



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

# PARADIGME DE PROGRAMARE (ÎN JAVA)

## CURS 7 - PROGRAMARE FUNCȚIONALĂ

COL(R) TRAIAN NICULA

# TEMATICĂ DE CURS

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

# CUPRINS CURS

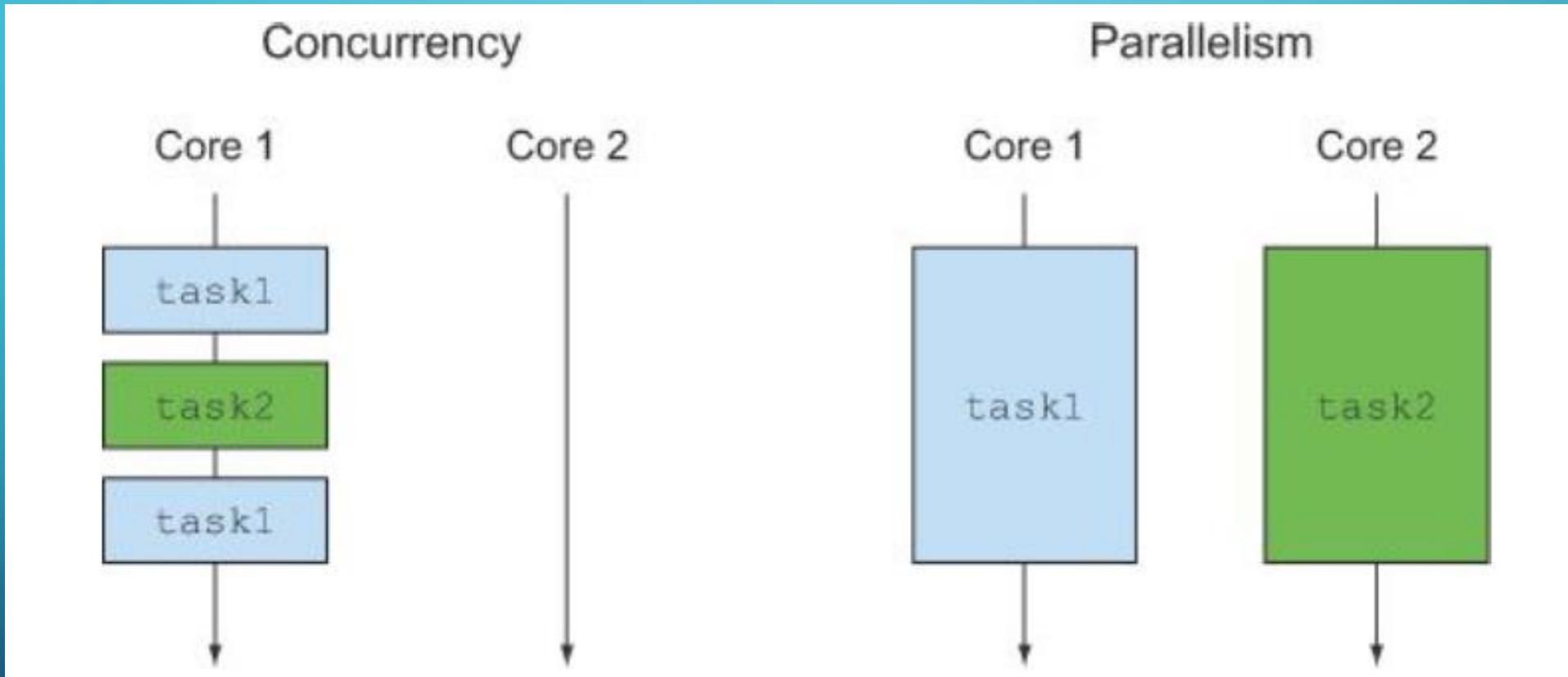
- Programare asincronă
- Noul API pentru date și timp
- Tehnici de programare funcțională
- Bibliografie



# PROGRAMARE ASINCRONĂ

- O altă **tendință** în dezvoltarea **software** (în afara evoluției hardware) o reprezentă **disponibilitatea și integrarea serviciilor de Internet** accesibile prin API publice
- În multe situații o aplicație web trebuie să agrege informații din mai multe surse (PayPal, Google Translate, Google Maps, OSM, aplicații de Geocoding etc.)
- **Serviciile Internet nu pot fi implementate în mod blocare/așteptare răspuns**
- O abordare o reprezintă **programarea paralelă multitasking** folosind fire de execuție, dar și în acest caz acestea se blochează în așteptarea răspunsului
- În acest caz **concurența** implementată prin interfața **CompletableFuture este cea mai bună abordare**

# PROGRAMARE ASINCRONĂ



# PROGRAMARE ASINCRONĂ

## INTERFAȚA FUTURE

- Interfața **Future** a fost introdusă în Java 5 pentru a **modela returnarea unui rezultat la un anumit moment în viitor**
- Reprezintă un **calcul asincron** care **returnează o referință la un rezultat ce va disponibil la finalizarea calculului**
- **Firul de execuție** care lansează o acțiune consumatoare de timp printr-un **Future poate continua să fac altceva** până la finalizarea sarcinii
- Pentru a lucra cu **Future** este nevoie ca operațiunea consumatoare de timp **să fie inclusă într-un obiect Callable și delegată unui ExecutorService**
- Acest stil de programare permite thread-ului principal să execute alte operații în timp ce **operația de lungă durată se execută într-un thread separat** creat de **ExecutorService**



# PROGRAMARE ASINCRONĂ

## INTERFAȚA FUTURE

### *Runnable vs Callable*

- **Runnable** este o interfață funcțională cu o singură metodă **run()**
- Se folosește pînă thread-uri care nu returnează

```
public interface Runnable {  
    public void run();  
}
```

- **Callable** este o interfață generică cu o singură metodă **call()** care returnează o valoare generică
- Rezultatul metodei **call()** este returnat ca **Future**

```
public interface Callable<V> {  
    V call() throws Exception;  
}
```



# PROGRAMARE ASINCRONĂ

## INTERFAȚA FUTURE

[Cuprins](#)

```
package atm.paradigms.tests;

import java.util.concurrent.*;

public class FutureTest {
    public static void main(String[] args) {
        ExecutorService executor = Executors.newCachedThreadPool();
        Future<Long> future = executor.submit(new Callable<Long>(){
            @Override
            public Long call() throws Exception {
                return doLongComputation();
            }
        });
        try {
            doSomethingElse();
            Long result = future.get(6, TimeUnit.SECONDS);
            System.out.println("Got result: " + result);
            System.exit(0);
        } catch (InterruptedException | ExecutionException | TimeoutException e) {
            e.printStackTrace();
        }
        ...
    }
}
```

Vezi proiect [course7](#)

Se creează o instanță **ExecutorService** către care se trimit o clasă anonimă **Callable**, ce returnează un **Future**. Acesta se va executa pe un thread separat ca o operație de durată. Thread-ul principal execută o altă operație, apoi așteaptă rezultatul de la **Future** cu metoda **get**. Metoda **get** se blochează în așteptarea rezultatului, pentru maxim 6s (timeout).

