PRIMITIVE și OBIECTE

PRIMITIVE: tipurile de primitive sunt tipuri încorporate în limbajul JAVA;
└ ele reprezintă valori simple și nu sunt obiecte
└ au dimensiune fixă; nu suportă metode: nu au metode asociate; sunt utilizarea valorilor brute

- eficiență: pt. că sunt stocate direct în memorie (în stack) sunt mai eficienți în ceea ce privește utilizarea memoriei și performanța
- inițializare implicită:

	octeți	val. implicită
byte	1	0
short	2	0
int	4	0
long	8	0L
float	4	0.0F
double	8	0.0D
char	2	'\0000' (null)
boolean	1	false

OBIECTE (WRAPPER CLASS): tipuri de obiecte (sau wrapper class) care înfășoară tipul de primitivă într-un obiect
└ suportă metode; pot fi null; trebuie inițializate explicit pt. a nu fi null; AUTOBOXING + UNBOXING

AUTOBOXING - procesul prin care un tip primitiv este convertit într-un obiect al clasei wrapper; este permisă conversia automată între tipurile primitive și obiectele wrapper corespunzătoare

UNBOXING - este procesul invers al autoboxing-ului, deci un obiect al unei clase wrapper este convertit într-un tip primitiv.

AUTOBOXING

```
int a = 50;
Integer a2 = new Integer(a);
Integer a3 = 5;
```

UNBOXING

```
Integer i = new Integer(50);
int a = i;
```

STROIE MONICA ♡

COLLECTIONS → se referă la un grup de obiecte reprezentate ca o unitate

→ clasele în Java collection → Collection – List, Set (java.util.Collection)

→ Map (java.util.Map)

Arrays – structuri fixe în dimensiune; stochează doar tipuri de de date omogene

Collection – crește în dimensiune ori de câte ori este nevoie; stochează elemente omogene + heterogene

Collection	vs	Collections
– interfață – rep. un grup de obiecte individuale ca un întreg		– clasă utilitară – definește diferite metode utilitare, folosite pt. a opera pe colecții

MAP = stochează perechi cheie-valoare, unde fiecare cheie este unică; permite obținerea, adăugarea, ștergerea și verificarea existenței unei valori legată pe cheia sa.

TreeMap => păstrează elementele ordonate în funcție de cheia lor (ordinea naturală sau una definită de un comparator)

=> op de căutare/inserare sunt mai lente decât în cazul HASHMAP(O(log n))

=> nu permite chei null

HASHMAP	HASHTABLE
- stochează structuri pe tip (cheie, valoare) - poate stoca o singură cheie nulă și multiple valori nule - întoarce valoarea în funcție de cheie - acces rapid: O(1) pt. adăugare/ștergere/obținere - folosește funcții hash - pt. a mapa cheile la un indice din vector - nu este ordonat: nu este păstrată ordinea în care elem. au fost adăugate - permite valori duplicate (NU CHEI) - thread-unsafe: dacă mai multe thread-uri accesează în același timp un hashMap ⇒ inconsistență în date	- stochează structuri de tip cheie-valoare - nici cheile și nici valorile nu pot fi nule - cheile trebuie să implementeze metodele hashCode() + equals() - valoarea rezultată din hash-urea cheii este folosită ca index la care valoarea este stocată în tabelă - thread-safe ⇒ synchronized → poate fi share-uit de mai multe thread-uri - demodată mai nou se folosește ConcurrentHashMap ↓ permite o valoare null pt. valori, dar nu și pt. chei
ARRAYLIST	**LINKED LIST**
- permite accesul direct la orice element prin index, ceea ce îl face să fie foarte rapid pt. op. de citire - complexitate: O(1) - accesul la un element (get/set), O(1) - adăugarea unui element la sfârșit, O(n) - inserarea/ștergerea - pt. care necesită deplasarea elementelor - ideal pt. momente când se dorește acces excesiv la elemente prin index / când adăugările + ștergerile se [illegible]	- listă dublu înlănțuită ceea ce înseamnă că fiecare element (nod) conține referințe către elementul precedent și următor; această structură facilitează inserările și ștergerile rapide - complexitate: O(n) - accesul la un element, deoarece trebuie parcursă lista până la acel element O(1) - adăugarea/ștergerea unui elem. de la începutul/finalul listei • inserarea/ștergerea unui elem. de la mij. listei - O(n): pt. a accesa locația

Utilizare LINKEDLIST: - ideal pentru liste care fac frecvent inserții / ștergere în orice parte a listei
- când performanța accesării elementelor nu este critică sau se întâmplă rar.

LIST	SET
- secvență indexată	- secvență neindexată
- permite elemente duplicate	- nu permite elemente duplicate
- elementele pot fi accesate după poziție	- accesul prin poziție nu este permis
- pot fi stocate mai multe elem. nule	- doar un element null
- ArrayList, LinkedList, Vector, Stack	- HashSet, LinkedHashSet

ITERATOR (cursor universal)	LISTIERATOR
- poate traversa elemente prezente în Collection numai în direcția înainte	- poate traversa elemente prezente în colecție atât în direcția înainte, cât și înapoi
- Map / List / Set	- List
- indecșii nu pot fi obținuți	- nextIndex() + previousIndex() ← pt. a obține indecși ai elementelor în orice moment de timp al parcurgerii listei
- nu poate modifica / înlocui elem. prezente în colecție	- putem modifica / înlocui elem. cu ajutorul set(E e).

HashSet	LinkedHashSet
Nu garantează nicio ordine a elementelor	Păstrează ordinea inserării elementelor

SPRING FRAMEWORK

- oferă infrastructura de suport necesară unei aplicații Java
- se ocupă de infrastructura de bază cum ar fi:
  - gestionarea tranzacțiilor
  - gestionarea sesiunilor
  - injectarea dependințelor
  - accesul la date

- CARACTERISTICI:
  - Dependency injection
  - Programe orientate pe aspect (ASPECT) = separă aspectele transversale (logare + tranzacții) de logica de business, îmbunătățind modularizarea
  - Acces la date = simplifică interacțiunea cu bazele de date prin abstractizarea codului boilerplate pt. op. CRUD
    - boilerplate → cod reutilizabil
  - Tranzacționalitate = este fundamentul, oferă funcționalități de bază și un set vast de capabilități

|| IOC = principiu de bază care guvernează modul în care obiectele sunt gestionate și interacționează între ele

- în centrul Spring Framework se află containerul Spring ioc
- containerul Spring ioc creează obiecte, le conectează, le configurează și le gestionează întregul ciclu de viață
  - folosește Dependency Injection pentru a gestiona componentele care alcătuiesc o aplicație
  - containerul primește instrucțiuni pentru care obiecte trebuie să instanțieze, să configureze și să asamblieze, citind metadatele de configurare furnizate
    - metadatele → pot fi furnizate prin xml/adnotări JAVA/cod JAVA

Avantaje IoC (Inversion of control)

→ flexibilitate : se pot modifica / înlocui dependințe ușor, fără a modifica logica internă a clasei

→ reutilizare : deoarece Spring controlează obiectele, atunci acestia pot fi refolosite și distribuite mai eficient în cadrul aplicației

→ testare ușoară : ușurința testării cu mock-uri

Dependency Injection = design pattern din Spring care ne ajută la reducerea cuplării dintre clase

↳ permite obiectelor să primească referințele lor externe (dependențe) în loc să le creeze intern. Astfel este facilitată testarea, flexibilitatea și gestionarea dependințelor în aplicație.

↳ se poate realiza prin 3 metode:

1) INJECȚIE PRIN CONSTRUCTOR

```
public class ServiceA {
    private final RepA repA;
not mandatory → @Autowired
    public ServiceA (RepA repA) {
        this.repA = repA;
    }
}
```

2) INJECȚIE PRIN SETTER

```
{ ... public setRepA (RepA repA) {
        this.repA = repA; }}
```

3) INJECȚIE PRIN CÂMP

```
@Autowired
{ ... private final RepA repA; ... }
```

@Autowired -> folosită pentru a injecta automat dependințe într-un spring bean.

• Tipuri de containere IOC → BeanFactory = clasă ce conține o colecție de bean-uri. Instanțiază bean-ul ori de câte ori este cerut.

↓ ApplicationContext = interfață construită deasupra interfeței BeanFactory

• Tipuri de BEAN-uri: - implicit Singleton (o instanță | SPRING IOC container)
BEAN SCOPES
- Prototype @Scope("prototype") (o instanță / fiecare request)
- Request | Session | Application → o instanță / App context

o instanță / HTTP request o instanță / HTTP session

BEANS – este un obiect care este instanțiat, asamblat și gestionat de containerul Spring IOC

- reprezintă coloana vertebrală a aplicației permițând injecția de dependințe și gestionarea ciclului de viață al unui obiect.

Definirea unui BEAN: → ANOTĂRI: @Component, @Service, @Repository care autodetectează și configurează bean-urile în timpul scanării claselor

↓ FISIERE XML sau CLASE de CONFIG JAVA, unde bean-urile sunt explicit definite și configurate.

@Component = indică faptul că sunt componente Spring și pot fi autodetectate printr-o scanare a documentelor și clasa o să fie configurată ca un BEAN

→ din ea derivă mai multe anotări ca:

@Repository = indică faptul că o clasă oferă mecanisme pentru stocare, recuperare, actualizare, căutare a obiectelor

→ facilitează gestionarea excepțiilor specifice persistenței

@Service = nu există nicium comportament diferit față de @Component

→ se poate folosi @Service peste @Component în clasele de nivel service

Scanned with CamScanner

@Controller - marchează o clasă ca controller Spring WEB MVC

↳ bean-urile marcate cu această adnotare permite să recunoască clasa ca un controller și să faciliteze maparea URURILOR și gestionarea răspunsului

↳ utilizat cu @RequestMapping | @GetMapping | @PostMapping

@RestController = @Controller + @ResponseBody

↖ pt. return type JSON | XML

SPRING BOOT = proiect din cadrul ecosistemului Spring care are scopul de a simplifica procesul de configurare și publicare a aplicațiilor bazate pe Spring.

↳ caracteristici:
- → oferă configurații prestabilite
- → oferă "starter" dependencies pt. a simplifica adăugarea de componente și dependințe în proiect
- → server incorporat Tomcat sau Jetty
- → oferă posibilitatea de a monitoriza aplicația cu Spring Boot Actuator (/actuator, /health, /info, /metrics)
- → încearcă să configureze automat aplicația pe baza dependințelor adăugate
- → se bazează pe Spring, eliminând configurarea manuală.

ContextLoaderListener = componentă utilizată pentru încărcarea contextului aplicației Spring

↳ inițializează containerul IoC, managerizează beans, permite componentelor să acceseze beans în proces

AOP => permite separarea transversală a punctelor de interes/ preocupărilor dintr-o aplicație

- în esență AOP ajută la separarea codului care nu ține de logica de business a aplicației, cum ar fi logarea, securitate, gestionarea tranzacțiilor, de codul care implementează efectiv funcționalitățile aplicației
- acest lucru se face pentru a crește modularitatea codului și a facilita întreținerea.

@Aspect = marchează o clasă ca fiind un aspect în Spring
└ un aspect este o clasă care implementează cross-cutting concerns. + @Component
↓
sunt părți ale unui program care se bazează pe sau trebuie să afecteze multe alte părți ale sistemului

@PointCut = definește un punct de interes, de exemplu toate metodele dintr-o clasă anume sau orice metodă care începe cu "get" dintr-un pachet
↳ este practic un filtru care determină în ce puncte din execuția programului se va aplica aspectul

ex: @PointCut("within(com.app.service..*)")
public void applicationServicePointCut()

@Around = este un "advice" care înconjoară un "PointCut".
↳ codul definit aici se va executa înainte și după un punct de interes

ex: @Around("applicationServicePoint")

Alte advice: Before, After, ~~Before~~ Returning, ~~Before~~ Throwing, Around, ...

@Transactional

→ indică faptul că o metodă sau o clasă ar trebui să fie executată în cadrul unei tranzacții

→ o tranzacție este un set de operațiuni asupra bazei de date care sunt tratate ca o singură unitate: fie toate reușesc, fie, în cazul unei erori, nu reușește niciuna.

CUM FUNCȚIONEAZĂ: când o metodă este marcată astfel, Spring o să creeze un PROXY (o clasă intermediară) în jurul obiectului care conține metoda respectivă. Un PROXY este un agent care interceptează apelurile către obiectul original și poate adăuga comportament suplimentar, cum ar fi deschiderea, commit-ul sau rollback-ul unei tranzacții.

Apelarea metodei → când este creată o metodă cu @Transactional, nu apelezi direct metoda originală, este apelat PROXY-ul creat de Spring.

@SpringBootApplication - compusă din:
1) @Configuration
2) @EnableAutoConfiguration
3) @ComponentScan

1) @Configuration - indică faptul că o clasă poate fi utilizată de containerul Spring ioc ca o sursă de definiții de bean-uri
   - permite definirea metodelor anotate cu @Bean pentru a crea / configura bean-uri

2) @EnableAutoConfiguration - spune către Spring Boot să configureze automat aplicația pe baza bibliotecilor și setărilor pe care le găsește în proiect

ex: bibliotecă de baze de date => Spring Boot va configura automat conexiunea la baza de date pt noi

3) @ ComponentScan – spune Spring-ului în ce pachete să caute clasele anotate (@ Component etc ...) care trebuie gestionate de Spring (clasele sunt identificate și înregistrate automat)

@ Qualifier => când sunt create mai multe bean-uri de același tip și se dorește conectarea doar unuia dintre ele cu o proprietate se poate utiliza această anotare alături de @ Autowired

@ Autowired

@ Autowired, pentru a elimina ambiguitatea, specificând exact ce bean ar trebui conectat (wired)

@ Primary => marchează un bean ca fiind central dacă ∃ mai multi cu aceleași nume

DISPACHER SERVLET = rol principal în fluxul Spring MVC

↳ primește cererile, selectează controllerul potrivit pe baza mapării URL, invocă metoda controllerului pt. a gestiona cererea, pregătește datele și vizualizarea modelului, rezolvă vizualizarea și trimite răspunsul înapoi clientului

	STACK	HEAP
	Def: Memoria STACK este partea de memorie care a fost atribuită fiecărui program individual și e fixă	Def: Memoria HEAP este partea care nu a fost alocată programului Java, dar va fi disponibilă pt. utilizare atunci când este necesar, în timpul execuției programului.
UTILIZARE	Stochează toate variabilele primitive, apelul metodelor și referințele obiectelor din HEAP	stochează obiecte și variabile globale referite în STACK.
ALOCARE/ DEALOCARE	automată, gestionată de JVM → LIFO (last in, first out)	gestionată de Garbage Collector
EXCEPȚII	StackOverFlow error ← când se depășește limita de stack	OutOfMemory error ← când nu mai există spațiu în HEAP
ACCESIBILITATE	variabilele sunt accesibile numai în cadrul metodei sau blocului în care sunt declarate ↑ durata de viață a obiectelor este scurtă și este direct legată de blocul de cod în care sunt definite	obiectele pot fi accesate global, cât timp există referința lor

Componente JVM:

1 - ClassLoader (încarcă clasele)
2 - Memory Area (Heap, Stack)
3 - Execution Engine (interpretează și compilează bytecode-ul în timp real - JIT compiler)
4 - Garbage Collector

- dezvoltare app.

JDK - java development kit
L JRE + Java Doc, Debugger, Compiler

JRE - java runtime environment
L provides classes and libraries necesare JDK to run the program

JVM = java virtual machine = rulează aplicații JAVA
L platform independent
L gestionează resursele, inclusiv alocarea memoriei și colectarea gunoiului

Pentru că este "platform independent" execuția programului este realizată în 2 pași:

1) COMPILARE "build":

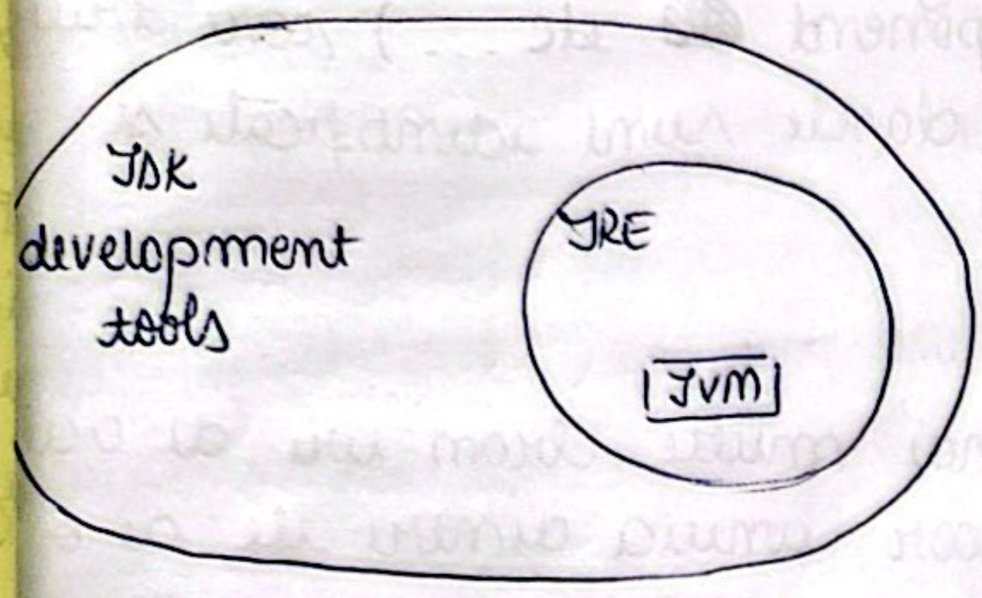
- procesul de build implică compilarea codului sursă JAVA în bytecode, care poate să fie executat de JVM; JDK folosește javac (java compiler) și rezultă .class

- în acest pas pot fi incluse și alte sarcini precum:
  - verificarea erorilor de sintaxă și compilare
  - generarea fișierelor clasă din codul sursă .java în bytecode .class
  - executarea unor taskuri automate definite în fișierul de build (maven/gradle), dependințe, etc.

