

# Elemente de programare funcțională în Java

## Caracteristici ale programării funcționale

- Se bazează pe apeluri structurate de funcții  
 $<function1>(<function2>(<function3> \dots ) \dots )$
- Fiecare funcție primește valori de la alte funcții și transmite valori altor funcții
- Numele sunt utilizate doar ca parametrii formali
- Un nume este asociat întotdeauna cu o singură valoare
- Ordinea executării operațiunilor nu influențează rezultatele
- Modificările repetitive se realizează prin apeluri de funcții imbricate

## Interfețe funcționale

- O interfață funcțională este o interfață care conține doar o singură metodă abstractă
- O interfață funcțională poate conține una sau mai multe metode *default* sau metode statice
- Utilizare extinsă în implementarea obiectelor *Stream* sau în interfața grafică

# Interfață *Function*

*@FunctionalInterface*

*public interface Function<T,R>;*

- Reprezintă o funcție care acceptă un argument de tip *T* și întoarce un rezultat de tip *R*

Metodă abstractă:

*R apply(T t); // Conține codul asociat funcției*

Metode default:

*default <V> Function<V,R> compose(Function<? super V,? extends T> before);*  
//Întoarce o funcție compusă din funcția *before* și funcția curentă - mai întâi execută *before* și apoi rezultatul este prelucrat prin funcția curentă

*default <V> Function<T,V> andThen(Function<? super R,? extends V> after);*  
//Întoarce funcția a cărei execuție este urmată de execuția unei funcții *after*

*Exemplu. E\_Function1*

## Exemplu

1. Să se definească o funcție care calculează suma elementelor dintr-un vector de numere reale

```
Double[] a = {10.0, 20.0, 30.0};  
Function<Double[], Double> f1 = (new Function<Double[], Double>() {  
    @Override  
    public Double apply(Double[] t) {  
        double s = 0;  
        for (double v : t) {  
            s += v;  
        }  
        return s;  
    }  
});  
System.out.println(f1.apply(a));
```

2. Să se calculeze pătratul sumei prin metoda *andThen* care va implementa calculul pătratului

```
System.out.println(f1.andThen(
        new Function<Double, Double>() {
    @Override
    public Double apply(Double t) {
        return t*t;
    }
}).apply(a));
```

2. Să se genereze vectorul aleator mai întâi folosind metoda *compose* și apoi să se calculeze suma

```
System.out.println(f1.compose(new Function<Integer, Double[]>() {
    @Override
    public Double[] apply(Integer t) {
        Double[] v = new Double[t];
        for (int i = 0; i < t; i++) {
            v[i] = Math.random() * t;
        }
        return v;
    }
}).apply(10));
```

# Interfața *BiFunction*

*@FunctionalInterface*

*public interface BiFunction<T,U,R>;*

T - tipul primului argument

U - tipul celui de-al doilea argument

R - tipul rezultatului

Metoda abstractă

*R apply(T t, U u)*

Metodă default:

*default <V> BiFunction<T,U,V> andThen(Function<? super R, ? extends V> after)*

## Interfețele *UnaryOperator* și *BinaryOperator*

- Sunt extensii ale interfețelor *Function* și *BiFunction* pentru situațiile în care rezultatele sunt de același tip cu argumentele

*@FunctionalInterface*

*public interface UnaryOperator<T> extends Function<T,T>*

*@FunctionalInterface*

*public interface BinaryOperator<T> extends BiFunction<T,T,T>*

*Exemplu. E\_Function2*

*Exemplu.* Conversie în majuscule a unui sir – *UnaryOperator*. Determinarea sirului cu lungime mai mare dintre două siruri - *BinaryOperator*

```
String nume1 = "Popa Marius", nume2 = "Pop Adrian";
UnaryOperator<String> uo = new UnaryOperator<String>() {
    @Override
    public String apply(String t) {
        return t.toUpperCase();
    }
};

BinaryOperator<String> bo = new BinaryOperator<String>() {
    @Override
    public String apply(String t, String u) {
        if (t.length() > u.length()) {
            return t;
        } else {
            return u;
        }
    }
};

System.out.println(uo.apply(nume1));
System.out.println(bo.apply(nume1, nume2));
```