# PROGRAMARE ASINCRONĂ

## INTERFAȚA FUTURE

```
public static long doLongComputation() throws InterruptedException{
    Thread.sleep(5000);
    System.out.println("Long task done!");
    return 5L;
}

public static void doSomethingElse() throws InterruptedException{
    Thread.sleep(2000);
    System.out.println("Something else done!");
}
```

Rezultat:  
*Something else done!*  
*Long task done!*  
*Got result: 5*

# PROGRAMARE ASINCRONĂ

## INTERFAȚA FUTURE

### Limitări Future

- Nu permite scrierea de **cod concurrent concis**
- Este dificil de a trece rezultatul de la un *Future* la un alt *Future*
- Nu se pot combina mai multe operații asincrone într-una
- Nu știe să aștepte finalizarea a mai multe sarcini *Future*
- *Future* nu se poate opri programatic
- Nu se pot exprima declarativ operații concurente
- Toate aceste limitări se rezolvă prin **CompletableFuture** care introduce o arhitectură similară *stream* bazată pe **expresii lambda** și pe ideea de **pipeline**



# PROGRAMARE ASINCRONĂ

## CLASA COMPLETABLEFUTURE

- Clasa **CompletableFuture** implementează interfața **Future**
- Vom utiliza un exemplu de aplicație care găsește prețul ce mai mic al unui produs
- Considerăm un **API** care se implementează de fiecare magazin și expune o metodă **getPrice(String product)**. Metoda returnează prețul pentru numele unui produs dat
- Pentru a simula o operație de durată vom folosi o funcție **delay** care simulează întârzierea cu **Thread.sleep**
- Prima versiunea a API este inacceptabilă deoarece metoda implică blocarea pentru 1s (slide următor)

# PROGRAMARE ASINCRONĂ

## CLASA COMPLETABLEFUTURE

API cu blocare

```
package atm.paradigms.async;
import java.util.Random;

public class Shop {

    public double getPrice(String product){
        return calculatePrice(product);
    }
    public static void delay(){
        try {
            Thread.sleep(2000L);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    private double calculatePrice(String product){
        delay();
        Random r = new Random();
        return r.nextDouble() * product.charAt(0)
                + product.charAt(1);
    }
}
```

Vezi proiect [course7](#)

Metoda **getPrice()** se blochează în aşteptarea răspunsului.  
Prețul produsului se calculează aleator în metoda **calculatePrice()**.

# PROGRAMARE ASINCRONĂ

## CLASA COMPLETABLEFUTURE

- Vom implementa o versiune asincronă bazată pe **CompletableFuture**. Metoda **getPriceAsync** returnează **imediat** permitând thread-ului principal să facă alte calcule
- **CompletableFuture** reprezintă un **calcul asincron** al cărui rezultat va deveni disponibil mai târziu
- Se **creează un nou thread** pentru efectuarea calculelor, dar care **returnează imediat**
- **Când rezultatul este disponibil** se poate finaliza explicit **CompletableFuture** cu metoda **complete**

# PROGRAMARE ASINCRONĂ

## CLASA COMPLETABLEFUTURE

```
public Future<Double> getPriceAsync(String product){  
    CompletableFuture<Double> futurePrice = new CompletableFuture<>();  
    new Thread(() -> {  
        double price = calculatePrice(product);  
        futurePrice.complete(price);  
    }).start();  
    return futurePrice;  
}
```

Thread creat și pornit manual.  
Interfața **Runnable** se implementează  
prin expresie lambda.

- Clientul **cere magazinului să returneze prețul** unui produs
- Metoda asincronă **getPriceAsync** **returnează aproape imediat** iar prețul va fi returnat ulterior clientului
- Clientul poate executa alte sarcini în aşteptarea răspunsului
- Invocă **get** pe **Future** pentru a **obține răspunsul**, dacă este disponibil, sau se **blochează** în aşteptarea acestuia



# PROGRAMARE ASINCRONĂ

## CLASA COMPLETABLEFUTURE

```

package atm.paradigms.async;
import java.util.concurrent.Future;

public class AsyncTest {
    public static void main(String[] args) throws InterruptedException {
        Shop shop = new Shop("eMag");
        Future<Double> futurePrice = shop.getPriceAsync("laptop");
        // do something else
        Thread.sleep(2000L);
        long start = System.nanoTime();
        try {
            double price = futurePrice.get();
            System.out.printf("Price is %.2f%n", price);
        } catch (Exception e){
            e.printStackTrace();
        }
        System.out.println("Price returned after "
                + (System.nanoTime() - start)/1_000_000 + "ms");
    }
}
  
```

Returnează imediat un **Future<Double>** și trece mai departe.

Așteaptă răspunsul cu **get**. Dacă este disponibil îl afișează, dacă nu se blochează în așteptarea acestuia.

Rezultat:  
*Price is 126.35*  
*Price returned after 16ms*



# PROGRAMARE ASINCRONĂ

## CLASA COMPLETABLEFUTURE

- Ce se întâmplă cu **erorile generate în thread?** Clientul nu va mai primi răspuns și va rămâne blocat în stare de așteptare
- O variantă este să folosim **versiunea supraîncărcată a metodei get care acceptă timeout.** Clientul nu va aștepta la infinit fiind generată excepția **TimeoutException**, dar care **nu spune prea multe despre cauza erorii**
- O altă variantă este **propagarea erori** în **CompletableFuture** folosind metoda **completeExceptionally**

```
public Future<Double> getPriceAsync(String product){  
    CompletableFuture<Double> futurePrice = new CompletableFuture<>();  
    new Thread(() -> {  
        try {  
            double price = calculatePrice(product);  
            futurePrice.complete(price);  
        } catch (Exception e){  
            futurePrice.completeExceptionally(e);  
        }  
    }).start();  
    return futurePrice;  
}
```

# PROGRAMARE ASINCRONĂ

## CLASA COMPLETABLEFUTURE

- Clasa **CompletableFuture** pune la dispoziție metode statice care fac procesul mai ușor
- Metoda **supplyAsync** acceptă un **Supplier** ca argument care **va fi completat asincron cu valoare obținută** prin invocarea acelui **Supplier**

```
public Future<Double> getPriceAsync(String product){  
    return CompletableFuture.supplyAsync(() -> calculatePrice(product));  
}
```

- Metoda **returnează același rezultat cu cea anterioară** creată manual
- Pe slide-ul următor sunt prezentate câteva metode utile



