# Ghid Examen Scris

**1. Clase, interfețe, pachete, moștenire, încapsulare, polimorfism**

- Cuvinte cheie (*final, static, transient, synchronized*)
- Clase (*public* vs *non-public*), clase abstracte (keyword *abstract*), clase *imbricate* (nested class), clasele *anonime*.
- Interfețe Java (metodele sunt publice), din Java 8 există metode implicite pentru interfață (*default methods*).
- Pachete (pachetele determină o ierarhie de nume, dacă nu specificăm pachetul =» pachetul implicit (*default package*)).
- Moștenire (clasele moștenesc dintr-o singură clasă, toate clasele sunt moștenite din **java.lang.Object**), o clasă poate implementa oricâte interfețe
- Încapsulare (*private*, *default*, *protected*, *public*)
- Polimorfism (aproape toate metodele în Java sunt virtuale)

**2. Tipuri generice și colecții**

- Tipuri de date (interfețe List, Set, Map) - ce reprezintă fiecare, 1-2 implementări, dar nu trebuie să cunoașteți metodele pe de rost.
- Tablou (arrays) - cum se definește, length, ...
- Cum definim un tip generic, compatibilitatea tipurilor generice (ex. List<Person>, List<Student> unde clasa Student e derivată din clasa Person), definirea și utilizarea tipurilor generice, inclusiv tipuri generice mărginite (eng. *bounded generics*)

**3. Excepții**

- clasele *Exception*, *RuntimeException checked* and *unchecked exceptions*, aruncarea și prinderea excepțiilor, blocuri catch care prind una sau mai multe tipuri de excepții (+efectul moștenirii asupra blocului *catch*), blocul *finally*

**4. I/O în Java**

- Lucrul cu fișiere text (clasele **Reader* și **Writer*), clasele de buffer (ex. BufferedReader)
- Lucrul cu fișiere binare (clasele **Stream)* și serializarea Java (*ObjectInputStream*, *ObjectOutputStream*), interfața *Serializable*, câmpul de versiune al clasei serializate, cuvântul *transient*.

**5. Utilizarea JDBC în Java**

- Doar pentru examenul practic (conectarea la BD, citire/scriere/actualizare)

**6. Programarea funcțională cu Java**

- Ce este o interfață funcțională (eng. *Functional Interface*), un predicat (eng. *Predicate*)
- Lucrul cu *stream*-uri (cum obțin un *stream* pornind de la o colecție, operații pe *stream*-uri, operații de colectare - doar cele care apar în notițele de curs, nu trebuie știute pe de rost)

**7. Interfața grafică folosind JavaFX**

- Doar pentru examenul practic (inițializarea interfeței, lucrul cu fișierul FXML, controale - liste, câmpuri text, butoane), procesarea evenimentelor din interfață, preluarea informațiilor din controalele interfeței grafice

**8. Concurență**

- Le utilizăm pentru **(1)** a nu bloca interfața grafică cu operații care durează mult sau **(2)** a mări viteza de prelucrare a datelor prin calcul paralel.
- Utilizarea firelor de execuție (eng. *threads*) - interfața *Runnable*, clasa *Thread*, pornirea unui fir de execuție (metoda *start()*) și așteptarea terminării execuției (metoda *join()*).

**9. Șabloane de proiectare**

- ***Singleton*** (de exemplu, citirea și accesarea fișierului *settings.properties*) - cum se definește, cum se limitează accesul la unicul obiect atât pe firul de execuție curent cât și în programarea concurentă.
- ***Composite*** (de exemplu, la definirea interfețelor grafice) - cum se definește, cum obținem structura ierarhică cu un număr nelimitat de nivele, clasa compozit și clasa frunză.
- ***Observer*** (de exemplu, lista observabilă din JavaFX) - rolurile de observat (*Observable*) și observator (*Observer*), cum se definește, cum adăugăm, păstrăm, ștergem și notificăm observatorii.
- ***Map-Reduce*** - nu se cere la examen.

**Altele**

- mecanismul *try-with-resources* (interfața *AutoCloseable*)
- Diagramele ***UML de clase*** (definirea unei clase și a unei interfețe, metode abstracte, reprezentarea relațiilor de moștenire și asociere între clase), implementarea unei diagrame de clase UML sub formă de cod Java.

