

Programare multiparadigmă - JAVA

Conf. univ. dr. **Claudiu Vintă**

claudiu.vinte@ie.ase.ro

Tematica

- Introducere în Java. Tipuri de date fundamentale
- Programare orientată obiect în Java
- Tratarea excepțiilor. Java Native Interface
- Java I/O (stream, channel). Serializare
- Java Generics și Java Collections Framework. Adnotări și introspecție
- Programare concurentă
- Programare în rețea. Protocolele TCP, UDP și HTTP
- Accesul la baze de date prin JDBC
- Utilizare XML și JSON
- Elemente de interfață grafică în JavaFX

Bibliografie

- Ken Arnold, James Gosling, David Holmes, **THE Java™ Programming Language**, Fourth Edition, 2005, Addison Wesley Professional, ISBN: 0-321-34980-6
- Daniel Liang, **Introduction to Java Programming**, 11th Edition, Pearson, 2018, ISBN 1-292-22203-4
- Bruce Eckel, **Thinking in Java**, 4th Edition, Prentice Hall, 2006, ISBN 0-13-187248-6
- Joshua Bloch, **Effective Java**, 3rd Edition, Addison-Wesley, 2018, ISBN 0-13-468599-7
- James Gosling, et. al, **The Java Language Specification**, 11th Edition, ORACLE, 2018,
<https://docs.oracle.com/javase/specs/jls/se11/html/index.html>
- Java Platform, Standard Edition & Java Development Kit v11 **API Specification**
<https://docs.oracle.com/en/java/javase/11/docs/api/index.html>
- Raymond Gallardo, et.al., **The Java Tutorial**, ORACLE, 2014
<https://docs.oracle.com/javase/tutorial/tutorialLearningPaths.html>

Evaluare și resurse

- Notare:
 - 70% - examen
 - 30% - seminar
- Resurse:
 - <http://online.ase.ro> (*parola arhiva cărți: biblio4java*)
 - <http://ase.softmentor.ro>
 - <http://acs.ase.ro>

Java – caracteristici generale

- Limbaj de uz general
- Limbaj OO imperativ cu elemente funcționale
- Ierarhie aproape unică cu moștenire simplă pentru clase și multiplă pentru interfețe
- Interpretat – bazat pe o mașină virtuală
- Portabil
- Cu gestiune automată a memoriei bazată pe garbage collection

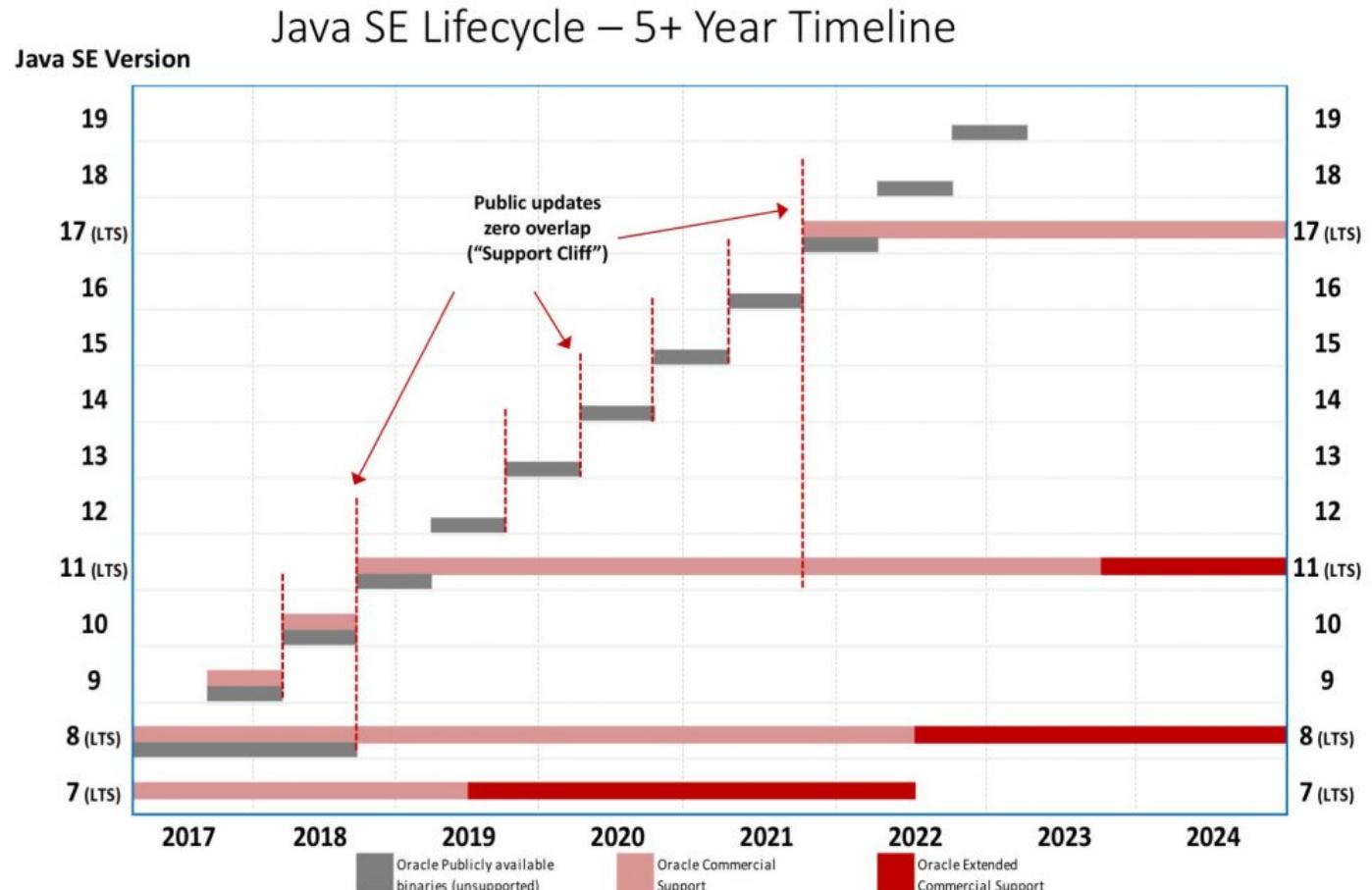
Java – Ediții / Versiuni

Java Platform, Standard Edition (Java SE)

Java Platform, Enterprise Edition (Java EE)

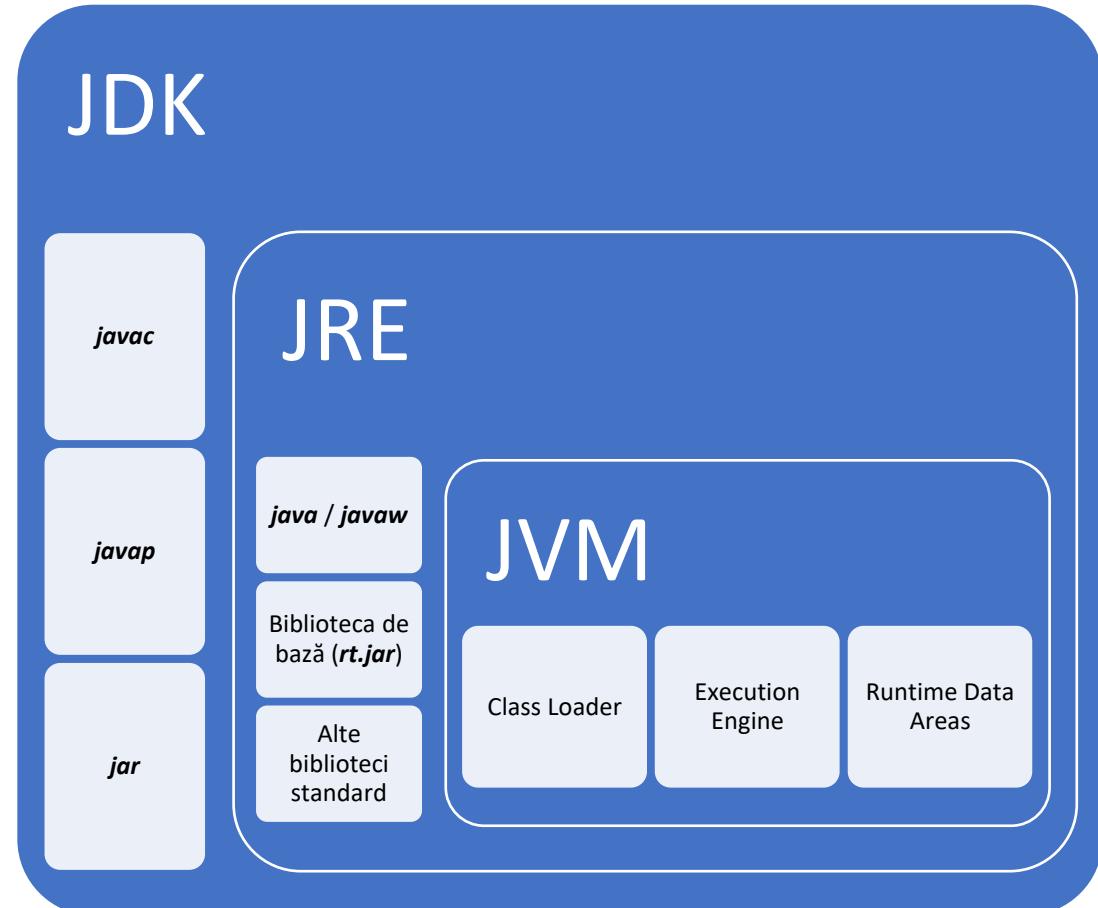
Java Platform, Micro Edition (Java ME)

JavaFX



Java – Componentele platformei

- Java Virtual Machine (JVM)
 - Mașina virtuală care rulează codul Java compilat în format *Java bytecode*
- Java Runtime Environment (JRE)
 - Conține instrumentele necesare pentru rularea aplicațiilor Java: mașina virtuală (JVM) și biblioteca standard Java (Java Class Library)
- Java Development Kit (JDK)
 - Conține instrumentele necesare pentru dezvoltarea de aplicații Java precum:
 - javac
 - jar
 - javap



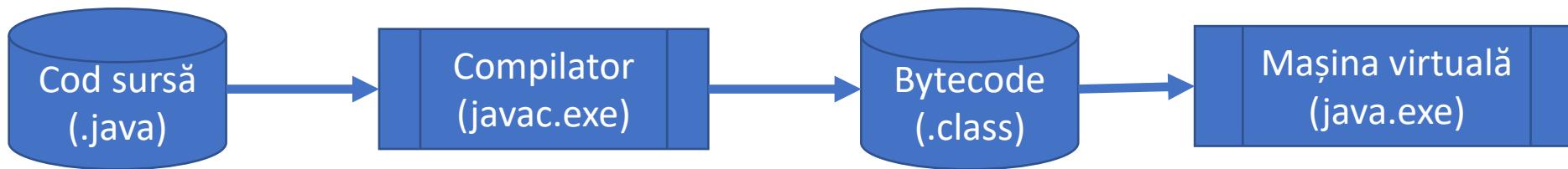
Anatomia unui program Java

- Un program Java este compus dintr-o serie de declarații de tipuri (clase, interfețe, ...)

```
class Program {  
    public static void main(String[] args){  
        System.out.println("Buna Java!");  
    }  
}
```

- Punctul de intrare – metoda statică **main**:
 - Trebuie să fie declarată publică
 - Trebuie să primească un vector de obiecte **String** ca unic parametru
 - Nu trebuie să întoarcă nimic (declarată ca **void**)

Procesul de compilare și execuție



1. Compilarea codului folosind ***javac***
 - Primește ca date de intrare
 - Fișierele sursă (fișiere *.java*)
 - Lista căi către bibliotecile utilizate (fișiere *.class* sau *.jar*)
 - Produce fișierul *.class* care conține:
 - Metadate – descrierea completă a tuturor claselor existente în fișier
 - Bytecode – codul executabil pentru fiecare metodă folosind setul de instrucțiuni al JVM
2. Rularea programului folosind mașina virtuală Java - ***java***

Alte instrumente:

jar – construire biblioteci (fișiere *.jar*)

javap – afișare detalii fișiere *.class* (metadate, dezasamblare)

Intrări / ieșiri la consolă

- Accesibile prin intermediul unor câmpuri statice ale clasei `System`:
 - `System.in` – fluxul de intrare pentru citire date (de tip `java.io.InputStream`)
 - `System.out` – fluxul de ieșire pentru afișare (de tip `java.io.PrintStream`)
- Afișare la consolă – se utilizează metodele clasei `PrintStream`:
 - `println(valoare)` – valoarea poate fi un tip de bază (`int`, `char`, `double`, ...) sau un obiect (caz în care se utilizează metoda `toString`)
 - `printf(stringFormatare, valori)` – similar cu funcția `printf` din C
- Documentația completă pentru formatare:
<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Formatter.html>
- Exemplu:

```
System.out.println("Salut!"); // afișează Salut!
System.out.println(111); // afișează 111
System.out.printf("%d + %d = %d", 11, 22, 33); // afișează 11 + 22 = 33
```

Citire de la consolă

- Pentru citire de la consolă se utilizează clasa **java.util.Scanner** care conține:
 - Constructor care primește ca parametru un obiect de tip **InputStream** (**System.in** pentru consolă)
 - Metode de forma **tip nextTip()** și **boolean hasNextTip()** pentru citire și determinare existență, unde tip este un tip de bază (**int, boolean, ...**)
 - Metodă **String nextLine()** pentru citire siruri de caractere până la sfârșitul liniei (inclusiv spații)
 - Metodă **String next ()** pentru citire siruri de caractere până la următorul separator
 - Metodă **setDelimiter(String)** pentru setarea delimitatorului (implicit spații)

Exemplu:

```
var scanner = new Scanner(System.in);
scanner.useDelimiter("[\\s,\\,]+");           // acceptăm spații sau virgulă ca delimitatori
int cod = scanner.nextInt();                 // introducem 3, Maria
String nume = scanner.next();
System.out.printf("#%d - %s", cod, nume);   // afișează #3 - Maria
```