# PROGRAMARE ASINCRONĂ

## CLASA COMPLETABLEFUTURE – METODE UTILE

Metodă	Descriere
<b>supplyAsync(Supplier&lt;U&gt; supplier)</b> <b>supplyAsync(Supplier&lt;U&gt; supplier, Executor executor)</b>	Primește ca parametru un Supplier<U>, care se execută asincron și returnează CompletableFuture<U>. Folosește implicit ForkJoinPool sau un Executor
<b>CompletableFuture&lt;String&gt; completableFuture = CompletableFuture.supplyAsync(() -&gt; "Hello");</b>	
<b>thenApply (Function&lt;T, U&gt; fn)</b>	Primește ca parametru o funcție, care se execută sincron și returnează CompletableFuture<U>
<b>CompletableFuture&lt;String&gt; finalResult = completableFuture.thenApply(s-&gt; s + “ world!);</b>	
<b>thenCompose(Function&lt;T, CompletableFuture&lt;U&gt;&gt; fn)</b>	Folosește rezultatul primit pentru a-l folosi într-un nou CompletableFuture<U> care depinde de acesta. Rezultatul returnat este de tip CompletableFuture<U>
<b>CompletableFuture&lt;String&gt; completableFuture = CompletableFuture.supplyAsync(() -&gt; "Hello") .thenCompose(s -&gt; CompletableFuture.supplyAsync(() -&gt; s + " World"));</b>	
<b>thenCombine(CompletableFuture&lt;U&gt; other, BiFunction&lt;T, U,V&gt; fn)</b>	Execută independent cele 2 CompletableFuture, iar când rezultatele sunt disponibile folosește BiFunction pentru a calcula un rezultat
<b>CompletableFuture&lt;String&gt; completableFuture = CompletableFuture.supplyAsync(() -&gt; "Hello") .thenCombine(CompletableFuture.supplyAsync( () -&gt; " World"), (s1, s2) -&gt; s1 + s2));</b>	



# PROGRAMARE ASINCRONĂ

## CLASA COMPLETABLEFUTURE

- Să presupunem că avem o **listă de magazine** și **implementăm metoda `findPrices(String product)`** care **returnează o listă de String-uri** conținând numele magazinului și prețul
- Prima **implementare cu blocare** (execuție secvențială: 3 magazine \* 2s fiecare)

```
public static List<String> findPricesBlocking(String product) {  
    return getShopList().stream()  
        .map(shop -> String.format("%s is %.2f",  
                                     shop.getName(), shop.getPrice(product)))  
        .collect(toList());  
}
```

Rezultat:  
[eMag is 106.31, Altex is 170.05, Flanco is 190.29]  
Done in 6055ms

```
public static List<Shop> getShopList() {  
    return Arrays.asList(new Shop("eMag"),  
                        new Shop("Altex"),  
                        new Shop("Flanco"));  
}
```

# PROGRAMARE ASINCRONĂ

## CLASA COMPLETABLEFUTURE

- Următoarea îmbunătățire pe care o putem face este să **paralelizăm calculul**

```
public static List<String> findPricesParallel(String product) {  
    return getShopList().parallelStream()  
        .map(shop -> String.format("%s is %.2f",  
                                     shop.getName(), shop.getPrice(product)))  
        .collect(toList());  
}
```

- Rezultat: [eMag is 176.52, Altex is 120.77, Flanco is 173.01]  
*Done in 2047ms*
- Acum **magazinele sunt interogate în paralel** iar răspunsul este cel așteptat în jur de 2s
- Vom transforma invocările sincrone în invocări asincrone cu **CompletableFuture**



# PROGRAMARE ASINCRONĂ

## CLASA COMPLETABLEFUTURE

- Am văzut că putem folosi metoda `supplyAsync` pentru a crea obiecte de tip **CompletableFuture**

```
List<CompletableFuture<String>> priceFutures =  
    shops.stream()  
        .map(shop -> CompletableFuture.supplyAsync(  
            () -> String.format("%s price is %.2f",  
                shop.getName(), shop.getPrice(product))))  
        .collect(toList());
```

- Codul generează `List<CompletableFuture<String>>` în care fiecare obiect **CompletableFuture** va conține String-ul dorit la finalizarea calculului
- Scopul este ca metoda `findPricesAsync` să returneze `List<String>`
- Trebuie să se finalizeze toate operațiile și să returneze lista



# PROGRAMARE ASINCRONĂ

## CLASA COMPLETABLEFUTURE

- Listei anterioare de tip `List<CompletableFuture<String>>` îi aplicăm o nouă metodă `map` care cheamă `join` pe elementele listei și așteaptă terminarea lor
- Metoda `join` a clasei `CompletableFuture` este similară cu `get` a interfeței `Future` cu diferența că nu generează excepții

```
public static List<String> findPricesAsync(String product) {
    List<CompletableFuture<String>> futurePrices = getShopList().stream()
        .map(shop -> CompletableFuture.supplyAsync(
            () -> String.format("%s is %.2f",
                shop.getName(), shop.getPrice(product))))
        .collect(toList());
    return futurePrices.stream()
        .map(CompletableFuture::join)
        .collect(toList());
}
```

- Rezultat: [eMag is 196.07, Altex is 136.79, Flanco is 194.36]  
Done in 2049ms

Se creează o listă de **CompletableFuture** care returnează imediat. Fiecare **Future** se execută pe câte un thread separat.  
Se așteaptă rezultatul cu metoda **join** aplicată cu **map** pe fiecare element din stream.

# PROGRAMARE ASINCRONĂ

## CLASA COMPLETABLEFUTURE

- Deși timpul de execuție între varianta paralelă și asincronă pare similar, **CompletableFuture** are **avantajul că permite configurarea *thread pool* pentru a satisface mai bine cerințele aplicației**
- Dacă **numărul de *thread-uri*** este prea mare se **vor afla în competiție pentru resursele CPU și memorie**, iar **performanța scade** datorită timpului necesar comutării contextului
- Având în vedere că aplicația exemplu petrece 99% din timp așteptând un rezultat (blocat), **numărul de *thread-uri* trebuie să fie egal cu numărul de magazine**
- Implementăm un **Executor** pe care îl trecem ca argument metodei **supplyAsync**

# PROGRAMARE ASINCRONĂ

## CLASA COMPLETABLEFUTURE

```
private static Executor getExecutor(){  
    return Executors.newFixedThreadPool(Math.min(getShopList().size(), 100),  
        new ThreadFactory() {  
            @Override  
            public Thread newThread(Runnable r) {  
                Thread t = new Thread(r);  
                // don't prevent termination of program  
                t.setDaemon(true);  
                return t;  
            }  
        }  
    );  
}
```

Creează un **thread pool** și stabilește unele proprietăți ale thread-urilor din acesta (setDaemon).

```
List<CompletableFuture<String>> futurePrices = getShopList().stream()  
    .map(shop -> CompletableFuture.supplyAsync(  
        () -> String.format("%s is %.2f",  
            shop.getName(), shop.getPrice(product)), getExecutor()))  
    .collect(toList());
```

Invocă **supplyAsync** cu **thread pool** creat anterior.

