

Teorie PP

1. Ce face functia map?

Returnează o listă care conține rezultatele aplicării funcției de transformare date fiecărui element din tabloul original.

For e în it:

$$\text{Func}(e) \Leftrightarrow \text{map}(\text{func}, \text{it})$$

2. Ce sunt generatorii recursivi?

Generatoarele reprezintă o modalitate simplă de a crea iteratori, toate cerințele anterioare fiind gestionate în mod automat de generator.

Definirea unui generator presupune utilizarea cuvântului cheie `yield` în loc de `return`. Acesta trebuie să conțină cel puțin un `yield` (poate conține mai multe alte `yield`-uri, `return`-uri).

Diferența între `return` și `yield`:

- `return` - încheie complet execuția funcției
- `yield` - pune pauză funcției, salvând stările și continuând mai târziu de unde a rămas.

Avantajele utilizării unui generator în loc de iterator:

- ușor de implementat
- eficient din punct de vedere al memoriei
- permite reprezentarea unui stream infinit
- generatoarele pot fi folosite pentru a realiza un pipeline cu o serie de operații

Generator recursiv: `yield from`

`yield from g` is equivalent to `for v in g: yield v`

3. Cum arata un when si ce face?

When defineste o expresie conditională cu mai multe ramuri; similar cu un switch din limbajul C

Asignam valoare unei variabile cu when

```
when (x) {  
    1 -> print("x == 1")  
    2 -> print("x == 2")  
    else -> { // Note the block  
        print("x is neither 1 nor 2")  
    }  
}
```

4. Ce este un efect lateral?

Este acel efect ce constă în modificarea variabilelor existente în exteriorul blocului funcției

5. Ce este coerenta datelor?

Se poate asigura prin sincronizarea corutinelor, prin lock-uri, prin variabile atomice.

6. Ce este list/set/dictionary comprehension?

List comprehension

Returnează o listă bazată pe valorile existente.

-modalitate mai simplă decât FOR Loop

Set comprehension

Returnează un set bazat pe valorile existente.

Lista=[expresie for element in lista-intrare if conditie]

Dictionary comprehension

Returnează un dicționar bazat pe valorile existente

7. De ce se folosesc exceptiile Custom?

Exceptie: Este un eveniment care apare în timpul executiei programului și care opreste fluxul normal al instrucțiunilor. Cand programul gaseste o situație care nu-i convine trimite o exceptie. Exceptia este un obiect python care reprezinta o eroare

Ai flexibilitatea de a adauga atribute și metode care nu fac parte din expresiile java standard și îți poti da seama mai usor ce e gresit având în vedere că pot să scrieți tu exceptia respectivă

8. Ce sunt functiile extensie?

Este o funcție ce extinde o clasa, ca și cum am face o nouă clăsă la care mai adaugăm o funcție

9. De explicat dependenta dintre două clase

Asociere - relație de asociere între două clase (clasa sursă folosește membri din țintă(target) sub formă de câmp / atribut / proprietate (terminologia diferă).

Dependență - relație de dependență între două clase (clasa sursă folosește clasa din țintă ca un parametru sau ca o variabilă în interiorul funcțiilor).

Moștenire - moștenire (clasa sursă derivează clasa țintă, adăugându-i noi funcționalități).

Implementare (Realizare) - Implementarea unei interfețe (clasa sursă implementează interfața din țintă)

Agregare - implică o relație în care o clasă A conține una sau mai multe instanțe ale clasei B, iar ștergerea instanței clasei A nu duce la ștergerea instanțelor clasei B (Exemplu: o sală de curs și studenții care participă la curs).

Compoziție - spre deosebire de agregare, ștergerea instanței clasei A duce la ștergerea tuturor instanțelor clasei B (Exemplu: Apartament, cameră -> dacă se șterge instanța de apartament, camerele nu pot exista fără acesta și se șterg).

10. Principiile S.O.L.I.D

Principiul Responsabilitatii Unice (**Single**) - O clasă trebuie să aibă o singură rațiune pentru a se schimba.

Principiul Inchis / Deschis (**Open / Closed**) - - Entitatele software trebuie să fie deschise pentru extindere dar închise pentru modificare

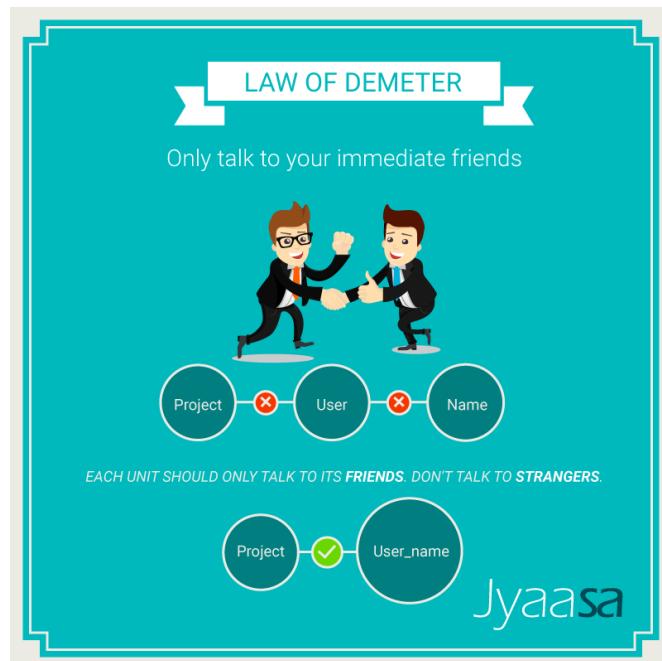
Principiul Substitutiei **Liskov**- - Copiii claselor nu au voie să încalce definițiile de tip din clasa părinte. Adică funcțiile care folosesc referințe la clase de bază, trebuie să poată manipula într-un mod transparent instanțele claselor derivate din acestea.

Principiul separarii **Interfetelor**- Nici un client nu trebuie forțat să depindă de metode pe care nu le folosește. Numărul de membri din interfață care este vizibil pentru clasele dependente trebuie minimizat. Clasele mari vor implementa mai multe interfețe mai mici care grupează funcțiile după maniera lor de utilizare.

Principiul Dependentei inverse- modulele de nivel arhitectural superior nu trebuie să depindă de cele de nivel inferior. Ambele trebuie să depindă de abstracții care, la rândul lor nu trebuie să depindă de detalii. Practic detaliiile depind de abstracții, nu invers. Dacă aceasta dependență nu este vizibilă în faza de proiectare atunci ea se construiește.

11. Principiul lui Demeter

Principiul Legii lui Demeter afirma că un modul nu ar trebui să cunoască detaliile interioare ale obiectelor pe care le manipulează. Cu alte cuvinte, o componentă software sau un obiect nu ar trebui să cunoască funcționarea internă a altor obiecte sau componente.



12. Ce este EDP(Event driven pattern)?

Programarea orientată eveniment este o paradigmă a programării calculatoarelor. Spre deosebire de programele tradiționale, care-și urmează propria execuție, schimbându-și câteodata doar cursul în puncte de ramificație, cursul execuției unui program orientat eveniment este condus în mare parte de evenimente externe.

13. Granularitatea firelor de executie

Granularitatea unei sarcini este o masura a cantitatii de munca sau de calcul realizata de acea sarcina.

Granularitatea=Raportul dintre timpul de calcul si timpul de comunicare

timpul de calcul= timpul necesar pt efectuarea calculului unei sarcini

Timpul de comunicare= timpul necesar pentru schimbul de date intre procesoare

14. Ce librerie din python pentru calcul paralel duce la cel mai bun timp de executie

Asyncio-----Corutine

15. Ce este UML?

UML(unified modeling language) este un limbaj grafic adoptat mondial folosit pentru reprezentarea,vizualizarea si documentarea componentelor unui sistem software de dimensiuni mari

16. Ce sunt corutinele? Ce difera in raport cu un thread? De ce pot apela 1mil de corutine, dar nu si 1mil de threaduri?

Ca si concept, o corutina este similara unui thread, adica poate contine un bloc de instructiuni care se executa concurent cu restul codului. Totusi, corutinele sunt mai light-weight si nu sunt legate de un anumit thread. Ele pot porni in cadrul unui thread si se pot termina in altul

Scopuri: Global, LifeCycle, ViewModel

17. Ce este un Deadlock (interblocarea)?

Un proces asteapta ceva ce a blocat (ocupat, folosit) celalalt si viceversa. (LIVELOCK -> problema filosofilor -> spre deosebire de deadlock, nu se blocheaza, ci se incerca mereu

18. Ce sunt Efectele laterale

În general, orice efect de durată care apare într-o funcție, nu prin valoarea sa returnată, se numește efect lateral.

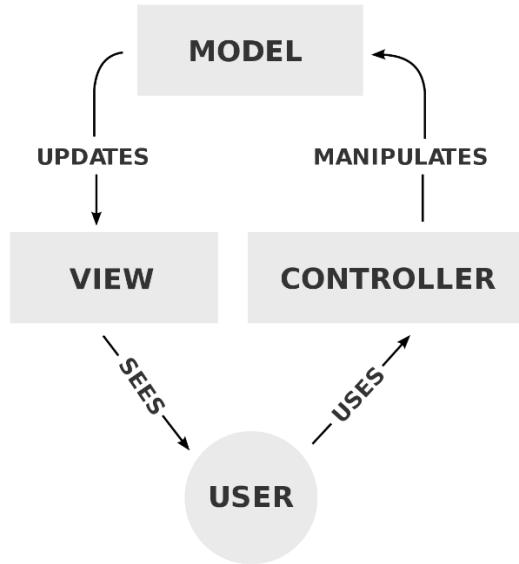
19. Ce e bariera?

Obiectele de barieră din python sunt utilizate pentru a aștepta un număr fix de fir pentru a finaliza execuția înainte ca orice thread special să poată continua cu executarea programului. Fiecare fir apelează funcția wait () la atingerea barierei. Bariera este responsabilă pentru urmărirea numărului de apeluri de așteptare (). Dacă acest număr depășește numărul de fire pentru care bariera a fost inițializată, atunci bariera oferă o modalitate firelor de așteptare pentru a continua execuția. Toate firele din acest punct de execuție sunt lansate simultan.

20. Ce este o functie lambda?

Puteam crea o functie fara sa le asignam un nume explicit(functii fara nume)

21. Sa desenez Model View Controller



22. Ce face itertools.tee si ce erori poate provoca?

Modulul itertools conține o serie de iteratori utilizate și funcții pentru combinarea mai multor iteratori. Funcțiile modulului se încadrează în câteva clase largi: -Funcții care creează un iterator nou pe baza unui iterator existent. -Funcții pentru tratarea elementelor unui iterator ca argumente funcționale. -Funcții pentru selectarea unor porțiuni a ieșirii unui iterator. -O funcție pentru gruparea ieșirilor unui iterator.

23. Ce este o funcție pură?

O funcție care nu are efecte laterale(nu schimba variabilele din exteriorul ei).

24. Generice

Programarea generică (sau genericele) este o tehnică prin care funcțiile pot fi scrise cu tipuri care nu sunt specificate în momentul scrierii și sunt folosite mai târziu cu diferite tipuri de date.

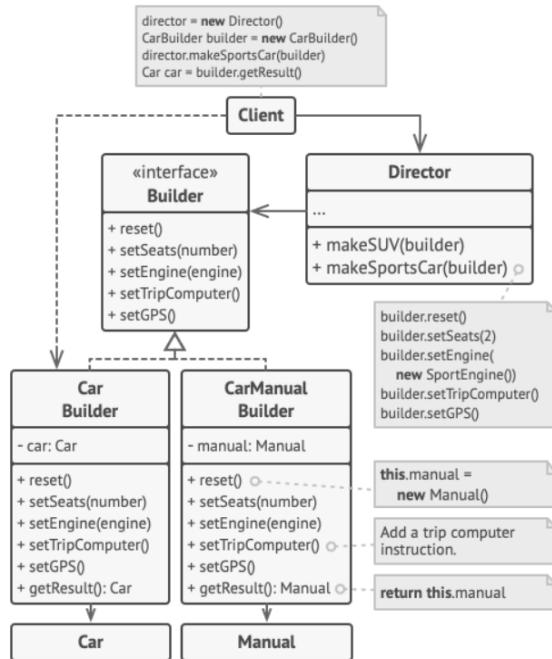
25. Explicație pt any și all (colectii)

any -> macar una dintre "ele" să respecte o anumita condiție (condiții cu || între ele) -> returnează true dacă macar o condiție e true

all -> toate dintre "ele" trebuie să respecte o anumita condiție (condiții cu && între ele) -> returnează true dacă toate condițiile sunt true

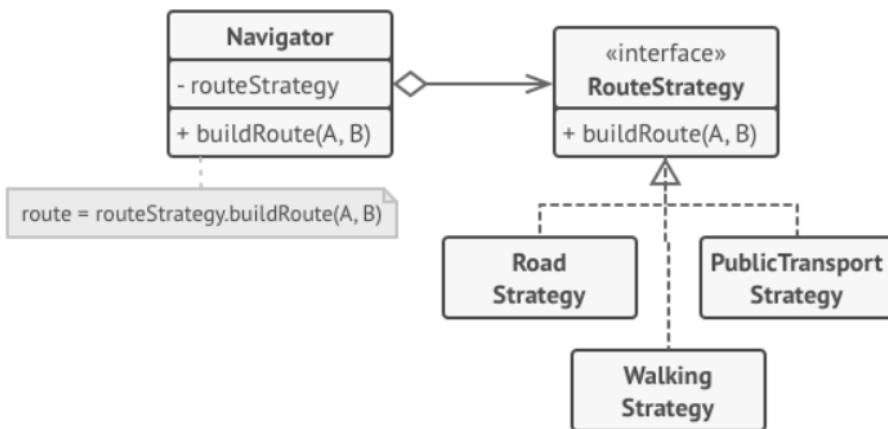
Builder:

Builder este un model de creație care vă permite să construiți obiecte complexe pas cu pas. Modelul vă permite să produceți diferite tipuri și reprezentări ale unui obiect folosind același cod de construcție.



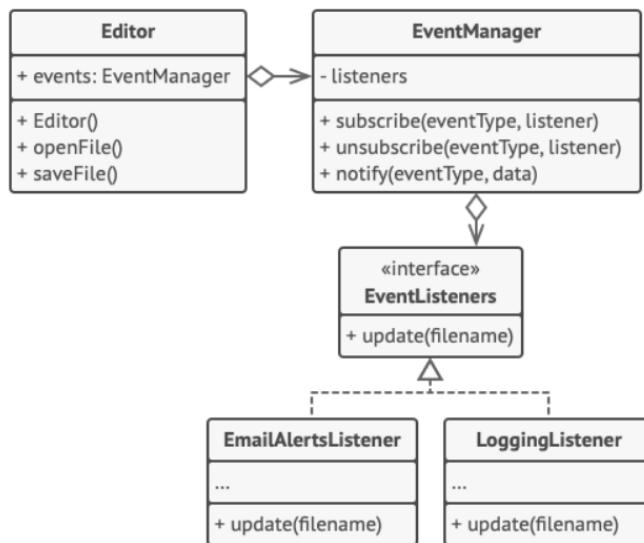
Strategy:

Strategia este un model de proiectare comportamentală care vă permite să definiți o familie de algoritmi, să le punetă pe fiecare într-o clasă separată și să le faceti obiectele interschimbabile.



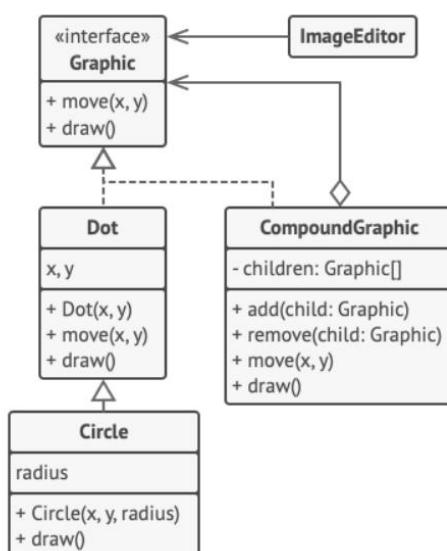
Observer:

Observatorul este un model de proiectare comportamentală care vă permite să definiți un mecanism de abonament pentru a notifica mai multe obiecte despre orice evenimente care se întâmplă cu obiectul pe care îl observă.



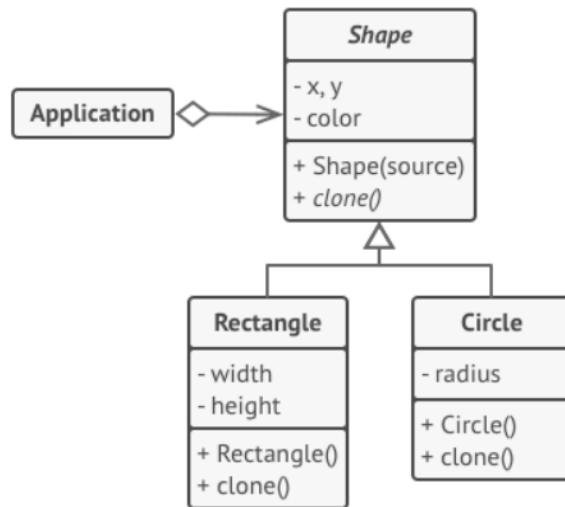
Composite:

Composite este un model de proiectare structurală care vă permite să compuneți obiecte în structuri de copac și apoi să lucrați cu aceste structuri ca și cum ar fi obiecte individuale.



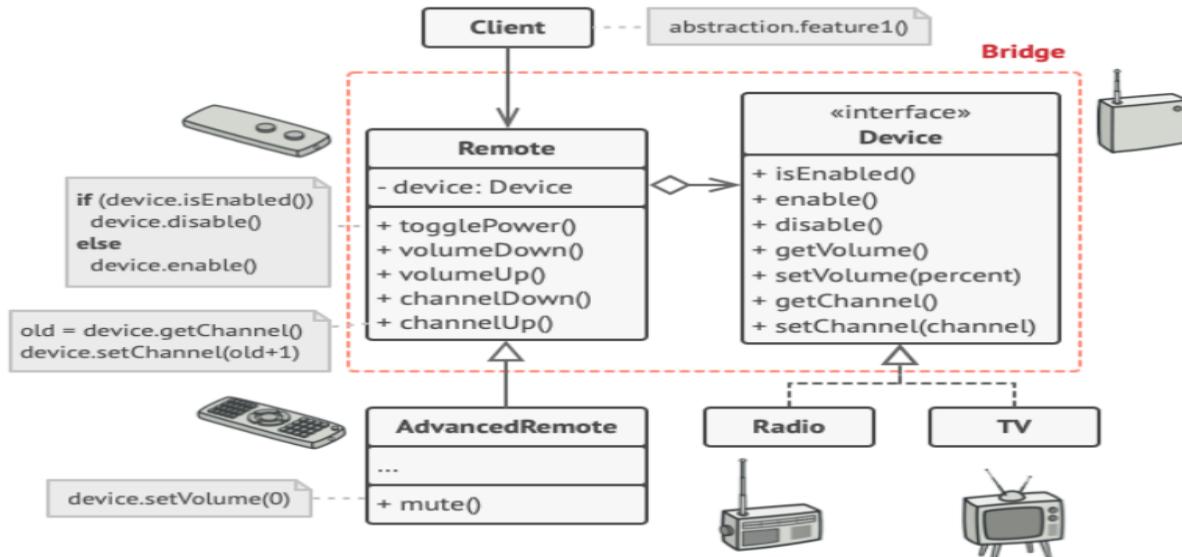
Prototype:

Prototipul este un model de design creațional care vă permite să copiați obiecte existente fără a face codul dvs. dependent de clasele lor.



Bridge:

Bridge este un model de proiectare structurală care vă permite să împărțiți o clasă mare sau un set de clase strâns legate în două ierarhii separate - abstractizare și implementare - care pot fi dezvoltate independent unul de celălalt.



Intrebari Curs 13

1) Ce este calculul Lambda?

R: Este un model matematic formal in care folosim functii cu parametri, fara nume, pt calculul unor expresii.

2) Cum s-ar putea scrie un if cu un lambda utilizand doua obiecte?

R:

true = lambda x, y: x

false = lambda x,y: y

if = lambda p, x, y: p(x,y)

3) La ce se refera *args si **kwargs?

R: Permit introducerea unui nr variabil de argumente la o functie

args - tupla

kwargs - dictionar

4) La ce se refera functiile de prim nivel?

R: In python totul este obiect (chiar si functiile).

Functiile de prim nivel sunt ca niste obiecte cu atribute, pe care le putem inspecta

5) La ce se refera datele imutabile?

R: Datele imutabile se refera la datele care nu-si pot modifica continutul

6) La ce se refera functiile pure?

R: Functiile pure sunt cele fara efecte laterale.

7) La ce se refera abordarea impacheteaza - proceseaza - despacheteaza?

R:

8) Cum se realizeaza evaluarea la cerere?

R:

9) Ce este un generator recursiv?

R: yield from

10) Pentru ce utilizam modulul iterTools?

R: Pt a folosi functii care genereaza si proceseaza serii si sechete de numere (pt partea hardware sunt f bune deoarece se pot genera repede si usor semnale)

11) Cand utilizam iteratorii infiniti?

R: Sunt utili pt generarea semnalelor (sinus, cosinus etc.)

12) Cum se calculeaza eroarea prin acumulare?

R: Se aduna putin cu putin.

13) Care-i diferența intre functiile "cycle" si "accumulate"?

R: cycle -> repeta argumentele la infinit

accumulate -> efectueaza o operatie asupra fiecarui argument

14) Cum se poate utiliza modulul iterTools pt a crea functii de ordin 2?

R:

15) Cand se utilizeaza functie "tee"?

R:

16) Cand utilizam operatorii "any" si "all"?

R: any -> macar una dintre "ele" sa respecte o anumita conditie (conditii cu || intre ele) -> returneaza true daca macar o conditie e true

all -> toate dintre "ele" trb sa respecte o anumita conditie (conditii cu && intre ele) -> returneaza true daca toate conditiile sunt true

17) Explicati data flatten.

R:

18) Ce este scurtcircuitul (nu ala electric)?

R: De ex intr-un if in care avem mai multe conditii cu "OR", daca prima cond e adevarata atunci nu mai trebuie verificate si celelalte -> scurtcircuit

19) Cum putem evalua deciziile in calculul functional?

R:

20) Ce este un "closure"?

