PROGRAMARE FUNCȚIONALĂ ÎN LIMBAJUL JAVA

Programarea funcțională este o paradigmă de programare creată de John Backus în 1977 ca o alternativă declarativă pentru programarea imperativă utilizată în momentul respectiv. Practic, în *programarea imperativă* un algoritm se implementează utilizând instrucțiuni pentru a descrie detaliat fiecare pas care trebuie efectuat, în timp ce în *programarea declarativă* este specificată doar logica algoritmului, fără a intra în detalii de implementare. De exemplu, folosind cele două paradigme de programare, suma numerelor naturale cuprinse între 1 și un n dat se poate calcula astfel:

Programare imperativă	Programare declarativă/funcțională

`int s = 0;`
`for(int i = 1; i <= n; i++)`
`  s = s + i;`
`System.out.println(s);`
	`int s = IntStream.rangeClosed(1, n).sum();`
	`System.out.println(s);`

Chiar dacă nu aveți încă suficiente cunoștințe pentru a înțelege în detaliu exemplul dat pentru programarea declarativă, se poate intui destul de ușor modul în care va fi executată secvența de cod respectivă: metoda `rangeClosed` va furniza numerele de la 1 la n (sub forma unui flux asemănător celor de intrare de la fișiere), după care metoda `sum` le va aduna. Foarte simplu, nu? Totuși, trebuie să menționăm faptul că metodele "magice" `rangeClosed` și `sum` sunt implementate folosind programarea imperativă…

Programarea funcțională este o paradigmă de programare de tip declarativ, bazată pe *lambda calculul* creat de Alonzo Church în anul 1936, în care funcțiile și proprietățile acestora sunt utilizate pentru a construi un program, fără a utiliza instrucțiuni de control. Totuși, dacă nu se pot evita anumite operații cu caracter iterativ, în programarea funcțională se preferă implementarea lor într-o manieră recursivă. Astfel, executarea unui program constă în evaluarea unor funcții într-o anumită ordine, într-un mod asemănător operației de compunere. De exemplu, expresia `rangeClosed(1,n).sum()` este chiar compunerea funcției `sum` cu funcția `rangeClosed(1,n)`, ceea ce matematic s-ar scrie `sum(rangeClosed(1,n))`. Mai mult, putem afișa direct rezultatul compunând 3 funcții: `System.out.println(sum(rangeClosed(1,n)))`.

Principiile programării funcționale pot fi implementate într-un limbaj de programare dacă acesta îndeplinește următoarele condiții:

- se pot defini și manipula ușor funcții complexe, care primesc funcții ca parametri sau returnează funcții ca rezultate;
- apelarea de mai multe ori a unei funcții cu aceleași valori ale parametrilor va furniza același rezultat (de exemplu, o metodă `int suma(int x) {return x + this.salariu;}` va returna valori diferite la două apeluri `suma(1000)` dacă între ele valoarea datei membre `salariu` a obiectului curent este modificată);

- apelarea unei funcții nu produce efecte colaterale, adică nu sunt modificate variabile externe și nu se modifică valorile parametrilor funcției (de exemplu, prin apelarea metodei `void suma(int x) { this.salariu = this.salariu + x;}` se vor produce efecte colaterale, deoarece se va modifica valoarea datei membre `salariu` a obiectului curent), deci se recomandă utilizarea obiectelor imutabile și transmiterea parametrilor unei metode prin valoare.

Lambda expresii

Evident, principalul impediment pentru implementarea programării funcționale în limbajul Java l-a constituit faptul că nu se pot defini și manipula ușor funcții complexe, deoarece în limbajul Java nu se pot defini funcții independente (ci doar metode în cadrul unor clase sau metode default în cadrul unor interfețe) și, în plus, utilizarea funcțiilor ca parametri ai unor metode sau ca rezultate furnizate de metode se realizează prin intermediul unui mecanism complicat (vezi exemplul de calcul al unei sume generice din cursul dedicat interfețelor). Astfel, pentru a permite implementarea unor concepte din programarea funcțională în limbajul Java, în versiunea 8 apărută în anul 2014, au fost introduse *lambda expresiile*.

O *lambda expresie* este o funcție anonimă care nu aparține niciunei clase. O lambda expresie are următoarea sintaxă:

`(lista parametrilor) -> {expresie` sau `instrucțiuni}`

Se observă faptul că pentru o lambda expresie nu se precizează tipul rezultatului returnat, acesta fiind dedus automat de compilator!

