



Academia Tehnică Militară "Ferdinand I"
Facultatea de Sisteme Informaticе și Securitate Cibernetică

PARADIGME DE PROGRAMARE (ÎN JAVA)

CURS 1 - INTRODUCERE

COL(R) TRAIAN NICULA

TEMATICĂ DE CURS

- Introducere în paradigmă de programare (1)
- Programare orientată pe obiecte (OOP) (3)
- Programare funcțională și asincronă (3)
- Programare reactivă (1)
- Servicii web REST (2)
- Quarkus Framework (4)



EVALUARE

Curs – 40% din nota finală

- **Minim nota 5** pentru promovare
- **Examen scris** cu subiecte tip interviu (grilă)

Laborator – 60% din nota finală

- **Primul laborator nu se evaluatează**
- Celelalte 13 laboratoare se notează astfel:
 - **maxim 10 puncte** pentru **rezolvarea exercițiilor pe timpul laboratorului**
 - **maxim 2 puncte** bonus pentru **temă** (se predă până la următorul laborator)
 - **Se promovează cu minim 65 puncte** obținute din **activitatea la laborator și teme**

ORGANIZARE

- Contact: traian.nicula01@gmail.com
- **Cursurile** se publică pe Google Drive la [link](#)
- **Laboratoarele și soluțiile exercițiilor** se publică pe Google Drive la [link](#)
- **Proiectele** conținând rezolvările exercițiilor și temele se **încarcă** pe Google Drive la [link](#):
 - Se **archivează** în format ZIP
 - Convenția de nume este: **labxx_nume_prenume.zip** pentru laborator și **temaxx_nume_prenume.zip** pentru temă (xx este numărul laboratorului)
 - Pentru a încărca fișierele ZIP **fiecare student** trebuie să aibă un **cont Google**

CUPRINS CURS

- Java — scurt istoric
- Paradigme de programare în Java
- Programare imperativă (OOP)
- Programare declarativă
 - Programarea funcțională
 - Programare reactivă
- Arhitectura aplicației
- Java virtual machine (JVM)
- Ecosistem web în Java
- Software și unelte laborator
- Bibliografie



JAVA – SCURT ISTORIC

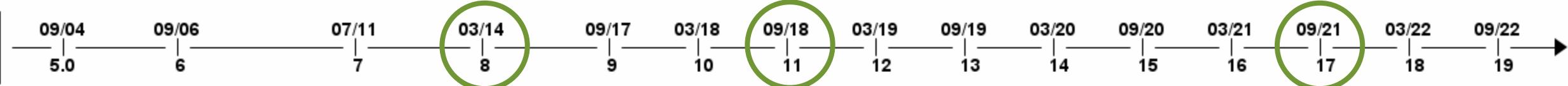
- Java 1 a apărut în 1995, fiind inspirat de C++ și Smalltalk. În 1999 a fost lansat JAVA Enterprise Edition (EE). **Versiunea curentă este 19**
- Limbaj **orientat pe obiecte**
- **Independent de platformă** – compilează codul la bytecode, care este apoi interpretat de Java Virtual Machine (JVM)
- **Securitate** – bytecode-ul rulează în JVM și nu poate accesa memoria sistemului.
- Nu are pointeri - garbage collector (GC) eliberează memoria
- Limbaj important pentru dezvoltarea de **aplicații web** prin tehnologiile Java EE și **mobile**
- **Multithreading** pentru performanță ridicată



JAVA – SCURT ISTORIC

JAVA SE

- De la versiunea 1.4 evoluția JAVA este guvernăt de **Java Community Process (JCP)** care folosește **Java Specification Requests (JSRs)** pentru adăugiri sau modificări la platforma Java
- În 2006 Sun Microsystems a făcut publice, sub licență free software/open-source (FOSS) cea mai mare parte din JVM
- Oracle a preluat Java în 2010
- **Oracle Java SE** asigură suport de lungă durată pentru versiunile 8, 11 și respectiv 17
- **OpenJDK** este o implementare importantă a Java SE licențiată ca GNU/GPL



JAVA – SCURT ISTORIC

JAKARTA EE

- Java 2 Platform Enterprise Edition (J2EE) a apărut cu versiunea 1.2 și a devenit Java Platform, Enterprise Edition (Java EE) în versiunea 5
- Java EE a fost menținută de Oracle prin JCP
- În 2017 Oracle a transferat Java EE către Eclipse Foundation
- Pentru că au apărut litigii legate de folosirea spațiului de nume javax, Java EE a fost redenumită Jakarta EE
- Versiunea curentă este Jakarta EE 10 ieșită în 2022
- Specificații:
 - Web – Servlets, servicii REST, servicii web XML, JSON, XML
 - Enterprise - Contexts and Dependency Injection (CDI), EJB, Persistence (JPA), Transactions (JTA), Messaging (JMS)



PARADIGME DE PROGRAMARE ÎN JAVA

- Paradigmă de programare – set de concepte, principii și reguli
- Java implementează mai multe paradigme:
 - **Imperativ**
 - Orientat pe obiecte (OOP)
 - **Declarativ**
 - Funcțional
 - Asincron
 - Reactiv



PROGRAMARE IMPERATIVĂ

- Codul constă dintr-un set de instrucțiuni, executate secvențial de procesor, care schimbă starea programului, stare ce este alcătuită din variabilele stocate în memorie
- Instrucțiunile dictează modul de execuție a programului
- Exemplu – Indicații pentru a ajunge de Gara de Nord la ATM
 - De la Gara de Nord iei metroul până la stația Piața Unirii
 - Iei tramvaiul 32 de la Piața Unirii până la Piața Chirigiu
 - Mergi pe jos câteva sute de metrii, pe lângă linia de tramvai, în sensul spre Centru



