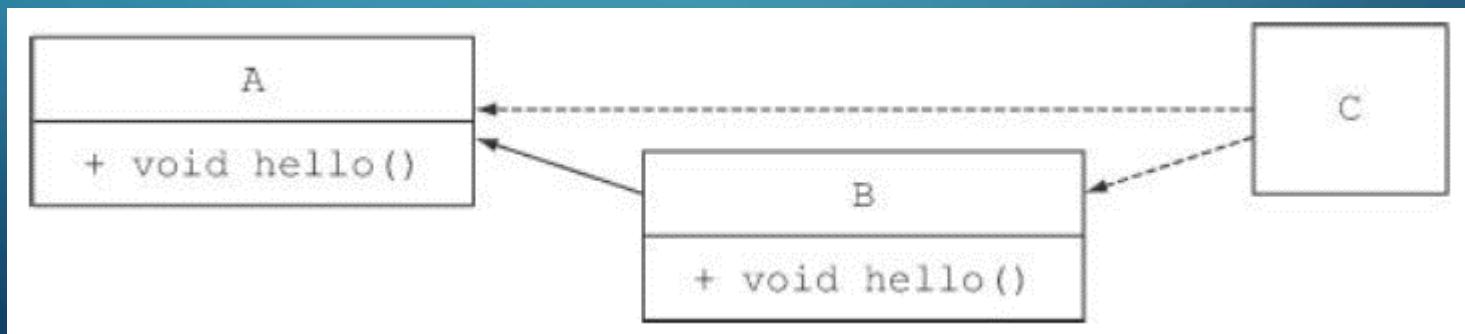
- Care metodă se execută?



METODE IMPLICITE

Reguli de rezoluție a metodelor:

- O metodă declarată în clasă sau superclasă are prioritate asupra metodelor implicate
- Metodele din subinterfețe au prioritate (B extinde A, B are prioritate fiind cea mai specifică)
- Dacă alegera este încă ambiguă se suprascrie metoda și se cheamă explicit metoda dorită





METODE IMPLICITE

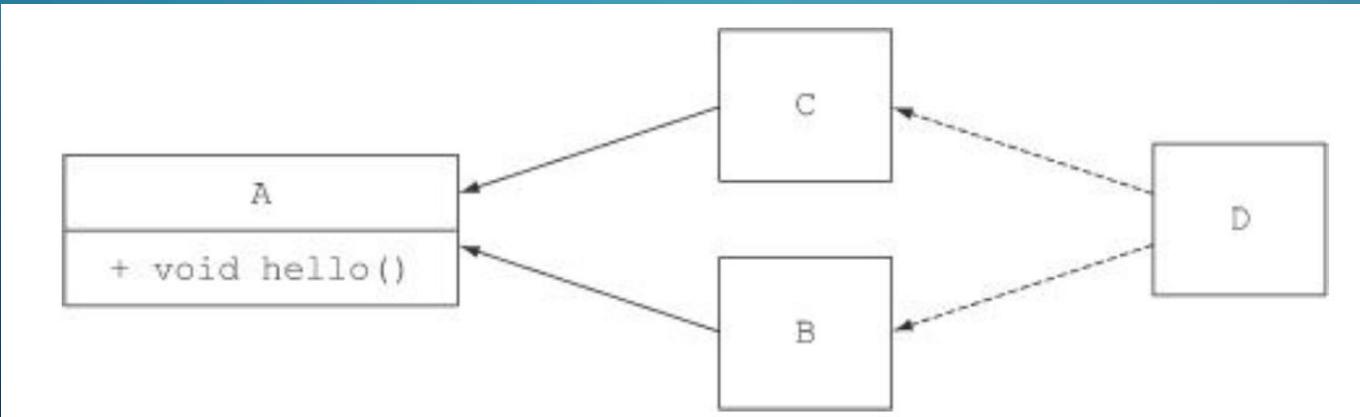
- Java 8 introduce sintaxă nouă pentru a invoca metode implicite dintr-o interfață implementată de o clasă

Interface.super.method()

```
public class C implements B, A {
    void hello(){
        B.super.hello();
    }
}
```

← | **Explicitly choosing to call the method from interface B**

- Poate să apară *diamond problem*



Dacă interfețele B și C au proprie metodă implicită `void hello()` se generează eroare de compilare. Conflictul se poate elimina cu regulile de mai sus.