# PROGRAMARE ASINCRONĂ

## CLASA COMPLETABLEFUTURE

- La versiunea anterioară a API adăugam un serviciu care calculează reducerea (discount)
- Se introduce o nouă clasă **Quote**, care păstrează datele ofertei
- Metoda **parse** convertește un String returnat de **getPrice** într-o instanță a clasei **Quote**

```
package atm.paradigms.async;

public class Quote {
    private final String shopName;
    private final double price;
    private final Code discountCode;
    public Quote(String shopName, double price, Code discountCode) {
        this.shopName = shopName;
        this.price = price;
        this.discountCode = discountCode;
    }

    public static Quote parse(String s){
        String[] split = s.split(":");
        String shopName = split[0];
        double price = Double.parseDouble(split[1]);
        Code code = Code.valueOf(split[2]);
        return new Quote(shopName, price, code);
    }
    . . . // getter methods
}
```

# PROGRAMARE ASINCRONĂ

## CLASA COMPLETABLEFUTURE

- Se introduce **enumerația Code** care ține tipurile de reduceri

```
package atm.paradigms.async;

public enum Code{
    NONE(0), SILVER(5), GOLD(10), PLATINUM(15), DIAMOND(20);
    private int percentage;
    Code(int percentage){
        this.percentage = percentage;
    }
    public int getPercentage() {
        return percentage;
    }
}
```

- Introducem clasa **Discount** care calculează oferta prin intermediul funcției **discount**, folosind codul de reducere și returnează un String



# PROGRAMARE ASINCRONĂ

## CLASA COMPLETABLEFUTURE

```
package atm.paradigms.async;

public class Discount {

    private static String apply(double price, Code code){
        Shop.delay();
        return String.format("%.2f", price * (100 - code.getPercentage())/100);
    }

    public static String discount(Quote quote){
        return quote.getShopName() + " price is "
            + apply(quote.getPrice(), quote.getDiscountCode());
    }
}
```

- Funcția **getPrice** din clasa **Shop** se modifică pentru a returna un **String** separat cu ‘:’ ce este utilizat de punctia **parse** din clasa **Quote** prezentată anterior



# PROGRAMARE ASINCRONĂ

## CLASA COMPLETABLEFUTURE

```
public String getPrice(String product){  
    double price = calculatePrice(product);  
    Code code = Code.values()[random.nextInt(Code.values().length)];  
    return name + ":" + price + ":" + code;  
}
```

- Reimplementăm versiunea secvențială cu serviciul de reducere

```
public static List<String> findPricesBlocking(String product) {  
    return getShopList().stream()  
        .map(shop -> shop.getPrice(product))  
        .map(Quote::parse)  
        .map(Discount::discount)  
        .collect(toList());  
}
```

- Rezultat: [eMag price is 94.38, Altex price is 125.94, Flanco price is 98.60]  
• Done in 12098ms



# PROGRAMARE ASINCRONĂ

## CLASA COMPLETABLEFUTURE

```
public static List<String> findPricesAsync(String product) {
    List<CompletableFuture<String>> futurePrices = getShopList().stream()
        .map(shop -> CompletableFuture.supplyAsync(
            () -> shop.getPrice(product), getExecutor()))
        .map(future -> future.thenApply(Quote::parse))
        .map(future -> future.thenCompose(quote -> CompletableFuture.supplyAsync(
            () -> Discount.discount(quote), getExecutor())))
        .collect(toList());
    return futurePrices.stream()
        .map(CompletableFuture::join)
        .collect(toList());
}
```

Execuță asincron și la distanță metoda `getPrice()`, returnează stream `CompletableFuture<String>`.

Execuță sincron și local conversia String la Quote (`thenApply`), returnează stream `CompletableFuture<Quote>`.

Execuță asincron și la distanță metoda `discount()` cu argumentul `Quote` produs anterior (`thenCompose`), returnează stream `CompletableFuture<String>`.

### Reimplementarea asincronă a metodei `findPricesAsync` cu 2 servicii

- Prima metodă `map` returnează `Stream<CompletableFuture<String>>` care va conține String-ul returnat de magazin
- A 2-a metodă `map` se execută sincron și local (thread principal) transformând String-ul într-o instanță a clasei `Quote`



# PROGRAMARE ASINCRONĂ

## CLASA COMPLETABLEFUTURE

- Această transformare se execută prin invocarea metodei ***thenApply***, care **convertește sincron Completable-Future<String> în CompletableFuture<Quote>**
- Ultimul ***map*** apelează un **serviciul la distanță** și trebuie să se execute **asincron**
- Pentru a **înlăntui o altă operație asincronă** se folosește metoda ***thenCompose*** care **trece rezultatul operației anterioare** (când este disponibil) **următoarei operații asincrone**
- **Rezultatul celor 3 operații** (asincron, sincron, asincron) **sunt colectate într-o listă** de tip ***CompletableFuture<String>***
- La finalizarea tuturor operațiilor se extrag valorile folosind metoda ***join***
- **Rezultat:** [eMag price is 186.83, Altex price is 125.21, Flanco price is 111.47]
- **Done in 4073ms**



# PROGRAMARE ASINCRONĂ

## CLASA COMPLETABLEFUTURE

- Pentru a demonstra cum pot fi **combinăte operații asincrone** de la servicii independente, implementăm un serviciu care returnează rata de schimb din RON în EUR. Vor folosi serviciul pentru a calcula prețul în EUR

```
package atm.paradigms.async;

public enum Money {
    EUR(1.0), RON(4.86),
    USD(0.99), GBP(0.86);
    private final double rate;

    Money(double rate) {
        this.rate = rate;
    }
    public double getRate() {
        return rate;
    }
}
```

```
package atm.paradigms.async;

public class ExchangeService {
    public static double getRate(Money src, Money dest){
        Shop.delay();
        return dest.getRate() / src.getRate();
    }
}
```

# PROGRAMARE ASINCRONĂ

## CLASA COMPLETABLEFUTURE

- Folosim prețul în RON returnat de magazine pe care îl combinăm cu rata de schimb RON/EUR dată de serviciu independent **ExchangeService** și returnăm asincron prețul în EUR
- **CompletableFuture** are metoda **thenCombine** cu argumente, primul este un **CompletableFuture** iar al doilea este o **BiFunction** care combină rezultatele
- **Prețul produsului și rata de schimb sunt cerute asincron, iar când ambele rezultate sunt disponibile este calculat sincron prețul în EUR**
- Există varianta asincronă de calcul cu **thenCombineAsync** dacă se dorește ca **BiFunction** să se execute asincron

# PROGRAMARE ASINCRONĂ

## CLASA COMPLETABLEFUTURE

```
public static List<Double> findPricesAsyncInEUR(String product) {  
    List<CompletableFuture<Double>> futurePrices = getShopList().stream()  
        .map(shop -> CompletableFuture.supplyAsync(  
            () -> shop.getPrice(product), getExecutor()))  
        .map(future -> future.thenApply(Quote::parse))  
        .map(future -> future.thenApply(Quote::getPrice))  
        .map(future -> future.thenCombine(  
            CompletableFuture.supplyAsync(  
                () -> ExchangeService.getRate(Money.RON, Money.EUR)),  
                (price, rate) -> price * rate))  
        .collect(toList());  
    return futurePrices.stream()  
        .map(CompletableFuture::join)  
        .collect(toList());  
}
```

Execuță asincron și la distanță metoda **getRate()**, iar când prețul produsului și rata de schimb sunt disponibile calculează sincron prețul în EUR. Returnează stream **CompletableFuture<Double>**.

