

Teorie Java - Metode Avansate de Programare

Clase și obiecte. Instantă

- Clasa:** Un şablon sau plan care defineşte proprietăţi (câmpuri) şi comportament (metode).
- Obiect:** O instantă a unei clase.
- Instantă:** Un obiect concret creat din clasă folosind `new`.

```
Dog myDog = new Dog(); // myDog este o instantă a clasei Dog
```

Modificatori de acces

- default (fără modificador):** Accesibil doar în cadrul pachetului.

Public *→ public test*

- Public** face ca elementul (metodă, variabilă, clasă) să fie accesibil din orice altă clasă, indiferent de pachetul în care se află.
- De exemplu, o metodă **public** poate fi apelată de orice altă clasă din orice pachet.

Private *→ doar în același clasă*

- Private** restricţionează accesul la un element astfel încât acesta poate fi accesat doar în cadrul aceleiaşi clase în care este definit.
- Acesta este cel mai restricţionat nivel de acces şi protejează datele de accesul extern.

Protected *→ clasa lui, subclase care-l moştenesc & clase din același pachet*

- Protected** permite accesul la un element în cadrul aceleiaşi clase, în cadrul claselor din acelaşi pachet şi în clasele care moştenesc clasa respectivă (adică subclasele).
- Dacă o clasă derivată se află într-un alt pachet, tot va putea accesa elementele protected ale clasei părinte.

Static

- Static** face ca elementul să fie legat de clasa în sine şi nu de o instantă a clasei.
- Aceasta înseamnă că variabilele şi metodele statice sunt accesibile fără a crea o instantă a clasei. Ele sunt partajate de toate instanțele clasei.
- De obicei se folosesc pentru valori sau funcții care nu depind de starea unui obiect, ci sunt comune pentru toate instanțele clasei.

Final *→ asemănător cu o constantă*

- Final** este folosit pentru a declara elemente care nu pot fi modificate după ce au fost inițializate:
 - final class:** o clasă nu poate fi moștenită.
 - final method:** o metodă nu poate fi suprascrisă în subclase.
 - final variable:** o variabilă nu poate fi reatribuită după ce a fost inițializată.

```
public final class Example {
    // nu poate fi moștenită
    public final void display() {
```

```

        System.out.println("This is a final method.");
    }
}

public class SubExample extends Example {
    // nu poate suprascrie display(), pentru ca este final
}

```

Principii OOP

1. Încapsulare (Encapsulation)

- **Definiție:** Ascunderea detaliilor interne ale unui obiect și expunerea doar a ceea ce este necesar.
- **Cum se realizează:**
 - Folosirea modificatorilor de acces (`private`, `public`, `protected`).
 - Furnizarea de metode `getter` și `setter` pentru acces controlat la câmpuri.
- **Exemplu:**

```

class Person {
    private String name; // Câmp privat
    public String getName() { // Getter
        return name;
    }
    public void setName(String name) { // Setter
        this.name = name;
    }
}

```

2. Moștenire (Inheritance)

- **Definiție:** Capacitatea unei clase de a prelua proprietăți și metode ale unei alte clase.
- **Cum se realizează:**
 - Folosind cuvântul cheie `extends`.
 - Clasa care moștenește se numește **subclasă** (sau clasă derivată).
 - Clasa de la care se moștenește se numește **superclasă** (sau clasă de bază).
- **Exemplu:**

```

class Animal { // Superclasă
    void sound() {
        System.out.println("Animal sound");
    }
}
class Dog extends Animal { // Subclasă
    @Override
    void sound() {
        System.out.println("Bark");
    }
}

```

3. Polimorfism (Polymorphism)

- **Definiție:** Capacitatea unui obiect de a se comporta în mai multe moduri, în funcție de context.

- **Tipuri:**

1. **Polimorfism de runtime (suprascrierea metodelor):**

- O metodă are implementări diferite în clasele derivate.
- Exemplu:

```
Animal myDog = new Dog();
myDog.sound(); // Va apela metoda din Dog
```

2. **Polimorfism de compile-time (suprîncărcarea metodelor):**

- Mai multe metode cu același nume, dar cu parametri diferiți.
- Exemplu:

```
void print(int x) { System.out.println(x); }
void print(String s) { System.out.println(s); }
```

4. Abstractizare (Abstraction)

- **Definiție:** Simplificarea complexității prin expunerea doar a caracteristicilor esențiale ale unui obiect.

- **Cum se realizează:**

- Folosind **clase abstracte și interfețe**.
- Clasele abstracte pot conține metode abstracte (fără implementare) și metode implementate.
- Interfețele definesc un contract (metode abstracte) care trebuie implementat de clase.

- **Exemplu clasă abstractă:**

```
abstract class Animal {
    abstract void sound(); // Metodă abstractă
    void sleep() { // Metodă implementată
        System.out.println("Sleeping");
    }
}
```

- **Exemplu interfață:**

```
interface Sound {
    void makeSound(); // Metodă abstractă
}
class Dog implements Sound {
    public void makeSound() {
        System.out.println("Bark");
    }
}
```

5. Asociere (Association)

- **Definiție:** Relația între două clase independente, unde un obiect folosește alt obiect.

- **Tipuri:**

1. **Agregare (Aggregation):** Relație "has-a" unde obiectul conținut poate exista independent.

- Exemplu: Un departament are profesori, dar profesorii pot exista fără departament.

```
class Department {
    private List<Professor> professors;
    Department(List<Professor> professors) {
        this.professors = professors;
    }
}
```

2. **Componere (Composition)**: Relație "part-of" unde obiectul conținut nu poate exista independent.

- Exemplu: O mașină are un motor, iar motorul nu poate exista fără mașină.

```
class Engine { ... }
class Car {
    private Engine engine;
    Car() {
        this.engine = new Engine(); // Motorul este creat odată cu mașina
    }
}
```

6. Colectarea gunoiului (Garbage Collection)

- Definiție**: Procesul automat de gestionare a memoriei în Java, care elimină obiectele nefolosite.
- Cum funcționează**:
 - JVM identifică și șterge obiectele care nu mai sunt referite.
 - Dezvoltatorul nu trebuie să elibereze manual memoria.

7. Legarea dinamică (Dynamic Binding)

- Definiție**: Metoda care trebuie apelată este determinată la runtime, în funcție de tipul obiectului.
- Exemplu**:

```
Animal myDog = new Dog();
myDog.sound(); // Metoda sound() din Dog este apelată, nu cea din Animal
```

8. Încapsularea datelor (Data Hiding)

- Definiție**: Ascunderea detaliilor de implementare și expunerea doar a funcționalităților necesare.
- Exemplu**:

```
class BankAccount {
    private double balance; // Ascuns
    public double getBalance() { // Expus
        return balance;
    }
}
```

9. Reutilizarea codului (Code Reusability)

- Definiție**: Capacitatea de a folosi același cod în mai multe locuri.
- Cum se realizează**:

- Prin moștenire și compozиie.
- Exemplu:

```
class Calculator {  
    int add(int a, int b) {  
        return a + b;  
    }  
}  
class ScientificCalculator extends Calculator {  
    // Poate folosi metoda add() din Calculator  
}
```

10. Modularitate (Modularity)

- **Definiție:** Împărțirea codului în module sau componente independente.
- **Cum se realizează:**
 - Folosind clase și pachete.
 - Exemplu:

```
package com.example.math;  
public class Calculator {  
    public int add(int a, int b) {  
        return a + b;  
    }  
}
```

Clase abstracte

- O **clăsă abstractă** este o clăsă care nu poate fi instantiată direct (nu poți crea obiecte din ea).
- O clăsă poate extinde (extends) o singură clăsă abstractă
- Este folosită ca un şablon pentru alte clase (clase derivate).
- Poate conține:
 - **Metode abstracte:** Metode fără implementare (doar semnătura).
 - **Metode concrete:** Metode cu implementare.

```
abstract class Animal {  
    abstract void sound(); // Metoda abstractă  
  
    void sleep() { // Metoda concreta  
        System.out.println("Sleeping");  
    }  
}  
  
class Dog extends Animal {  
    @Override  
    void sound() { // Implementare metodei abstracte  
        System.out.println("Bark");  
    }  
}  
  
public class Main {
```

```

public static void main(String[] args) {
    Animal myDog = new Dog();
    myDog.sound(); // Bark
    myDog.sleep(); // Sleeping
}
}

```

Interfețe

O **interfață** este un tip special în Java folosit pentru a defini un contract pe care clasele trebuie să-l respecte. Interfețele pot include metode abstracte, metode default, metode statice și constante.