# Referințele la metode

- Referințele la metode permit reutilizarea unei metode ca funcție
- Sintaxa: *target\_reference::method\_name*
- Pot fi referite atât metode statice cât și metode de instanță

## *Exemplul 1*

```
Double[] a = {10.0, 20.0, 30.0};  
Function<Double[],Double> f2 = Operatiuni::produs;  
System.out.println(f2.apply(a));  
unde clasa Operatiuni are o metodă statică produs astfel definită:  
public static Double produs(Double[] s) { ... }
```

## *Exemplul 2*

```
Operatiuni o = new Operatiuni();  
Function<Integer,Double> f3 = o::produs;  
System.out.println(f3.apply(3));
```

Clasa *Operatiuni* are o metodă de instanță *produs* astfel definită:

```
public Double produs(Integer m) { ... } // Calculează produsul primelor m valori reale  
dintr-un vector câmp al clasei
```

## *Exemplu. E\_RerferinteFunctii*

## Expresiile *lambda*

- Expresiile Lambda simplifică modul de transmitere a comportamentului ca parametru
- Comportamentul transmis este cel implementat printr-o metodă a unei interfețe funcționale
- Sintaxa unei expresii lambda este:  
 $(args1, args2, \dots) \rightarrow \{ body \}$ 
  - dacă *body* constă dintr-o singură instrucțiune nu sunt necesare acoladele și nici instrucțiunea *return*;
  - dacă există un singur parametru nu sunt necesare parantezele rotunde
  - dacă nu există parametrii sunt obligatorii parantezele rotunde
  - specificarea tipului pentru argumente este optională

## Accesarea variabilelor locale din domeniul de vizibilitate

- Accesarea variabilelor locale se face în aceleasi condiții ca la clasele locale sau anonime - variabilele locale trebuie să fie finale sau efectiv finale (nu se modifică după inițializare)
- Variabilele locale nu pot fi mascate in expresiile lambda - expresiile lambda nu creează un nou nivel de domeniu de vizibilitate
- referința *this* se refera la clasa care conține expresia și nu la expresie in sine

*Exemplu. E\_Lambda*

# Referințele la metode și expresiile lambda

- Referințele la metode permit reutilizarea unei metode și ca funcție lambda

*Exemplul 1 - Pentru o metodă statică*

```
Function<Double[], Double> f4 = (Double[] ta) -> Operatiuni.produs(ta);  
System.out.println(f4.apply(a));
```

echivalent cu

```
Function<Double[], Double> f2 = Operatiuni::produs;  
System.out.println(f2.apply(a));
```

*Exemplul 2*

```
Operatiuni o = new Operatiuni();  
Function<Integer, Double> f5 = (Integer t) -> o.produs(t);  
System.out.println(f5.apply(3));
```

echivalent cu

```
Operatiuni o = new Operatiuni();  
Function<Integer, Double> f3 = o::produs;  
System.out.println(f3.apply(3));
```

# Prelucrarea funcțională a colecțiilor

# Stream

- Facilitate oferita de Java 1.8. Consta in prelucrarea prin intermediul unor obiecte **Stream** care permit diverse operatii pe elementele colectiei
- Interfata la baza:

public interface **BaseStream**<T,S extends BaseStream<T,S>> extends AutoCloseable;  
T – tipul elementului din Stream

S – tipul de stream

- Interfete derivate:

public interface **DoubleStream** extends BaseStream<Double,DoubleStream>;

public interface **IntStream** extends BaseStream<Integer,IntStream>

public interface **Stream**<T> extends BaseStream<T,Stream<T>>;

- Obtinerea stream-urilor se face prin metode ale colectiilor:

**Stream**<E> *stream()*;

**Stream**<E> *parallelStream()*;

# Interfețele **Consumer** și **Predicate**

- Sunt utilizate pentru a facilita implementarea unor operațiuni asupra elementelor colecțiilor generice în procesul de traversare/filtrare

Declarație:

public interface **Consumer**<T>;

Metodă:

void **accept**(T t); // Execută operațiunea implementată asupra obiectului *t*

Metode default:

default Consumer<T> **andThen**(Consumer<? super T> *after*); // Întoarce consumerul care efectueaza operatia curenta si apoi operatia specificata in *after*

Declarație:

public interface **Predicate**<T>;

Metodă:

boolean **test**(T t); // Execută testul asupra obiectului *t*

Metode default:

default Predicate<T> **and**(Predicate<? super T> *other*); // Execută și logic cu un alt predicat

default Predicate<T> **negate**(); // Întoarce predicatul negație

default Predicate<T> **or**(Predicate<? super T> *other*); // Sau logic

# Crearea obiectelor *Stream* prin *StreamSupport*

- Clasa *StreamSupport* permite crearea de obiecte Stream în colecțiile proprii
- Obiectele *Stream* sunt create prin metode statice *stream* ale clasei *StreamSupport* care crează obiecte de tip *DoubleStream*, *IntStream*, *LongStream* sau *Stream<T>*

## Exemplu *Stream<T>*

```
public static <T> Stream<T> stream(Spliterator<T> spliterator, boolean parallel);  
// un obiect Spliterator<T> este folosit pentru pentru partitōnarea și traversarea  
elementelor dintr-o structură-sursă (vector, colecție, flux de intrare etc.)
```

- O implementare simplificată și incompletă a interfeței **Spliterator** este clasa **Spliterators.AbstractSpliterator<T>**:

```
public abstract static class Spliterators.AbstractSpliterator<T> extends Object  
implements Spliterator<T>;
```

Constructor:

protected **AbstractSpliterator**(long est, int additionalCharacteristics);

Crează un spliterator cu o mărime estimată și cu specificarea unor caracteristici precum ORDERED, DISTINCT, SORTED, SIZED, NONNULL, IMMUTABLE, CONCURRENT, sau SUBSIZED

Metodă abstractă:

boolean *tryAdvance*(Consumer<? super T> action); // Funcționalitate asemănătoare metodelor *hasNext* și *next* din Iterator. Întoarce true dacă există elemente în spliterator, iar elementul este furnizat prin *Consumer*

*Exemplu.* Metoda stream de mai jos crează un obiect Stream într-o colecție proprie de tip listă simplu înlățuită. Referința *first* este referința la primul element din listă iar referința *t* este o referință de lucru folosită pentru accesarea elementelor din listă. (Vezi *E\_ColectiiProprii* – curs 4)

```
public class MyList<T> implements Iterable<T> {  
    private MyElement<T> first, t;  
    ...  
    public Stream<T> stream() {  
        t = first;  
        return StreamSupport.stream(new Spliterators.AbstractSpliterator<T>(0, 0) {  
            @Override  
            public boolean tryAdvance(Consumer<? super T> action) {  
                if (t == null) {  
                    return false;  
                } else {  
                    action.accept(t.getInfo());  
                    t = t.getNext();  
                    return true;  
                }  
            }  
        }, true);  
    }  
    ...
```

# Metode Stream

- Traversare

```
void forEach(Consumer<? super T> action); // executa o actiune pe fiecare element din stream
```

- Filtrare

```
Stream<T> filter(Predicate<? super T> predicate); // filtreaza datele din stream dupa filtrul creat in predicate
```

- Sortare

```
Stream<T> sorted(); // T trebuie sa fie un tip care implementeaza Comparable<T>
```

```
Stream<T> sorted(Comparator<? super T> comparator); // se implementeaza comparatorul
```

- Determinarea elementelor distincte

Stream<T> ***distinct()***;

Elementele sunt diferențiate în funcție de implementarea din *equals*

- Prelucrare personalizată a elementelor din stream

<R> Stream<R> ***map(Function<? super T,? extends R> mapper)***

unde mapper este o funcție de transformare care se aplică fiecărui element din *stream*.

