METODE AVANSATE DE PROGRAMARE

Conf.univ.dr. Ana Cristina DĂSCĂLESCU

Temtică curs 11

- Programarea funcțională

- Interfețe descriptor

Programarea funcțională

- 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!!!

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);

Programarea funcțională

- ***Programarea funcțională*** este o paradigmă de programare de tip declarativ, bazată pe *lambda calculul* (Alonzo Church, 1936), în care funcțiile și proprietățile acestora sunt utilizate pentru a construi un program, fără a utiliza instrucțiuni de control!!!

- Executarea unui program constă în evaluarea unor funcții într-o anumită ordine, într-un mod asemănător operației de compunere.

  - `rangeClosed(1,n).sum()` → `sum(rangeClosed(1,n))`
  - `System.out.println(sum(rangeClosed(1,n))).`

Programarea funcțională

- 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),

Lambda expresii

➢ **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) -> {return a+b;}`

Lambda expresii

➢ **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;}
```

Lambda expresii

- 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

- Implementarea mecanismului callback pentru interfețe funcționale.

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

- 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.

Lambda expresii

➢ 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`.

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

Descriptori

- În API-ul din Java 8, în pachetul **`java.util.function`**, au fost introduse mai multe interfețe funcționale numite ***descriptori funcționali***

- Rolul unui descriptor este acela de 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
  - Consumer
  - Function
  - Supplier

Descriptori

- **`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:**

✓ Afișarea persoanelor din tabloul `tp` care au cel puțin 30 de ani

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

➢ Observații

- Folosind un predicat, se poate parametriza foarte ușor o metodă!

```
void afisare(Persoana[] tp, Predicate<Persoana> criteriu)
```

- Interfața `Predicate` conține metode default corespunzătoare operatorilor logici **`and`**, **`or`** și **`negate`**.

```
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);
}
```

Descriptori

- **`Consumer<T>`** – descrie o metodă cu un argument de tip `T` care nu returnează

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

Exemplu:

- 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);}}
Consumer<Persoana> actiune = pers ->System.out.println(pers.getNume());
```

Descriptori

- Interfața **Consumer** conține 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);
```

Descriptori

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

- 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));
```

Descriptori

- 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
```

Descriptori

- **`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();
```

Descriptori

- 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, cu un argument de tip `int` (sunt definite în mod asemănător și pentru alte tipuri de date primitive);

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

Descriptori

- ***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;
c.accept(un șir de caractere).!
```

Referințe către metode

- ***referință către o metodă statică*: lambda expresia**

`(args)->Class.staticMethod(args)`

`Class::staticMethod`.

Exemplu

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

- ***referință către o metodă de instanță a unui obiect arbitrar:***

`(obj,args)->obj.instanceMethod(args)`

`ObjectClass::instanceMethod`.

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

Referințe către metode

- ***referință către o metodă de instanță a unui obiect particular***:

lambda expresia **`(args) -> obj.instanceMethod(args)`**
obj::instanceMethod.

- **Exemplu:**

✓
```
Persoana p = new Persoana("Ionescu Ion", 35, 1500.5),
Supplier<String> numep = () -> p.getNume()
Supplier<String> numep = p::getNume.
```

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

Exemplu:
```
Supplier<Persoana> pnoua = () -> new Persoana()
Supplier<Persoana> pnoua = Persoana::new.
```

Metoda forEach

- În interfața `Iterable`, a fost adăugată în versiunea 8 o nouă metodă denumită **`forEach`** care permite parcurgerea unei structuri de date.

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

- Exemplu

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