PROGRAMARE ORIENTATĂ PE OBIECTE (OOP)

- OOP aparține paradigmiei **imperative**
- Obiecte și clase
 - Clasele reprezintă **șabloane** ale obiectelor care conțin atât **proprietățile** cât și **comportamentul** acestora
 - Obiectele reprezintă **particularizări**/instanțe ale claselor
 - Obiectele din lumea reală se reprezintă natural prin OOP
- Principii fundamentale OOP
 - **Încapsulare** – atât proprietățile cât și comportamentul sunt conținute în obiect
 - **Abstractizare** – expune ce face obiectul și nu cum face
 - **Moștenire** – permite construirea unui obiect pe baza altuia părinte
 - **Polimorfism** – obiecte de tipuri diferite folosesc aceeași interfață



PROGRAMARE ORIENTATĂ PE OBIECTE (OOP)

ÎNCAPSULARE

- **Proprietățile** (atributele) și **comportamentul** (metodele) sunt legate de obiect și păstrate cu acesta
- Facilitează extinderea obiectelor prin moștenire și asigură o mențenanță mai ușoară a codului
- Permite controlul accesului la atributele și metodele obiectului prin specificalor de acces
- Permite decuplarea modulelor și dezvoltarea independentă a codului



PROGRAMARE ORIENTATĂ PE OBIECTE (OOP)

ABSTRACTIZARE

- Pune la dispoziție mecanisme care expun **ce face obiectul și nu cum face**
- Exemplu de abstractizare din realitate – mașina
- Ca să conduci o mașină nu trebuie să ști cum este făcută și nici toate componentele de sub capotă
 - Starea mașinii (proprietățile) este afișată pe bord
 - Comportamentul mașinii este controlat prin volan, pedale, schimbător de viteză, frână
- Abstractizarea este strâns legată de încapsulare (chiar o include)

PROGRAMARE ORIENTATĂ PE OBIECTE (OOP)

MOȘTENIRE

- Posibilitatea de a crea un obiect pe baza altuia
- Clasa părinte conține comportamentul cât mai general al obiectului
- Fiecare obiect creat pe baza unei subclase, moștenește proprietățile și comportamentul clasei părinte precum și pe cele adăugate local
- Obiectele create pe baza unei subclase, moștenesc și pot accesa toate atributele și metodele clasei părinte pe baza specificatorilor de acces
- Exemplu: clasa **Mașină** moștenește clasa **Vehicul**
 - Un obiect de tip **Mașină** va avea acces la toate atributele și metodele clasei **Vehicul** dacă aceștia au specificatorii de acces corespunzători

PROGRAMARE ORIENTATĂ PE OBIECTE (OOP)

MOȘTENIRE

```
src > Vehicle.java > ...
1  public class Vehicle {
2      private String name;
3
4      public Vehicle(String name){
5          this.name = name;
6      }
7
8      @Override
9      public String toString() {
10         return "Vehicle [name=" + name + "]";
11     }
12
13 }
14 }
```

```
:> Car.java > Car
1  public class Car extends Vehicle {
2      public Car(String name){
3          super(name);
4      }
5  }
6 }
```

```
src > App.java > ...
1  public class App {
2      Run | Debug
3      public static void main(String[] args) throws Exception {
4          Vehicle car = new Vehicle("Ford Focus");
5          System.out.println(car.toString());
6      }
7  }
```



PROGRAMARE ORIENTATĂ PE OBIECTE (OOP)

POLIMORFISM

- Oferă opțiunea de a folosi aceeași interfață pentru obiecte de tipuri diferite
- Există 2 tipuri de polimorfism: **la compilare** și **la rulare**

Polimorfism la compilare

- O clasă **Formă** are 2 metode cu același nume - **aria**:
 - Una acceptă un singur argument și returnează aria cercului
 - Cealaltă acceptă 2 parametrii, lungimea și lățimea și returnează aria unui dreptunghi
- **Compilatorul** decide pe baza numărului de parametrii ce metodă să utilizeze

PROGRAMARE ORIENTATĂ PE OBIECTE (OOP)

POLIMORFISM

Polimorfism la rulare

- Apare când o subclasă moștenește o clasă părinte și îi suprascrie metodele
- Compilatorul nu poate decide dacă metoda din clasa părinte sau metoda din subclasă va fi executată
- Acest tip de polimorfism se numește **polimorfism de subtip**
- Decizia privind metoda care se execută este luată la execuția programului

PROGRAMARE ORIENTATĂ PE OBIECTE (OOP)

POLIMORFISM

```
src > Vehicle.java > ...
1  public class Vehicle {
2      protected String name;
3
4      public Vehicle(String name){
5          this.name = name;
6      }
7
8      @Override
9      public String toString() {
10         return "Vehicle [name=" + name + "]";
11     }
12
13 }
14
```

```
src > Car.java > Car
1  public class Car extends Vehicle {
2      public Car(String name){
3          super(name);
4      }
5
6      @Override
7      public String toString() {
8          return "Car [name=" + this.name + "]";
9      }
10
11 }
12
13 }
```

```
src > App.java > App > main(String[])
1  public class App {
2      Run | Debug
3      public static void main(String[] args) throws Exception {
4          Vehicle vehicle = new Vehicle("Vehicle");
5          System.out.println(vehicle.toString());
6          Vehicle car = new Car("Ford Focus");
7          System.out.println(car.toString());
8      }
9  }
```

```
Vehicle [name=Vehicle]
Car [name=Ford Focus]
PS D:\dev\java\paradigms>
```