2) EXECUȚIE "run"

- JRE-ul lansează JVM pt. a executa bytecode-ul care este transformat în instrucțiuni pe care sistemul de operare și hardware-ul le poate executa

- încărcarea fișierelor .class (bytecode) generate în timpul procesului de build → Class Loader (JVM) → după încărcare codul este verificat de Bytecode Verifier care se asigură că:
  - variabilele sunt inițializate înainte de folosire
  - runtime stack does not exceed
  - apelul metodelor are numărul



- dezvoltare app.

(JDK) – java development kit
└ JRE + Java Doc, Debugger, Compiler

(JRE) – java runtime environment
└ provides classes and libraries necesare JDK to run the program

JVM = java virtual machine = rulează aplicații JAVA
└ platform independent
└ gestionează resursele, inclusiv alocarea memoriei și colectarea gunoiului

Pentru că este "platform independent" execuția programului este realizată în 2 pași:

1) COMPILARE "build":

- procesul de build implică compilarea codului sursă JAVA în bytecode, care poate să fie executat de JVM; JDK folosește javac (java compiler) și rezultă .class

- în acest pas pot fi incluse și alte sarcini precum:
  - verificarea erorilor de sintaxă și compilare
  - generarea fișierelor clasă din codul sursă .java în bytecode .class
  - executarea unor taskuri automate definite în fișierul de build (maven/gradle), dependințe, etc.

2) EXECUȚIE "run"

- JRE-ul lansează JVM pt. a executa bytecode-ul care este transformat în instrucțiuni pe care sistemul de operare și hardware-ul le poate executa

- încărcarea fișierelor .class (bytecode) generate în timpul procesului de build → Class Loader (JRE) → după încărcare codul este verificat de Bytecode Verifier care se asigură că:
  - variabilele sunt inițializate înainte de folosire
  - runtime stack does not exceed
  - apelul metodelor corespund

Execuția codului începe de la metoda main.

JIT → just-in-time compiler
↳ codul încărcat bytecode este transformat în cod mașină

STATIC - este utilizat pentru a indica membrul unei clase care poate fi accesat fără a fi necesară crearea unei instanțe a clasei respective

Membrii statici:
- câmpuri
- metode
- blocuri
- clase interne

↳ sunt asociați cu clasa întâi, mai degrabă decât cu o instanță specifică a clasei

Câmpuri statice - un câmp static este cunoscut ca o variabilă de clasă, este partajat de toate instanțele clasei

ex: pt. a numărra câte instanțe au fost create

```
public static int count = 0;
```

Metode statice - sunt folosite pentru a realiza operații care nu rim / depind de starea unei instanțe specifice a clasei

cannot be @Override

ex: suma a două numere

Blocuri statice - sunt folosite pentru inițializarea membrilor statici; acest bloc este executat o singură dată, la încărcarea clasei JVM

Clase interne statice - o clasă internă statică este asociată cu o clasă exterioară și poate fi accesată fără a avea o instanță a clasei exterioare; este utilă pentru a grupa elemente care sunt folosite doar de clasa exterioară.

↳ Suprascrierea (polimorfismul dinamic) are loc în timpul rulării dar metodele statice sunt încărcate și prinse în timpul compilării în mod static

FINAL - keyword care poate să fie folosit pe:
- variabilă
- metodă
- clasă

VARIABILĂ - o variabilă finală nu își poate schimba valoarea odată ce i-a fost atribuită una;
- variabilele pot să fie
  - de instanța
  - statice
  - locale

- folosită pentru a declara constante

METODE - o metodă declarată finală nu poate fi suprascrisă de către clasele care o extind; acest lucru este util când se dorește blocarea comportamentului unei metode

CLASĂ - o clasă finală nu poate fi extinsă; acest lucru fiind util pentru securitate sau poate îmbunătăți performanța, prevenind ierarhia de moștenire.

FINALLY - este folosit cu blocul try-catch pentru a executa cod indiferent dacă o excepție a fost aruncată sau nu

- este folosit adesea pt. curățarea resurselor, închiderea conexiunilor cu baza de date
- blocul finally se execută mereu
- try-with-resources pt. gestionarea automată
- totuși este posibil să nu fie executat atunci când:
  - System.exit() a fost folosit mai sus
  - în caz de erori ca StackOverflow, MemoryAccessError

FINALIZE - metodă definită în clasa Object pe care Java o apelează pe un obiect înainte ca memoria ocupată de acesta să fie curățată de Garbage Collector
- deprecat din JAVA 9

EXCEPTII → Checked
→ Unchecked

↳ sunt folosite pt. a indica și gestiona erori în timpul execuției unui program.

Checked Exceptions: – sunt excepții pe care codul care le apelează trebuie să se ocupe explicit
– sunt verificate în timpul compilării, dacă nu sunt tratate → eroare de compilare
– tratate cu try-catch sau sunt aruncate cu throws
ex: IOException, FileNotFoundException, SQLException

Unchecked Exceptions: – includ RUNTIME exceptions și ERRORS care nu sunt verificate în timpul compilării
– pot fi prinse cu ajutorul blocului try-catch
– în categoria ERRORS: – OutOfMemory
StackOverflowException

ex: NullPointerException, NumberFormatException

try-with-resources → introdus în Java 7
↳ bloc de tip "try" care asigură închiderea automată a resurselor la finalul execuției blocurilor
– clasele implementează interfața AutoCloseable sau Closable
↓ throws Exception
↓ throws IOException

STREAM – reprezintă o secvență de elemente care suportă diferite tipuri de operații
– permite procesarea funcțională a colecțiilor

Operații → intermediare: • filter (Predicate)
(op. leneșe) • map (Function)
adică evaluarea efectivă a op. lazy este amânată până la aplicarea unei op. finale

→ finale: • collect
• forEach (Consumer)
• reduce

Fără op. finale, streamul nu va produce rezultate, nu va modifica datele și nu va declanșa nicium calcul

→ pot fi secvențiale sau paralele
↓
folosesc mai multe fire de execuție pentru a procesa elemente în paralel, cu potențialul de a îmbunătăți performanța
↑
parallelStream()
stream.parallel()

infinite streams ← limit, findFirst, findAny = pentru a controla numărul de elemente procesate

stream.builder() → interfață ce ajută la construirea de stream-uri

MAP	vs	FLAT MAP
- transformă fiecare element al stream-ului într-un alt element		- "aplatizează" mai multe stream-uri în unul singur

→ stream-urile nu modifică sursa de date, ele furnizează doar rezultatele operațiilor efectuate

→ NULL values în Java stream → dacă aplicăm funcții pe elementele unui stream acestea pot returna null, valoare ce ar trebui gestionată pt. a evita NullPointerException:
• map(Optional::ofNullable)
• filter(Objects::nonNull)

INTERFEȚE FUNCȚIONALE → interfață care conține o singură metodă abstractă, dar poate avea alte metode implicite și statice - @FunctionalInterface

ex: Runnable, Comparable, ActionListener

Există 4 tipuri de interfețe funcționale:

- CONSUMER: acceptă un singur parametru și nu returnează nimic BiCONSUMER (2 intrări)
- PREDICATE: primește un singur argument și returnează un boolean BiPREDICATE (condiție cu 2 intrări)
- FUNCTION: primește un argument și returnează o valoare BiFUNCTION (2 intrări, un rezultat)
- SUPPLIER: nu primește nimic și returnează o valoare

EXPRESII LAMBDA → o expresie lambda este o funcție anonimă, adică o funcție fără nume, care poate fi creată fără să fie nevoie să aparțină unei clase anume
→ este folosită pt. a implementa metode din interfețe funcționale

UNARY OPERATOR - formă specială de Function, acceptă un parametru și întoarce alt parametru de același tip
(1:1)

TO INT FUNCTION, TO LONG FUNCTION

EQUALS() - metodă definită în clasa Object și este folosită pt. a evalua egalitatea dintre două obiecte; în implementarea implicită acesta verifică egalitatea referințelor lor, adică două referințe sunt considerate egale dacă indică același obiect în memorie.

- pentru a ne asigura că avem o comparație semnificativă și corectă, putem suprascrie metoda

HASHCODE() – metodă definită în clasa Object și este folosită pt. a calcula un cod hash pt. obiect

– codul hash este utilizat în structuri de date bazate pe hash pt. a căuta un obiect sau a determina în ce bucket al structurii să fie stocat

! 1) Dacă 2 obiecte sunt considerate egale prin metoda equals(), atunci ele trebuie să returneze același cod hash prin metoda hashCode()

2) Dacă 2 obiecte returnează același cod hash, nu sunt neapărat egale.

STRING POOL = zonă din memoria HEAP destinată stocării Stringurilor, instanțe unice

↳ mecanismul de pooling ajută la economisirea memoriei prin reutilizarea obiectelor String existente, în loc să fie creată o nouă instanță de fiecare dată când este declarat un șir de caractere identic.

AJUNGE ÎN STRING POOL: { String s1 = "Hello";
String s2 = new String("Hello").intern();

NU AJUNGE ÎN STRING POOL: String s3 = new String("Hello");

- intern() → creează o copie exactă a unui String locat în heap și îl stochează în String Pool.

STRING ← de ce imutabil?

Pentru că obiectele de tip String sunt preponderente s-a dorit optimizarea lor de la început. String-urile sunt stocate în SPRING POOL. Se dorește scăderea nr. de obiecte SPRING temporare cu ajutorul partajării. Pentru a facilita partajarea este necesară o clasă imutabilă.

MULTITHREADING = nu este necesară sincronizarea internă dacă obiectele String sunt imutabile

↳ obiecte ce nu pot fi modificate odată create

↳ clase ce au doar get()

COLECȚII = În cazul structurilor de date HashTable și HashMap cheile sunt obiecte de tip String. Dacă obiectele nu ar fi imutabile atunci cheia se poate modifica în perioada în care rezidă în HashTable/HashMap și obiectele nu se mai pot modifica.

STRING	STRING BUILDER	STRING BUFFER
String Pool	Heap	Heap
Imutabil	Mutabil	Mutabil
lent: fiecare modificare creează un obiect nou, ceea ce necesită alocare suplimentară	cel mai rapid, ideal pt operații de modificare, adăugări sau ștergere, fără mecanisme de sincronizare	mai rapid ca String, dar mai lent ca String Builder, dar are sincronizare suplimentară în context de multithreading
thread safe	!thread safe	thread safe

• CLASĂ SINGLETON = o clasă a cărei obiecte sunt create o singură dată

Pentru a implementa o clasă Singleton:

1) Să ne asigurăm că clasa are o singură instanță

2) Oferă acces global acelei instanțe!

OOP 1) Încapsularea → restricționarea capacității de a citi și modifica datele

Variabilele de instanță sunt lăsate private și metodele de acces (getter + setter) sunt făcute publice pt. a realiza acest lucru

2) Abstractizarea = permite definirea comportamentului, dar lasă implementarea specifică subclaselor; clase abstracte + interfețe

Scanned with CamScanner

3) Moștenirea = "is-a" ("este-un"), permite unei clase copil să extindă o altă clasă părinte (/ implementează o interfață) și să-i moștenească proprietățile.

4) Polimorfism → un nume, mai multe forme
- dinamic: suprascrierea metodelor (runtime)
- static: supraîncărcarea metodelor (compile-time)

Clase abstracte: - atât metode abstracte, cât și concrete
- poate avea variabile finale, statice, non-final, non-static
- poate extinde alte clase abstracte și implementa interfețe
- poate extinde o singură clasă și multiple interfețe
- acces modif: public, protected, private
- pt. a împărtăși cod comun între clasele derivate

Interfață:
- până în JAVA 8 - doar metode abstracte
  - după JAVA 8 (inclusiv) - are și metode concrete, statice, private
- doar variabile statice și finale
- nu poate implementa clase abstracte
- poate extinde multiple interfețe
- acces modif: public
- pt. a împărtăși comportament comun

SOLID - promovează dezvoltarea codului într-un mod mai flexibil și sustenabil

- SINGLE RESPONSABILITY PRINCIPLE = o clasă trebuie să aibă doar un singur motiv să se schimbe, ceea ce înseamnă că o clasă ar trebui să aibă o singură responsabilitate

- OPEN/CLOSE PRINCIPLE = clasele/modulele/funcțiile ar trebui să fie deschise pentru extindere, dar închise pt. modificare; încurajează extinderea comportamentului

- LISKOV SUBSTITUTION PRINCIPLE = obiectele unei clase ar trebui să poată fi înlocuite cu obiecte ale subclaselor sale fără a afecta corectitudinea programului. Acest principiu este legat de design-ul robust al ierarhiilor de clase.

- INTERFACE SEGREGATION PRINCIPLE = nicio clasă nu ar trebui să fie forțată să depindă de metode pe care nu le folosește; acest principiu sugerează utilizarea interfeței specifice în locul unei interfețe generale.

- DEPENDENCY INVERSION PRINCIPLE = modulele de nivel înalt nu ar trebui să depindă de modulele de nivel jos, ambele trebuie să depindă de ceva abstract; acest principiu încurajează decuplarea comportamentelor.

DESIGN PRINCIPLES = ajută la scrierea unui cod mai eficient, ușor de înțeles și întreținut, fiind mai flexibil

- KISS (Keep it simple, stupid): codul trebuie să fie simplu de înțeles și de întreținut, astfel este simplificată și testarea

- DRY (Don't repeat yourself): codul trebuie să poată să fie reutilizabil acolo unde este necesar, iar duplicarea metodelor nu este încurajată / permisă; încurajează abstractizarea acolo unde este posibil pt. reutilizarea codului

- YAGNI (You aren't gonna need it): încurajează dezvoltatorii să lucreze la funcționalități doar atunci când este cu adevărat necesar. Se recomandă evitarea overengineering-ului.

THREADS

- permit unui program să fie mai eficient prin executarea simultană a mai multor task-uri
- permit executarea mai multor task-uri în background, fără a întrerupe firul principal de execuție

- CREAREA: utilizând clasa Thread sau implementând interfața Runnable
- CICLU DE VIAȚĂ: new, runnable, blocked, waiting, timed waiting, terminated cu priorități de la 1 la 10
- SINCRONIZAREA: pentru a evita inconsistența datelor în cazul accesului concurent, se poate folosi "synchronized" (care se asigură că metoda poate fi accesată doar de un singur thread pe când)
- COMUNICAREA ÎNTRE THREAD-URI: wait(), notify(), notifyAll()
- DEADLOCK(): apare atunci când două sau mai multe threadd-uri, așteaptă reciproc să elibereze resursele blocate, rezultând într-o așteptare infinită

- Reentrant lock = permite unui thread să reintre într-o resursă, oferind o blocare cu capacitate de reintrare; un thread blocat nu va genera un deadlock

- Volatile = se asigură că valoarea unei variabile este citită și scrisă direct în memoria principală, evitând astfel cache-urile locale ale thread-urilor
  - se asigură că modificările de pe o variabilă sunt vizibile pentru toate celelalte thread-uri

- Synchronized = este folosit pentru a controla accesul la secțiuni de cod ce nu ar trebui executate concurent de mai multe thread-uri
  - metode sincronizate
  - blocuri sincronizate

synchronized = reentrant lock, volatile, atomic variables, concurrent collections.