- Rezultat: [20.064207527816055, 36.32746052751177, 24.5168787026664]

Done in 2073ms

# PROGRAMARE ASINCRONĂ

## CLASA COMPLETABLEFUTURE

- *Pipeline CompletableFuture* pentru un singur magazin

```
package atm.paradigms.async;
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.ExecutionException;

public class CompletableFutureTest {
    public static void main(String[] args) throws InterruptedException, ExecutionException {
        Shop shop = new Shop("eMag");
        CompletableFuture<String> future = CompletableFuture.supplyAsync(() ->
                shop.getPrice("laptop"));
        CompletableFuture<Double> res = future.thenApply(Quote::parse)
                .thenApply(Quote::getPrice)
                .thenCombine(
                        CompletableFuture.supplyAsync(
                                () -> ExchangeService.getRate(Money.RON, Money.EUR)),
                        (price, rate) -> price * rate);
        System.out.println(res.get());
    }
}
```



# NOUL API PENTRU DATĂ ȘI TEMP

- Clasa ***java.util.Date*** a fost introdusă în Java 1.0 pentru **a reprezenta data și timpul**
- Clasa reprezintă **timpul pentru un moment dat** cu precizie de milisecunde, având ca **temp de referință anul 1900**. Prima lună **ianuarie** are **indicele 0**

**Date date = new Date(114, 2, 18);**

- produce

**Tue Mar 18 00:00:00 CET 2014**

- Java 1.1 a introdus clasa ***java.util.Calendar*** pentru a rezolva problemele clasei Date
- Ambele clase **Calendar și Date sunt mutabile**



# NOUL API PENTRU DATĂ ȘI TEMPORALITATE LOCALDATE ȘI LOCALTIME

- Java 8 introduce clasele `java.time.LocalDate` și `java.time.LocalTime`
- Se poate crea o instanță a `LocalDate` folosind metoda statică `of`

```
package atm.paradigms.datetime;
import java.time.LocalDate;

public class NewDateTest {
    public static void main(String[] args) {
        LocalDate date = LocalDate.of(2014, 3, 18);
        System.out.println("Year: " + date.getYear());
        System.out.println("Leap year: " + date.isLeapYear());
        System.out.println("Month: " + date.getMonth());
        System.out.println("Month length: " + date.lengthOfMonth());
        System.out.println("Day: " + date.getDayOfMonth());
        System.out.println("Day of week: " + date.getDayOfWeek());
    }
}
```

Rezultat:

Year: 2014

Leap year: false

Month: MARCH

Month length: 31

Day: 18

Day of week: TUESDAY

# NOUL API PENTRU DATĂ ȘI TEMPORALDATE ȘI LOCALTIME

- În mod similar se pot obține informații folosind metoda `get` cu parametru enumerația `ChronoField` care implementează interfața `TemporalField`

```
package atm.paradigms.datetime;
import java.time.LocalDate;
import java.time.temporal.ChronoField;

public class NewDateTest {
    public static void main(String[] args) {
        LocalDate date = LocalDate.now();
        System.out.println("Year: " + date.get(ChronoField.YEAR));
        System.out.println("Month: " + date.get(ChronoField.MONTH_OF_YEAR));
        System.out.println("Day: " + date.get(ChronoField.DAY_OF_MONTH));
    }
}
```

- Rezultat: Year: 2022  
Month: 9  
Day: 9

# NOUL API PENTRU DATĂ ȘI TEMPORAL DATE ȘI LOCALTIME

- Similar pentru *LocalTime*

```
package atm.paradigms.datetime;
import java.time.LocalTime;
import java.time.temporal.ChronoField;

public class NewDateTest {
    public static void main(String[] args) {
        LocalTime time = LocalTime.of(12, 53, 40);
        System.out.println("Hour: " + time.getHour());
        System.out.println("Minute: " + time.getMinute());
        System.out.println("Second: " + time.get(ChronoField.SECOND_OF_MINUTE));
    }
}
```

- Rezultat:  
Hour: 12  
Minute: 53  
Second: 40



# NOUL API PENTRU DATĂ ȘI TEMPORAL DATE ȘI LOCAL TIME

- *LocalDate* și *LocalTime* pot fi create prin **parsarea unui String**

```
package atm.paradigms.datetime;
import java.time.LocalDate;
import java.time.LocalTime;
import java.time.temporal.ChronoField;

public class NewDateTest {
    public static void main(String[] args) {
        LocalDate date = LocalDate.parse("2022-09-09");
        System.out.println(date.toString());
        LocalTime time = LocalTime.parse("13:03:34");
        System.out.println(time.toString());
    }
}
```

Rezultat:  
2022-09-09  
13:03:34

- Există **varianta supraîncărcată a metodei parse** care primește și **DateTimeFormatter** (înlocuiește **java.util.DateFormat**) ca al doilea argument

# NOUL API PENTRU DATĂ ȘI TEMPORAL

```
package atm.paradigms.datetime;
import java.time.*;

public class NewDateTest {
    public static void main(String[] args) {
        LocalDate date = LocalDate.parse("2022-09-09");
        LocalTime time = LocalTime.parse("13:03:34");
        LocalDateTime dt1 = LocalDateTime.of(date, time);
        System.out.println(dt1.toString());
        // extract date and time
        LocalDate date1 = dt1.toLocalDate();
        LocalTime time1 = dt1.toLocalTime();
        LocalDateTime dt2 = LocalDateTime.of(2014, Month.SEPTEMBER, 9, 13, 14, 10);
        System.out.println(dt2.toString());
    }
}
```

Clasa **LocalDateTime** conține atât data cât și timpul.

- Rezultat: 2022-09-09T13:03:34  
2014-09-09T13:14:10



# NOUL API PENTRU DATĂ ȘI TEMPORALITATE

## CLASA INSTANT

- Clasa `java.time.Instant` reprezintă **numărul de secunde trecute de la Unix epoch time**, 1 ianuarie 1970 UTC
- Are **precizie de nanosecunde** dar nu se convertește în valori înțelese de oameni

```
package atm.paradigms.datetime;
import java.time.*;

public class NewDateTest {
    public static void main(String[] args) {
        Instant i = Instant.now();
        System.out.println(i.getEpochSecond());
        i = Instant.ofEpochSecond(10);
        System.out.println(i.getEpochSecond());
        i = Instant.ofEpochSecond(2, 1_000_000_000);
        System.out.println(i.getEpochSecond());
    }
}
```

Rezultat:  
1662719456  
10  
3



# NOUL API PENTRU DATĂ ȘI TEMPORALITATE DURATION ȘI PERIOD

- Clasa **Duration** manipulează **durata în secunde sau milisecunde** între 2 obiecte temporale precum **LocalTime**, **LocalDateTime** sau **Instant**

```
package atm.paradigms.datetime;  
import java.time.*;  
  
public class NewDateTest {  
    public static void main(String[] args) {  
        Instant i1 = Instant.now();  
        Instant i2 = Instant.ofEpochSecond(0);  
        Duration d1 = Duration.between(i2, i1);  
        System.out.println(d1.getSeconds());  
        LocalTime t1 = LocalTime.now();  
        LocalTime t2 = LocalTime.of(9, 0, 0);  
        Duration d2 = Duration.between(t2, t1);  
        System.out.println(d2.getSeconds());  
    }  
}
```

