Rețeaua socială

Seminar 5 – Interfețe funcționale, expresii lambda, referințe la metode

Enunț

În seminarul 4, comportamentul era fixat în clase: câte o clasă pentru fiecare validator, câte un `case` pentru fiecare tip de stocare, iar fiecare căutare nouă (după nume, după prenume, după numărul de prieteni) ar fi cerut o metodă nouă în repository. Vrem ca aceste comportamente să devină parametri: le transmitem ca expresii lambda sau referințe la metode, fără clase noi și fără să modificăm codul existent.

- `Validator<T>` are o singură metodă abstractă, deci este o interfață funcțională: un validator se poate scrie ca lambda.
- Căutările primesc criteriul ca parametru (`Predicate<E>`), sortările primesc ordinea (`Comparator<E>`), rapoartele primesc transformarea (`Function<T, R>`).
- `RepositoryFactory` nu mai folosește `switch`: fiecare tip de stocare are un `Supplier` care știe să creeze repository-ul.
- Folosim interfețele din `java.util.function` și `Comparator`. Nu folosim încă Stream API: parcurgerile se scriu cu `for`, `forEach` și `removeIf`.

Încălzire. De la clasă anonimă la lambda

> În pachetul `curs.ex.functional`, definiți interfața `Arie<E>` cu metoda `double calculeaza(E figura)` și clasele `Cerc` (rază) și `Patrat` (latură). Scrieți metoda generică `static <E> void printArie(List<E> figuri, Arie<E> arie)`, care afișează aria fiecărei figuri.
> Calculați aria cercurilor în trei feluri: clasă anonimă, expresie lambda și referință la metoda statică `Cerc::arie`. Pentru pătrate, transmiteți direct o lambda la apelul `printArie`.

Întrebări

1. Ce condiție trebuie să îndeplinească `Arie<E>` ca să putem scrie `c -> Math.PI * c.getRaza() * c.getRaza()` în locul clasei anonime? Este obligatorie adnotarea `@FunctionalInterface`?

   > **Răspuns:** Interfața trebuie să aibă exact o metodă abstractă: lambda nu are nume de metodă, deci compilatorul trebuie să știe fără ambiguitate ce metodă implementează. Adnotarea nu este obligatorie, dar cere compilatorului să verifice regula.

2. Ce afișează programul `curs.ex.G11`? Explicați diferența.

```
Runnable anonima = new Runnable() {
    private final String nume = "anonima";
    public void run() { System.out.println(this.nume); }
};
Runnable lambda = () -> System.out.println(this.nume);  // "G11"
```

   > **Răspuns:** Afișează `anonima`, apoi `G11`. Clasa anonimă este o clasă nouă, cu domeniul ei: `this` este obiectul anonim, iar câmpul lui ascunde câmpul din `G11`. Lambda nu introduce un domeniu nou: `this` este obiectul `G11` care o conține. Într-o metodă statică, `this` într-o lambda nici nu compilează.

Soluție (Arie, Cerc, Patrat, ArieDemo, G11)

Cerința 7. Validator ca expresie lambda

> Adăugați adnotarea `@FunctionalInterface` pe interfața `Validator<T>`. În `Main.cerinta7`, scrieți regula „prenumele nu lipsește” în trei feluri: clasă anonimă, expresie lambda și referință la metoda statică `Main::verificaPrenume`. Aplicați cele trei validatoare pe utilizatorul `7;;Pop`.
> Adăugați în `Validator<T>` metoda default `Validator<T> andThen(Validator<T> other)`, care aplică întâi validatorul curent, apoi pe `other`. Pentru `other == null` metoda aruncă `IllegalArgumentException`.
> **Fără** să modificați `UtilizatorValidator`, scrieți ca lambda regula „prenumele și numele au cel mult 10 caractere” și compuneți-o cu `andThen`. Injectați validatorul compus într-un `InMemoryRepository` și salvați `1;Ana;Pop`, `2;Maximilian-Alexandru;Pop` și `3;ana;Pop` (folosiți `salveaza` din cerinta2).

(axa de schimbare: reguli noi de validare)

Întrebări

1. Ce se întâmplă dacă adăugăm în `Validator<T>` o a doua metodă abstractă? Dar dacă adăugăm metoda default `andThen`?

   > **Răspuns:** Cu două metode abstracte, interfața nu mai este funcțională: nicio lambda de tip `Validator` nu mai compilează, iar `@FunctionalInterface` semnalează eroarea chiar în interfață („Unexpected @FunctionalInterface annotation”). Metodele default și statice nu contează, pentru că au implementare.