### Prelucrari particulare:

DoubleStream ***mapToDouble(ToDoubleFunction<? super T> mapper)***;

IntStream ***mapToInt(ToIntFunction<? super T> mapper)***

LongStream ***mapToLong(ToLongFunction<? super T> mapper)***

*Exemplu. Maparea pe lista mediilor unor studenți aflați într-o colecție*

```
double mediaMaxima = listaStudenti.stream().mapToDouble(student->
    student.getMedia()).max().getAsDouble();
```

# Colectori

- Prin colectori se pot executa diverse procesări terminale asupra elementelor din obiecte Stream, precum: grupări, summarizări, selecții, agregări etc.
- Outputul obișnuit este o colecție rezultată prin selecție sau agregare dar și o valoare rezultată prin summarizare sau alte tipuri de operațiuni
- Colectarea se realizează prin invocarea metodelor *collect()* ale interfeței Stream
- Un collector este un obiect de tip *Collector<T,A,R>*

public interface Collector<T,A,R>

unde:

T - este tipul elementelor colectate

A - este tipul operațiunii de acumulare aplicate asupra elementelor colectate

R - este tipul rezultatului operațiunii de acumulare

- Metoda *collect()* din *Stream* are două forme:  
 $\langle R, A \rangle R \text{ collect}(\text{Collector} < ? \text{ super } T, A, R > \text{ collector})$   
 $\langle R \rangle R \text{ collect}(\text{Supplier} < R > \text{ supplier}, \text{BiConsumer} < R, ? \text{ super } T > \text{ accumulator}, \text{BiConsumer} < R, R > \text{ combiner})$
- În componența unui colector este obligatorie prezența a trei elemente:
- *initializator* sau *supplier* - crează structura asociată colectorului
- *acumulator* - actualizează structura asociată colectorului
- selector sau *combiner* - efectuează operațiuni asupra structurii asociate colectorului în vederea obținerii rezultatului final
- În mod optional colectorul poate să aibă un *finalizator* care este o funcție aplicată asupra rezultatului selecției (combiner) pentru a obține rezultatul final. Dacă nu este furnizat în mod explicit se consideră finalizator funcția identitate cea care nu modifică outputul combinerului.
- Structura colectorului este reflectată de metodele abstracte ale interfeței *Collector*

# Metodele interfeței Collector

Supplier<A> **supplier()**; //Întoarce un obiect *Supplier* de tip A. Interfața funcțională *Supplier* are o funcție *get()* prin care se întoarce un obiect A (tipul *Supplier*).

BiConsumer<A,T> **accumulator()**; //Întoarce un obiect *BiConsumer*. Interfața funcțională are o metodă void *accept(A t, T u)* prin care este implementat "consumul" obiectului de tip T.

BinaryOperator<A> **combiner()**; // Întoarce un obiect *BinaryOperator*. Înterfața *BinaryOperator* are metoda funcțională *A apply(A t, A u)* care întoarce rezultatul aplicării colectorului

Function<A,R> **finisher()**; // Întoarce un finalizator de tip *Function* al colectorului.

Metoda funcțională furnizează rezultatul colectorului: *R apply(A t)*

Set<Collector.Characteristics> **characteristics()**; //Furnizează caracteristicile colectorului sub formă de Set. Acestea pot fi constante din enumerarea *Collector.Characteristics*, precum: CONCURRENT (colectarea în regim concurențial), UNORDERED (în colectare nu contează ordinea) sau IDENTITY\_FINISH (funcția de finalizare este ignorată, combiner-ul este cel care furnizează rezultatul colectării).

## Exemplu creare și utilizare colector personalizat

Fie clasele Persoana și Student astfel definite:

```
public class Persoana {  
    private String nume;  
    private long cnp;  
...  
}
```

```
public class Student extends Persoana  
{  
    private int grupa,anul;  
    private Map<String,Integer>  
        note=new HashMap<>();  
...  
}
```

Să se creeze lista cu cnp-urile studentilor pornind de la lista de studenti.