R: Functiile in python sunt referinte catre obiecte. Deci ele pot suporta si apeluri de alte obiecte???
- inchiderile in python retin si contextul in care au fost incheiate

21) Care operatori pe colectii sunt mai rapizi si in ce domeniu?

R:

22) Care este diferența intre un decorator in stil macro si unul in stil OOP?

R: - stil macro (pe functie)
 - stil OOP (pe clasa)

23) Cum putem implementa un state machine (FSM)?

R: - Utilizam lambda cand avem expresii destul de simple pt care nu merita sa cream o functie
- mapReduce are 2 etape -> mapare si reducere (exemplificam pe exemplu din lab)

1) mapare -> se elimina semnele de punctuatie si se fac litere mici, se sparge textul dupa spatiu si se fac perechi de genul

- ("ana", 1), ("are", 1), ("mere", 1), ...

2) shuffle & sorting (etapa intermediara) -> sorteaza

3) reducere -> suma valorilor pt elementele cu aceeasi cheie

In urma acestor operatiuni a rezultat practic un Word Counter.

Intrebari Curs 12

1. Dati exemple de transformari intre spatii

R:

2. Ce este banda "moebils"?

R: Este un model de suprafață cu o singură față și o singură margine.

Banda are proprietatea matematică de a fi neorientabilă

3. Care este ipoteza lui Church?

R: Fiecare functie care poate fi calculata poate fi scrisa recursiv

4. Care e ipoteza lui Turing?

R:

5. Ce este o functie anonima?

R: Este o functie fara nume

6. Ce e lambda?

R: Functii pe care le putem crea fara a le asigna un nume explicit

7. Care sunt limitarile Java in cazul calculului functional?

R:

8. Cum se poate utiliza o functie ca o proprietate?

R:

9. Ce tip de date au functiile lambda in kotlin?

R:

10. Dati un exemplu de o functie de nivel superior in kotlin

R: O functie care accepta parametri si poate returna o alta functie fold

11. Ce este un efect lateral?

R: Este acel efect ce consta in modificarea variabilelor existente
in exteriorul blocului functii

12. Ce este o functie pura?

R: O functie care nu are efecte laterale(nu schimba variabilele din exteriorul ei)

13. Ce face cuvantul cheie vararg?

R: Permite primirea unui numar variabil de argumente

14. Explicati parametru alias

R: Ca un typedef, redenumim tipuri existente

15. Ce este o functie extensie?

R: Ce este o functie ce extinde o clasa, ca si cum am face o noua clasa la care mai adaugam o
functie

16. Care e diferenta dintre functia unei clase, functia supraincarcata intr-o clasa derivata
si functia extensie acelei clase?

R:

17. Ce este un dispatcher receiver?

R:

18. Cand exista posibilitatea unui conflict de nume in utilizarea functiilor de extensie?

R:

19. Ce sunt functiile de extensie pt obiecte?

R:

20. Ce sunt functiile infix?

R:

21. La ce este buna functia map?

R: Returneaza o lista ce contine rezultatele aplicatii unei transformari
listei initiale

22. Cand utilizam functia filter?

R: Cand dorim sa selectam doar anumite elemente dintr-un container

23. Ce face functia flatMap?

R: Imi "aplatizeaza", de exemplu, mai multe liste intr-o lista.

Transforma dintr-o lista de liste intr-o lista

24. Ce rol au functiile drop si take?

R:

25. Ce este functorul?

R: Este o clasa ce poate fi apelata

26. Ce sunt functiile curry?

R:

Intrebari Curs 10

1) Ce este calculul paralel?

R: Permite executarea mai multor programe in acelasi timp (desfasurarea simultana a mai multor procese)

- nr programe/procese = nr procesoare (avand in vedere ca ale noastre computere au maxim 8 procesoare, nu prea se aplica paralelismul, ci concurrenta)

2) Ce este concurrenta? (pseudoparalelism)

R: Concurrenta = paralelism simulat (practic, mai multe procese au nevoie de o resursa comună și concurează pentru a obține acces la ea)

- nr de entități care efectuează ceva > nr procesoare

3) Ce este concurrenta la nivel de date?

R: Se folosesc date (variabile) comune.

- Pot apărea probleme de coerență: Dacă un thread scrie o nouă valoare într-o variabilă, iar altul citeste variabilă respectivă, nu se știe sigur care valoare va fi citită (cea veche sau cea nouă)

4) Ce este paralelismul la nivelul datelor?

R: Nu există probleme în asigurarea coerenței. Se folosesc seturi de date separate (sau se împart datele initiale în bucăți și se lucrează separat -> fiecare proces/thread cu bucata lui)

5) Ce este o corutina?

R: Ca și concept, o corutina este similară unui thread, adică poate conține un bloc de instrucțiuni care se executa concurrent cu restul codului. Totuși, corutinile sunt mai light-weight și nu sunt legate de un anumit thread. Ele pot porni în cadrul unui thread și se pot termina în altul.

6) Explicati ciclul de viata al unei corutine.

R: stored -> on-stack -> running -> finished

- stored -> start minimal sub forma de obiecte copiate pe stiva

- on-stack -> datele corutinei au fost introduse in stiva de lucru, dar corutina nu a fost lansata in executie

- running -> o corutina intra in executie

- finished -> unde se ajunge cu terminarea corutinei respective

7) La ce se refera distrugerea la ordin?

R:

8) Ce face cuvantul cheie "suspend"?

R: Cuvantul cheie suspend practic ne spune ca acea functie poate fi intrerupta oricand, dar poate fi si reluată oricand. Lucrul asta este util dacă vrem să facem o planificare de genul Round - Robin

9) Ce face instructiunea "run blocking"?

R: Incepe o nouă corutină și cumva desparte tot ce este în afara corutinei și ce este în interiorul corutinei. Practic blochează thread-ul care execuția codului din afara acestei corutini și îl deblochează când s-a terminat de făcut tot ce era în interiorul corutinei

10) Ce scope-uri există pt corutine?

R: Global, LifeCycle, ViewModel

11) Ce este un thread local?

R: Thread Local este o clasa ce se folosește la a declara variabile ce se pot citi sau scrie de același thread. De exemplu, avem 2 thread-uri care accesează același cod care conține o referință spre o variabilă de tip thread local, atunci niciunul din thread-uri nu va vedea modificările facute de celalalt thread la acea variabilă.

12) Cum se poate asigura coerenta (integritatea) datelor?

R: Prin sincronizarea corutinelor, prin lock-uri, prin variabile atomice

13) Ce face instructiunea "async"?

R: Incepe o noua corutina, returneaza o valoare de tip Deferred, cumva promite ca va primi rezultatul mai tarziu. Folosim await pentru a-i primi valoarea unei variabile de tip Deferred

14) Cum se poate crea un thread in Kotlin?

R: Mostenind clasa Thread sau implementand interfata Runnable sau apeland functia thread() din kotlin

15) Ce tipuri de dispatcheri exista?

R: Main Dispatcher, IO Dispatcher, Default Dispatcher, Unconfined Dispatcher

16) La ce se refera reflexia si adnotarea in Kotlin?

R:

17) Ce este exclusiunea mutuală?

R: Se refera la faptul ca la un moment dat de timp doar o corutina executa o bucată de cod. În Kotlin exclusiunea mutuală se implementează în astfel încât nu sunt blocante: adică dacă un proces are acces acum la "lacat" pe o bucată de cod atunci el așteaptă că alte procese să-si facă treaba iar el va primi acel "lacat" când va fi liber ca apoi să ruleze bucată lui de cod

18) Ce este interbloarea (deadlock)?

R: Un proces așteaptă ceva ce a blocat (ocupat, folosit) celalalt și viceversa.

(LIVELOCK -> problema filosofilor -> spre deosebire de deadlock, nu se blochează, ci se încearcă mereu)

19) Diferența între thread-uri simple și thread-uri locale.

R:

20) Metode de wait și notify.

R: Wait - opreste un thread până la funcția notify îi spune să revină în execuție

Notify - trezeste din somn un anumit thread

21) Memorizarea (mecanism de caching) in contextul paralelismului.

R:

22) Ceva cuvinte cheie... (fold, ...)

23) Modalitati de sincronizare

R: Variabile atomice, lock-uri, actori

24) Care e diferenta intre memoria comună și cea cu transfer de mesaje?

R: În memoria cu transfer de mesaje, datele sunt transmise către thread-uri prin intermediul unor cozi pentru a evita apariția problemelor

R: În cazul memoriei comune fiecare thread ia datele din același loc din memorie, pot apărea probleme

25) Relatia intre corutine si threaduri.

R: Corutinile folosesc mult mai puține resurse și sunt foarte lightweight în comparație cu threadurile (putem lansa și 100k corutine fără să avem probleme)

26) Cum pot fi mapate corutine pe threaduri?

R: Corutinile își pot porni execuția pe un thread și pot termina pe orice altul

27) Relatia intre corutine si JVM?

R: Corutinile pot fi lansate în cadrul unui thread și sunt active atât timp cât thread-ul este activ.

28) Ce este un actor? (în contextul corutinelor)

R: Un actor este format din 3 lucruri (o corutina, un canal de comunicare și o stare internă)

Intrebari Curs 9

TutorialsPoint

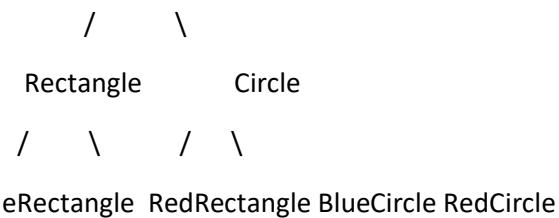
1) Explicati modelul Bridge.

R: Modelul Bridge este un model structural care iti permite sa separi o clasa mare in doua ierarhii separate, abstractizare si implementare care pot fi dezvoltate independent una de cealalta. Practic decuplam abstractizarea de implementare.

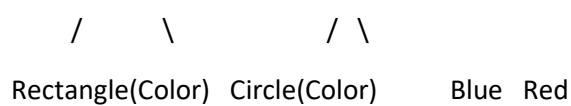
2) Cand utilizam modelul Bridge?

R: Cand avem ierarhii preferam sa folosim acest model

----Shape---



----Shape--- Color



3) Explicati modelul Composite.

R: Modelul composite este un model de partitionare si descrie un grup de obiecte care sunt tratate ca si o singura instanta a unui acelasi tip de obiect

4) Cand utilizam modelul Composite?

R: Cand vedem ca mai multe tipuri de obiecte sunt tratate in acelasi fel, repetandu-se codul pentru fiecare dintre ele, atunci e o idee buna sa folosim modelul composite, tratandu-le pe toate omogen.

5) Explicati modelul Facade?

R: Modelul Facade ascunde complexitatea sistemului si pune la dispozitie o interfata utilizatorului ce poate fi folosita pentru a utiliza functionalitatatile sistemului

6) Cand utilizam modelul Facade?

R: Cand construiesti o biblioteca (de ex), vrei sa ii oferi clientului "functiile" pe care le folosesti.

- Nu vrei sa-i arati functiile pe care le folosesti doar tu, intrinsec. Il arati doar ce poate reutiliza el.

7) Explicati modelul Decorator.

R: Modelul Decorator este un model structural ce permite adaugarea de noi functionalitati unei clase, fara a-i afecta functionalitatatile anterioare si modul cum relationeaza cu alte clase.

8) Cand utilizam modelul Decorator?

R: Ex. un context Manager (in python, context manager poate fi folosit ca un decorator)

9) Explicati modelul Proxy. (proxy = un intermediar)

R: ((Intermediarul proceseaza cererea noastra si o modifica sub o anumita forma pt a ne oferi rezultatul))

- Practic, proxy proceseaza o anumita cerere, un task pe care i-l dam.

10) Cand folosim Proxy?

R: Ex aplicatii de control parental: copilul vrea sa intre pe un URL (e trimis URL-ul ca si cerere unui proxy), iar daca acesta este permis, atunci copilul va putea intra pe site. Daca nu, va primi un mesaj "Nu ai voie!".

11) Explicati modelul Flyweight.

R: Flyweight e un model de proiectare structural care ne permite sa salvam mai multe obiecte in memorie prin pastrarea unor parti comune mai multor obiecte in loc sa punem obiectele intregi.

12) Ce este un model comportamental? (Behavioral)

R: Se ocupa de descrierea modului de interactiune intre clase

Descrie fluxul de control al unei aplicatii

13) Explicati modelul Chain of Responsibility.

R: Clientul trimite o cerere, iar aceasta este trimisa de la clasa la clasa pana se gaseste una care o poate procesa.

14) Cand utilizam un lant de responsabilitati?

R: De exemplu cand dorim tratarea unei cereri intr-o anumita ordine.

De exemplu, daca avem un request pe web, la tratarea lui il putem trece prin mai multe middleware-uri

de exemplu un middleware de securitate, care sa verifice daca persoana este logata, iar abia apoi ii putem trimite informatiile despre contul bancar, de exemplu

15) Explicati modelul Observer.

R: Observer este un model de comportament in care avem un Subject si mai multi observatori, iar cand se schimba ceva intr-o instanta a unui obiect de tip Subject toti observatorii lui trebuie anuntati, fiind acestia depind de subject

16) Cand folosim modelul Observer?

R: Cand avem o relatie one to many iar componentecele many depind de componenta principala.

17) Explicati modelul automatului finit.

R: Are la baza modelul State.

18) Explicati modelul Visitor.

R: Visitor e un model de proiectare comportamental care ne permite sa separam algoritmii de obiectele asupra carora opereaza.

19) Explicati modelul Command.

R: Gestioneaza realizarea unei actiuni

20) Explicati modelul Memento.

R: Undo/redo

21) Explicati modelul Iterator.

R: Gestioneaza parcurgerea unei colectii de elemente.

22) Explicati modelul Strategy.

R: Se schimba obiectul in fct de strategie abordata (Ex. o parcare cu plata care are pret diferit de stationare pt zi si pt noapte)

23) Explicati modelul Mediator.

R: Un model care are ca scop reducerea complexitatii.

Intrebari Curs 8

1) Ce este un model de proiectare (un design pattern)?

R: Modele de proiectare

2) Cum a aparut termenul de model de proiectare (termenul de design pattern)?

R: -> Cristopher Alexander

3) Cum a evoluat conceptul de design pattern?

R: 1987 - Cunningham si Beck - limbaj

1990 - Gasca celor patru - catalog

1995 - Gasca celor patru - carte

4) Explicati modelele Real si Zuligoven ??? (not sure)

R: -model conceptual:descrie in termeni si concepte domeniul aplicatiei

-model de proiectare:descrie proiectarea software folosind constructii software

-model de programare:se fol. constructii de limbaj pentru a descrie forme

5) Unde sunt utile modelele de proiectare?

R: Modelul aplicatiei

GoF

OOP + ADT

6) Care sunt elementele unui model de proiectare?

R: -numele:trb. ales a.i. sa descrie pe scurt problema de proiectare

-problema:se descriu cazurile in care se aplica acest model

-solutia:descrie elem. care compun proiectul, relatiile si colaborarile dintre ele

-consecintele:rez. obtinute si compromisurile care trb facute cand se aplica modelul

7) Ce este o arhitectura inchisa?

R:

8) Ce sunt modelele creationale?

R:

9) Explicati fabrica de obiecte (factory method).

R:

10) Cand se utilizeaza fabrica de obiecte?

R: Clasa nu poate anticipa ce tipuri de obiecte trebuie generate

11) Explicati fabrica de fabrici de obiecte.

R:

12) Ce este singleton si unde se utilizeaza?

R:

13) Explicati modelul Builder.

R: Permite crearea de obiecte complexe pornind de la unele simple

14) Cand se utilizeaza modelul Builder?

R: Obiectul nu poate fi creat intr-un singur pas

Evitarea crearii de prea multi constructori pt acelasi obiect

15) Explicati modelul Prototype.

R: Prototype e un blueprint pe baza caruia se creeaza alte obiecte.

16) Cand se utilizeaza modelul Prototip?

R: Cand nu stim dinainte (din timp) detaliile obiectului pe care vrem sa-l cream.

17) Ce sunt modelele structurale?

R: Modalitatea de combinare a claselor si obiectelor

18) Explicati modelul Adapter.

R: Dorim folosirea unei librarii, dar interfata pe care vrem sa o folosim
nu este

19) Modelul POD

-Decuplam abstractizarea de modelul ei

Intrebari Curs 7

1. Ce sunt functiile parametrizate?

R: Functii bazate pe generice(argumentele sunt parametrizabile)

Any(Kotlin) = Object(Java)

Unit(Kotlin) = void(Java)

2. Ce sunt tipurile parametrizate?

R: Ca si containerele din C++, de ex List<Int>
parametrii au un tip(se specifica intre paranteze ascunse)

3. La ce se refera polimorfismul limitat superior?

R: Nu putem duce orice tip de date in orice tip de date

De ex pt comparable: se limiteaza la subgrupul asteptat care contine elemente ce pot fi comparate

R: Se refera la limitarea tipului de date la subtipurile celui mentionat.

R: In Kotlin restrictioneaza tipurile de parametri la subclasele unei anumite clase.

4. Ce sunt limitarile superioare multiple?

R: Limitare multiple: Ex: fun<T> minSerializable(first: T, second: T):T

where T: Comparable<T>, T: Serializable

-> se foloseste clauza "where"

5. Ce sunt tipurile generice de date?

R: Tipuri de date parametrizabile(au un parametru <T> care poate fi inlocuit cu Int, Double sau chiar ADT-uri)

ex Class<Int>

6. Modificatorii de tip?

R: out -> poate fi folosit doar ca tip de return

in -> poate fi folosit doar ca parametrii la functii

7. Ce este declaration site variance?

R: abilitatea de a specifica variance annotation la declararea clasei
iesire -> produs(out) -> covariantă
intrare -> consumat(in) -> contravariantă

7'. Ce este variance annotation?

R: un modificator aplicat unui parametru/argument generic, cu scopul de a-i declara varianta.
In Kotlin sunt 2 variance annotations: out si in

8. Explicati problema claselor duale producator - consumator

R: Clase care folosesc un parametru covariant out si un parametru contravariant in.

9. Explicati type projection

R: E util atunci cand cineva a declarat o clasa invariantă și eu am nevoie să fie folosită într-un mod covariant sau contravariant. Practic specificăm tipul de variantă la utilizare

11. Ce sunt functiile generice?

R: Sunt funcții ce au și ele la randul lor tipuri parametrice

12. Ce sunt constrangerile generice?

R: Limitarea superioară a constrangerilor

13. Ce este type erasure? (pierderea tipului)

R: La cast type instantele cu tip generic nu știu de ce tip sunt

14. Ce sunt tipurile de date algebrice?

R: Similar grupurilor din algebra - un set închis de tipuri și funcții
funcțiile pot folosi doar anumite tipuri(cu ceil)

15. Ce sunt clasele "sealed"?

R: Clasa a carei functionalitate e restransa la fisierul in care e descripta.

- contine operatiile asociate

16. Ce sunt colectiile?

R: Niste entitati ce permit tratarea unitara a mai multor elemente ca o multime matematica.

17. Ce operatii ne ofera colectiile lista?

R: add, remove, set, contains, get, isEmpty, size, indexOf

lastIndexOf, removeAt, clear()

18. Ce sunt liste in kotlin?

R: Colectii generice ordonate de elemente

19. Care e diferenta intre list si mutableListOf ?

R: Se pot adauga si sterge elemente

20. Ce sunt colectiile set?

R: O colectie de obiecte fara duplicate

21. Ce operatii ofera colectiile Set?

R: size, contains, add, remove

containsAll, isEmpty, retainAll

22. Ce este LinkedHashSet?

R: Implementarea unui set folosind un Hash_table

lista dubla în lanțuită, hash table

Ma protejează de ordinea aleatorie a lui hashSet, pastrează ordinea

de la inserare

contains = O(1)

size, add = O(1)

23. Ce este un treeSet?

R: TreeSet este o interfață din SortedSet care folosește un arbore ca metodă de stocare

log(n) - add, remove, contain

24. Ce este o colecție Map?

R: O colecție ce conține perechi de elemente

key -> value

suportă extragerea optimizată

cheile sunt unice

relații 1 la 1

25. Ce operații ne conferă colecțiile de tip Map?

R: containsValue, containsKey, isEmpty, size

keys, value, entries

put, remove, putAll

26. Cum se poate parcurge un map cu un for?

R: for elem in map.keys{

print("%v]: %v", elem, map[elem])

}

27. Avantaje LinkedHashMap? Avantaje TreeMap?

R: LinkedHashMap = HashMap + LinkedList (lista dublu inlantuita)

TreeMap -> implementare pe arbori red-black

28. Ce este colectia Array?

R: Pentru Backward compatibility

E un container ce retine un nr fix de elemente de un tip dat

29. Care este relatia dintre colectiile Java si colectiile Kotlin?

R: Colectiile Kotlin sunt mai bune. Se distinge o data dintre mutable si immutable

30. Ce sunt tipurile platforma?

Marcate cu !

Deseori librariile din Java(fiind un limbaj null-unsafe)

au metode ce returneaza un tip! deoarece Kotlin nu-si da seama daca rezultatul e nullable sau nu.

Practic ! spune ca poate fi si nullable si non-null, venind dintr-o platforma care nu are aceste clasificari

Se foloseste in string-uri, punem de exemplu

"%d".format(x)

31. Cand nu este imutabilul garantat in Kotlin?

In kotlin, e garantat de o interfata, dar Kotlin relaxeaza principiile fata de java, deci pot aparea probleme la combinarea celor 2 limbaje.

Practic, atunci cand interoperam cu java(tipuri cochetice).

Mutable - se poate modifica

Immutable - nu se modifica, creeaza o noua colectie cu update-ul facut

Intrebari Curs 6

1. Enumerati domeniile posibile de utilizare python

R: Backend, front end, inteligenta artificiala, securitate, aplicatii web, hardware low level

2. Care sunt paradigmile de programare ale lui Python?

R: functionala

orientata obiect

imperativa

procedurala

3. Ce tipuri de conversii explicite pt tipurile de date suporta python?

R: integer, string, complex, floating point, long

4. Explicati instructiunea try-except

R: except exception

5. Ce este o exceptie?

R: Este un eveniment care apare in timpul executiei programului si care opreste fluxul normal al instructiunilor. Cand programul gaseste o situatie care nu-i convine trimite o exceptie. Exceptia e un obiect python care reprezinta o eroare

6. La ce este utilizata pass?

R: De exemplu pentru a spune unei functii ca va fi interpretata mai tarziu.