- RUNNABLE – are o singură metodă "run" care nu returnează valori și nu aruncă excepții
  - clasele care implementează această interfață pot fi folosite pt. a crea un thread prin pasarea unei instanțe a clasei în constructorul unui thread

- CALLABLE – are o singură metodă "call" care poate returna o valoare și arunca o excepție
  - poate fi folosit cu "ExecutorService" pt. a executa și a obține un rezultat în viitor, Future

- FUTURE = reprezintă rezultatul unei metode asincrone
  ↳ oferă metode pt. a verifica dacă operația s-a terminat, așteaptă terminarea sa pt. a întoarce rezultatul (future.get())

- EXECUTOR SERVICE – execută un nr. mare de thread-uri simultan
  - returnează rezultatul
  - executor.shutdown() ← include executor service după finalizarea tuturor sarcinilor

- SLEEP() – face ca thread-ul curent să se oprească temporar pt. o perioadă specifică de timp → în timp threadul nu consumă CPU, doar ocupă memoria și resursele asociate threadului
  - utilizată pt. a întârzia execuția unui thread, de exemplu pt. a simula întârzieri sau pt. a reduce frecvența unei operațiuni (ex: interogări repetate sau pooling)
  - nu eliberează lock-urile folosite de thread

- WAIT() – este folosită pentru a pune un thread în așteptare până când un alt thread îl notifică, folosind notify/notifyAll
  - eliberează lock-ul obiectului pe care este apelat, permițând altor thread-uri să acceseze acel obiect
  - CÂND: se folosește atunci când un thread trebuie să aștepte ca o anumită condiție să fie îndeplinită înainte de a continua execuția
  - trebuie să fie însoțit de un bloc synchronized

- NOTIFY + NOTIFYALL -> folosite pentru a notifica unul sau toate thread-urile care așteaptă pe obiectul respectiv
  - CÂND: după o condiție pe care threadul sau threadurile care așteptau s-a schimbat și trebuie să continue execuția
  - trebuie să fie înconjurat de un bloc synchronized

- AWAIT() - asociată cu un mecanism de sincronizare, cum ar fi "Condition" în Java.
  - este folosită în contextul lock-urilor pt. a face ca un thread să aștepte până când o anumită condiție devine adevărată, fără a consuma resurse ca CPU/procesor
  - un thread poate fi "trezit" de un alt thread care apelează "signal()" sau "signalAll()" pe același obiect "Condition"

- Lock() - pe un thread este un mecanism care împiedică accesul simultan la o resursă partajată (orice obiect sau variabilă care poate fi modificată de mai multe threaduri în același timp).

RABBITMQ

- Coadă de mesaje = o coadă este un canal de comunicație point-to-point în care mesajele transmise de producători sunt consumate de un singur consumator.
  - Dacă mai multi consumatori ascultă aceeași coadă, mesajele sunt distribuite în mod egal (sau pe baza unei anumite logici de scheduling) între consumatori, dar fiecare mesaj este procesat o singură dată de un singur consumer.

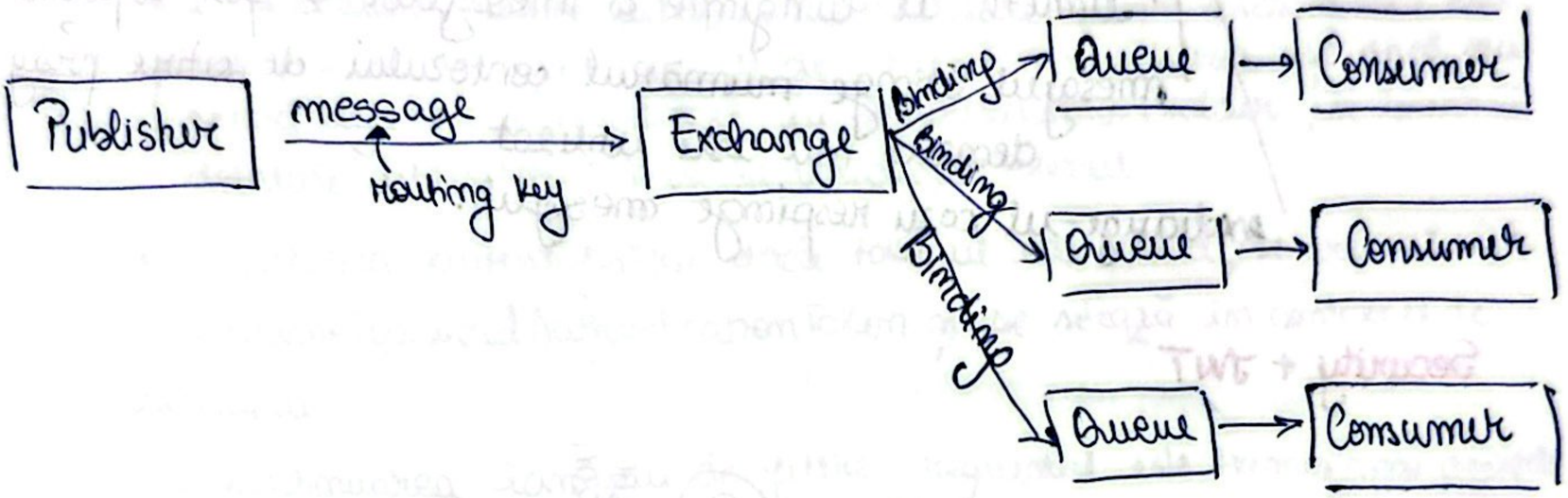
- Topic de mesaje = un canal de comunicare publish-subscribe, în care mesajele trimise către un topic sunt distribuite tuturor abonaților acestora. Fiecare abonat primește o copie a unui mesaj, permițând procesarea simultană a aceluiași mesaj de către mai mulți consumatori.

- Binding = "punte" ce creează conexiunea între o coadă și un exchange.

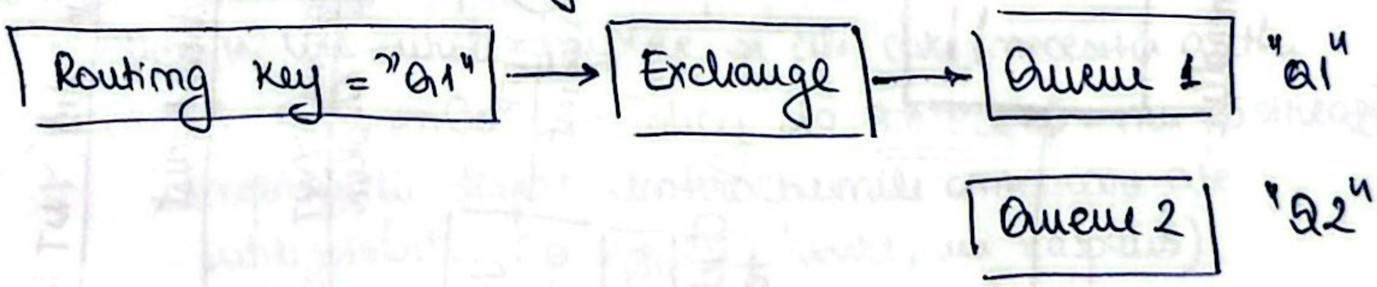
- Routing Key = atributul mesajului pe care exchange-ul îl verifică în timp ce determină cum direcționează mesajele către cozi

  RabbitMQ → payload + routing key
  ↳ folosită pt. a discerne payload-ul trimis de către sistem

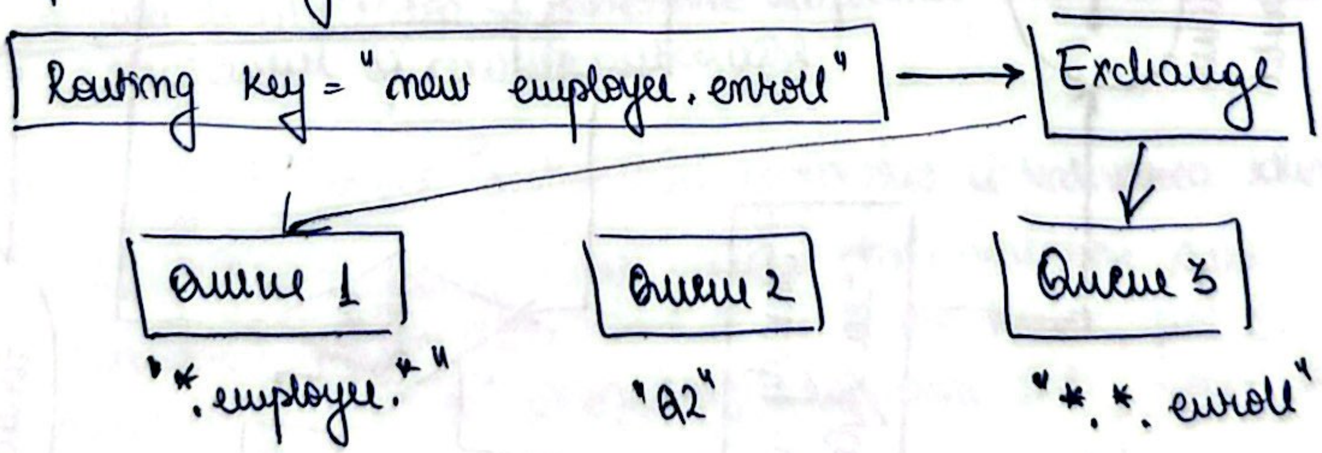
- Exchange - mesajele nu sunt postate direct pe cozi
  - user-ul trimite mai întâi mesajul către un exchange
  - reprezintă responsabilul pt. rutarea mesajelor către diferite cozi



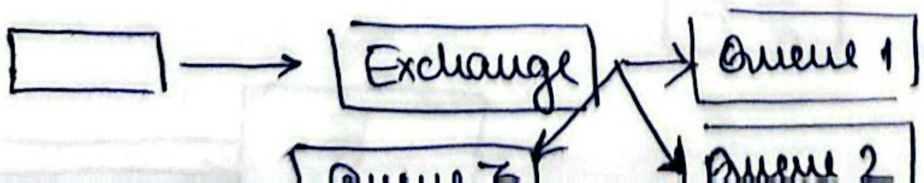
Exchanges: 1) Direct exchanges



2) Topic exchange

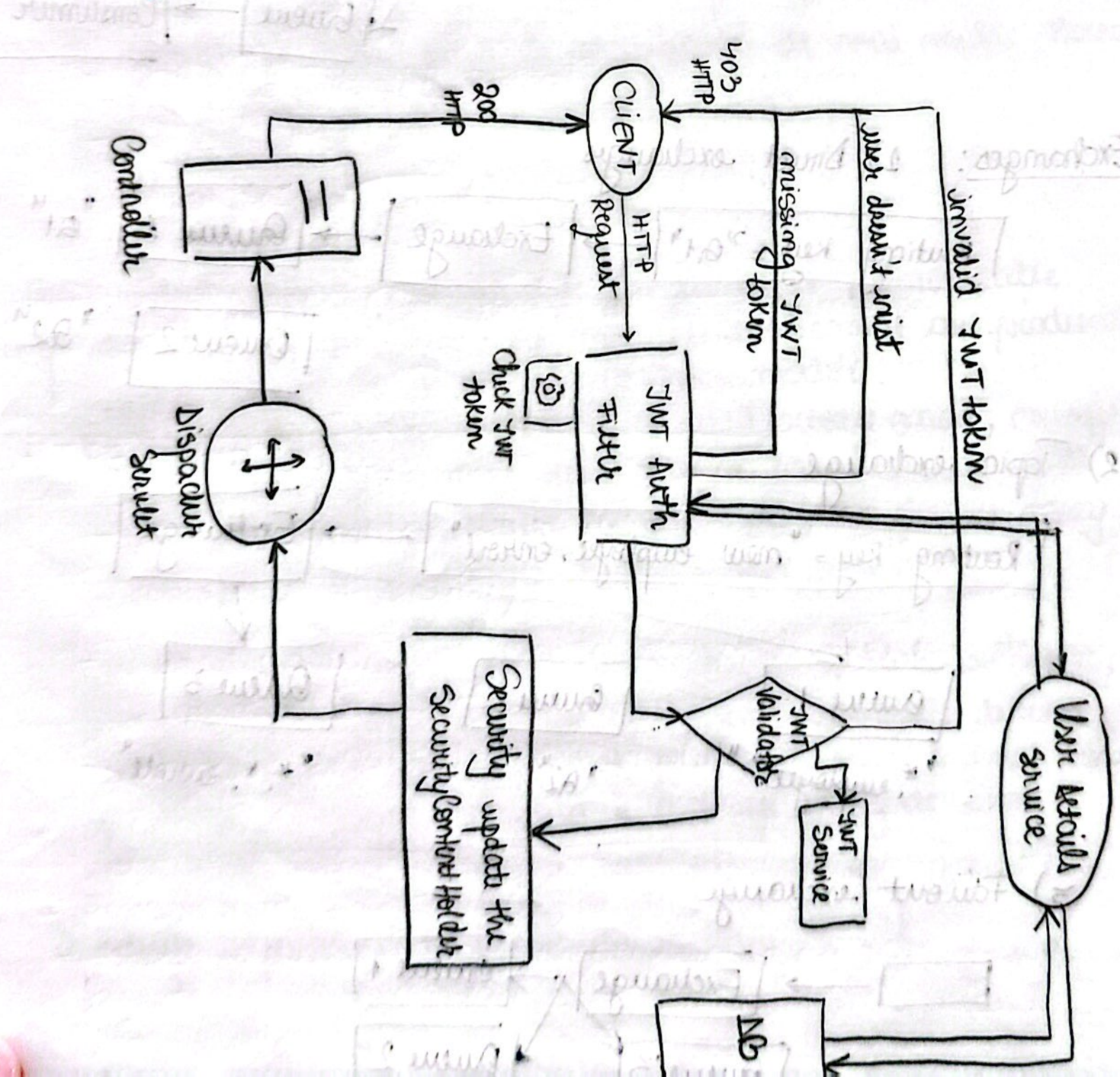


3) Fanout exchange



- Dead Letter Queue = coada de mesaje nelivrate
  - ↳ coada de reținere a mesajelor care nu pot fi transmise către destinații din anumite motive
  - ↳ în RabbitMQ, este implementarea serviciului pentru stocarea mesajelor care îndeplinesc 1 sau mai multe dintre criteriile de eșec de mai jos:
    - limita de lungime a cozii a fost depășită
    - mesaj trimis la o coadă indisponibilă
    - limita de lungime a mesajelor a fost depășită
    - mesajul atinge numărul contorului de citire prag, deoarece nu este utilizat
    - exchange-ul cozii respinge mesajul.

Security + JWT



1) Utilizatorul trimite un request HTTP la server → fără JWT (Json Web Token)
→ cu tokenul JWT în headerul de autentificare

2) Clasa JWT Authentication Filter este responsabilă pt. interceptarea fiecărui request și verificarea unui token JWT.

2.1. verificare JWT: dacă headerul de autentificare lipsește sau nu începe cu "Bearer ", requestul este trimis mai departe fără autentificare

2.2. extracție nume utilizator + validare token dacă acesta este prezent – se extrage numele de utilizator din token și dacă nu există deja o autentificare în SecurityContextHolder, se încarcă detaliile utilizatorului și se validează tokenul

2.3. setarea autentificării dacă tokenul este valid, se creează un UserNamePasswordAuthenticationToken și se setează în contextul de securitate

2.4. continuarea lanțului de filtre: requestul este trimis mai departe

3) Clasa SecurityConfig definește lanțul de securitate și politici de acces:
- definește URL-urile publice și cele care necesită auth
- setează SessionCreationPolicy la STATELESS = nu păstrează informații despre interacțiunile anterioare ale utilizatorului (o singură cerere, un răspuns)

STATEFUL = serverul păstrează evidența stării fiecărei sesiuni de utilizator și menține informații despre interacțiunile utilizatorului și cererile anterioare

4) Clasa JWT Service gestionează generarea și validarea tokenului
- Claims (revendicări) sunt perechi cheie-valoare care reprezintă informațiile stocate într-un token, pot fi: subject (nume de utilizator), expiration date, roles, etc...
- este obținută cheia de semnare pt. token-uri

· Cum funcționează cheia de semnare?

1) Crearea unui token JWT

Când un server creează un token JWT, acesta semnează tokenul folosind o cheie secretă. Acest lucru implică aplicarea unui algoritm de semnare (HMAC SHA-256) asupra conținutului token-ului și a cheii secrete.

Token rezultat conține 3 părți
- header
- claims
- semnătura = generată folosind headerul, claims și cheia de semnare

2) Validarea unui token

Când un server primește un token JWT, acesta verifică integritatea și autenticitatea token-ului recalculând semnătura folosind aceeași cheie secretă și comparând-o cu semnătura inclusă în token, dacă coincid, tokenul este considerat valid.

