

Curs2

La ce e bun calculul Lambda?

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

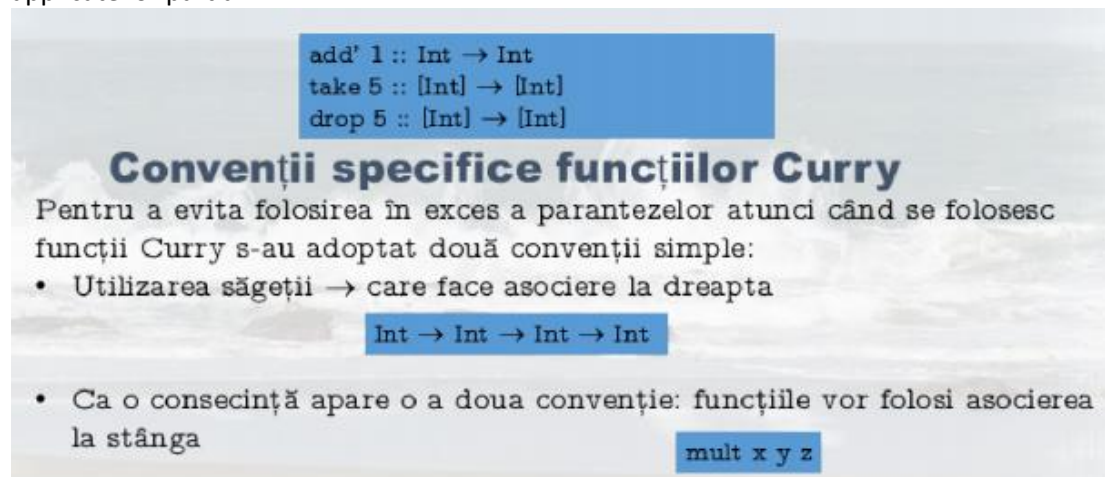
Expresiile Lambda pot fi folosite pentru a da un înțeles formal funcțiilor Curry.

Ce este o funcție Curry?

Funcție curry: funcție care returnează o nouă funcție atunci când este aplicată pe mai puține argumente decât aștepta ea

Care este avantajul funcțiilor Curry?

Sunt mult mai flexibile decât funcțiile bazate pe tuple, în special datorită faptului că 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 structuri ale compilatorului

Curs 3

Paradigme orientate obiect

Kotlin

Variabile var & val

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

```
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

Deși Kotlin este un limbaj cu tipuri tari de date el nu necesită declararea obligatorie de tip, deci ca și Python suportă inferența de tip

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

Câteodată este util să lucrăm explicit:

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

Tipuri de date

Ca și în Python, în 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 implicite?

```
val int = 123
val long = int.toLong()

val float = 12.34F
val double = float.toDouble()
```

<i>Long</i>	64
<i>Int</i>	32
<i>Short</i>	16
<i>Byte</i>	8
<i>Double</i>	64
<i>Float</i>	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()`

Variable 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 șir 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 si "current receiver"

Scope

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

Vizibilitate (public, private, protected, internal)

<code>class Person</code>	<code>internal class Person</code>
<code>{ private fun age(): Int = 21 }</code>	<code>{ fun age(): Int = 21 }</code>

NULL

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

Conversia explicita 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 listă) și permite și duplicate. Interfata List pe lângă operațiile moștenite de la interfata Collection conține operații pentru:

- accesul elementelor pe baza poziției lor în listă
- cautarea unui obiect specificat prin returnarea poziției
- 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. Modelează mulțimea din matematică. Nu conține metode noi, doar cele moștenite de la interfata Collection, la care adaugă restricția cu duplicatele. Două obiecte de tip Set sunt egale dacă ele conțin aceleași elemente

Curs 4

Principiile SOLID

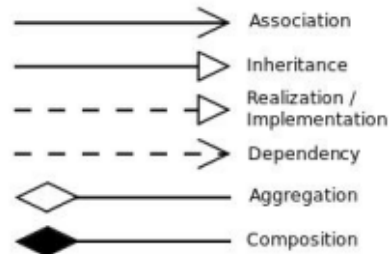
Ce este de fapt UML?

- *" 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."*

Sageți

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-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)

Principiile SOLID

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

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

Principiul Substitutiei **Liskov**- - Copii 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 depind de abstracții, nu invers. Dacă aceasta depedință nu este vizibilă în faza de proiectare atunci ea se construiește.

Le reluaaaaaaaaaaaaaaam

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 Principiul 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? Încălțările 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

-interacțiunea este guvernata de o bucla infinita

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/întreruperi)
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 bucla

```
main()
{
    ...inițializează structurile de date ale aplicației ...
    ...inițializează 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
- Incurajează 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 Correlation

- **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 (opțional 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 corupt, 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, clienții nu salvează 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) aceluiași 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 comenzile 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 comenzile 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 clienții 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 functionala in Python

Programare functionala reprezinta o metoda de proiectare si dezvoltare a aplicatiilor bazata pe evaluarea de expresii care nu au in vedere modificarea starii. In cadrul acestui stil de programare codul este oferit prin intermediul functiilor. Programarea functionala promoveaza un cod fara efecte secundare, fara modificari ale starii variabilelor.