- **Metode abstracte:** Metodele fără corp, pe care clasele implementatoare trebuie să le definească.
- **Metode default:** Metode cu implementare care pot fi utilizate direct de clasele implementatoare sau pot fi suprascrise.
- **Metode statice:** Pot fi apelate direct de pe interfață, fără o instanță.
- **Constante:** Variabilele dintr-o interfață sunt implicit `public static final`.

```

interface Vehicle {
    int MAX_SPEED = 120; // constanta (implicit public static final)

    void start(); // metoda abstracta (implicit public)

    default void stop() { // metoda default
        System.out.println("Vehicle stopped.");
    }

    static void service() { //metoda statica
        System.out.println("Service your vehicle.");
    }
}

```

Excepții

Excepțiile în **Java** sunt mecanisme utilizate pentru a gestiona erorile sau condițiile anormale care apar în timpul execuției unui program. Ele permit separarea logicii normale de tratarea erorilor și asigură un program mai robust și mai ușor de întreținut.

Tipuri de Excepții

1. Checked Exceptions (Verificate la compilare):

- Excepții pe care compilatorul obligă dezvoltatorul să le gestioneze (`try-catch` sau `throws`).
- Exemple: `IOException`, `SQLException`

```

import java.io.*;

public class FileReadExample {
    public void readFile() throws IOException {
        BufferedReader reader = new BufferedReader(new FileReader("file.txt"));
        System.out.println(reader.readLine());
    }
}

```

2. Unchecked Exceptions (Ne-verificate la compilare):

- Apar în timpul execuției și nu sunt verificate de compilator.
- Moștenesc clasa `RuntimeException`.
- Exemple: `ArithmaticException`, `NullPointerException` `ArrayIndexOutOfBoundsException`

```
public class DivisionExample {
    public static void main(String[] args) {
        int result = 10 / 0; // Va arunca ArithmaticException
    }
}
```

3. Errors:

- Probleme critice legate de JVM, care nu pot fi gestionate de aplicație.
- Exemple: `OutOfMemoryError`, `StackOverflowError`

Tratarea Excepțiilor

Java oferă mai multe mecanisme pentru gestionarea excepțiilor.

1. try-catch

Blochează și gestionează excepțiile care apar într-o secțiune de cod.

```
public class TryCatchExample {
    public static void main(String[] args) {
        try {
            int result = 10 / 0; // Posibilă excepție
        } catch (ArithmaticException e) {
            System.out.println("Eroare: Împărțire la zero.");
        }
    }
}
```

2. finally

Un bloc optional care rulează întotdeauna, indiferent dacă apare o excepție sau nu.

```
public class FinallyExample {
    public static void main(String[] args) {
        try {
            int[] arr = new int[2];
            System.out.println(arr[5]); // ArrayIndexOutOfBoundsException
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Index invalid.");
        } finally {
            System.out.println("Acest bloc este întotdeauna executat.");
        }
    }
}
```

3. throw

Utilizat pentru a arunca explicit o excepție.

```

public class ThrowExample {
    public static void validateAge(int age) {
        if (age < 18) {
            throw new IllegalArgumentException("Vârsta minimă este 18 ani.");
        }
    }

    public static void main(String[] args) {
        validateAge(16); // Va arunca IllegalArgumentException
    }
}

```

4. throws

Declară excepțiile pe care o metodă le poate arunca.

```

import java.io.*;

public class ThrowsExample {
    public void readFile() throws IOException {
        BufferedReader reader = new BufferedReader(new FileReader("file.txt"));
        System.out.println(reader.readLine());
    }
}

```

Crearea Excepțiilor Personalizate

Poți defini propriile excepții prin extinderea clasei `Exception` sau `RuntimeException`.

- Checked Exception Personalizată:

```

class InvalidAgeException extends Exception {
    public InvalidAgeException(String message) {
        super(message);
    }
}

public class CustomCheckedExample {
    public static void validateAge(int age) throws InvalidAgeException {
        if (age < 18) {
            throw new InvalidAgeException("Vârsta minimă este 18 ani.");
        }
    }

    public static void main(String[] args) {
        try {
            validateAge(16);
        } catch (InvalidAgeException e) {
            System.out.println("Eroare: " + e.getMessage());
        }
    }
}

```

- Unchecked Exception Personalizată:

```

class InvalidAgeRuntimeException extends RuntimeException {
    public InvalidAgeRuntimeException(String message) {
        super(message);
    }
}

public class CustomUncheckedExample {
    public static void validateAge(int age) {
        if (age < 18) {
            throw new InvalidAgeRuntimeException("Vârsta minimă este 18 ani.");
        }
    }

    public static void main(String[] args) {
        validateAge(16); // Va arunca InvalidAgeRuntimeException
    }
}

```

Colecții

Colecțiile sunt structuri de date care gestionează grupuri de obiecte. Se găsesc în pachetul `java.util`.

1. List

- Colecție ordonată care permite elemente duplicate.
- Implementări populare: `ArrayList`, `LinkedList`.

```

import java.util.*;

public class ListExample {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("apple");
        list.add("banana");
        list.add("apple");
        System.out.println(list); // [apple, banana, apple]
    }
}

```

2. Set

- Colecție care NU permite duplicate (Multime)
- Implementări populare: `HashSet` (neordonat), `TreeSet` (ordonat).

```

import java.util.*;

public class SetExample {
    public static void main(String[] args) {
        Set<String> set = new HashSet<>();
        set.add("apple");
        set.add("banana");
        set.add("apple");
        System.out.println(set); // [banana, apple]
    }
}

```

3. Map

- Stochează perechi cheie-valoare (Dicționar)
- Implementări populare: `HashMap` (neordonat), `TreeMap` (ordonat).

```
import java.util.*;  
  
public class MapExample {  
    public static void main(String[] args) {  
        Map<String, Integer> map = new HashMap<>();  
        map.put("apple", 10);  
        map.put("banana", 5);  
        System.out.println(map); // {apple=10, banana=5}  
    }  
}
```

4. Iteratori

- Utilizați pentru a traversa colecțiile.

```
import java.util.*;  
  
public class IteratorExample {  
    public static void main(String[] args) {  
        List<String> list = Arrays.asList("apple", "banana", "cherry");  
        Iterator<String> iterator = list.iterator();  
        while (iterator.hasNext()) {  
            System.out.println(iterator.next());  
        }  
    }  
}
```

5. Comparatori (Comparator și Comparable)

Comparable: Folosit pentru ordonare naturală.

- Implementare în clasă.

```
import java.util.*;  
  
class Fruit implements Comparable<Fruit> {  
    String name;  
    int quantity;  
  
    Fruit(String name, int quantity) {  
        this.name = name;  
        this.quantity = quantity;  
    }  
  
    @Override  
    public int compareTo(Fruit other) {  
        return this.name.compareTo(other.name);  
    }  
}
```

```
public class ComparableExample {
    public static void main(String[] args) {
        List<Fruit> fruits = Arrays.asList(new Fruit("apple", 10), new Fruit("banana", 5));
        Collections.sort(fruits);
        for (Fruit f : fruits) {
            System.out.println(f.name);
        }
    }
}
```

Comparator: Folosit pentru ordonare personalizată.

- Implementare în afara clasei.

```
import java.util.*;

class Fruit {
    String name;
    int quantity;

    Fruit(String name, int quantity) {
        this.name = name;
        this.quantity = quantity;
    }
}

class QuantityComparator implements Comparator<Fruit> {
    @Override
    public int compare(Fruit f1, Fruit f2) {
        return Integer.compare(f1.quantity, f2.quantity);
    }
}

public class ComparatorExample {
    public static void main(String[] args) {
        List<Fruit> fruits = Arrays.asList(new Fruit("apple", 10), new Fruit("banana", 5));
        Collections.sort(fruits, new QuantityComparator());
        for (Fruit f : fruits) {
            System.out.println(f.name + ": " + f.quantity);
        }
    }
}
```

Streams

Stream-urile sunt o API puternică introdusă în **Java 8** pentru procesarea colecțiilor de date într-un mod funcțional. Ele permit efectuarea de operații precum filtrare, mapare, reducere și sortare pe date, evitând scrierea de cod boilerplate.