Variabile și Tipuri de date

- Declarare variabile

- Similar cu declarațiile din C:

```
int i = 11;  
double k;
```

- Detectie automată a tipului folosind cuvântul cheie **var** (începând cu JDK 10):

```
var scanner = new Scanner(System.in);
```

- Declarare constante

- Folosesc cuvântul cheie **final**:

```
final int NUMAR_MAXIM_ELEMENTE = 10;
```

- Variabilele și constantele pot conține:

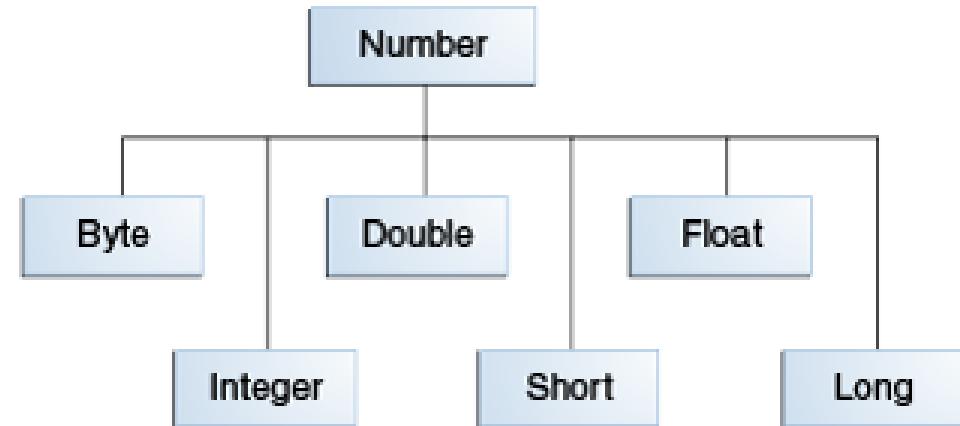
- O valoare corespunzătoare unui tip de bază (int, double, ...)
 - O referință la un obiect

Tipuri Primitive

Keyword	Dimensiune	Clasa	Literali
<i>boolean</i>	1 bit	Boolean	<i>false, true</i>
Tipuri numerice întregi			
<i>char</i>	16 bits	Character	'\u0000', 'a', '\u0041', '\\', '\'', '\n', 'ß'
<i>byte</i>	8 bits	Byte	-
<i>short</i>	16 bits	Short	-
<i>int</i>	32 bits	Integer	0, -12, 19, 2562234
<i>Long</i>	64 bits	Long	0L, -12L, 19L, 9223372036854775807L
Tipuri numerice în virgulă mobilă			
<i>float</i>	32 bits	Float	0.0f, 1.23e100f, -1.23e-100f, .3f, 3.14F
<i>double</i>	64 bits	Double	0.0d, 1.3, 1.23456e300d, -1.23456e-300d, 1e1d

Clase wrapper. Autoboxing și unboxing

- Utilitate:
 - Conțin metode de conversie între tipurile primitive și între *String* și tipurile primitive
 - Permit utilizarea tipurilor de bază în situații în care este necesară o variabilă de tip referință (în special în cazul colecțiilor)
 - Furnizează constante utile (MIN_VALUE, MAX_VALUE)
- Conversii automate:
 - Autoboxing: tip primitiv → obiect wrapper
 - Unboxing: obiect wrapper → tip primitiv



```
int val = Integer.parseInt("11"); // conversie din String
Integer i = val; // Autoboxing
String test = i.toString(); // utilizare metodă
int temp = i; // Unboxing
```

Operatori pentru tipuri primitive

- Operatorii pentru tipurile primitive sunt similari cu operatorii din C++
Exemple:

```
boolean areProiect = true, prezentaExamen = true;  
int notaExamen = 9;  
boolean estePromovat = areProiect && prezentaExamen && notaExamen > 9;  
  
long numarSecunde = 4451752342L;  
int numarOre = (int)(numarSecunde / 60 / 60);  
  
int varsta = 22;  
varsta++;  
  
double pi = 4.0 / 1 - 4.0 / 3 + 4.0 / 5 - 4.0 / 7 + 4.0 / 9;
```

Structuri de control

- Structurile de control sunt similare cu cele din C++
- *if / else* – instrucțiune alternativă simplă
- *switch / case / default / break* – instrucțiune alternativă multiplă
- *for / while / do* – instrucțiuni repetitive
- *break / continue* – instrucțiuni pentru întreruperea execuției instrucțiunilor repetitive

Şiruri de caractere

- Sunt manipulate prin intermediul obiectelor clasei **String** care:
 - Conțin o colecție de caractere
 - Sunt imutabile
- Construirea de obiecte de tip **String**:

```
String string1 = new String();           // construire sir vid
String string2 = new String("Sir existent"); // construire pe baza de sir existent
String string3 = "Tot existent";          // echivalent cu declaratia anterioara
```

- Determinarea dimensiunii și accesarea caracterelor:

```
String test = "Test";
for (int i = 0; i < test.length(); i++) { // determinare numar caractere - .length()
    System.out.println(test.charAt(i));   // accesare caracter individual - .charAt(index)
}
```

Şiruri de caractere

- Concatenare și formatare:

```
// 1. Utilizare operator de concatenare
String rezultat1 = Buna + " " + Java + "!";
```

```
// 2. Utilizare funcție statică format (sintaxa la fel ca la printf)
String rezultat2 = String.format("%s %s!", Buna, Java);
```

- Funcții de comparare și căutare:

- boolean **equals(string s1)**: întoarce *true* dacă sirul curent este egal cu *s1*
 - Variantă *case insensitive*: boolean **equalsIgnoreCase(string s1)**
- int **compareTo(string s1)**: întoarce 0 dacă sirul curent este egal cu *s1*, valoare negativă dacă este mai mic, sau valoare pozitivă dacă este mai mare
 - Variantă *case insensitive*: int **compareToIgnoreCase(string s1)**
- int **indexOf(string s1)**: întoarce poziția ($\Rightarrow 0$) sirului *s1* în sirul curent sau -1 dacă *s1* nu a fost găsit
- boolean **startsWith(string s1)**: întoarce *true* dacă sirul curent începe cu *s1*
 - similar boolean **endsWith(string s1)**

Şiruri de caractere

- Segmentare şiruri de caractere - `String[] split(String regex)`

```
String lista = "11,6,12,4,22,15,2";
String[] valoriLista = lista.split(",");
int[] valori = new int[valoriLista.length];
for (int index = 0; index < valori.length; index++) {
    valori[index] = Integer.parseInt(valoriLista[index]);
}
```

- Extragere subşir - `String substring(int begin, int end)`

```
String data = "20210226"; // yyyyymmdd
int zi = Integer.parseInt(data.substring(6, 8));
```

- Metode de transformare:

- `StringtoUpperCase()` – transformare litere mici în litere mari
- `StringtoLowerCase()` – transformare litere mari în litere mici
- `Stringtrim()` – eliminare spaţii de la începutul şi sfârşitul şirului

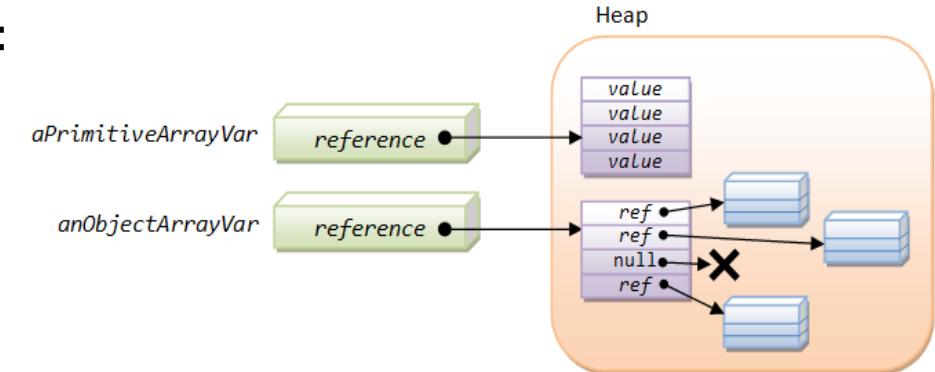
Clase – Elemente de bază

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

Masive unidimensionale - Declarare

- Masivele unidimensionale (vector) sunt obiecte care:
 - Conțin elemente de același tip (primitive sau referințe)
 - Au o dimensiune fixă specificată la initializare
- Declarare și inițializare:

```
Student[] vStudenti;           // declarare fără inițializare
int[] vector1 = new int[3];    // declarare și inițializare cu valori default
int[] vector2 = {1, 2, 4};     // declarare și inițializare cu literal
```



- Accesare elemente, accesare dimensiune vector și parcurgere secvențială:

```
int n = vector1.length;
int val = vector1[1];

for (int element : vector1) {
    /* utilizare element */
}
```

Masive unidimensionale

- **Metode statice utile** – în clasa *java.util.Arrays*:
 - int binarySearch(vector, valoare) – căutare element (-1 dacă elementul nu este găsit)
 - void sort(vector) – sortare elemente vector
 - boolean equals(vector1, vector2) – verifică dacă doi vectori sunt egali
- **Funcții cu număr variabil de argumente – *tip... denumire***
 - Au un singur parametru de acest fel, obligatoriu pe ultima poziție
 - Parametrul este tratat ca un vector în interiorul funcției

```
static int suma(int... valori) {
    int suma = 0;
    for (int element : valori) {
        suma += element;
    }
    return suma;
}
// Apeluri: suma(1,5,2); sau int[] v = {1,2,3}; suma(v)
```

Masive multidimensionale

- Pot fi declarate massive multidimensionale - comportament similar cu *vector de vector*

Exemplu:

```
int[][][] cub = {
    {{1, 2}, {3, 4}, {5, 6}},
    {{1, 2}, {3, 4}, {5, 6}}
};

int[][] m = cub[0]; // {{1, 2}, {3, 4}, {5, 6}},
int n = cub.length; // 2

// Parcuregere
for (int[][] matrice : cub) {
    for (int[] vector : matrice) {
        for (int element : vector) {
            System.out.println(element);
        }
    }
}
```

Programare multiparadigmă - JAVA

Conf. univ. dr. **Claudiu Vintă**

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

modifier 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)

- Instantiere 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 modificador) 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			

Modifierul *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 initializare
 - Dacă variabila este de tip referință atunci:
 - va referi mereu același obiect
 - valorile din interiorul obiectului *pot fi modificate*
 - Initializarea poate surveni ulterior declarării

Modificatorul *static*

- Poate fi aplicat atât metodelor cât și câmpurilor
- Comportament similar cu C++:
 - Membrii statici pot fi accesati 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 initializare:

```
class Test {  
    static int cod;  
  
    // bloc de initializare 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 accesăți 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

- Similară 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:

Clase interne

- Tipuri:
 - *static nested class*
 - Similară 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

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 obligatoriu *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 neschimbăt 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 publică metodă *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

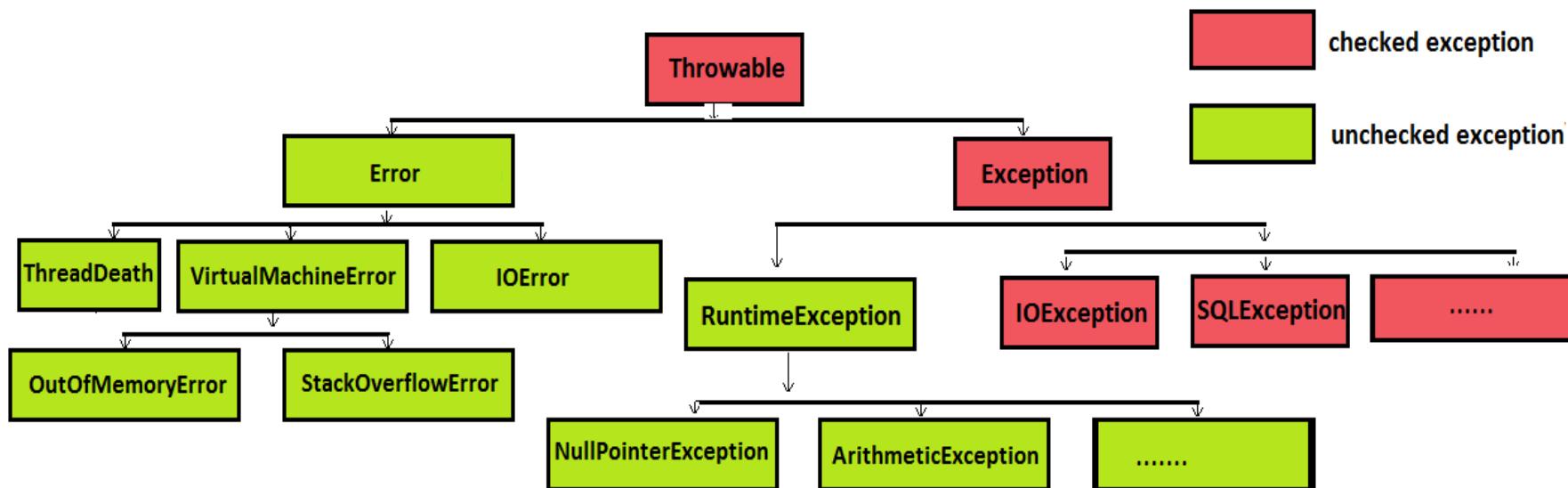
- Clase speciale derivate din *java.lang.Enum* ale cărei câmpuri sunt obligatoriu constante

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);
    }
}
```

Programare multiparadigmă - JAVA

Conf. univ. dr. **Claudiu Vintă**

claudiu.vinte@ie.ase.ro

Java *Generics*

- Mecanismul permite parametrizarea tipurilor utilizate la definirea de clase, interfețe sau metode
- Concept similar celui de *template* din C++ sau *generics* din C#
- Mecanism de implementare – eradicarea tipurilor
 - Nu există expandarea tipurilor la compilare sau la rulare
 - Fără suport în mașina virtuală
- Utilizat în special pentru definirea colecțiilor:
 - eliminare necesitate cast la extragerea elementelor
 - eliminare posibilitate de adăugare elemente de tipuri diferite în colecții