7. Ce diferenta sunt in python fata de C in termeni de operatori simpli?

R: putere, //, membership si identitate la op pe biti

in python nu avem incrementari

8. Ce tipuri de operatori de atribuire sunt suportati in python?

R: =, +=, *=, %=, /= etc

9. Diferenta dintre op de apartenenta si cei de identitate

R: daca un obiect e intr-o seventa
daca au aceeasi referinta

10. Ce standard de caractere este implicit in python?

R: UTF-8

11. De ce se poate executa un cod in interiorul unui sir in python?

R: Deoarece python este interpretat
cu exec si eval

12. Ce este eval?

R: Evalueaza un string ca o expresie python si returneaza rezultatul

13. Care e diferenta dintre o tupla si un dictionar?

R: Dictionarul e mutable, tuple e immutable

14. Care sunt atributele unui obiect de tip fisier?

R: close(), mode(), name(), softspace(), next(), read(),
readline(), seek(), tell(), write()

15. Care e metoda corecta de tratate a op cu fisiere?

R: Deschidem fisierul intr-un try tratem erorile si inchidem la final fisierul.
Daca sunt exceptii le tratam cu except

16. Cum se trateaza apelul unei functii in python?

R: Se opreste executia, se trimit valorile la parametrii formali, functia isi face treaba si
returneaza valoare

17. De ce pot intoarce valori multiple in python?

R: Cu tuple, poate sa descheteze tuple(unpacking iterable)

18. Cum pot simula transferul prin referinta la iesirea din functie in python?

R: tablouri, lista

19. La ce ne folosesc cuvintele cheie utilize ca parametru in python?

R: ca sa fim mai expliciti

20. Care sunt diferentele dintre C++ si suportul primar de OOP oferit de python?

R: this vs self(se trimite implicit, dar trebuia primit explicit)

constructorul este `__init__()`

indentare la python

nu avem ;

in python avem conceptul ala Duck typing si astfel exista cumva o interfata

generata automat

21. Cum se poate adauga rapid persistenta in python?

R: fisier, baza de date, shelve

22. De ce as fi nevoie sa construiesc un GUI de la 0 in python?

R: vrei sa modifici comportamentul default al acelui GUI

(de exemplu vrei sa schimbi background-ul si nu iti permite GUI existent)

Intrebari Curs 5

1. Ce este un eveniment?

R: Un semnal generat de catre o aplicatie, un sistem hardware.

Poate fi definit ca un tip de semnal generat de cate program.

Indica ca s-a intamplat ceva

Ex: buton de mouse, miscare de mouse,

2. Prezentati o posibila maniera de gestionare a unui eveniment, inclusiv OS-ul

R: Tastatura, Mouse -> OS -> coada evenimente -> programe -> OS -> monitor

3. Cum se gestioneaza dpdv arhitectural un eveniment?

R: Se baga intr-o coada de evenimente si cand trebuie sa fie tratate se scot pe rand din coada

??

4. Care este diferența dintre evenimentele sincrone și cele asincrone?

R: Sincron : trimit mesaj, astept confirmare

asincron : trimit toate mesajele, nu mai astept confirmare

mouse tastatura : asincron

5. Care sunt sursele comune pt evenimentele sincrone si asincrone?

R: Mouse-ul si tastatura

6. Cum se trateaza un eveniment dpvd al programatorului?

R: Se trimit la distribuitor care le trimit la un anumit gestionant

Polling (citire sequentiala intr-o bucla infinita a tuturor comenzilor dispozitivelor de intrare)

7. Care sunt avantajele procesarii orientate pe evenimente?

R: mai portabile

permit tratarea rapida a erorilor

se pot folosi in time-slicing

incurajeaza reutilizarea codului

merge mana in mana cu oop

8. La ce se refera EDP si care sunt componentele principale implicate?

R: Event driven programming

Generatoare de evenimente

Sursa evenimentelor

Bucla de evenimente

Gestionari de evenimente

Event mapper

Inregistrarea evenimentelor

9. Ce cuprinde diagrama de secventa specifica EDP?

R: AppUser - actiuni

Event Object - genereaza evenimente

Event Mapper - inregistreaza gestionarul de evenimente si distribuie

Event Handler - proceseaza evenimentele

10. Ce este dispecerul in EDP?

R: Cel care controleaza evenimentele

Distribuie evenimentele

11. Ce inseamna wizzy-wyg si care este legatura cu EDP?

R: What you see is what you get

12. Explicati look-n-feel?

R:

13. Care sunt criteriile minimale care trebuie implementate de un GUI pentru a se mula pe utilizator?

R:

14. La ce se refera PIC correlation?

R:

15. Ce este un GUI chat?

R:

16. Explicati MVC-ul

R: Model view controller - modelezi datele, view inseamna cum arati datele, controller da operatii modelului

Controller modifica starea modelului, cand se schimba starea view-ul afiseaza noua stare

17. Ce este SDL si unde se foloseste?

R: Este o biblioteca folosita la creare de software

SimpleDirectMediaPlayer

18. Care sunt modulele SDL si echivalentele lui cu DIVX?

R: Video - DirectDraw

Gestiunea evenimentelor - DirectInput

Joystick - DirectInput

Audio - DirectSound

CD-ROM - NU are echivalent

Firul de executie - NU are echivalent

Timere - NU are echivalent

19. Ce este list comprehension? Cele mai comune utilizari

R: Scriem liste/structuri multidimensionale pe o linie

list1 = [i**2 for i in range(0,8)] (pentru for)

list1 = [i for i in range(0,8) if(i%2==)] (pentru if else)

list1 = [i**2 if (i%2==) else i**3 for i in range(0,8)] (if else)

//matrice : list1 = [j for j in range(0,6) for i in range(0,3)]

20. Ce este un eveniment virtual? Cum se

R:

21. Ce este si unde este necesara organizarea matriceala a entitatilor intr-un GUI python?

R: Fiecare element este pe o linie si o coloana.

22. Explicati maniera arborescenta de creare a meniurilor in tkINTER

R: Creez submeniurile si apoi le asamblez intr-un meniu mare

23. Ce ar trebui urmarit cand trebuie tratat un eveniment Frame

R:

24. Explicati tratarea evenimentelor in Android

R:

25. Cum se pot adauga servicii in Android?

R:

Polling : bucla infinita

- Interrupt-driven : asteapta aparitia unor intreruperi
- Widget : obiecte din cadrul unui GUI orientat obiect
- Sincron : trimit mesaj, astept confirmarea ca primul a ajuns, apoi trimit al doilea mesaj. eu rezerv resurse care saprimeasca evenimentele
- asincron : trimit toate deodata nu astept confirmare, nu ne mai batem capul daca va fi receptionat si tratat, nu se mai blocheaza aplicatia sa astepte confirmari

Intrebari Curs 4

1. Cum definiti proiectarea aplicatiilor software?

R: Cu ajutorul diagramelor UML

Proiectarea software reprezinta un proces de rezolvare a unor probleme, obiectivul fiind sa se gasesca si sa se descrie o cale de a implementa necesitatile functionale ale sistemului, tinand cont de constrangerile clientului

2. Care e diferența intre must have si nice to have ?

R: produsele principale(must have), cele secundare(nice to have)

3. Definiti o componenta?

R: Entitate fie fie soft fie hard care are un rol bine determinat si poate fi inlocuita cu alta componenta

4. Din ce este alcătuit un sistem daca il analizam din prisma analizei modulare? Subsisteme, componenete si module

R: Subsisteme, componente si module

5. Ce este modelul domeniului?

R: System,susbsisteme implementat de Component, care este definit la nivelul limbajului de catre module si framework

6. Ce este UML(unified modeling language)

R:Este un limbaj grafic adoptat mondial folosit pentru reprezentarea,vizualizarea si documentarea componentelor unui sistem software de dimensiuni mari

7. Ce este modelul de proiectare si implementare in cascada? AVANTAJ

R: Fiecare nou proces se implementeaza dupa ce acela anterior este complet realizat si testat.

Etape: specificarea cerintelor, analiza sistemului, proiectarea sistemului, implementarea sistemului, instalarea sistemului, mentenanța sistemului.

8. In ce conditii se pot suprapune AGILE si WATERFALL?

R: In waterfall, o etapa viitoare nu poate incepe inainte ca o etapa din trecut sa se fi incheiat (o etapa noua poate depinde de o etapa din trecut)

In AGILE, noile etape depind de cele vechi, dar se proiecteaza simultan (pot depinde unul de celalalt in paralel)

9. Enumerati modelele sistem :

R: Modelul obiect (structura sistemului, obiectele si relatiile)

Modelul functional (fluxuri de date din sistem)

Modelul dinamic (cum reacționează sistem la stimuli externi)

10. Enumerati modelele task-urilor: ___, planificarea

R: Harta PERT (care sunt legăturile dintre task-uri),

Planificarea (cum poate fi aceasta realizată în durata de timp rezervată?)

Harta organizatională (care sunt rolurile în proiect?)

11. Eumerati mecanismele centrale ale unei aplicatii OOP

R:

12. Cum se defineste un model dpdv al UML:

R: Un model este o descriere completa a unui sistem dintr-o perspectiva particulară

12'. Tipuri de proiectare specifice

R: proiectare arhitecturală

proiectarea claselor

proiectarea interfetei vizuale

proiectarea bazelor de date

proiectarea algoritmilor

proiectarea protocoalelor

13. La ce este buna abordarea bazata pe componenete in proiectare?

R: Componentele pot fi inlocuite ulterior cu unele mai bune, acest lucru fiind posibil datorita nivelului ridicat de incapsulare.

14. Dati exemple de aplicare a principiului cresterii coeziunii

R: Grupeaza ce este apropiat cat mai mult

15. Exemple de aplicare a principiului reducerii cuplarii(cuplare = interdependente in

R: De exemplu: reducem numarul de dependente intre clase, daca fiecare clasa ar fi un nod intr-un graf ne-am dori sa minimizam numarul de arce, sa fie cat mai independente pentru a ne fi usor sa modificam lucrurile ulterior

16. La ce se refera principiul de baza in gestiunea abstractiilor?

R: ascund padurea ca sa nu ma incurc de copaci
entitate generica rezultate din analize (nu e necesar sa stim detalii de implementare (tip de date))

17. Care sunt principiile complementare reutilizarii?

R: proiecteaza pentru reutilizare, proiecteaza prin reutilizare

18. La ce se refera proiectarea pt flexibilitate?

R:Anticiparea activa a modificarilor pe care un proiect le poate suferi si pregatirea din timp pt acestea

R: Anticiparea schimbarilor ce pot aparea pe parcurs. (reducerea cuplarii + cresterea coeziunii)

19. Cum se gestioneaza problemele de inlocuire sau disparitie a unor componente

puse la dispozitie de o tehnologie sau framework?

R: De ex, in cazul inlocuirii, se asigura backwards compatibility prin respectarea principiilor SOLID (Liskov substitution)

20. La ce se refera proiectarea pt portabilitate? Programam aplicatia astfel incat sa fie cat mai portabila pe mai multe sisteme de operare

R: Utilizarea de limbaje interpretate(de ex Java), ce nu sunt dependente de arhitectura hardware

21. La ce se refera proiectarea pt testabilitate?

R: Iei masuri pentru usurarea testarii

Masuri precum:

- utilizarea design patternurilor
- utilizarea unor feature-uri care accepta linie de comanda

21') Scopurile proiectarii OOP

R:

- > usurinta in modificare sau adaugarea unor noi functionalitati
- > gestionarea independentei intre clase si pachete
- > scaderea motivelor pt care modelul s-ar putea schimba daca ar aparea schimbari in clase/pachete

22. La ce se refera coeziunea claselor?

R: plasarea metodelor care opereaza pe anumite date cat mai aproape de datele respective

23. Ce este principiul de raspundere unica?

R: Fiecare clasa ar trebui sa faca o singura chestie

24. Ce este principiul separarii interfetelor?

R: Clientii nu trebuie sa fie obligati sa depinda de interfete de care nu au nevoie

25. Principiul inchis/deschis?

R: Clasele trebuie sa fie deschise la extindere si inchise la modificare

26. Ce este BPP-ul?

R: Bune practici de proiectare (Divide and conquer)

27. Principiul substitutiei Liskov?

R: Clasele derivabile trebuie sa fie substituibile in clasele de baza. Adica clasele derivate trebuie sa aduca functionalatati noi, nu sa le modifice pe cele vechi

28. Principiul dependentei inverse?

R: Modulele de nivel arhitectural superior nu trebuie sa depinde de cele de nivel inferior. Ambele trebuie sa depind de abstractii, care la randul lor nu depind de implementarile concrete

29. TDD = Test-Driven Development

R:

scrie(un test)

executa(toate testele)

scrie(codul tinta)

executa(testele)

refactorizeaza(codul)

Intrebari Curs 3

1.Ce este descompunere functională dpdv al oop?

R: Înseamnă ca atunci când proiectăm un program să ne facem un plan, să distingem niste funcționalități și apoi să le modulăm și să le algoritmizăm împărțindu-i într-un număr de pași

Ex: Să se acceseze descrierea unor forme existente într-o bază de date apoi să se afiseze aceste forme

1. Identifică lista de forme în baza de date
2. Deschide lista de forme din baza de date
3. Ordenează lista de forme conform cu un set de reguli
4. Afisează formelele pe monitor

2.Care sunt principalele probleme ale specificațiilor de la client?

R: Clientul nu are ce vrea și da info incomplete, gresite, îl mai vin idei pe parcurs, greseli de comunicare/intelegeră

3.Care sunt motivele pt care clientul nu da ce trebuie?

R: Diferente de comunicare, clientul omite informații care crede că sunt evidente deși sunt relevante pentru proiectant, nu se gandeste înainte la toate aplicațiile

4.Ce efect au coeziunea scăzuta și cuplarea stransă?

R: coeziune - că de apropiație sunt operațiile dintr-o metodă

cuplarea - că de stransă este legătura dintre două metode

integritate internă(coeziune stransă)

relații directe, vizibile, flexibile și directe(strans cuplate)

Coeziunea scăzuta = operațiile dintr-o metodă sunt strans legate între ele și o modificare mică se propagă peste tot

5.Care sunt pașii din proiectarea OOP?

R: Izolează obiectele din lumea reală, abstractizează obiectele, determină responsabilitatea grupului față de alte grupuri

6.Ce este brainstorming-ul?

R: aruncari spontane de idei, persoanele care participa ar trebui sa stie pt a da idei pertinente

7.Ce pasi implica metodologia de proiectare OOP?

R: brainstorming - pentru a localiza clasele posibile, se considera toate ideile
filtrarea claselor - pt a gasi duplicatele si a elimina pe cele in plus
scenarii - necesare pentru a fi siguri ca s-a intelese colaborarea intre obiecte
algoritmi - sunt proiectati pentru a defini toate actiunile pe care clasele trebuie sa le poata efectua

8.La ce se folosesc fisierele CRC?

R: Pt a mentine informatiile primare despre clase, subclase, superclase, responsabilitati, colaborari, cum relateaza intre ele clasele

9.Care este flow-ul general pt rez si proiect unei probleme in OOP?

?R: Se grupeaza elementele in clase

10.Cum creeaza Kotlin obiectele asociate unei clase?

?R: Instantiere prin constructori

11.Care este rolul lui this?

R: Ne ajuta sa ne referim la obiectul respectiv(in metodele unei clase de ex)

this -> instanta curenta

cand vrem sa ne referim la instanta externa

12.Care este rolul lui scope?

R:

13.Cum se poate face controlul fluxului de executie ca o expresie?

R: cu if, else, try, catch

14.Ce este NULL? Ce rol are NULL?

?R: Null este un pointer la nimic. Null nu se poate converti la un tip

...

15.Explati verificarea si conversia de tip in Kotlin

R: Implicita(dar poate fi facuta explicita).

Verificam cu "is" = "instance of"(Java)

Kotlin face cast a unei variabile la ultima verificare de tip

16.Cum se poate utiliza when ca un switch case?

R: In loc de default se pune else; la case se pune doar valoarea pe care sa o ia variabila din when()

17.Cum se poate utiliza when ca o expresie?

?R: Asignam valoare unei variabile cu when

18.Ce este clasa anonyma?

R: Este o clasa ce se poate utiliza pt apeluri singulare si nu este recomandata

19.Ce sunt clasele pt gestiunea datelor?

R: Clasele de tip enum si data(pastreaza date)

20.Ce sunt metodele statice in Kotlin?

R: Nu avem in clase metode statice(in Kotlin). La nivel de pachet avem metode statice.

21.Ce sunt obiectele companion si unde se folosesc?

R: Functiile companion care apartin unei clase iau locul metodelor statice, putand fi apelata fara un obiect instatiat

22.Suporta Kotlin mostenirea simpla direct?

R: Mostenirea multipla se simuleaza

Se suporta mostenirea simpla directa. Clasele fara parinte vine de la clasa de baza(object, aia de baza) Pt a face o clasa derivabila avem cuvantul cheie "open"

23.Ce sunt functiile cu expresie unica?

R: Functie ce contine o simpla expresie si folosind inferenta de tip isi da seama pe baza evaluarii expresiei ce tip are de returnat

24.Ce sunt functiile membru?

R: Functii in interiorul unei clase, obiect sau interfata.

25.Cand sunt recomandate functiile locale?

R: Functii mici, definite in interiorul altor functii

Cand iesim din scope-ul in care au fost definite nu le mai putem folosi

Dorim sa ascundem detalii de implementare a unei functii mai mari

Cand nu mai avem nevoie de ele in afara scope-ului

26.Ce sunt functiile top-level?

R: Functiile globale din C++

27.Ce este list comprehension si cum se poate face in Kotlin?

R: Facem liste pe o linie, e mai rapid, am facut in graba

izoleaza obiecte din lumea reala, abstractizeaza obiectele care au prop si comportamente similare
determina resp grupului dpdv al interactiunii cu alte grupuri

Metodologie POO

brainstorming - pentru a localiza clasele posibile

filtrarea claselor - pt a gasi duplicatele si a elimina pe cele in plus

scenarii - necesare pentru a fi siguri ca s-a inteles colaborarea intre obiecte

algoritmi - sunt proiectati pentru a defini toate actiunile pe care clasele trebuie sa le poata efectua

Intrebari curs 2

studentul la curs se afla in starea s0 plictisit

pauza = 1

studentul tranzitioneaza in stare 2 numit relaxat

pauza = 0

studentul revine la starea s0 plictisit

1. Model church-turing

R: Spune ca un calcul poate fi facut de o masina doar daca este calculabil in sens Turing. Orice problema de calcul este calculabila daca se incadreaza in una din categoriile: general recursive functions, lambda-computable sau Turing computable(de fapt toate 3 sunt echivalente)

2. Care sunt problemele masinii virtuale java sau c# ?

R: Performanta (datorita interprotoarelor) este 1-3 ori mai lenta decat implementarile similare in C(unsafe). Java e pe cale de disparitie, dar poate exista ptc sunt deja multe aplicatii in java(20 ani)

3. Ce este polyglot?

R: Polyglot este un limbaj de programare care are mai multe interprotoare(graal etc) si practic poti folosi pentru a rula mai multe limbaje. Se foloseste de AST pentru a scapa de diferentele de sintaxa

4. Implementare AST? De ce e mai buna decat o masina virtuala?

R: ast = abstract syntax tree; utilizare ast = ascunde diferentele de tratare a fiecarui limbaj; ajuta sa punem cap la cap instructiunile din toate limbajele -> nu mai conteaza din ce limbaj vin instructiunile

5. Ce este hotspot? Ce este graal?

R: Graal este interpretorul lui Polyglot. Hotspot este o masina virtuala java. Graal este scris in Java si se bazeaza pe Hotspot care e scris in C

6. Care sunt avantajele Graal fata de hotspot?

R: hotspot este doar proiectul initial, graal e bazat pe hotspot si aduce alte functionalitati: libraria truffle(care permite accesul la limbaje precum R,Ruby,js etc) + tool-uri + compiler nou ==> avantaj de timp la rularea programelor

7.Care sunt limbajele suportate de Graal la ora actuala?

R: js,ruby,r,python,java

8.Ce este calculul functional?

R:

9.Ce este o functie lambda?

R: Putem crea o functie fara sa le asignem un nume explicit(functii fara nume)

10.Explcati functiile Curring(?)

R: functii cu argument multiplu pot returna mai multe valori + functii ce primesc serial argumentele(primul apel are n parametri, urmatorul n-1, urmatorul n-2 pana se ajunge la functia cu un parametru care returneaza o valoare = recursivitate)

11.Diferenta dintre tipurile dinamice si tipurile statice de date?

R: tipuri statice = fiecare variabila este bine legat de un tip de data(de ex int c++). NU SE POATE SCHIMBA TIPUL UNEI VARIABILE IN TIMPUL PROGRAMULUI; tipuri dinamice = de ex python

12.Limbaje ce suporta tipuri "tari" de date si tipuri "slabe" de date

R: python suporta ambele(de ex poti face int(1) + float(2.3), dar nu poti 1 + "2.3"), javascript(tip slab de date) de ex: 1 + "2" = "12" -> face conversii implice

13.Enumerati tehniciile curente de optimizare a unui cod

R: reorganizare noduri AST, evaluare partiala

14.Reorganizarea AST? Avantajele utilizarii acestei metode

R: Reorganizarea unui AST practic schimba structura arborelui astfel incat sa se evite pasi in plus, sporind practic eficienta codului

15.Ce inseamna evaluarea partiala?

R: Evaluatezi diferite parti ale programului in valoarea optimizarii(de exemplu partile pe care deja le stii inainte de compile time, codul)

16.Ce este un evaluator partial?

R: Un evaluator partial particularizeaza programele. De exemplu, daca un program este o functie prog definita pe date intrare(cunoscute la compile time) + date ce urmeaza a fi introduse in program de utilizator cu valori in Output, atunci un evaluator partial transforma programul intr-o versiune noua ce incapareaaza datele deja cunoscute, specializand practic programul si facandu-l mai eficient(eventual se rescriu liniile de cod)

17.La ce se refera specializarea unui program?

R: Pentru un set de date(in1) programul se specializeaza, scriindu-se mai eficient instructiunile(de ex trecere de la program structurat la program nestructurat)

18.Optimizare in Graal?

R: Mai putine instructiuni datorita evaluarii partiale

19.Metode depanare Graal?

R: Loggings(trimitere de mesaje), mecanismul try-throw-catch

20.Ce sunt metricele unui program si de ce ne-ar trebui?