- **Stream** reprezintă un flux de date secvențial sau paralel, care poate proveni dintr-o colecție, un array, fișiere, etc.
- Stream-urile funcționează pe bază de **pipeline-uri**, care constau în operații intermediere și terminale.

Caracteristici ale Stream-urilor

1. **Flux de date:** Un Stream procesează datele element cu element.
2. **Nu stochează date:** Stream-urile nu păstrează datele, ci operează direct pe sursa lor.

- ⚠️ 3. **Leneșe**: Operațiile intermediare sunt evaluate doar când apare o operație terminală.
4. **Imutabilitate**: Stream-urile nu modifică sursa (colecția sau array-ul).
5. **Paralelizare**: Stream-urile pot funcționa secvențial sau paralel pentru procesare eficientă.

Cum se creează un Stream?

1. Din colecții:

```
List<String> list = Arrays.asList("apple", "banana", "cherry");
Stream<String> stream = list.stream();
```

2. Din array-uri:

```
String[] array = {"one", "two", "three"};
Stream<String> stream = Arrays.stream(array);
```

3. Din valori directe:

```
Stream<Integer> stream = Stream.of(1, 2, 3, 4, 5);
```

4. Din fișiere (folosind `Files.lines`):

```
Stream<String> lines = Files.lines(Paths.get("file.txt"));
```

5. **Stream infinit** (generat din funcții):

```
Stream<Integer> infiniteStream = Stream.iterate(0, n -> n + 2); // 0, 2, 4, 6...
```



Operații pe Stream-uri

Stream-urile au două tipuri de operații:

1. **Operații intermediare** (lazily evaluated – leneșe): → `filter`, `map`, `sorted`, `distinct`
 - Transformă Stream-ul în alt Stream.
 - Exemplu: `filter`, `map`, `sorted`, `distinct`.
2. **Operații terminale** (consumă Stream-ul): → `forEach`, `collect`, `reduce`, `count`
 - Produc un rezultat sau un efect secundar.
 - Exemplu: `forEach`, `collect`, `reduce`, `count`.

Operații intermediare

1. `filter` – Selectează elementele care respectă o condiție:

```
List<String> list = Arrays.asList("apple", "banana", "cherry");
list.stream()
    .filter(s -> s.startsWith("a"))
    .forEach(System.out::println); // apple
```

2. `map` – Transformă elementele din Stream:

```
List<String> list = Arrays.asList("apple", "banana", "cherry");
list.stream()
    .map(String::toUpperCase)
    .forEach(System.out::println); // APPLE, BANANA, CHERRY
```

3. `sorted` – Sortează elementele:

```
List<String> list = Arrays.asList("banana", "cherry", "apple");
list.stream()
    .sorted()
    .forEach(System.out::println); // apple, banana, cherry
```

4. `distinct` – Elimină duplicatele:

```
List<Integer> list = Arrays.asList(1, 2, 2, 3, 3, 3);
list.stream()
    .distinct()
    .forEach(System.out::println); // 1, 2, 3
```

5. `limit` și `skip` – Controlează dimensiunea Stream-ului:

```
Stream<Integer> stream = Stream.iterate(1, n -> n + 1);
stream
    .skip(2) // Sare peste primele 2 elemente
    .limit(3) // Ia următoarele 3 elemente
    .forEach(System.out::println); // 3, 4, 5
```

Operații terminale

1. `forEach` – Iterează prin elemente:

```
List<String> list = Arrays.asList("apple", "banana");
list.stream()
    .forEach(System.out::println); // apple, banana
```

2. `collect` – Colectează rezultatele într-o structură de date:

```
List<String> list = Arrays.asList("apple", "banana");
List<String> result = list.stream()
    .filter(s -> s.startsWith("a"))
    .collect(Collectors.toList());
System.out.println(result); // [apple]
```

3. `count` – Numără elementele:

```
long count = list.stream()
    .filter(s -> s.startsWith("b"))
    .count();
System.out.println(count); // 1
```

4. `reduce` – Combina elementele într-o valoare:

```
List<Integer> list = Arrays.asList(1, 2, 3);
int sum = list.stream()
    .reduce(0, Integer::sum); // 0 + 1 + 2 + 3
System.out.println(sum); // 6
```

Paralelizare

Stream-urile pot fi procesate în mod paralel folosind `parallelStream` sau `stream().parallel()`:

```
List<String> list = Arrays.asList("a", "b", "c");
list.parallelStream()
    .forEach(System.out::println); // Procesare în paralel
```

Exemplu practic

```
import java.util.*;
import java.util.stream.*;

public class StreamDemo {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("john", "anna", "mike", "john", "paul");

        List<String> uniqueSortedNames = names.stream()
            .distinct()                      // Elimină duplicatele
            .filter(name -> name.length() > 3) // Filtrează numele mai lungi de 3 caractere
            .sorted()                         // Sortează alfabetic
            .collect(Collectors.toList());    // Colecțează rezultatele într-o listă

        System.out.println(uniqueSortedNames); // [anna, john, mike, paul]
    }
}
```

Funcții lambda

Funcțiile **lambda** sunt o caracteristică introdusă în **Java 8** care permite definirea de funcții anonime (fără nume). Ele sunt utile pentru implementarea interfețelor funcționale și pentru reducerea boilerplate-ului în cod.

O funcție lambda este o expresie care ia argumente și returnează o valoare, având forma:

```
(parameters) -> expression
```

sau

```
(parameters) -> { statements; }
```

Caracteristici principale

- Compacte:** Elimină nevoie de a crea clase anonime pentru interfețele funcționale.
- Expressive:** Ușurează citirea și scrierea codului.

3. **Utilizează interfețe funcționale:** Lambda-urile sunt compatibile cu interfețele care au o singură metodă abstractă (SAM - Single Abstract Method).

Definirea unei funcții lambda pentru a aduna două numere:

```
// Funcție lambda pentru adunare
(int a, int b) -> a + b
```

Cum funcționează?

- Interfețe funcționale predefinite:** Lambda-urile funcționează împreună cu interfețele funcționale din pachetul `java.util.function`. Exemple:
 - `Function<T, R>` – transformă un `T` într-un `R`.
 - `Consumer<T>` – acceptă un `T` și nu returnează nimic.
 - `Supplier<T>` – nu ia parametri și returnează un `T`.
 - `Predicate<T>` – testează o condiție și returnează `true` sau `false`.
- Sintaxă scurtată:** Dacă există un singur parametru, parantezele pot fi omise:

```
name -> name.toUpperCase()
```

3. **Blocuri de cod:** Dacă logica implică mai multe instrucțiuni, se folosesc acolade `{}` și `return`:

```
(int a, int b) -> {
    int sum = a + b;
    return sum;
}
```

Exemple detaliate

- Predicate – Filtrarea unei liste:**

```
import java.util.*;
import java.util.function.Predicate;

public class LambdaExample {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("john", "jane", "mike");

        // Folosind Predicate pentru filtrare
        Predicate<String> startsWithJ = name -> name.startsWith("j");

        names.stream()
            .filter(startsWithJ)
            .forEach(System.out::println); // john, jane
    }
}
```

- Consumer – Procesarea fiecărui element:**

```
import java.util.function.Consumer;

public class LambdaExample {
```

```

public static void main(String[] args) {
    Consumer<String> printUpperCase = s -> System.out.println(s.toUpperCase());
    printUpperCase.accept("hello"); // HELLO
}
}

```

3. Function – Transformarea datelor:

```

import java.util.*;
import java.util.function.Function;

public class LambdaExample {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("john", "jane");

        // Transformare în litere mari
        Function<String, String> toUpperCase = String::toUpperCase;

        names.stream()
            .map(toUpperCase)
            .forEach(System.out::println); // JOHN, JANE
    }
}

```

4. Supplier – Generarea unei valori:

```

import java.util.function.Supplier;

public class LambdaExample {
    public static void main(String[] args) {
        Supplier<Double> randomValue = () -> Math.random();
        System.out.println(randomValue.get()); // Ex: 0.873456123
    }
}

```

Lambda și Comparator

Lambda-urile sunt des folosite pentru a implementa comparatorii:

```

import java.util.*;

public class LambdaExample {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("john", "jane", "mike");

        // Sortare alfabetica inversă
        names.sort((a, b) -> b.compareTo(a));

        System.out.println(names); // [mike, john, jane]
    }
}

```

Avantajele funcțiilor lambda

- Reducerea complexității și boilerplate-ului.
- Ușurează utilizarea programării funcționale în Java.
- Permit procesarea colecțiilor în mod fluent (cu Stream-uri).