Exemple:

```
(int a, int b) -> a+b
(int a, int b) -> {return a+b;}
```

Definirea unei lambda expresii se realizează ținând cont de următoarele reguli de sintaxă:

- lista parametrilor poate fi vidă:

```
() -> System.out.println("Hello lambdas!")
```

- tipul unui parametru poate fi indicat explicit sau poate fi ignorant, fiind dedus din context:

```
(a, b) -> {return a+b;}
```

- dacă lambda expresia are un singur parametru fără tip, atunci se pot omite parantezele:

```
a -> {return a*a;}
```

- dacă lambda expresia nu conține instrucțiuni, ci doar o expresie, atunci acoladele și instrucțiunea `return` pot fi omise:

```
a -> a*a
(a, b) -> a+b
(x, y) -> {if(x>y) return x; else return y;}
```

Utilitatea lambda expresiilor

În cursul dedicat interfețelor, am văzut cum putem să transmitem o metodă ca parametru al altei metode, folosind o interfață în cadrul mecanismului de callback. De regulă, interfața respectivă conține o singură metodă, cea pe care dorim să o transmitem ca parametru, și poate să fie implementată diferit, în funcție de context.

O *interfață funcțională* este o interfață care conține o singură metodă abstractă.

Astfel, putem să definim mai multe clase care vor implementa câte o variantă a metodei respective, anonime sau nu, iar instanțele lor vor fi manipulate printr-o referință de tipul interfeței

Exemplul 1:
Reluăm, pe scurt, exemplul de calcul al unei sume generice din cursul dedicat interfețelor:

- Definim interfața funcțională `FuncțieGenerică`:

```
public interface FuncțieGenerică{
    int funcție(int x);
}
```

- În clasa utilitară `Suma` definim o metodă care să calculeze suma celor n termeni generici:

```
public class Suma{
    private Suma(){ }

    public static int CalculeazăSuma(FuncțieGenerică fg , int n){
        .........
    }
}
```

- Definim clase care implementează interfața respectivă, oferind implementări concrete ale funcției generice:

```
public class TermenGeneral_1 implements FuncțieGenerică{
    @Override
    public int funcție(int x){ return x; }
}

public class TermenGeneral_2 implements FuncțieGenerică{
    @Override
    public int funcție(int x){ return x * x; }
}
```

- La apel, metoda `CalculeazăSuma` va primi o referință de tipul interfeței, dar spre un obiect de tipul clasei care implementează interfața:

```
FuncțieGenerică tgen_1 = new TermenGeneral_1();
int S_1 = Suma.CalculeazăSuma(tgen_1, 10);
```

```
//putem utiliza direct un obiect anonim
int S_2 = Suma.CalculeazăSuma(new TermenGeneral_2(), 10);

//putem utiliza o clasă anonimă
int S_3 = Suma.CalculeazăSuma(new FuncțieGenerică() {
            public int funcție(int x) {
                return (int) Math.tan(x);
            }
        }, 10);
```

Observați faptul că este necesară definirea unei clase pentru fiecare implementare concretă a funcției generice, iar utilizarea clasei anonime conduce la un cod destul de greu de urmărit, mai ales dacă metoda ar fi fost una mai complexă. O soluție mult mai elegantă o constituie utilizarea unor lambda expresii:

```
FunctieGenerica f = x -> x;
int S_1 = Suma.CalculeazaSuma(f, 10);
int S_2 = Suma.CalculeazaSuma(x -> 1/x, 10);
int S_3 = Suma.CalculeazaSuma(x -> Math.tan(x), 10);
```

Astfel, utilizând lambda expresii, codul devine mai scurt (nu mai este necesară definirea claselor `TermenGeneral_1` și `TermenGeneral_2`), mai ușor de urmărit și, foarte important, mult mai ușor scalabil (dacă dorim să calculăm o altă sumă, nu vom fi obligați să mai definim o altă clasă, ci doar vom utiliza altă lambda expresie).