Java Generics

- Declarare clase / interfețe generice:
[modificatori] **class NumeClasa<T1, T2, ..., Tn>** { ... }
- Declarare metode generice:
[modificatori] <T1, T2, ..., Tn> tip numeMetoda(ListaParametri)
[throws E1,...,Em] {...}
- Raw type – tipul obținut prin procesul de eradicare a tipului din tipul parametrizat

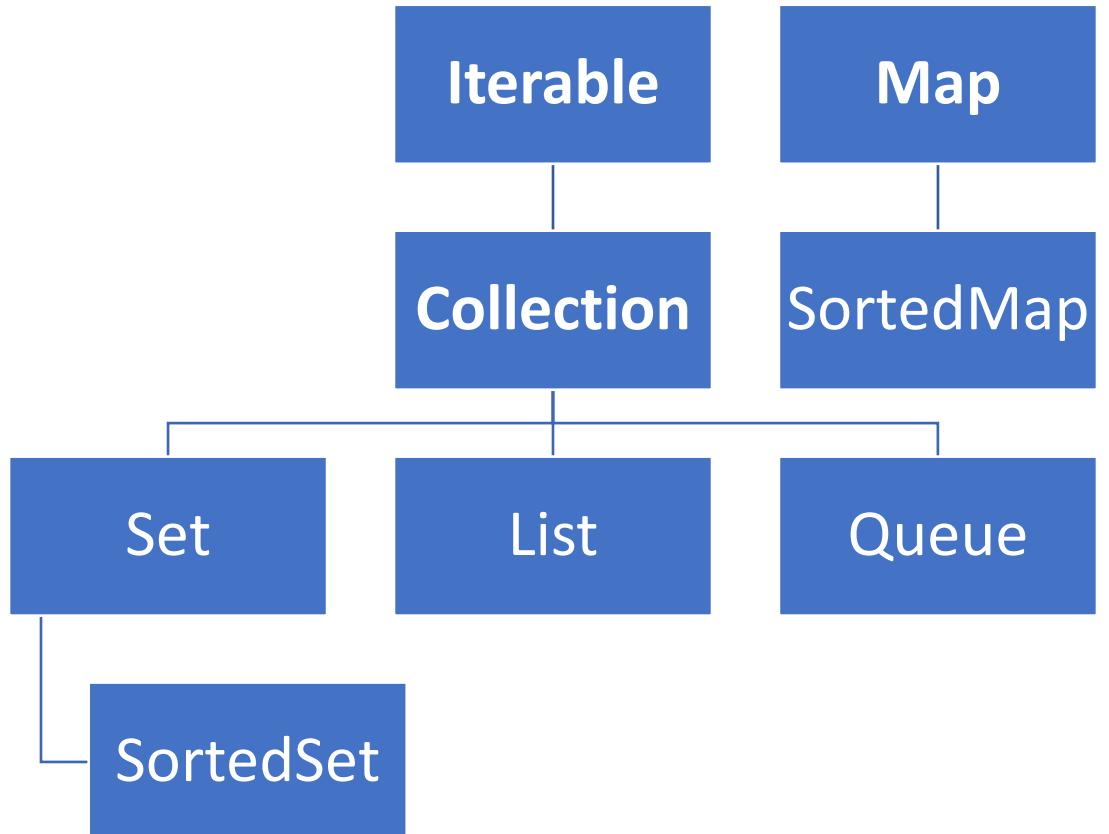
```
class Test<T> {
    public void metoda(Object element) {
        if (element instanceof T) { // eroare compilare
            // ...
        }
        T item2 = new T();          // eroare compilare
        T[] iArray = new T[10];     // eroare compilare
        T obj = (T)new Object();   // warning la compilare
    }
}
```

Java Collection Framework - JCF

- Colecție = un obiect care grupează mai multe obiecte
- JCF este compusă din:
 - Interfețe
 - Tipuri abstracte care reprezintă colecțiile
 - Permit utilizarea obiectelor colecție independent de modalitatea de implementare
 - Modul preferat de utilizare a colecțiilor
 - Implementări
 - Clase care furnizează implementări concrete ale interfețelor JCF
 - Algoritmi
 - Metode care furnizează implementări polimorfice pentru diverse operații uzuale (sortare, căutare)
 - Lucrează pe baza interfețelor JCF
- Toate componente JCF sunt generice

JCF – Interfețele de bază

- **Collection** – reprezintă un grup de obiecte (elemente) și furnizează metodele de bază pentru enumerarea, adăugarea, ștergerea și căutarea elementelor
- Colecții:
 - **Set** – nu poate conține valori duplicate
 - **List** – poate conține valori duplicate și permite accesul direct la elemente
 - **Queue** – utilizate pentru a organiza elementele în vederea prelucrării conform unei strategii (FIFO, prioritate, etc.)
- **Map** – reprezintă o colecție de perechi cheie (unică) – valoare și furnizează metodele de bază pentru citirea și manipularea perechilor
- Determinarea egalității se realizează pe baza metodei ***Object.equals***, iar ordonarea pe baza implementării interfeței ***Comparable<T>*** sau ***Comparator<T>***



JCF – Implementări de bază

Interfață	<i>Hash Table</i>	<i>Vectori</i>	<i>Arbore</i>	<i>Liste</i>	<i>Hash + Listă</i>
Set	<i>HashSet</i>		<i>TreeSet</i>		<i>LinkedHashSet</i>
List		<i>ArrayList</i>			
Queue				<i>LinkedList</i>	
Map	<i>HashMap</i>		<i>TreeMap</i>		<i>LinkedHashMap</i>

Reprezintă clasele recomandate pentru majoritatea aplicațiilor

Nu sunt clase abstracte – furnizează implementarea completă pentru interfețele respective

Permit elemente ***null*** (atât chei cât și valori)

JCF - Algoritmi

- Implementați sub formă de metode statice în clasa *java.util.Collections*
- Aranjare elemente
 - **sort** – sortare simplă sau pe bază de obiect comparator
 - **shuffle** – amestecare elemente
 - **reverse** – inversarea ordinii elementelor
- Căutare
 - **binarySearch** – căutare în liste sortate
 - **frequency** – numărare apariții element
- Manipulări de bază
 - **addAll** - adăugare elemente multiple
 - **fill** – umplere lista cu o anumită valoare
 - **copy** – copiere elemente între două liste
 - **min / max** – determinare elemente minime și maxime

Programare multiparadigmă - JAVA

Conf. univ. dr. **Claudiu Vintă**

claudiu.vinte@ie.ase.ro

Java Standard I/O (*java.io*)

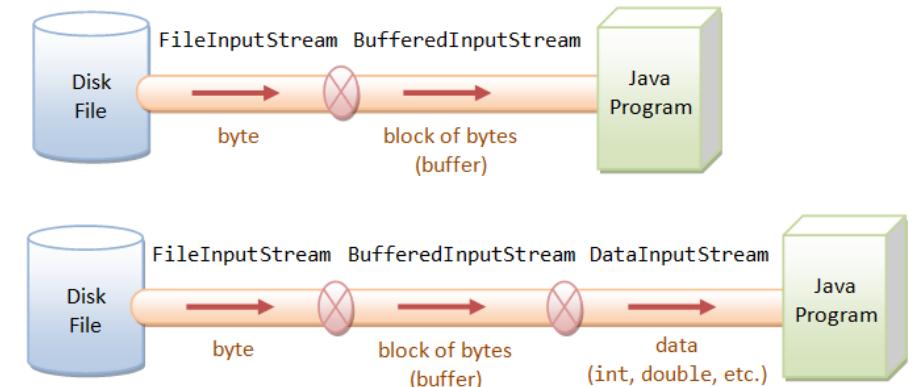
- Clasele care implementează operațiile de intrare / ieșire sunt grupate în două pachete
 - ***java.io*** – Standard I/O: conține clasele de bază pentru intrare / ieșire bazate pe obiecte stream
 - ***java.nio*** – Non-blocking I/O: adaugă suport pentru facilități avansate de intrare / ieșire
- Principalele funcționalități implementate în ***java.io***:
 - Accesul la sistemul de fișiere (creare, citire proprietăți, verificare existență, ...) pentru fișiere și directoare
 - Accesul secvențial pentru citire și scriere la conținutul diverselor surse de date
 - Procesarea datelor (decodificare caractere, compresie, ...)
 - Serializarea și deserializarea obiectelor
 - Acces aleator la datele din fișiere binare

Manipulare fișiere și directoare

- Se realizează prin intermediul obiectelor de tip ***File*** care:
 - Reprezintă o cale către un fișier sau director
 - Permite manipularea fișierelor și directoarelor, dar nu și a conținutului
- Principalele operații disponibile sunt:
 - ***isFile()*** / ***isDirectory()***: determinarea tipului căii
 - ***exists()***: determinare existență fișier sau director
 - ***length()***: determinare dimensiune în bytes pentru un fișier
 - ***File[] listFiles()***: obține obiectele descendente (fișiere și directoare); ***File getParentFile()***: obține părinte
 - ***mkdir()*** / ***mkdirs()*** sau ***createNewFile()***: construire directoare sau fișier gol
 - ***renameTo(File dest)***: redenumește fișierul sau directorul
 - ***delete()***: șterge fișierul sau directorul
 - ***canRead()*** / ***canWrite()*** / ***canExecute()***: verificare permisiuni

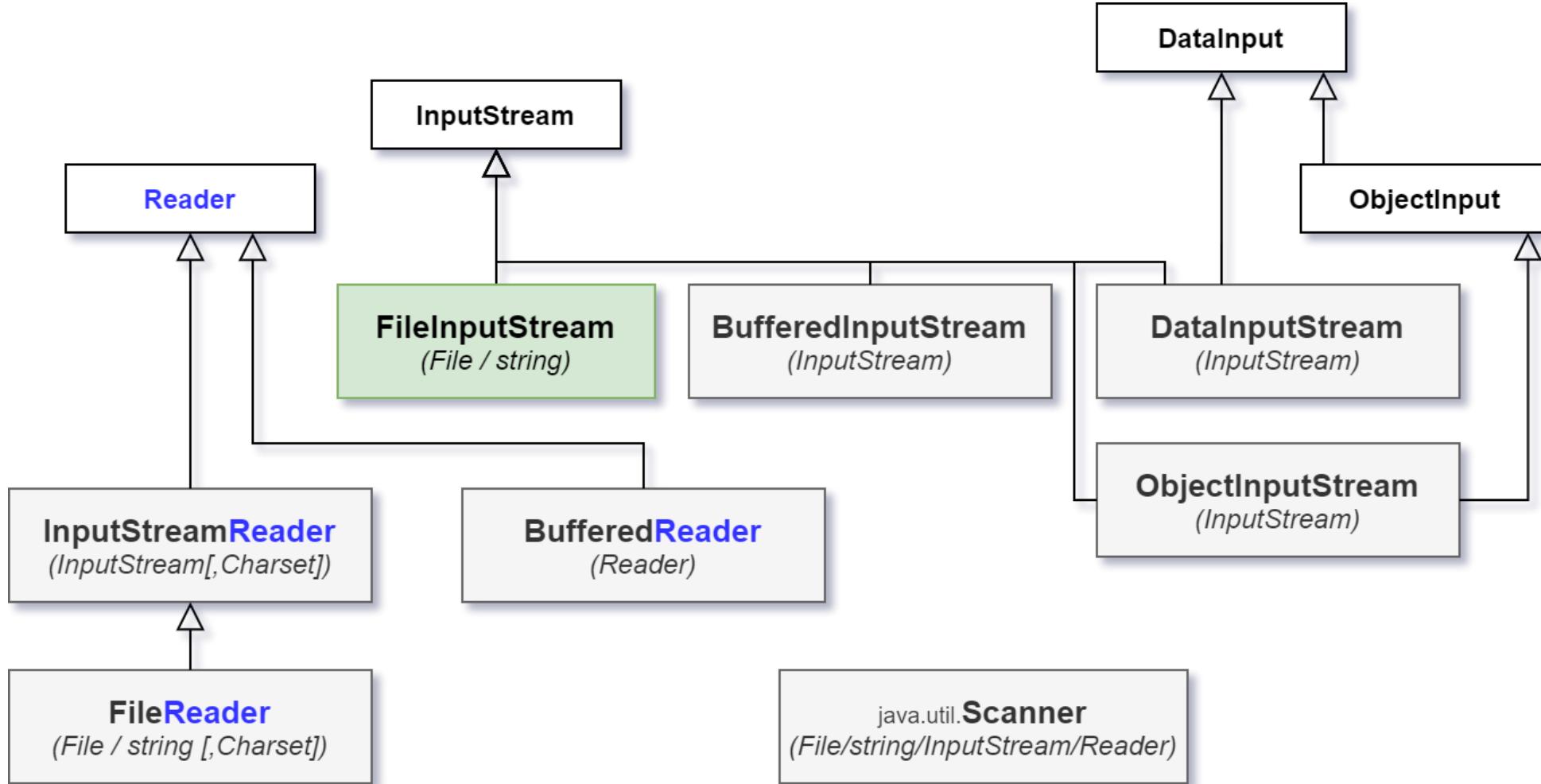
Manipularea conținutului

- Se bazează pe obiecte de tip **stream** care reprezintă o sursă (*input stream*) sau o destinație (*output stream*) pentru datele manipulate de către program.
- Permit accesul secvențial la date.
- Pot fi înlăncuite – *decorator design pattern*.
- În funcție de tipul de date manipulate se clasifică în *byte streams* și *character streams*.
- Există obiecte stream care permit accesul la o multitudine de surse de date (fișiere, rețea, *memory pipes*, ...).
- Pentru că obiectele de tip **stream** gestionează resurse externe se recomandă folosirea construcțiilor de tip **try-with-resources** pentru gestionarea acestora.