OPTIONAL CA ALTERNATIVĂ LA NULL

- **NullPointerException** este o **excepție comună** printre programatorii Java
- Are la bază **existența în limbaj a referințelor nule** (null)

```
public class Insurance {
    private String name;
    public String getName() {
        return name;
    }
}
```

```
public class Car {
    private Insurance insurance;
    public Insurance getInsurance() {
        return insurance;
    }
}
```

```
public class Person {
    private Car car;
    public Car getCar() {
        return car;
    }
}
```

```
public class NullPointerDemo {
    public static void main(String[] args) {
        Person p = new Person();
        String insuranceName = p.getCar()
            .getInsurance().getName();
        System.out.println(insuranceName);
    }
}
```

Rezultat: Exception in thread "main" java.lang.NullPointerException at atm.paradigms.optional.NullPointerDemo.main(NullPointerDemo.java:6)

Metoda **getCar** va returna referință nulă (null) deoarece nu a fost inițializată. Aplicarea **getInsurance** pe ea va genera excepție.



OPTIONAL CA ALTERNATIVĂ LA NULL

- Clasa **Person** nu are atribuită o valoare pentru variabila instanței **car**.
getCar() va returna o referință nulă care la rulare va genera
NullPointerException
- O modalitate de evitare o constituie verificarea excesivă condițiilor care pot genera **NullPointerException** (chiar și când nu este nevoie)

```
package atm.paradigms.optional;

public class NullPointerDemo {
    public static void main(String[] args) {
        Person p = new Person();
        System.out.println(getCarInsuranceName(p))
    }
}
```

```
public static String getCarInsuranceName(Person p){
    if (p != null) {
        Car c = p.getCar();
        if (c != null)[
            Insurance i = c.getInsurance();
            if (i != null){
                return i.getName();
            }
        ]
    }
    return "Unknown";
}
```

OPTIONAL CA ALTERNATIVĂ LA NULL

- Soluția Java 8 este clasa `java.util.Optional<T>` (deja introdusă) care încapsulează o valoare optională. Absența valorii este modelată cu valoarea `empty` returnată de metoda `Optional.empty()`
- Cu `Optional` devine explicit că o persoană poate avea sau nu mașina, iar o mașină poate să fie sau nu asigurată

```
package atm.paradigms.optional;

import java.util.Optional;

public class Person {
    private Optional<Car> car =
        Optional.empty();
    public Optional<Car> getCar() {
        return car;
    }
}
```

- **Insurance** rămâne la fel
compania de asigurare trebuie să aibă un nume

```
package atm.paradigms.optional;

import java.util.Optional;

public class Car {
    private Optional<Insurance> insurance =
        Optional.empty();
    public Optional<Insurance> getInsurance() {
        return insurance;
    }
}
```

Reimplementarea cu `Optional` a claselor anterioare. Variabilele de instanță se initializează cu `Optional.empty()`

OPTIONAL CA ALTERNATIVĂ LA NULL

- Crearea obiectelor de tip Optional:

```
Optional<Car> optCar = Optional.empty();
```

- dintr-o **valoare nenulă**. Generează excepție dacă valoarea este nulă

```
Optional<Car> optCar = Optional.of(car);
```

- dintr-o **valoare potențial nulă**

```
Optional<Car> optCar = Optional.ofNullable(car);
```

- **Optional** suportă metoda **map** care ne permite să extragem din Optional (asemănător stream-urilor)

```
Optional<Insurance> optInsurance = Optional.ofNullable(insurance);
Optional<String> name = optInsurance.map(Insurance::getName);
```