Exemplul 2:
Interfața `Comparator` este o interfață funcțională, utilizată pentru sortarea colecțiilor de obiecte, care conține o singură metodă abstractă:

```
int compare(Object ob1, Object ob2)
```

De exemplu, pentru a sorta o listă `tp` cu elemente de tip `Persoana(nume, vârstă, salariu)` putem să procedăm în mai multe moduri:

- **Soluția "clasică", folosind o clasă anonimă:**

```
Arrays.sort(tp, new Comparator(){
                 public int compare(Object p1 , Object p2) {
                           return ((Persoana)p1).getNume().
                                    compareTo(((Persoana)p2).getNume());
                 }
 });
```

- **Soluții cu lambda expresii:**

```
Arrays.sort(tp, (Object p1, Object p2) -> ((Persoana)p1).getNume().
                                    compareTo(((Persoana)p2).getNume()));

Arrays.sort(tp, (p1, p2) -> ((Persoana)p1).getNume().
                                  compareTo(((Persoana)p2).getNume()));

Arrays.sort(tp, (p1, p2) -> p1.getNume().compareTo(p2.getNume()));
```

Descriptori

O lambda expresie nu este de sine stătătoare, ci ea trebuie apelată într-un context care implică o interfață funcțională. Practic, signatura metodei din interfață precizează forma lambda expresiei.

Exemplu:
Considerăm următoarea interfață funcțională:

```
public interface calculSuma{
      long suma(int a, int b);
}
```

Se observă faptul că interfața poate fi asociată cu o lambda expresie de forma `(int,int) -> long`.

În API-ul din Java 8, în pachetul `java.util.function`, au fost introduse mai multe interfețe funcționale numite *descriptori funcționali* pentru a descrie signatura metodei abstracte dintr-o interfață funcțională, deci, implicit, și forma unei lambda expresii care poate fi utilizată pentru a implementa respectiva metodă abstractă.

Principalele interfețe funcționale definite în acest pachet sunt:

- **`Predicate<T>`** – descrie o metodă cu un argument generic de tip `T` care returnează `true` sau `false` (un predicat).

  Interfața conține metoda abstractă **`boolean test(T ob)`** care evaluează predicatul definit prin lambda expresie.

  Exemplu:
  Pentru a afișa persoanele din tabloul `tp` care au cel puțin 30 de ani, definim un predicat `criteriu` corespunzător și apoi îl aplicăm asupra fiecărui element din tablou pentru a verifica dacă îndeplinește condiția cerută:

  ```
  Predicate<Persoana> criteriu = pers -> pers.getVarsta() >= 30;
  for (Persoana p : tp)
     if (criteriu.test(p))
           System.out.println(p);
  ```

  Folosind un predicat, se poate parametriza foarte ușor o metodă care să afișeze persoanele dintr-un tablou care îndeplinesc un anumit criteriu:

  ```
  static void afisare(Persoana[] tp, Predicate<Persoana> criteriu) {
     for(Persoana p : tp)
           if(criteriu.test(p))
                 System.out.println(p);
  }

  afisare(tp , criteriu);
  ```

În plus, interfața `Predicate` conține și câteva metode default corespunzătoare operatorilor logici `and`, `or` și `negate`.

Exemplu:
Definim o metodă parametrizată pentru afișarea persoanelor dintr-un tablou care îndeplinesc simultan două criterii:

```
static void afisare(Persoana[] tp, Predicate<Persoana> criteriu_1,
                                   Predicate<Persoana> criteriu_2){
   for(Persoana p : tp)
        if(criteriu_1.and(criteriu_2).test(p))
             System.out.println(p);
}
```

Definim două predicate corespunzătoare celor două criterii și apelăm metoda `afisare`:

```
Predicate<Persoana> pred_1 = pers -> pers.getVarsta() >= 30;
Predicate<Persoana> pred_2 = pers -> pers.getNume().startsWith("P");

afisare(tp, pred_1, pred_2);
```

De asemenea, putem să apelăm direct metoda `afisare`, fără a mai defini separat cele două predicate:

```
afisare(tp, p -> p.getVarsta() >= 20, p -> p.getNume().startsWith("P"));
```

- **`Consumer<T>`** – descrie o metodă cu un argument de tip `T` care nu returnează nimic (un consumator, deoarece doar consumă parametrul).

Interfața conține metoda abstractă **`void accept(T ob)`** care efectuează acțiunea indicată prin lambda expresie.