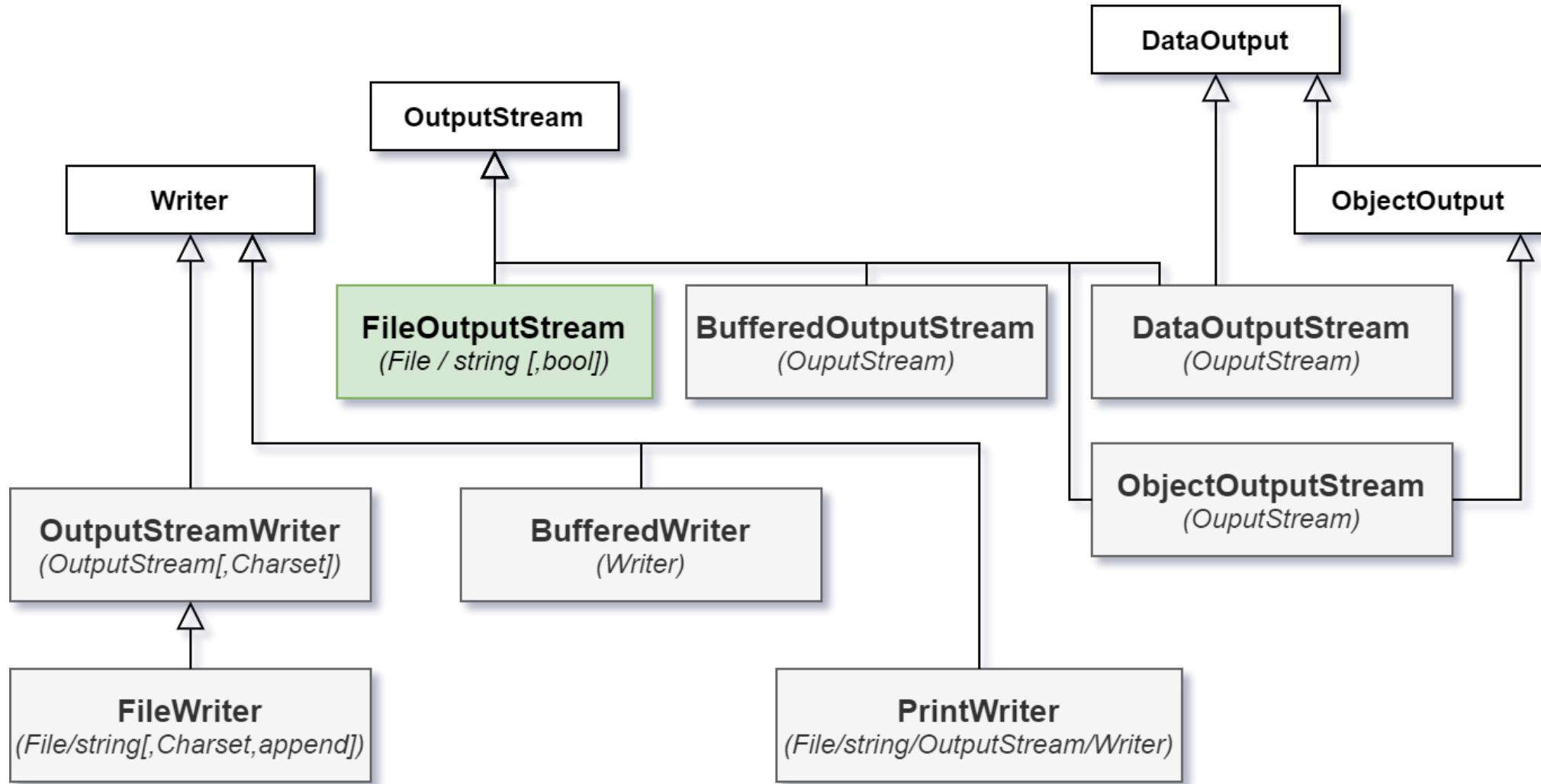


```
try (FileInputStream stream =
      new FileInputStream("date.txt")) {
    // utilizare obiect stream
} // închidere automată la părăsirea blocului
```

Structura de clase - Citire



Structura de clase - Scriere



Citirea / scrierea la nivel de octet

- Clasa abstractă ***InputStream*** – permite citirea de octeți din sursa de date
- Metoda principală este *public abstract int read() throws IOException* care:
 - Întoarce octetul citit – o valoare de la **0** la **255** în caz de succes
 - Valoarea **-1** dacă sursa de date a fost epuizată (de exemplu sfârșit de fișier)
 - Aruncă o excepție în cazul unei erori de citire
 - **Blochează execuția** până la apariția unuia dintre cele trei cazuri de mai sus
- Clasa abstractă ***OutputStream*** – permite scrierea de octeți în destinație
- Implementări concrete pentru fișiere:
 - ***FileInputStream(File fisier / string cale)*** – citire din fișier
 - ***FileOutputStream(File fisier / string cale[, boolean append])*** – scriere în fișier
- ***BufferedInputStream*** și ***BufferedOutputStream*** – decoratori care implementează citirea (dintr-un *InputStream*) și scrierea (într-un *OutputStream*) folosind zone tampon

Citirea / scrierea la nivel de caracter

- Clasa abstractă **Reader** – permite citirea de caractere din sursa de date
- Metoda principală este *public abstract int read() throws IOException* care:
 - Funcționează similar cu metoda `read` din `InputStream`
 - Întoarce caracterul citit – o valoare de la **0** la **65535** în caz de succes
- Clasa abstractă **Writer** – permite scrierea de caractere în destinație
- Legătura se realizează prin intermediul clasei ***InputStreamReader(InputStream[, Charset]***)
- Implementări concrete pentru fișiere:
 - ***FileReader(File fisier / string cale[, Charset])*** – citire din fișier
 - ***FileWriter(File fisier / string cale[, Charset, boolean append])*** – scriere în fișier
- ***BufferedReader*** și ***BufferedWriter*** – decoratori care implementează citirea (într-un `InputStream`) și scrierea (într-un `OutputStream`) folosind zone tampon
 - `BufferedReader` permite citirea de linii prin intermediul metodei ***readLine***

Fișiere text

- Clasa ***java.util.Scanner***
 - Constructor pe bază de *File* sau *InputStream* sau *Reader*
 - Permite citirea de tipuri de bază sau linii de text
- Clasa ***PrintWriter***
 - Constructor pe bază de *File* sau *OutputStream* sau *Writer*
 - Suprîncărcări pentru metoda *print* pentru scrierea tipurilor fundamentale
 - Metoda *printf* pentru scriere cu formatare

Fișiere binare

- Clasa ***DataInputStream***
 - Constructor pe bază de *InputStream*
 - Permite citirea de tipuri de bază dintr-un *stream* binar (exemplu *FileInputStream*)
 - Metode de forma *double readDouble()*, *int readInt()*, *String readUTF()*, ...
- Clasa ***DataOutputStream***
 - Constructor pe bază de *OutputStream*
 - Permite scrierea de tipuri de bază într-un *stream* binar (exemplu *FileOutputStream*)
 - Metode de forma *void writeDouble(double)*, *void writeInt(int)*, *void writeUTF(String)*, ...

Serializarea obiectelor

- Procesul de transformare a unui graf de obiecte într-un sir de octeți
- Interfața de marcat **Serializable** – serializare implicită
 - Poate fi ajustată prin intermediul metodelor *writeObject* și *readObject*
- Interfața **Externalizable** – controlul serializării prin metodele
 - *void writeExternal(ObjectOutput out) throws IOException*
 - *void readExternal(ObjectInput in) throws IOException,*
ClassNotFoundException
- Pentru scrierea și citirea din fișier se utilizează clasele **ObjectOutputStream** și **ObjectInputStream** cu metodele **writeObject** și **readObject**

Fișiere cu acces direct

- Clasa **RandomAccessFile**:
 - Permite acces bidirectional (citire și scriere) și poziționare în cadrul fluxului
 - Constructori pe bază de obiect *File* sau *String cale* și *String mode* ("r", "rw", ...)
 - Metode de poziționare
 - *void seek(long pos)*
 - *int skipBytes(int numBytes)*
 - *long getFilePointer()*
 - *long length()*
 - Metode de citire și scriere la nivel de octet similare cu *InputStream* și *OutputStream*
 - Metode de citire și scriere pentru tipuri de bază conform interfețelor *DataInputStream* și *DataOutputStream*

Programare multiparadigmă - JAVA

Conf. univ. dr. **Claudiu Vintă**

claudiu.vinte@ie.ase.ro

Interfețe – Metode *default* și *static*

- Metodele ***default*** în cadrul unei interfețe:
 - Sunt precedate de modificadorul *default*
 - Furnizează obligatoriu o implementare pe baza metodelor existente în interfață
 - Pot fi suprascrise de către clasele care implementează interfața (dar nu este obligatoriu)
 - Sunt utilizate în special pentru extinderea interfețelor existente
- Metodele ***static*** în cadrul unei interfețe:
 - Sunt precedate de modificadorul *static*
 - furnizează obligatoriu o implementare pe baza metodelor *static* existente
 - Se accesează folosind *NumeInterfata.NumeMetodaStatica*
 - Nu pot fi suprascrise în clasele care implementează interfața

Interfețe funcționale și expresii lambda

- O **interfață funcțională** este o interfață care conține:
 - Exact o metodă abstractă
 - Zero sau mai multe metode *default*
 - Zero sau mai multe metode *statische*
- Interfețele funcționale pot fi implementate folosind expresii lambda (similar claselor anonime):

$(args1, args2, \dots) \rightarrow \{ body \}$

- Valoarea unei expresii lambda este o referință la o clasă anonimă care implementează o interfață funcțională
- Interfața funcțională implementată este stabilită prin context
- Semnătura din expresia lambda trebuie să corespundă semnăturii unicei metode abstracte din interfață
- Dacă body constă într-o singură instrucțiune de forma $\{ return expresie; \}$ atunci secțiunea poate fi înlocuită cu *expresie* ($(a, b) \rightarrow \{ return a + b; \}$ este echivalent cu $(a, b) \rightarrow a + b$)
- Dacă există un singur parametru, atunci parantezele rotunde pot lipsi

Expresii lambda

- Accesarea variabilelor locale se face în aceleasi conditii ca la clasele locale sau anonte: variabilele locale utilizate în expresia lambda trebuie să fie finale sau efectiv finale (nu se modifică după initializare)
- Variabilele locale nu pot fi mascate in expresiile lambda - expresiile lambda nu creează un nou nivel de domeniu de vizibilitate
- Referinta *this* se referă la clasa care conține expresia și nu la expresie in sine
- O referință la o interfață funcțională poate fi initializată prin intermediul unei **referinte** la o **metodă** existentă într-o clasă în forma *NumeClasă::NumeMetoda*, *numeObiect::NumeMetodă* sau *NumeClasă::new* pentru constructor.
- **Exemplu:** Program01_InterfeteFunctionaleLambda.java

Interfețe funcționale standard

- ***Function<T, R>*** - funcție *apply* care primește un parametru de tip T și întoarce o valoare de tip R
- ***Predicate<T>*** - funcție *test* care primește un parametru de tip T și întoarce o valoare booleană
- ***UnaryOperator<T>*** - funcție *apply* care primește un parametru de tip T și întoarce o valoare de tip T
- ***BinaryOperator<T>*** - funcție *apply* care primește doi parametri de tip T și întoarce o valoare de tip T
- ***Consumer<T>*** - funcție *accept* care primește un parametru de tip T și consumă valoarea fără a întoarce un rezultat
- ***Supplier<T>*** - funcție *get* care nu primește parametri și generează un rezultat de tip T

Compunere interfețe funcționale

- Operația de compunere presupune combinarea mai multor funcții într-o singură care utilizează intern funcțiile componente
- **Compunerea funcțiilor** (Function<T, R>)
 - $h = f.\text{compose}(g)$: funcția h va calcula valoarea finală aplicând întâi funcția g asupra parametrului, după care funcția f pe rezultatul întors de g ;
 - $h = f.\text{andThen}(g)$: similar cu *compose*, dar ordinea de aplicare va fi inversată (întâi f , după care g)
- **Compunerea predicatorilor** (Predicate<T>)
 - $h = f.\text{or}(g)$: funcția h va aplica funcțiile f și g pe valoarea primită și va întoarce rezultatele combinate prin intermediul operatorului logic OR
 - $h = f.\text{and}(g)$: funcția h va aplica funcțiile f și g pe valoarea primită și va întoarce rezultatul combinate prin intermediul operatorului logic AND
- **Exemplu:** Program02_InterfeteStandard.java

Java Stream API

- Definit în pachetul *java.util.stream* – permite manipularea în stil funcțional pentru colecții
- Un *stream* este un obiect care poate parcurge și procesa, secvențial sau paralel, o colecție și implementează interfața ***BaseStream<T, S extends BaseStream<T,S>>***
- Caracteristici:
 - Nu este o structură de date; datele propriu-zise se află într-un vector, colecție, canal IO, ...
 - Nu modifică structura de date (nu pot adăuga / șterge elemente)
 - Operațiile specificate se execută doar la finalul procesării (la întâlnirea unei operații terminale)
- Metode de obținere:
 - Metoda *stream()* disponibilă în clasele colecție
 - Metoda *lines()* din clasa *BufferedReader*
 - Metoda statică *of()* din clasa *Stream*
 - Metoda statică *stream()* din clasa *Arrays*

Java Stream API

- **Operații intermediare** – au ca rezultat un alt obiect *stream* și pot fi înălțuite:
 - *map(Function)* – rezultatul aplicării funcției pe elementele colecției originale; rezultatul va avea același număr de elemente, dar cu tip și / sau valori diferite
 - *filter(Predicate)* – un *stream* cu un subset al elementelor colecției originale
 - *distinct()* – un *stream* cu elementele distincte din colecția originală
 - *sorted(Comparator)* – un *stream* cu elementele originale sortate în funcție de comparatorul furnizat
 - *flatMap(Function)* – produce un *stream* care conține concatenarea rezultatului aplicării funcției pe elementele colecției; funcția trebuie să întoarcă un *stream* pentru fiecare element;
 - *limit(int)* – întoarce un *stream* care conține maxim numărul de elemente specificate