Rezultat:  
1662720291  
17091

# NOUL API PENTRU DATĂ ȘI TEMPORALITATE DURATION ȘI PERIOD

- Clasa **Period** poate fi folosite pentru **manipularea intervalelor de timp în termeni de ani, luni, zile**

```
package atm.paradigms.datetime;
import java.time.*;

public class NewDateTest {
    public static void main(String[] args) {
        LocalDate date1 = LocalDate.now();
        LocalDate date2 = LocalDate.of(2014, 3, 18);
        Period p1 = Period.between(date2, date1);
        System.out.println("Years: " + p1.getYears());
        System.out.println("Months: " + p1.getMonths());
        System.out.println("Days: " + p1.getDays());
    }
}
```

Rezultat:  
Years: 8  
Months: 5  
Days: 22



# NOUL API PENTRU DATĂ ȘI TEMPORALITATE DURATION ȘI PERIOD

- Atât pentru **Duration** cât și pentru **Period** se pot crea instanțe folosind metode statice

```
Duration threeMinutes = Duration.ofMinutes(3);  
threeMinutes = Duration.of(3, ChronoUnit.MINUTES);
```

```
Period tenDays = Period.ofDays(10);  
Period threeWeeks = Period.ofWeeks(3);  
Period twoYearsSixMonthsOneDay = Period.of(2, 6, 1);
```

# NOUL API PENTRU DATĂ ȘI TEMP MANIPULARE, PARSARE ȘI FORMATARE

- Manipularea datei în mod absolut

```
package atm.paradigms.datetime;
import java.time.*;
import java.time.temporal.ChronoField;

public class NewDateTest {
    public static void main(String[] args) {
        LocalDate date1 = LocalDate.of(2021, 11, 15);
        System.out.println(date1.toString());
        LocalDate date2 = date1.withYear(2011)
            .withMonth(5)
            .withDayOfMonth(18)
            .with(ChronoField.DAY_OF_MONTH, 9);
        System.out.println(date2.toString());
    }
}
```

Rezultat:

2021-11-15

2011-05-09

- Se observă metoda **with** care ia **TemporalField** ca primul argument

# NOUL API PENTRU DATĂ ȘI TEMP MANIPULARE, PARSARE ȘI FORMATARE

- Manipularea datei în mod relativ

```
package atm.paradigms.datetime;
import java.time.*;
import java.time.temporal.ChronoUnit;

public class NewDateTest {
    public static void main(String[] args) {
        LocalDate date1 = LocalDate.of(2021, 11, 15);
        System.out.println(date1.toString());
        LocalDate date2 = date1.plusDays(40);
        System.out.println(date2.toString());
        date2 = date1.minusWeeks(11);
        System.out.println(date2.toString());
        date2 = date1.plus(9, ChronoUnit.MONTHS);
        System.out.println(date2.toString());
    }
}
```

Rezultat:

2021-11-15  
2021-12-25  
2021-08-30  
2022-08-15

Pentru metoda **plus** pot fi folosite ca argumente **Duration** și **Period**.

# NOUL API PENTRU DATĂ ȘI TEMP MANIPULARE, PARSARE ȘI FORMATARE

- Uneori este necesar să se efectueze operații mai complexe cu data precum următoarea zi de duminică, următoarea zi lucrătoare, ultima zi a lunii etc.
- Noul API pentru dată și timp pune la dispoziție **metode statice pentru ajustarea datei** numite *TemporalAdjusters*

```
package atm.paradigms.datetime;  
import java.time.*;  
import static java.time.temporal.TemporalAdjusters.*;  
  
public class NewDateTest {  
    public static void main(String[] args) {  
        LocalDate date1 = LocalDate.of(2021, 11, 15);  
        LocalDate date2 = date1.with(nextOrSame(DayOfWeek.SATURDAY));  
        System.out.println(date2);  
        System.out.println(date2.with(lastDayOfMonth()));  
        System.out.println(date2.with(firstDayOfMonth()));  
        System.out.println(date2.with(lastDayOfYear()));  
    }  
}
```

Rezultat:

2021-11-20  
2021-11-30  
2021-11-01  
2021-12-31

# NOUL API PENTRU DATĂ ȘI TEMP MANIPULARE, PARSARE ȘI FORMATARE

- Interfața **TemporalAdjuster** are o singură metodă ceea ce o transformă într-o interfață funcțională

**@FunctionalInterface**

```
public interface TemporalAdjuster {  
    Temporal adjustInto(Temporal temporal);  
}
```

- Interfața definește **modul de conversie a unui obiect de tip Temporal într-un alt obiect temporal**
- Interfața poate fi privită ca un **UnaryOperator<Temporal>**
- Vom prezenta ca exemplu modul de calcul al următoarei zile lucrătoare (sare peste sămbătă și duminică)

# NOUL API PENTRU DATĂ ȘI TEMP MANIPULARE, PARSARE ȘI FORMATARE

```
package atm.paradigms.datetime;
import java.time.*;
import java.time.temporal.*;

public class NewDateTest {
    public static void main(String[] args) {
        LocalDate date1 = LocalDate.of(2022, 9, 9);
        TemporalAdjuster nextWorkingDay = TemporalAdjusters.ofDateAdjuster(
            temporal -> {
                DayOfWeek d = DayOfWeek.of(temporal.get(ChronoField.DAY_OF_WEEK));
                int dayToAdd = 1;
                if (d == DayOfWeek.FRIDAY) dayToAdd = 3;
                if (d == DayOfWeek.SATURDAY) dayToAdd = 2;
                return temporal.plus(dayToAdd, ChronoUnit.DAYS);
            });
        System.out.println(date1.with(nextWorkingDay));
    }
}
```

Se implementează prin expresie lambda. Calculează următoarea zi lucrătoare sărind peste weekend.

Rezultat: 2022-09-12

# NOUL API PENTRU DATĂ ȘI TEMP MANIPULARE, PARSARE ȘI FORMATARE

- Parsarea și formatarea datei

```
ackage atm.paradigms.datetime;
import java.time.*;
import java.time.format.DateTimeFormatter;

public class NewDateTest {
    public static void main(String[] args) {
        LocalDate date1 = LocalDate.of(2021, 11, 15);
        System.out.println(date1.format(DateTimeFormatter.BASIC_ISO_DATE));
        System.out.println(date1.format(DateTimeFormatter.ISO_LOCAL_DATE));
        DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd.MM.yyyy");
        System.out.println(date1.format(formatter));
        String sDate = "11.09.2022";
        LocalDate date2 = LocalDate.parse(sDate, formatter);
        System.out.println(date2.format(DateTimeFormatter.ISO_LOCAL_DATE));
    }
}
```

Rezultat:  
20211115  
2021-11-15  
15.11.2021  
2022-09-11

# NOUL API PENTRU DATĂ ȘI TEMP MANIPULARE, PARSARE ȘI FORMATARE

```
package atm.paradigms.datetime;
import java.time.*;
import java.time.format.DateTimeFormatter;
import java.time.format.DateTimeFormatterBuilder;
import java.time.temporal.ChronoField;
import java.util.Locale;

public class NewDateTest {
    public static void main(String[] args) {
        LocalDate date1 = LocalDate.of(2021, 11, 15);
        DateTimeFormatter rouFormatter = new DateTimeFormatterBuilder()
            .appendText(ChronoField.DAY_OF_MONTH)
            .appendLiteral(" ")
            .appendText(ChronoField.MONTH_OF_YEAR)
            .appendLiteral(" ")
            .appendText(ChronoField.YEAR)
            .parseCaseInsensitive()
            .toFormatter(Locale.forLanguageTag("ro"));
        System.out.println(date1.format(rouFormatter));
    }
}
```

Rezultat:  
15 noiembrie 2021

Crearea unui **DateTimeFormatter** personalizat folosind clasa  
**DateTimeFormatterBuilder**.