Interfețe funcționale

O interfață funcțională conține o singură metodă abstractă și este utilizată frecvent cu expresii lambda. Poate avea metode implicate sau statice, dar trebuie să aibă doar o metodă abstractă.

- Adnotarea `@FunctionalInterface` este optională, dar recomandată pentru verificare.

1. Interfață personalizată:

```
@FunctionalInterface
interface Calculator {
    int calculate(int a, int b);
}

public class Main {
    public static void main(String[] args) {
        Calculator add = (a, b) -> a + b;
        System.out.println(add.calculate(5, 3)); // 8
    }
}
```

2. Interfețe funcționale predefinite:

- `Predicate<T>` : Testează o condiție.

```
Predicate<String> isShort = str -> str.length() < 5;
System.out.println(isShort.test("Java")); // true
```

- `Function<T, R>` : Transformă un obiect.

```
Function<Integer, String> toString = num -> "Number: " + num;
System.out.println(toString.apply(10)); // "Number: 10"
```

- `Consumer<T>` : Acceptă un parametru, dar nu returnează nimic.

```
Consumer<String> print = str -> System.out.println(str);
print.accept("Hello");
```

- `Supplier<T>` : Nu acceptă parametri, dar returnează un obiect.

```
Supplier<Double> random = Math::random;
System.out.println(random.get()); // Ex: 0.345
```

- `BiFunction<T, U, R>` : Acceptă doi parametri și returnează un rezultat.

```
BiFunction<Integer, Integer, Integer> multiply = (a, b) -> a * b;
System.out.println(multiply.apply(2, 3)); // 6
```

Avantaje:

- Cod mai concis.
- Permite utilizarea expresiilor lambda.
- Esențiale în programarea funcțională (ex.: Streams).

Fișiere I/O. Serializare și deserializare

Fișierele I/O permit citirea și scrierea datelor din și în fișiere. Java oferă clase în pachetul `java.io` pentru acest scop.

Citirea și scrierea fișierelor text

1. Scriere într-un fișier:

```
import java.io.FileWriter;
import java.io.IOException;

public class WriteFile {
    public static void main(String[] args) {
        try (FileWriter writer = new FileWriter("file.txt")) {
            writer.write("Hello, world!");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

2. Citire dintr-un fișier:

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class ReadFile {
    public static void main(String[] args) {
        try (BufferedReader reader = new BufferedReader(new FileReader("file.txt"))) {
            String line;
            while ((line = reader.readLine()) != null) {
                System.out.println(line);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Serializare și Deserializare

Procesul de **serializare** convertește un obiect într-un flux de octeți pentru a fi salvat într-un fișier sau transmis prin rețea.
Deserializarea reconvertește fluxul de octeți înapoi într-un obiect.

1. Clasa trebuie să implementeze interfața `java.io.Serializable`.
2. Câmpurile marcate cu `transient` nu sunt serializate.

```

import java.io.FileOutputStream;
import java.io.ObjectOutputStream;
import java.io.Serializable;

class Person implements Serializable {
    private static final long serialVersionUID = 1L; // Versiune serializare
    String name;
    int age;

    Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
}

public class SerializeExample {
    public static void main(String[] args) {
        Person person = new Person("John", 30);

        try (ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("person.ser"))) {
            oos.writeObject(person);
            System.out.println("Object serialized.");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

```

import java.io.FileInputStream;
import java.io.ObjectInputStream;

public class DeserializeExample {
    public static void main(String[] args) {
        try (ObjectInputStream ois = new ObjectInputStream(new FileInputStream("person.ser"))) {
            Person person = (Person) ois.readObject();
            System.out.println("Name: " + person.name + ", Age: " + person.age);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

Avantaje serializare/deserializare:

- Salvarea stării unui obiect pentru utilizare ulterioară.
- Transmiterea obiectelor prin rețea.

Concurență

Fire de execuție (Thread)

Un **fire de execuție** (Thread) reprezintă un flux de execuție independent al unui program. Fiecare aplicație Java rulează într-un thread principal, dar poate crea și alte thread-uri pentru a realiza sarcini paralele.

- Creare thread:

1. Extinderea clasei Thread :

```
class MyThread extends Thread {  
    public void run() {  
        System.out.println("Thread running");  
    }  
}
```

2. Implementarea interfeței Runnable :

```
class MyRunnable implements Runnable {  
    public void run() {  
        System.out.println("Thread running");  
    }  
}
```

- Pornire thread:

```
MyThread thread = new MyThread();  
thread.start(); // Rulează metoda run()
```

Sincronizare (Synchronization)

Sincronizarea se folosește pentru a preveni conflictele atunci când mai multe thread-uri accesează resurse comune.

- Metode sincronizate:

```
synchronized void synchronizedMethod() {  
    // Cod accesat de un singur thread la un moment dat  
}
```

- Blocuri sincronizate:

```
void method() {  
    synchronized(this) {  
        // Cod sincronizat  
    }  
}
```

Execuție paralelă

- Java oferă clase din pachetul `java.util.concurrent` (ex. `ExecutorService`, `Callable`, `Future`) pentru gestionarea thread-urilor și execuția paralelă eficientă.

Generic Types

Tipurile generice permit crearea de clase, interfețe și metode care pot lucra cu orice tip de date, fără a specifica un tip concret. Acestea sunt utile pentru a crește reutilizarea codului și pentru a asigura siguranța tipurilor (type safety) la compilare.

1. Generice la nivel de clasă

- Permite definirea unor clase care pot manipula diferite tipuri de obiecte.

```
class Box<T> { // T este un tip generic
    private T value;

    public T getValue() {
        return value;
    }

    public void setValue(T value) {
        this.value = value;
    }
}

public class Main {
    public static void main(String[] args) {
        Box<Integer> integerBox = new Box<>();
        integerBox.setValue(10);
        System.out.println(integerBox.getValue()); // 10

        Box<String> stringBox = new Box<>();
        stringBox.setValue("Hello");
        System.out.println(stringBox.getValue()); // "Hello"
    }
}
```

2. Generice la nivel de metodă

- Permite folosirea tipurilor generice în metode pentru a le face mai flexibile.

```
public class Main {
    public static <T> void printArray(T[] array) { // T este un tip generic
        for (T element : array) {
            System.out.println(element);
        }
    }

    public static void main(String[] args) {
        Integer[] intArray = {1, 2, 3};
        String[] strArray = {"A", "B", "C"};

        printArray(intArray);
        printArray(strArray);
    }
}
```

3. Restricționarea tipurilor (Bounded Types)

- Permite limitarea tipurilor care pot fi folosite ca argumente generice.

```
class Box<T extends Number> { // T trebuie să fie un subtip al lui Number
    private T value;

    public T getValue() {
```

```

        return value;
    }

    public void setValue(T value) {
        this.value = value;
    }
}

public class Main {
    public static void main(String[] args) {
        Box<Integer> integerBox = new Box<>();
        integerBox.setValue(10);

        Box<Double> doubleBox = new Box<>();
        doubleBox.setValue(3.14);
    }
}

```

4. Tipuri multiple de generice

- Poți folosi mai multe tipuri generice într-o clasă sau metodă.

```

class Pair<K, V> { // K și V sunt tipuri generice
    private K key;
    private V value;

    public Pair(K key, V value) {
        this.key = key;
        this.value = value;
    }

    public K getKey() {
        return key;
    }

    public V getValue() {
        return value;
    }
}

public class Main {
    public static void main(String[] args) {
        Pair<Integer, String> pair = new Pair<>(1, "Apple");
        System.out.println(pair.getKey() + " - " + pair.getValue()); // 1 - Apple
    }
}

```

5. Tipuri generice în interfețe

- Interfețele pot de asemenea să utilizeze tipuri generice pentru a asigura flexibilitate.