R: De ex: memorie RAM utilizata, viteza de executie, memorie utilizata(octeti),numar linii cod !!CounterKey(timp scurs). Poate gasesti leaks de memorie

21.La ce se refera dumping-ul?(poate unui executabil)

R: Se ingheata memoria pt functia care s-a oprit incorrect(a crapat) din functionare si se salveaza datele cu scopul de a face debug pe ele

link ascuns = dino.zip

lambda calculabila = calculabila conforma modelului turing

mutable types = cand dorim sa ii asociem o noua valoare pur si simplu se schimba(de exemplu lista in python)

immutable types = cand dorim sa ii asociem o noua valoare, de fapt se creeaza un nou obiect(cu alt id) spre care variabila noastra pointeaza

dynamic typing = nu e nevoie să declari tipul obiectului, și da el singur seama(ex python: `a = 7`, și da seama că e vorba de un int); face type-checking la run-time

static typing = trebuie să declari tipul obiectului la declarare(ex: `int a = 7`); face type-checking la compile time

strongly typed = nu face conversii implicite(de exemplu `1 + "2" = eroare`)

weakly typed = face conversii implicite(de exemplu în javascript: `1 + "2" = "12"`)

Intrebari Curs 13

1) Ce este calculul Lambda?

R: Este un model matematic formal in care folosim functii cu parametri, fara nume, pt calculul unor expresii.

2) Cum s-ar putea scrie un if cu un lambda utilizand doua obiecte?

R:

true = lambda x, y: x

false = lambda x,y: y

if = lambda p, x, y: p(x,y)

3) La ce se refera *args si **kwargs?

R: Permit introducerea unui nr variabil de argumente la o functie

args - tupla

kwargs - dictionar

4) La ce se refera functiile de prim nivel?

R: In python totul este obiect (chiar si functiile).

Functiile de prim nivel sunt ca niste obiecte cu atribute, pe care le putem inspecta

5) La ce se refera datele imutabile?

R: Datele imutabile se refera la datele care nu-si pot modifica continutul

6) La ce se refera functiile pure?

R: Functiile pure sunt cele fara efecte laterale.

7) La ce se refera abordarea impacheteaza - proceseaza - despacheteaza?

R:

8) Cum se realizeaza evaluarea la cerere?

R:

9) Ce este un generator recursiv?

R: yield from

10) Pentru ce utilizam modulul iterTools?

R: Pt a folosi functii care genereaza si proceseaza serii si sevante de numere (pt partea hardware sunt f bune deoarece se pot genera repede si usor semnale)

11) Cand utilizam iteratorii infiniti?

R: Sunt utili pt generarea semnalelor (sinus, cosinus etc.)

12) Cum se calculeaza eroarea prin acumulare?

R: Se aduna putin cu putin.

13) Care-i diferenta intre functiile "cycle" si "accumulate"?

R: cycle -> repeta argumentele la infinit

accumulate -> efectueaza o operatie asupra fiecarui argument

14) Cum se poate utiliza modulul iterTools pt a crea functii de ordin 2?

R:

15) Cand se utilizeaza functie "tee"?

R:

16) Cand utilizam operatorii "any" si "all"?

R: any -> macar una dintre "ele" sa respecte o anumita conditie (conditii cu || intre ele) -> returneaza true daca macar o conditie e true

all -> toate dintre "ele" trb sa respecte o anumita conditie (conditii cu && intre ele) -> returneaza true daca toate conditiile sunt true

17) Explicati data flatten.

R:

18) Ce este scurtcircuitul (nu ala electric)?

R: De ex intr-un if in care avem mai multe conditii cu "OR", daca prima cond e adevarata atunci nu mai trebuie verificate si celelalte -> scurtcircuit

19) Cum putem evalua deciziile in calculul functional?

R:

20) Ce este un "closure"?

R: Functiile in python sunt referinte catre obiecte. Deci ele pot suporta si apeluri de alte obiecte???
- inchiderile in python retin si contextul in care au fost incheiate

21) Care operatori pe colectii sunt mai rapizi si in ce domeniu?

R:

22) Care este diferența intre un decorator in stil macro si unul in stil OOP?

R: - stil macro (pe functie)
 - stil OOP (pe clasa)

23) Cum putem implementa un state machine (FSM)?

R:

- Utilizam lambda cand avem expresii destul de simple pt care nu merita sa cream o functie
- mapReduce are 2 etape -> mapare si reducere (exemplificam pe exemplu din lab)

1) mapare -> se elimina semnele de punctuatie si se fac litere mici, se sparge textul dupa spatiu si se fac perechi de genul

- ("ana", 1), ("are", 1), ("mere", 1), ...

2) shuffle & sorting (etapa intermediara) -> sorteaza

3) reducere -> suma valorilor pt elementele cu aceeasi cheie

In urma acestor operatiuni a rezultat practic un Word Counter.

Curs2

La ce e bun calculul Lambda?

Lambda calculul (engleză: lambda calculus) este un model de calcul care surprinde esențial, programările funcționale. Lambda-calculul este un limbaj de programare aparent foarte simplu (cu doar trei constructii sintactice), dar care este complet, în sensul în care poate descrie orice calcul efectuat de un calculator.

Expresiile Lambda pot fi folosite pentru a da un înțeles formal funcțiilor Curry.

Ce este o funcție Curry?

Functie curry: functie care returneaza o noua functie atunci cand este aplicata pe mai putine argumente decat asteapta ea

Care este avantajul funcțiilor Curry?

Sunt mult mai flexibile decat funcțiile bazate pe tuple, în special datorita faptului ca pot fi aplicate lor parțial.

```
add' 1 :: Int → Int  
take 5 :: [Int] → [Int]  
drop 5 :: [Int] → [Int]
```

Convenții specifice funcțiilor Curry

Pentru a evita folosirea în exces a parantezelor atunci când se folosesc funcții Curry s-au adoptat două convenții simple:

- Utilizarea săgeții → care face asociere la dreapta

```
Int → Int → Int → Int
```

- Ca o consecință apare o a doua convenție: funcțiile vor folosi asocierea la stânga

```
mult x y z
```

Evaluator parțial ?

Programele care efectuează evaluarea parțială, extinderea beta și anumite optimizări ale programelor, sunt studiate cu privire la implementare și aplicare. Sunt descrise două implementări, una de evaluare parțială „interpretativă”, care operează direct pe programul care urmează să fie parțial evaluat și un sistem de „compilare”, în care programul care va fi parțial evaluat este folosit pentru a genera un program specializat, care la rândul său este executat pentru a face evaluarea parțială. Sunt descrise trei aplicații cu cerințe diferite pentru aceste programe. Se dau dovezi pentru echivalența utilizării sistemului interpretativ și a sistemului de compilare în două din cele trei cazuri. Este discutată utilizarea generală a evaluatorului parțial ca instrument pentru programator împreună cu anumite tehnici de programare.

Dumping ?

În afara de logarea Graal furnizează și suport pentru generarea de informații detaliate despre anumite结构ure ale compilatorului

Curs 3

Paradigme orientate obiect

Kotlin

Variabile var & val

Variabilele locale sunt de obicei declarate și initializate în același timp, caz în care tipul variabilei este *dedus* ca fiind tipul expresiei cu care o initializezi:

```
număr var = 42  
var mesaj = "Bună ziua"
```

Frecvent, veți vedea că în timpul vieții variabilei dvs., trebuie doar să se refere la un singur obiect. Apoi, îl puteți declara cu **val**(pentru „valoare”) în loc de:

```
val message = "Hello"  
număr val = 42 ///// val=constanta, nu se poate modifica
```

Inferenta de tip

Desi kotlin este un limbaj cu tipuri tari de date el nu necesita declararea obligatory de tip, deci ca si python suporta inferenta de tip

```
fun plusOne(x:Int)=x+1
```

Cateodata este util sa lucrăm explicit:

```
val explicitType: Number= 12.3
```

Tipuri de date

Ca si in python, in kotlin orice este un obiect.

NUMERE

```
val int = 123  
val long = 123456L  
val double = 12.34  
val float = 12.34F  
val hexadecimal = 0xAB  
val binary = 0b01010101
```

Conversii implicate?

```
val int = 123  
val long = int.toLong()  
val float = 12.34F  
val double = float.toDouble()
```

Long	64
Int	32
Short	16
Byte	8
Double	64
Float	32

toByte(),
toShort(),
toInt(),
toLong(),
toFloat(),
toDouble(),
toChar().

Operatori pe biti

- Nu sunt definiți ca operatori speciali dar pot fi apelați ca atare
- val leftShift = 1 shl 2
- val rightShift = 1 shr 2
- val unsignedRightShift = 1 ushr 2
- val and = 1 and 0x00001111
- val or = 1 or 0x00001111
- val xor = 1 xor 0x00001111
- val inv = 1.inv()

Variabile logice (bool)

- val x = 1 val y = 2 val z = 2
- val isTrue = x < y && x < z
- val alsoTrue = x == y || y == z

Caractere (Siruri de caractere, tablouri, tablouri cu tip, Exemple initializari variabile)

- Sunt clasice cu simple ghilimele și suportă caracterele de control standard - \t, \b, \n, \r, ', ", \\", \\$.

Siruri de caractere

- val string = "string with \n new line"
- mai există ceva numit sir brut(raw)

Tablouri

- val array = arrayOf(1, 2, 3)
- val perfectSquares = Array(10, { k -> k * k })
- val element1 = array[0] val element2 = array[1] array[2] = 5

Tablouri cu tip

- ByteArray, CharArray, ShortArray, IntArray, LongArray, BooleanArray, FloatArray, and DoubleArray

Exemple inițializări variabile

- val aToZ = "a".."z"
- val isTrue = "c" in aToZ
- val oneToNine = 1..9
- val isFalse = 11 in oneToNine

Cicluri

Gestiunea exceptiilor

Instantierea unei clase

```
val file = File("/etc/nginx/nginx.conf")
val date = BigDecimal(100)
```

Egalitatea de referinta si de structura

- pentru egalitatea de referință vom folosi === sau !==
- exemplu de gândire greșită:
 - val a = File("/mobydick.doc")
 - val b = File("/mobydick.doc")
 - val sameRef = a === b //va fi False
- pentru egalitatea de structură vom folosi == sau !=
 - val a = File("/mobydick.doc")
 - val b = File("/mobydick.doc")
 - val structural = a == b //va fi True

This

- class Person(name: String)
- { fun printMe() = println(this) }
- I se mai spune și "current receiver"

Scope

```
class Building(val address: String)
{
    inner class Reception(telephone: String)
        { fun printAddress() = println(this@Building.address) }
}
```

Vizibilitate (public, private, protected, internal)

class Person	internal class Person
{ private fun age(): Int = 21 }	{ fun age(): Int = 21 }

NULL

- var str: String? = null
- NULL SAFETY!!!!**
- Nullable and non-nullable types**
- val name: String = null // grr...errr
 - var name: String = "mike"
 - name = null // grr...errr
 - val name: String? = null // i'mm happy
 - var name: String? = "harry"
 - name = null // i'mm happy
 - fun name1(): String = ... fun name2(): String? = ...

Conversia explicită de tip

- fun length(any: Any): Int
- { val string = any as String return string.length }
- val string: String? = any as String
- atunci:
- val any = "/home/mike"
- val string: String? = any as String
- val file: File? = any as File

Interfata Map

Map este un obiect care asociază chei la valori și nu permite duplicate pentru cheie. Deci fiecare cheie are asociată o singură valoare.

Interfata List

Lista este o colecție ordonată (fiecare element este caracterizat prin poziția sa în lista) și permite și duplicate. Interfata List pe lângă operațiile mostenite de la interfata Collection conține operații pentru:

- accesul elementelor pe baza poziției lor în lista
- căutarea unui obiect specificat prin returnarea pozitiei
- obținerea unui iterator pentru efectuarea parcurgerilor specifice listei, se extinde semantica interfetei Iterator
- obținerea unei subliste

Interfata Set

Interfata Set reprezintă o colecție care nu permite duplicate. Modelază multimea din matematică. Nu conține metode noi, doar cele mostenite de la interfata Collection, la care adaugă restricția cu duplicatele. Două obiecte de tip Set sunt egale dacă ele contin aceleasi elemente

Curs 4

Principiile SOLID

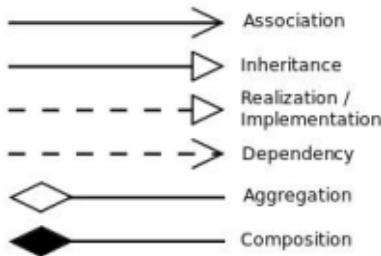
Ce este de fapt UML?

- *" Unified Modeling Language (UML) reprezintă un limbaj grafic pentru vizualizarea, specificarea, dezvoltarea și documentarea componentelor unui sistem software de dimensiuni medii sau mari.*
- *UML oferă o manieră standard pentru a crea schema unui sistem pornind de la aspecte concrete cum ar fi blocuri de cod, scheme pentru bazele de date, componente reutilizabile și ajungând la aspecte abstracte precum capturarea fluxului de desfășurare a unei afaceri sau funcții ale sistemului."*

Sageti

Orice sageata are:

- Baza săgeții - Clasa luată în considerare, sursa (subiectul 1).
- Vârful săgeții - se referă la cine este în legătură este subiectul 1, de cine depinde ierarhic, "target" (subiectul 2).
- În cazul vârfului romb, clasa unde se află vârful conține un potențial tablou (array) de elemente de tipul clasei din partea săgeții de la capătul opus.



Association - relație de asociere între două clase (clasa sursă folosește membri din țintă (target)).

- Dependency - relație de dependență între două clase (clasa sursă folosește clasa din țintă ca un parametru sau ca o variabilă în interiorul funcțiilor).
 - Inheritance - moștenire (clasa sursă derivează clasa țintă, adăugându noi funcționalități).
 - Realization - Implementarea unei interfețe (clasa sursă implementează interfața din țintă)
 - Aggregation - agregare (clasa sursă folosește de mai multe ori clasa din target)
 - Composition - compozitie (clasa sursă folosește de mai multe ori clasa din țintă și în același timp o derivă (prin moștenire). Un exemplu pentru acest caz este structura arborescentă a unui GUI, unde fiecare element grafic conține un tablou de alte elemente grafice pe care îl poate deriva prin moștenire)

Principiile SOLID

Principiul Responsabilitatii Unice (Single) - O clasă trebuie să aibă o singură rațiune pentru a se schimba.

Principiul Inchis / Deschis (Open / Closed) - - Entitatele software trebuie să fie deschise pentru extindere dar închise pentru modificare

Principiul Substitutiei Liskov- - Copiii claselor nu au voie să încalce definițiile de tip din clasa părinte. Adică funcțiile care folosesc referințe la clase de bază, trebuie să poată manipula într-un mod transparent instanțele claselor derivate din acestea.

Principiul separarii Interfetelor- Nici un client nu trebuie forțat să depindă de metode pe care nu le folosește. Numărul de membri din interfață care este vizibil pentru clasele dependente trebuie minimizat. Clasele mari vor implementa mai multe interfețe mai mici care grupează funcțiile după maniera lor de utilizare.

Principiul Dependentei inverse- modulele de nivel arhitectural superior nu trebuie să depindă de cele de nivel inferior. Ambele trebuie să depindă de abstracții care, la rândul lor nu trebuie să depindă de detalii. Practic detaliile depend de abstracții, nu invers. Dacă aceasta dependință nu este vizibilă în faza de proiectare atunci ea se construiește.

Le reluaaaaaaaaaaaaaam

Single responsibility principle

O clasă ar trebui să aibă unul și un singur motiv de schimbare.

Acest principiu se bazează pe faptul că o clasă sau un modul trebuie să fie preocupat doar de un aspect al unui program sau să fie responsabil pentru un lucru. Cerințele se schimbă și software-ul evoluează tot timpul. Când se întâmplă acest lucru, clasele, modulele și funcțiile trebuie să reflecte această schimbare. Cu cât este mai îngrijorată o clasă sau mai multe responsabilități, cu atât ea trebuie schimbată. Schimbarea unor astfel de clase poate fi consumatoare de timp și dificilă, ducând adesea la efecte secundare.

Open-Closed Principle

Ar trebui să poți extinde un comportament al claselor, fără a-l modifica.

Al doilea principiu SOLID este Priniciul cu Închidere deschisă. Aceasta înseamnă că clasele, modulele și metodele ar trebui să fie deschise pentru extensie, dar închise pentru modificare. Facem acest lucru creând abstractizări, în limbi precum kotlin putem folosi interfețe, această abstractizare ar trebui apoi injectată acolo unde este nevoie. Scopul acestui lucru este de a conduce un design modular.

Liskov Substitution Principle

Dacă S este un subtip de T, atunci obiectele de tip T dintr-un program pot fi înlocuite cu obiecte de tip S fără a modifica niciuna dintre proprietățile dezirabile ale programului respectiv.

În esență, ar trebui să putem înlocui orice instanță a unei clase de părinți cu cea a unei clase de copii, fără să se rupă nimic. Atunci când începeți cu o programare orientată pe obiecte,

referirea la expresia „este o” pentru a ajuta la determinarea relațiilor dintre clase poate fi utilă. Acest principiu SOLID ne ajută să ne asigurăm că facem acest lucru în mod corespunzător.

Interface Segregation Principle

niciun client nu trebuie obligat să depindă de metodele pe care nu le folosește

Principiul de segregare a interfeței sună destul de bine? Încălcările ISP se pot strecura în sistemul dvs. de-a lungul timpului, pe măsură ce funcțiile sunt adăugate și cerințele se schimbă. Codurile care încalcă acest principiu tind să fie puțin dificile de modificat, deoarece se pot produce multe efecte secundare din cauza interfețelor și claselor mai mari care le pun în aplicare. Ceea ce ne propunem este câteva interfețe mai mici, pentru sarcini specifice, mai degrabă decât cele mai mari și mai generice.

Dependency Inversion Principle

Depinde de abstractizări, nu de concreții.

Modulele la nivel înalt nu ar trebui să depindă de modulele de nivel scăzut. Ambele ar trebui să depindă de abstractizări (de ex. Interfețe).

Abstracțiile nu ar trebui să depindă de detalii. Detaliile (implementări concrete) ar trebui să depindă de abstractizări.

Curs 5

Metoda 1 – Polling

-interactiunea este guvernata de o buclă infinită

Loop forever:

```
{ i=1..n  
  read input i  
  answer to input i  
  inc i }
```

Metoda 2 – Interrupt- Driven

1. Activează dispozitivul, apoi
2. Începe procesarea de bază (instalează sistemul de gestiune a evenimentelor/întreruperii)
3. Așteaptă apariția unei întreruperi
4. La apariția unei întreruperi
 1. Salvează starea curentă (schimbare context)
 2. Încarcă și execută metoda de tratare a întreruperii
 3. Restaurează contextul anterior
 4. Go to #1

Metoda 3 – Event-driven

-interacțiunea este din nou guvernata de o buclă

```
main()
{
    ...initializează structurile de date ale aplicației ...
    ...initializează și lansează în execuție GUI....
    // intră în bucla de eveniment
    while(true)
    {
        Event e = get_event(); // primește evenimentul
        process_event(e); // tratează evenimentul
    }
}
```

Avantajele procesarii orientate eveniment

- Sunt mai portabile
- Permit tratarea mai rapidă
- Se pot folosi în time-slicing
- Încurajează reutilizarea codului
- Se potrivesc cu OOP

Componentele unui program simplu bazat pe EDP

- Generatoare de evenimente
- Sursa evenimentelor
- Bucla de evenimente
- Event mapper
- Înregistrarea evenimentelor

Pink Corelation

- Procesul de selecție a unei ferestre sau aplicații care trebuie să trateze un eveniment oarecare (deoarece le aparține) se numește corelația de selecție (pick correlation)

Ce sunt widgets?

- Sunt obiectele din cadrul unui GUI orientat obiect

Lab5- tkinter

Controlul versiunilor este un sistem ce înregistrează schimbările unui fișier sau a unui set de fișiere de-a lungul timpului, pentru a putea reveni la versiuni specifice mai târziu. Cu alte cuvinte, în cazul în care se dorește „Undo” la un anumit fișier, se poate reveni la o versiune anterioară (cu precizarea că trebuie făcut commit, ideal după orice funcție/funcționalitate scrisă).

Sisteme de versionare locale

Problema: Metoda de backup la fișierul/fișierele la care se lucra utilizată frecvent era o simplă arhivă (optional cu data curentă). Această metodă e predispusă la erori, deoarece este ușor să uiți în ce folder te afli și poți modifica alt set de fișiere.

Soluția: Sisteme de control al versiunilor cu o bază de date simplă ce „reține” toate modificările pe setul de fișiere.

Sisteme de versionare centralizate

Problema: Colaborarea cu alți colegi/dezvoltatori.

Soluția: Sisteme de control al versiunilor centralizate. Acestea au un singur server ce conține toate fișierele versionate și un număr de clienți care copiază versiunea curentă a fișierelor respective.

Sisteme de versionare distribuite

Problema: Sistemele de versionare centralizate reprezintă un „single point of failure”. Cu alte cuvinte, dacă server-ul devine offline pentru o anumită perioadă, în acel interval nu se poate colabora sau salva schimbări de versiune. De asemenea, dacă hard disk-ul pe care se află repository-ul central devine corrupt, se pierde tot istoricul proiectului. Acest lucru este valabil și la sistemele de versionare locale.

Soluția: Sistemele de versionare distribuite (Git, Mercurial, etc). Într-un astfel de sistem, clientii nu salveză doar ultima versiune a fișierelor, ci copiază întregul repository (inclusiv istoricul complet). Așadar, dacă vreun server „moare”, orice repository al unui client poate fi copiat înapoi pe acel server pentru a-l restaura.

Git

În cele ce urmează, se va utiliza Git ca sistem de versionare distribuit. Fișierele dintr-un repository git pot avea una dintre următoarele trei stări:

Modified - s-au efectuat modificări asupra fișierului dar nu s-a făcut încă commit în baza de date

Staged - s-a marcat un fișier modificat în versiunea lui curentă pentru a fi inclus în următorul commit

Committed - datele sunt stocate în siguranță în baza de date locală

Cozi de mesaje

O coadă de mesaje este utilizată pentru comunicarea între procese, sau între firele de execuție (thread-urile) același proces. Acestea oferă un protocol de comunicare asincron în care emițătorul și receptorul nu au nevoie să interacționeze în același timp (mesajele sunt reținute în coadă până când destinatarul le citește)

Avantajele utilizării cozilor de mesaje:

1. redundanță - procesele trebuie să confirme citirea mesajului și faptul că acesta poate fi eliminat din coadă

2. vârfuri de trafic (traffic spikes) - adăugarea în coadă previne aceste spike-uri, asigurând stocarea datelor în coadă și procesarea lor (chiar dacă va dura mai mult)

3. mesaje asincrone

4. îmbunătățirea scalabilității

5. garantarea faptului că tranzacția se execută o dată

6. monitorizarea elementelor din coadă

SQLite3

sqlitebrowser este un GUI pentru vizualizarea unei baze de date sqlite. Acesta poate fi pornit executând în terminal comanda: sqlitebrowser

Se observă utilizarea repetată a unui bloc with. Acesta asigură închiderea automată a conexiunii deschise și este recomandat să fie folosit atunci când este posibil. De asemenea, se remarcă faptul că în locul unei clase, s-a utilizat o tuplă cu nume pentru Book (deci un obiect imutabil după inițializare). Funcțiile au fost grupate într-o clasă DatabaseManager care conține comenzi SQL în variabile (pentru simplitate în cazul în care se dorește modificarea acestora și lizibilitate).