2. În `u -> { ... }` nu am scris tipul lui `u`. De unde știe compilatorul că `u` este un `Utilizator`? Compilează `var v = u -> {};`?

   > **Răspuns:** Din tipul țintă (target type): variabila este declarată `Validator<Utilizator>`, deci lambda implementează `validate(Utilizator)`. La `new InMemoryRepository<Long, Utilizator>(u -> ...)`, tipul țintă vine din parametrul constructorului. `var v = u -> {};` nu compilează („cannot infer type”): o lambda nu are tip propriu, primește tipul interfeței funcționale cerute de context. Aceeași lambda poate fi un `Validator<Utilizator>` sau un `Consumer<Utilizator>`.

3. Clasa anonimă, lambda și `Main::verificaPrenume` dau același rezultat. De ce se potrivește `verificaPrenume`, deși nu se numește `validate`?

   > **Răspuns:** Se potrivește forma, nu numele: `verificaPrenume` primește un `Utilizator` și întoarce `void`, exact ca `validate`. `Main::verificaPrenume` este o prescurtare pentru `u -> Main.verificaPrenume(u)`. Clasa anonimă poate avea în plus câmpuri și alte metode și generează un fișier `.class` separat (`Main$1.class`).

4. În lambda `lungime` folosim variabila locală `lungimeMaxima`. Ce se întâmplă dacă, după declararea lambda-ului, scriem `lungimeMaxima = 20;`?

   > **Răspuns:** Nu compilează: variabilele locale folosite într-o lambda trebuie să fie `final` sau effectively final. Lambda reține o copie a valorii și se execută mai târziu (la fiecare `save`), chiar după ce `cerinta7` s-a terminat și variabila locală nu mai există pe stivă. Câmpurile unei clase se pot modifica, pentru că lambda reține referința la obiect, nu o copie a câmpului.

5. Pentru un utilizator cu prenumele `null`, `new UtilizatorValidator().andThen(lungime)` și `lungime.andThen(new UtilizatorValidator())` dau rezultate diferite. De ce?

   > **Răspuns:** `andThen` aplică validatoarele în ordine și se oprește la prima excepție. În prima variantă, `UtilizatorValidator` aruncă `ValidationException` („prenume invalid”) înainte ca `lungime` să apeleze `getFirstName().length()`. În a doua, `lungime` se execută prima și apelează `length()` pe `null`: `NullPointerException`, pe care `salveaza` nu o prinde. Compunerea nu este comutativă: regulile care presupun date corecte vin după validarea de bază.

6. Am adăugat o regulă nouă fără clasă nouă și fără să modificăm `UtilizatorValidator`. Când merită totuși o clasă și când este suficientă o lambda?

   > **Răspuns:** O clasă, când regula este lungă, are un nume în domeniu, se folosește în mai multe locuri sau trebuie testată și documentată separat. O lambda, când regula este scurtă, folosită într-un singur loc și se înțelege din cod. În ambele cazuri avem Strategy: `Validator` este Strategy, `InMemoryRepository` este Context, iar lambda este o ConcreteStrategy fără nume.

Soluție (Validator, Main.cerinta7, Main.verificaPrenume)

Cerința 8. Căutări cu criterii

> Adăugați în `Repository<ID, E>` metoda default `List<E> findAll(Predicate<E> filter)`, care întoarce entitățile ce satisfac criteriul. Pentru `filter == null` metoda aruncă `IllegalArgumentException`. Nu modificați nicio implementare de repository.
> Creați clasa utilitară `CriteriiUtilizator` în pachetul `curs.domain`, cu metodele statice `prenumeleIncepeCu(String prefix)`, `areNumele(String nume)` și `areCelPutinPrieteni(int n)`, care întorc `Predicate<Utilizator>`, și metoda `boolean faraPrieteni(Utilizator u)`.
> În `Main.cerinta8`, pe rețeaua construită de `reteaDemo()`, afișați utilizatorii: cu numele Pop; al căror prenume începe cu I **și** au numele Pop; cu cel puțin 2 prieteni **sau** cu numele Matei; care **nu** au numele Pop; care au prieteni (fără metodă nouă: folosiți `faraPrieteni`).
> Afișați fiecare utilizator cu numărul lui de prieteni, compunând două `Consumer` cu `andThen`. Copiați utilizatorii într-o listă cu `Colectii.catreLista` (clasă utilitară în `curs.util`) și eliminați cu `removeIf` pe cei fără prieteni.