Va fi invocată metoda *collect()* din Stream în forma:

<R,A> R collect(Collector<? super T,A,R> collector);

În codul de mai jos *lista* este un obiect ArrayList<Student> cu lista studentilor.

```
List<Long> cnp = lista.stream().map(Student::getCnp).collect(  
    new Collector<Long, List<Long>, List<Long>>() {  
        public Supplier<List<Long>> supplier() {  
            Supplier<List<Long>> s = () -> new ArrayList<>();  
            return s;  
        }  
        public BiConsumer<List<Long>, Long> accumulator() {  
            BiConsumer<List<Long>, Long> bc = (lst, val) -> { lst.add(val); };  
            return bc;  
        }  
        public BinaryOperator<List<Long>> combiner() {  
            BinaryOperator<List<Long>> bo = (t, u) -> { u.addAll(t); return u; };  
            return bo;  
        }  
        public Function<List<Long>, List<Long>> finisher() {  
            Function<List<Long>,List<Long>> f = (List<Long> t) -> t;  
            return f;  
        }  
        public Set<Collector.Characteristics> characteristics() {  
            Set<Collector.Characteristics> ch = new HashSet<>();  
            ch.add(Characteristics.UNORDERED);ch.add(Characteristics.CONCURRENT);  
            return ch;  
        }  
    });
```

- Același rezultat se poate obține prin invocarea celei de a doua forme a metodei `collect()`. Pe exemplul precedent, în codul de mai jos sunt prezentate colectările pentru grupele și anii distincti din care fac parte studenții din listă:

```
List<Integer> ani=lista.stream().map(Student::getAnul).distinct().  
        collect(  
            () -> new ArrayList<>(),  
            (t,u) -> {t.add(u);},  
            (t,u) -> { u.addAll(t);});
```

```
List<Integer> grupe = lista.stream().map(Student::getGrupa)  
        .distinct()  
        .collect(ArrayList::new, ArrayList::add,  
                 ArrayList::addAll);
```

- Finalizatorul va fi implicit funcția identitate
- Caracteristicile colectorului vor fi *CONCURRENT* și *UNORDERED*
- În prima variantă metodele funcționale sunt pasate prin funcții lambda
- În varianta a doua metodele funcționale sunt pasate prin operatorul de referențiere

## Colectori standard

- Sunt creați prin metode statice ale clasei utilizare *Collectors* în contextul efectuării a diverse operațiuni de colectare
- Colectarea în colecții standard:

```
public static <T> Collector<T,?,List<T>> toList();
```

```
public static <T,C extends Collection<T>> Collector<T,?,C> toCollection(Supplier<C> collectionFactory);
```

```
public static <T> Collector<T,?,Set<T>> toSet();
```

```
public static <T,K,U,M extends Map<K,U>> Collector<T,?,M> toMap(  
    Function<? super T,? extends K> keyMapper,  
    Function<? super T,? extends U> valueMapper)
```

unde *keyMapper* este o funcție care furnizează cheile iar *valueMapper* o funcție care furnizează valorile. Această formă a metodei *toMap* presupune inexistența cheilor după.

*Exemplu.* Colectarea numelor studenților din lista de studenți:

```
List<String> nume =  
lista.stream().map(Student::getNume).collect(Collectors.toList());
```

*Exemplu. Colectarea numelor studenților după cnp (structură Map):*

```
Map<Long, String> nume =  
lista.stream().collect(Collectors.toMap(Student::getCnp,  
Student::getNume));
```

```
public static <T,K,U,M extends Map<K,U>> Collector<T,?,M> toMap(  
Function<? super T,? extends K> keyMapper,  
Function<? super T,? extends U> valueMapper,  
BinaryOperator<U> mergeFunction);
```

*mergeFunction* este o funcție care indică modul de tratare a valorilor avute pentru cheile duplicate

*Exemplu. Listarea studenților pe grupe:*

```
Map<Integer, String> grupe = lista.stream().collect(Collectors.toMap(  
Student::getGrupa,  
Student::getNume,  
(nume1,nume2)->nume1+" , "+nume2));
```

A treia formă a metodei *toMap()* include un ultim parametru inițializator (supplier) pentru a inițializa obiectul *Map* rezultat într-o manieră explicită.