```

interface Pair<K, V> {
    K getKey();
    V getValue();
}

```

```

class SimplePair<K, V> implements Pair<K, V> {
    private K key;
    private V value;

    public SimplePair(K key, V value) {
        this.key = key;
        this.value = value;
    }

    public K getKey() {
        return key;
    }

    public V getValue() {
        return value;
    }
}

public class Main {
    public static void main(String[] args) {
        Pair<String, Integer> pair = new SimplePair<>("Age", 30);
        System.out.println(pair.getKey() + ": " + pair.getValue()); // Age: 30
    }
}

```

Avantaje:

- **Siguranță tipurilor:** Erorile legate de tipuri sunt depistate la compilare.
- **Reutilizarea codului:** Codul poate fi folosit pentru tipuri diferite fără a fi necesar să fie duplicat pentru fiecare tip.

Anotații

Anotațiile sunt metadate care pot fi aplicate la clase, metode, câmpuri, parametri sau alte elemente de cod. Ele nu modifică comportamentul codului în mod direct, dar pot fi utilizate pentru a oferi informații suplimentare sau pentru a influența procesul de compilare sau execuție, de exemplu, prin utilizarea unor framework-uri ca Spring sau Hibernate.

1. Definirea anotățiilor personalizate

O anotație este definită folosind cuvântul cheie `@interface`.

```

@interface MyAnnotation {
    String value() default "Default Value"; // Element optional
}

```

2. Aplicarea anotățiilor

Anotațiile se aplică înaintea unei clase, metode, câmpuri, etc.

```

@MyAnnotation(value = "Custom Value")
class MyClass {
    @MyAnnotation
    public void myMethod() {
        System.out.println("Method with custom annotation.");
    }
}

```

```
    }  
}
```

3. Anotații predefinite în Java

Java include câteva anotații predefinite utilizate frecvent:

- **@Override** : Indică faptul că o metodă suprascrie o metodă din clasa părinte.

```
@Override  
public String toString() {  
    return "MyClass instance";  
}
```

- **@Deprecated** : Marchează o metodă sau clasă ca fiind învechită.

```
@Deprecated  
public void oldMethod() {  
    // Cod vechi  
}
```

- **@SuppressWarnings** : Utilizată pentru a suprime avertismentele de compilare.

```
@SuppressWarnings("unchecked")  
public void myMethod() {  
    // Cod fără avertismente  
}
```

- **@FunctionalInterface** : Indică faptul că o interfață este funcțională (conține o singură metodă abstractă).

```
@FunctionalInterface  
interface MyFunctionalInterface {  
    void doSomething();  
}
```

4. Anotații pentru procesarea la runtime

Unele anotații sunt disponibile la runtime și pot fi accesate prin reflecție. Acestea sunt adnotările cu `@Retention(RetentionPolicy.RUNTIME)`.

Exemplu de procesare la runtime:

```
import java.lang.annotation.*;  
import java.lang.reflect.*;  
  
@Retention(RetentionPolicy.RUNTIME)  
@interface Info {  
    String author() default "Unknown";  
    int version() default 1;  
}  
  
@Info(author = "John Doe", version = 2)  
class MyClass {}
```

```
public class Main {  
    public static void main(String[] args) throws Exception {  
        Class<MyClass> clazz = MyClass.class;  
        Info info = clazz.getAnnotation(Info.class);  
        System.out.println("Author: " + info.author());  
        System.out.println("Version: " + info.version());  
    }  
}
```

5. Anotații de tipul meta-anotațiilor

Există anotații care sunt folosite pentru a marca alte anotații. Exemple:

- `@Retention`: Specifică perioada de viață a unei anotații (la compilare, runtime sau sursă).
- `@Target`: Specifică ce elemente de cod pot folosi o anotație (ex. metode, clase, câmpuri).
- `@Inherited`: Permite moștenirea anotațiilor de către subclase.
- `@Documented`: Face ca anotația să fie inclusă în documentația Javadoc.

6. Exemple de utilizare a anotațiilor

```
@Target(ElementType.METHOD) // Anotare valabilă pentru metode  
@Retention(RetentionPolicy.RUNTIME) // Disponibilă la runtime  
@interface MyMethodAnnotation {}  
  
class MyClass {  
    @MyMethodAnnotation  
    public void myMethod() {  
        System.out.println("Method with custom annotation.");  
    }  
}
```

Avantaje ale anotațiilor:

- Îmbunătățesc lizibilitatea și organizarea codului.
- Permite framework-urilor să efectueze operații specifice (ex. injectie de dependențe, validări).
- Reduc necesitatea de a utiliza cod de tip "boilerplate".

Design Patterns

Design Patterns sunt soluții testate pentru probleme comune de design care apar în dezvoltarea software-ului. Acestea sunt şabloane de proiectare care oferă soluții eficiente pentru rezolvarea problemelor specifice, cum ar fi crearea, structurarea și comportamentul obiectelor. În Java, modelele de design sunt esențiale pentru a crea aplicații flexibile și ușor de întreținut.

Există 3 tipuri principale de Design Patterns:

1. **Creational Patterns** – Modele care se concentreză pe crearea obiectelor.
2. **Structural Patterns** – Modele care se concentreză pe organizarea și relațiile dintre obiecte.
3. **Behavioral Patterns** – Modele care se concentreză pe comportamentele și interacțiunile între obiecte.

1. Creational Patterns

Aceste modele sunt folosite pentru a gestiona crearea de obiecte.

Singleton

Asigură că o clasă are o singură instanță și oferă un punct global de acces la aceasta.

```
public class Singleton {  
    private static Singleton instance;  
  
    private Singleton() {} // Constructor privat  
  
    public static Singleton getInstance() {  
        if (instance == null) {  
            instance = new Singleton();  
        }  
        return instance;  
    }  
}
```

Factory Method

Permite crearea obiectelor fără a specifica tipul exact al clasei care va fi instantiată.

```
abstract class Animal {  
    public abstract void makeSound();  
}  
  
class Dog extends Animal {  
    public void makeSound() {  
        System.out.println("Bark");  
    }  
}  
  
class Cat extends Animal {  
    public void makeSound() {  
        System.out.println("Meow");  
    }  
}  
  
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;  
    }  
}
```

2. Structural Patterns

Aceste modele se concentrează pe organizarea obiectelor și pe relațiile dintre acestea.

Adapter

Permite compatibilitatea între două interfețe incompatibile. Acesta convertește o interfață într-o altă interfață.

```

interface MediaPlayer {
    void play(String fileName);
}

class AudioPlayer implements MediaPlayer {
    public void play(String fileName) {
        System.out.println("Playing audio file: " + fileName);
    }
}

interface AdvancedMediaPlayer {
    void playAdvanced(String fileName);
}

class MP4Player implements AdvancedMediaPlayer {
    public void playAdvanced(String fileName) {
        System.out.println("Playing MP4 file: " + fileName);
    }
}

class MediaAdapter implements MediaPlayer {
    AdvancedMediaPlayer advancedMusicPlayer;

    public MediaAdapter(AdvancedMediaPlayer advancedMusicPlayer) {
        this.advancedMusicPlayer = advancedMusicPlayer;
    }

    public void play(String fileName) {
        advancedMusicPlayer.playAdvanced(fileName);
    }
}

```

Decorator

Permite adăugarea de comportamente suplimentare unui obiect existent, fără a modifica structura acestuia.

```

interface Coffee {
    double cost();
}

class SimpleCoffee implements Coffee {
    public double cost() {
        return 5.0;
    }
}

class MilkDecorator implements Coffee {
    private Coffee coffee;

    public MilkDecorator(Coffee coffee) {
        this.coffee = coffee;
    }

    public double cost() {
        return coffee.cost() + 2.0;
    }
}

```

```
    }  
}
```

3. Behavioral Patterns

Aceste modele se concentrează pe modul în care obiectele interacționează între ele.

Observer

Permite unui obiect să notifice alte obiecte atunci când se produce o schimbare.

