Programare multiparadigmă - **JAVA**

Prof. univ. dr. **Claudiu Vințe**

claudiu.vinte@ie.ase.ro

Pachete

- Tipurile de date (*class, enum, interface*) care compun un program Java sunt organizate într-o structură ierarhică de spații de nume denumite ***package***
- O definiție de tip este inclusă într-un pachet prin declarația ***package*** *nume;* prezentă ca primă instrucțiune în codul sursă.
- Tipurile pot fi referite din exteriorul pachetului folosind denumirea completă (*NumePachet.NumeClasă*) sau folosind instrucțiunea ***import***

```
package PachetClase;

public class Student {
    // ...
}
```

```
package ProgramPersoane;
import  PachetClase.*;

class Program {
    public static void main(String[] args) {
        Student p = Student();
        // sau, dacă nu folosim import:
        // PachetClase.Persoana p = ...
    }
}
```

Clase – Elemente de bază

- Declarare clasă
- Declarare câmpuri și modificatori de acces
- Declarare metode și constructor
- Referire câmpuri prin **`this`**
- Instanțiere clasă cu **`new`**

Clase – Elemente de bază

- Declarare clasă

```
modificator class denumire_clasa extends clasă implements lista_interfețe {
        /* declarații câmpuri, metode, ... */
}
```

- Declarare câmpuri, constructori și metode:
  - Similar cu C++
  - Modificatorii de acces se adaugă la fiecare declarație
  - Constructori:
    - fără listă de inițializare
    - Apel alți constructori din clasă prin ***this****(parametri);* sau din clasa de bază prin ***super****(parametri);*
  - Referire câmpuri prin referința ***this*** (sau ***super*** pentru clasa de bază)
- Instanțiere clasă – folosind operatorul ***new***:
  - ***new*** *denumire_clasa(parametri);*

Modificatori de acces

La nivelul clasei:

- ***public*** – clasa este vizibilă în interiorul și în exteriorul pachetului
- implicit (fără modificator) este vizibilă doar în interiorul pachetului curent

La nivel de membru în clasă:

Modificator	Clasa curentă	Pachet curent	Clase derivate	Exterior pachet
public	**DA**	**DA**	**DA**	**DA**
protected	**DA**	(doar derivate)	**DA**	
-	**DA**	**DA**	(doar din pachet)	
private	**DA**			

Modificatorul *final*

Utilizat pentru:

- **Clase** → nu pot fi definite clase derivate

- **Metode** → metoda nu poate fi suprascrisă / ascunsă în clasele derivate

- **Variabile sau câmpuri**:
  - Conținutul nu poate fi modificat după inițializare
  - Dacă variabila este de tip referință atunci:
    - va referi mereu același obiect
    - valorile din interiorul obiectului ***pot fi modificate***
  - Inițializarea poate surveni ulterior declarării

Modificatorul ***static***

- Poate fi aplicat atât metodelor cât și câmpurilor
- Comportament similar cu C++:
  - Membrii statici pot fi accesați prin denumirea clasei și nu necesită o instanță
  - Câmpurile statice sunt partajate de către toate obiectele clasei
  - Metodele statice nu primesc referința ***this***
- Limbajul suportă blocuri statice de inițializare:

```
class Test {
    static int cod;

    // bloc de inițializare static:
    static {
        cod = 0;
    }
}
```

Moștenire și clase abstracte

- Este permisă doar moștenirea simplă – toate clasele cu excepția ***`Object`*** au exact o clasă de bază
- Constructorii și alți membri ai clasei de bază sunt accesați prin referința ***`super`***
- Toate metodele care nu sunt ***`static`*** sau ***`final`*** sunt implicit virtuale
- Metodele care nu sunt marcate ca ***`final`*** pot fi suprascrise:
  - Metode statice: metoda din clasa de bază este ascunsă
  - Metode simple: metoda din clasa de bază este suprascrisă

- **Clase abstracte**
  - Declarate folosind modificatorul ***`abstract`***
  - Metodele fără implementare trebuie să fie marcate cu modificatorul ***`abstract`***