HTTP (Hypertext Transfer Protocol)
- protocolul pt. transferul de date web
- securitate: informațiile sunt transmise în clar, fără criptare, ceea ce le face vulnerabile la interceptări
- utilizare: folosit pt. transferul de date nesensibile

HTTPS (HTTP Secure)
- HTTP combinat cu securitate prin SSL/TLS (secure socket layer transport layer security)
- securitate: datele sunt criptate ceea ce le protejează de interceptări și manipulări
- utilizare: folosit pt. datele sensibile

RESTful Web Services = servicii ce urmăresc arhitectura REST (Representation State Transfer)
- folosește protocolul HTTP; oferă mijloace de accesare a resurselor puse la server necesare clientului prin intermediul browserului web prin intermediul: request header, request body,

HTTP methods: GET, POST, PUT, DELETE, PATCH, OPTIONS
↓
descrie opțiunile de comunicare pt. sursa țintă

GET + OPTIONS = safe + idempotent methods
PUT + DELETE = idempotent methods
POST + PATCH ≠ safe + idempotent methods

GET = folosită pt. a citi / prelua date de pe un server web
↳ returnează status 200 OK dacă datele sunt preluate cu succes de pe server

POST = utilizată pt. a trimite datele (fișier, date de formular, etc.) catre server (201 OK)

PUT = folosită pentru a modifica datele de pe server
↳ înlocuiește întregul conținut dintr-o anumită locație cu date care sunt transmise în body payload
↳ dacă nu sunt resurse care să corespundă cererii, acesta va genera una

PATCH = similar cu PUT, singura diferență este că modifică doar o parte a datelor
↳ acesta va înlocui doar conținutul pe care doriți să îl actualizați

DELETE = folosită pt. a șterge datele de pe server într-o locație specificată

Idempotent

O metodă HTTP este idempotentă dacă efectul dorit asupra serverului de a face o singură cerere este același cu efectul de a face mai multe solicitări identice.

APACHE TOMCAT = server web open-source [servlet container]

└ Spring Boot configurează automat Tomcat ca server implicit

Tomcat integrat ca server incorporat => se pot crea aplicații web care funcționează independent, fără să fie nevoie de instalarea unui server

Tomcat extern -> aplicația poate fi rulată direct cu un singur fișier JAR/WAR

JAR/WAR = artefacte principale

JAR = pachet Java standard, care conține toate clasele, fișierele de resurse și biblioteci necesare pt. a rula aplicația
- în Spring Boot, JAR-ul conține inclusiv un server web incorporat (de obicei TOMCAT), ceea ce permite aplicației să fie rulată direct din linia de comandă, fără să fie necesară instalarea unui server web

WAR = pachet Java folosit în mod tradițional pt. a implementa aplicații web pe un server de aplicații extern
- conține toate fișierele și resursele necesare pt. a rula o aplicație web, dar nu include serverul web în sine

JAR	WAR	
incorporat	se bazează pe un server extern	server web
java - jar	implementat pe un server web extern	pornire app
simplu, rapid de configurat	mai complex	complexitate
microservicii, app standalone	medii enterprise, servere dedicate	utilizare
portabil, rulează oriunde	mai puțină portabilitate, necesită server compatibil	flexibilitate
configurație limitată (server incorporat)	configurare completă asupra serverului	control asupra serverului

pom.xml (Project Object Model)

- fișier esențial folosit în proiectele Java care utilizează MAVEN ca sistem de build și gestionare a dependințelor.
- conține configurații necesare pt. a construi și gestiona proiectul
- fișier XML în care sunt specificate informații ca:
  - dependințe externe
  - versiunea java
  - plugin-uri de build etc.

SECȚIUNI:
1) Informații despre proiect
   - groupId (grupul/organizația)
   - artifactId (Numele proiectului)
   - version (versiunea proiectului)
2) Dependințe ← biblioteci externe pe care proiectul le folosește
3) Plugin Maven - extind funcționalitatea Maven
4) Profiluri de build → care activează configurări diferite în funcție de mediu (dev, test, prod)

INDEX = structură de date care îmbunătățește viteza operațiilor de căutare pe o tabelă, dar poate afecta negativ viteza op. de ștergere adăugare, actualizare, deoarece indexul se modifică de fiecare dată când se modifică datele

- scopul principal al unui index este de a optimiza interogările care accesează o fracțiune semnificativă de rânduri a unei tabele, fără a fi nevoie să scaneze întreaga tabelă

SCOP:
- îmbunătățește performanța
- unicitate: indexul poate să fie utilizat pt. a impune unicitatea valorilor dintr-o coloană

TIPURI:
- INDEXURI PRIMARE = impune unicitate, unul sg/tabelă
- INDEXURI SECUNDARE - nu impune unicitate, pot fi mai mulți pe o tabelă

Scanned with CamScanner

SQL

SQL pattern: LIKE, NOT LIKE

- 'K%' ← starts with K
- '__K%' ← a 3a pozitie e K

STORED PROCEDURE = subrutină disponibilă aplicațiilor care accesează un sistem de gestionare a' bazelor de de date relaționale

- stocate în dicționarul de date al bazei de date
- nu poate fi executată decât în DB + ocupă multă memorie
- oferă securitate și funcționalitate

SINTAXA:

```
CREATE PROCEDURE procedure_name
AS
sql_statement
GO;
```

```
EXEC procedure_name; // execute a Store Procedure
```

Store Procedure with one Parameter

```
CREATE PROCEDURE SelectAllCostumers
@City nvarchar(30)
AS
SELECT * FROM Costumers WHERE City = @City
GO;
```

```
EXEC SelectAllCostumers @City = 'London';
```

Scanned with CamScanner

Avantaje Store Procedure:

1) Reutilizarea: mai mulți utilizatori / aplicații pot folosi și reutiliza cu ușurință procedurile stocate, doar apelând-o.

2) User de modificat: se pot schimba rapid instrucțiunile dintr-o "store procedure", după cum se dorește, cu ajutorul comenzii ALTER TABLE.

3) Securitate: acestă procedură permit sporirea securității unei aplicații sau a unei baze de date, restricționând accesul direct la tabel.

4) Trafic de rețea scăzut: serverul transmite doar numele procedurii nu și întreaga interogare

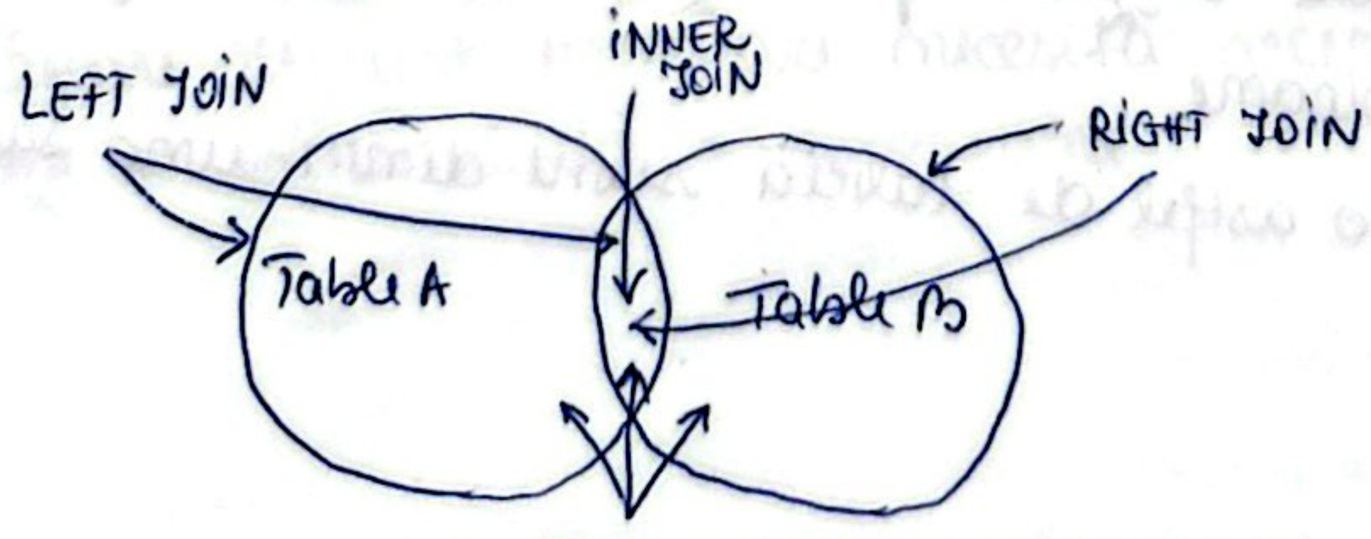
5) Crește performanța: La prima utilizare, un plan pt. "store procedure" este creat și stocat în pool-ul de buffer pt. o execuție rapidă pt. data următoare.

UNIQUE CONSTRAINT = se asigură de faptul că toate valorile dintr-o coloană sunt unice
↳ spre deosebire de PRIMARY KEY, pot exista mai multe astfel de constrângeri într-o tabelă

CROSS JOIN = produsul cartezian al celor 2 tabele incluse în JOIN

CROSS JOIN + WHERE => INNER JOIN

SELF JOIN = regular JOIN unde o tabelă face JOIN cu ea însăși bazându-se pe relația dintre coloanele sale.



FOREIGN KEY = cuprinde un singur câmp sau o colecție de câmpuri dintr-un tabel care se referă în esență la cheia primară [PRIMARY KEY] dintr-un alt tabel
→ asigură integritatea referențială în relația dintre 2 tabele

PRIMARY KEY = identifică în mod unic fiecare rând dintr-un tabel
→ trebuie să conțină valori UNIQUE + NOT NULL
o tabelă în SQL este limitată să aibă una și o singură cheie primară, care este compusă din câmpuri (coloane) unice sau multiple.

SELECT query în SQL:

→ WHERE = folosită pt. a filtra intrările bazându-se pe o condiție specifică

→ ORDER BY = folosită pt. a ordona intrările bazându-se pe niște field-uri (ASC/DESC)

→ GROUP BY = folosită pt. a grupa intrările cu date identice și poate fi utilizată cu unele funcții de agregare pt. a produce rezultate sumarizate din DB

→ HAVING = folosită pt. a filtra intrările în combinație cu clauza GROUP BY.
→ diferită de WHERE, deoarece clauza WHERE nu poate filtra intrările agregate

VIEW = o tabelă virtuală bazată pe rezultatul unei op. SQL
- conține linii și coloane
- câmpurile dintr-o astfel de tabelă sunt dintr-una sau mai multe tabele

Scanned with CamScanner

DELETE = folosit pt. a șterge intrări (linii)

TRUNCATE = folosit pt. a șterge toate intrările dintr-o tabelă + free space

DROP = folosit pentru a șterge un obiect din DB

AGGREGATE FUNCTIONS = AVG(), COUNT(), MIN(), MAX(), SUM(), FIRST(), LAST()

returnă a single value

→ + GROUP BY + HAVING clauses

SCALAR FUNCTIONS = LEN(), UCASE(), LCASE(), MID(), CONCAT(), RAND(), ROUND(), NOW(), FORMAT()

→ returnează o singură valoare bazându-se pe o val. de input

CONNECTION POOL (ex: HikariCP)

- tehnică utilizată pt. a gestiona eficient conexiunile către o bază de date sau o altă resursă de rețea
- în loc să includ și să deschid o nouă conexiune de fiecare dată când ai nevoie să accesezi baza de date, un pool de conexiuni păstrează un set de conexiuni deschise care pot fi reutilizate.

Avantaje:
- Performanță îmbunătățită
  - Reducerea timpului de inițializare (crearea unei conexiuni cu DB = proces costisitor)
  - Utilizarea eficientă a resurselor
- Scalabilitate
- Stabilitate - rezistența la erori
- Gestionarea eficientă a resurselor

Fiecare request HTTP care necesită acces la DB ar trebui să creeze și să includă o conexiune.

TRANSACTIONS ÎN ORACLE

Oracle = transactions oriented → utilizează tranzacții pt. a asigura integritatea datelor

<u>Transaction</u> = serie de una sau mai multe instrucțiuni SQL legate logic ce sunt definite pt. a îndeplini o anumită sarcină.

Oracle tratează seria de instrucțiuni SQL ca o unitate, aî. toate modificările aduse de instrucțiuni sunt fie comise (devin permanente), fie anulate în același timp. Dacă aplicația eșuează în mijlocul unei tranzacții, baza de date este restaurată la starea anterioară.

1. Inițierea tranzacțiilor → începe implicit atunci când este efectuată prima op. SQL care modifică datele

2. COMMIT → comandă care finalizează o tranzacție, făcând permanente toate modificările făcute în DB în timpul tranzacției respective.

3. ROLLBACK → anulează toate modificările efectuate de tranzacție, revenind baza de date la starea sa anterioară. Este util în cazul în care apar erori sau când o tranzacție nu mai poate fi finalizată cu succes.

4. SAVEPOINT → permite crearea de puncte intermediare în cadrul unei tranzacții, la care te poți întoarce dacă este nevoie. În cazul ROLLBACK, poți reveni până la un anumit punct fără a anula întreaga tranzacție.

Scanned with CamScanner

ORACLE sequence = obiect de bază de date care generează o secvență unică de numere, care pot fi utilizate pt. a crea valori unice

→ se folosește când se dorește:
- generarea cheilor primare
- valori unice pt. alte scopuri
- optimizarea performanței
- control asupra identificatorilor

SQL injection

- vulnerabilitate de securitate care apare atunci când un atacator poate introduce cod SQL rău intenționat într-o interogare SQL validă, prin intermediul unui câmp de intrare dintr-o aplicație (ex: formular web) => acest lucru poate permite atacatorului să manipuleze interogări SQL pt. a obține acces neautorizat la date, a modifica sau a șterge date din DB, sau chiar să preia controlul asupra serverului.

- prevenire:
  1) Utilizarea interogărilor parametrizate
  2) Validarea și filtrarea inputului
  3) Utilizarea ORM (Object-Relational Mapping)
  4) Escaparea corectă a caracterelor speciale
  5) Limitarea privilegiilor în DB
  6) Dezactivarea mesajelor detaliate de eroare în aplicațiile live
  7) Utilizarea controalelor de acces.

ACID = atomicitate, consistență, izolare, durabilitate

Dacă o operațiune de bază de date are aceste proprietăți ACID, atunci poate fi numită o tranzacție ACID, iar sistemele de stocare a datelor care aplică aceste operațiuni sunt numite sisteme tranzacționale.

- Atomicitate = tranzacția se execută complet sau deloc
- Consistență = baza de date trb să fie consistentă atât înainte, cât și după executarea tranzacției
- Izolare = mai multe tranzacții se pot executa simultan fără a interfera
- Durabilitate = modificările aduse de o tranz. cu succes se propagă și în

ARHITECTURA DE MICROSERVICII

- microserviciile se referă la un stil arhitectural pt. dezvoltarea aplicațiilor
- permit ca o aplicație mare să fie separată în părți identice mai mici, fiecare parte' având propriul său domeniu de responsabilitate
- pt. a servi o solicitare a unui utilizator, o aplicație bazată pe microservicii poate apela la mai multe microservicii interne pt. a-și compune răspunsul.
- sunt folosite pt. a accelera dezvoltarea aplicațiilor

AVANTAJE: - scalabilă și flexibilă
- fiecare microserviciu poate fi scalat independent în funcție de cerințele specifice, fără a afecta întreaga aplicație
- datorită izolării microserviciilor, aplicația prezintă <u>reziliență</u> și astfel nu ar trebui ca întreaga aplicație să fie afectată în caz de eșec al unor microservicii

REZILIENȚĂ - în microservicii implică funcționalități cheie, cum ar fi <u>întrerupătoarele de circuit</u> - timeout, failover

previn într-un ecosistem de microservicii, eșecurile de serviciu în cascadă.

Timeout = împiedică sistemul să rămână blocat în operațiuni care au puține șanse de reușită

Funcțiile failover = permit redirectionarea automată a cererilor de la un serviciu defect la unul funcțional

combinând aceste strategii, ne putem menține reziliența aplicației, putem gestiona și ne putem recupera cu eleganță

SPRING BOOT VS STANDARD JAVA

1) Configurare automată

- Spring Boot automatizează configurarea necesară pentru a porni și a rula o aplicație, aî. programatorul nu trebuie să scrie sau să configureze manual componente ca: server web, conexiunile cu baza de date, securitate.
- Într-o aplicație JAVA Standard, configurarea acestor componente necesită o cantitate semnificativă de cod și configurare manuală.

2) Server încorporat

- Spring Boot vine cu server web încorporat, precum Tomcat, Jetty sau Undertow. Aceasta înseamnă că nu este nevoie de configurare manuală a unui server extern sau crearea unui WAR pt. a-l distribui pe un server de aplicații. Astfel, aplicația se poate rula direct dintr-un JAR.
- Într-o aplicație JAVA Standard, este nevoie de un server extern sau trebuie configurat un server de aplicații pt. a rula aplicația, ceea ce crește complexitatea implementării.