# NOUL API PENTRU DATĂ ȘI TEMP ZONE DE TEMP

- O zonă de temp reprezintă un set de reguli corespunzătoare unei regiuni în care timpul standard este același. Clasa `ZonedDateTime` conține setul de reguli corespunzătoare unei zone de temp

```
package atm.paradigms.datetime;
import java.time.*;
import java.util.TimeZone;

public class NewDateTest {
    public static void main(String[] args) {
        ZoneId localZone = TimeZone.getDefault().toZoneId();
        System.out.println(localZone);
        LocalDateTime dt = LocalDateTime.of(2022, Month.SEPTEMBER, 12, 14, 30);
        ZonedDateTime zdt1 = dt.atZone(localZone);
        ZonedDateTime zdt2 = zdt1.withZoneSameInstant(ZoneId.of("Europe/Rome"));
        System.out.println(zdt2);
        Instant instant = Instant.now();
        LocalDateTime timeFromInstant = LocalDateTime.ofInstant(instant,
                ZoneId.of("Europe/Rome"));
        System.out.println(timeFromInstant);
        timeFromInstant = LocalDateTime.ofInstant(instant, ZoneId.of("Europe/London"));
        System.out.println(timeFromInstant);
    }
}
```

Rezultat:

*Europe/Athens*

*2022-09-12T13:30+02:00[Europe/Rome]*

*2022-09-12T12:22:34.888838600*

*2022-09-12T11:22:34.888838600*

# TEHNICI DE PROGRAMARE FUNCȚIONALĂ

## FUNCȚII PESTE TOT

- Am văzut în cursul anterior că funcțiile/metodele trebuie să fie asemenea funcțiilor matematice fără efecte colaterale
- În programarea funcțională funcțiile sunt folosite ca orice alte valori: trecute ca argumente, returnate ca rezultate, stocate în structuri de date
- Astfel de funcții poartă numele de **funcții de primă clasă (first class)**
- Începând cu Java 8 se pot folosi metode ca valori de tip funcție prin operatorul :: ca referințe de metode sau prin expresii lambda
- Putem stoca o metodă într-o variabilă de tip funcție:  
**Function<String, Integer> strToInt = Integer::parseInt;**

# TEHNICI DE PROGRAMARE FUNCȚIONALĂ

## FUNCȚII PESTE TOT

- Am folosit în cursurile anterioare metoda statică **Comparator.comparing** care primește o funcție ca parametru și returnează o altă funcție (Comparator)

**Comparator<Apple> c = comparing(Apple::getWeight);**

**Funcțiile de ordin înalt (higher-order)** au următoarele proprietăți:

- Primesc unul sau mai mulți parametrii de tip funcții
- Returnează o funcție ca rezultat
- Un program **calculator numeric** poate avea tipul de date **Map<String, Function<Double, Double>>** care mapează String-ul “sin” la funcția **Function<Double, Double>**

# TEHNICI DE PROGRAMARE FUNCȚIONALĂ

## FUNCȚII PESTE TOT

- Componerea funcțiilor cu metoda `andThen`

```
package atm.paradigms.tests;
import java.util.function.Function;

public class ComposeFunction {
    public static void main(String[] args) {
        Function<Integer, Integer> f = x -> x + 1;
        Function<Integer, Integer> g = x -> x * 2;
        Function<Integer, Integer> h = f.andThen(g);
        System.out.println(h.apply(1));
    }
}
```

- Rezultat: 4
- Este echivalent cu  $g(f(x))$

# TEHNICI DE PROGRAMARE FUNCȚIONALĂ

## FUNCȚII PESTE TOT

- Componerea funcțiilor cu metoda **compose**

```
package atm.paradigms.tests;
import java.util.function.Function;

public class ComposeFunction {
    public static void main(String[] args) {
        Function<Integer, Integer> f = x -> x + 1;
        Function<Integer, Integer> g = x -> x * 2;
        Function<Integer, Integer> h = f.compose(g);
        System.out.println(h.apply(1));
    }
}
```

- Rezultat: 3
- Este echivalent cu **f(g(x))**



# TEHNICI DE PROGRAMARE FUNCȚIONALĂ

## FUNCȚII PESTE TOT

- *Currying* este o tehnică prin care o funcție cu 2 argumente ( $x$  și  $y$ ) este reprezentată ca o funcție  $g$  cu un argument care returnează o funcție la care se aplică celălalt argument  $f(x,y) = (g(x))(y)$

```
package atm.paradigms.tests;
import java.util.function.DoubleUnaryOperator;

public class Currying {
    public static void main(String[] args) {
        DoubleUnaryOperator convertC2F = curriedConverter(9.0/5, 32);
        System.out.println("Celsius to Fahrenheit: " + convertC2F.applyAsDouble(40));
        DoubleUnaryOperator convertKm2MI = curriedConverter(0.6214, 0);
        System.out.println("Kilometers to Miles:" + convertKm2MI.applyAsDouble(100));
    }
    public static DoubleUnaryOperator curriedConverter(double f, double b){
        return x -> x * f + b;
    }
}
```

- Rezultat: Celsius to Fahrenheit: 104.0  
Kilometers to Miles:62.13

# TEHNICI DE PROGRAMARE FUNCȚIONALĂ

## STRUCTURI DE DATE PERSISTENTE

- În **programarea funcțională** se discută **despre structuri de date imutabile**, dar cea mai comună formă poartă numele de **structuri de date persistente** (a nu se înțelege persistență prin salvare în baze de date)
- Reamintim că **în stilul de programare funcțional nu se permite modificarea structurilor de date globale sau a oricărei structuri date ca parametru unei metode**
- Pentru a explica folosim un exemplu care modelează călătorii cu trenul din punctul A în B
- Utilizăm o clasă mutabilă **TrainJourney** (implementată ca o listă simplu înălățuită) cu 2 câmpuri prețul segmentului și legătura către următorul tren
- Destinația este marcată prin legătură nulă (null)

# TEHNICI DE PROGRAMARE FUNCȚIONALĂ

## STRUCTURI DE DATE PERSISTENTE

```
package atm.paradigms.tests;

public class TrainJourney {
    public int price;
    public TrainJourney onward;

    public TrainJourney(int p, TrainJourney t) {
        price = p;
        onward = t;
    }

    public static TrainJourney link(TrainJourney a, TrainJourney b) {
        if (a == null)
            return b;
        TrainJourney t = a;
        while (t.onward != null) {
            t = t.onward;
        }
        t.onward = b;
        return a;
    }
}
```

Parurge elementele listei înlăncuite  
începând cu **a** până la ultimul.  
Conectează ultimul element găsit  
cu **b** și returnează **a**.