- Sumarizări

```
public static <T> Collector<T,?,Double> summingDouble(ToDoubleFunction<? super T> mapper); // Sumă de tip double. Idem pentru long și int.
```

```
public static <T> Collector<T,?,Double> averagingDouble(ToDoubleFunction<? super T> mapper); //Medie de tip double. Idem pentru long și int.
```

```
public static <T> Collector<T,?,Long> counting(); //Contorizare
```

```
public static <T> Collector<T,?,Optional<T>> minBy(Comparator<? super T> comparator); // Minim
```

```
public static <T> Collector<T,?,Optional<T>> maxBy(Comparator<? super T> comparator); //Maxim
```

*Exemplu.* Calcul medie generală și media maximă pe lista de studenți

```
double mediaGenerala = lista.stream().collect(  
Collectors.averagingDouble(stud -> stud.calculMedie()));
```

```
Student studMax = lista.stream().collect(Collectors.maxBy((stud1,stud2)->  
(stud1.calculMedie()>stud2.calculMedie())?1:-1)).get();
```

- Grupări și summarizări

public static <T,K> Collector<T,?,Map<K,List<T>>> groupingBy(Function<? super T,? extends K> *classifier*); // **Grupare** după funcția de clasificare specificată

*Exemplu.* Gruparea studenților după grupa din care fac parte

```
Map<Integer, List<Student>> grupe =  
lista.stream().collect(Collectors.groupingBy(Student::getGrupa));
```

public static <T,K,A,D> Collector<T,?,Map<K,D>> groupingBy(Function<? super T,? extends K> *classifier*, Collector<? super T,A,D> *downstream*); //**Grupare cu summarizare** conform colectorului *downstream*

*Exemplul 1.* Contorizarea numărului de studenți pe grupe

```
Map<Integer, Long> grupe = lista.stream().collect(Collectors.groupingBy(  
Student::getGrupa, Collectors.counting()));
```

*Exemplul 2.* Calculul mediei studenților pe grupe

```
Map<Integer, Double> grupe = lista.stream().collect(Collectors.groupingBy(  
Student::getGrupa, Collectors.averagingDouble(Student::calculMedie)));
```

unde *Student::calculMedie* este referențierea la metoda de instanță *calculMedie()* din clasa *Student*

- **Partitionarea.** Este un caz particular al grupării. Împărțirea se face în două grupe identificate prin cheile *true* și *false*

```
public static <T> Collector<T,?,Map<Boolean,List<T>>> partitioningBy(Predicate<? super T> predicate); //Colectare (partitionare) într-o colecție Map cu două chei - true, false - după predicatul specificat
```

*Exemplu.* Partiționarea listei de studenți în promovați și nepromovați în funcție de note

```
Map<Boolean,List<Student>> promovabilitate =  
lista.stream().collect(Collectors.partitioningBy(  
    stud -> stud.getNote().values().stream().allMatch((nota) -> (nota>=5))));
```

- Partiționare cu summarizare

```
public static <T,D,A> Collector<T,?,Map<Boolean,D>>> partitioningBy(Predicate<? super T> predicate, Collector<? super T,A,D> downstream); // Colectorul downstream este folosit pentru summarizare
```

*Exemplu.* Contorizare studenți promovați și nepromovați

```
Map<Boolean,Long> promovabilitate =  
lista.stream().collect(Collectors.partitioningBy(  
    stud -> stud.getNote().values().stream().allMatch((nota) -> (nota>=5)),  
    Collectors.counting()));
```