3) Simplificarea dezvoltării

- Spring Boot tools, eliminarea config XML, folosirea anotării, focus pe logica de business
- În JAVA Standard, este nevoie de mai mult cod pt. a gestiona aspecte precum ciclul de viață al aplicației, conexiunile cu BD, etc.

4) DIP, IOC

- Spring Boot implementează IOC+DI prin intermediul containerului Spring ⇒ gestionarea dependințelor între clase - mult mai ușoară și mai modulară
- În JAVA Standard, dependințele trebuie gestionate manual sau folosite alte biblioteci ⇒ complexitate mai mare

Scanned with CamScanner

5) Securitate integrată

• Spring Boot include integrare facilă cu Spring Security, care permite adăugarea rapidă a autentificării și autorizării în aplicație.

• În JAVA Standard, securitatea trebuie gestionată prin configurare și implementare manuală, ceea ce poate crește riscul de erori și vulnerabilități.

6) Monitorizare și observabilitate

• Spring Boot Actuator oferă capacități de monitorizare integrate, oferind informații despre performanța aplicației, starea sa, metrici, health checks și endpoint-uri de monitorizare

• În JAVA Standard, aceste capabilități trebuie implementate manual sau cu ajutorul unor librării externe => complexitate

7) Microservicii și Scalabilitate

• Spring Boot este adesea folosit în dezvoltarea arhitecturii de microservicii, datorită configurării simple și independente.

• JAVA Standard nu este conceput pt. microservicii din start și necesită mai multe eforturi pt. a structura aplicația a.î. să fie scalabilă în mod eficient.

8) Testare simplificată

• Spring Boot oferă un set logat de instrumente pt. UNIT TESTS, INTEGRATION TESTS și AUTOMATION TESTS. De exemplu, este ușor de scris test pt. componentele Spring, datorită Spring Test Framework.

• În JAVA Standard, configurarea și testarea necesită mult mai mult cod pt. a seta contextul aplicației și dependințele necesare.

Scanned with CamScanner

CLASĂ IMUTABILA ÎN JAVA:

1) Declarăm clasa final
2) Toate fields trebuie să fie private
3) No setters!
4) Toate fieldurile mutable să fie final ai. valoarea acestora să fie asignată o singură dată
5) Inițializăm toate field-urile folosind un constructor cu deep-copy
6) Cloning the objects in getters.

COPY CONSTRUCTOR = constructor folosit când dorim să inițializăm valoarea noului obiect cu vechiul obiect al aceleiași clase.

JAVA 17 features

1) Text blocks ("""") permite definirea șirurilor multi-linie într-un mod mai simplu și mai lizibil

'\s' – pentru adăugarea de spații

'\' – removing newline

2) Switch expressions (îmbunătățiri)

```
String result = switch(day) {
        case MONDAY -> "Start of the week";
        case FRIDAY -> "End of the week";
        default -> "Mid week";
    };
```

String text = ... case APPLE, PEAR -> { sout("...");
yield "Common fruit"; }

cuvânt cheie dependent de context ce desemnează valoarea

3) record keyword → în loc să folosim cuvântul "class" când definim o clasă, putem defini folosi record

→ modalitate concisă de a defini clase imutabile cu constructori, metodele equals(), hashCode(), toString() generate automat

→ menite să înlocuiască DTOs

4) 'sealed' classes (clase sigilate) = permit controlul ierarhiilor de moștenire; specifică exact ce alte clase pot extinde acea clasă

```
public abstract sealed class Fruit permits Apple, Pear {}
public non-sealed class Apple extends Fruit {}
public final class Pear extends Fruit {}
```

orice clasă ce extinde o 'sealed' clasă trebuie să fie ori "final" or "non-sealed".

5) Pattern Matching for 'instanceof' → permite o verificare mai simplă și mai concisă a tipurilor de obiecte

```
Object o = new Grape(Color.BLUE, 2);
if(o instanceof Grape) {
    Grape grape = (Grape) o;
    sout(grape.getPits());
};
```

JAVA 11

```
Object o = new Grape(
        Color.BLUE, 2);
if (o instanceof Grape grape
    sout(grape.getPits());
}
```

JAVA 17

→ putem împerechea 'instanceof' check cu && (and), dar nu cu (

(grape) → există și în afara if-ului!

6) Helpful NullPointerException → ne spune exact metoda la care este aruncată excepția

DESIGN PATTERNS

– este o soluție generică repetabilă la o problemă care apare frecvent în proiectarea software, care este utilizată în ingineria software. Nu este un design complet ce poate fi scris în cod imediat. Este o descriere sau un model pentru rezolvarea problemelor care poate fi aplicat într-o varietate de contexte.

TIPURI de Design Patterns în Java:

1) Creational Design Patterns (CDP)
   - Singleton
   - Factory Method
   - Abstract Factory
   - Builder
   - Prototype

2) Structural Design Patterns (SDP)
   - Adapter
   - Bridge
   - Composite
   - Decorator
   - Facade
   - Proxy
   - Flyweight

3) Behavioral Design Patterns (BDP)
   - Command
   - Iterator
   - Mediator
   - Memento
   - Observer
   - State
   - Strategy
   - Template
   - Visitor
   - Interpreter

Scanned with CamScanner

1) CDP – se ocupă de procesul de creare al obiectelor, încercând să-l facă mai flexibil și mai eficient
– acesta face sistemul independent și modul în care obiectele sale sunt create, compuse și reprezentate

Singleton = asigură că o clasă are o singură instanță și oferă un punct de acces global la acea instanță

UTILIZARE: adesea folosită pt. clase care gestionează conexiuni la baza de date, configurații, jurnale, etc

ex:
```
public class Singleton {
    private static Singleton instance;
    private Singleton () { }
    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

Factory Method = oferă o metodă pt. a crea obiecte, permițând subclaselor să decidă instanțierea obiectului.

UTILIZARE: folosit pt. crearea de obiecte fără a specifica clasa exactă care va fi instanțiată

ex:
```
public interface Animal { void speak(); }
public class Dog implements Animal { public void speak(sout("W
public class Cat implements Animal { public void speak(sout("Meow!"));
public class AnimalFactory {
    public static Animal createAnimal (String type) {
        if (type.equals("dog")) return new Dog();
        else if (type.equals("cat")) return new Cat();
        return null;
```

Builder = separă construcția unui obiect complex de reprezentarea sa, permițând construirea pas cu pas

UTILIZARE: util pt. crearea de obiecte cu multe atribute opționale

ex:
```
public class User {
    private String firstName;
    private String lastName;
    private int age;

    private User(UserBuilder builder) {
        this.firstName = builder.firstName;
        this.lastName = builder.lastName;
        this.age = builder.age;
    }

    public static class UserBuilder {
        private String firstName;
        private String lastName;
        private int age;

        public UserBuilder setFirstName(String firstName) {
            this.firstName = firstName;
            return this;
        }
        public UserBuilder setLastName() setAge(...) {...}

        public User build() {
            return new User(this);
        }
    }
}
```

2) SDP - se concentrează pe compoziția claselor sau a obiectelor pt. a forma structuri mai mari și mai complexe
- ele ajută la organizarea și gestionarea relațiilor dintre obiecte pt. a obține o mai mare flexibilitate, reutilizare și mentenanță într-un sistem software.

Decorator - permite adăugarea dinamică de funcționalități noi la un obiect, fără a modifica structura sa de bază

UTILIZARE: util pt. extinderea funcționalității unor obiecte într-un mod flexibil (de ex, pt. adăugarea de caracteristici suplimentare la interfețe grafice sau fluxuri de date

ex:
```
public interface Shape { void draw(); }
public class Circle implements Shape {
    public void draw() { sout("Drawing Circle"); }
}

public class ShapeDecorator implements Shape {
    protected Shape decoratedShape;
    public ShapeDecorator(Shape decoratedShape) {
        this.decoratedShape = decoratedShape;
    }
    public void draw() {
        decoratedShape.draw();
    }
}
public class RedBorderDecorator extends ShapeDecorator {
    public RedBorderDecorator(Shape decoratedShape) {
        super(decoratedShape);
    }
    public void draw() {
        super.draw();
        sout("Adding Red Border");
    }
}
```

Adapter = permite ca două clase incompatibile să lucreze împreună, prin conversia interfeței unuia dintre ele într-o interfață compatibilă

UTILIZARE: utilizat atunci când trebuie să faci o clasă existentă să lucreze cu o altă clasă care are o interfață incompatibilă.

ex:
```
public interface MediaPlayer {
    void play(String audioType, String fileName);
}

public class MP3Player implements MediaPlayer {
    public void play(String audioType, String
                                    fileName) {
        sout("Playing mp3 " + fileName);
    }
}

public class AudioAdapter implements MediaPlayer {
    MediaPlayer advancedMusicPlayer;

    public AudioAdapter(String audioType) {
        if (audioType.equalsIgnoreCase("mp4")) {
            advancedMusicPlayer = new MP4Player();
        }
    }

    public void play(String audioType,
                     String fileName) {
        if (audioType.equalsIgnoreCase("mp4")) {
            advancedMusicPlayer.play("mp4",
                                     fileName);
        }
    }
}
```

Facade – oferă o interfață simplificată pt. un subsistem complex, permițând utilizatorilor să interacționeze cu subsistemul printr-o singură clasă.

UTILIZARE: utilizat pentru a ascunde complexitatea unui sistem mare prin intermediul unei interfețe simplificate

ex:
```
public class ComputerFacade {
    private CPU cpu;
    private Memory memory;
    private HardDrive hardDrive;

    public ComputerFacade() {
        this.cpu = new CPU();
        this.memory = new Memory();
        this.hardDrive = new HardDrive();
    }

    public void start() {
        cpu.start();
        memory.load();
        hardDrive.read();
    }
}
```

3) BDP – se ocupă de comunicarea și interacțiunea dintre obiecte și clase
– se concentrează pe modul în care obiectele și clasele colaborează și comunică pt. a îndeplini sarcini și responsabilități.

Observer = definește o relație între obiecte a.î. atunci când un obiect își schimbă starea, toate obiectele dependente sunt notificate automat

UTILIZARE: frecvent utilizat în implementarea evenimentelor sau sistemelor de notificări (ex

ex: public interface Observer {
void update (String message);
}

public class Subscriber implements Observer {
public void update (String message) {
sout ("Received update:" + message);
}
}

public class Publisher {
private List <Observer> observers = new ArrayList <>();
public void addObserver (Observer observer) {
observers.add(observer);
}
public void notifyObservers (String message) {
for (Observer observer : observers) {
observer.update (message);
}
}
}

Strategy = definirea unei familii de algoritmi, încapsularia fiecăruia și făcându-i interschimbabili. Clientul poate alege dinamica strategia care va fi aplicată.

UTILIZARE: folosit atunci când mai multe metode sunt disponibile pt. a realiza o sarcină, iar alegerea metodei este făcută la runtime.

ex: public interface PaymentStrategy {
void pay(int amount);
}

public class CreditCardStrategy implements PaymentStrategy {
public void pay (int amount) {
sout("Paid with credit card:" + amount);
}
}

```
public class PayPalStrategy implements PaymentStrategy {
    public void pay (int amount) {
        sout ("Paid with PayPal: " + amount);
    }
}