PROGRAMARE DECLARATIVĂ

- Este o paradigmă de programare care spune **programului ce să facă, nu cum să facă**
- Exemple de limbi pur declarative **SQL** și **XPATH**
- **Limbajele declarative** sunt abstracte, nu modifică starea programului ci creează o stare nouă
- Sunt apropiate de logica matematică
- **Programare funcțională** este un exemplu de programare declarativă
- Exemplu: În paradigma declarativă, în loc să spui prietenului cum să ajungă la ATM îi dai adresa și îl lași pe el să decidă cum ajunge



PROGRAMAREA FUNCȚIONALĂ

- Spre deosebire de programarea imperativă, nu schimbă starea internă a programului
- În programarea imperativă (OOP în Java), funcțiile/metodele depind de starea internă a programului și pot să o modifice
- În programarea funcțională, funcțiile sunt similare funcțiilor din matematică, rezultatul unei funcții depinde numai de argumente, indiferent de starea programului
 - Starea programului nu este afectată de execuția funcției
 - Populară în ultimul timp, deoarece permite procesarea paralelă (a nu se confunda cu multithreading-ul)



PROGRAMAREA FUNCȚIONALĂ (FP)

- Punctul de plecare, **calculul Lambda** dezvoltat de matematicianul Alonzo Church în 1930
- Limbajul LISP a implementat pe principiile FP
- În FP, programele sunt construite prin compunerea funcțiilor și nu prin obiecte
- Compunerea funcțiilor promovează **imutabilitatea**, evită partajarea și modificarea datelor și efectele secundare
- Codul este mai concis și mai ușor de întreținut
- **Predictibilitatea** este un alt beneficiu. Aceeași funcție cu aceleași argumente va genera același rezultat
- Permite **paralelizarea** și îmbunătățirea performanței prin stocarea rezultatului funcției după prima execuție și utilizarea acestuia la execuțiile ulterioare ale funcției

PROGRAMAREA FUNCȚIONALĂ (FP)

- Concepte și principii:
 - Lambda expressions
 - Pure functions
 - First-class functions
 - Higher-order functions
 - Function composition
 - Currying
 - Closure
 - Immutability

PROGRAMAREA FUNCȚIONALĂ (FP)

LAMBDA EXPRESSIONS

- **Expresia lambda** este un bloc de cod care primește parametrii și returnează o valoare
- Exemplu de expresie lambda pentru calculul pătratului ariei cercului

$$(x, y) \rightarrow x^2 + y^2$$

- Avantajul funcțiilor lambda față de instrucțiuni este că acestea pot fi compuse și reduse la forme mai simple
- Au fost introduse în Java 8, deși existau anterior sub forma claselor anonte
- Utilizarea funcției lambda

$$((x, y) \rightarrow x^2 + y^2)(1, 2) = 5$$



PROGRAMAREA FUNCȚIONALĂ (FP)

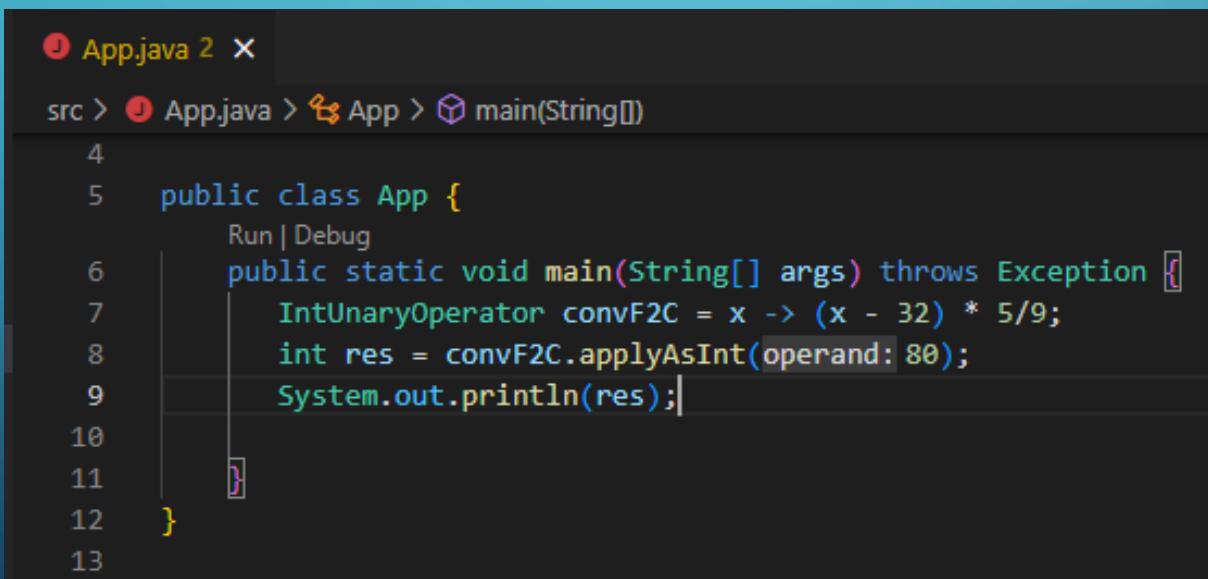
PURE FUNCTIONS

- O **funcție pură** este o funcție care nu produce efecte secundare, iar rezultatul funcției este același pentru aceleași argumente
- Efect secundar este o acțiune care modifică contextul exterior funcției
- Efectele secundare nu pot fi întotdeauna evitate – exemplu: operații de tip I/O
- În FP, se recomandă izolarea funcțiilor cu efecte secundare de restul codului
- Deoarece rezultatul unei funcții pure este predictibil, acesta poate fi înlocuit cu rezultatul stocat

PROGRAMAREA FUNCȚIONALĂ (FP)