OPTIONAL CA ALTERNATIVĂ LA NULL

- Codul de mai jos nu compilează

```
Optional<Person> optPerson = Optional.of(person);
Optional<String> name =
    optPerson.map(Person::getCar)
    .map(Car::getInsurance)
    .map(Insurance::getName);
```

- **optPeople** este de tip **Optional<People>** deci putem chama **map** pe el
- **getCar** returnează un obiect de tip **Optional<Car>**, deci metoda **map** va returna **Optional<Optional<Car>>** care nu suportă metoda **getInsurance**
- Ca și în cazul stream-urilor **soluția este metoda flatMap** care reduce 2 nivele de Optional la unul singur

OPTIONAL CA ALTERNATIVĂ LA NULL

- Exemplul anterior:

```
package atm.paradigms.optional;
import java.util.Optional;

public class NullPointerDemo {
    public static void main(String[] args) {
        Person p = new Person();
        Optional<Person> optPerson = Optional.of(p);
        String name = getCarInsuranceName(optPerson);
        System.out.println(name);
    }
    public static String getCarInsuranceName(Optional<Person> p){
        return p.flatMap(Person::getCar)
            .flatMap(Car::getInsurance)
            .map(Insurance::getName)
            .orElse("Unknown");
    }
}
```

Rezultat: Unknown

Primul **flatMap** va returna **Optional<Car>**. Al 2-lea **flatMap** va returna **Optional<Insurance>**. Se poate aplica **map** pe rezultat, iar apoi **orElse** în caz că **Optional<String>** este **empty**.

- Variabilele de instanță de tip **Optional** trebuie inițializate cu **empty** (**Optional.empty()**)



OPTIONAL CA ALTERNATIVĂ LA NULL

Metoda clasei Optional	Descriere
empty	Returnează un obiect Optional gol (empty)
filter	Dacă valoarea este prezentă și corespunde argumentului Predicate returnează acel Optional, dacă nu unul gol
flatMap	Dacă valoarea este prezentă returnează Optional cu valoarea rezultată din aplicarea funcției de mapare, altfel returnează Optional gol
map	Dacă valoarea este prezentă aplică funcția de mapare pe ea
get()	Returnează valoarea dacă există, dacă nu generează excepția <i>NoSuchElementException</i> . Se folosește doar când există o valoare
orElse(T other)	Furnizează o valoare implicită dacă Optional nu conține nici una
orElseGet(Supplier<? extends T> other)	Similar cu orElse doar că generează valoarea implicită printr-un Supplier dat ca argument
orElseThrow(Supplier<? extends X> exception)	Permite generarea unei excepții dacă Optional nu conține valoare
IfPresent(Consumer<? super T> consumer)	Se execută acțiunea dacă ca argument dacă există valoarea

OPTIONAL CA ALTERNATIVĂ LA NULL

- Metoda **filter** – primește un **Predicate ca argument**, dacă valoarea există returnează **Optional cu valoarea respectivă**, dacă nu returnează **empty**

Insurance insurance = ...;

```
if(insurance != null && "CambridgeInsurance".equals(insurance.getName())){  
    System.out.println("ok");  
}
```

- Varianta cu filter

Optional<Insurance> optInsurance = ...;

```
optInsurance.filter(insurance ->  
    „City Insurance”.equals(insurance.getName()))  
    .ifPresent(x -> System.out.println("ok"));
```