(axa de schimbare: criterii noi de căutare)

Întrebări

1. Alternativa ar fi fost metode ca `findByLastName`, `findByFirstNamePrefix`, `findByMinFriends` în `Repository`. Ce problemă are această variantă?

   > **Răspuns:** Fiecare criteriu nou modifică interfața `Repository` și toate implementările ei (memorie, fișier, bază de date): se încalcă OCP. Combinațiile (prenume **și** nume) ar cere alte metode. Cu `Predicate`, repository-ul doar parcurge entitățile, iar apelantul decide ce înseamnă „potrivire”. Interfața rămâne mică (ISP).

2. `findAll(Predicate)` este default în `Repository`. Ce primesc `InMemoryRepository`, `UtilizatorFileRepository` și `UtilizatorDbRepository`? Ce se întâmplă dacă o apelăm pe `UtilizatorDbRepository`?

   > **Răspuns:** Toate primesc metoda fără nicio modificare, pentru că ea folosește doar `findAll()`, care face parte din contract. Pe `UtilizatorDbRepository`, `findAll()` aruncă `UnsupportedOperationException`, deci și `findAll(filter)`. Când repository-ul în baza de date va fi implementat, nu vom vrea să aducem toate rândurile și să filtrăm în Java. Metoda poate fi suprascrisă, dar un `Predicate` este cod Java opac și nu poate fi tradus într-un `WHERE`.

3. Interfața `Iterable` există din Java 5. Cum a primit în Java 8 metoda `forEach`, fără să se strice clasele care o implementau deja?

   > **Răspuns:** `forEach` este metodă default în `Iterable`: clasele existente o moștenesc fără să fie modificate. Pentru asta au fost introduse metodele default, și din același motiv am adăugat `findAll(filter)` ca default în `Repository`. La fel au apărut `removeIf` în `Collection` și `sort` în `List`.

4. `prenumeleIncepeCu("I")` întoarce o lambda care folosește `prefix`. Când lambda este apelată în `findAll`, metoda s-a terminat deja. De unde mai știe lambda valoarea lui `prefix`?

   > **Răspuns:** Lambda capturează valoarea lui `prefix` în momentul în care este creată (closure). Fiecare apel `prenumeleIncepeCu("I")`, `prenumeleIncepeCu("A")` produce un alt obiect `Predicate`, cu propria valoare. O metodă care întoarce o funcție este o funcție de ordin superior: aici, o „fabrică de criterii".

5. De ce scriem `Predicate.not(CriteriiUtilizator::faraPrieteni)` și nu `CriteriiUtilizator::faraPrieteni.negate()`?

   > **Răspuns:** O referință la metodă, ca și o lambda, nu are tip până nu apare într-un context cu tip țintă, deci nu putem apela metode pe ea: eroarea este „method reference not expected here". `Predicate.not` primește un `Predicate` și oferă astfel tipul țintă. Alternativ: `Predicate<Utilizator> fara = CriteriiUtilizator::faraPrieteni;` apoi `fara.negate()`.

6. De ce folosim `removeIf` și nu un for-each în care apelăm `lista.remove(u)`?

```
for (Utilizator u : lista)
    if (u.getFriends().isEmpty())
        lista.remove(u);
```

   > **Răspuns:** Modificarea listei în timpul parcurgerii cu iterator aruncă, de obicei, `ConcurrentModificationException` la pasul următor. Nu întotdeauna: dacă elementul șters este penultimul, `hasNext()` întoarce `false`, bucla se oprește fără excepție, iar ultimul element nu mai este verificat deloc. `removeIf` șterge corect, într-un singur apel, și primește doar criteriul.