FIRST-CLASS FUNCTIONS

- Sunt funcții care pot fi tratate asemănător obiectelor din OOP – pot fi create, stocate, folosite ca parametrii sau ca valori returnate
- Expresiile **lambda** sunt exemple de funcții de **primă clasă**
- Expresia lambda din exemplu face conversia din Farenheit la Celsius



The screenshot shows a Java code editor with the following code:

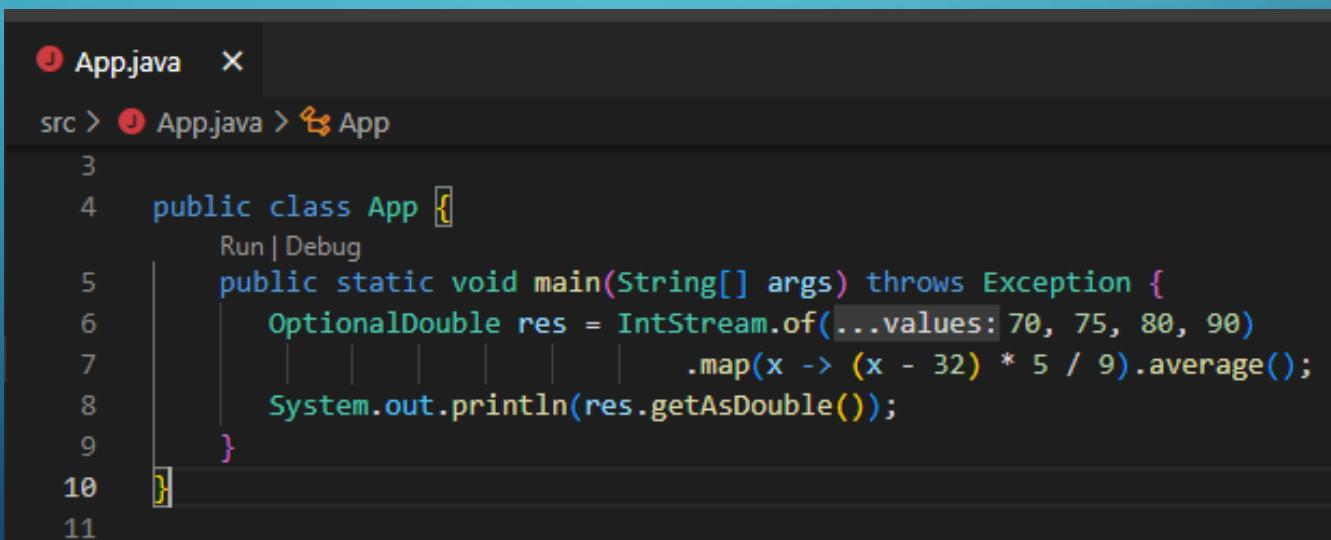
```
App.java 2 ×  
src > App.java > App > main(String[])  
4  
5  public class App {  
6      Run | Debug  
7      public static void main(String[] args) throws Exception {  
8          IntUnaryOperator convF2C = x -> (x - 32) * 5/9;  
9          int res = convF2C.applyAsInt(operand: 80);  
10         System.out.println(res);  
11     }  
12 }  
13
```

Rezultat: 26

PROGRAMAREA FUNCȚIONALĂ (FP)

HIGHER-ORDER FUNCTIONS

- Sunt funcții care pot lua alte funcții ca parametrii, pot crea alte funcții și le pot returna ca parametrii
- Exemplu de folosire a unei expresii lambda într-o funcție de ordin înalt
Expresia lambda face conversia din Farenheit la Celsius



The screenshot shows a Java code editor with a file named App.java. The code defines a class App with a main method. Inside the main method, there is a line of code that uses Java's Stream API to map an array of integers to a stream of doubles, then calculates the average of those doubles. The resulting OptionalDouble object is then printed to the console.

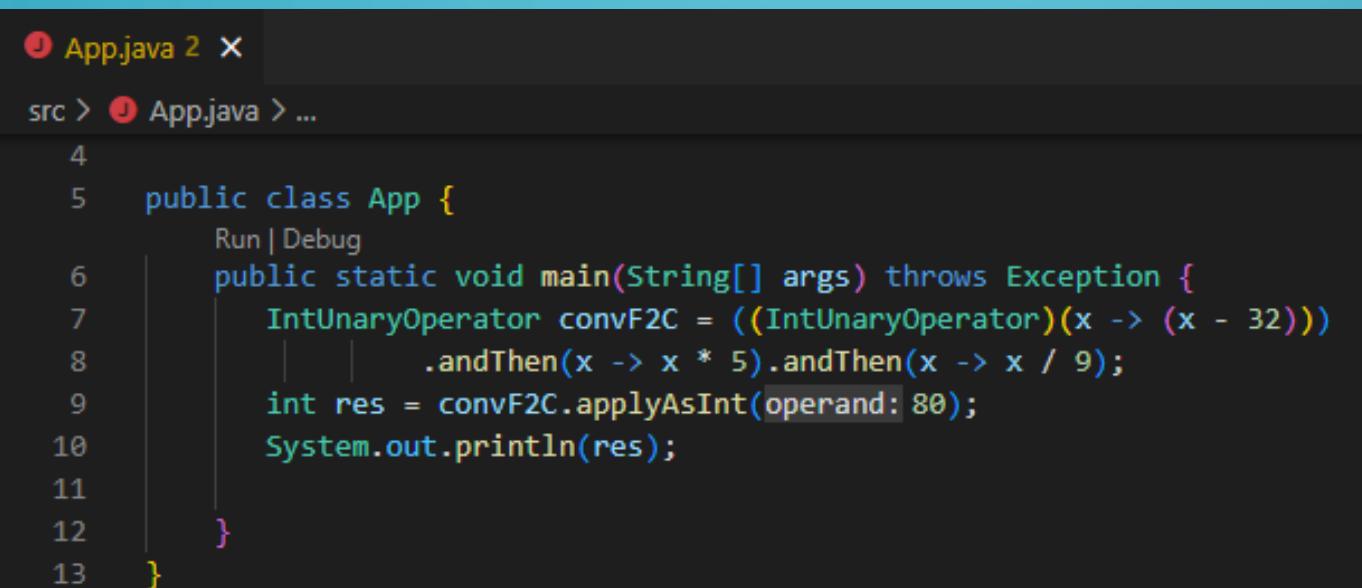
```
App.java
src > App.java > App
3
4 public class App {
5     Run | Debug
6     public static void main(String[] args) throws Exception {
7         OptionalDouble res = IntStream.of(...values: 70, 75, 80, 90)
8             .map(x -> (x - 32) * 5 / 9).average();
9         System.out.println(res.getAsDouble());
10    }
11}
```