Java Stream API

- **Operații terminale** – produc rezultatul final și declanșează procesarea:
 - *collect (Collector)* – furnizează elementele procesate într-o colecție
 - *toArray()* – furnizează elementele procesate într-un vector
 - *count()* – returnează numărul de elemente din colecția procesată
 - *anyMatch/allMatch/noneMatch(Predicate)* – întorc o valoare booleană
 - *forEach(Consumer)* – aplică funcția furnizată pentru fiecare element din *stream*
 - *reduce(T, BinaryOperator<T>)* – agregă elementele din *stream* pe baza funcției asociative furnizate
- Operațiile pe obiecte stream se pot executa concurent folosind operația intermediară *parallel()*
- **Exemplu:** Program03_Streams.java

Programare multiparadigmă - JAVA

Conf. univ. dr. **Claudiu Vintă**

claudiu.vinte@ie.ase.ro

Fire de execuție

- Un **fir de execuție** sau **thread**
 - specifică execuția secvențială a unui set de instrucțiuni
 - partajează același spațiu de adrese cu celelalte fire de execuție din cadrul procesului
- Fiecare fir de execuție deține o stivă proprie.
- Zona heap și zonele statice sunt partajate de către toate firele de execuție din cadrul procesului
- Firele de execuție sunt create prin intermediul clasei *java.lang.Thread*

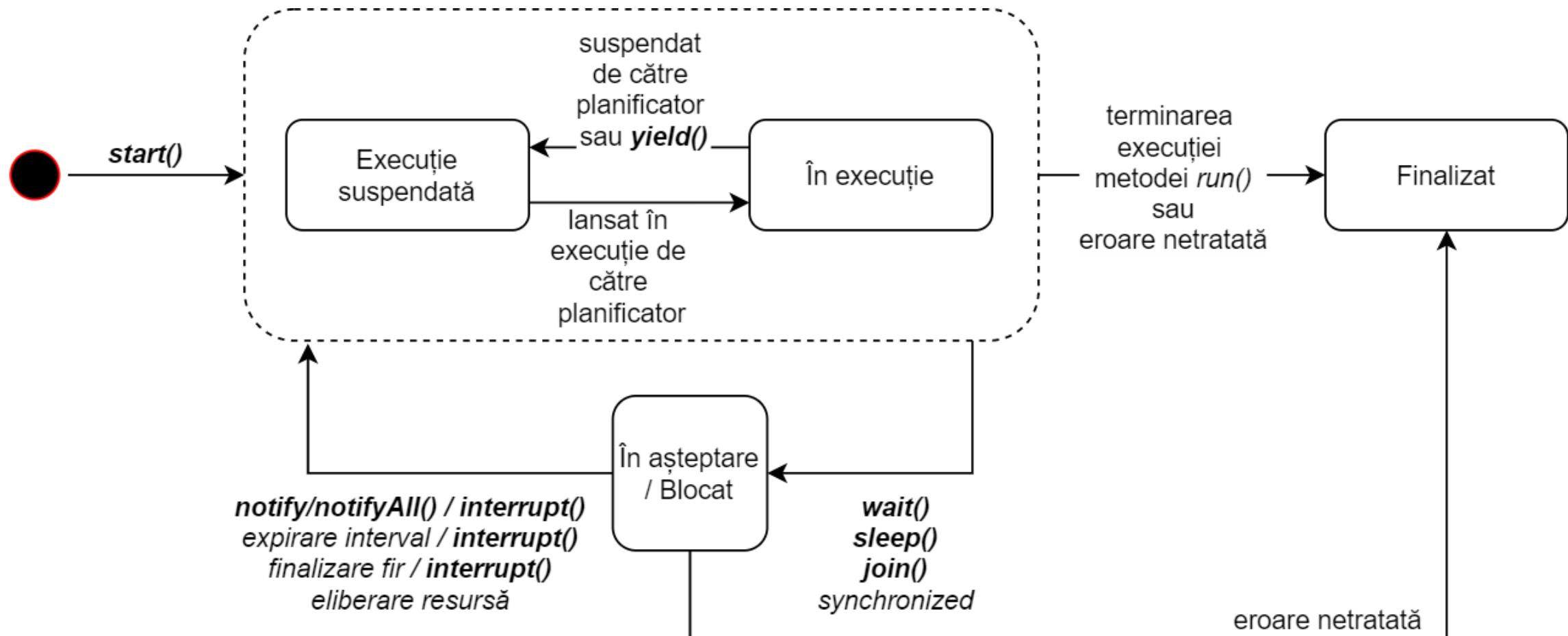
Crearea firelor de execuție

- Varianta 1: Prin **derivare** din clasa ***Thread*** și suprascrierea metodei ***run***
- Varianta 2: Prin implementarea interfeței ***Runnable*** și utilizarea unui obiect de tip ***Thread***

```
class ThreadVarianta1 extends Thread {  
    @Override  
    public void run() {  
        // cod fir execuție...  
    }  
}  
  
public class Scratchpad {  
    public static void main(String[] args) {  
        Thread thread1 = new ThreadVarianta1();  
        thread1.start();  
    }  
}
```

```
class ThreadVarianta2 implements Runnable {  
    public void run() {  
        // cod fir execuție...  
    }  
}  
  
public class Scratchpad {  
    public static void main(String[] args) {  
        Thread thread2 = new Thread(new ThreadVarianta2());  
        thread2.start();  
    }  
}
```

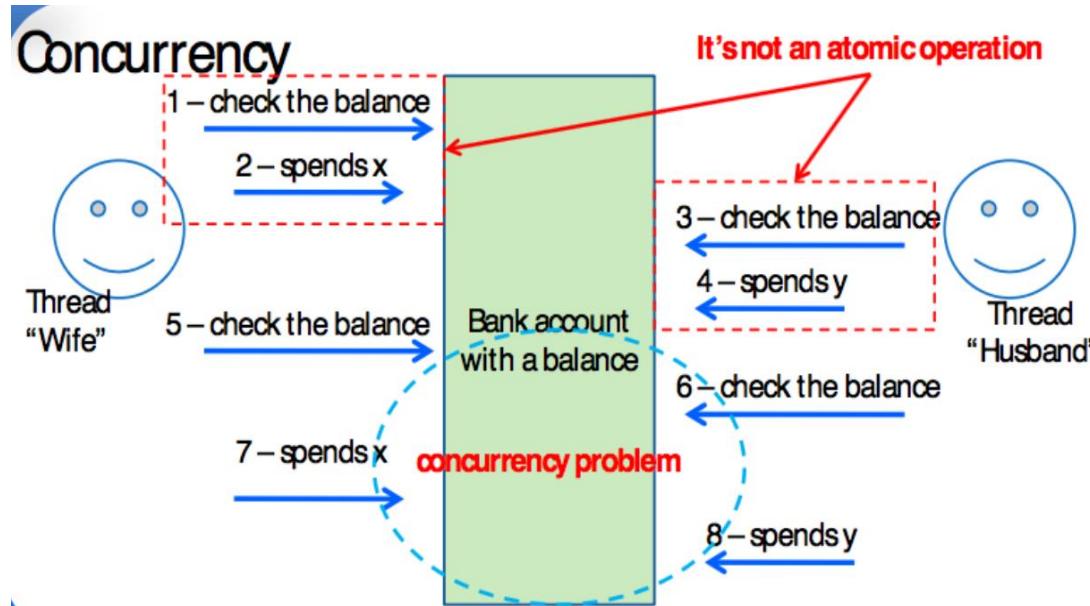
Stări și operații de bază



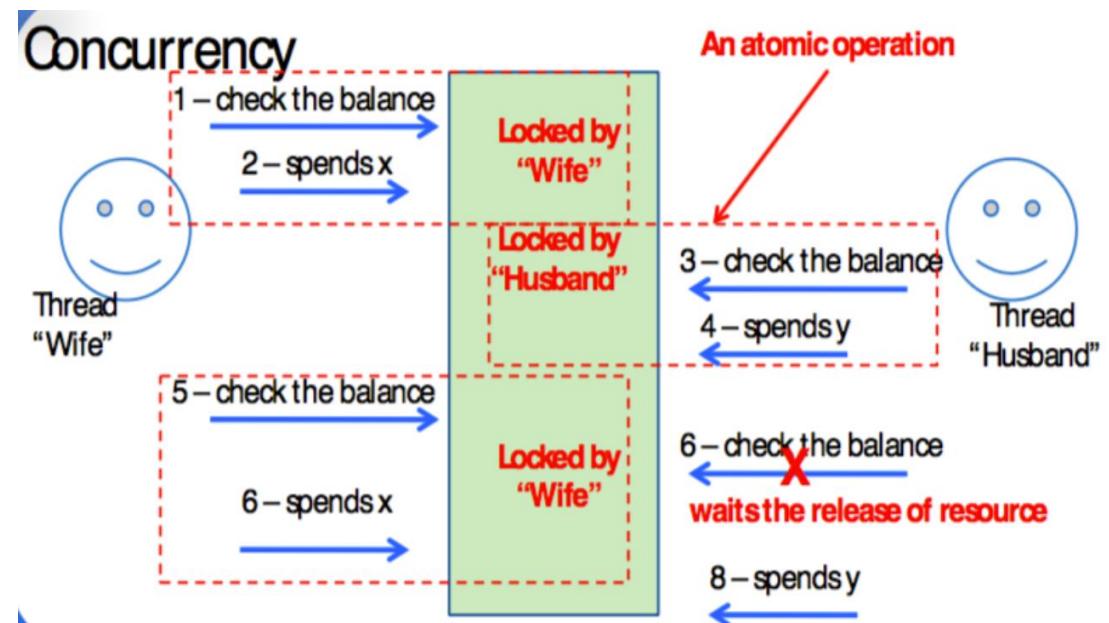
Operații de bază

- *void start()* – pornește firul de execuție; se va executa metoda **run** din firul de execuție specificat în paralel cu firul de execuție curent
- **static void sleep(long millis)** – oprește execuția firului de execuție **current** până la expirarea intervalului de timp specificat sau până când este întrerupt
- *void interrupt()* – întrerupe firul de execuție **specificat**; generează *InterruptedException* în firul de execuție specificat dacă firul acesta este în așteptare (apel *sleep / wait / join*)
- *void join()* – oprește execuția firului de execuție **current** până la terminarea execuției firului de execuție **specificat**
- **static void yield()** – transmite planificatorului că firul de execuție **current** este dispus să renunțe la timpul său de procesor

Sincronizare acces



Exemplu utilizare cont bancar familial



Sincronizare acces

- **Race condition** – mai multe fire de execuție:
 - acceseză o resursă comună
 - rezultatul depinde de ordinea execuției
- Mecanisme de sincronizare:
 - Pentru citiri și scrieri *individuale* pentru tipuri fundamentale se poate utiliza **volatile**
 - *Blocuri synchronized (obiect) {...cod...}* – permite execuția codului doar de către firul care deține controlul obiectului; un fir preia controlul la intrarea în bloc (dacă obiectul este liber) și îl cedează la ieșirea din bloc; dacă obiectul nu este liber firul de execuție este blocat până la eliberarea acestuia
 - *Modifierul synchronized la nivel de metodă* – permite execuția oricărei metode **synchronized** doar de către un fir de execuție la un moment dat; similar cu includerea codului metodei într-un bloc *synchronized(this) {}*

Sincronizare acces

- Problema principală – **deadlock**: fiecare membru al unui grup de fire de execuție așteaptă după o resursă blocată de alt membru din grup
- Condiții necesare și suficiente - Edward G. Coffman Jr.:
 - Mutual exclusion – resursele implicate nu sunt partajabile
 - Hold and wait – un fir deține controlul asupra unei resurse și cere controlul asupra alteia
 - No preemption – resursele pot fi eliberate doar de către firele care le dețin (fără drept de preemptiune)
 - Circular wait – fiecare fir așteaptă o resursă care este deținută de către alt fir
- Soluții:
 - Blocarea resurselor într-o ordine prestabilită la începutul tuturor operațiilor
 - Introducerea unui obiect arbitru (centralizarea cererilor de blocare)

Metodele *wait / notify*

- Clasa *Object* conține metode pentru facilitarea comunicării între fire de execuție:
- *void wait()* – pune firul curent în aşteptare până la primirea unei notificări sau întreruperi
 - Obligatoriu, firul curent trebuie să aibă controlul asupra obiectului (prin *synchronized*)
 - Firul curent cedează controlul obiectului pe durata suspendării
 - La primirea notificării execuția continuă doar după ce obiectul a recâștigat controlul
- *void notifyAll()* – notifică toate firele blocate de către un apel *wait* pe obiectul curent
 - Obligatoriu, firul curent trebuie să aibă controlul asupra obiectului (prin *synchronized*)
- *void notify()* – notifică aleator unul dintre firele blocate de către un apel *wait* pe obiectul curent
 - Obligatoriu, firul curent trebuie să aibă controlul asupra obiectului (prin *synchronized*)
- Exemplu de utilizare: scenariul producător - consumator

Programare multiparadigmă - JAVA

Conf. univ. dr. **Claudiu Vintă**

claudiu.vinte@ie.ase.ro

Fire de execuție

- Un **fir de execuție** sau **thread**
 - specifică execuția secvențială a unui set de instrucțiuni
 - partajează același spațiu de adrese cu celelalte fire de execuție din cadrul procesului
- Fiecare fir de execuție deține o stivă proprie.
- Zona heap și zonele statice sunt partajate de către toate firele de execuție din cadrul procesului
- Firele de execuție sunt create prin intermediul clasei *java.lang.Thread*