```
import java.util.ArrayList;  
import java.util.List;  
  
interface Observer {  
    void update(String message);  
}  
  
class NewsAgency {  
    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);  
        }  
    }  
}  
  
class NewsChannel implements Observer {  
    private String name;  
  
    public NewsChannel(String name) {  
        this.name = name;  
    }  
  
    public void update(String message) {  
        System.out.println(name + " received message: " + message);  
    }  
}
```

Strategy

Permite alegerea unui algoritm la runtime, fără a modifica obiectul care îl folosește.

```
interface PaymentStrategy {  
    void pay(int amount);  
}  
  
class CreditCardPayment implements PaymentStrategy {  
    public void pay(int amount) {  
        System.out.println("Paying " + amount + " using Credit Card.");  
    }  
}
```

```
}

class PayPalPayment implements PaymentStrategy {
    public void pay(int amount) {
        System.out.println("Paying " + amount + " using PayPal.");
    }
}

class ShoppingCart {
    private PaymentStrategy paymentStrategy;

    public void setPaymentStrategy(PaymentStrategy paymentStrategy) {
        this.paymentStrategy = paymentStrategy;
    }

    public void checkout(int amount) {
        paymentStrategy.pay(amount);
    }
}
```

Expresii Lambda și variabile capture

1. Ce este o variabilă capturată?

- O variabilă locală utilizată într-o expresie lambda.
- Aceasta este preluată din contextul în care lambda a fost definită.

2. Regula pentru variabilele capture:

- Nu poți modifica variabilele locale captureate în lambda.** Acestea trebuie să fie **efectiv finale**.
- Efectiv final:** Variabila nu este modificată după initializare, chiar dacă nu este declarată explicit cu `final`.

Exemple

Exemplu valid (variabilă efectiv finală):

```
public class LambdaExample {
    public static void main(String[] args) {
        String message = "Hello, Lambda!"; // efectiv final

        Runnable task = () -> System.out.println(message);
        task.run(); // Hello, Lambda!
    }
}
```

- `message` este utilizată în lambda și nu este modificată, deci este validă.

Exemplu invalid (variabilă modificată):

```
public class LambdaExample {
    public static void main(String[] args) {
        String message = "Hello, Lambda!";
        message = "Modified message"; // Modificare - eroare de compilare
        Runnable task = () -> System.out.println(message);
        task.run();
    }
}
```

- Modificarea valorii lui `message` face ca aceasta să nu mai fie **efectiv finală**, deci compilatorul va genera eroare.

Excepții: Ce poți modifica?

1. Câmpuri ale clasei:

- Poți modifica variabile care sunt câmpuri ale clasei sau ale clasei exterioare.

```
public class LambdaExample {
    private static int counter = 0;

    public static void main(String[] args) {
        Runnable task = () -> {
            counter++; // Modificare permisă
            System.out.println("Counter: " + counter);
        }
    }
}
```

```
    };
    task.run(); // Counter: 1
    task.run(); // Counter: 2
}
}
```

2. Colecții captureate:

- Referința unei colecții este **efectiv finală**, dar conținutul ei poate fi modificat.

```
import java.util.ArrayList;
import java.util.List;

public class LambdaExample {
    public static void main(String[] args) {
        List<String> names = new ArrayList<>();
        names.add("Alice");

        Runnable task = () -> names.add("Bob"); // Modificare permisă
        task.run();

        System.out.println(names); // [Alice, Bob]
    }
}
```

De ce trebuie variabilele să fie efectiv finale?

- Lambda este un obiect care poate supraviețui metodei în care a fost creată. Dacă variabila capturată s-ar modifica, ar fi greu de garantat un comportament consistent (mai ales în aplicații concurente).
- Regula asigură siguranța execuției și previne efectele secundare neașteptate.

Concluzie

- Ce nu poți modifica:**
 - Variabile locale captureate. Acestea trebuie să fie **efectiv finale**.
- Ce poți modifica:**
 - Câmpuri ale clasei.
 - Elemente din colecții captureate (dar nu referința colecției).

Modele grila

Page 1 of 4

Numele si Prenumele: _____

Grupa: _____ Data: _____

1. Care este rezultatul executiei codului urmator? (G5)

```
abstract class Abstract {
    private int val;
    Abstract() { val = 10; }
    public void set(int val){ this.val= val+getValue(); }
    public abstract int getValue();
    public int get() { return val; }
}
class G5 extends Abstract {
    public int getValue() {
        return 1111;
    }
    public static void main(String[] args) {
        G5 obj = new G5();
        obj.set(11);
        System.out.println(obj.get());
    }
}
```

- a) eroare la compilare c) niciun raspuns corect
 b) 21 d) 1122

3. Ce afiseaza urmatorul program? (G12)

```
public class R12 {
    public static void main(String[] args) {
        Stream<String> ss = Stream.of("asd ", "bus ", "aop ");
        var res = ss
            .filter(s -> {
                System.out.print(s); asd, bus, aop
                return s.contains("a"); asd, aop
            })
            .map((x) ->
            {
                System.out.print(x); asd aop
                return x.toUpperCase(); ASD AOP
            })
            .reduce("", (x, y) -> x + y); ASD AOP
        System.out.println(res);
    }
}
```

- a) asd bus aop asd bus aop ASD AOP c) ASD AOP
 b) asd asd bus aop aop ASD AOP d) eroare

stream - urmă lucraza cu elementele seconțial pt fiecare element din stream

2. Ce afiseaza urmatorul program?

```
class A {
    public int x = 0;
}
public class G10 {
    public A foo() {
        A a = new A(); a.x = 10
        try { a.x = 1; execută nu verificare
            throw new NullPointerException();
        } catch (Exception e) {
            a.x = 2; a.x = 2
            return a; a.x = 2
        } finally { a.x = 3; } execută imediat
    }
    public static void main(String[] args) {
        G10 ex = new G10();
        System.out.println(ex.foo().x);
    }
}
```

- a) 2 c) se va arunca o exceptie la executie
 b) 3 d) niciun raspuns corect

4. Ce afiseaza urmatorul program?

```
class AA<E,T>{
    private E e;
    private T t;
    public void setValueE(E e){ this.e = e;}
    public void setValueT(T t){ this.t = t;}
    public E getValue(){ return e; }
    public T getValue1(){ return t; }
}
public class Test {
    public static void main(String[] args) {
        AA bb = new AA<Integer, String>();
        bb.setValueE(1010); bb.setValueE("asfd");
        System.out.print(bb.getValue()+" "+o să mă face
bb.getValue1());
    }
}
E = int
T = string
```

- a) asfd null c) 1010 asfd
 b) 1010 null d) eroare la compilare

- Creează un flux (Stream) de siruri de caractere: "asd ", "bus ", "aop ".
- Filtrează sirurile care conțin litera "a", afișând fiecare sir verificat.
- Transformă sirurile filtrate în majuscule, afișând fiecare sir transformat.
- Combină toate sirurile transformate într-un singur sir folosind reduce.

Codul va afișa următoarele:

- În timpul filtrării, va afișa toate sirurile verificate: "asd ", "bus ", "aop ".
- În timpul mapării, va afișa sirurile care conțin "a" și sunt transformate în majuscule: "asd ", "asd ", "aop ", "aop ".

Rezultatul final combinat va fi: "ASD AOP ".

5. Ce afiseaza urmatorul program?

```

class Pizza {
    protected int id;
    public Pizza(int id) {this.id = id;}
    public Pizza() {}
    public boolean equals(Pizza obj) { return obj.id ==this.id;
} } //end Pizza class
class PizzaWithCheese extends Pizza {
    private String topping;
    public PizzaWithCheese(int id, String topping) {
        super(id);
        this.topping=topping;
    }
    public boolean equals(PizzaWithCheese obj) {
        return super.equals(obj) &&
               this.topping.equals(obj.topping);
    }
}
public class G6 {
    public static void main(String[] args) {
        PizzaWithCheese pizza1 =
            new PizzaWithCheese(1, "mozzarella");
        PizzaWithCheese pizza2 =
            new PizzaWithCheese(1, "feta");
        Pizza pizza3 = new PizzaWithCheese(1, "burduf");
        Pizza[] x={pizza1,pizza2};
        System.out.print(pizza1.equals(pizza2) + " "); False
        System.out.print(x[0].equals(x[1]) + " "); True
        System.out.print(pizza3.equals(pizza2) + " "); True
    }
}

```

*do tip Pizza si
compara id-urile
faer dicim*

- a) false true true c) false false false
 b) false true false d) eroare la compilare

6. Ce afiseaza urmatorul program?