OPTIONAL CA ALTERNATIVĂ LA NULL

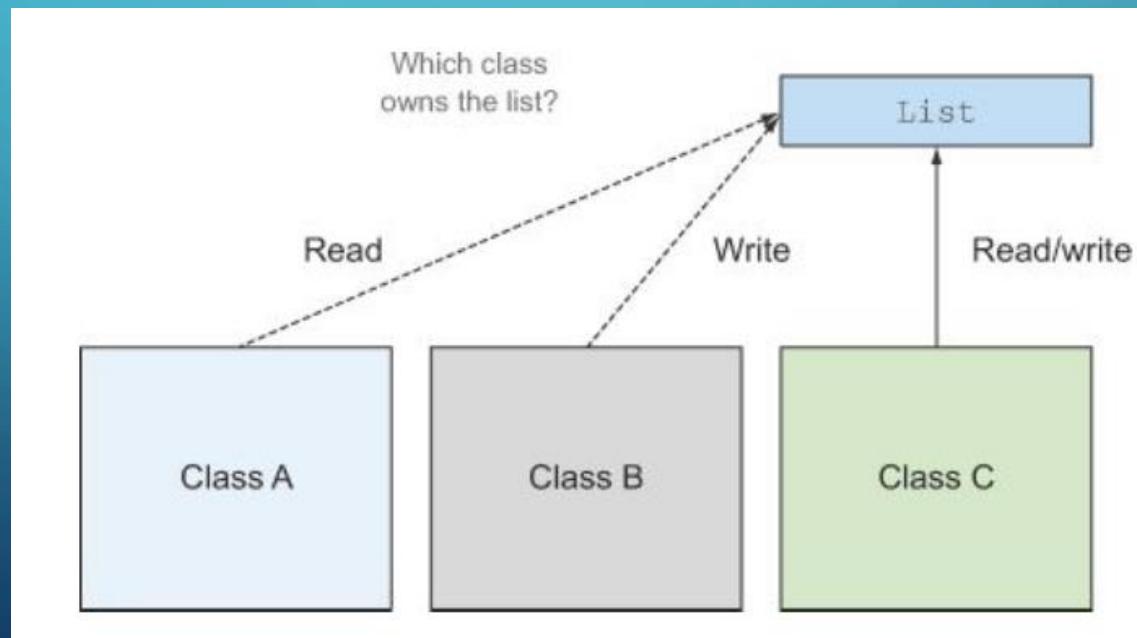
- O modalitate comună de a **evita returnarea valorii null este generarea de excepții**
- Un exemplu tipic este conversia unui **String** în **int** folosind metoda statică **Integer.parseInt(String)**. Dacă **String-ul nu poate fi convertit** generează **excepția NumberFormatException**.
- Putem modela valoarea invalidă produsă de conversia String-ului folosind **Optional**

```
public static Optional<Integer> stringToInt(String s){  
    try {  
        return Optional.of(Integer.parseInt(s));  
    } catch (NumberFormatException e){  
        return Optional.empty();  
    }  
}
```



GÂNDIRE FUNCȚIONALĂ

- Programarea funcțională promovează concepe precum **lipsa efectelor colaterale și imutabilitatea**, concepe atractive pentru programatorii care se ocupă menenanța codului scris de alții
- Motivul comun pentru care codul crapă îl reprezintă valori neașteptate pe care unele variabile le primesc
- Presupunând că **mai multe clase au referințe la o listă** (shared mutable data), este **dificilă observarea modificărilor** de către diferite componente ale codului





GÂNDIRE FUNCȚIONALĂ

- O metodă care nu modifică starea clasei care o conține, nici starea altor obiecte și returnează rezultatul folosind **return**, se numește **pură** sau **liberă de efecte colaterale**

Efectele colaterale sunt acțiuni care nu sunt total limitate la funcția însăși:

- **Modificarea structurii de date** prin atribuirea de valori unui câmp sau prin metode setters (excepție fac constructorii)
- **Generarea de excepții**
- **Operații I/O**
- Un **obiect imutabil** este un **obiect care nu își mai poate schimba starea după ce a fost instanțiat** și nu poate fi afectat de acțiunile funcției