Se observă utilizarea repetată a unui bloc with. Acesta asigură închiderea automată a conexiunii deschise și este recomandat să fie folosit atunci când este posibil. De asemenea, se remarcă faptul că în locul unei clase, s-a utilizat o tuplă cu nume pentru Book (deci un obiect imutabil după inițializare). Funcțiile au fost grupate într-o clasă DatabaseManager care conține comenzi SQL în variabile (pentru simplitate în cazul în care se dorește modificarea acestora și lizibilitate)

Exemplul 1: Cozi de mesaje în python

Această aplicație creează două procese. Primul proces va trimite un mesaj celui de-al doilea și își va încheia execuția. Cel de-al doilea va citi mesajul din coadă și îl va afișa, după care își încheie execuția. Acest lucru este posibil folosind o coadă de mesaje din modulul multiprocessing. Se poate testa rezultatul dacă se înlocuiește importul Queue din multiprocessing cu: **from queue import Queue**

Exemplul 2: Cozi de mesaje în python - arhitectură client-server

Arhitectura Client-Server permite crearea și înregistrarea unei cozi de mesaje pe server (metoda register('get_queue', callable=lambda: queue)), care va fi utilizată în continuare de clientii care se conectează la server-ul creat. Se remarcă definirea și inițializarea unui proces Worker care adaugă un mesaj în coadă.

Observație: Apelând metoda server.serve_forever() server-ul va rămâne pornit, până la închiderea acestuia prin comanda **CTRL+C**.

Clientul 0 se va conecta la server-ul definit anterior și va adăuga un mesaj în coadă.

Clientul 1 se va conecta la server-ul definit anterior și va afișa întreaga coadă de mesaje.

Exemplul 3: Intercomunicare între procese C și procese Python prin cozi de mesaje System V

Executabilul generat în urma compilării fișierului sender.c va crea coada de mesaje și va scrie în ea un mesaj citit de la tastatură. Este important ca sender-ul să fie executat primul. În caz contrar, receiver-ul din C va aștepta la infinit, iar scriptul în python va afișa un mesaj cu coada de mesaje neinițializată.

receiver.c

Se observă faptul că diferența majoră față de programul anterior e înlocuirea apelului funcției msgsnd (message send) cu msgrcv (message receive).

Curs 6

Programare functională în Python

Programare functională reprezintă o metodă de proiectare și dezvoltare a aplicațiilor bazată pe evaluarea de expresii care nu au în vedere modificarea stării. În cadrul acestui stil de programare codul este oferit prin intermediul funcțiilor. Programarea functională promovează un cod fără efecte secundare, fără modificări ale stării variabilelor.

Programare imperativa

Calcule descrise prin instrucțiuni care modifică starea programului. Orientat pe acțiuni și efectele sale

Programare procedurală

Programul este format din mai multe proceduri (funcții, subroutines)

Programare orientată obiect

Un obiect este o reprezentare a unei entități din lumea reală asupra căruia se poate întreprinde o acțiune sau care poate întreprinde o acțiune

Pass –

este o operație nulă - atunci când este executată, nu se întâmplă nimic. Este util ca un marcator de loc atunci când o instrucțiune este necesară sintactic, dar nu trebuie executat niciun cod

Extragere subsiruri

H	e	l	l	o		B	o	b
0	1	2	3	4	5	6	7	8

```
>>> greet[0:3]→'Hel' | >>> greet[5:9]→' Bob' | >>> greet[:5]→'Hello' |
>>> greet[5:]→' Bob' | >>> greet[:]→'Hello Bob'
```

Concatenare (+) Repetitie (-)

```
>>> "spam" + "eggs"→'spameggs' | >>> "Spam" + "And"+ "Eggs"→'SpamAndEggs' |
>>> 3 * "spam"→'spamspamspam' | >>> "spam" * 5→'spamspamspamspamspam' |
>>> (3 * "spam") + ("eggs" * 5)→'spamspamspameggseggseggseggseggs'
>>> len("spam") →4
>>> for ch in "Spam!":
    print (ch, end=" ")
s.capitalize() – creeaza o copie a lui s cu prima litera facuta mare
s.title()- creeaza o copie a lui s cu toate literele facute mari
s.center(width)- centreaza pe s intr-o zona de o latime data
s.count(sub)-numara aparitiile lui sub in s
s.find(sub)- cauta prima aparitie a lui sub in s
s.join(list)-concateneaza o lista de siruri avand s ca separator
s.ljust(width)-ca si center dar in stanga
s.lower()- creeaza o copie a lui s cu toate litere facute mici
s.lstrip() – creeaza o copie a lui s cu toate spatiile albe eliminate
s.replace(oldsub,newsub)-inlocuieste aparitiile in s a lui oldsub cu newsub
s.rfind(sub) – cauta prima aparitie a lui sub in s si intoarce cea mai in dreapta aparitie
s.rjust(width) – are efect ca si center, dar s este identat la dreapta(right-justified)
s.rstrip() – sterge spatiile de la inceput si de la sfarsit
s.split() – extrage o lista de substringuri functie de un separator implicit ‘ ’ sau explicit oarecare
s.upper- creeaza o copie a lui s cu toate literele facute mari
```

Operatii pe lista

Metoda	Intelesul acesteia
<list>.append(x)	Adaugă x la sfârșitul listei
<list>.sort()	Sortează lista (o funcție de sortare poate fi trimisă ca parametru)
<list>.reverse()	Inversează lista
<list>.index(x)	Indexul pentru prima apariție a lui x în listă
<list>.insert(i, x)	Iinserează x pe poziția i
<list>.count(x)	Numărul de aparitii al lui x în listă
<list>.remove(x)	Sterge prima apariție a lui x în listă
<list>.pop(i)	Sterge element i și îl întoarce valoarea

Atributele unui obiect fisier

- **Atributele unui obiect fisier:**

După ce s-a deschis un fișier se obține o referință (obiect fișier) care are mai multe atribută.

• Atribut	Descriere
-----------	-----------

file.closed returnează true dacă fișierul este închis, false altfel.

file.mode returnează modul de acces în care a fost deschis fișierul.

file.name returnează numele fișierului.

file.softspace returnează false dacă este necesar spațiu explicit pentru afișare (print), true altfel

Laboratorul 6: Utilizarea POO în Python

Introducere

După cum am discutat limbajul Python suportă patru tipuri de paradigme de programare.

Dintre ele cele mai importante sunt cea orientată obiect și cea funcțională.

Deoarece aspectele specifice programării funcționale vor fi tratate în viitor s-a ales pentru acest laborator prezentarea abilităților limbajului cu privire la programarea orientată obiect.

Vom începe prin a reaminti o serie de noțiuni minimale după cum urmează

Tipurile de variabile sunt inițializate automat ca la Kotlin cu care se aseamănă parțial

Python folosește spațierea (tab-ul nu mai este recomandat dar pycharm îl suportă) pe post de demarcare bloc de instrucțiuni asociat

Toate clasele din Python sunt derivate din clasa mamă object. Dacă nu vom specifica altă clasă de bază clasa nou creată va fi derivată din object

- self este echivalentul lui this
- variabilă simplă se citește cu val=input("dati valoarea")
- variabilă se afișează cu print("\n val="+str(val))
- Conversiile de tip sunt realizate cu metode ajutătoare
- Comentariile încep cu #

Scheletul unei funcții main():

```
def main():
    pass
    # functia main() nu este obligatorie in pycharm

if __name__ == "__main__":
    main()
```

pentru a forța clasele care o implementează (în Python se va folosi moștenirea) să definească funcțiile respective.

Tratarea parametrilor din linia de comandă se bazează pe funcții din biblioteca sys, fiind limbaj interpretat primul parametru se află în sys.argv[1]

Limbajul Python determină în mod dinamic tipul variabilei la asignare. Deci specificarea tipului nu influențează acest aspect.

Spre exemplu:

example: int = 'String'

Deși tipul specificat este int, tipul variabilei example va fi String. Cu alte cuvinte, acestea au rolul de a ușura înțelegerea codului prin specificarea efectivă a tipului de date așteptat.

Se poate specifica și tipul returnat de o funcție:

```
def greeting(name: str) -> str:  
    return 'Hello ' + name
```

Se poate crea și un alias:

```
from typing import List  
Vector = List[float]  
  
def scale(scalar: float, vector: Vector) -> Vector:  
    return [scalar * num for num in vector]  
  
# typechecks; a list of floats qualifies as a Vector.  
new_vector = scale(2.0, [1.0, -4.2, 5.4])
```

De asemenea, se poate crea și un tip nou (pe baza unui tip existent):

```
from typing import NewType  
  
UserId = NewType('UserId', int)  
some_id = UserId(524313)
```

Specifierii de acces sunt introdusi practic prin convenția de notare:

- Pentru variabile și funcții publice, convenția **snake_case** (fără vreun underscore ca prefix)
- Pentru variabile și funcții protejate, convenția **_snake_case** (un underscore ca prefix)
- Pentru variabile și funcții private, convenția **__snake_case** (două underscore-uri ca prefix)

Convenții de denumire

- Pentru nume de clase: PascalCase
- Pentru nume de variabile / metode: snake_case
- Pentru constante: NUME_CONSTANTA

Curs 7

1. Colecții

Kotlin face distincție între colecțiile **mutabile** și **cele imutabile**. O **colecție mutabilă** poate fi actualizată pe loc prin adăugarea, ștergerea sau înlocuirea unui element. O **colecție imutabilă**, deși oferă aceleași operații (adăugare, ștergere, înlocuire) prin funcțiile operator, va crea o nouă colecție, lăsând-o pe cea veche neschimbată. Toate colecțiile se regăsesc în namespace-ul *kotlin.collections*.

Iterable vs Sequence vs Java Stream

Întrucât singura diferență care se observă la prima vedere între **Iterable** și **Sequence** este denumirea interfeței, în cele ce urmează se vor evidenția diferențele și cazurile de utilizare

O **secvență** este un tip iterabil 1 pe care se pot efectua operații fără crearea de colecții intermediare inutile, prin executarea tuturor operațiilor aplicabile pe fiecare element înainte de trecerea la următorul.

Secvențele sunt leneșe (lazy), funcțiile intermediare corespunzătoare pentru procesarea secvențelor nu fac calcule, ci returnează o nouă secvență ce o decorează pe cea anterioară cu o nouă operație. Toate calculele sunt evaluate în timpul operației terminale (cum ar fi `toList` sau `count`).

Procesarea secvențelor este în general mai rapidă decât procesarea directă a colecțiilor unde există mai mult de un pas de procesare.

ATENȚIE: Există cazuri în care secvențele nu sunt recomandate. Spre exemplu, în cazul sortării prin apelul funcției `sorted`, întrucât este necesară parcurgerea întregii colecții.

O soluție posibilă pentru sortare cu secvențe este utilizarea funcției `sortedBy`

În ceea ce privește stream-urile din Java, acestea sunt tot „leneșe”, fiind colectate în pasul final de procesare. De asemenea, stream-urile Java sunt mult mai eficiente pentru procesarea colecțiilor decât funcțiile de procesare corespunzătoare din Kotlin

Diferențe între stream-urile Java și secvențele Kotlin:

- secvențele Kotlin au mult mai multe funcții de procesare (fiind definite ca funcții extensie)
- Stream-urile Java pot fi pornite în mod paralel, utilizând o funcție `parallel`

Se recomandă utilizarea stream-urilor Java doar pentru procesări computaționale grele, unde se poate beneficia de modul paralel.

2. Generice

Programarea generică (sau genericele) este o tehnică prin care funcțiile pot fi scrise cu tipuri care nu sunt specificate în momentul scrierii și sunt folosite mai târziu cu diferite tipuri de date.

După cum se observă în funcția `random` avem un singur parametru tip (`T`) care este folosit pentru toți cei trei parametri și pentru tipul returnat. Se remarcă faptul că funcțiile generice au un parametru tip introdus prin paranteze unghiulare `<T>`.

În funcția `put` au fost inclusi mai mulți parametri `tip <K,V>`. Corpul celor două funcții trebuie implementat, apelul funcției `TODO()` generând o eroare de tipul `NotImplemented`.

Kotlin pot avea parametri tip

Pentru a crea o instanță a unei asemenea clase, trebuie precizat tipul argumentelor.

2.1. Polimorfism mărginit

Funcțiile care sunt generice pentru **orice** tip sunt utile, dar cumva limitate. Adesea, va fi nevoie de scrierea unor funcții care sunt generice pentru **unele** tipuri care au o caracteristică comună. Spre exemplu, definirea unei funcții care returnează minimul dintre două valori, pentru oricare valori ce suportă noțiunea de comparare.

Pentru a forța valorile să aibă acea noțiune de comparare, trebuie restricționate tipurile generice la cele care suportă funcțiile care trebuie invocate. Cu alte cuvinte, mărginim funcția polimorfică (generică), acest lucru fiind numit **polimorfism mărginit**.

2.1.1. Mărginiri superioare

Kotlin suportă un tip de mărginire a polimorfismului, mai precis mărginirea superioară. Din denumire, se remarcă că tipurile generice sunt restricționate la cele care sunt subclase ale mărginirii. Mărginirea se declară împreună cu parametrul tip

Comparable este un tip din biblioteca standard care definește metoda compareTo ce returnează o valoare mai mică decât 0 dacă primul element este mai mic, mai mare decât 0 dacă al doilea element este mai mic, egală cu 0 dacă elementele sunt egale.

Observație: Dacă nu se specifică o mărginire superioară pentru parametrul tip, compilatorul va folosi tipul Any ca mărginire superioară implicită.

2.1.2. Mărginiri multiple

Uneori, este necesară declararea de mărginiri superioare multiple. Spre exemplu, dacă se dorește extinderea funcției *min()* pentru a funcționa pe valori care sunt de asemenea serializabile, se mută declararea mărginirii superioare într-o clauză *where* separate

Observație: Toate mărginirile superioare sunt scrise ca și clauze *where* și formează o uniune de mărginire superioară.

Clasele pot defini de asemenea mărginiri superioare multiple

2.2. Varianță (Variance)

Una dintre cele mai complicate părți ale sistemului de tipuri Java sunt tipurile **wildcard**.

Kotlin nu are asemenea tipuri, dar oferă:

- varianță la momentul declarării (declaration-site variance)
- proiecțiile de tip
- proiecțiile stea

2.2.1. Varianță la momentul declarării

Regulă: Când un parametru tip T este declarat ca out al unei clase, atunci acest tip T poate fi folosit doar ca tip de return în membrii clasei respective.

Modifierul **out** se numește **adnotare de varianță** (variance annotation) și fiind vorba despre declararea parametrului tip, se numește varianță la momentul declarării. Despre interfața Source se spune că e **covariantă** în parametrul tip T.

Pe lângă adnotarea de varianță (**out**), Kotlin oferă și o **adnotare complementară de varianță, in**. Această adnotare face un parametru tip să fie **contravariant**: acel tip poate fi doar consumat (parametru de intrare), nu și produs (tip returnat).

Un exemplu de contravarianță este *interfața Comparable*

2.2.2. Proiecțiile de tip: varianță la momentul utilizării (use-site variance)

Unele clase nu pot fi constrânse să returneze un singur parametru tip T. Spre exemplu, clasa *Array*

Observație: clasa *Array* nu poate fi nici covariantă nici contravariantă

În funcția *copy* de mai sus, tipurile elementelor din *Array*-ul *from* sunt declarate cu *out* pentru a interzice modificarea acestora. Acesta este un exemplu de varianță la momentul utilizării. Cu alte cuvinte *array*-ul *from* este un array restricționat.

2.2.3. Proiecțiile stea

Uneori este necesară utilizarea unui argument tip necunoscut. Pentru a rezolva această problemă printr-o *modalitate sigură*, Kotlin a introdus aşa *numitele proiecții stea*. Trebuie definită o proiecție a unui tip generic pentru care fiecare instanțiere concretă a aceluui tip generic să fie un subtip al proiecției.

Sintaxa Kotlin pentru proiecțiile stea:

- **Foo<out T: TUpper>** unde T este un parametru tip **covariant** cu mărginirea superioară (upper bound) TUpper, Foo<*> este echivalent cu Foo<out TUpper>. Cu alte cuvinte, când T este necunoscut, se poate citi în mod *sigur* valori TUpper din Foo<*>
 - **Foo<in T>**, unde T este un parametru tip **contravariant**, Foo<*> este echivalent cu Foo<in Nothing>. Cu alte cuvinte, nu se poate scrie nimic în Foo<*> într-un mod sigur, când T este necunoscut
 - **Foo<T: TUpper>** unde T este un parametru tip **invariant** cu mărginirea superioară TUpper, Foo<*> este echivalent cu Foo<out TUpper> pentru citirea valorilor și echivalent cu Foo<in Nothing> pentru scrierea valorilor

Dacă un tip generic are mai mulți parametri tip, fiecare poate fi proiectat independent. Spre exemplu, pentru:

```
interface Function<in T, out U>
```

se pot defini urmatoarele proiecții stea:

- **Function<*, String>** echivalent cu **Function<in Nothing, String>**
- **Function<Int, *>** echivalent cu **Function<Int, out Any?>**
- **Function<*, *>** echivalent cu **Function<in Nothing, out Any?>**

Curs 8

Modelul fabrica de obiecte (factory)

Modelul din fabrică oferă o modalitate de a delega logica instantanării în clase de copii. Modelul de proiectare din fabrică este utilizat atunci când avem o *super-clasă* cu mai multe *sub-clase* și bazată pe intrare, trebuie să *returnăm una dintre sub-clase*. Acest model preia responsabilitatea instantaniei unei clase de la programul client la clasa din fabrică.

Super clasa în modelul din fabrică poate fi o interfață sau o clasă Java normală. În Python este o clasă abstractă în majoritatea cazurilor.

Beneficii:

- Modelul din fabrică oferă abordarea codului pentru interfață, mai degrabă decât implementarea.
- Modelul din fabrică elimină instantaneele claselor de implementare reale din codul clientului, ceea ce îl face mai robust, mai puțin cuplat și ușor de extins.

- Modelul din fabrică asigură abstractizarea între implementarea și clasele client prin moștenire.

Rezumat Modelul fabricii este o fabrică de fabrici. Acesta grupează întreprinderile individuale, dar asociate / dependente, fără a specifica clasele lor concrete.

Modelul fabrica abstracta (abstract factory)

O fabrică abstractă este o fabrică care returnează fabricile, nu vă faceți griji cu privire la pun. De ce este util acest strat de abstractizare? O fabrică normală poate fi utilizată pentru a crea seturi de obiecte conexe. O fabrică abstractă returnează fabricile. Astfel, o fabrică abstractă este folosită pentru a returna fabrici care pot fi utilizate pentru a crea seturi de obiecte conexe.

Modelul abstract de proiectare a fabricii este aproape similar cu [modelul fabricii](#), cu excepția faptului că seamănă mai mult cu fabrica de fabrici. Dacă sunteți familiarizați cu modelul de proiectare a fabricii, veți observa că avem o singură *clasă Factory* care returnează diferențele sub-clase bazate pe intrarea furnizată și utilizările *if-else*sau *switch*declarația clasei de fabricație pentru a realiza acest lucru. În modelul Abstract Factory, scăpăm de *if-else*bloc și avem o clasă de fabrică pentru fiecare sub-clasă și apoi o clasă Abstract Factory care va returna sub-clasa bazată pe clasa de fabrică de intrare.

Beneficii:

- Modelul Abstract Factory oferă o abordare a codului pentru interfață și nu a implementării.
- Modelul de fabricație abstract este *fabrica de fabrici* (sau super-fabrică) și poate fi extins cu ușurință pentru a găzdui mai multe produse, de exemplu, putem adăuga o altă sub-clasă *Celerio*și o fabrică *MarutiFactory*.
- Rezumatul Modelului din fabrică este robust și evită logica condiționată a modelului Fabrică.

Utilizarea modelului abstract din fabrică :

- Când sistemul trebuie să fie independent de modul în care obiectul său este creat, compus și reprezentat.
- Atunci când familia de obiecte conexe trebuie folosită împreună, atunci această constrângere trebuie aplicată.
- Când doriți să furnizați o bibliotecă de obiecte care nu prezintă implementări și dezvăluie doar interfețe.
- Când sistemul trebuie configurat cu una din mai multe familii de obiecte.

Modelul burlacului (singleton)

Modelul Singleton asigură că un singur obiect al unei anumite clase este creat vreodată. Uneori este important ca unele clase să aibă exact o singură instanță. Există multe obiecte de care avem nevoie doar de o singură instanță a acestora și dacă noi, instantaneu mai mult de unul, ne vom confrunta cu tot felul de probleme, cum ar fi comportamentul incorect al programului, utilizarea excesivă a resurselor sau rezultatele inconsecvențe.

De obicei, singletonii sunt folosiți pentru gestionarea centralizată a resurselor interne sau externe și oferă un punct de acces global pentru ei însăși.

Există doar două puncte în definirea unui model de design singleton:

- Trebuie să existe o singură instanță permisă pentru o clasă.
- Ar trebui să permitem accesul global la acea singură instanță.

Utilizarea modelului de design Singleton :

- Modelul Singleton este folosit mai ales pentru economisirea memoriei, deoarece obiectul nu este creat la fiecare solicitare. Doar o singură instanță este reutilizată din nou.

Modelul constructor (builder)

Modelul Builder vă permite să creați diferite arome ale unui obiect evitând în același timp poluarea constructorilor. Este util când pot exista mai multe arome ale unui obiect. Modelul Builder construiește un obiect complex folosind obiecte simple și utilizând o abordare pas cu pas.