Soluție (Repository.findAll(filter), CriteriiUtilizator, Colectii.catreLista, Main.reteaDemo, Main.cerinta8)

Cerința 9. Sortări și rapoarte

> Adăugați în `Colectii` metodele generice `<T, R> List<R> transforma(Iterable<T> elemente, Function<T, R> f)`, care aplică `f` fiecărui element, și `<E> E reduce(Iterable<E> elemente, BinaryOperator<E> op)`, care combină elementele două câte două și întoarce `null` pentru o colecție vidă.
> În `Main.cerinta9`, sortați utilizatorii: (a) după nume, apoi după prenume; (b) descrescător după numărul de prieteni, iar la egalitate după nume. Construiți comparatorii cu `Comparator.comparing`, `comparingInt`, `thenComparing` și `reversed`, **fără** să scrieți `compare` de mână.
> Afișați cu `transforma`: numele complet; numele complet cu majuscule (compuneți două funcții cu `andThen`); numele complet urmat de numărul de prieteni. Determinați cel mai popular utilizator cu `reduce` și `BinaryOperator.maxBy`.

(axa de schimbare: ordini și rapoarte noi)

Întrebări

1. `Comparator<Utilizator> c = Comparator.comparingInt(u -> u.getFriends().size());` compilează, dar `lista.sort(Comparator.comparingInt(u -> u.getFriends().size()).reversed());` nu. De ce? Cum îl reparăm?

   > **Răspuns:** `comparingInt(...)` se evaluează înainte de `reversed()`, iar tipul țintă cerut de `sort` se aplică doar rezultatului final, al lui `reversed()`. Compilatorul nu poate deduce tipul lui `u`, îl consideră `Object`, iar `Object` nu are `getFriends()`. Reparații: tip explicit în lambda, `(Utilizator u) -> ...`; type witness, `Comparator.<Utilizator>comparingInt(...)`; sau o variabilă intermediară, ca în soluție.

2. Pe slide-ul 27 din curs, comparatorul este `return (int) (a.getMedia() - b.getMedia());`. Ce întoarce pentru mediile 9.6 și 9.1? Ce problemă are `(x, y) -> x - y` pentru numere întregi?

   > **Răspuns:** `(int) 0.5` este 0, deci cei doi studenți sunt considerați egali și nu sunt ordonați. La `int`, scăderea poate depăși intervalul: pentru `Integer.MIN_VALUE` și `1` rezultatul este `2147483647`, adică un număr pozitiv

> greșit. Corect: `Double.compare`, `Integer.compare` sau `Comparator.comparingDouble` / `comparingInt`, care nu fac scăderi.

3. Sortarea (b) folosește `dupaNrPrieteni.reversed().thenComparing(dupaNume)`. Ce se schimbă dacă scriem `dupaNrPrieteni.thenComparing(dupaNume).reversed()`?

   > **Răspuns:** `reversed()` inversează tot comparatorul pe care este apelat. În prima variantă, doar numărul de prieteni este descrescător: Mihai Ionescu, Ioana Marin, Ion Pop. În a doua, și numele devin descrescătoare: Ion Pop, Ioana Marin, Mihai Ionescu.

4. Clasificați referințele la metode din soluție: `Main::verificaPrenume`, `CriteriiUtilizator::faraPrieteni`, `System.out::println`, `rezultat::add`, `Utilizator::getLastName`, `String::toUpperCase`, `Utilizator::new`.

   > **Răspuns:** Metodă statică: `Main::verificaPrenume`, `CriteriiUtilizator::faraPrieteni` (`u -> CriteriiUtilizator.faraPrieteni(u)`).
   > Metodă de instanță a unui obiect anume: `System.out::println`, `rezultat::add` (`x -> rezultat.add(x)`); obiectul este fixat când se creează referința.
   > Metodă de instanță a unui obiect oarecare de un tip: `Utilizator::getLastName`, `String::toUpperCase` (`u -> u.getLastName()`); obiectul este primul parametru.
   > Constructor: `Utilizator::new` (`(p, n) -> new Utilizator(p, n)`), `ArrayList::new`.

5. `AbstractFileRepository` (Template Method) cere subclaselor două metode: `extractEntity` și `createEntityAsString`. Cum ar arăta cu interfețe funcționale? Ce câștigăm și ce pierdem?

   > **Răspuns:** Un singur `FileRepository<ID, E>` care primește prin constructor `Function<String, E> parser` și `Function<E, String> formatter`:
   > ```
   > new FileRepository<Long, Utilizator>(validator, "data/utilizatori.txt",
   >         linie -> utilizatorDinText(linie),
   >         u -> u.getId() + ";" + u.getFirstName() + ";" + u.getLastName());
   > ```
   > Câștigăm: nu mai avem câte o subclasă pentru fiecare entitate (compoziție în loc de moștenire), iar constructorul nu mai apelează o metodă abstractă înainte ca subclasa să fie inițializată (warning-ul `this-escape` dispare). Pierdem: conversia nu mai are un nume și un loc propriu, documentat și testabil separat.