GÂNDIRE FUNCȚIONALĂ

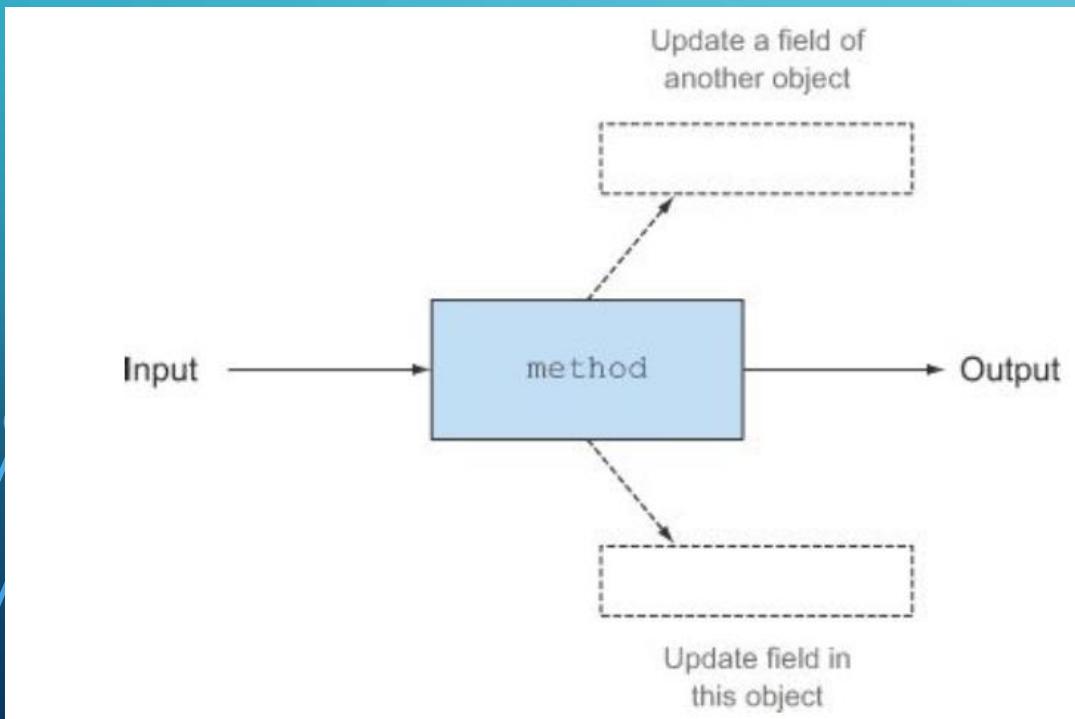
- Există **2 moduri de a scrie un program**:
- **Imperativ** – stilul de programare **cum**, se potrivește perfect cu OOP și reprezintă o secvență de comenzi apropriate de limbajul calculatorului
- **Declarativ** – stilul de programare **ce**, programatorul declară ceea ce dorește codul fiind mult mai ușor de înțeles decât o secvență de instrucțiuni
- **Programarea funcțională** combină ideea de **programare declarativă** combinat cu **conceptul de calcul fără efecte secundare**
- **Concepte prezentate anterior** precum **expresii lambda, stream-uri**, ce pot fi înlántuite în operații complexe, **constituie baza programării funcționale** în Java



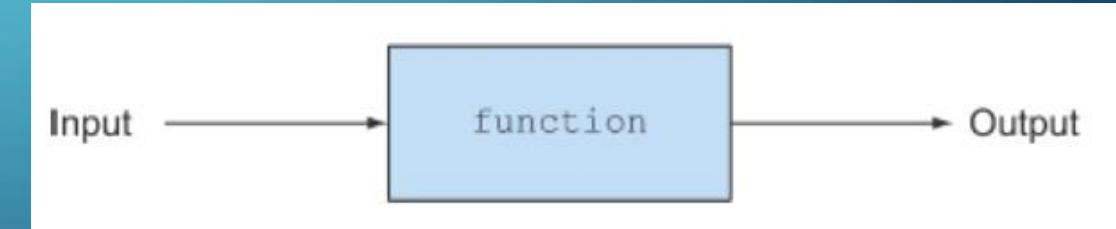
GÂNDIRE FUNCȚIONALĂ

- În contextul programării funcționale o funcție corespunde unei funcții matematice, adică primește zero sau mai multe argumente și returnează unul sau mai multe rezultate dar fără efecte colaterale