Interfețe

- Similare cu o clasă abstractă
- Poate conține doar:
  - Constante – implicit statice, finale și publice
  - Metode fără implementare – implicit publice
  - Clase interne
- O clasă poate implementa un număr nelimitat de interfețe
- O interfață poate extinde alte interfețe
- Exemple: *Comparable, Comparer*
- Java 8 → interfețele pot conține și:
  - metode de extensie, marcate ca *default*, care au și implementare
  - metode statice: nu pot fi suprascrise în clasa ce extinde interfața

Clase interne

- Tipuri:
  - *static nested class*
    - Similare cu clasele normale
  - *inner class*
    - Conțin o referință la obiectul părinte
  - *local inner class*
    - Definite în cadrul unui bloc și accesibile doar din interiorul acestuia
  - *anonymous inner class*
    - Expresii care declară o clasă și construiesc un obiect
    - Pe baza unei interfețe / clase de bază

```
class OuterClass {
    ...
    class InnerClass {
        ...
    }
    static class StaticNestedClass {
        ...
    }
}
```

Clasa `System.lang.Object`

- *String* ***toString****()* – permite conversia către *String* a oricărui obiect
- *protected void* ***finalize****()* – apelat de JVM înaintea distrugerii obiectului
- *Class<?>* ***getClass****()* – permiterea obținerii obiectului *Class* asociat
- *protected Object* ***clone****()* – produce o copie a obiectului
- *boolean* ***equals****(Object obj)* – permite compararea valorilor a două obiecte
- *int* ***hashCode****()* – produce codul *hash* asociat instanței; utilizat în special pentru tabele de dispersie
- *notify, notifyAll, wait* - metode pentru sincronizare între fire de execuție

Implementare `equals` și `hashCode`

- Reguli:
  - Dacă două obiecte sunt egale conform *equals*, atunci obigatoriu *hashCode* trebuie să întoarcă aceeași valoare
  - Dacă două obiecte nu sunt egale conform *equals* valorile obținute prin *hashCode* pot fi egale sau diferite
  - *hashCode* trebuie să rămână neschimbat atât timp cât valorile folosite pentru *equals* rămân neschimbate
- Comportament implicit:
  - *equals* – true dacă referințele sunt egale
  - *hashCode* – adresa obiectului
- Metode utile pentru implementare:
  - Arrays.equals / Arrays.hashCode
  - Objects.hash

```
public class Persoana {
    int cod;
    String nume;
    String descriere;

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass())
            return false;
        Persoana persoana = (Persoana) o;
        return cod == persoana.cod &&
                Objects.equals(nume, persoana.nume);
    }

    @Override
    public int hashCode() {
        return Objects.hash(cod, nume);
    }
}
```

Implementare clone

- Tipuri de copiere:
  - Atribuire simplă între referințe
  - *Shallow copy*
  - *Deep copy*

- Reguli pentru suprascriere ***clone*** pentru deep copy:
  - Implementare interfață **Cloneable**
  - Suprascriere metodă publică *clone*
  - Utilizare *super.clone()* în interiorul metodei pentru a obține o copie shallow a obiectului curent
  - Construire copii pentru toate obiectele referite (câmpuri de tip reference)
  - Tratare excepție CloneNotSupportedException

Obiecte / clase imutabile

- Valorile stocate în obiectele clasei nu pot fi modificate după execuția constructorului
- Reguli:
  - Clasa este marcată cu modificatorul *final*
  - Câmpurile sunt marcate ca *private* și *final*
  - Câmpurile de tip referință sunt către obiecte imutabile
  - Oferă doar metode de tip *getter*
- Exemple
  - Clasa *String*
  - Clasele *wrapper* pentru tipurile fundamentale: *Integer, Long, Short, Double, Float, Character, Byte, Boolean*

```
final class Persoana {
    private final int cod;
    private final String nume;

    public Persoana(int cod, String nume) {
        this.cod = cod;
        this.nume = nume;
    }

    public int getCod() {
        return cod;
    }

    public String getNume() {
        return nume;
    }
}
```

Enumerații

- Clase speciale derivate din *java.lang.**Enum*** ale cărei câmpuri sunt obligatoriu constante