Determinați rezultatul execuției următoarelor programe Java. Justificați răspunsul. Dacă apar erori de compilare/execuție, care sunt acestea? - în acest caz, corectați-le și detaliați rezultatul obținut după corectarea erori(lor). Presupunem că toate metodele statice sunt definite într-o clasă **Main** și sunt apelate din funcția publică statică **main**.

```
interface Shape {
    public double getArea();
}

interface Polygon extends Shape {
    public double getArea();
}

class Rectangle implements Polygon {
    private int length, width;

    public Rectangle(int length, int width) {
        this.length = length;
        this.width = width;
    }

    public double getArea() {
        return this.length * this.width;
    }

    @Override
    public String toString() {
        return "Rectangle: " + getArea();
    }
}

class Square extends Rectangle {
    public Square(int length) {
        super(length, length);
    }

    @Override
    public String toString() {
        return "Square: " + getArea();
    }
}

class Circle implements Shape
{
    private int radius;

    public Circle(int radius) {
        this.radius = radius;
    }

    public double getArea() { return Math.PI *
Math.pow(radius, 2); }

    @Override
    public String toString() {
        return "Circle: " + getArea();
    }
}
```

```
class AreaCalculator<T extends Polygon> {
    private T[] elements;

    public AreaCalculator(T[] elements) {
        this.elements = elements;
    }

    double[] computeAreas() {
        double[] res = new double[elements.length];
        int i = 0;
        for (T e: elements) {
            res[i++] = e.getArea();
        }
        return res;
    }
}
```

**A**

```
public static void function1() {
    Polygon r1 = new Rectangle(2, 3);
    Rectangle r2 = new Square(3);

    System.out.println(r1 instanceof
Rectangle);
    System.out.println(r2 instanceof
Rectangle);
    System.out.println(r1 instanceof
Square);
    System.out.println(r2 instanceof
Polygon);
    System.out.println(r2.getArea());
}
```

**B**

```
public static void
function2_aux1(List<Polygon> l){
    for (Polygon p: l)

System.out.println(p.getArea());
}

public static void
function2_aux2(Polygon ... l) {
    for (Polygon p: l)

System.out.println(p.getArea());
}


public static void function2() {
        List<Square> l = new
ArrayList<>();
        Square s1 = new Square(2);
        Square s2 = new Square(3);
        l.add(s1);
        l.add(s2);
        function2_aux1(l);
        function2_aux2(s1, s2);
    }
```

**C**

```
public static void function3(){
    Square[] squares = {new Square(2), new Square(1), new Square(5)};
    AreaCalculator<Square> calculator1 = new AreaCalculator<>(squares);
    double[] areasSquares = calculator1.computeAreas();
    for (double a: areasSquares)
        System.out.println(a);

    Circle[] circles = {new Circle(1), new Circle(2)};
    AreaCalculator<Circle> calculator2 = new AreaCalculator<>(circles);
    double[] areasCircles = calculator2.computeAreas();
    for (double a: areasCircles)
        System.out.println(a);
}
```

**D**

```
public static void function4() {
    ArrayList<Shape> shapes = new ArrayList<>(Arrays.asList(new Rectangle(1, 2), new
Square(1), new Square(2), new Circle(1)));
    shapes.stream().filter(s -> s instanceof Polygon).sorted(new Comparator<Shape>() {
        @Override
        public int compare(Shape o1, Shape o2) {
            if (o1.getArea() > o2.getArea())
                return -1;
            else if (o1.getArea() < o2.getArea())
                return 1;
            else
                return 0;
        }
    }).forEach(System.out::println);
}
```

**E**

```
@FunctionalInterface
interface StringProcess {
    String process(String s);
}

public static void function5() {
    StringProcess process1 = s -> s.toUpperCase();
    StringProcess process2 = s -> {
        String[] parts = s.split(" ");
        String newString = "";
        for (String part : parts){
            newString = newString + part.substring(0, 1).toUpperCase() +
part.substring(1).toLowerCase();
        }
        return newString;
    };

    System.out.println(process1.process("welcome to the examination"));
    System.out.println(process2.process("welcome to the examination"));
}
```

---

Proiectați și implementați folosind limbajul Java un sistem orientat-obiect care permite verificarea automată a secțiunilor unei cărți. Fiecare secțiune are un titlu, un conținut și are agregat un verificator. Mai mult, o secțiune are o funcție *generate()* care afișează titlul și conținutul doar dacă secțiunea este corectă dpdv. al verificatorului. Verificarea va fi făcută prin intermediul unui *Verificator* abstract, care conține o funcție *verific(s: Sectiune)* ce returnează o valoare booleană. Cerințe:

1. Implementați clasa *Sectiune* **[1p]**

2. Implementați clasa *Verificator* și implementați doi verificatori: primul validează o secțiune dacă titlul începe cu literă mare și conținutul include mai mult de 2 propoziții; al doilea validează o secțiune dacă titlul conține un singur cuvânt și conținutul e mai scurt de 300 cuvinte **[2p]**.

3. Creați 2 Sectiuni și generați-le: prima va fi verificată cu primul tip de Verificator și va trece verificarea, fiind generată; a doua va fi verificată cu al doilea tip de verificator și va nu va fi generată **[1p]**.