public class ShoppingCart {
    private PaymentStrategy paymentStrategy;
    public void setPaymentStrategy (PaymentStrategy ps) {
        paymentStrategy = ps;
    }
    public void checkout (int amount) {
        paymentStrategy. pay (amount);
    }
}
```

JPA VS HIBERNATE

JPA = Java Persistance Api = folosită pentru a examina, controla și persista date între obiectele Java și bazele de date relaționale.

→ considerată legătura între un model orientat pe obiect și un sistem de baze de date relaționale.

- prin urmare, pentru persistența datelor, instrumentele ORM precum Hibernate implementează specificațiile JPA

JPA: - este doar o specificație, nu este o implementare
- set de reguli și linii directoare pt. a seta interfețe pentru implementarea mapării obiect-relaționale.
- are nevoie de câteva clase și interfețe
- susține polimorfismul și moștenirea

Hibernate = [illegible] JAVA, open source, [illegible] și ORM (Object Relational Mapping) instrument pt. limbajul JAVA, care simplifică construirea aplicației Java pt. a interacționa cu baza de date.

JPA	Hibernate
javax.persistence package	org.hibernate package
Descrie manipularea datelor relaționale în app. JAVA	instrument ORM care este utilizat pt. a salva obiectele JAVA în sistemul de baze de date relaționale.
Nu este o implementare! Este doar o specificație JAVA.	implementare JPA
API standard care permite efectuarea op. de bază de date	utilizat în maparea tipurilor de date JAVA cu tipuri de date SQL și tabele de baze de date
Ca limbaj de interogare orientat pe obiecte, folosește Java Persistence Query Language (JPQL) pt. a executa op. DB	Ca limbaj de interogare orientat pe obiecte, folosește Hibernate Query Language (HQL) pt. a executa op. DB
folosește EntityManagerFactory	folosește SessionFactory
pentru a face, citi, elimina acțiuni pt. instanțe ale claselor de entități mapate, folosește interfața EntityManager	—||— interfața Session

(Plain old Java object)

Anotări: @Entity = folosită pt. declararea oricărui POJO ca o entitate pt. baza de date

@Table = folosită pt. a schimba detaliile tabelei, atribute ca: name (override the table name), schema, catalogue, enforce unique constraints

@Id = folosită pt. declararea cheii primare în interiorul POJO-ului nostru

@ GeneratedValue = Hibernate generează automat valorile cu referire la secvența internă și nu trebuie să setăm valorile manual

@ Column = folosit pt. a specifica mapările coloanelor. Înseamnă că dacă în cazul în care nu avem nevoie de numele coloanei pe care o declarăm în POJO, dar trebuie să ne referim la acea entitate, putem schimba numele pt. tabelul bazei de date.

Unele atribute sunt : • Name
• lungime
• unique
• nullable

@ Transient = spune hibernate-ului să nu adauge această coloană specială

@ Temporal = folosită pt. a formata data pt. stocare în DB

@ Lob = folosit pt. a spune hibernate că este un obiect mare și nu este un simplu obiect

@ OrderBy = folosită pt. a spune Hibernate la OrderBy aza cum facem în SQL

de ex: trebuie să ordonăm după prenumele elevului în ordine crescătoare

@ OrderBy(" prenume asc")

• În JPA, relațiile între entități sunt gestionate folosind anotări ca @OneToMany, @ManyToMany, @OneToOne.

CONCURENȚA și SCALABILITATEA

- două concepte critice pt. a construi aplicații eficiente, robuste și capabile să facă față unor cerințe ridicate

1) CONCURENȚA = se referă la abilitatea unei aplicații de a gestiona mai multe sarcini simultan
   - într-o aplicație web, acest lucru înseamnă gestionarea mai multor cereri (requests) din partea utilizatorilor, fără a afecta performanța sau comportamentul aplicației
   - Java oferă mai multe mecanisme pt. a gestiona concurența:
     - Threading: Java are un model de threading robust care permite crearea de thread-uri (fire de execuție) separate pt. a rula sarcini concurente. În aceste app. web, acest lucru este esențial pt. a răspunde simultan la mai multe cereri HTTP.
     - Executors & Thread Pools: În loc să creze un nou thread pt. fiecare cerere, Java conferă conceptul de thread pool, care reutilizează thread-urile existente, îmbunătățind astfel eficiența și reducând overhead-ul creat de crearea și distrugerea thread-urilor.
     - Sincronizare: Mecanismele de sincronizare (ex: synchronized + locks) sunt folosite pt. a preveni problemele cauzate de accesul concurent la resursele partajate, cum ar fi obiectele din memorie sau bazele de date.
     - Framework-uri pt. concurență: Java oferă de asemenea java.util.concurrent, un set extins de clase pt. a gestiona concurența într-un mod eficient și sigur, inclusiv clase precum Future, Callable, Semaphore, CountDownLatch.
     - Non-blocking I/O (NIO): Biblioteca NIO (New I/O) din Java permite gestionarea conexiunilor și datelor într-un mod non-blocking, ceea ce

2) SCALABILITATE = reprezintă capacitatea unei aplicații de a se extinde eficient pt. a gestiona creșteri în volum de lucru (ex: nr. mai mare de utilizatori).

TIPURI DE SCALABILITATE:

- Scalare pe verticală (Vertical Scaling)
  = adăugarea de resurse suplimentare unui singur server (ex procesor mai puternic, memorie mai multă). Deși aceasta poate îmbunătăți performanța, are limite fizice.

- Scalare pe orizontală (Horizontal Scaling)
  = adăugarea de servere suplimentare pt. a partaja încărcarea aplicației. În acest model, mai multe instanțe ale aplicației sunt rulate pe diferite mașini, iar cererile sunt distribuite între ele folosind un load balancer.

Java oferă mai multe abordări pt. a facilita scalabilitatea

- Stateless Services: Aplicațiile stateless sunt mai ușor de scalat, deoarece nu păstrează starea utilizatorilor între cereri, ceea ce permite distribuirea uniformă a cererilor între mai multe servere.

- Distribuția sarcinilor (Load Balacing): Load Balancer-ele distribuie cererile între instanțele aplicației, permițând astfel scalarea orizontală și îmbunătățind performanța.

Algoritmul C= Round Robin

- Caching: Utilizarea caching-ului (ex Ehcache | Redis) pentru a reduce accesările inutile la baza de date sau la alte resurse costisitoare. Acest lucru îmbunătățește timpul de răspuns și reduce sarcini pe sistem.

- Connection Pooling: în loc să creze o conexiune la baza de date pentru fiecare cerere, JAVA utilizează CP, care permite reutilizarea conexiunilor existente, reducând overhead-ul și timpul de procesare.

- Microservicii: trecerea de la o arhitectură monolit la o arhitectură de microservicii este o altă modalitate de a îmbunătăți scalabilitatea. Fiecare serviciu este independent și poate fi scalat separat în funcție de necesități.

Concurență și scalabilitate în framework-uri JAVA:

→ Spring Framework: Spring oferă multe caracteristici pt a îmbunătăți concurența și scalabilitatea. De exemplu, SPRING WEBFLUX, suportă model non-blocking și este optim pt. aplicațiile reactive

→ JAVA EE: Platforma Java Enterprise Edition (acum Jakarta EE) oferă specificații precum EJB și JMS care ajută la gestionarea concurenței și scalabilității într-un mod eficient.

→ AKKA: este un framework bazat pe modelul actorilor, care facilitează construirea de aplicații concurente și distribuite.

JMS = Jakarta Messaging
└ Java Messaging Service

SPRING WEBFLUX (SPRING 5) = oferă suport pt. programarea reactivă pt. aplicații web

reactive REST app = $\underbrace{\text{RestController + WebClient}}_{\text{reactive web components}}$

Random Questions:

1. Cum funcționează Garbage Collector în Java? Care sunt cele mai comune tipuri de GC?

R: GC automatizează gestionarea memoriei prin identificarea și eliberarea obiectelor care nu mai sunt accesibile.

Tipuri: Serial GC, Parallel GC, Concurrent Mark-Sweep (CMS) și G1 Garbage Collector.

2. HashMap vs. ConcurrentHashMap

R: HashMap nu este sincronizată și nu este thread safe. ConcurrentHashMap este sincronizată și poate fi utilizată în medii multithreading. Folosește tehnici de blocare segmentată pt. a îmbunătăți performanța.

3. Race conditions apar când două sau mai multe threaduri accesează și modifică simultan aceleași resurse. Pentru a le evita, se folosesc mecanisme de sincronizare ca synchronized, Locks sau Atomic classes.

4. Spring Security se configurează cu anotări ca @EnableWebSecurity și clase de configurare care extind WebSecurityConfigurerAdapter. Autentificarea poate fi configurată prin filtre, token-uri JWT sau bazată pe sesiuni.

5. SQL = relațional, structurat și folosește tabele cu relații predefinite

NOSQL = nerelațional, utilizează documente, key-value sau grafuri.

6. Relațiile bidirecționale sunt gestionate în Hibernate folosind "mappedBy" pt. a indica proprietarul relației și "cascade" pt. a propaga modificările.

7. Cum integrezi CI/CD pipelines în dezvoltarea aplicațiilor JAVA? Cum funcționează un pipeline în Jenkins?

Jenkins este un server de automatizare open-source folosit pt. integrarea continuă (CI) și livrarea continuă (CD). Cu Jenkins, se pot automatiza toate etapele procesului de dezvoltare software, de compilare și testare până la livrare în producție. Este folosit pt. a implementa pipeline-uri CI/CD care permit echipelor să îmbunătățească fluxul de lucru și să livreze cod de calitate mai rapid.

CI/CD sunt practici esențiale în dezvoltarea modernă a aplicațiilor inclusiv pt. aplicații Java. CI/CD pipeline-ul automatizează procesele de testare, construcție și livrare a codului, asigurând că orice schimbare în cod este verificată automat și livrată rapid.

Pașii principali într-un pipeline CI/CD tipic pt. o aplicație Java:

1. Scrierea codului și gestionarea versiunilor (Git);
2. Testarea automată la fiecare commit:
   - Jenkins detectează automat un nou commit și pornește procesul CI
   - Codul este compilat (Maven / Gradle)
   - Sunt executate testele unitare pt. a verifica că funcționalitatea este corectă
3. Testare avansată
   - Teste de integrare, de performanță, sau de securitate pot fi incluse într-un pipeline.
4. Build și deploy
   - Dacă testele trec cu succes, codul este compilat și pachetat (.jar/.war)
   - În cazul CD, Jenkins poate livra automat aplicația pe un server de testare sau producție.
5. Monitorizare și notificare:
   - Jenkins trimite notificări prin email sau alte canale de comunicare în caz de eșecuri sau succes.

Cum funcționează un pipeline CI/CD cu Jenkins?

- Jenkins pipeline este un set de pași care descriu cum este construit și livrat codul. Se pot crea pipeline-uri simple sau complexe folosind Jenkinsfile, care este un fișier de configurare scris în format declarativ.

8. SWAGGER = tool care generează automat documentația API-urilor RESTful. În Spring Boot, se poate integra folosind biblioteci ca SpringFox sau OpenAPI. Odată configurat, Swagger expune o interfață grafică la un endpoint ca /swagger-ui.html.

9. DOCKER = platformă de virtualizare la nivel de sistem de operare care permite dezvoltatorilor să creeze, să distribuie și să ruleze aplicații în containere. Un container este o unitate standardizată de software care împachetează codul aplicației și toate dependințele necesare pt. a rula (biblioteci, runtime, configurații) într-un mediu izolat și portabil. Acest lucru face ca aplicațiile să fie consistente și ușor de rulat indiferent de mediul în care au fost implementate (dezvoltare, testare, producție)

- Cum containerizezi o aplicație JAVA cu Docker?

1. Scrierea codului JAVA $\xRightarrow{\text{maven}}$ .jar / .war (fișier executabil)
2. Crearea fișierului Dockerfile ← fișier text simplu ce conține instrucțiuni pt. a construi o imagine Docker. O imagine Docker este o copie a mediului în care aplicația va rula
3. Construirea imaginii Docker: docker build -t myapp
4. Rulare container: imaginea Docker construită poate fi rulată ca un container folosind:

   docker run -p 8080:8080 myapp

- izolarea aplicațiilor
- portabilitate
- scalabilitate
- gestionarea ușoară a dependințelor
- CI/CD

10. OAuth 2.0 permite autentificarea utilizatorilor prin furnizori de identitate terți (Google, Facebook, etc). În Spring, se poate integra folosind Spring Security OAuth. Aplicația redirecționează utilizatorul către furnizorul de autentificare, obține un token de acces și îl folosește pt. a accesa resurse partajate.

11. Cum se poate proteja o aplicație împotriva atacurilor Cross-Site Scripting (XSS) și SQL injection?

Pentru XSS, se folosește validarea și sanitizarea inputurilor și outputurilor. În Spring, se poate activa protecția XSS în configurația de securitate.

Pentru SQL injection, se folosește Prepared Statements sau Hibernate / JPA pt. a preveni injecțiile directe în interogări SQL.

12. Cum se configurează HTTPS într-o aplicație Spring Boot?

- Se adaugă un certificat SSL și se configurează application.properties pt. a activa HTTPS:

```
server.port=8443
server.ssl.key-store=classpath:keystore.p12
server.ssl.key-store-password=password
```

13. ATOMIC CLASSES - parte a pachetului java.util.concurrent.atomic și sunt concepute pt. a oferi un mod simplu și eficient de a gestiona variabilele care sunt accesate de mai multe threaduri în aplicații multithread. Aceste clase asigură operații atomice, adică operații care sunt complet executate fără a fi întrerupte de alte threaduri, astfel prevenind problemele de concurență, cum ar fi race conditions.

Scanned with CamScanner

- Atomic classes oferă o performanță mai bună decât folosirea sincronizării (ex blocuri synchronized) în anumite scenarii, deoarece nu blochează thread-urile, ci permit accesul concurent la variabilele partajate.
- Operații de citire, scriere, actualizare sunt operații atomice, ceea ce înseamnă că nu există posibilitatea ca un alt thread să intervină în timpul acestor operațiuni.
- Aceste clase includ metode suplimentare care permit operații atomice, cum ar fi incrementarea sau decrementarea valorilor.

14. Un <u>context de aplicație</u> în Spring este un container care conține informații despre configurarea aplicației și gestionează ciclul de viață al bean-urilor (obiectelor). Contextul de aplicație extinde capacitățile ApplicationContext, permițând gestionarea configurării, a resurselor și a comportamentului aplicației.

15. Access Modifiers

→ public access = înseamnă că oricine poate accesa membrul respectiv de oriunde

→ protected access = elementele protected sunt accesibile în cadrul aceluiași pachet și, în plus, pot fi accesate de către clasele care extind (moștenesc) clasa care definește aceste elemente, chiar dacă se află în alt pachet.

→ private access = sunt accesibile doar în interiorul clasei în care sunt declarate. Acestea nu pot fi accesate din alte clase, nici măcar de clasele care se află în același pachet

→ default access = dacă un element nu este marcat cu un modificator de access explicit, acesta este considerat 'default'. În acest caz, elementele sunt accesibile doar din clasele care fac parte din același pachet.

Acești modificatori de acces oferă controlul asupra vizibilității datelor și metodelor în aplicația JAVA, permițând o mai bună încapsulare și modularizare a codului.

- Inversion of control (IoC) este un principiu de design care inversează fluxul de control într-un program prin decuplarea componentelor, făcându-le mai flexibile și mai ușor de întreținut. Într-o abordare tradițională, componentele de nivel înalt controlează fluxul și apelează componente de nivel inferior. Cu **ioc**, acest control este inversat, iar componentele de nivel inferior sunt responsabile de flux, permițând componentelor de nivel înalt să se concentreze pe funcționalitățile lor de bază.

Inversion of control în JAVA : Există mai multe modalități de a implementa IoC în Java, însă cea mai comună abordare este prin DI (Dependency Injection). DI este o (design pattern) = tehnică folosită pt. a injecta dependințele (adică obiectele) într-un obiect, în loc ca obiectul să fie responsabil de crearea și gestionarea propriilor dependințe.

- Ce este un Enterprise Java Bean (EJB) și când ar trebui folosit?

EJB este un component Java EE pt. scrierea aplicațiilor enterprise, care necesită servicii tranzacționale, securitate și scalabilitate. EJB-urile sunt utile pt. 'aplicații distribuite' și de mare scală, dar au fost în mare parte' înlocuite de framework-uri moderne precum SPRING.

- Cum ai optimiza o interogare SQL pentru performanță?

Optimizarea interogărilor SQL poate include utilizarea indexurilor, evitarea subinterogărilor inutile, folosirea JOIN-urilor adecvate în locul (IN) sau (EXISTS) și reducerea volumului de date returnate prin utilizarea unor filtre (WHERE, LIMIT).

- Cum se configurează un proiect multi-module în Maven?

Un proiect multi-module în Maven este organizat într-un director părinte (parent), care conține un fișier 'pom.xml' ce definește configurația comună pt. toate modulele. Fiecare modul are propriul său 'pom.xml', iar toate modulele sunt specificate în secțiunea <modules> din POM-ul părinte

-

POJO	JavaBean
1) nu are restricții specifice altele decât cele impuse de limbajul Java	1) reprezintă un POJO special ce are câteva restricții
2) Nu oferă prea mult control asupra membrilor	2) oferă control complet asupra membrilor
3) poate implementa interfața Serializable	3) trebuie să implementeze interfața Serializable
4) membrii pot fi accesați prin numele lor	4) membrii sunt accesați doar prin intermediul metodelor getter și setter
5) membrii pot avea orice vizibilitate	5) membrii trebuie să aibă doar vizibilitate privată
6) Poate/Nu poate să existe un no-arg constructor	6) Trebuie să existe un no-arg constructor.
7) folosit atunci când nu dorești restricționare asupra membrilor și se oferă acces complet utilizatorului la respectiva entitate.	7) folosit când se dorește să se ofere utilizatorului entitate, lăsă doar o parte din aceasta.

- "==" : compară referințele a două obiecte
  folosită pt. a verifica dacă 2 referințe pointează către același obiect din memorie
  vs.
  "equals()" : compară valorile / conținutul a două obiecte
  folosită pt. a verifica dacă 2 obiecte sunt logic egale (their data is the same).

- De ce metodele hashCode() și equals() trebuie suprascrise?

  În momentul în care creăm o clasă custom, de obicei dorim să comparăm conținutul a două obiecte (nu referințele acestora), astfel suprascriem metoda "equals()" pt. a defini logica noastră de evaluare a egalității.

  Dacă suprascriem metoda "equals()", atunci va trebui să suprascriem și metoda "hashCode()" neapărat, pentru a menține contractul dintre cele două metode.

- Synchronized = keyword folosit pt. a controla accesul asupra unei secțiuni de cod ce nu ar trebui să fie executată concurent de mai multe threaduri
  - metode synchronized
  - blocuri synchronized

  Metode synchronized = se asigură că numai un thread poate executa acea metodă pe o anumită instanță a clasei la un moment dat! Blocarea este asociată cu instanța.

  Metode synchronized statice = asigură că doar un thread poate executa orice metodă statică sincronizată pt. întreaga clasă.

  Blocuri synchronized = permit blocarea unei anumite secțiuni de cod, oferind control mai fin decât sincronizarea întregii metode.

COMMON CONCURRENCY ISSUES

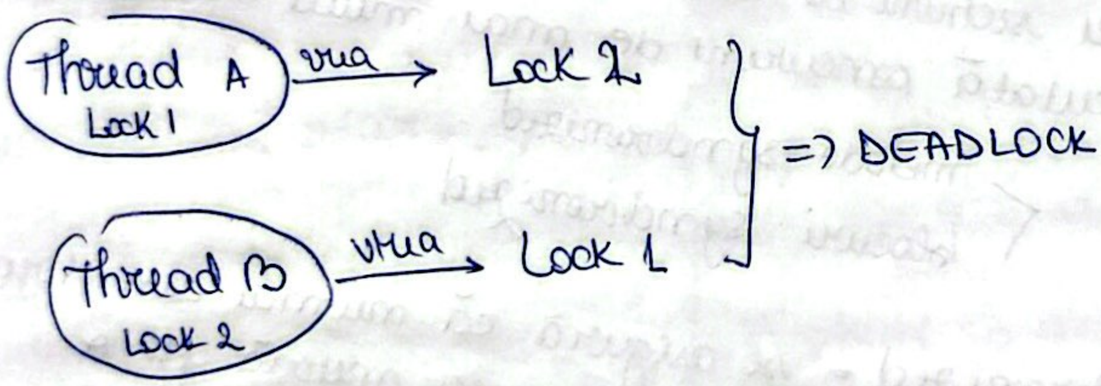
1) RACE CONDITIONS = apar atunci când două sau mai multe thread-uri accesează date partajate simultan, iar rezultatul final depinde de ordinea în care se execută thread-urile.

=> acest lucru duce la un comportament imprevizibil, deoarece ordinea de execuție corectă nu este garantată

=> HOW TO AVOID: synchronized, reentrant lock, atomic ops.

2) DEADLOCK = apare atunci când două sau mai multe fire sunt blocate pt. totdeauna, fiecare așteptând o blocare deținută de celălalt thread.

=> apar atunci când sunt obținute mai multe lock-uri în ordine diferită, ceea ce duce la o situație circulară de așteptare.



=> HOW TO AVOID: • Lock ordering
• Timeouts

3) LIVELOCK = firele continuă să își schimbe starea ca răspuns unul la altul, dar niciunul nu poate face niciun progres. Spre deosebire de deadlock, firele aici rămân active, dar nu pot continua.

ex: 2 fire care cedează constant control unul altuia

=> HOW TO AVOID: strategii de retragere

4) INCONSISTENT STATE/DATA CORRUPTION = în momentul în care mai multe threaduri updatează simultan o resursă partajată fără sincronizare, poate duce la corupere de date sau inconsistență

=> HOW TO AVOID: synchronized method/blocks, atomic ops.

THREAD LOCAL = clasă din Java care furnizează variabile locale ale firului de execuție.

În loc să partajeze o singură instanță a unei variabile între mai multe fire de execuție, fiecare fir de execuție care accesează o variabilă threadLocal primește propria cheie independentă. Acest lucru asigură că modificările făcute de un thread pe copia lui nu impactează copiile utilizate de alte fire.

Un ThreadLocal este util în momentul în care se dorește evitarea stării partajată și complicațiile sincronizării, în special în mediile multithread.

- Un caz obișnuit de utilizare este atunci când fiecare fir de execuție are nevoie de propriul context sau stare, cum ar fi stocarea informațiilor despre sesiunea utilizatorului, conexiunile la baza de date sau instanțele SimpleDateFormat (nu este sigur pt. firele de execuție)

ex: Dacă avem un obiect non-thread-safe, cum ar fi SimpleDateFormat, putem folosi un ThreadLocal pt. a oferi fiecărui thread propria instanță a formatorului, evitând sincronizarea sau alte probleme de concurență.