Programare imperativa

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

Programare procedurala

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

Programare orientate 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) → 'spamspamspameggseggseggseggseggsegg'  
>>> 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	Întelesul acesteia
<code><list>.append(x)</code>	Adaugă x la sfârșitul listei
<code><list>.sort()</code>	Sortează lista (o funcție de sortare poate fi trimisă ca parametru)
<code><list>.reverse()</code>	Inversează lista
<code><list>.index(x)</code>	Indexul pentru prima apariție a lui x în listă
<code><list>.insert(i, x)</code>	Inserează x pe poziția i
<code><list>.count(x)</code>	Numărul de apariții al lui x în listă
<code><list>.remove(x)</code>	Șterge prima apariție a lui x în listă
<code><list>.pop(i)</code>	Șterge element i și îi 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 atribute.

• 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 în 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 bibliotecasys, 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)
```

Specificatorii de acces sunt introduși 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 incluși 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. Variația (Variance)

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

Kotlin nu are asemenea tipuri, dar oferă:

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

2.2.1. Variația 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.

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

Pe lângă adnotarea de variață (**out**), Kotlin oferă și o **adnotare complementară de variață, 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: variația 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 variață 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 așa *numitele proiecții stea*. Trebuie definită o proiecție a unui tip generic pentru care fiecare instanțiere concretă a acelui 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 mărghirea 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 mărghirea 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 proiectii 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 instantanării 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 diferitele sub-clase bazate pe intrarea furnizată și utilizările *if-elses* 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 inconsecvente.

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

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 de-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 assemblează 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:

- Acesta reduce nevoia de sub-clasificare.
- Ascunde complexități ale creării de obiecte.
- Clienții 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 în **Hashtable** și returnează clona lor atunci când este solicitat.

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 aceleași 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 delegați 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 decuplează 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 așa 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 dependente 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 partițiilor 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* caută 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 comenzilor. Un *broker* de clasă este creat ca obiect de invocator. Poate lua 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 retrimite iteratorul. Clasele de beton care implementează interfața *Container* vor fi responsabile pentru implementarea interfeței *Iterator* și utilizarea acesteia

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 dovadă 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 așa 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 . ^[1] 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 diferenta dintre Coroutine si thread-uri?

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

Firele de executie sunt (cel putin conceptual) o fforma 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 coroutine. 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 așteptare 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 așteptare 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

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 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 modificatorul *tailrec* și are forma corespunzătoare, compilatorul optimizează recursivitatea, rezultând o versiune bazată pe o buclă rapidă și eficientă în loc

Corutine 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 subrutine

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 starea 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 starea blocat în neblocat.

O funcție din fabrică (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 (reintrat). 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 numărul 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 thread-urile 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 intample. 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 intample (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 thread-urile care asteptau event-ul cu `wait()` vor fi trezite

Cozi de mesaje

Sunt folosite de procese pentru a comunica între ele prin mesaje. Aceste mesaje își păstrează ordinea în interiorul cozii de mesaje. Sunt mecanisme de comunicare unidirecționale atât pe Linux, cât și pe Windows.

Memorie partajata

Acest mecanism permite comunicarea între procese prin accesul direct și partajat la o zonă de memorie bine determinată. Este un mod mai rapid de comunicare între 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, modulul 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 coroutine în bucle de eveniment. Dacă o corupție așteaptă un viitor, sarcina suspendă execuția coroutinei și așteaptă finalizarea viitorului. Când se termină Viitorul, se reia execuția coroutinei învelite.

Asyncio

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

Event loops

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

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 singură 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 modifier `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 parantezele 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.

Funcția map

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

Funcția filter

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

Funcția FlatMap

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

Funcția drop – taiere sumultimi

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

Funcția take – extragere din submultimi

Returns a list containing first n elements.

Funcția 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 assignată

1.6.2.4 Funcția Delegates.vetoable

Dreptul de veto permite o verificare logică la fiecare assignare a unei proprietăți, unde se poate decide dacă se continuă cu assignarea 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 șiruri, liste, tuple etc.

enumerate()

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

Reversed()

returnează iteratorul inversat al secvenței date.

Decorator-stil macro

Decoratoarele macro vă permit să-ți secționezi 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