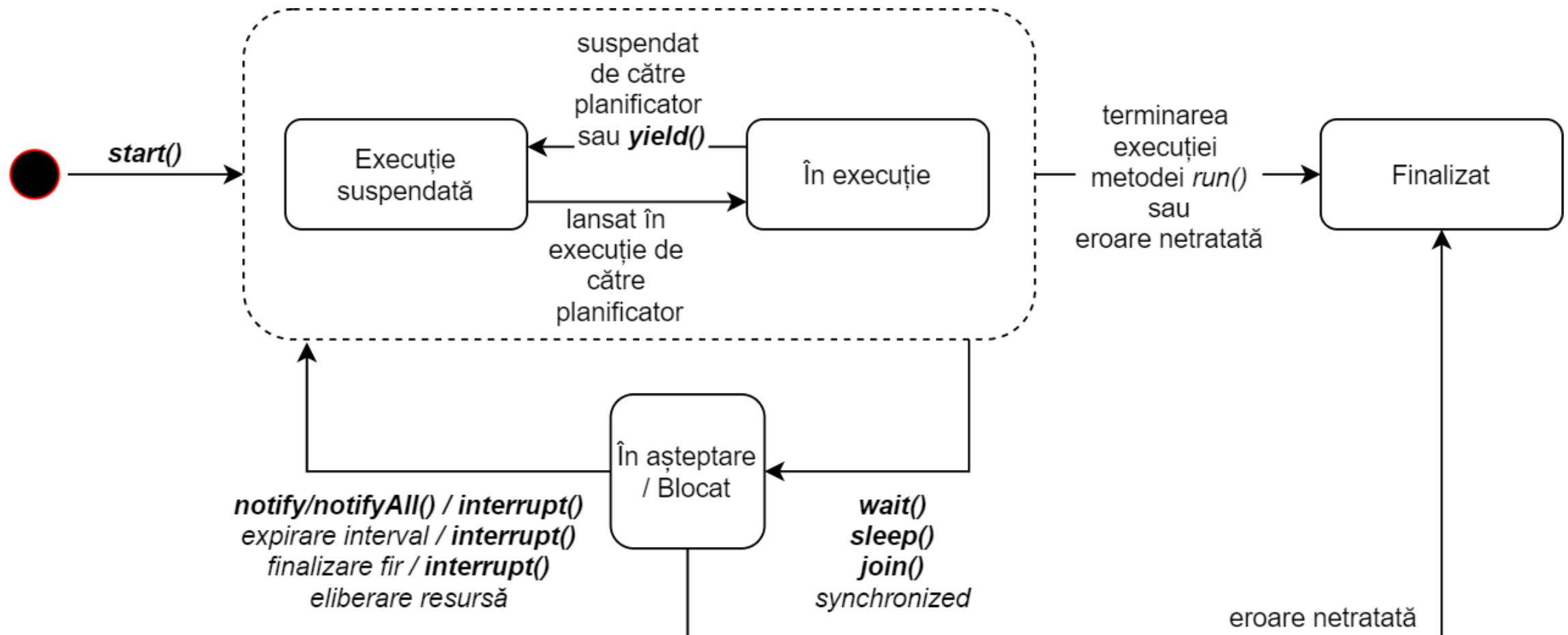
Crearea firelor de execuție

- Varianta 1: Prin **derivare** din clasa ***Thread*** și suprascrierea metodei ***run***
- Varianta 2: Prin implementarea interfeței ***Runnable*** și utilizarea unui obiect de tip ***Thread***

```
class ThreadVarianta1 extends Thread {  
    @Override  
    public void run() {  
        // cod fir execuție...  
    }  
}  
  
public class Scratchpad {  
    public static void main(String[] args) {  
        Thread thread1 = new ThreadVarianta1();  
        thread1.start();  
    }  
}
```

```
class ThreadVarianta2 implements Runnable {  
    public void run() {  
        // cod fir execuție...  
    }  
}  
  
public class Scratchpad {  
    public static void main(String[] args) {  
        Thread thread2 = new Thread(new ThreadVarianta2());  
        thread2.start();  
    }  
}
```

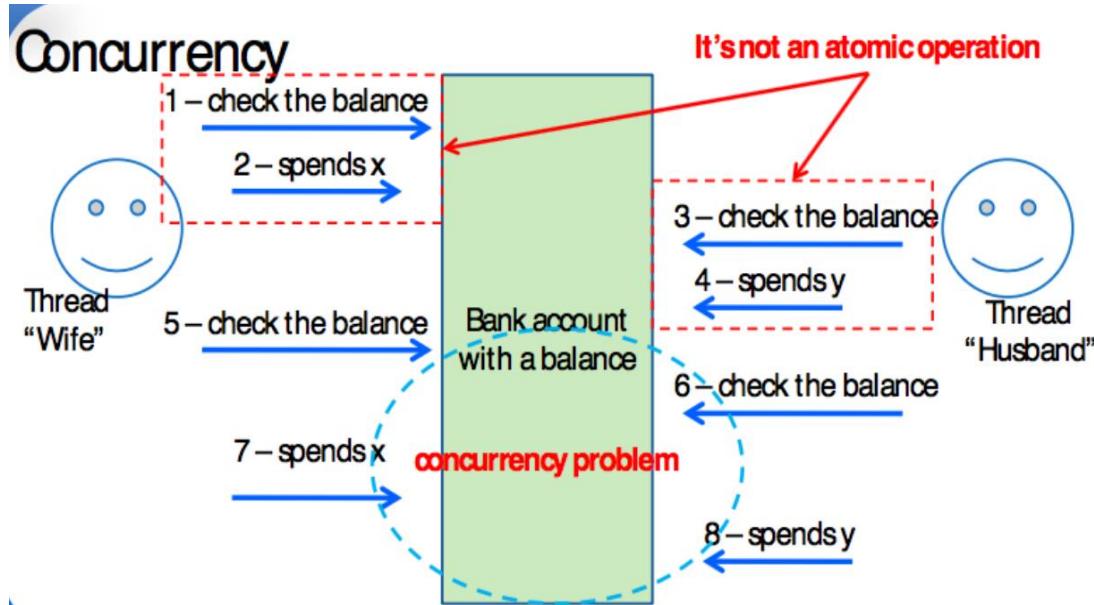
Stări și operații de bază



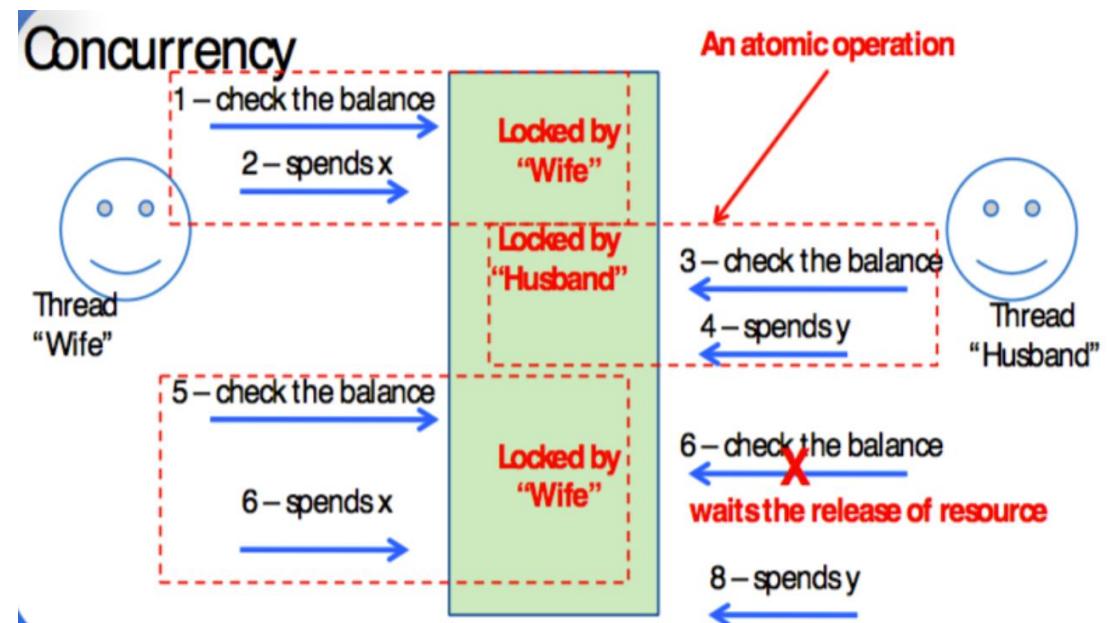
Operații de bază

- *void start()* – pornește firul de execuție; se va executa metoda **run** din firul de execuție specificat în paralel cu firul de execuție curent
- **static void sleep(long millis)** – oprește execuția firului de execuție **current** până la expirarea intervalului de timp specificat sau până când este întrerupt
- *void interrupt()* – întrerupe firul de execuție **specificat**; generează *InterruptedException* în firul de execuție specificat dacă firul acesta este în așteptare (apel *sleep / wait / join*)
- *void join()* – oprește execuția firului de execuție **current** până la terminarea execuției firului de execuție **specificat**
- **static void yield()** – transmite planificatorului că firul de execuție **current** este dispus să renunțe la timpul său de procesor

Sincronizare acces



Exemplu utilizare cont bancar familial



Sincronizare acces

- **Race condition** – mai multe fire de execuție:
 - acceseză o resursă comună
 - rezultatul depinde de ordinea execuției
- Mecanisme de sincronizare:
 - Pentru citiri și scrieri *individuale* pentru tipuri fundamentale se poate utiliza **volatile**
 - *Blocuri synchronized (obiect) {...cod...}* – permite execuția codului doar de către firul care deține controlul obiectului; un fir preia controlul la intrarea în bloc (dacă obiectul este liber) și îl cedează la ieșirea din bloc; dacă obiectul nu este liber firul de execuție este blocat până la eliberarea acestuia
 - *Modifierul synchronized la nivel de metodă* – permite execuția oricărei metode **synchronized** doar de către un fir de execuție la un moment dat; similar cu includerea codului metodei într-un bloc *synchronized(this) {}*

Sincronizare acces

- Problema principală – **deadlock**: fiecare membru al unui grup de fire de execuție așteaptă după o resursă blocată de alt membru din grup
- Condiții necesare și suficiente - Edward G. Coffman Jr.:
 - Mutual exclusion – resursele implicate nu sunt partajabile
 - Hold and wait – un fir deține controlul asupra unei resurse și cere controlul asupra alteia
 - No preemption – resursele pot fi eliberate doar de către firele care le dețin (fără drept de preemptiune)
 - Circular wait – fiecare fir așteaptă o resursă care este deținută de către alt fir
- Soluții:
 - Blocarea resurselor într-o ordine prestabilită la începutul tuturor operațiilor
 - Introducerea unui obiect arbitru (centralizarea cererilor de blocare)

Metodele *wait / notify*

- Clasa *Object* conține metode pentru facilitarea comunicării între fire de execuție:
- *void wait()* – pune firul curent în aşteptare până la primirea unei notificări sau întreruperi
 - Obligatoriu, firul curent trebuie să aibă controlul asupra obiectului (prin *synchronized*)
 - Firul curent cedează controlul obiectului pe durata suspendării
 - La primirea notificării execuția continuă doar după ce obiectul a recâștigat controlul
- *void notifyAll()* – notifică toate firele blocate de către un apel *wait* pe obiectul curent
 - Obligatoriu, firul curent trebuie să aibă controlul asupra obiectului (prin *synchronized*)
- *void notify()* – notifică aleator unul dintre firele blocate de către un apel *wait* pe obiectul curent
 - Obligatoriu, firul curent trebuie să aibă controlul asupra obiectului (prin *synchronized*)
- Exemplu de utilizare: scenariul producător - consumator

Executors

- Permit separarea logicii de execuție de codul care implementează sarcina
- Principalele interfețe:
 - ***java.util.concurrent.Executor***
 - ***void execute(Runnable command)***: execută un task la un moment de timp din viitor
 - ***java.util.concurrent.ExecutorService***
 - Future<?> submit(Runnable task) / Future<T> submit(Callable<T> task): execută un task la un moment de timp din viitor și permite obținerea valorii rezultate, anularea task-ului și verificarea stării acestuia
 - ***void shutdown()***: oprește execuția (nu mai permite executarea de task-uri noi; se poate aștepta finalizarea celor existente prin metoda ***awaitTermination***)
 - Construire obiecte: prin intermediul metodelor statice din clasa ***java.util.concurrent.Executors*** (***newCachedThreadPool()***, ***newFixedThreadPool(n)***, ***newSingleThreadExecutor()***, ...)