Rezultat: 25.5

PROGRAMAREA FUNCȚIONALĂ (FP)

FUNCTION COMPOSITION

- În matematică funcțiile se compun folosind rezultatul uneia ca parametru în cealaltă
- În FP, funcțiile de primă clasă sunt folosite de funcțiile de ordin înalt



The screenshot shows a Java code editor with the file `App.java` open. The code defines a class `App` with a `main` method. Inside the `main` method, there is a line of code that uses the `IntUnaryOperator` interface to perform function composition. The code is as follows:

```
4
5  public class App {
6      Run | Debug
7      public static void main(String[] args) throws Exception {
8          IntUnaryOperator convF2C = ((IntUnaryOperator)(x -> (x - 32)))
9              .andThen(x -> x * 5).andThen(x -> x / 9);
10         int res = convF2C.applyAsInt(operand: 80);
11         System.out.println(res);
12     }
13 }
```

Rezultat: 26



PROGRAMAREA FUNCȚIONALĂ (FP)

CURRYING

- Este procesul de transformare a unei funcții de ordin n într-o serie de funcții unare (cu un singur parametru)
- Exemplu pentru o funcție cu 2 parametrii $f(x, y) = x^2 + y^2$

The screenshot shows a Java code editor with two files open: App.java and Pair.java. The current file is App.java, which contains the following code:

```
12 public final class App {  
13     private App() {  
14     }  
15     public static void main(String[] args) {  
16         Function<Integer, Function<Integer, Integer>> sqrad = x -> y -> x*x + y*y;  
17         List<Integer> res = Arrays.asList(new Pair<Integer, Integer>(x: 1, y: 2),  
18                                         new Pair<Integer, Integer>(x: 4, y: 5)).stream()  
19                                         .map(a -> sqrad.apply(a.y).apply(a.x))  
20                                         .collect(Collectors.toList());  
21         System.out.println(res);  
22     }  
23 }  
24 }
```

The code defines a function `sqrad` that takes an integer `x` and returns a function that takes an integer `y` and returns the sum of their squares. This is achieved using Java's functional interface support and streams.

Rezultat: [5, 41]

PROGRAMAREA FUNCȚIONALĂ (FP)

CLOSURE

- Closure – funcția este stocată cu contextul
- Este o tehnică ce permite implementarea scopului lexical
- Scopul lexical permite accesarea variabilelor dintr-un context extern într-un context local

App.java 3 ● MyInt.java

src > main > java > atm > paradigms > App.java > App > main(String[])

```

13     private App() {
14 }
15     Run | Debug
16     public static void main(String[] args) {
17         final MyInt a = new MyInt(val: 100);
18         Function<Integer, Integer> add100 = x -> x + a.getval();
19         System.out.println(add100.apply(t: 9));
20         a.setval(val: 101);
21         System.out.println(add100.apply(t: 9));
22     }

```

App.java 3 × MyInt.java ×

src > main > java > atm > paradigms > MyInt.java > MyInt.java

```

1 package atm.paradigms;
2
3 public class MyInt {
4     private Integer val;
5
6     public MyInt(Integer val) {
7         this.val = val;
8     }
9
10    public Integer getval() {
11        return val;
12    }
13
14    public void setval(Integer val) {
15        this.val = val;
16    }
17
18
19 }
20

```

Rezultat:
109
110



PROGRAMAREA FUNCȚIONALĂ (FP)

IMMUTABILITY

- Promovarea imutabilității simplifică codul și evită efectele secundare
- Codul mutabil devine complex și este dificil de înțeles și corectat
- Codul mutabil este dificil de paralelizat



PROGRAMARE REACTIVĂ

- Este un stil de programare focalizat pe reacția la schimbări, care pot fi valori ale datelor sau evenimente
- Este o paradigmă de programare bazată pe **fluxuri** (streams) **asincrone** de date în scopul capturării actualizărilor la date
- Fluxurile de date sunt create continuu sau aproape continuu de către o **sursă** (exemple: senzor de mișcare, senzor de temperatură sau inventarul unui produs din baza de date)
- Răspunsul unui endpoint REST poate fi tratat ca un flux de date pe care se așteaptă, poate fi filtrat sau poate fi combinat cu răspunsul de la un alt endpoint
- Definirea unui flux de date este similar cu o foaie de date din Excel unde celula C1 este suma celulelor A1 și B1
- Când valorile celulelor A1 sau B1 sunt actualizate, modificarea este **observată** și se **reacționează** la ea prin actualizarea valorii celulei C1