```
private static ThreadLocal<SimpleDateFormat> dateFormat =
        = ThreadLocal.withInitial(() -> new SimpleDateFormat(
                                        "yyyy-MM-dd"));
public static formatDate(Date date) {
    return dateFormat.format(date); // fiecare thread
                                       are propria lui
                                       instanță
}
```

• Deși ThreadLocal este puternic, ar trebui utilizat cu foarte mare atenție. O capcană majoră o reprezintă posibilele memory leaks în pool-urile de fire, unde firele de execuție pot fi reutilizate. Pentru a preveni acest lucru, este o practică bună să eliminați valorile ThreadLocal odată ce acestia nu mai sunt necesare folosind metoda 'remove()'.

• Handle errors in Spring Boot:

- @ControllerAdvice for global exception handling
- Create custom exceptions classes for specific error
- Return 'ResponseEntity' for more control over HTTP responses
- Custom errors response classes

GENERICS

Implementarea unei clase sau metode generice în Java permite crearea de clase și metode care pot funcționa pe obiecte de diferite tipuri, oferind în același timp siguranța tipului de compilare. Această caracteristică este utilă în special pentru crearea structurilor de date și a metodelor utilitare care pot funcționa cu orice tip de date.

1. Generic Class - definită cu unul sau mai multi parametri de tip incluși între paranteze unghiulare (<>). Acești parametri tip pot fi utilizați ca substituenți pt. tipuri reale care vor fi specificate la crearea unei instanțe a clasei.

2. Generic Method - definită cu propriii parametri de tip, ceea ce permite definirea unei metode care poate opera pe diferite tipuri. Parametrii de tip sunt specificați înaintea tipului de returnare al metodei.

3. Bounded type parameters = se pot restricționa tipurile care pot fi utilizate ca parametrii de tip prin specificarea limitelor. Acest lucru se face folosind cuvântul cheie 'extends'.

4. Wildcards = folosite pentru a reprezenta un tip necunoscut. Se pot folosi în parametrii metodelor sau în definirea claselor pentru a oferi o mai mare flexibilitate.

JAVA EE (Enterprise Edition) = set de specificații care extind Java SE (Standard Edition) cu specificații pt. caracteristici enterprice, cum ar fi calculatia distribuită și serviciile web.

Componente:

- Servlets: clase Java care gestionează cererile și răspunsurile HTTP. Sunt folosite pt. a crea conținut web dinamic.
- JavaServer Pages (JSP): tehnologie care permite încorporarea codului Java în pagini HTML, simplificând crea de conținut web dinamic
- Enterprise JavaBeans (EJB): arhitectură de componente pe partea de server pt. construcția modulară a aplicațiilor enterprise, facilitând logica de business.
- Java Persistance API (JPA): pt. gestionarea datelor relaționale în aplicații Java, permițând maparea obiect-relațională
- JMS (Java Messaging Service): standard de mesagerie care permite componentelor aplicației să comunice între ele.
- JTA: interfață standard pt. gestionarea tranzacțiilor în Java EE
- JAX-RS: permite crearea de servicii web RESTful
- JAX-WS: SOAP
- CDI (Contexts and D.I.): gestionarea ciclului

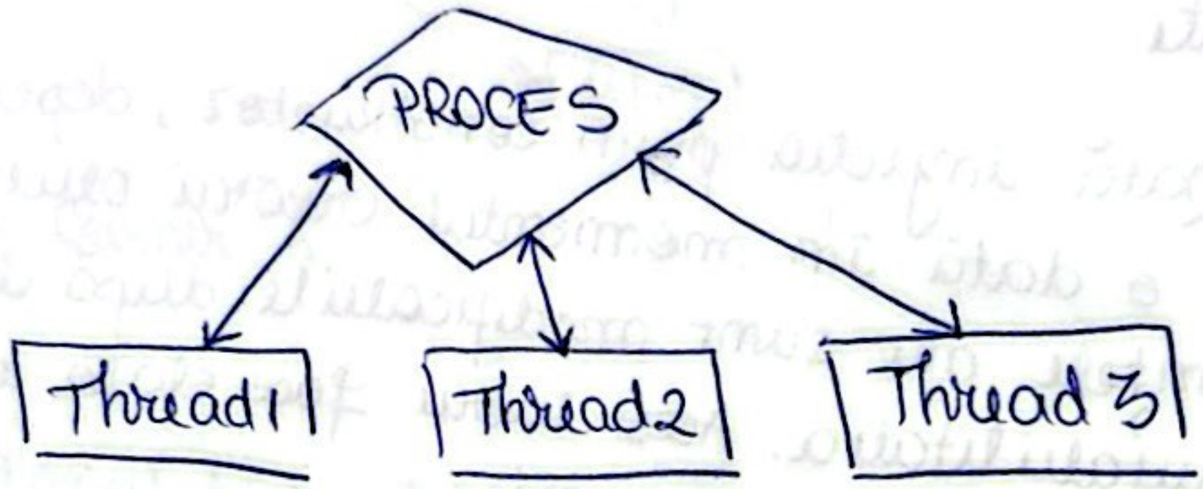
EJB (Enterprise JavaBeans) Arhitecture

EJB este o arhitectură de componente pe partea de server, concepută pt. a încapsula logica de business într-un mediu distribuit.

Principalele scopuri EJB:

- Simplificarea dezvoltării: EJB oferă servicii precum gestionarea tranzacțiilor, securitate, controlul concurenței și accesul de la distanță, permițând dezvoltatorilor să se concentreze pe logica de business.
- Scalabilitate și performanță: Componentele EJB pot fi distribuite pe mai multe servere, acceptând echilibrarea încărcăturii și fail-overul.
- Reutilizabilitate: modularitate + reducerea timpului de dezvoltare
- Integrare: EJB + JPA + JMS

1) Multithreading înseamnă mai multe fire și este considerată una dintre cele mai importante caracteristici ale Java. Reprezintă capacitatea unui procesor de a executa mai multe fire de execuție independent în același timp, dar partajează resursele procesului simultan. Scopul său principal este de a oferi execuția simultană a mai multor fire de execuție pt. a utiliza cât mai mult posibil timpul CPU.



- start(): metodă folosită pt. a începe execuția unui fir nou creat. Când este apelată această metodă, este creat un fir nou și acest fir nou creat execută sarcina care este păstrată în metoda 'run()'. Se poate apela metoda start o singură dată.

- run(): metodă folosită pt. a începe execuția aceluiași fir. Când este apelată această metodă, nu este creat niciun fir nou ca în cazul metodei 'start()'. Această metodă este executată de firul curent. Se poate apela metoda 'run()' de mai multe ori.

- User și Daemon = 2 tipuri de fire utilizate în Java prin utilizarea unei "Thread Class".

User Thread = firele de execuție user au un ciclu de viață specific și viața sa este independentă de orice alt fir. JVM așteaptă ca oricare dintre firele de execuție a utilizatorului să își finalizeze sarcinile înainte de a-l termina. Când firele de execuție ale utilizatorului sunt terminate, JVM termină întregul program împreună cu firele daemon

- Daemon thread: sunt denumite în principiu un furnizor de servicii care oferă servicii și suport pentru firele de execuție. Are 2 metode: setDaemon()
  isDaemon()

Constructor-based DI over @Autowired

1. Imutabilitate

Când este utilizată injecția prin constructor, dependențele sunt transmise o dată în momentul creării obiectului și, deoarece dependențele nu sunt modificabile după inițializare, promovează imutabilitatea. Acest lucru face clasa mai sigură și mai puțin predispusă la erori.

Cu @Autowired, în special pt. injectarea câmpului, dependențele pot fi modificate dacă câmpul @Autowired nu este final, făcându-l mutabil.

2. Testarea

Injecția prin constructor facilitează scrierea testelor unitare, deoarece dependințele pot fi ușor mockuite și transmise ca argumente de constructor.

Injecția în câmp cu @Autowired îngreunează testarea unitară, deoarece nu se pot injecta direct mockuri fără a utiliza reflection sau un mecanism specific Spring, cum ar fi SpringJUnit4ClassRunner.

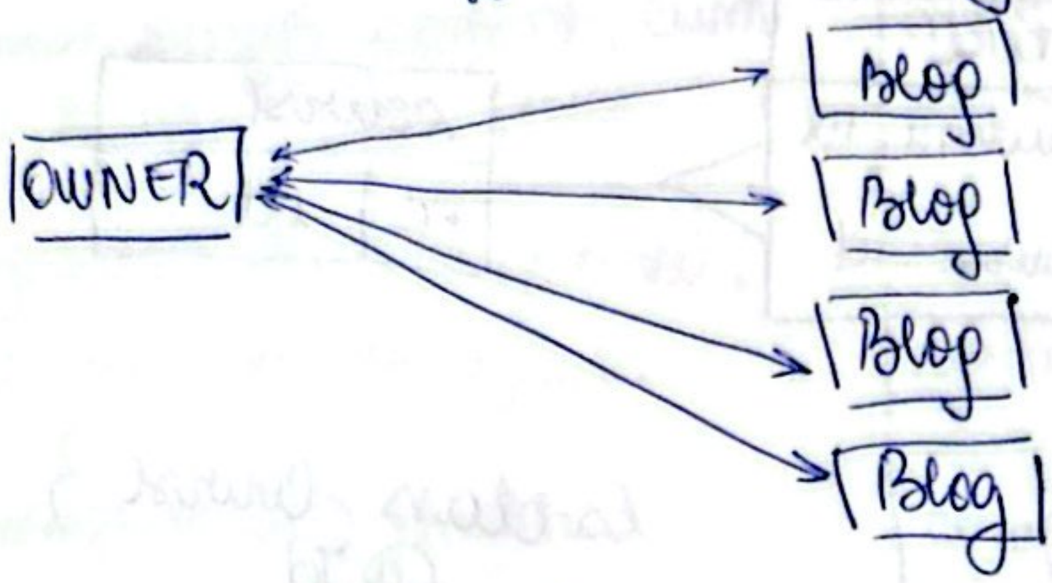
3. Face dependințele explicite și clare
4. Prinde dependințele lipsă în timpul compilării
5. Evită probleme de rulare și reflection.

Scanned with CamScanner

Relationship using @OneToOne
@OneToMany | @ManyToOne
@ManyToMany

} Annotation in Spring Boot

One to Many




```
public class Owner {

  ....

  @OneToMany(fetch = FetchType.LAZY, mappedBy = "owner",
  private List<Blog> blogList;                cascade = CascadeType.ALL)

}

public class Blog {

  @ManyToOne(cascade = CascadeType.ALL)
  @JoinColumn(name = "owner_id")
  private Owner owner;

}
```

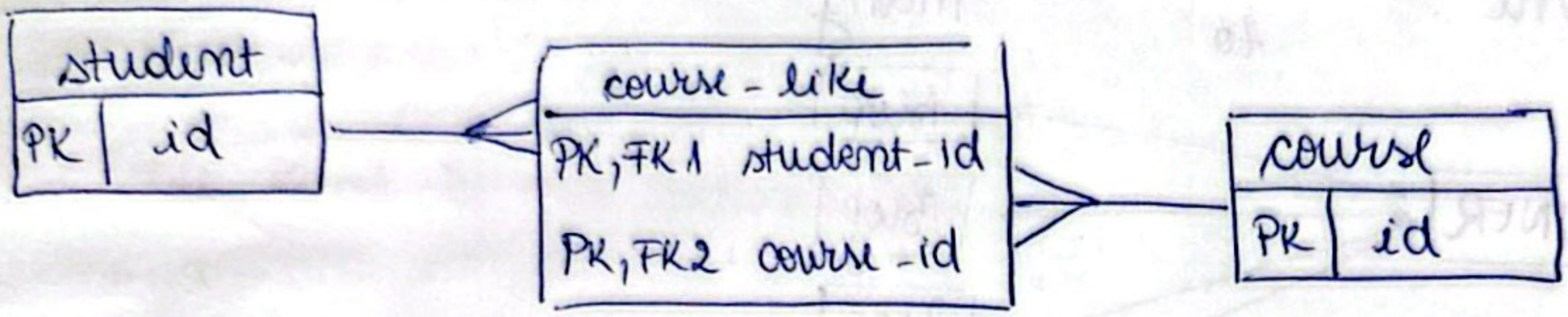
@JoinColumn = specifică coloana folosită pt. join în baza de date, de obicei în relațiile @ManyToOne, @OneToOne

- atribute: name (numele coloanei), nullable, unique
- folosită pt. a defini care coloană repr. foreign key

@JoinTable = folosită în relația @ManyToMany pt. a defini tabela de join

- atribute: name, joinColumns, inverseJoinColumns

eg




```
class Student {
    @id
    Long id;
    @ManyToMany
    Set<Course> likedcourses;

}
```

```
class Course {
    @Id
    Long id;
    @ManyToMany
    Set<Student> likes;
}
```

(mappedBy = "likedCourses")

The owner side is where we configure the relationship.
We'll use the Student class.

```
@JoinTable( name = "course_like",
            joinColumns = @JoinColumn (name = "Student_id"),
            inverseJoinColumns = @JoinColumn (name = "course_id"))
```

	Comparable	Comparator
1)	Comparable oferă o singură secvență de sortare. Deci putem sorta secvența pe baza unui singur element, cum ar fi id, name, price.	The Comparator oferă mai multe secvente de sortare. Deci putem sorta 'colecția pe baza mai multor elemente cum ar fi id, name sau price.
2)	Afectează clasa originală, aceasta este modificată.	Nu afectează clasa originală, aceasta rămâne nemodificată.
3)	compareTo() method	compare() method
4)	java.lang package	java.util package
5)	List < Comparable type > => => Collections.sort(List) method	List < Comparator type > => => Collections.sort(List, comparator) method.

```
public class Student implements
        Comparable < Student > {

---
int age;
// constructor
public int compareTo(Student st){
    if(age == st.age) return 0;
    else if(age > st.age) return 1;
    else return -1;
}
```

```
public class Student {
--- String name; int age; ---}

class AgeComparator implements
        Comparator < Student > {

 public int compare(Student s1,
        Student s2) {
  if(s1.age == s2.age) return 0;
  else if (s1.age > s2.age) return 1;
  else return -1;
}}