Exemplu:
Definim o metodă parametrizată pentru a efectua o anumită acțiune asupra persoanelor dintr-un tablou care îndeplinesc un anumit criteriu:

```
static void afisare(Persoana[] persoane, Predicate<Persoana> criteriu,
                                         Consumer<Persoana> prelucrare) {
   for(Persoana p:persoane)
        if(criteriu.test(p))
             prelucrare.accept(p);
}
```

Definim un criteriu sub forma unui predicat și acțiunea de afișare a numelui persoanei folosind un obiect de tip `Consumer`:

```
Predicate<Persoana> criteriu = pers -> pers.getVarsta() >= 30;
Consumer<Persoana> actiune = pers -> System.out.println(pers.getNume());
```

```
afisare(tp, criteriu, actiune);
```

În plus, interfața `Consumer` conține și metoda default `andThen` care permite efectuarea secvențială a mai multor prelucrări.

Exemplu:
Sortăm persoanele din tablou în ordinea crescătoare a vârstelor și apoi le afișăm:

```
Consumer<Persoana[]> sortare = tablou -> Arrays.sort(tablou,
                          (p1, p2) -> p1.getVarsta() - p2.getVarsta());

Consumer<Persoana[]> afisare = tablou -> {
   for (Persoana aux : tablou)
                System.out.println(aux);
};

sortare.andThen(afisare).accept(tp);
```

- **`Function<T,R>`** – descrie o metodă cu un argument de tip `T` care returnează o valoare de tip `R` (o funcție de tipul $f: T \to R$).

Interfața conține metoda abstractă **`R apply(T ob)`** care returnează rezultatul obținut prin aplicarea operației indicate prin lambda expresie asupra obiectului curent.

Exemplu:

Definim o funcție care calculează cât ar deveni salariul unei persoane după o majorare cu 20%:

```
Function<Persoana , Double> marire = pers -> pers.getSalariu() * 1.2;

for(Persoana crt:tp)
   System.out.println(crt.getNume() + " " + marire.apply(crt));
```

În plus, interfața `Function` conține și metodele default `andThen` și `compose` care permit efectuarea secvențială a mai multor prelucrări.

Exemplu:
Definim funcțiile $f(x) = x^2$ și $g(x) = 2x$, după care calculăm $(f \circ g)(x)$ și $(g \circ f)(x)$ în mai multe moduri:

```
Function<Integer,Integer> f = x -> x*x;
Function<Integer,Integer> g = x -> 2*x;

System.out.println("f ∘ g = " + f.compose(g).apply(2)); //va afișa 16
System.out.println("f ∘ g = " + g.andThen(f).apply(2)); //va afișa 16
System.out.println("g ∘ f = " + g.compose(f).apply(2)); //va afișa 8
System.out.println("g ∘ f = " + f.andThen(g).apply(2)); //va afișa 8
```

- **`Supplier<R>`** – descrie o metodă fără argumente care returnează o valoare de tip `R` (un furnizor).

  Interfața conține metoda abstractă **`R get()`** care returnează rezultatul obținut prin aplicarea operației indicate prin lambda expresie.

```
Supplier<Persoana> furnizor = () -> new Persoana("",0,0.0);
Persoana p = furnizor.get();
```

  Acest tip de metodă este utilizat, de obicei, în cadrul claselor de tip factory.

În afară celor 4 descriptori funcționali fundamentali de mai sus, în pachetul `java.util.function` mai sunt definiți și alți descriptori funcționali suplimentari, obținuți fie prin particularizarea celor fundamentali, fie prin extinderea lor:

- *funcții cu două argumente* (unul de tipul generic `T` și unul de tipul generic `U`): `BiPredicate<T,U>`, `BiFunction<T,U,R>` și `BiConsumer<T,U>`

- *funcții specializate:*
  - `IntPredicate`, `IntConsumer`, `IntSupplier`: descriu un predicat, un consumator și un furnizor cu un argument de tip `int` (sunt definite în mod asemănător și pentru alte tipuri de date primitive);
  - `IntFunction<R>`, `LongFunction<R>`, `DoubleFunction<R>`: descriu funcții având un parametru de tipul indicat în numele descriptorului, iar rezultatul este de tipul generic `R`;

  - `ToIntFunction<T>`, `ToLongFunction<T>`, `ToDoubleFunction<T>`: descriu funcții având un parametru de tipul generic `T`, iar rezultatul este de tipul indicat în numele descriptorului;

  - `DoubleToIntFunction`, `DoubleToLongFunction`, `IntToDoubleFunction`, `IntToLongFunction`, `LongToIntFunction`, `LongToDoubleFunction`: descriu funcții care au tipul argumentului și tipul rezultatului indicate în numele descriptorului