PROGRAMARE REACTIVĂ

- Abstracții utilizate în programarea reactivă:
 - **Futures/promises** – mecanisme de acționare asupra unor date ce vor fi disponibile în viitor
 - **Streams** – fluxuri de date pe care se bazează programarea reactivă
 - **Dataflow operators** – rezultatul funcțiilor aplicate pe fluxuri de date
 - **Backpressure** – mecanism de reglare a vitezei de procesare a fluxurilor de date prin eliminarea unor elemente din flux
 - **Push mechanism** – când datele sunt disponibile este notificat observatorul relevant, care va procesa datele primite
- Vom utiliza RxJava 2



ARHITECTURA APLICAȚIEI

- Arhitectura aplicației sau **designul** nu trebuie să se bazeze doar pe cerințele actuale, ci trebuie să anticipateze și să ia în considerare eventuale modificări viitoare
- În afara **cerințelor funcționale**, există **cerințe nefuncționale** care trebuie luate în considerare de către arhitect: performanța, scalabilitatea, securitatea, mențenanța, disponibilitatea, dezvoltare ulterioară
- Arhitectura sau designul nu sunt universal valabile, arhitectura unei aplicații bancare poate fi diferită de o aplicație de comerț online (e-commerce)
- În cadrul unei aplicații, componentele pot urma diferite abordări de design
- O componentă poate schimba date folosind REST endpoints, iar o alta se poate baza pe cozi de mesaje (message queues)

ARHITECTURA APLICAȚIEI

TIPURI

- **Layered architecture** – aplicație tradițională Java EE
- **Model View Controller (MVC)** – popular cu Spring dar și în aplicații Android
- **Microservice architecture** - popular în cloud (Quarkus, String etc.)
- **Serverless architecture** – nu implică software server iar aplicațiile sunt executate ca funcții individuale de către provider-ii de cloud

ARHITECTURA APLICAȚIEI

LAYERED ARCHITECTURE

- Aplicația este împărțită **pe straturi**, fiecare având o responsabilitate bine stabilită
- Numărul de straturi depinde de proiect
- Exemplu – straturile care alcătuiesc o aplicație web sunt:
 - Stratul de prezentare
 - Stratul controller/serviciu web
 - Stratul aplicație
 - Stratul de business
 - Stratul de acces la date
- **Stratul de prezentare** – conține interfața utilizator, dezvoltată în HTML/JavaScript/JSP etc.
- **Stratul controller/serviciu web** – punct de intrare pentru cererile HTTP(S) de la client, sau de la un alt serviciu. În acest strat se fac filtrări inițiale asupra cererilor, validări, îndeplinirea cerințelor de securitate precum autentificare și autorizare.