# TEHNICI DE PROGRAMARE FUNCȚIONALĂ

## STRUCTURI DE DATE PERSISTENTE

- Testăm metoda *link*

```
public static void main(String[] args) {  
    TrainJourney tj1 = new TrainJourney(40, new TrainJourney(30, null));  
    TrainJourney tj2 = new TrainJourney(20, new TrainJourney(50, null));  
    TrainJourney linked = link(tj1, tj2);  
    System.out.println(tj1);  
}
```

Creează 2 segmente independente:  
40 – 30 și 20 – 50.  
Le conectează cu metoda statică **link**.  
Nodul 30 se conectează cu nodul 20.

- Rezultat:

*TrainJourney [onward=TrainJourney [onward=TrainJourney  
[onward=TrainJourney [onward=null, price=50], price=20], price=30],  
price=40]*

# TEHNICI DE PROGRAMARE FUNCȚIONALĂ

## STRUCTURI DE DATE PERSISTENTE

- Codul anterior poate avea **efecte colaterale** deoarece **modifică starea internă a obiectelor**
- În **abordarea funcțională** structura de date care reprezintă **rezultatul unui calcul** trebuie să fie o **structură nouă de date** și nu trebuie să o afecteze pe cea anterioară
- O **obiectie adusă programării funcționale este copierea în exces a obiectelor**
- Soluția funcțională a metodei link:

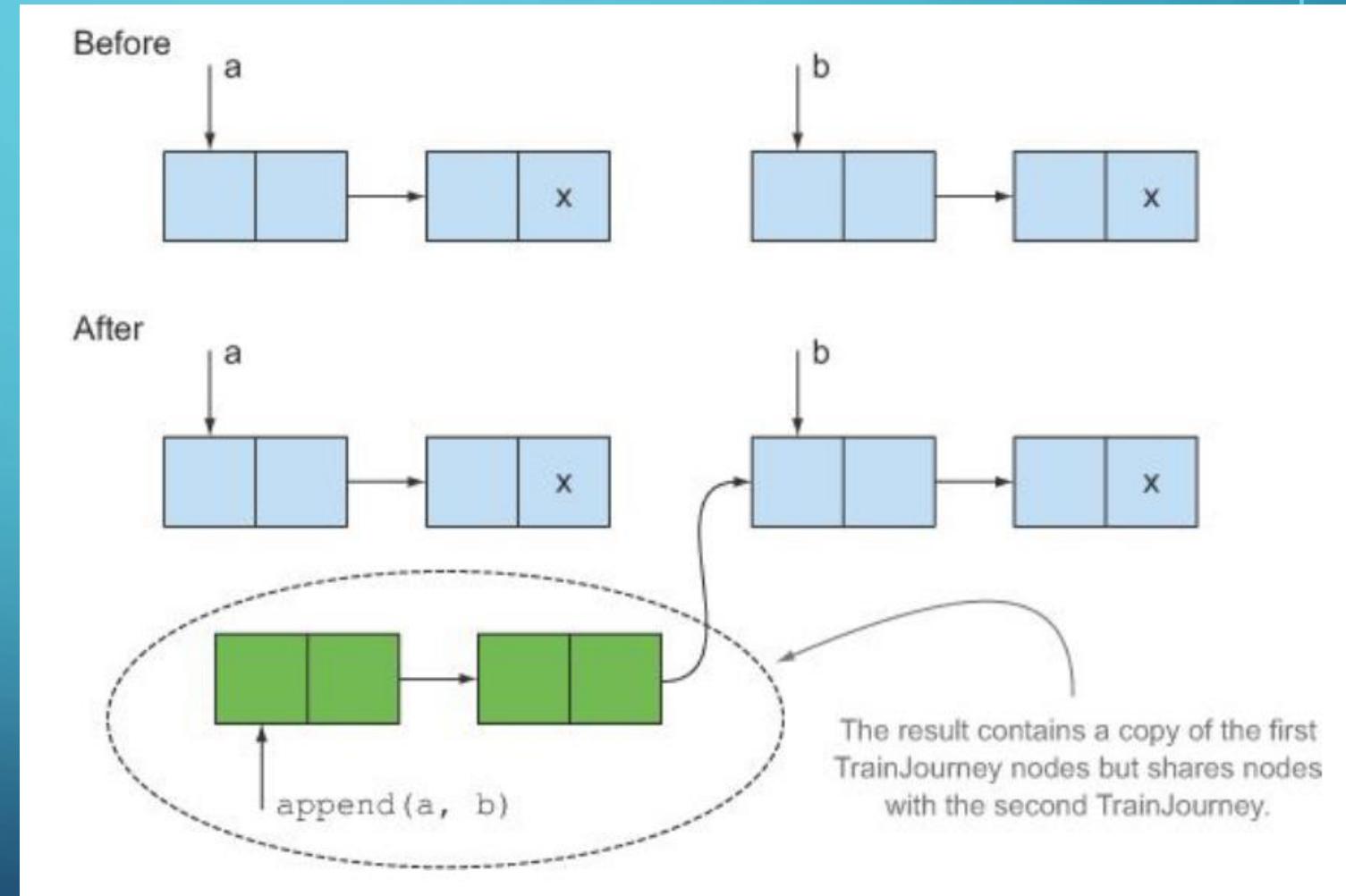
```
public static TrainJourney append(TrainJourney a, TrainJourney b){  
    return a == null ? b : new TrainJourney(a.price, append(a.onward, b));  
}
```

Creează copii ale nodurilor pentru primul segment care începe cu **a**. Conectează segmentul copiat cu celălalt care începe cu **b**.

# TEHNICI DE PROGRAMARE FUNCȚIONALĂ

## STRUCTURI DE DATE PERSISTENTE

- Codul nu modifică structuri de date existente
- Dacă avem o secvență a de n elemente și o secvență b de m elemente, metoda **returnează o secvență de n + m elemente** din care doar primele n elemente sunt noi



# TEHNICI DE PROGRAMARE FUNCȚIONALĂ

## CONCLUZII

- Este **Java un limbaj funcțional?**
- Chiar dacă la suprafață bifează multe din **caracteristicile paradigmelor funcționale, rămâne un limbaj OOP ca nucleu**
- **Elementele de programare funcționale introduse în Java sunt valoroase și ar trebui folosite ori de câte ori are sens**
- **Imutabilitatea este o practică excelentă care ar trebui folosită atât în programarea OOP cât și funcțională**
- Se recomandă **evitarea referințelor nule (null), excepțiilor și a mecanismelor de sincronizare/blocare**

# BIBLIOGRAFIE

- **Java 8 in Action**, Raoul-Gabriel Urma, Mario Fusco, Alan Mycroft, Manning Publications Co, 2015
- **Java EE 8 Design Patterns and Best Practices**, Rhuan Rocha, João Purificação, Packt Publishing, 2018
- **Java™: The Complete Reference, Tenth Edition**, Herbert Schildt, McGraw-Hill, 2005