Este folosit mai ales atunci când obiectul nu poate fi creat într-o singură etapă ca în dez-serializarea unui obiect complex.

Intenția modelului Builder este de a separa construcția unui obiect complex de reprezentarea sa, astfel încât același proces de construcție poate crea reprezentări diferite. Acest tip de separare reduce dimensiunea obiectului. Designul se dovedește a fi mai modular cu fiecare implementare conținută într-un obiect constructor diferit. Adăugarea unei noi implementări (adică adăugarea unui constructor nou) devine mai ușoară. Procesul de construcție a obiectului devine independent de componentele care alcătuiesc obiectul. Aceasta oferă mai mult control asupra procesului de construcție a obiectului.

Modelul Builder sugerează utilizarea unui obiect dedicat la care se face referire ca *director*, care este responsabil de invocarea diferitelor metode de constructor necesare pentru construcția obiectului final. Diferite obiecte client pot folosi obiectul *Director* pentru a crea obiectul necesar. Odată ce obiectul este construit, obiectul client poate solicita direct de la constructor obiectul complet construit. Pentru a facilita acest proces, o nouă metodă `getResult` poate fi declarată în interfața *Builder* comună pentru a fi implementată de diferiți constructori de beton.

Clasele și obiectele care participă la acest tipar sunt:

- **Builder** (VehicleBuilder)
 - specifică o interfață abstractă pentru crearea părților unui **Product** obiect
- **ConcreteBuilder** (MotorCycleBuilder, CarBuilder, ScooterBuilder)
 - construiește și asamblează părți ale produsului prin implementarea **Builder** interfeței
 - definește și ține evidența reprezentării pe care o creează
 - oferă o interfață pentru preluarea produsului
- **Director** (Magazin)
 - construiește un obiect folosind **Builder** interfața
- **Product** (Vehicul)
 - reprezintă obiectul complex în construcție. **ConcreteBuilder** construiește reprezentarea internă a produsului și definește procesul prin care este asamblat
 - include clase care definesc părțile componente, inclusiv interfețe pentru asamblarea pieselor în rezultatul final

Avantajul modelului de design al constructorului:

- Oferă o separare clară între construcția și reprezentarea unui obiect.
- Oferă un control mai bun asupra procesului de construcție.
- Sprijină pentru a schimba reprezentarea internă a obiectelor.

Modelul Builder este utilizat atunci când:

- Algoritmul de creare a unui obiect complex este independent de părțile care compun efectiv obiectul.
- Sistemul trebuie să permită reprezentări diferite pentru obiectele care sunt construite.

Modelul prototip (prototype)

Modelul prototip creează un obiect bazat pe un obiect existent prin clonare.

Modelul prototip este folosit atunci când crearea obiectului este o afacere costisitoare și necesită mult timp și resurse și aveți un obiect similar deja existent. Prin urmare, acest model oferă un mecanism pentru a copia obiectul original pe un obiect nou și apoi a-l modifica în funcție de nevoile noastre.

De exemplu, un obiect trebuie creat după o operațiune de bază de date costisitoare. Putem cache obiectul, returnăm clona acesteia la următoarea solicitare și actualizăm baza de date în funcție de când este nevoie, reducând astfel apelurile la baza de date.

Când să utilizați:

- când clasele care se vor instaura sunt specificate în timpul rulării, de exemplu, prin încărcare dinamică;
- evitarea construirii unei ierarhii de clasă a fabricilor care să fie paralelă cu ierarhia de clasă a produselor;
- când instanțele unei clase pot avea una dintre doar câteva combinații diferite de stare. Poate fi mai convenabil să instalezi un număr corespunzător de prototipuri și să le clonezi, decât să instantanezi clasa manual, de fiecare dată cu starea corespunzătoare.

Principalele avantaje ale modelului prototipului sunt următoarele:

- Aceasta reduce nevoia de sub-clasificare.
- Ascunde complexități ale creării de obiecte.
- Clientii pot obține obiecte noi fără să știe ce tip de obiect va fi.
- Vă permite să adăugați sau să eliminați obiecte în timp de execuție.

Vom crea o clasă abstractă **Shape** și clase concrete care să extindă **Shape** clasa. O clasă **ShapeCache** este definită ca următoarea etapă care stochează obiecte de formă într-un **Hashtable** și returnează clona lor atunci când este solicitată.

[PrototypPatternDemo](#), clasa noastră demo va folosi **ShapeCache** clasa pentru a obține un **Shape** obiect.

Modelul adaptor (adapter)

Modelul adaptorului este utilizat pentru conectarea a două interfețe incompatibile care altfel nu pot fi conectate direct. Un adaptor înfășoară o clasă existentă cu o interfață nouă, astfel încât să devină compatibilă cu interfața necesară.

Modelul pod (bridge)

Modelul Bridge este folosit pentru a decupla o abstractizare de la punerea în aplicare, astfel încât cele două să poată varia independent.

Aceasta înseamnă a crea o interfață bridge care folosește principiile OOP pentru a separa responsabilitățile în diferite clase abstracte.

Puncte cheie de diferențiere :

- Un model Bridge nu poate fi implementat decât înainte de proiectarea aplicației.
- Permite modificarea independentă a abstractizării și implementării, în timp ce un model adaptor face posibilă colaborarea claselor incompatibile

Modelul compus (composite)

Modelul compozit este folosit acolo unde trebuie să tratăm un grup de obiecte în mod similar ca un singur obiect. Modelul compus compune obiectele în termenii unei structuri de arbori pentru a reprezenta o parte și o întreagă ierarhie. Acest tip de model de design se încadrează în modelul structural, deoarece acest model creează o structură arbore a grupului de obiecte.

Acest model creează o clasă care conține un grup de obiecte proprii. Această clasă oferă modalități de a-și modifica grupul de aceleasi obiecte.

Modelul fatada(facade)

Modelul de fațadă ascunde complexitățile sistemului și oferă o interfață pentru client folosind care poate accesa sistemul. Acest tip de model de design se încadrează în structura structurală, deoarece acest model adaugă o interfață la sistemul existent pentru a-și ascunde complexitățile.

Acest model implică o singură clasă care furnizează metode simplificate solicitate de către client și delegă apeluri la metodele claselor de sistem existente.

Modelul lant de responsabilitati (chain of responsibility)

După cum sugerează și denumirea, modelul de lanț de responsabilitate creează un lanț de obiecte receptoare pentru o solicitare. Acest model decouplează expeditorul și receptorul unei cereri în funcție de tipul de solicitare. Acest model se înscrie sub tipare comportamentale.

În acest model, în mod normal, fiecare receptor conține trimitere la un alt receptor. Dacă un obiect nu poate gestiona cererea, atunci aceasta trece la următorul receptor și aşa mai departe.

Modelul observator

Modelul de observare este utilizat atunci când există o relație unu-la-multe între obiecte, cum ar fi dacă un obiect este modificat, obiectele sale depenedente trebuie notificate automat. Modelul observator se încadrează în categoria modelului comportamental.

Punerea în aplicare

Modelul de observator folosește trei clase de actori. Subiect, observator și client. Subiectul este un obiect care are metode de atașare și detașare a observatorilor de un obiect client. Am creat o clasă abstractă *Observer* și o clasă concretă *Subiect* care extinde clasa *Observer*.

ObserverPatternDemo, clasa noastră demo, va folosi obiectul clasei *subiect* și concret pentru a arăta modelul observatorului în acțiune.

Modelul automatului finit (state)

Scopul modelului: Permite unui obiect să își modifice comportamentul atunci când starea sa internă se schimbă. Obiectul va părea că își schimbă clasa.

Aplicabilitate: Modelul stare se utilizează în următoarele cazuri:

- Comportamentul unui obiect depinde de starea sa și trebuie să-și schimbe comportamentul în timpul execuției, în funcție de starea respectivă;
- Operațiile au declarații condiționale compuse, mari, care depind de starea obiectului.

Consecințe:

- Localizează un comportament specific stării și comportamentul partitilor pentru diferite stări;
- Face tranzițiile între stări explicite;
- Obiectele stării pot fi partajate.

În modelul de stat, un comportament de clasă se schimbă în funcție de starea sa. Acest tip de model de design se încadrează în modelul de comportament.

În modelul de stat, creăm obiecte care reprezintă diferite stări și un obiect de context al cărui comportament variază pe măsură ce obiectul său de stare se schimbă.

Punerea în aplicare

Vom crea o interfață de *stat* care definește o acțiune și clase concrete de stat care implementează interfața de *stat*. *Contextul* este o clasă care poartă un stat.

StatePatternDemo, clasa noastră demo, va folosi obiecte *Context* și *State* pentru a demonstra schimbarea comportamentului *Context* în funcție de tipul de stare în care se află.

Modelul vizitator

În modelul de vizitator, folosim o clasă de vizitator care schimbă algoritmul de execuție al unei clase de elemente. Astfel, algoritmul de execuție al elementului poate varia în funcție de momentul în care vizitatorul variază. Acest model intră în categoria modelului de comportament. Conform modelului, obiectul element trebuie să accepte obiectul vizitator, astfel încât obiectul vizitator să se ocupe de operația asupra obiectului element.

Punerea în aplicare

Vom crea o interfață *ComputerPart* care definește acceptarea operațiunii. *Tastatura*, *mouse-ul*, *monitorul* și *computerul* sunt clase concrete care implementează interfața *ComputerPart*. Vom defini o altă interfață *ComputerPartVisitor* care va defini operațiile clasei de vizitatori. *Computerul* folosește vizitatorul concret pentru a face acțiuni corespunzătoare.

VisitorPatternDemo, clasa noastră demo, se va folosi de *calculator* și *ComputerPartVisitor* clase pentru a demonstra utilizarea modelului de vizitator.

Modelul comanda (command)

Modelul de comandă este un model de proiectare bazat pe date și se încadrează în categoria modelului comportamental. O solicitare este înfășurată sub un obiect sub formă de comandă și transmisă obiectului invocator. Obiectul Invoker cauță obiectul adecvat care poate gestiona această comandă și trece comanda către obiectul corespunzător care execută comanda.

Punerea în aplicare

Am creat o *comandă* de interfață care acționează ca o comandă. Am creat o clasă de *stocuri* care acționează ca o solicitare. Avem clase de comandă

concrete *BuyStock* și *SellStock* care implementează interfața *Comenzilor*, care va face procesarea efectivă a comenzi. Un *broker* de clasă este creat ca obiect de invocator. Poate lăua și plasa comenzi.

Obiectul *Broker* folosește modelul de comandă pentru a identifica ce obiect va executa ce comandă se bazează pe tipul de comandă. *CommandPatternDemo*, clasa noastră demo, va folosi clasa *Broker* pentru a demonstra modelul de comandă.

Modelul restaurare/reamintire (lat. memento)

Modelul Memento este folosit pentru a restabili starea unui obiect la o stare anterioară. Modelul Memento se încadrează în categoria modelului comportamental.

Punerea în aplicare

Modelul Memento folosește trei clase de actori. Memento conține starea unui obiect de restaurat. Originator creează și stochează stări în obiectele Memento, iar obiectul Caretaker este responsabil de restaurarea stării obiectelor din Memento. Am creat clase *Memento*, *Originator* și *Caretaker*.

MementoPatternDemo, clasa noastră demo, va folosi obiecte *Caretaker* și *Originator* pentru a arăta restaurarea stărilor obiectelor.

Modelul iterator

Modelul Iterator este foarte des utilizat modelul de design în mediul de programare Java și .Net. Acest model este utilizat pentru a obține o modalitate de a accesa elementele unui obiect de colecție în mod secvențial, fără a fi necesară cunoașterea reprezentării sale de bază.

Modelul Iterator se încadrează în categoria modelului comportamental.

Punerea în aplicare

Vom crea o interfață *Iterator* care prezintă metoda de navigație și o interfață *Container* care retrimit iteratorul. Clasele de beton care implementează interfața *Container* vor fi responsabile pentru implementarea interfeței *Iterator* și utilizarea acestela.

IteratorPatternDemo, clasa noastră demo va folosi *NomiRepozitoriu*, o implementare a clasei concrete pentru a imprima *Nume* stocate ca o colecție în *NumeRepozitoriu*.

Modelul strategie (strategy)

În modelul Strategiei, un comportament de clasă sau algoritmul său poate fi modificat în timpul rulării. Acest tip de model de design se încadrează în modelul de comportament.

În modelul de strategie, creăm obiecte care reprezintă strategii diverse și un obiect de context al cărui comportament variază în funcție de obiectul său de strategie. Obiectul de strategie schimbă algoritmul de executare a obiectului de context.

Punerea în aplicare

Vom crea o interfață *Strategie* care definește o acțiune și clase de strategie concrete care implementează interfața *Strategie*. *Context* este o clasă care folosește o strategie.

Strategia PatternDemo, clasa noastră demo, va folosi *Context* și obiecte de strategie pentru a demonstra schimbarea comportamentului *Context* pe baza strategiei pe care o implementează sau o folosește.

Modelul mediator

Modelul de mediator este utilizat pentru a reduce complexitatea comunicării între mai multe obiecte sau clase. Acest model oferă o clasă de mediator care gestionează în mod normal toate comunicațiile dintre diferite clase și sprijină întreținerea ușoară a codului prin cuplaj liber. Modelul de mediator se încadrează în categoria modelului comportamental.

Punerea în aplicare

Dăm doavă de un model de mediator, de exemplu, o cameră de chat în care mai mulți utilizatori pot trimite un mesaj în camera de chat și este responsabilitatea camerei de chat să afișeze mesajele tuturor utilizatorilor. Am creat două clase *ChatRoom* și *User*. Obiectele *utilizatorului* vor folosi metoda *ChatRoom* pentru a partaja mesajele lor.

MediatorPatternDemo, clasa noastră de demonstrații, va folosi obiecte de *utilizator* pentru a afișa comunicarea între ei.

Modelul decorator

Modelul Decorator permite utilizatorului să adauge funcționalități noi la un obiect existent, fără a-i modifica structura. Acest tip de model de design se încadrează în modelul structural, deoarece acest model acționează ca un înveliș pentru clasa existentă.

Acest model creează o clasă decorator care înfășoară clasa originală și oferă funcționalitate suplimentară, păstrând intactă semnătura metodelor clasei.

Vom demonstra utilizarea modelului de decorator prin următorul exemplu în care vom decora o formă cu o anumită culoare fără o clasă de formă modificată.

Punerea în aplicare

Vom crea o interfață *Shape* și clase concrete care implementează interfața *Shape*. Vom crea apoi o clasă de decorator abstract *ShapeDecorator* care implementează interfața *Shape* și având obiectul *Shape* ca variabilă de instanță.

RedShapeDecorator este o clasă concretă care implementează *ShapeDecorator*.

DecoratorPatternDemo, clasa noastră demo va folosi *RedShapeDecorator* pentru a decora obiecte *Shape*.

Modelul proxy (intermediar)

În modelul proxy, o clasă reprezintă funcționalitatea altei clase. Acest tip de model de design se încadrează în structura structurală.

În modelul proxy, creăm un obiect având obiect original pentru a-i interfața funcționalitatea cu lumea exterioară.

Punerea în aplicare

Vom crea o interfață *Image* și clase concrete care implementează interfața *Image*. *ProxyImage* este o clasă proxy pentru a reduce amprenta de memorie a încărcării obiectului *RealImage*.

ProxyPatternDemo, clasa noastră de demonstrații, va folosi *ProxyImage* pentru a obține un obiect *Image* pe care să îl încarce și să îl afișeze aşa cum este nevoie.

Exemplul 5:Generator

Generatoarele reprezintă o modalitate simplă de a crea iteratori, toate cerințele anterioare fiind gestionate în mod automat de generator.

Definirea unui generator presupune utilizarea cuvântului cheie *yield* în loc de *return*.

Acesta trebuie să conțină cel puțin un *yield* (poate conține mai multe alte *yield*-uri, *return*-uri).

Diferența între **return** și **yield**:

- *return* - încheie complet execuția funcției
- *yield* - pune pauză funcției, salvând stările și continuând mai târziu de unde a rămas.

Avantajele utilizării unui generator în loc de iterator:

- ușor de implementat
- eficient din punct de vedere al memoriei
- permite reprezentarea unui **stream infinit**
- generatoarele pot fi folosite pentru a realiza un pipeline cu o serie de operații.

Ce este un pipeline ?

Un pipeline este un lant de elemente de procesare aranjate astfel încât ieșirea fiecărui element este intrarea următorului

Ce este o Coroutina ?

Corutinele sunt fire light-weight ceea ce inseamna ca durata de viata a noii coroutine este limitata doar de durata de viata a intregii aplicatii.

Ce este un Thread?

un fir de execuție este cea mai mică secvență de instrucțiuni programate care pot fi gestionate independent de un planificator, care este de obicei o parte a sistemului de operare. Implementarea thread-urilor și proceselor diferă de la sistemele de operare, dar în cele mai multe cazuri un thread este o componentă a unui proces. Mai multe fire pot exista în cadrul unui singur proces, executând simultan și partajând resurse, cum ar fi memoria, în timp ce diferite procese nu împărtășesc aceste resurse.

Care este diferența dintre Coroutine si thread-uri?

Corutinele sunt o forma de procesare sequentiala : doar una se executa la un moment dat

Firele de executie sunt (cel putin conceptual) o forma de procesare simultana: mai multe fire pot fi executate la un moment dat

Async

Face o coroutina

Deadlock

este o situatie cu care se confrunta sistemele de operare actuale pentru a face fata mai multor procese.

Canale (channel)

Valorile amânate oferă o modalitate convenabilă de a transfera o singură valoare între corutine. Canalele oferă o modalitate de a transfera un flux de valori.

Wait()

Acesta spune firului apelant să renunțe la blocare și să plece la somn până când un alt fir intră în același monitor și sună notificarea (). Metoda wait () eliberează blocarea înainte de aşteptare și atinge blocarea înainte de a reveni din metoda wait (). Metoda wait () este de fapt strâns integrată cu blocarea de sincronizare, folosind o caracteristică care nu este disponibilă direct din mecanismul de sincronizare.

Notify()

Se trezește un singur fir care a numit wait () pe același obiect. Trebuie menționat faptul că apelarea notify() nu renunță de fapt la o blocare la o resursă. Spune unui fir de aşteptare că acel fir se poate trezi. Cu toate acestea, blocarea nu este de fapt renunțată până la blocarea sincronizată a notificatorului.

Așadar, dacă un notificator apelează notify() la o resursă, dar notificatorul trebuie să efectueze încă 10 secunde de acțiuni asupra resursei din blocul său sincronizat, firul care așteptat va trebui să aștepte cel puțin încă 10 secunde suplimentare pentru notificator pentru a elibera blocarea pe obiect, chiar dacă notify() a fost apelată.

NotifyAll()

Se trezește toate firele numite wait () pe același obiect. Firul cu prioritate maximă va rula primul în cea mai mare parte a situației, deși nu este garantat. Alte lucruri sunt la fel ca metoda notify() de mai sus.

Laborator 10

Functii recursive

Kotlin suportă un stil de programare funcțională cunoscut ca recursivitatea coadă (tail recursion). Aceasta permite unor algoritmi care în mod normal ar fi fost scriși cu bucle să fie scris cu o funcție recursivă, dar fără riscul de „stack overflow”.

Spre deosebire de recursivitatea normală în care toate apelurile recursive sunt efectuate la început și apoi se calculează rezultatul din valorile returnate la urmă, în recursivitatea coadă calculele

sunt executate primele, apoi apelurile recursive (apelul recursiv trimite rezultatul pasului curent către următorul apel recursiv).

Când o funcție este marcată cu modificadorul tailrec și are forma corespunzătoare, compilatorul optimizează recursivitatea, rezultând o versiune bazată pe o buclă rapidă și eficientă în loc

Coroutine recursive

Pentru ca o funcție recursivă *tailrec* să poată fi utilizată într-o corutină, trebuie utilizat cuvântul cheie *suspend*

exemplu cu actori (discutat la curs) care reprezintă entitatea creată prin combinarea unei corutine, o stare care este izolată în interiorul acestei corutine și un canal de comunicație cu alte subroutines

lock() threading

Pentru a asigura accesul exclusiv la o secțiune de cod în Python se folosesc obiecte de tip Lock. Un lock se poate afla într-unul din două stări: blocat sau neblocat (este creat neblocat).

Când este neblocat și se apelează funcția *acquire()* se trece în stare blocat și apelul se întoarce imediat. Când este blocat și se apelează *acquire()*, apelul nu se întoarce decât atunci când alt thread îl deblochează. Deblocarea este făcută de funcția *release()* care are rolul de a trece un obiect de tip Lock din stare blocat în neblocat.

O funcție din fabric (factory function) care returnează un nou obiect de blocare primitiv. Odată ce un fir l-a achiziționat, încercările ulterioare de a-l achiziționa se blochează, până când este eliberat; orice fir îl poate elibera.

Rlock() threading

O funcție din fabrică care returnează un nou obiect de blocare reentrant (reintrant). Un blocant reentrant trebuie eliberat de firul care l-a achiziționat. Odată ce un fir a obținut un blocaj reentrant, același fir îl poate achiziționa din nou fără blocare; firul trebuie să-l elibereze o dată pentru fiecare dată când l-a dobândit.

Semafoare

Semafoarele sunt obiecte de sincronizare diferite de lock-uri prin faptul că salvează numarul de operații de deblocare efectuate asupra lor. Un semafor gestionează un contor intern care este decrementat de un apel *acquire()* și incrementat de apelul *release()*. Contorul nu poate ajunge la valori negative deci atunci când este apelată funcția *acquire()* și contorul este 0 threadul se

blocheaza pana cand alt thread apeleaza release(). Atunci cand este creat un semafor contorul are valoarea 1.

Fir cu conditie (conditii)

O variabila de tip condition este asociata cu un lock; acesta poate fi pasat (atunci cand mai multe variabile condition partajeaza un lock) sau poate fi creat implicit. Sunt prezente aici metodele acquire() si release() care le apeleaza pe cele corespunzatoare lock-ului si mai exista functiile wait(), notify() si notifyAll(), apelabile doar daca s-a reusit obtinerea lock-ului.

Metoda wait() elibereaza lock-ul si se blocheaza in asteptarea unei notificari ca urmare a unui apel notify(), care deblocheaza un singur thread care astepta si notifyAll(), care deblocheaza toate threadurile care asteptau conditia. De mentionat ca apelurile notify() si notifyAll() nu elibereaza lock-ul, deci un thread nu va fi trezit imediat ci doar cand cele doua apeluri de mai sus au terminat de folosit lock-ul si l-au eliberat.