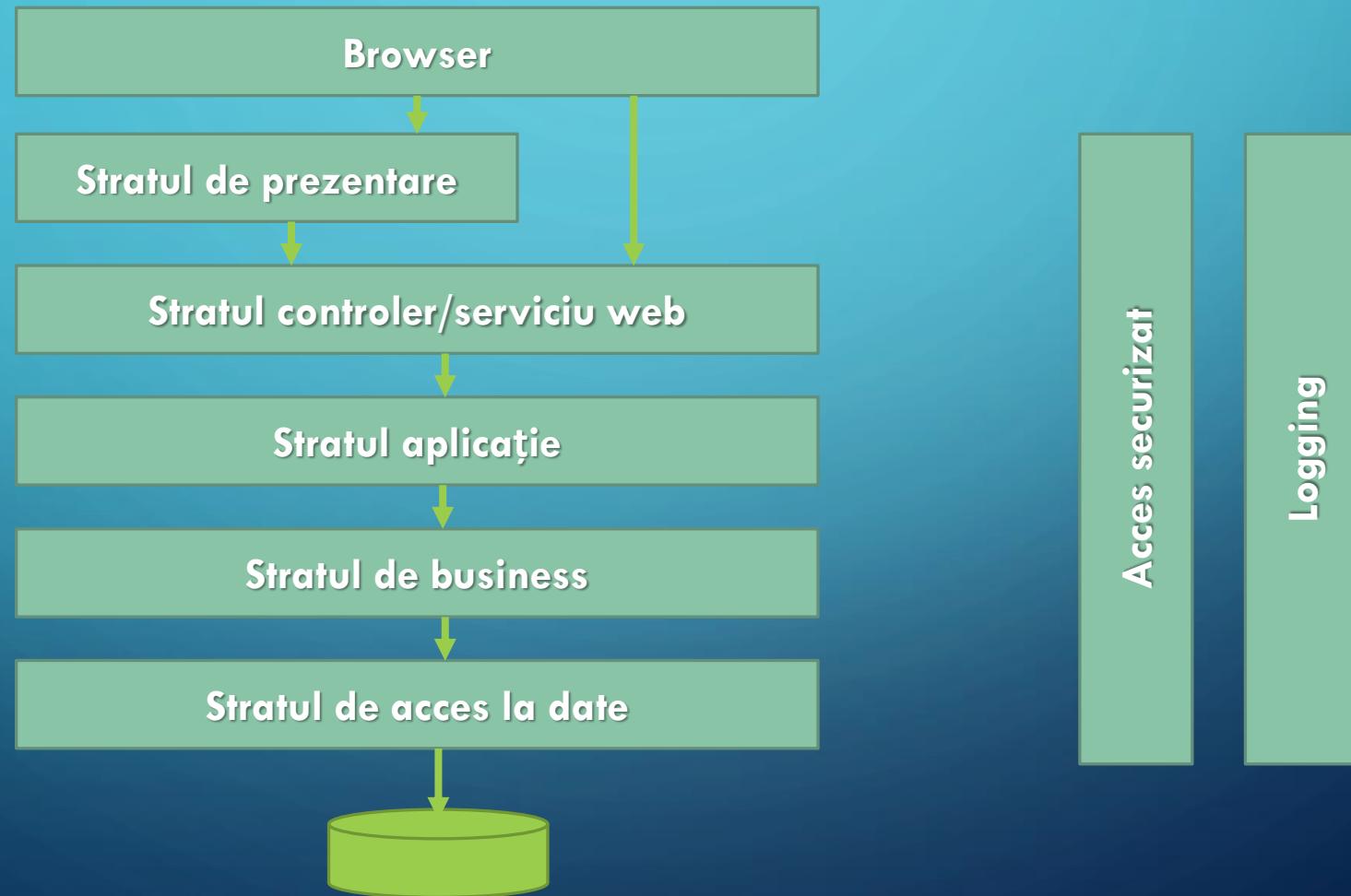
ARHITECTURA APLICAȚIEI

LAYERED ARCHITECTURE

- **Stratul aplicație** – responsabil cu alte servicii precum trimitera de emailuri, descărcarea de fișiere, generarea de rapoarte, administrare aplicație etc. Pentru aplicații mici poate fi comasat cu stratul controller
- **Stratul de business** – conține logica aplicației. La acest nivel se verifică toate regulile și constrângerile necesare. Se poate diviza în mai multe substraturi
- **Stratul de acces la date** – responsabil cu toate operațiile legate de date precum aducerea datelor și reprezentarea în formatul cerut, stocarea datelor, actualizarea și ștergerea acestora
- Avantaje: Code organization, Ease in deployment
- Provocări: Deployment, Testability

ARHITECTURA APLICAȚIEI

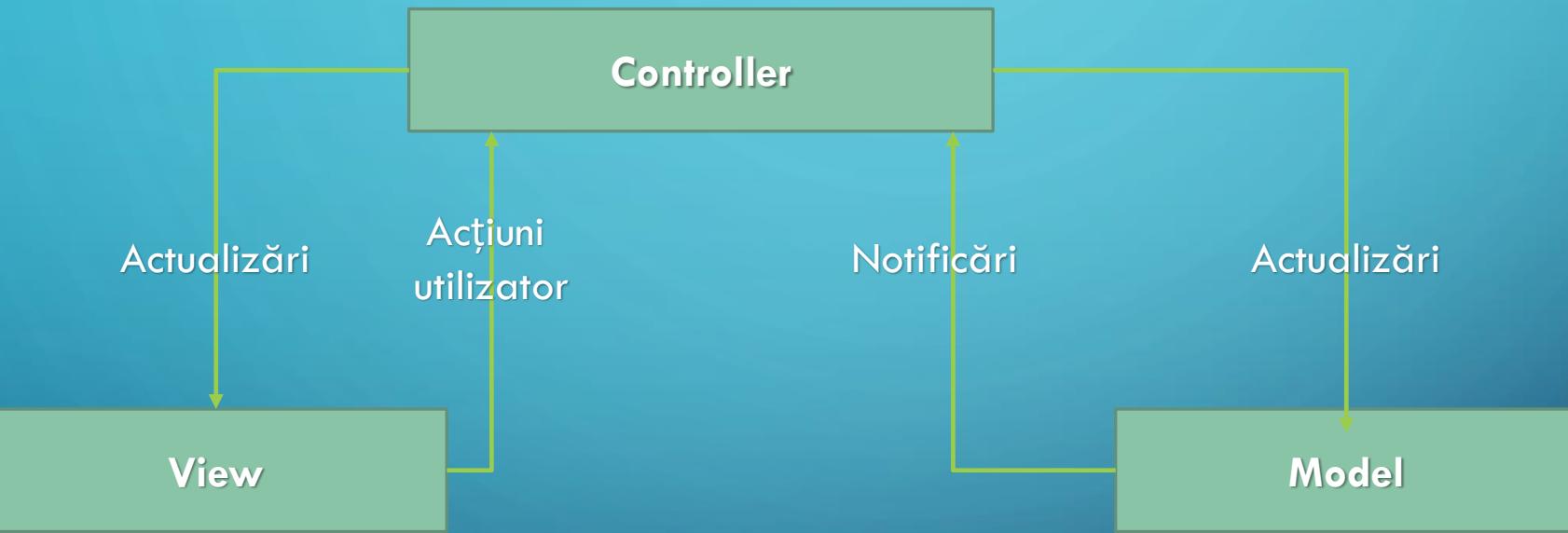
LAYERED ARCHITECTURE



ARHITECTURA APLICAȚIEI MODEL VIEW CONTROLLER (MVC)

- Aplicația este organizată pe 3 componente independente cu responsabilități clare:
- **Model** – responsabil cu organizarea și implementarea logicii necesare manipulării și modificării datelor. Tratează evenimente generate de modificările datelor. Este componenta care conține logica aplicației
- **View** - reprezintă interfața client. Este responsabil cu afișarea informației la client și preluarea inputurilor acestuia
- **Controller** – controlează fluxul de date. Când o acțiune are loc la View, Controller-ul este notificat de View iar acesta va determina dacă transmite mai departe către Model sau View
 - Avantaje: Separation of concerns, Ease in deployment
 - Provocări: Scalability, Testability

ARHITECTURA APLICAȚIEI MODEL VIEW CONTROLLER (MVC)



ARHITECTURA APLICAȚIEI SERVICE-ORIENTED ARCHITECTURE (SOA)