```

1  class Contact {
2      public String Name = "Mapescu I ";
3      class Numar1 {
4          private String nr = "0987654321";
5          public String ContactNou() {
6              return Name + nr;
7          }
8      }
9      static class Numar2 {
10         private String nr = "0945654321";
11         public String ContactNou() {
12             return Name + nr;
13         }
14     }
15 }
16 public class G12 {
17     public static void main(String[] args) {
18         Contact c=new Contact();
19         Contact.Numar1 c1=c.new Numar1();
20         Contact.Numar2 c2=new Contact.Numar2();
21         System.out.println(c1.ContactNou());
22         System.out.println(c2.ContactNou());
23     }
24 }

```

*Venind la compilarea: Numar1
nu are acces la campurile
memoria ale Contact*

- a) eroare la compilare in linia 12 c) Mapescu I 0987654321
 Mapescu I 0945654321
 b) eroare la compilare d) eroare la executie

7. Ce afiseaza urmatorul program?

```

class EmailSender {
    private String message;
    public EmailSender(String s) { message=s; }
    public String run() {
        System.out.print(message+ " ");
        return "sent";
    }
}
public class G30 {
    static final Integer NTHREDS=5;
    public static void main(String[] args)
        throws InterruptedException {
        ExecutorService executor =
            Executors.newFixedThreadPool(NTHREDS);
        List<Callable<String>> l=new ArrayList<>();
        for (int i = 0; i < 3; i++) { 3 m0, m1, m2
            Callable<String> worker = new EmailSender("m"+i)::run;
            l.add(worker); l.add(worker); sent x3
        }
        List<Future<String>> futures = executor.invokeAll(l);
        executor.shutdown();
    }
}

```

- a) m1 m2 m0 b) se arunca exceptia
*InterruptedException, la executie,
executorul fiind oprit fortat*
 sau o permutare a lor
 c) m1 sent m2 sent m0 sent
sau o permutare a lor
 d) eroare la compilare

1. Class EmailSender:
 • This class has a private field `message` and a constructor that initializes this field.
 • The `run` method prints the message and returns the string "sent".

2. Class G30:
 • This class contains the `main` method, which is the entry point of the program.
 • A constant `NTHREDS` is defined to specify the number of threads in the thread pool (5 in this case).

3. Main Method:
 • An `ExecutorService` is created with a fixed thread pool of size `NTHREDS`.
 • A list of `Callable<String>` tasks is created.
 • A loop runs three times, creating an `EmailSender` instance with a message "m" followed by the loop index (e.g., "m0", "m1", "m2").
 • Each `EmailSender` instance's `run` method is added to the list of tasks using method reference syntax (`::run`).
 • The `invokeAll` method of `ExecutorService` is called with the list of tasks, which executes all tasks and returns a list of `Future<String>` objects representing the results of the tasks.
 • The `executor` is then shut down to prevent new tasks from being submitted.

8. Care este rezultatul executiei codului urmator?

```
interface Formula {
    double calculate(double a);
}

class A implements Formula {
    static int var1 = 100;
    double x=9;
    public double calculate(double a) {
        double x = A.this.var1*a; 100
        x++; 100*a + 1
        Formula f = (double b) -> { ERORRE:
            return Math.abs(x); nu e final/
        }; efectu final
        return f.calculate(a);
    }
}

public class R11 {
    public static void main(String[] args) {
        System.out.printf("%.0f", new A().calculate(10));
    }
}
```

a) 1001

b) eroare la compilare

c) 1000

d) niciun raspuns corect

9. Care este rezultatul executiei codului urmator?

```
class Student {
    String name;
    int varsta;
    Student(int v) { varsta = v; }
    Student(String n, int v) { name = n; varsta = v; }
    @Override public String toString() {
        return name+" "+varsta;
    }
}

public class G16 {
    public static void main(String[] args) {
        TreeSet<Integer> i = new TreeSet<Integer>();
        TreeSet<Student> s = new TreeSet<Student>(
            (x,y)->x.varsta-y.varsta
        );
        s.add(new Student("S",19));
        s.add(new Student("D",20));
        s.add(new Student("M",19));
        i.add(1); i.add(2); i.add(1);
        System.out.println(s+" "+i);
    }
}
```

Set = Multime

a) eroare la compilare

b) eroare la executie

c) [S 19, D 20] [1, 2]

d) [S 19, M 19, D 20] [1, 2]

10. Ce afiseaza urmatorul program?

```
public class G226 {
    public static void main(String[] args) {
        Stream<String> ss = Stream.of("ee ","xe ","xe y ");
        var res = ss
            .filter(s -> { ee xe xe y
                System.out.print(s);
                return s.contains("x"); xe , xe y
            })
            .map((x) -> {
                System.out.print(x); xe > xe y
                return x.toUpperCase(); XE XE Y
            })
            .reduce((x, y) -> x + y); XE XE Y
        res.ifPresent(System.out::print); XE XE Y
    }
}
```

a) ee xe xe y xe xe y XE XE Y

b) niciun raspuns corect

c) ee xe xe ke y xe y XE XE Y

d) ee xe xe x e y xe y

11. Ce afiseaza urmatorul program?

```
class AA<E,T>{
    private E e;
    private T t;
    public void setValueE(E e){ this.e = e;} get Value
    public void setValueT(T t){ this.t = t;} get Value
    public E getValue(){ return e; }
    public T getValue1(){ return t; }
}

public class Test {
    public static void main(String[] args) {
        AA bb = new AA<Integer, String>();
        bb.setValueE(1010); bb.setValueE("xyzt");
        System.out.print(bb.getValue()+" "+xyzt);
        bb.getValue1(); null
    }
}
```

a) xyzt null

b) 1010 xyzt

c) niciun raspuns corect

d) eroare la compilare

12. Ce afiseaza urmatorul program?

```
class Patrat{
    public int latimea;
    public Patrat(int l){
        latimea =l;
    }
}
class Dreptunghi extends Patrat{
    public int lungimea;
    public Dreptunghi(int l, int L) {
        super(2);lungimea=L;
    }
    public void zoomIn(int dx) {
        this(latimea+dx, lungimea+dx);
    }
    public int getArea() {
        return latimea *lungimea;
    }
}
public class G8 {
    public static void main(String[] args) {
        Dreptunghi d=new Dreptunghi(2,3);
        d.zoomIn(2);
        System.out.println(d.getArea()); 2x3
    }
}
```

The error "Call to 'this()' only allowed in constructor body" occurs because the `this()` call is used incorrectly in the `zoomIn` method. The `this()` call is only allowed in the constructor to call another constructor of the same class.

To fix this, you should create a new `Dreptunghi` object with the updated dimensions instead of trying to call `this()`.

to this must be first statement in constructor

- | | | |
|-----------------------|--------------------------|--------------------------|
| a) 20 | c) niciun raspuns corect | <i>eronee la răspuns</i> |
| b) eroare la executie | d) 6 | |

- | | |
|-------------------------|------------------------|
| a) nu se afiseaza nimic | c) Artist Artist |
| b) Pictor Pictor | d) eroare la compilare |

13. Ce afiseaza urmatorul program?

```
class A {
    public int x = 0;
}
public class G10 {
    public A foo() {
        A a = new A();
        try { a.x = 1;
            throw new NullPointerException();
        } catch (Exception e) {
            a.x = 2;
            return a;
        } finally { a.x = 3; }
    }
    public static void main(String[] args) {
        G10 ex = new G10();
        System.out.println(ex.foo().x);
    }
}
```

14. Ce afiseaza urmatorul program?