Fir cu eveniment (event-uri)

Evenimentele reprezinta una din cele mai simple metode de comunicatie intre thread-uri: un thread semnalizeaza un eveniment, iar altul asteapta ca evenimentul sa se intampla. In Python, un obiect de tip event are un flag intern, initial setat pe false. Acesta poate fi setat pe true cu functia set() si resetat folosind clear(). Pentru a verifica starea flag-ului, se apeleaza functia isSet().

Un alt thread poate folosi metoda wait([timeout]) pentru a astepta ca un eveniment sa se intampla (ca flag-ul sa devina true): daca in momentul apelarii wait(), flag-ul este true, thread-ul apelant nu se blocheaza, dar daca este false se blocheaza pana la setarea evenimentului. De altfel, la un set(), toate threadurile care asteptau event-ul cu wait() vor fi trezite

Cozi de mesaje

Sunt folosite de procese pentru a comunica intre ele prin mesaje. Aceste mesaje isi pastreaza ordinea in interiorul cozii de mesaje. Sunt mecanisme de comunicare unidirectionale atat pe Linux, cat si pe Windows.

Memorie partajata

Acest mecanism permite comunicarea intre procese prin accesul direct si partajat la o zonă de memorie bine determinată. Este un mod mai rapid de comunicare intre procese decât celelalte mijloace IPC, dar are un mare dezavantaj, procesele ce comunică trebuie să fie pe aceeași mașină (spre deosebire de socketi, pipe-urile cu nume și cozile de mesaje din Windows)

Comunicare pipe

pipe (conducta). "Conducta" este o cale de legatura care poate fi stabilita intre doua procese inrudite (au un stramos comun sau sunt in relatia stramos-urmas). Ea are doua capete, unul prin care se pot scrie date si altul prin care datele pot fi citite, permitand o comunicare intr-o singura directie. In general, sistemul de operare permite conectarea a unuia sau mai multor procese la fiecare din capetele unui pipe, astfel incat, la un moment dat este posibil sa existe mai multe procese care scriu, respectiv mai multe procese care citesc din pipe. Se realizeaza, astfel, comunicarea unidirectionala intre procesele care scriu si procesele care citesc.

Bariere (barrier)

Obiectele de barieră din python sunt utilizate pentru a aștepta un număr fix de fir pentru a finaliza execuția înainte ca orice thread special să poată continua cu executarea programului. Fiecare fir apelează funcția wait () la atingerea barierei. Bariera este responsabilă pentru urmărirea numărului de apeluri de așteptare (). Dacă acest număr depășește numărul de fire pentru care bariera a fost inițializată, atunci bariera oferă o modalitate firelor de așteptare pentru a continua execuția. Toate firele din acest punct de execuție sunt lansate simultan.

Multiproces

multiprocesarea este un pachet care acceptă procesele de depunere folosind o API similară modulului de filetare. Pachetul de multiprocesare oferă concurență locală și de la distanță, trecând efectiv în lateral la Global Interpreter Lock prin utilizarea subproceselor în loc de fire. Datorită acestui fapt, modul de multiprocesare permite programatorului să utilizeze complet mai multe procesoare pe o anumită mașină.

Pool de procese

Modulul de multiprocesare. Un exemplu principal în acest sens este obiectul Pool care oferă un mijloc convenabil de paralelizare a execuției unei funcții pe mai multe valori de intrare, distribuind datele de intrare pe procese (paralelismul de date). Următorul exemplu demonstrează practica comună de a defini astfel de funcții într-un modul astfel încât procesele copil să poată importa cu succes acel modul

Cozi in multiprocessing

Returnează o coadă partajată a procesului implementată folosind o conductă și câteva blocaje / semafoare. Când un proces pune prima dată un element pe coadă, este început un fir de alimentare care transferă obiecte dintr-un tampon în conductă.

Task

Tasks/Sarcinile sunt utilizate pentru a rula coroutinee în bucle de eveniment. Dacă o corupție așteaptă un viitor, sarcina suspendă execuția corouinei și așteaptă finalizarea viitorului. Când se termină Viitorul, se reia execuția corouinei învelite.

Asyncio

asyncio este o bibliotecă pentru a scrie cod simultan folosind sintaxa `async / await`.

Event loops

Buclele de evenimente rulează sarcini și apeluri asincrone, execută operațiuni de rețea IO și execută subprocese

Asyncio si Future

Obiectele viitoare/future sunt utilizate pentru a pune în legătură codul bazat pe apeluri de nivel redus cu cod `async / așteptare` la nivel înalt

Functii lambda

Sunt functii care nu sunt declarate, dar sunt folosite imediat ca o expresie.

Expresiile lambda și funcțiile anonime sunt funcții care nu sunt declarate, dar sunt folosite imediat ca o expresie

Functia capitalize

Funcția *capitalize* este o **funcție lambda** a cărei tip este `(String) -> String` (syntactic sugar). Cu alte cuvinte, primește un String ca parametru și returnează un String. Acest tip de date este o scurtătură pentru `Function1<String, String>`, unde `Function1<P1, R>` este o interfață definită în biblioteca standard Kotlin și are o singura metodă, `invoke (P1) : R` marcată ca operator.

1.2 Funcții de ordin superior (higher-order function)

Funcția de ordin superior este o funcție care primește funcții ca parametri sau returnează o funcție. Funcțiile lambda pot fi utilizate ca parametri în alte funcții

ATENȚIE: parametrul `it` ar trebui folosit doar în cazul în care este clar ce tip de date se folosește. De asemenea, dacă se dorește ca o funcție clasică să fie trimisă ca parametru, se poate utiliza double colon (`::`)

Utilizând funcții de ordin superior, se pot aplica operații dacă sunt îndeplinite anumite condiții. Funcția `getAnotherFunction()` primește un număr întreg și returnează o funcție care primește un String și nu returnează nimic (doar afișează). În main, se apelează totodată și funcția returnată de `getAnotherFunction()`

1.3 Funcții pure și efecte secundare (side effects)

1.3.1 Efecte secundare

Într-un program, când o funcție modifică orice obiect/date din afara scopului propriu, acest lucru se numește **efect secundar**.

O funcție care modifică o proprietate statică sau globală, modifică un argument, generează o excepție, scrie output-ul la consolă / în fișier, sau apelează o altă funcție, **are un efect secundar**

1.3.2 Funcții pure

O funcție pură este o funcție a cărei rezultat returnat este complet dependent de parametrii acesteia. Pentru fiecare apel al unei funcții pure cu același parametru, aceasta va produce mereu același rezultat. De asemenea, o funcție pură NU trebuie să cauzeze efecte secundare, sau să apeleze alte funcții.

Număr variabil de argumente (Vararg)

Un parametru al unei funcții (în mod normal ultima) poate fi marcat cu un modificador `vararg`

Un singur parametru poate fi marcat ca vararg. Dacă un parametru vararg nu este ultimul din listă, valorile pentru următorii parametri pot fi transmise folosind sintaxa argumentului numit sau, dacă parametrul are un tip de funcție, trecând o lambda în afara parantezelor.

Alias

Alias-urile de tip furnizează nume alternative pentru tipurile existente. [...] Este util să scurtăm tipurile generice îndelungate

1.4 Funcții extensie

Funcțiile extensie permit modificarea tipurilor existente cu funcții noi. Sintaxa pentru adăugarea unei funcții extensie la un tip existent: *NumeClasă.FuncțieExtensie(listăParametri)*

Infix

Funcțiile marcate cu cuvântul cheie infix pot fi, de asemenea, apelate folosind notația infix (omiterea punctului și parantezelor pentru apel). Funcțiile Infix trebuie să satisfacă următoarele cerințe:

- Ele trebuie să fie funcții de membru sau funcții de extensie;
- Trebuie să aibă un singur parametru;
- Parametrul nu trebuie să accepte un număr variabil de argumente și nu trebuie să aibă o valoare implicită.

Functia map

Returnează o listă care conține rezultatele aplicării funcției de transformare date fiecărui element din tabloul original.

Functia filter

Returnează o listă care conține doar elemente care se potrivesc cu predicatul dat.

Functia FlatMap

Returnează o listă unică cu toate elementele obținute din rezultatele funcției de transformare invocate pe fiecare element al tabloului original.

Functia drop – taiere sumultimi

Returnează o subsecvență a acestei secvențe de caractere cu primele n caractere eliminate.

Functia take – extragere din submultimi

Returns a list containing first [n](#) elements.

Functia zip

Returnează o listă de perechi construite din elementele acestui tablou și celălalt tablou cu același index. Lista returnată are lungimea celei mai scurte colecții.

[1.6 Functori, funcții ca functori și delegați](#)

[1.6.1 Functori](#)

Un functor este un tip care definește o modalitate de a transforma (transform/map) conținutul lui.

Functor cu liste

Interfața List este una dintre cele mai importante abstractizări atunci când trebuie să gestionați colecțiile de date. Acesta este, de asemenea, un functor, deoarece are o funcție de hartă (map function) care are această semnătură

```
public inline fun <T, R> Iterable<T>.map(transform: (T)
```

[1.6.2 Delegați](#)

[1.6.2.1 Funcția Delegates.notNull și lateinit](#)

Delegatul Delegates.notNull permite continuarea programului fără inițializarea proprietății notNullStr. Dacă acea variabilă este utilizată înainte de a fi inițializată, va genera o excepție.

Funcția lateinit este o variantă mai simplă pentru a obține același comportament.

Observație: Delegates.notNull și lateinit funcționează numai pentru proprietăți declarate cu var.

[1.6.2.3 Funcția Delegates.Observable](#)

Funcția Delegates.observable are nevoie de doi parametri pentru a crea delegatul: valoarea inițială a proprietății și o funcție lambda care să fie executată de fiecare dată când se schimbă valoarea proprietății. Funcția lambda primește 3 parametri: o instanță de KProperty (o proprietate - val sau var) valoarea veche a proprietății valoarea nou asignată

[1.6.2.4 Funcția Delegates.vetoable](#)

Dreptul de veto permite o verificare logică la fiecare asignare a unei proprietăți, unde se poate decide dacă se continuă cu asignarea sau nu.

Curs 13

Keywords argument

Dacă aveți unele funcții cu mulți parametri și doriți să specificați doar unii dintre ei, atunci puteți da valori pentru astfel de parametri numindu-i - aceasta se numește argumente de cuvinte cheie (keywords argument)- folosim numele (cuvântul cheie) în loc de poziție (pe care noi au folosit tot timpul) pentru a specifica argumentele funcției.

Există două avantaje - unul, utilizarea funcției este mai ușoară, deoarece nu trebuie să ne facem griji cu privire la ordinea argumentelor. Doi, putem da valori numai acelor parametri pe care îi dorim, cu condiția ca ceilalți parametri să aibă valori ale argumentului implicit.

Generatoare recursive

Înainte de eliberarea Python 3.3, a trebuit să folosim bucle în cadrul unei funcții pentru a realiza recursivitatea. În versiunea 3.3, Python a permis utilizarea randamentului din declarație, facilitând utilizarea recursivului.
un exemplu :pentru a afișa numere impare folosind recursivitate în generatoarele Python.

Tee() –clonare iteratori

List comprehension

Returnează o listă bazată pe valorile existente.
-modalitate mai simplă decât FOR Loop

set comprehension

Returnează un set bazat pe valorile existente.

Dictionary comprehension

Returnează un dicționar bazat pe valorile existente.

ANY

Returnează adevărat dacă oricare dintre elemente este adevărat. Se returnează False dacă sunt goale sau toate sunt false. Orice poate fi gândit ca o secvență de operațiuni OR pe elementele furnizate.

Acesta scurtcircuită execuția, adică oprește execuția de îndată ce rezultatul este cunoscut.

Syntax : any(list of iterables)

ALL

Returnează adevărat dacă toate elementele sunt adevărate (sau dacă iterabilul este gol). Toate pot fi gândite ca o secvență de operațiuni AND pe valorile furnizate. De asemenea, scurtcircuitul execuției, adică oprește execuția imediat ce rezultatul este cunoscut.

Syntax : all(list of iterables)

Slice()

returnează un obiect de felie care poate fi folosit pentru a tăia siruri, liste, tuple etc.

enumerate()

Metoda enumerate () adaugă contor la un iterabil și îl returnează (obiectul enumerații).

Reversed()

returnează iteratorul inversat al secvenței date.

Decorator-stil macro

Decoratoarele macro vă permit să-ți sechestri codul prin crearea unui decorator care poate dubla funcționalitatea unui getter / setter în mai multe locuri!

Decorator cu parametri

Sintaxa pentru decoratori cu argumente este puțin diferită - decoratorul cu argumente ar trebui să returneze o funcție care va prelua o funcție și va returna o altă funcție. Deci ar trebui să întoarcă într-adevăr un decorator normal

Laboratorul 13: Paradigma calculului funcțional în Python
Laboratorul 12: Paradigma calculului funcțional în Kotlin
Laboratorul 11: Utilizarea bibliotecilor Python pentru paralelism
Laboratorul 10: Corutine - suport nativ pentru paralelism în Kotlin
Laboratorul 9: Design patterns în Python
Laboratorul 8: Design patterns în Kotlin
Laboratorul 7: Colecții și generice în Kotlin
Laboratorul 6: Utilizarea POO în Python
Laboratorul 4: Utilizarea principiilor SOLID în Kotlin

Laboratorul 13: Paradigma calculului funcțional în Python

-Introducere-

Limbajele de programare suportă descompunerea problemelor în mai multe moduri diferite:

Majoritatea limbajelor de programare sunt procedurale: programele reprezintă liste de instrucțiuni care spun computerului ce să facă cu intrarea programului. C, Pascal și chiar Unix sunt limbaje procedurale.

În limbajele declarative, descrieți problema ce necesită rezolvare iar implementarea arată cum să efectuați calculul în mod eficient. SQL este limbajul declarativ cel mai cunoscut; o interogare SQL descrie setul de date pe care doriți să îl recuperăți, iar motorul SQL decide dacă să scaneze tabele sau să utilizeze indexuri, care subclauzele trebuie efectuate mai întâi etc.

Programele orientate pe obiecte manipulează colecțiile de obiecte. Obiectele au stare internă și suportă metode care interogeză sau modifică într-un fel această stare internă. Smalltalk și Java sunt limbaje orientate pe obiecte. C++ și Python sunt limbaje care acceptă programarea orientată pe obiecte, dar nu forțează utilizarea funcțiilor orientate pe obiecte.

Programarea funcțională descompune o problemă într-un set de funcții. În mod ideal, funcțiile doar iau intrări și produc rezultate și nu au nicio stare internă care să afecteze rezultatul pentru o intrare dată. Limbaje funcționale cunoscute includ familia ML (Standard ML, OCaml și alte variante) și Haskell.

Într-un program funcțional, intrarea trece printr-un set de funcții. Fiecare funcție acționează pe intrarea sa și produce o ieșire. Stilul funcțional descurajează funcțiile cu efecte secundare care modifică starea internă sau fac alte modificări care nu sunt vizibile în valoarea returnată de funcție. Funcțiile care nu au efecte secundare sunt denumite pur funcționale. Evitarea efectelor secundare înseamnă a nu utiliza structuri de date care se actualizează pe măsură ce un program rulează; ieșirea fiecărei funcții trebuie să depindă numai de intrarea sa.

Există avantaje teoretice și practice ale stilului funcțional: probabilitate formală, modularitate, compozabilitate, ușurință de depanare și testare.

Probabilitate formală

Un avantaj teoretic este că e mai ușor să dovedești matematic că un program funcțional este corect.

Modularitate

Un beneficiu mai practic al programării funcționale este că te obligă să desparți problema în bucăți mai mici. Ca urmare, programele sunt mai modulare. Este mai ușor să specifici și să scrii o funcție mică care face un lucru decât o funcție mare care realizează o transformare complicată. Funcțiile mici sunt, de asemenea, mai ușor de citit și de verificat când vine vorba de erori.

Compozabilitatea

Pe măsură ce lucrăți la un program în stil funcțional, veți scrie o serie de funcții cu intrări și ieșiri diferite. Unele dintre aceste funcții vor fi inevitabil specialize pentru anumită aplicație, dar altele pot fi utile pentru o mare varietate de programe.

Ușurință de depanare și testare

Testarea și depanarea unui program în stil funcțional este mai ușoară.

-Iteratori-

Un iterator este un obiect care reprezintă un flux de date; acest obiect returnează valoarea unui element la un moment dat.

-Expresii generatoare și liste de comprehensiune-

Două operații comune care acționează pe ieșirea unui iterator sunt 1) efectuarea unei operații pentru fiecare element, 2) selectarea unui subset de elemente care îndeplinește anumită condiție. Listele de comprehensiune și expresiile generatoare sunt notări concise pentru astfel de operațiuni. Cu o listă de comprehensiune obțineți înapoi o listă Python. Expresiile generatoare returnează un iterator care calculează valorile după cum este necesar, nefiind necesar să materializeze toate valorile simultan. Expresiile generatoare sunt înconjurate de paranteze rotunde („()”) și liste de comprehensiune sunt înconjurate de paranteze pătrate („[]”).

-Generator-

Generatoarele sunt o clasă specială de funcții care simplifică sarcina de a scrie iteratorii. Funcțiile obișnuite calculează o valoare și o returnează, dar generatoarele returnează un iterator care returnează un flux de valori.

-Modulul itertools-

Modulul itertools conține o serie de iteratori utilizate și funcții pentru combinarea mai multor iteratori. Funcțiile modulului se încadrează în câteva clase largi:

- Funcții care creează un iterator nou pe baza unui iterator existent.
- Funcții pentru tratarea elementelor unui iterator ca argumente funcționale.
- Funcții pentru selectarea unor porțiuni a ieșirii unui iterator.
- O funcție pentru gruparea ieșirilor unui iterator.

-Modulul functools-

Modulul funcoools din Python 2.5 conține câteva funcții de ordin superior. O funcție de ordin superior ia una sau mai multe funcții ca intrare și returnează o nouă funcție.

-Modulul operator-

Conține un set de funcții corespunzătoare operatorilor din Python. Unele dintre funcțiile din acest modul sunt:

- Operații matematice: add(), sub(), mul(), floordiv(), abs(), ...
- Operații logice: not_(), truth().
- Operații în timp: and_(), or_(), invert().
- Comparări: eq(), ne(), lt(), le(), gt(), and ge().
- Identitate obiect: is_(), is_not().

-Funcții mici și expresia lambda-

Când scrieți programe în stil funcțional, veți avea nevoie adesea de mici funcții care să acționeze ca predicat sau care să combine elemente într-un fel. Un mod de a scrie funcții mici este să folosiți expresia lambda. lambda preia o serie de parametri și o expresie care combină acești parametri și creează o funcție anonimă care returnează valoarea expresiei.

Laboratorul 12: Paradigma calculului funcțional în Kotlin

Programarea funcțională este o paradigmă în care accentul este pus pe transformarea datelor cu expresii (ideal, aceste expresii ar trebui să nu aibă side effects).

Funcția de ordin superior 1 este o funcție care primește funcții ca parametri sau returnează o funcție. Funcțiile lambda pot fi utilizate ca parametri în alte funcții. Parametrul it ar trebui folosit doar în cazul în care este clar ce tip de date se folosește. De asemenea, dacă se dorește ca o funcție clasică să fie trimisă ca parametru, se poate utiliza double colon (::).

Într-un program, când o funcție modifică orice obiect/date din afara scopului propriu, acest lucru se numește efect secundar. O funcție care modifică o proprietate statică sau globală, modifică un argument, generează o excepție, scrie output-ul la consolă/în fișier sau apelează o altă funcție, are un efect secundar.

O funcție pură este o funcție a cărei rezultat returnat este complet dependent de parametrii acesteia. Pentru fiecare apel al unei funcții pure cu același parametru, aceasta va produce mereu același rezultat. De asemenea, o funcție pură NUtrebuie să cauzeze efecte secundare sau să apeleze alte funcții.

Funcțiile extensie permit modificarea tipurilor existente cu funcții noi. Sintaxa pentru adăugarea unei funcții extensie la un tip existent: NumeClasă.functieExtensie(listăParametri).

Un functor este un tip care definește o modalitate de a transforma (transform/map) conținutul lui.

Delegatul Delegates.notNull permite continuarea programului fără inițializarea proprietății notNullStr. Dacă acea variabilă este utilizată înainte de a fi inițializată, va genera o excepție. Funcția lateinit este o variantă mai simplă pentru a obține același comportament. Observație: Delegates.notNull și lateinit funcționează numai pentru proprietăți declarate cu var.

Spre deosebire de lateinit și Delegates.notNull, la funcția lazy trebuie specificată variabila de inițializare în momentul declarării, avantajul fiind că inițializarea nu va fi executată până când variabila este folosită.

Functia Delegates.observable are nevoie de doi parametri pentru a crea delegatul: valoarea inițială a proprietății și o funcție lambda care să fie executată de fiecare dată când se schimbă valoarea proprietății. Funcția lambda primește 3 parametri:

- o instanță deKProperty<out R>(o proprietate -valsauvar)
- valoarea veche a proprietății
- valoarea nou asignată

Dreptul de veto permite o verificare logică la fiecare asignare a unei proprietăți, unde se poate decide dacă se continuă cu asignarea sau nu.

Laboratorul 11: Utilizarea bibliotecilor Python pentru paralelism

-Introducere-

Execuția paralelă reprezintă executarea a două sau mai multe programe simultan de către un singur computer.

-multiprocessing-

multiprocesarea este un pachet care suportă inițializarea proceselor prin utilizarea subprocesselor. Pachetul oferă concurență locală și la distanță. În funcție de platformă, multiprocesarea acceptă trei moduri de a începe un proces. Aceste metode de pornire sunt:

spawn

Procesul părinte începe un nou proces. Procesul copil va moșteni doar acele resurse necesare pentru a rula metoda run(). În special, descriptorii și handler-ele de fișiere inutile din

procesul părinte nu vor fi moștenite. Începerea unui proces folosind această metodă este destul de lentă în comparație cu utilizarea fork sau forkserver.

fork

Procesul părinte folosește os.fork(). Procesul copilului, atunci când începe, este efectiv identic cu procesul părintelui. Toate resursele părintelui sunt moștenite de procesul copilului.

forkserver

Când programul pornește și selectează metoda de pornire a unui forkserver, se începe un proces de server. De atunci, ori de câte ori este nevoie de un nou proces, procesul părinte se conectează la server și solicită ca acesta să initializeze un nou proces. Procesul unui forkserver este single threaded, deci este sigur pentru acesta să utilizeze os.fork(). Nu sunt moștenite resurse inutile.

multiprocesarea acceptă două tipuri de canale de comunicare între proceze: cozi și conducte(pipe-uri). Pentru comunicare mesajelor se poate utiliza Pipe() (pentru o conexiune între două proceze) sau o coadă (care permite mai mulți producători și consumatori).