- Aplicația este alcătuită din diferite servicii independente care pot fi reutilizate
- Serviciile pot fi dezvoltate, testate, instalate (deployed) și actualizate independent unul de altul
- Chiar dacă un serviciu este nefuncțional celelalte funcționează normal
- Aceste pot fi dezvoltate în limbiage diferite
- Serviciile pot interacționa unul cu altul și de multe ori folosesc în comun resurse partajate baze de date sau spațiu de stocare
- SOA se implementează prin servicii web
- Cele mai comune servicii web sunt Simple Object Access Protocol (SOAP) și Representational State Transfer (REST)
- Avantaje: Ease in deployment, Loose coupling, Testability
- Provocări: Deployment, Scalability

ARHITECTURA APLICAȚIEI

MICROSERVICE ARCHITECTURE

- Structurează aplicația ca o colecție de servicii:
 - Ușor de actualizat și testat
 - Slab cuplate
 - Instalate independent
 - Organizate în jurul logicii aplicației
- Serviciile din SOA pot și creațe ca o colecție de microservicii
- Gradul de granularitate a microserviciilor reprezintă o provocare
- Comunicarea între microservicii se face fie prin REST fie folosind cozile de mesaje (message queueing)
- Avantaje: Scalability, Continuous delivery, Ease in deployment, Testability
- Provocări: Dependency on devops, Maintaining the balance, Repeated code

ARHITECTURA APLICAȚIEI

SERVERLESS ARCHITECTURE

- Arhitecturile prezentate au un factor în comun: depind de infrastructură
- **Devops** asigură funcționarea normală a serviciilor, instalarea și scalare acestora, precum și monitorizarea resurselor disponibile
- Pentru degrevarea echipelor de dezvoltare de problemele de mai sus a apărut arhitectura **serverless**
- Echipa de dezvoltare se ocupă exclusiv de logica aplicației (business), iar furnizorii de servii de cloud se ocupă de necesarul de infrastructură, inclusiv scalarea aplicației

JAVA VIRTUAL MACHINE (JVM)

JUST-IN-TIME (JIT)

- Inițial Java a pornit ca un limbaj interpretat
- Fișierul sursă cu extensia `.java`, prin rularea `javac` generează fișierul `.class` conținând **bytecode**
- Prin rularea codului cu `java yourcode.class`, Java Runtime Environment (JRE) convertește codul în instrucțiuni specifice sistemului de operare. Acest pas se execută ori de câte ori se rulează codul
- JVM moderne recunosc codul rulat de mai multe ori și îl marchează ca **hotspot**
- JVM execută compilarea Just-In-Time (JIT) pentru porțiunile de cod **hotspot**
- Părțile aplicației compilate cu JIT se execută mult mai rapid decât partea interpretată

JAVA VIRTUAL MACHINE (JVM)

AHEAD-OF-TIME COMPILATION (AOT)

- Compilarea **AOT** ia fișierul .java și îl convertește în **cod binar nativ** ce poate fi executat imediat de sistemul de operare
- Aplicația pornește mult mai rapid iar memoria necesară se reduce semnificativ (comparabil ca performanță cu codul C++)
- Codul compilat AOT poate deveni un executabil de sine stătător care poate fi rulat fără JVM
- Codul Java trebuie să fie mult simplificat
- **GraalVM** este un JVM de performanță înaltă care suportă compilarea AOT și mai multe limbiage de programare în afara de Java (Scala, Kotlin, R, JavaScript, Python)

ECOSISTEM WEB ÎN JAVA

Servlet/Web Container

- Gestionează *servlets*, *filters*, *listeners* organizați cu o aplicație web
- Servlets – sunt clase Java care acceptă, procesează și transmit un răspuns HTTP
- Exemple: Tomcat, Grizzly, Jetty

Application Server

- Implementează specificațiile Jakarta EE
- Conține atât un container EJB cât și un container web
- Exemple: Glassfish, Jboss, GlassFish

Application framework

- Este o colecție de biblioteci care implementează părți din specificațiile Jakarta EE
- Se folosește pentru dezvoltarea de aplicații web și oferă programatorului un control mai mare asupra codului
- Exemple: Spring, Quarkus

SOFTWARE ȘI UNELTE LABORATOR

- **Java SE 11 și Java SE 17**
- **Visual Studio Code (VSC)** – IDE realizat de Microsoft, bazat pe *Electron framework* și cu suport pentru: debugging, syntax highlighting, intelligent code completion, snippets, code refactoring

Extensiile VSC:

- **Extension Pack for Java** (suport pentru Java, debugger, test runner, Maven, project manager, IntelliCode)
- **Community Server Connectors** (suport Tomcat)
- **Quarkus**
- Baza de date: **MySQL 8**



SOFTWARE ȘI UNELTE LABORATOR

- **Maven** – este o *unealtă open source* dezvoltată de Apache Group pentru **compilarea, împachetarea, publicarea și instalarea** proiectelor

Caracteristici Maven:

- O colecție (repository) uriașă de biblioteci menținută la zi
- Posibilitatea de a crea proiecte folosind cele mai bune practici și şablonane (artefacts)
- Managementul automat al dependențelor
- Versionare automată
- Verificarea integrității și raportarea erorilor

SOFTWARE ȘI UNELTE LABORATOR

- **Maven** este folositor mai ales datorită **Project Object Model (POM)** care este un fișier XML ce conține:
 - Descrierea proiectului
 - Detalii de versionare
 - Managementul configurației proiectului
- **pom.xml** este localizat în rădăcina proiectului
- **Maven** este folosit în proiectele Java pentru descărcarea dependențelor corecte pentru proiect (fișiere JAR)

BIBLIOGRAFIE

- **Design Patterns and Best Practices in Java**, Kamalmeet Singh, Adrian Ianculescu, Lucian-Paul Torje, 2018 Packt Publishing
- Functional Programming And Programming Paradigms in Java
- Wikipedia - Functional programming
- What are microservices?
- Wikipedia- Serverless computing
- GravitVM