```
class EmailSender {
    private String message;
    public EmailSender(String s) { message=s; }
    public String run() {
        System.out.print(message+" ");
        return "done";
    }
}
public class G30 {
    public static void main(String[] args)
        throws InterruptedException {
        ExecutorService executor =
            Executors.newFixedThreadPool(3);
        List<Callable<String>> l=new ArrayList<>();
        for (int i = 1; i < 3; i++) {
            Callable<String> worker = new
                EmailSender("meeting"+i)::run;
            l.add(worker);
        }
        List<Future<String>> futures = executor.invokeAll(l);
        executor.shutdown();
    } meeting1 meeting2 sau meeting2 meeting1
}
```

- | | | |
|------|--|---|
| a) 2 | c) se va arunca o exceptie la executie | <i>a "meeting1 meeting2"
sau "meeting2 meeting"</i> |
|------|--|---|

- | |
|--------------------------|
| b) niciun raspuns corect |
|--------------------------|

- | | |
|--------------|--------------------------|
| <i>(b)</i> 3 | d) niciun raspuns corect |
|--------------|--------------------------|

- | | |
|---|-----------------------|
| c) "meeting1 done meeting2 done" sau
"meeting2 done meeting1 done" | d) eroare la executie |
|---|-----------------------|

Nume și prenume student: _____

Grupa: _____

EXAMEN SCRIS - GRILĂ cu justificare răspuns, 10% din nota finală**Obs. La examenul scris vor fi două grile – 5p fiecare, vor exista și punctaje parțiale pt explicatii corecte**

Ce se afișează la rularea codului urmator? (Aceasta grila este un exemplu)

```

class AA {
    int x;
    protected AA() { init(1008); }
    protected void init(int x) { this.x = x; }
}

class BB extends AA { init(2018) 2018
    public BB() { init(super.x * 2); } ✓ 1009
    public void init(int x) { super.x = x + 1; }
} 2019

public class Ex2 {
    public static void main(String[] args) {
        BB a = new BB();
        System.out.println(a.x);
    }
}

```

- Variante de răspuns:**

- 2018
- 2019
- eroare
- 0

- Justificare raspuns:**

- Se apelează constructorul clasei BB, apoi constructorul Clasei AA, apoi init(1008) din BB (polimorfism) x devin 1009, se revine în Constructorul BB, se apelează init(2018), se apelează init(2018) și valoarea atributului x din clasa AA devine 2019.

Obs. Se acceptă și alte explicații care surprind esența, dar nu povestesc care nu au legătura cu grila.

- Identificați conceptele din curs care au legătura cu grila propusă:**

- moștenire, polimorfism

- Dificultate:**

(Easy, Medium, Difficult)

Timp de lucru examen scris: 15min.

Nume și prenume student: _____

Grupa: _____

EXAMEN PRACTIC - MODEL - COMENZI RESTAURANT 40% din nota finală

Se doreste o aplicatie cu o interfata grafica intuitiva, care sa permita plasarea comenziilor in cadrul unui restaurant.

Cerinte functionale - 7p:

- 1) **1p** La pornirea aplicatiei se va deschide o fereastra pt. angajatii restaurantului si cate o fereastra pt. fiecare masa citita din fisierul/tabelul **Tables**. Fereastra destinata angajatilor va afisa numele "Staff", si fiecare fereastra asociata unei mese va afisa id-ul acesteia (ex: "Table 3").
- Se va defini clasa:
Table: { **id**: int }

- 2) **2p** Fiecare dintre ferestrele asociate meselor va afisa **meniul restaurantului, grupat pe categorii**. Meniul se va citi din fisierul/tabelul **Menu**.

- Se va defini clasa:
MenuItem: { **id**: int, **category**: String, **item**: String, **price**: float, **currency**: String }
- Ex. de intrari din fisierul/tabelul **Menu**:
 - 1, Antreuri, Bruschete cu rosii, 15, RON
 - 2, Antreuri, Salata Caprese, 20, RON
 - 3, Fel Principal, Paste cu sos pesto, 25, RON
 - 4, Fel Principal, Vinete parmigiana, 25, RON

- Meniul va fi afisat sub forma de tabele, cate unul pentru fiecare Categorie. De ex:

Antreuri

<i>Bruschete cu rosii</i>	<i>15 RON</i>
<i>Salata Caprese</i>	<i>20 RON</i>

Fel Principal

<i>Paste cu sos pesto</i>	<i>25 RON</i>
<i>Vinete parmigiana</i>	<i>25 RON</i>

- 3) **2p** In ferestrele asociate meselor, clientii pot **selecta mai multe intrari din meniu** (prin multiple selection sau checkbox-uri etc.), iar apoi pot **plasa comanda** aferenta produselor selectate prin actionarea unui buton "Place order".

Comenziile plasate se vor salva in fisierele/tabelele Orders si OrderItems, avand data calculata automat si statusul PLACED.

Observatie: Pt. simplitate vom considera ca o comanda poate continuta o singura bucată dintr-un anumit produs.

- Se vor defini clasele:
Order: { **id**: int, **table**: int/Table, **menuItems**: List<Integer>/List<MenuItem>, **date**: LocalDateTime, **status**: OrderStatus }, unde OrderStatus este un enum {PLACED, PREPARING, SERVED}
- Ex. de continut fisiere/tabele Orders si OrderItems:

Order:

id=1, tableId=3, date=2022-01-20T12:00, status=PLACED

OrderItems:

orderId=1, menuItemId=1
orderId=1, menuItemId=2

- 4) **2p** In momentul in care o comanda este plasata aceasta va aparea **instantaneu** (fara a fi nevoie de vreo actiune din partea utilizatorului) in **fereastra "Staff"**, in cadrul unui tabel Placed Orders. Intrarile din acest tabel vor fi

ordonate crescator dupa data la care s-a plasat comanda. Se vor afisa: id-ul mesei, data, numele produselor comandate (nu id-urile!).

Cerinte non-functionale - 3p: 0.6p*5

- Validarea datelor de intrare
Exemplu: Nu se poate plasa o comanda fara a selecta niciun produs
- Procesarea va avea loc numai la nivel de service sau de controller; interactiunea cu sursa de date se va face numai prin intermediul repository-ului (bază de date)
- Interactiunea cu utilizatorul va avea loc numai in UI (GUI)
- Se incurajează construirea unei aplicații de la 0 sau eliminarea codului care nu este folosit, precum și a funcționalităților care nu s-au cerut (daca ati lucrat cu ceva template de la lab)
- Clasele, atributele și metodele lor vor avea exact numele cerute in problema sau nume sugestiv daca nu s-a specificat explicit numele lor

IMPORTANT

- Se punteaza doar cerintele functionale care ruleaza
- Orice cod care nu poate fi explicat, atrage dupa sine nepunctarea cerintei/cerintelor din care face parte
- Nu aveti voie sa comunicati in timpul examenului, in nicio modalitate posibila de a face acest lucru (chat, mail, ...etc.)

Cerinte extra – pt. bonus: 1p in plus la nota finala:

Pt. 1p in plus la nota finala se poate rezolva oricare dintre urmatoarele cerinte (nu ambele):

- 1) Adaugarea posibilitatii de a selecta cantitatea dorita pt. fiecare produs atunci cand se plaseaza o comanda.
 - Se va adauga clasa **OrderItem** { **order**: int/Order, **menuitem**: int/MenuItem, **quantity**: int }
 - Field-ul **menuitems** din clasa **Order** va continua o lista de id-uri de OrderItem sau o lista de OrderItem
 - Pentru fiecare intrare din fisierul/tabelul OrderItems se va adauga si cantitatea
 - Pe fiecare rand din tabelele in care este afisat meniul se va adauga o noua coloana ce va continua un input field "Quantity". Acesta va fi disabled pt. produsele care nu au fost selectate.
- 2) Fereastra "Staff" va permite schimbarea statusului comenzilor in felul urmator:
 - O comanda PLACED poate fi marcata ca PREPARING DELIVERED (prin actionarea unui buton). In acest caz comanda se va muta din tabelul Placed Orders intr-un nou tabel Preparing Orders (ordonat tot dupa data ascending). Fereastra corespunzatoare mesei asociata comenzii va afisa instantaneu o notificare "Your order is being prepared"
 - O comanda PREPARING (din tabelul Preparing Orders) poate fi marcata ca DELIVERED (prin actionarea unui buton). In acest caz comanda va disparea din tabelul Preparing Orders. Fereastra corespunzatoare mesei asociata comenzii va afisa instantaneu o notificare "Your order has been delivered to your table"
 - Toate aceste modificari de status vor fi salvate corespunzator in fisierul/tabelul Orders

Timp de lucru examen practic: 2h 30min.