Fork / Join

- Presupune descompunerea problemei în subprobleme independente care se pot rezolva în paralel și combinarea ulterioră a rezultatelor
- Utilizare:
 - Se construiește o clasă derivată din **RecursiveTask<T>** (unde T este tipul rezultatului) / **RecursiveAction**
 - Se suprascrie metoda **compute()**
 - Dacă dimensiunea problemei este suficient de mică se rezolvă direct
 - Altfel se împarte problema în subprobleme care
 - se lansează în execuție prin metoda **fork**
 - se obțin rezultatele prin apelul metodei **join**
 - Se construiește un obiect de tip **ForkJoinPool**
 - Se construiește un obiect din clasa derivată și se lansează în execuție folosind metoda **invoke**

Utilizarea colecțiilor

- Java Collection Framework
 - Clasele din JCF pot fi utilizate simultan din mai multe fire de execuție doar pentru citire
 - Soluții pentru scriere din mai multe fire de execuție:
 - Folosirea de secțiuni critice (blocuri *synchronized*) pentru serializarea accesului
 - Folosirea de colecții sincronizate – se pot obține pe baza unei colecții existente prin apelarea metodelor statice *synchronizedList* / *synchronizedMap* / *synchronizedSet* din clasa *java.util.Collections*
 - Operațiile pe aceste obiecte sunt sincronizate automat; dacă un *thread* parcurge o colecție folosind un iterator și alt *thread* modifică structural colecția se va genera o excepție
- Concurrent Collections
 - Se regăsesc în pachetul *java.util.concurrent*
 - Permit manipularea din mai multe fire de execuție simultan
 - Oferă o performanță superioară față de colecțiile sincronizate
 - Exemple de interfețe / clase: *BlockingQueue* / *ArrayBlockingQueue*, *ConcurrentMap* / *ConcurrentHashMap*

Programare multiparadigmă - JAVA

Conf. univ. dr. **Claudiu Vintă**

claudiu.vinte@ie.ase.ro

Java *Reflection* (introspectie)

- Permite unui program în execuție operații de tipul:
 - Examinează un obiect oarecare pentru a determina clasa din care face parte
 - Să obțină informații despre membrii unei clase (câmpuri, constructori, metode etc.)
 - Să invoce dinamic metode
 - Să manipuleze informațiile stocate în câmpurile unui obiect
 - Să citească metadatele (adnotările) asociate unei clase sau membru al acesteia
- Implementarea sistemului de introspectie în Java este realizat în principal prin:
 - Clasa *java.lang.Class*
 - Clasele grupate în pachetul *java.lang.reflect*

Java *Reflection* - Utilizare

- Exemple de scenarii de utilizare pentru introspectie:
 - Serializare
 - Testare automată
 - Vizualizarea datelor în *debugger*
 - Generare automată pentru interfețe grafice
 - Editare vizuală pentru interfețe grafice
- Toate scenariile de mai sus au în comun elemente precum:
 - Lucrează cu obiecte din clase necunoscute la momentul dezvoltării instrumentului
 - Nu se pot limita la o interfață comună care să specifiche funcționalitățile obiectelor
 - Utilizează obiectele primite ca sursă de date
- Dezavantaje principale:
 - Performanța (de exemplu citirea valorii unui câmp prin introspectie este mult mai lentă față de citirea directă)
 - Permite evitarea (încărcarea) mecanismului de încapsulare

Java *Reflection* - Clase

- Toate obiectele și primitivele din Java au asociată o instanță imutabilă de obiect de tip *java.lang.Class* care permite aplicației să examineze la execuție un obiect oarecare.
- Metode de obținere pentru obiectul *Class*:
 - Metoda *final Class<?> getClass()* moștenită din clasa *Object* pentru un obiect existent
 - Folosind *denumireTip.class* (exemplu *boolean.class* sau *Persoana.class*)
 - Folosind metoda statică *static Class<?> forName(String className)* din clasa *Class* (în cazul în care se cunoaște denumirea clasei)
 - Diverse alte metode disponibile în clasa *Class* (*getSuperclass*, *getClasses*, *getDeclaringClass*) sau în clasele asociate membrilor

```
Persoana ion = new Persoana();
Class<?> clasaPersoana1 = ion.getClass();
Class<?> clasaPersoana2 = Persoana.class;
Class<?> clasaString = Class.forName("java.lang.String");
```

Java *Reflection* - Clase

Principalele operații disponibile sunt:

- 1) Obținere informații generale despre clasă:
 - Modificatori aplicați clasei (public, final, abstract, ...) – *getModifiers*
 - Adnotările aplicate clasei – *getAnnotations*
 - Obținerea informațiilor despre clasa de bază și interfețele implementate
- 2) Obținere informații despre membrii clasei
 - Câmpuri: *getDeclaredField* / *getField* / *getDeclaredFields* / *getFields*
 - Metode: *getDeclaredField* / *getField* / *getDeclaredFields* / *getFields*
 - Constructori: *getDeclaredField* / *getField* / *getDeclaredFields* / *getFields*
- 3) Construire obiecte folosind metoda *newInstance*

Java *Reflection* – Câmpuri

- Clasa *java.lang.reflect.Field* conține informațiile referitoare la un câmp dintr-o clasă
- Obținerea obiectelor asociate câmpurilor se realizează prin intermediul metodelor din Class
- Operații de bază:
 - Determinarea tipului
 - Obținerea valorii
 - Modificarea valorii
- Exemplu:

```
Persoana ion = new Persoana();
Class<?> clasaPersoana = ion.getClass();

Field campCod = clasaPersoana.getDeclaredField("cod");
System.out.println(campCod.getName() + " - " + campCod.getType().getName());
System.out.println(campCod.getInt(ion));
campCod.setInt(ion, 13);
System.out.println(campCod.getInt(ion));
```

Java *Reflection* – Metode

- Clasa *java.lang.reflect.Method* conține informațiile referitoare la o metodă dintr-o clasă
- Obținerea obiectelor asociate metodelor se realizează prin intermediul metodelor din *Class*
- Operații de bază:
 - Determinarea tipului returnat, precum și a tipului și denumirilor parametrilor
 - Determinarea modificatorilor aplicați metodei
 - Invocarea (apelul) metodei
- Exemplu:

```
for (Method metoda : clasaPersoana.getDeclaredMethods()) {  
    System.out.println(metoda.getName() + " -> " + metoda.getReturnType().getName());  
}  
  
Method setNume = clasaPersoana.getDeclaredMethod("setNume", String.class);  
setNume.invoke(ion, "Ion");
```

Java *Reflection* – Constructori

- Clasa *java.lang.reflect.Constructor* conține informațiile referitoare constructorii unei clase
- Obținerea obiectelor asociate constructorilor se realizează prin intermediul metodelor din *Class*
- Operații de bază:
 - Determinarea modificatorilor, tipului și denumirilor parametrilor
 - Invocarea (apelul) constructorului pentru crearea de noi obiecte
- Exemplu:

```
for (Constructor<?> constructor : clasaPersoana.getConstructors())
{
    System.out.println("Constructor cu parametri:");
    for (Parameter parametru : constructor.getParameters()) {
        System.out.println(parametru.getType().getName() + " " + parametru.getName());
    }
}

var constructor = clasaPersoana.getConstructor(int.class, String.class);
Persoana maria = (Persoana) constructor.newInstance(2, "Maria");
System.out.println(maria);
```

Java – Adnotări

- Reprezintă metadate asociate secțiunilor de program care pot fi utilizate ulterior:
 - La compilare – de către compilator pentru detectare de erori, suprimarea unor avertizări,
 - De către diverse instrumente – generarea automată de cod, documentație, fișiere de date etc.
 - La execuție – de către aplicație în combinație cu *Reflection*
- Adnotări predefinite:
 - @Override – indică compilatorului că se intenționează suprascrierea unei metode (va genera o eroare de compilare dacă nu există o metodă corespunzătoare pentru suprascriere)
 - @SuppressWarnings – permite dezactivarea selectivă de avertizări la compilare
 - @Deprecated – marchează o metodă ca fiind nerecomandată (va genera o avertizare la compilare în cazul în care este apelată)
 - @FunctionalInterface – marchează o interfață ca fiind de tip funcțional

Adnotări - Declarare

- Se realizează similar cu interfețele folosind cuvântul cheie **@interface**
- Tipuri de adnotări:

- Simple – nu conțin informații (doar pentru marcaj)

public @interface AdnotareSimpla {}

- Single Element – conțin exact o valoare

public @interface AdnotareValoare { String value(); }

- Multi-value – pot conține mai multe valori

• *public @interface AdnotareValori { String mesaj(); int repetari(); }*

Adnotări - Aplicare

- Adnotările pot fi aplicate:
 - Declarațiilor: clase, câmpuri, metode, ...
 - La utilizarea tipurilor: creare de obiecte, operații de cast, ...
- Exemplu:

```
@AdnotareSimpla
@AdnotareValoare("Test")
@AdnotareValori(mesaj = "Test", repetari = 12)
class Test {
    @AdnotareSimpla
    int numar;
}
```

Adnotări - Utilizare

- Utilizarea adnotărilor și a valorilor conținute se realizează prin introspectie folosind clasele care implementează interfața *AnnotatedElement* (*Class*, *Constructor*, *Field*, *Method*, *Package*, *Parameter*)
- Principalele metode disponibile sunt:
 - *T getAnnotation(Class<T>)*: permite obținerea adnotării de tipul specificat dacă adnotarea este prezentă direct sau într-o clasă de bază (dacă adnotarea este declarată ca moștenibilă)
 - *Annotation[] getAnnotations()*: permite obținerea tuturor adnotărilor care sunt aplicate direct sau într-o clasă de bază (dacă adnotarea este declarată ca moștenibilă)
 - *T[] getAnnotationsByType(Class<T>)*: permite obținerea tuturor adnotărilor de tipul specificat care sunt aplicate direct sau într-o clasă de bază (dacă adnotarea este declarată ca moștenibilă)
- Similar există și *getDeclaredAnnotation*, *getDeclaredAnnotations* și *getDeclaredAnnotationsByType* care furnizează doar adnotările aplicate direct (fără cele moștenite)

Programare multiparadigmă - JAVA

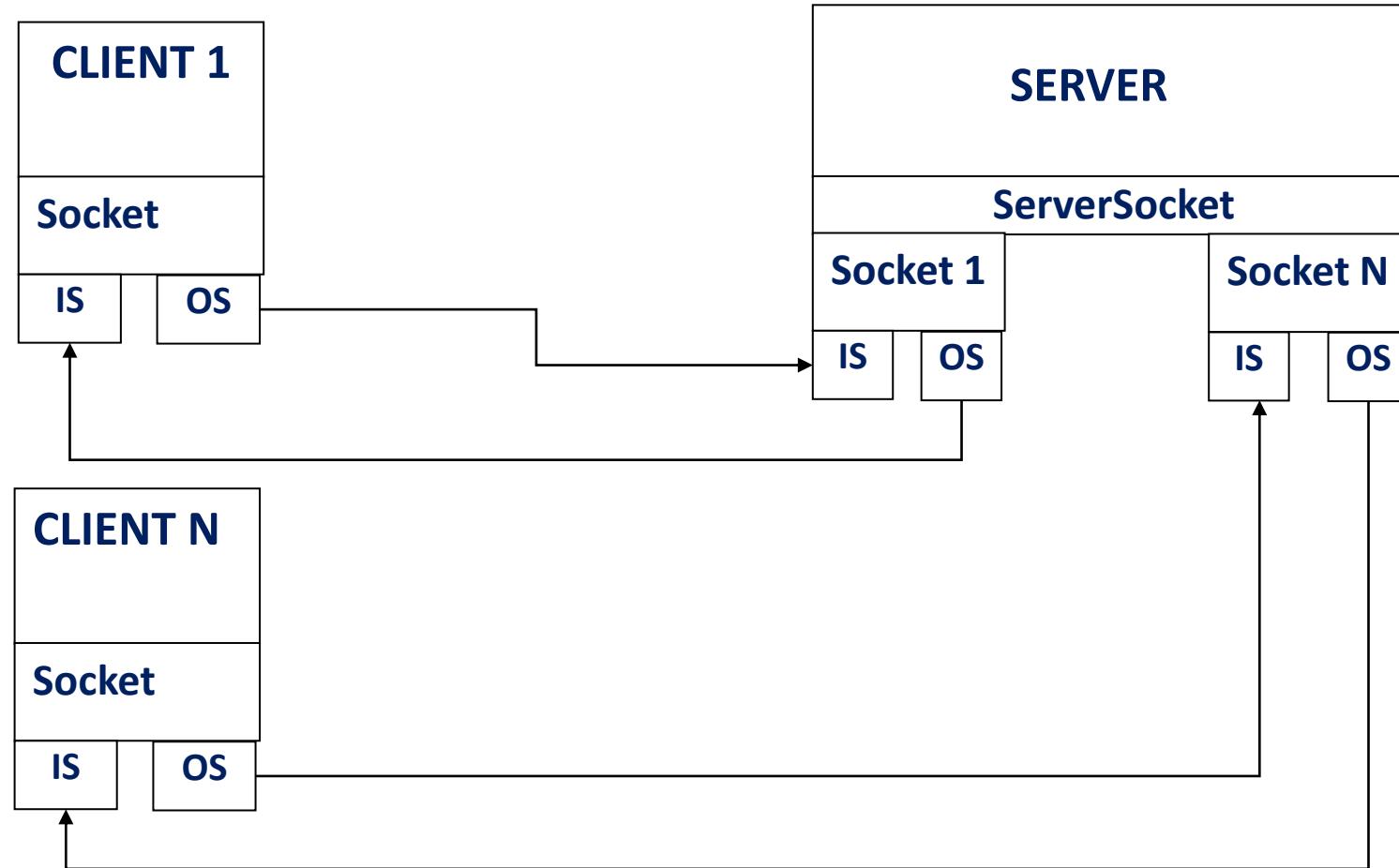
Conf. univ. dr. **Claudiu Vintă**

claudiu.vinte@ie.ase.ro

Comunicarea prin *TCP/IP*

- *Internet Protocol – IP*:
 - Permite transmisia de pachete de date (*datagram*) între calculatoare
 - Calculatoarele sunt identificate printr-o adresă (32 biți – IPv4 sau 128 biți – IPv6)
- *Transmission Control Protocol – TCP*:
 - Protocol bazat pe conexiuni care utilizează *IP* pentru transferul datelor
 - Poate fi utilizat simultan de mai multe procese identificate unic printr-un număr pe 16 biți denumit **port**
 - Asigură:
 - Retransmisia pachetelor pierdute
 - Reordonarea pachetelor primite în altă ordine decât ordinea în care au fost transmise
 - Detectia erorilor
 - Controlul vitezei de transfer
- O aplicație bazată pe TCP conține ușual:
 - O componentă server care răspunde la cererile inițiate de către clienți
 - Mai multe componente client care inițiază conexiunile

Conexiuni TCP - Arhitectura



Crearea unui server TCP/IP

- **Etapa 1:** Construirea unui obiect de tip ***ServerSocket*** și specificarea portului prin intermediul constructorului
 - Numărul portului poate fi orice port liber (uzual >1024) sau 0 pentru un port alocat automat.
- **Etapa 2:** Acceptarea unei conexiuni și obținerea obiectului ***Socket***
 - Se realizează prin apelul metodei ***accept***; aceasta blochează firul de execuție curent până la primirea unei conexiuni sau expirarea timpului de așteptare setat prin intermediul metodei ***setSoTimeout***
- **Etapa 3:** Obținerea obiectelor *stream* din *Socket* și utilizarea acestora pentru comunicare
 - Datele transmise de către client sunt disponibile prin intermediul unui obiect de tip ***InputStream*** obținut prin apelul metodei ***getInputStream***;
 - Datele sunt transmise prin scrierea în obiectul de tip ***OutputStream*** obținut prin metoda ***getOutputStream***
- Resursele trebuie eliberate prin utilizarea *try-with-resources* sau apelarea metodei ***Close*** în interiorul unei clause ***finally***
- Excepția ***IOException*** trebuie declarată sau tratată
- Adresa clientului (de tip ***InetAddress***) poate fi obținută prin apelul metodei ***getInetAddress***.