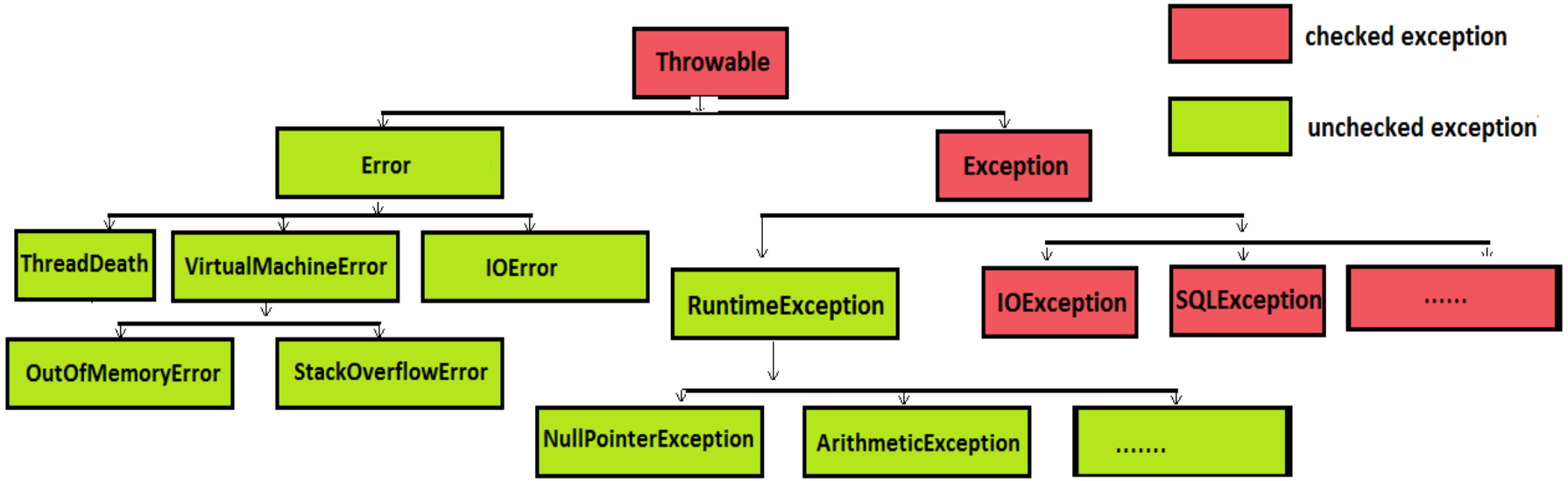
Exemplu:

```
enum Zi {
    LUNI, MARTI, MIERCURI
}

public class ProgramEnum {
    public static void main(String[] args) {
        Zi zi = Zi.LUNI;
        for (Zi ziCurenta : Zi.values()) {
            if (zi == ziCurenta) {
                System.out.print("*");
            }
            System.out.println(ziCurenta);
        }
    }
}
```

Tratarea excepțiilor

- Toate obiectele de tip excepție sunt derivate direct sau indirect din ***Throwable***
- Tipuri:
  - **Checked** – trebuie tratate printr-un bloc *try / catch* sau specificate prin clauza ***throws***
  - **Unchecked** – derivate din *Error* sau *RuntimeException* și nu este obligatorie tratarea sau raportarea



Clasa **`Throwable`**

- Throwable() - constructor fără parametri
- Throwable(String message) - primește mesajul de eroare
- Throwable(String message, Throwable cause) - primește mesajul și excepția inițială

- String getMessage() - întoarce mesajul de tip String asociat erorii
- Throwable getCause() - întoarce cauza care a generat eroare atunci cand excepția a fost provocata de o altă eroare

- void printStackTrace() - afișează detaliile complete pentru eroare
- StackTraceElement[] getStackTrace() - întoarce detaliile complete pentru eroare
- String toString() - întoarce mesajul de eroare și numele clasei

Tratarea excepțiilor

- Sintaxa *try/catch/throw/finally* și declarare clasă excepție:

```
try {
    // secvența de cod
    // eventual throw new Exceptie1(“mesaj eroare”);
} catch (Exceptie1 e1) {
    // tratare Exceptie1
} catch (Exceptie2 e2) {
    // tratare Exceptie2
}
finally {
    // cod de executat la final
}
```

```
class Exceptie1 extends Exception {
    public Exceptie1(String mesaj) {
        super(mesaj);
    }
}
```