Soluție (Colectii.transforma, Colectii.reduce, Main.cerinta9)

Cerința 10. Factory fără switch

> Rescrieți `RepositoryFactory.createUtilizatorRepository` fără `switch`: construiți un `Map<RepositoryType, Supplier<Repository<Long, Utilizator>>>`, în care fiecărui tip de stocare îi corespunde un `Supplier` care creează repository-ul. Metoda alege `Supplier`-ul după tip și îl apelează. Pentru un tip fără `Supplier`, aruncă `IllegalArgumentException`.
> Nu modificați `Main.cerinta4`. În `Main.cerinta10`, creați pe rând repository-ul pentru `MEMORY`, `FILE` și `DATABASE`, schimbând doar proprietatea `repository.type`, și afișați clasa obținută.

(axa de schimbare: tipuri noi de stocare)

Întrebări

1. De ce `Map<RepositoryType, Supplier<...>>` și nu direct un `Map.of(MEMORY, new InMemoryRepository<>(validator), FILE, new UtilizatorFileRepository(...), ...)`?

   > **Răspuns:** `Map.of` cu obiecte ar crea toate cele trei repository-uri la fiecare apel: fișierul ar fi citit și când lucrăm în memorie, iar un repository real în baza de date ar deschide o conexiune. `Supplier` amână crearea (lazy): lambda doar descrie cum se creează obiectul, iar `get()` o execută numai pentru tipul ales. Este aceeași problemă ca fall-through-ul din seminarul 4, rezolvată altfel.

2. Ce s-a modificat în `Main.cerinta4` față de seminarul 4?

   > **Răspuns:** Nimic. `Main` depinde doar de `RepositoryFactory` și de interfața `Repository`; felul în care factory-ul alege implementarea este un detaliu intern (încapsulare). Am schimbat implementarea, nu contractul.

3. Comparați cu `switch`-ul din seminarul 4: ce trebuie modificat pentru un al patrulea tip de stocare (XML)? Ce greșeli posibile au dispărut?

   > **Răspuns:** O clasă nouă, o constantă în enum și o intrare în map. Au dispărut fall-through-ul (`break` uitat) și ramura `default` necesară doar pentru compilare. Factory-ul se modifică în continuare la fiecare tip nou; ca să nu-l mai modificăm, map-ul ar putea fi completat din afară, cu o metodă `inregistreaza(RepositoryType tip, Supplier<...> creator)` (registry).

4. Map-ul se construiește la fiecare apel, pentru că `Supplier`-ii capturează `validator`. Cum l-am construi o singură dată, în constructor? Cu ce interfață funcțională se potrivește `InMemoryRepository::new`?

   > **Răspuns:** Validatorul devine parametru al funcției: `Map<RepositoryType, Function<Validator<Utilizator>, Repository<Long, Utilizator>>>`, construit în constructor, iar metoda apelează `creatori.get(type).apply(validator)`. Constructorul `InMemoryRepository(Validator<E>)` primește un argument și întoarce un obiect, deci `InMemoryRepository::new` se potrivește cu această `Function`, nu cu `Supplier` (care nu primește argumente). O referință la constructor ia forma interfeței cerute de context.

5. Cu `repository.type=XML`, eroarea vine din `RepositoryType.valueOf`, nu din map. Când se ajunge la verificarea `creator == null`?

   > **Răspuns:** Când constanta există în enum, dar nu i s-a asociat un `Supplier` (adăugăm `XML` în `RepositoryType` și uităm map-ul). `Map.get` întoarce `null`, iar fără verificare `creator.get()` ar arunca `NullPointerException`, al cărei mesaj nu spune ce tip lipsește.

Soluție (RepositoryFactory, Main.cerinta10)

Exercițiu rapid. Completați tipul

Completați tipul fiecărei variabile (soluția compilabilă este în `curs.ex.functional.CompletatiTipul`).

```
____ ex1 = x -> "".equals(x.get(0));
____ ex2 = (Long l) -> System.out.println(l);
____ ex3 = s -> s.contains("a");
____ ex4 = () -> new ArrayList<String>();
____ ex5 = String::concat;
____ ex6 = Utilizator::getFirstName;
____ ex7 = String::length;
____ ex8 = String::toUpperCase;
____ ex9 = Utilizator::new;
```