---
Collections.sort(students, new
        AgeComparator());
```

STRING CLASS are de asemenea metoda compareTo()!!!

THREADS p2.

Callable + Runnable = utilizate pt. a defini o sarcină care poate fi executată de un thread.

- pentru a rula un Runnable, putem folosi clasa Thread sau un ExecutorService (interfață ce poate executa atât un Callable cât și un Runnable, returnând un Future - metoda submit).

- pentru a rula un Callable, putem folosi doar ExecutorService

- Runnable are o singură metodă (e interfață funcțională) - -void run() => executorService.submit(Runnable) returnează un Future fără valoare.

- Callable e o interfață generică cu o singură metodă call() care returnează tipul generic.

- Runnable nu are nicio clauză de throws specificată => => excepțiile checked nu pot fi propagate

- Callable are specificată clauza de throws, iar în cazul în care este aruncată o excepție, aceasta va fi prinsă în obiectul Future returnat după execuția thread-ului și nu va fi aruncată mai departe decât dacă apelăm Future.get()

---

ex1
```
public interface Runnable { void run(); }
Runnable task = () -> { sout("..") ); };
Thread thread = new Thread(task);
thread.start();
```

---

ex2
```
public interface Callable<V> { V call() throws Exception; }
Callable<Integer> task = () -> {return 42;};
ExecutorService executor = Executors.newCachedThreadPool();
Future<Integer> future = executor.submit(task);
try { Integer result = future.get(); sout(result);
}catch (Exception e) { e.printStackTrace(); }
executor.shutdown();
```

Interfața Future = folosită pentru a reprezenta rezultatul unei sarcini asincrone, care poate fi obținut la un moment anterior

O instanță de Future este returnată atunci când o sarcină Runnable sau Callable este trimisă către un ExecutorService.

Metode:
- get() : blochează execuția până când sarcina este completă și returnează rezultatul
- cancel(): poate fi folosit pt. a încerca anularea sarcinii
- isDone(): verifică dacă sarcina a fost finalizată

Clasa CompletableFuture = implementare a interfeței Future, care adaugă o mulțime de metode pentru lucrul cu sarcini asincrone și compoziționale.

Permite:
- executarea sarcinilor asincrone fără a bloca firul curent
- Compoziție: Posibilitatea de a combina mai multe sarcini asincrone.
- Manipularea erorilor

Metode:
- supplyAsync() (Supplier<T>) : creează și execută o sarcină asincronă care returnează un rezultat
- thenApply(): transformă rezultatul unei sarcini asincrone
- thenAccept() (Consumer<T>): acceptă rezultatul unei sarcini asincrone și nu returnează nimic
- thenCompose() : lanțează mai multe CompletableFuture-uri, unde rezultatul unei sarcini este folosit ca intrare pentru următoarea sarcină.

thenRun() → Runnable

Combinarea sarcinilor: se pot combina mai multe CompletableFutures ca. să ruleze
- secvențial
- paralel : CompletableFuture.allOf(future1, future2);

KAFKA = este o platformă pentru procesarea de stream-uri de date în timp real.

= este high-throughput (poate produce un nr. mare de mesaje într-o perioadă stabilită de timp), low-latency (răspuns foarte rapid, aproape de real-time).

= arhitectura se bazează pe un log imutabil de date care poate fi organizat pe topicuri în vederea consumării de multiplii utilizatori.

= se fac replicări ale mesajelor pt. recuperarea și consistența datelor în cazul în care pică sistemul.

PERFORMANCE, LOAD, STRESS TESTING - teste esențiale pt. evaluarea comportamentului unei aplicații în condiții diferite de utilizare
- previn problemele de performanță din producție

1) Performance testing => scop: măsurarea performanței generale a a aplicației, cum ar fi
- timpul de răspuns
- viteza de procesare
- eficiența sub sarcini normale

=> obiectiv: identificarea potențialelor probleme de performanță cum ar fi latențe mari sau timpi de procesare mari

-> Metricile cheie:
- timpul de răspuns
- throughput: numărul de op (cereri) pe secundă pe care aplicația le poate proc
- utilizarea resurselor: monitorizarea utilizării resurselor hardware, cum ar fi CPU, memorie, disc, rețea

-> Instrumente populare:
- JMeter
- Gatling
- VisualVM

2) Load testing => scop: evaluarea performanței aplicației sub o sarcină specifică, care reflectă utilizarea anticipată în producție

=> obiectiv: verificarea cum se comportă aplicația atunci când este expusă la un număr ~~mare~~ crescut de utilizatori sau cereri simultane

=> Metrici cheie:
- throughput și timp de răspuns sub încărcare
- capacitate maximă
- rata de eroare

=> Instrumente populare:
- Apache JMeter
- LoadRunner
- Blaze Meter

3) Stress Testing => scop: verifică comportamentul aplicației sub o încărcătură extremă sau chiar dincolo de limitele sale normale, pt. a observa cum răspunde la condiții critice

=> obiectiv: identificarea punctelor de eșec ale aplicației și evaluarea capacității de recuperare. Stress testing ajută să vezi cum aplicația se comportă când este suprasolicitată sau când întâlnești condiții neobișnuite, cum ar fi o creștere bruscă a traficului

=> Metrici cheie:
- capacitatea de recuperare
- Rezistența la failover
- Degradarea performanței

=> Instrumente populare:
- JMeter + Gatling
- Chaos Monkey (Netflix)

KIBANA = instrument de vizualizare care funcționează cu Elastic Search pentru a analiza și vizualiza jurnalele (logs) și valorile.
- permite dezvoltatorilor să creze vizualizări, jurnale de căutare și să configureze tablouri de bord pt. monitorizarea în timp real al aplicațiilor Java.

GRAFANA = se concentrează în primul rând pe time-series data și metrici, ceea ce îl face ideal pt. monitorizarea performanței, în timp ce Kibana este mai bun pt. căutarea și vizualizarea datelor din logs.
- se integrează cu Prometheus pt. a monitoriza valorile în timp real, cum ar fi utilizarea CPU, consumul de memorie, timpii de răspuns în aplicații JAVA.

Grafana în Spring Boot app
- valorile pt. Grafana sunt aduse de Prometheus
- în aplicație activăm metricile actuator pt. a expune datele de performanță ale aplicației și ale JVM
- în Grafana creăm un dashboard pt. a urmări valori cheie ca
  - utilizarea memoriei HEAP JVM
  - utilizarea procesorului
  - timpii de răspuns pt. REST endpoint
  - rata de eroare.

PROMETHEUS - integrat cu Java folosind biblioteci precum Micrometer sau Prometheus Client Java
- în Spring Boot oferă asistență imediată pt. expunerea valorilor folosind punctul final /actuator/prometheus
- scrapuiește periodic acest punct periodic pt. a colecta metrici JVM ca utilizarea memoriei, activitatea GC, numărul de threaduri

REFLECTION în Java reprezintă un mecanism care permite unui program să inspecteze și să modifice comportamentul la runtime al claselor, metodelor și obiectelor. Cu Reflection, se pot accesa informații despre clase, metode, câmpuri, constructori și chiar se pot executa metode sau accesa câmpuri private, care în mod normal nu ar fi accesibile.

JOIN în threads: folosită pentru a sincroniza execuția thread-urilor. Atunci când un fir de execuție apelează metoda 'join()' pe un alt fir, se suspendă execuția firului curent până când firul pe care s-a apelat join() își termină execuția.

Cum funcționează?

- Dacă firul A apelează join() pe firul B, firul A va aștepta până când firul B își finalizează execuția.
- După ce firul B a terminat de rulat, firul A își reia execuția de unde a rămas.

threadB.join()

void
- join(): așteaptă ca firul respectiv să se termine
- join(long ms): așteaptă pt. un anumit timp (milisecunde) și apoi continuă execuția chiar dacă firul respectiv nu s-a terminat.
- join(long ms, int nanos): așteaptă o anumită durată (în milisecunde și nanosecunde) și apoi continuă execuția.

→ util pt. SINCRONIZARE: permite sincronizarea execuției firelor, asigurându-ne că un fir va aștepta finalizarea altuia

→ util pt. CONTROL: se poate controla ordinea execuției anumitor task-uri care rulează în fire diferite.

LOAD BALANCERS

- adăugarea mai multor servere sistemului (scalarea pe orizontală) => mai mult de un server poate fi folosit pentru a servi un request => la care server să trimitem requestul? =>
=> LOAD BALANCERS

- cum funcționează: un 'load balancer' îl va distribui eficient traficul primit pentru a maximiza capacitatea sistemului și pentru a minimiza timpul de răspuns
  -> folosește diferite strategii de distribuire a traficului:
    - Round Robin
    - Least connection
    - Consistent hashing (based on IP sau URL)

LOAD BALANCERS (LB) = eficiente și highly available depinzând de strategia aleasă și de regulile de performanță

↳ pot fi implementate în hardware

De obicei, inginerii nu-și implementează propriul LB, în schimb folosesc un 'industry standard reverse proxy' (ex: AT proxy || nginx) pt. a realiza LB

Providerii de cloud pun la dispoziție LB ca de exemplu Amazon cu Amazon's elastic LB.

Când să folosim LB?
- de fiecare dată când considerăm că sistemul nostru ar beneficia de o capacitate sporită sau redundanță
- LB stau de obicei între traficul extern și serverele aplicației dintr-o arhitectură bazată pe microservicii.

În cazul unei arhitecturi bazate pe microservicii, este important să folosim LB în fața oricărui serviciu intern ai. fiecare parte a sistemului să poată fi scalat independent.

DATABASE SHARDING

- tehnică de divizare a unei baze de date mari în mai multe părți mai mici, numite "shards", pentru a îmbunătăți performanța și scalabilitatea.
- fiecare shard reprezintă o instanță separată a bazei de date care conține o parte din datele complete, iar toate shard-urile combinate formează întregul set de date.
- cum funcționează? Sharding-ul împarte datele fie după anumite coloane, fie după intervale de valori, în funcție de schema bazei de date și nevoile aplicației.

metode: 1) Sharding pe bază de cheie: Împarte datele fie după anumite coloane specifice, precum un ID unic al utilizatorului, folosind un alg. de hash. Aceasta este utilă când vrei ca fiecare shard să conțină un nr. similar de date.

2) Sharding pe interval: Împarte datele pe baza unor intervale de valori (ex: utilizatori cu ID între 1 și 1000 merg într-un shard, 1001-2000 alt shard etc). Este mai ușor de gestionat, dar poate duce la un dezechilibru în distribuția datelor.

3) Sharding geografic: Datele sunt distribuite în funcție de locație, util pt. aplicații internaționale unde datele locale sunt accesate mai des de utilizatorii din acea regiune.

AVANTAJE DATABASE SHARDING

1) Scalabilitate orizontală
- ajută la adăugarea de noi instanțe (shards) pe măsură ce crește volumul de date, fără a afecta performanța

2) Reducerea încărcării
- fiecare shard gestionează o parte din solicitări, ceea ce reduce supraîncărcarea serverelor

3) Îmbunătățirea performanței
- reduce latența deoarece interogările sunt mai rapide pe volume de date mici

DEZAVANTAJE 1) Complexitate în administrare
2) Risc de inconsistență - sincronizarea între shards poate fi dificilă, mai ales pt. op. tranzacționale
3) Rebalansarea datelor - redistribuirea datelor în caz de modificări poate afecta performanța și necesită un management suplimentar.

SYSTEM DESIGN INTERVIEW

STEP 1 understand the problem
- functional + non-functional requirments
- what should be included + excluded
- clients + consumers
- existing system parts

Non-functional requirments:
- availability
- consistency
- speed
- security
- realibility
- And cost

Questions: What is the scale of the system?
How many users should it support?
How many requests should the server handle?
Are most use-cases read-only?
Are most users on mobile devices?

Estimating Data: • Data volume
- queries / second
- storage size
- bandwidth requirements

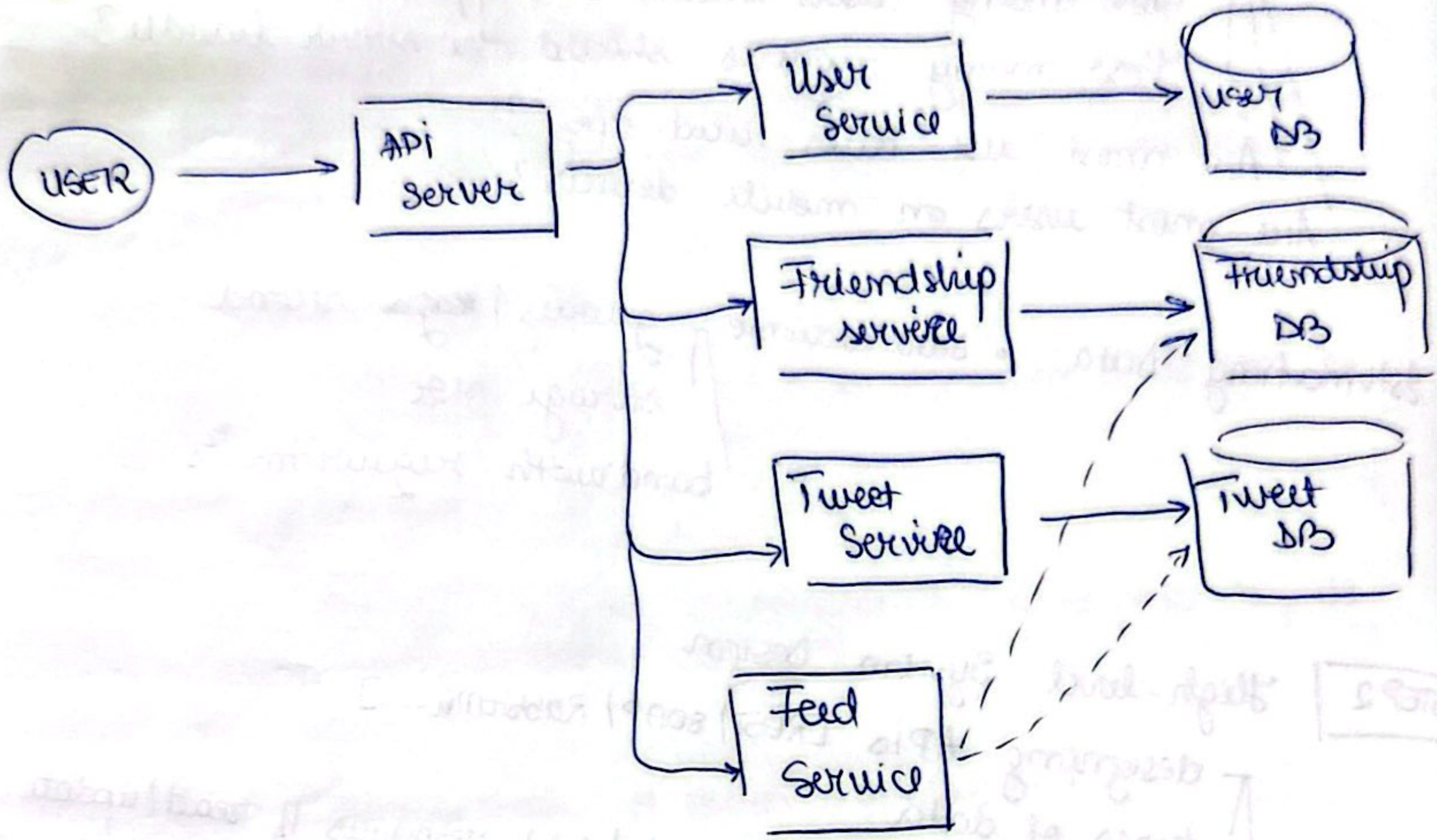
STEP 2 High-level System Design
- designing APIs [REST / SOAP / RabbitMQ ...]
- types of data
- clients may want to create / delete resources || read / update the existing ones

Server - Client communication → web sockets
|| real-time communication

Data modeling:
- simplified schema that lists only the most important fields
- data access patterns
- indexing

Database - relational (structured) ⇐ ACID transactions (AWS RDS)
- non-relational / unstructured) ← text search, blob (AWS S3
- cache (semi-structured)

High-level design diagram



STEP 3 Explore the design: Deep-dive

Transactions => dacă sistemul cere tranzacții, se ia în considerare o bază de date ce oferă ACID

Data refreshness -> dacă un sistem online cere fresh data, atunci trebuie să luăm în calcul cum să:
- speed up the data ingestion
- processing
- query process

Data size => dacă dimensiunea datelor este destul de mică să încapă în memorie (<= 100 GBs), atunci le putem păstra în memorie. Altfel, RAM -> data loss => dacă nu ne permitem să pierdem date -> găsim o soluție pt. a le face persistente

este imens => partition database

offline processing: dacă unele procesări se pot face offline sau amânate => message queues + consumers

Access patterns: QPS number, read/write ratio => impact database

STEP 4 Improve the design => Bottlenecks + scale

- Single points of failure: is there a single point that could cause the entire system to fail? How could the system be more robust and maintain uptime?
- Data replication: Is the data important enough to make copies? How important is it to keep all copies the same?
- CDNs: Does it provide a service for people all over the world? Would data centers in different parts of the world makes it faster?
- High traffic: Are there any special situations, like when many people use the system simultaneously, that could make it slow or even break it?
- Scalability: How can the system work for 10 times more people?

- By breaking down processes and implementing queues mechanisms to manage traffic, systems can be optimized for high performance scale.

SORTING ALGORITHMS

→ Merge Sort: algoritm recursiv și funcționează împărțind lista în jumătăți până când fiecare jumătate conține un singur element; apoi, acești liste mici sunt "îmbinate" într-o manieră ordonată, până când obținem o singură listă sortată

O(*) ┌ de timp: $O(n \log n)$
└ de spațiu: $O(n)$

→ Bubble Sort: compară elementele adiacente și le interschimb dacă sunt în ordine greșită

O(*) ┌ $O(n)$ cel mai bun caz când lista este deja sortat
└ $O(n^2)$

→ Selection Sort: găsește cel mai mic element și îl pune la începutul listei, repetând acest proces pt. toate pozițiile

O(*) – $O(n^2)$

→ Insertation Sort: ia elementele pe rând și le inserează în poziția corectă în lista sortată parțial

O(*) ┌ $O(n)$ cel mai bun caz când lista e sortat
└ $O(n^2)$

→ Quick Sort: Alege un pivot și se reorganizează elementele ai. toate cele mai mici să fie la stânga pivotului, iar cele mai mari la dreapta acestuia. Procedeul se repetă recursiv pt. subsecvențele rezultate.

O(*) ┌ $O(n \log n)$ cazul mediu
└ $O(n^2)$ când lista este deja sortată sau pivotul este prost ales

→ Heap Sort: Folosește o structură de date de tip heap pt. a găsi rapid cel mai mare sau cel mai mic element și a-l plasa în poziția corectă

$O(*) - O(n \log n)$

Scanned with CamScanner