Atunci când efectuați o execuție paralelă, este de obicei cel mai bine să evitați utilizarea stării partajate pe cât posibil. Acest lucru este valabil în special atunci când se utilizează mai multe proceze. Cu toate acestea, dacă aveți nevoie cu adevărat de a utiliza unele date partajate, atunci multiprocesarea oferă câteva modalități de a face acest lucru: memoria partajată sau procesul server.

Clasa Pool reprezintă o serie de proceze de lucrători.

-asyncio-

asyncio este o bibliotecă pentru a scrie cod paralel folosind sintaxa async/await. asyncio oferă un set de API-uri de nivel înalt care:

- rulează coroutines Python concomitent și au control deplin asupra execuției lor;
- execută IO și IPC de rețea;
- controlează subprocese;
- distribuie sarcini prin cozi;
- sincronizează codul concurrent.

-threading-

Un fir este un flux de execuție separat.

Clasa Thread reprezintă o activitate care este rulată într-un fir de control separat. Există două moduri de a specifica activitatea: transmitând un obiect care poate fi apelat către constructor sau suprasolicitând metoda run() într-o subclasă. Nici o altă metodă (cu excepția constructorului) nu ar trebui să fie suprascrisă într-o subclasă.

Un lock este o primitivă de sincronizare care nu este deținută de un anumit fir atunci când este locked. Un lock se poate afla într-una din cele două stări, „lock” sau „unlock”. Este creat în starea unlocked. Are două metode de bază, acquire() și release().

O rlock este o primitivă de sincronizare care poate fi dobândită de mai multe ori de

către același fir.

O variabilă de condiție este întotdeauna asociată cu un fel de lock; aceasta poate fi transmisă sau una va fi creată implicit. Transmiterea unei intrări este utilă când mai multe variabile de condiție trebuie să partajeze același lock. Lock-ul face parte din obiectul condiției: nu trebuie să îl urmăriți separat.

Un semafor este una dintre cele mai vechi primitive de sincronizare și gestionează un contor intern care este decrementat de fiecare apel acquire() și incrementat de fiecare apel release().

Un eveniment este unul dintre cele mai simple mecanisme de comunicare între thread-uri: un thread semnalează un eveniment și alte thread-uri îl așteaptă.

-concurrent.futures-

Modulul concurrent.futures oferă o interfață la nivel înalt pentru executarea asincronă a apelurilor.

Execuția asincronă poate fi efectuată cu fire de execuție, folosind ThreadPoolExecutor sau procese separate, folosind ProcessPoolExecutor. Ambele implementează aceeași interfață, definită de clasa abstractă Executor.

ThreadPoolExecutor este o subclăsă Executor care folosește un grup de fire pentru a executa apeluri în mod asincron. Deadlock-uri pot apărea atunci când apelul asociat cu un Future așteaptă rezultatele unui alt Future.

Clasa ProcessPoolExecutor este o subclăsă Executor care folosește un grup de procese pentru a executa apeluri în mod asincron. Apelarea metodelor Executor sau Future dintr-un apel trimis la un ProcessPoolExecutor va avea ca rezultat un impas.

Clasa Future încapsulează execuția asincronă a unui apelabil. Instanțele viitoare sunt create de Executor.submit().

-subprocess-

Modulul subprocessului vă permite să generați noi procese, să vă conectați la conductele de intrare/ieșire/eroare și să obțineți codurile lor de return. Abordarea recomandată pentru invocarea subprocesselor este utilizarea funcției run() pentru toate cazurile de utilizare pe care le poate gestiona.

Laboratorul 10: Corutine - suport nativ pentru paralelism în Kotlin

-Funcții recursive-

Kotlin suportă un stil de programare funcțională cunoscut ca recursivitatea coadă (tail recursion). Aceasta permite unor algoritmi care în mod normal ar fi fost scriși cu bucle să fie scriși cu o funcție recursivă, dar fără riscul de „stack overflow”. Spre deosebire de recursivitatea normală în care

toate apelurile recursive sunt efectuate la început și apoi se calculează rezultatul din valorile returnate la urmă, în recursivitatea coadă calculele sunt executate primele, apoi apelurile recursive (apelul recursiv trimite rezultatul pasului curent către următorul apel recursiv).

-Coroutine-

În esență, coroutinile sunt fire light-weight, ceea ce înseamnă că durata de viață a noii coroutine este limitată doar de durata de viață a întregii aplicații. Întârzierea (delay) este o funcție specială de suspendare, care nu blochează un fir, ci suspendă coroutina și poate fi utilizată doar dintr-o corutină.

Dacă vorbim despre concurență structurată, în loc să lansăm coroutine în GlobalScope, la fel cum facem de obicei cu firele de execuție (firele sunt întotdeauna globale), putem lansa coroutine în domeniul specific al operației pe care o efectuăm. Este posibil să vă declarați propriul domeniu de aplicare folosind constructorul coroutineScope. Creează un scop și nu se termină până când nu se finalizează toți copiii lansați.

runBlocking și coroutineScope pot arăta similar, deoarece ambele așteaptă corpul lor și toți copiii să se finalizeze. Principala diferență este că metoda runBlocking pune firul curent pe așteptare, în timp ce coroutineScope doar suspendă, eliberând firul de bază pentru alte utilizări.

Coroutinile sunt o formă de procesare secvențială: doar una se execută la un moment dat. Firele de execuție sunt (cel puțin conceptual) o formă de procesare simultană: mai multe fire pot fi executate la un moment dat.

Laboratorul 9: Design patterns în Python

-Decorator-

Modelul Decorator permite utilizatorului să adauge funcționalități noi la un obiect existent, fără a-i modifica structura. Acest tip de model de design se încadrează în modelul structural, deoarece acest model acționează ca un înveliș pentru clasa existentă.

Acest model creează o clasă decorator, care „înfășoară” clasa originală și oferă o funcționalitate suplimentară, păstrând intactă semnătura metodelor clasei.

Motivul unui model de decorator este să atașeze dinamic responsabilități suplimentare unui obiect.

-Iterator-

Modelul de design iterator se încadrează în categoria de modele de design comportamental. Dezvoltatorii întâlnesc modelul iterator în aproape fiecare limbaj de programare. Acest model este utilizat astfel încât ajută la accesarea elementelor unei colecții (clasa) în mod secvențial, fără a înțelege designul stratului de bază.

-Generator-

Generatoarele reprezintă o modalitate simplă de a crea iteratori, toate cerințele anterioare fiind gestionate în mod automat de generator.

Definirea unui generator presupune utilizarea cuvântului cheie `yield` în loc de `return`. Acesta trebuie să conțină cel puțin un `yield` (poate conține mai multe alte `yield`-uri, `return`-uri).

Diferența între `return` și `yield`:

`return` - încheie complet execuția funcției

`yield` - pune pauză funcției, salvând stările și continuând mai târziu de unde a rămas.

Avantajele utilizării unui generator în loc de iterator:

-ușor de implementat

-eficient din punct de vedere al memoriei

-permite reprezentarea unui stream infinit

-generatoarele pot fi folosite pentru a realiza un pipeline cu o serie de operații.

-Fabrică-

Modelul fabrică intră în categoria design pattern-urilor de creație. Oferă unul dintre cele mai bune moduri de a crea un obiect. În modelul fabrică, obiectele sunt create fără a expune logica clientului și se referă la obiectul nou creat folosind o interfață comună.

Modelele fabrică sunt implementate în Python prin metoda `__init__`. Tipul de obiect utilizat în metoda fabrică este determinat de sirul care este transmis prin metodă.

-Pipeline-

În programare, o conductă, cunoscută și sub denumirea de conductă de date, este un set de elemente de prelucrare a datelor conectate în serie, unde ieșirea unui element este intrarea următorului. Elementele unei conducte sunt adesea executate în paralel sau în porțiuni de timp.

-Lanț de responsabilități-

Modelul lanțului de responsabilități este utilizat pentru a realiza cuplarea liberă în software unde o solicitare specificată de client este transmisă printr-un lanț de obiecte incluse în acesta. Ajută la construirea unui lanț de obiecte. Cererea intră de la un capăt și se mută de la un obiect la altul.

Acest model permite unui obiect să trimită o comandă fără să știe ce obiect va gestiona cererea.

-Comandă-

Modelul comandă adaugă un nivel de abstractizare între acțiuni și include un obiect, care invocă aceste acțiuni. În acest model de proiectare, clientul creează un obiect de comandă care include o listă de comenzi care trebuie executate. Obiectul de comandă creat implementează o interfață specifică.

-Automat finit de stări-

Metoda stării este un model de design comportamental care permite unui obiect să-și schimbe comportamentul atunci când apare o schimbare în starea sa internă. Ajută la implementarea stării ca o clasă derivată a interfeței cu modelul de stare. Dacă trebuie să schimbăm comportamentul unui obiect în funcție de starea lui, putem avea o variabilă de stare în obiectul respectiv și putem folosi blocul de condiții if-else pentru a efectua diferite acțiuni bazate pe stare. Poate fi denumită ca automat de stări orientat pe obiecte. Implementează tranzițiile de stare invocând metode din superclasa modelului.

-Proxy-

Metoda Proxy este un model de design structural care vă permite să furnizați înlocuirea unui alt obiect. Aici, folosim diferite clase pentru a reprezenta funcționalitățile altrei clase. Partea cea mai importantă este că aici creăm un obiect care are o funcționalitate originală pe care să o ofere lumii exterioare.

-Cache-

În calcul, un cache este un strat de stocare a datelor de mare viteză care stochează un subset de date, de natură tranzitorie, astfel încât cererile viitoare pentru aceste date sunt furnizate mai repede decât este posibil accesând locația de stocare primară a datelor. Caching-ul vă permite să reutilizați eficient datele preluate anterior sau calculate.

-Strategie-

Modelul strategie este un tip de model comportamental. Principalul obiectiv al modelului strategie este de a permite clientului să aleagă dintre diferiți algoritmi sau proceduri pentru a finaliza sarcina specificată. Diferiți algoritmi pot fi schimbați înăuntru și înafara fără complicații pentru sarcina menționată.

Acest model poate fi utilizat pentru a îmbunătăți flexibilitatea atunci când sunt accesate resurse externe.

Laboratorul 8: Design patterns în Kotlin

-Stare-

Scopul modelului: Permite unui obiect să își modifice comportamentul atunci când starea sa internă se schimbă. Obiectul va părea că își schimbă clasa.

Aplicabilitate: Modelul stare se utilizează în următoarele cazuri:

-Comportamentul unui obiect depinde de starea sa și trebuie să-și schimbe comportamentul în timpul execuției, în funcție de starea respectivă;

-Operațiile au declarații conditionale compuse, mari, care depind de starea obiectului.

Consecințe:

- Localizează un comportament specific stării și comportamentul partitilor pentru diferite stări;
- Face tranzițiile între stări explicite;
- Obiectele stării pot fi partajate.

-Compus-

Modelul compus este un model de proiectare a compartimentării și descrie un grup de obiecte care este tratat la fel ca o singură instanță a același tip de obiect. Intenția unui componit este de a „compune” obiecte în structuri de arbori pentru a reprezenta ierarhii parțiale. Vă permite să aveți o structură de arbore și să cereți fiecărui nod din structura copacului să efectueze o sarcină.

-Fabrică abstractă-

Modelul fabrică abstractă funcționează în jurul unei super-fabrici care creează alte fabrici. Această fabrică mai este numită și fabrică de fabrici. Acest tip de model de design se încadrează în modelul creațional, deoarece oferă una dintre cele mai bune metode de a crea un obiect.

În modelul Abstract Factory, o interfață este responsabilă pentru crearea unei fabrici de obiecte conexe, fără a specifica explicit clasele lor. Fiecare fabrică generată poate oferi obiectele conform modelului Factory.

-Observator-

Modelul de observare este un model de proiectare software în care un obiect, numit subiect, menține o listă a dependentilor săi, numiți observatori și îi notifică automat cu privire la orice modificări de stare, de obicei apelând la una dintre metodele lor. Modelul observator se încadrează în categoria modelului comportamental.

-Memento-

Modelul memento este un model de design comportamental. Modelul memento este folosit pentru a restabili starea unui obiect la o stare anterioară. Pe măsură ce aplicația dvs. progresează, poate doriți să salvați punctele de control din aplicație și să le restabiliți mai târziu la aceste puncte de control.

-Pod-

Modelul pod este utilizat atunci când trebuie să decuplăm o abstractizare de la punerea în aplicare a acesteia, astfel încât cele două să poată varia independent. Acest tip de model de design se încadrează în modelul structural, deoarece decuplează clasa de implementare și clasa abstractă, oferind o structură de puncte între ele.

Acest model implică o interfață care acționează ca o punte care face ca funcționalitatea claselor concrete să fie independentă de clasele implementatoare de interfață. Ambele tipuri de clase pot fi modificate structural, fără a se afecta reciproc.

-Constructor-

Modelul constructor construiește un obiect complex folosind obiecte simple și utilizând o abordare pas cu pas. Acest tip de model de design se încadrează în modelul creațional, deoarece oferă una dintre cele mai bune metode de a crea un obiect.

O clasă Builder construiește final obiectul pas cu pas. Acest constructor este independent de alte obiecte.

-Prototip-

Modelul prototip se referă la crearea unui obiect duplicat, ținând cont de performanță. Acest tip de model de design se încadrează în modelul creațional, deoarece oferă una dintre cele mai bune metode de a crea un obiect.

Acest model implică implementarea unei interfețe prototip care să creeze o clonă a obiectului curent. Acest model este utilizat atunci când crearea obiectului este direct costisitoare. De exemplu, un obiect trebuie creat după o operațiune de bază de date costisitoare. Putem memora în cache obiectul, îi returnăm clona la următoarea solicitare și actualizăm baza de date, după cum este necesar, reducând astfel apelurile la baza de date.

-Fațadă-

Modelul fațadă ascunde complexitățile sistemului și oferă o interfață pentru client care poate accesa sistemul. Acest tip de model de design se încadrează în structura structurală, deoarece adaugă o interfață la sistemul existent pentru a-și ascunde complexitățile.

Acest model implică o singură clasă care furnizează metode simplificate solicitate de către client și apeluri delegați la metodele claselor de sistem existente.

-Mediator-

Modelul mediator este utilizat pentru a reduce complexitatea comunicării între mai multe obiecte sau clase. Acest model oferă o clasă mediator care gestionează în mod normal toate comunicațiile dintre diferite clase și sprijină întreținerea ușoară a codului prin cuplaj liber. Modelul mediator se încadrează în categoria modelului comportamental.

-Adaptor-

Modelul adaptor funcționează ca o punte de legătură între două interfețe incompatibile. Acest tip de model de design se încadrează în modelul structural, deoarece acest model combină capacitatea a două interfețe independente.

Acest model implică o singură clasă responsabilă de unirea funcționalităților interfețelor independente sau incompatibile.

-Singleton(Bonus)-

Modelul singleton este unul dintre cele mai simple modele de design în Java. Acest tip de model de design se încadrează în modelul creational, deoarece acest model oferă una dintre cele mai bune metode de a crea un obiect.

Acest model implică o singură clasă care este responsabilă de crearea unui obiect, asigurându-se în același timp că este creat doar un singur obiect. Această clasă oferă o modalitate de a accesa singurul său obiect care poate fi accesat direct fără a fi necesar să instaureze obiectul clasei.

-Vizitator(Bonus)-

Modelul vizitator este unul dintre modelele de design comportamental. Este utilizat atunci când trebuie să executăm o operație pe un grup de obiecte similare. Cu ajutorul modelului vizitator, putem muta logica operațională de la obiecte la o altă clasă.

Laboratorul 7: Colecții și generice în Kotlin

-Colecții-

Kotlin face distincție între colecțiile mutabile și cele imutabile. O colecție mutabilă poate fi actualizată pe loc prin adăugarea, ștergerea sau înlocuirea unui element. O colecție imutabilă, deși oferă aceleași operații (adăugare, ștergere, înlocuire) prin funcțiile operator, va crea o nouă colecție, lăsând-o pe cea veche neschimbată.

O **secvență** este un tip iterabil pe care se pot efectua operații fără crearea de colecții intermediare inutile, prin executarea tuturor operațiilor aplicabile pe fiecare element înainte de trecerea la următorul.

Secvențele sunt leneșe (lazy), funcțiile intermediare corespunzătoare pentru procesarea secvențelor nu fac calcule, ci returnează o nouă secvență ce o decorează pe cea anterioară cu nouă operație. Toate calculele sunt evaluate în timpul operației terminale.

Procesarea secvențelor este în general mai rapidă decât procesarea directă a colecțiilor unde există mai mult de un pas de procesare.

ATENȚIE: Există cazuri în care secvențele nu sunt recomandate. Spre exemplu, în cazul sortării prin apelul funcției `sorted`, întrucât este necesară parcurgerea întregii colecții.

În ceea ce privește stream-urile din Java, acestea sunt tot „leneșe”, fiind colectate înpasul final de procesare. De asemenea, stream-urile Java sunt mult mai eficiente pentru procesarea colecțiilor decât funcțiile de procesare corespunzătoare din Kotlin.

Diferențe între stream-urile Java și secvențele Kotlin:
secvențele Kotlin au multe multe funcții de procesare (fiind definite ca funcții extensie)
Stream-urile Java pot fi pornite în mod paralel, utilizând o funcție `parallel`.

Se recomandă utilizarea stream-urilor Java doar pentru procesări computaționale grele, unde se poate beneficia de modul paralel.

-Generice-

Programarea generică (sau genericele) este o tehnică prin care funcțiile pot fi scrise cu tipuri care nu sunt specificate în momentul scrierii și sunt folosite mai târziu cu diferite tipuri dedate.

-Polimorfism mărginit-

Funcțiile care sunt generice pentru orice tip sunt utile, dar cumva limitate. Adesea, va fi nevoie de scrierea unor funcții care sunt generice pentru unele tipuri care au o caracteristică comună. Spre exemplu, definirea unei funcții care returnează minimul dintre două valori, pentru oricare valori ce suportă noțiunea de comparare.

Pentru a forța valorile să aibă aceea noțiune de comparare, trebuie restricționate tipurile generice la cele care suportă funcțiile care trebuie invocate. Cu alte cuvinte, mărginim funcția polimorfică (generică), acest lucru fiind numit polimorfism mărginit.

-Măginirii superioare-

Kotlin suportă un tip de măginire a polimorfismului, mai precis măginirea superioară. Din denumire, se remarcă că tipurile generice sunt restricționate la cele care sunt subclase ale măginirii.

Comparable este un tip din biblioteca standard care definește metoda compareTo care returnează o valoare mai mică decât 0 dacă primul element este mai mic, mai mare decât 0 dacă al doilea element este mai mic, egală cu 0 dacă elementele sunt egale.

-Măginirii multiple-

Toate măginirile superioare sunt scrise ca și clauze where și formează o uniune de măginire superioară.

-Varianța-

Kotlin oferă:

- varianță la momentul declarării (declaration-site variance)
- proiecțiile de tip
- proiecțiile stă (utilizarea unui argument tip necunoscut)

Laboratorul 6: Utilizarea POO în Python

-Concepție de bază-

Abstractizare: este procesul de grupare de proprietăți/carakteristici esențiale ale unui obiect ignorând detaliile neneccesare.

Încapsulare: constă în separarea aspectelor externe ale unui obiect care sunt accesibile altor obiecte de aspectele interne ale obiectului care sunt ascunse celorlalte obiecte.

Moștenire: este relația dintre clase în care o clasă moștenește structura și comportamentul (funcțiile) definite în una sau mai multe clase.

Polimorfism: descrie conceptul prin care obiecte de diferite tipuri pot fi accesate prin intermediul aceleiași interfețe.

Laboratorul 4: Utilizarea principiilor SOLID în Kotlin

-Principiile S.O.L.I.D-

Principiul Responsabilității unice (Single) - O clasă trebuie să aibă o singură rațiune pentru a se schimba.

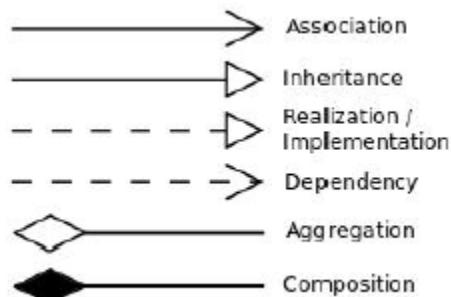
Principiul Închis/Deschis (Open/Closed) - Entitățile software trebuie să fie deschise pentru extindere dar închise pentru modificare.

Principiul substituției Liskov - Copiii claselor nu au voie să încalce definițiile de tip din clasa părinte. Adică funcțiile care folosesc referințe la clase de bază, trebuie să poată manipula într-un mod transparent instanțele claselor derivate din acestea.

Principiul separării Interfețelor - Nici un client nu trebuie forțat să depindă de metode pe care nu le folosește. Numărul de membri din interfață care este vizibil pentru clasele dependente trebuie minimizat. Clasele mari vor implementa mai multe interfețe mai mici care grupează funcțiile după maniera lor de utilizare.

Principiul Dependenței inverse - modulele de nivel arhitectural superior nu trebuie să depindă de cele de nivel inferior. Ambele trebuie să depindă de abstracții care, la rândul lor nu trebuie să depindă de detalii. Practic detaliile depend de abstracții, nu invers. Dacă aceasta dependență nu este vizibilă în faza de proiectare atunci ea se construiește.

-Elemente UML-



-Association - relație de asociere între două clase (clasa sursă folosește membri din țintă (target)).

-Dependency - relație de dependență între două clase (clasa sursă folosește clasa din țintă ca un parametru sau ca o variabilă în interiorul funcțiilor).

-Inheritance - moștenire (clasa sursă derivează clasa țintă, adăugându-i noi funcționalități).

-Realization - Implementarea unei interfețe (clasa sursă implementează interfața din țintă)

-Aggregation - agregare (clasa sursă folosește de mai multe ori clasa din target)

-Composition - compoziție (clasa sursă folosește de mai multe ori clasa din țintă și în același timp o derivă (prin moștenire). Un exemplu pentru acest caz este structura arborescentă a unui GUI, unde fiecare element grafic conține un tablou de alte elemente grafice pe care îl poate deriva prin moștenire)