Funcție cu efecte colaterale



Funcție fără efecte colaterale





GÂNDIRE FUNCȚIONALĂ

- În programarea funcțională o funcție (sau metodă) **poate modifica doar variabilele locale**. Obiectele pe care le referențiază trebuie să fie imutabile (variabilele trebuie să fie finale)
- Pentru a avea un **cod strict funcțional** funcțiile (metodele) nu ar trebui să genereze **excepții**
- Funcția sau metoda **poate folosi biblioteci de funcții care produc efecte secundare** atât timp cât **comportamentul lor nefuncțional este ascuns local și nu propagat**
- Chiar și în programarea funcțională este nevoie să generăm informații de debug și să le scriem într-un fișier de log

GÂNDIRE FUNCȚIONALĂ

- Restricțiile de a avea efecte colaterale vizibile (fără structuri mutabile vizibile, fără I/O și fără excepții) poartă numele de **transparentă referențială**
- O funcție este **transparentă referențial** dacă returnează același rezultat când este chemată cu aceleași argumente
- “**word**”.**replace**(‘w’, ‘W’) întotdeauna returnează același rezultat ca un nou String și poate fi considerat o funcție
- Contraexemple: **Random.nextInt** sau **Scanner.nextLine**
- În Java **OOP** și **programarea funcțională sunt folosite împreună**



GÂNDIRE FUNCȚIONALĂ

- *Recursivitate sau iterare*
- Limbajele de **programare funcționale pure** nu conțin **construcții iterative precum while sau loop**
- Se consideră că produc mutații. Dar cum am stabilit putem folosi mutații atâtă timp cât nimeni nu le vede, precum actualizarea variabilelor locale

```
Iterator<Apple> it = apples.iterator();
```

```
while (it.hasNext()) {
```

```
    Apple apple = it.next();
```

```
    // ...
```

```
}
```

- Codul este acceptabil dacă este folosit într-o metodă pentru că mutațiile nu sunt folosite în afara metodei
- Soluția oferită de limbajele pur funcționale este **recursivitatea**



GÂNDIRE FUNCȚIONALĂ

- Calcul factorial:

```
public static int iterativeFactorial(int n){  
    int r = 1;  
    for (int i = 1; i <= n; i++){  
        r *= i;  
    }  
    return r;  
}
```

- Buclă standard, variabilele r și i sunt modificate la fiecare iterare

```
public static long recursiveFactorial(long n){  
    return n == 1 ? 1 : n * recursiveFactorial(n - 1);  
}
```

- **Abordare recursivă apropiată de forma matematică.** În Java recursivitatea nu este foarte eficientă
- Rularea metodei cu numere mari poate genera excepția **StackOverflowError**
- În alte limbiage există tehnici de optimizare **tail-call optimization**

GÂNDIRE FUNCȚIONALĂ

```
public static long streamsFactorial(long n){  
    return LongStream.rangeClosed(1, n)  
        .reduce(1, (a, b) -> a * b);  
}
```

- În **Java 8** recomandarea este ca **în loc de iterătie să se folosească stream-uri** pentru evitarea mutațiilor
- Mai multe **în cursul următor**

BIBLIOGRAFIE

- **Java 8 in Action**, Raoul-Gabriel Urma, Mario Fusco, Alan Mycroft, Manning Publications Co, 2015
- **Java EE 8 Design Patterns and Best Practices**, Rhuan Rocha, João Purificação, Packt Publishing, 2018
- **Java™: The Complete Reference, Tenth Edition**, Herbert Schildt, McGraw-Hill, 2005