- *operatori:*
  - `interface UnaryOperator<T> extends Function<T,T>`: descrie un operator unar, adică o funcție cu un parametru de tipul generic `T` care întoarce un rezultat tot de tip `T`;

    Exemplu:
```
UnaryOperator<Integer> sqr = x -> x*x;
System.out.println(sqr.apply(4));  //va afișa 16
```

  - `public interface BinaryOperator<T> extends BiFunction<T,T,T>`: descrie un operator binar

    Exemplu:
```
BinaryOperator<Integer> suma = (x, y) -> x + y;
System.out.println(suma.apply(4, 5));  //va afișa 9
```

Referințe către metode

Referințele către metode pot fi utilizate în locul lambda expresiilor care conțin doar apelul standard al unei anumite metode.

Exemplu:
Următoarea lambda expresie afișează șirul de caractere primit ca parametru

```
Consumer<String> c = s -> System.out.println(s);
```

și poate fi rescrisă folosind o referință spre metoda `println` astfel:

```
Consumer<String> c = System.out::println;
```

Practic, metoda `println` este referită direct prin numele său, argumentul său fiind dedus în mod automat din apelul de forma `c.accept(un șir de caractere).!`

În funcție de context, există următoarele 4 tipuri de referințe către metode:

- *referință către o metodă statică*: lambda expresia **`(args) -> Class.staticMethod(args)`** este echivalentă cu **`Class::staticMethod`.**

  Exemplu: lambda expresia `Function<Double,Double> sinus = x -> Math.sin(x)` este echivalentă cu `Function<Double,Double> sinus = Math::sin`.

- *referință către o metodă de instanță a unui obiect arbitrar*: lambda expresia **`(obj, args) -> obj.instanceMethod(args)`** este echivalentă cu **`ObjectClass::instanceMethod`**.

  Exemplu: lambda expresia `BiFunction<String,Integer,String> subsir = (a,b) -> a.substring(b)` este echivalentă cu `BiFunction<String,Integer,String> subsir = String::substring`.

- *referință către o metodă de instanță a unui obiect particular*: lambda expresia **`(args) -> obj.instanceMethod(args)`** este echivalentă cu **obj::instanceMethod**. Atenție, în acest caz obiectul particular `obj` trebuie să existe și să fie accesibil din lambda expresie! O lambda expresie poate accesa și variabile locale, dar acestea trebuie să fie efectiv finale, adică fie sunt declarate cu final, fie nu sunt declarate cu final, dar sunt inițializate și apoi nu mai sunt modificate!

  Exemplu: considerând obiectul `Persoana p = new Persoana("Ionescu Ion", 35, 1500.5)`, lambda expresia `Supplier<String> numep = () -> p.getNume()` este echivalentă cu `Supplier<String> numep = p::getNume`.

- *referință către un constructor*: lambda expresia **`(args) -> new Class(args)`** este echivalentă cu **`Class::new`**.

  Exemplu: lambda expresia `Supplier<Persoana> pnoua = () -> new Persoana()` este echivalentă cu `Supplier<Persoana> pnoua = Persoana::new`.

Metoda `forEach`

În interfața `Iterable`, existentă în limbajul Java încă din versiunea 1.5, a fost adăugată în versiunea 8 o nouă metodă denumită `forEach` care permite parcurgerea unei structuri de date. Implementarea implicită a acestei metode este următoarea:

```
default void forEach(Consumer<? super T> action) {
      for (T t : this)
            action.accept(t);
}
```

Practic, metoda `forEach` reprezintă o nouă modalitate de a parcurge o colecție, folosind lambda expresiile sau referințele spre metode.

Exemplu:
Considerăm o listă care conține numele unor orașe și prezentăm mai multe modalități de parcurgere a sa:

```
ArrayList<String> listaOrase = new ArrayList<>(Arrays.asList("București",
                                    "Paris", "Londra", "Berlin", "Roma"));

//accesând direct fiecare element
for (int i = 0; i < listaOrase.size(); i++)
      System.out.println(listaOrase.get(i));

//folosind un iterator
Iterator it = listaOrase.iterator();
while (it.hasNext())
      System.out.println(it.next());

//folosind instrucțiunea enhanced for
for (String oras : listaOrase)
      System.out.println(oras);

//folosind metoda forEach și lambda expresii
listaOrase.forEach((oras) -> System.out.println(oras + " "));

//folosind metoda forEach și referințe spre metode
listaOrase.forEach(System.out::println);
```