Crearea unui server TCP/IP

Exemplu:

```
final int PORT_NUMBER = 8193;
try (ServerSocket serverSocket = new ServerSocket(PORT_NUMBER)) {
    while (true) {
        try (Socket socket = serverSocket.accept();
            BufferedReader in = new BufferedReader(
                new InputStreamReader(socket.getInputStream()));
            PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
        ) {
            String mesajPrimit = in.readLine();
            out.printf("Mesajul primit este: %s%n", mesajPrimit);
        }
    }
}
```

Crearea unui client TCP/IP

- *Etapa 1:* Construirea unui obiect de tip **Socket** și specificarea adresei și portului aplicației server prin intermediul constructorului
 - Constructorul inițiază conexiunea cu aplicația server
- *Etapa 2:* Obținerea obiectelor *stream* din **Socket** și utilizarea acestora pentru comunicare
 - Datele sunt transmise folosind obiecte *InputStream* și *OutputStream* obținute prin apelul metodelor *getInputStream* și *getOutputStream*

Exemplu:

```
final int PORT_NUMBER = 8193;
try (Socket socket = new Socket("127.0.0.1", PORT_NUMBER);
    BufferedReader in = new BufferedReader(
        new InputStreamReader(socket.getInputStream()));
    PrintWriter out = new PrintWriter(socket.getOutputStream(), true);) {

    out.println("Test");
    System.out.println(in.readLine());
}
```

Server TCP/IP multi-client

Este pornit câte un fir de execuție pentru fiecare cerere:

```
public static void main(String[] args) throws IOException {
    final int PORT_NUMBER = 8193;
    try (ServerSocket serverSocket = new ServerSocket(PORT_NUMBER)) {
        while (true) {
            Socket socket = serverSocket.accept();
            new Thread(() -> procesareCerere(socket)).start();
        }
    }
}

private static void procesareCerere(Socket paramSocket) {
    try (Socket socket = paramSocket;
        BufferedReader in = new BufferedReader(
            new InputStreamReader(socket.getInputStream()));
        PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
    ) {
        // procesare cerere
    } catch (IOException exception) {
        exception.printStackTrace();
    }
}
```

Comunicarea prin UDP

- *User Datagram Protocol – UDP*
 - un protocol ce trimit pachete de date independente, numite *datagrame*, de la un calculator către altul fără a garanta în vreun fel ajungerea acestora la destinație, ordinea sau eliminarea mesajelor duplicate
 - nu stabilește o conexiune permanentă între cele două calculatoare
- O *datagramă* conține:
 - Portul sursă – identifică portul utilizat de către procesul care a transmis mesajul; utilizat de către aplicația server atunci când construiește obiectul răspuns;
 - Portul destinație – identifică portul utilizat de către procesul care primește mesajul; utilizat de către sistemul de operare pentru transmiterea mesajului către procesul corect
 - Lungimea datelor în octeți
 - Conținutul mesajului (un vector de octeți)

Crearea unui server UDP

Crearea unui server UDP presupune parcurgerea următoarelor etape:

1. Crearea unui obiect de tip **DatagramSocket** cu *specificarea portului* utilizat
2. Crearea unui vector de octeți și a unui obiect **DatagramPacket** pentru recepționarea mesajului
3. Recepționarea mesajului prin apelul metodei **receive**; apelul blochează firul de execuție până la primirea unui mesaj de la un client
4. Extragerea datelor, adresei și portului clientului din mesajul primit
5. Construirea obiectului **DatagramPacket** pentru răspuns pe baza adresei și portului specificate în cerere
6. Transmiterea răspunsului către aplicația client prin metoda **send**

Crearea unui server UDP

Exemplu

```
final int PORT_NUMBER = 1193;
final int MAX_MESSAGE_SIZE = 65535;

try (var socket = new DatagramSocket(PORT_NUMBER)) {
    while (true) {
        var buffer = new byte[MAX_MESSAGE_SIZE];
        var cerere = new DatagramPacket(buffer, buffer.length);
        socket.receive(cerere);

        // preluare date: cerere.getData()

        InetAddress adresaClient = cerere.getAddress();
        int portClient = cerere.getPort();
        // buffer = ...; // construire răspuns
        var raspuns = new DatagramPacket(buffer, buffer.length, adresaClient, portClient);
        socket.send(raspuns);
    }
}
```

Crearea unui client UDP

Crearea unui client UDP presupune parcurgerea următoarelor etape:

1. Crearea unui obiect de tip **DatagramSocket** fără port specificat
2. Crearea unui vector de octeți și a unui obiect **DatagramPacket** pentru mesajul cerere. Trebuie specificate adresa și portul serverului.
3. Transmiterea cererii către aplicația server prin metoda **send**
4. Recepționarea răspunsului prin apelul metodei **receive**; apelul blochează firul de execuție până la primirea răspunsului de la server.
5. Extragerea datelor din răspunsul primit de la server

Crearea unui client UDP

Exemplu

```
final String SERVER_HOST_NAME = "localhost";
final int SERVER_PORT_NUMBER = 1193;
final int MAX_MESSAGE_SIZE = 65535;

try (var socket = new DatagramSocket()) {
    // 1. Transmitere cerere UDP
    var buffer = "cerere".getBytes();
    var cerere = new DatagramPacket(
        buffer,
        buffer.length,
        InetAddress.getByName(SERVER_HOST_NAME),
        SERVER_PORT_NUMBER);
    socket.send(cerere);

    // 2. Receptionare raspuns
    buffer = new byte[MAX_MESSAGE_SIZE];
    var raspuns = new DatagramPacket(buffer, buffer.length);
    socket.receive(raspuns);
    // utilizare răspuns obținut din raspuns.getBytes()
}
```

Utilizare URL

- URL este acronimul pentru *Uniform Resource Locator* și reprezintă adresa unei resurse aflată pe Internet
- Un URL conține următoarele informații:
 - identificatorul de protocol (Ex. http)
 - identificator resursă (nume de host, port comunicare, cale resursă în cadrul serverului)
- Clasa care permite lucrul cu URL-uri este `java.net.URL`:
 - *public final class URL extends Object implements Serializable;*
- Constructori:
 - *URL(String spec)* - Creează un URL pornind de la reprezentarea String a acestuia
 - exemplu: `URL urlServer = new URL("http://www.ase.ro");`
 - *URL(String protocol, String host, int port, String file)* - Creează un obiect URL specificând distinct toate elementele componente
- Există o serie de metode care furnizează informațiile conținute într-un URL: *getProtocol, getHost, getPort, getPath, getFile, ...*

Utilizare URL – Citirea datelor

- Citirea conținutului unui URL se realizează prin intermediul unui obiect *InputStream* furnizat prin metoda *openStream()* a clasei URL

Exemplu:

```
try {
    URL url = new URL("https://www.ase.ro");
    try (BufferedReader in = new BufferedReader(new
        InputStreamReader(url.openStream()))) {
        in.lines().forEach(linie -> System.out.println(linie));
    }
} catch (Exception ex) {
    System.err.println(ex);
}
```

Utilizare URL – Transmiterea datelor

- Se utilizează metoda *openConnection()* a clasei *URL*, care furnizează un obiect ***URLConnection*** responsabil cu stabilirea unei conexiuni bidirectionale cu resursa specificată:
 - *public URLConnection openConnection() throws IOException;*
- Clasa abstractă *URLConnection* este superclasa tuturor claselor care reprezintă o legătură de comunicare între aplicație și o adresă URL (Exemplu: *HttpURLConnection*)
- Fluxurile de comunicare sunt furnizate de metodele *URLConnection*:
 - *public InputStream getInputStream() throws IOException;*
 - *public OutputStream getOutputStream() throws IOException;*
- O conexiune URL poate fi utilizată pentru citire și/sau scriere. Controlul scrierii și citirii se realizează prin două câmpuri de tip boolean, *doInput* și *doOutput*, care pot fi modificate prin metodele:
 - *public void setDoInput(boolean doinput);*
 - *public void setDoOutput(boolean dooutput);*

Server HTTP

- *Hypertext Transfer Protocol - HTTP* este un protocol de comunicatie responsabil cu transferul de hipertext (text structurat ce contine legaturi) dintre un client (de regulă un navigator web) și un server web
- Clasa care permite crearea unui server HTTP simplu este ***HttpServer***
- Crearea unui obiect ***HttpServer*** se realizează prin metoda statică ***create()*** a clasei ***HttpServer***:
 - *public static HttpServer create(InetSocketAddress addr, int backlog)* - sunt specificate adresa serverului și numărul maxim de conexiuni în așteptare
- Pornirea serverului se face prin invocarea metodei ***start()***

Server HTTP

- Presupune crearea unui obiect ***HttpContext***.
- Un obiect *HttpContext* reprezintă o mapare dintre o cale URI și o metodă care tratează cererile pe acest *HttpServer*.
 - *public abstract HttpContext createContext(String path, HttpHandler handler);*
- Metoda *handler* este furnizată printr-un obiect care implementează interfața *HttpHandler*.
- Interfața *HttpHandler* are o singură metodă abstractă prin care se implementează codul metodei *handler* care tratează cererile:
 - *void handle(HttpExchange exchange) throws IOException;*
- Clasa *HttpExchange* încapsulează o cerere HTTP și un răspuns care urmează să fie generat. Aceasta oferă metode pentru examinarea cererii de la client și pentru construirea și trimiterea răspunsului.
- Pentru construirea unui server simplu care să comunice cu aplicații client prin conexiunile *URLConnection* ale clientilor, vor fi utilizate metodele *HttpExchange* care furnizează fluxuri de acces la conținutul cererii și răspunsului
 - *public abstract InputStream getRequestBody();*
 - *public abstract OutputStream getResponseBody();*

Programare multiparadigmă - JAVA

Conf. univ. dr. **Claudiu Vintă**

claudiu.vinte@ie.ase.ro

Java Database Connectivity - JDBC

- Permite accesul la baze de date (citire și manipulare date) din aplicațiile Java
- Arhitectura JDBC este compusă din două **componente**:
 - **JDBC API** – independent de SGBD, utilizat de către aplicație pentru execuția operațiilor
 - **Driver Manager** – componenta care gestionează driverele pentru fiecare SGBD / tehnologie de acces
- Tipuri de drivere
 - *Tip 1 – ODBC Bridge*: transformă apelurile JDBC în apeluri către altă bibliotecă generică de acces la baze de date (exemplu ODBC)
 - *Tip 2 – Native API*: transformă apelurile JDBC în apeluri către bibliotecile native ale SGBD-ului (exemplu Oracle Cloud Infrastructure - OCI)
 - *Tip 3 - Middleware*: utilizează doar cod Java pentru conectare la un server *middleware* generic care comunică mai departe cu SGBD-ul dorit
 - *Tip 4 – Pure Java*: utilizează cod Java pentru implementarea protocolului de comunicație cu SGBD
- Clasele utilizate se regăsesc în pachetul *java.sql*.

Java Database Connectivity - JDBC

- Etapele de bază pentru accesarea unei baze de date:
 1. Utilizarea clasei ***DriverManager*** pentru obținerea unui obiect care implementează interfața ***Connection*** pe baza parametrilor de conectare
 2. Utilizarea metodelor obiectului *Connection* pentru construirea unui obiect care implementează interfața ***Statement*** (sau derivată) pe baza expresiei SQL și a parametrilor
 3. Executarea cererii utilizând metodele obiectului *Statement*
 4. În cazul expresiei SELECT, utilizarea obiectului ***ResultSet*** pentru parcurgerea datelor furnizate de către SGBD
 5. Eliberarea resurselor utilizate de obiectele Connection / Statement / ResultSet

JDBC – Deschiderea conexiunii

- Referințele la conexiuni se obțin prin invocarea metodei statice ***getConnection*** din clasa ***DriverManager***
- Metoda primește ca parametru un sir de caractere de forma *jdbc:subprotocol:subname* care depinde de SGBD-ul și driver-ul utilizat
- Exemple:
 - ODBC: *jdbc:odbc:denumire_DSN*
 - SQL Server:
jdbc:sqlserver://host\\server:1433;database=baza;user=sa;password=parola;
 - MySQL *jdbc:mysql://host:3306/baza*
 - Oracle: *jdbc:oracle:thin:@ip-host:1521:instanta*
 - SQLite: *jdbc:sqlite:cale\fisier.db*

JDBC – Deschiderea conexiunii

- Exemplu de cod pentru obținerea unei conexiuni la o bază de date SQLite denumită *studenti.db* care se află în subdirectorul *date*:

```
String url = "jdbc:sqlite:date\\studenti.db";
try (Connection connection = DriverManager
     .getConnection(url)) {

    // utilizare obiect connection
}
```

- Observații:
 - Este obligatorie **eliberarea resurselor** prin utilizarea unei instrucțiuni de tip ***try-with-resources*** – varianta recomandată – sau prin utilizarea metodei *close()* într-un bloc *finally*
 - Exceptia *checked SQLException* trebuie declarată la nivel de metodă sau tratată într-un bloc *catch*

JDBC – Construirea cererii

- Cererile către SGBD sunt gestionate prin obiecte care implementează interfața **Statement** sau una dintre interfețele derivate:
 - **PreparedStatement** – permite trimitera de expresii SQL parametrizate către SGBD
 - **CallableStatement** – permite apelarea procedurilor stocate (*stored procedures*)
- Construirea unui obiect cerere se realizează prin apelul metodei **createStatement()** (sau **prepareStatement(String sql)** sau **prepareCall(String sql)**) din obiectul conexiune.
- În cazul apelului **createStatement()** expresia SQL este furnizată la apelul metodei de execuție.
- Este obligatorie **eliberarea resurselor** prin utilizarea unei instrucțiuni de tip **try-with-resources** – varianta recomandată – sau prin utilizarea metodei **close()** într-un bloc **finally**

JDBC – Cereri simple

- Pentru expresii SQL care îndeplinesc simultan următoarele condiții:
 - Sunt fixe (nu sunt generate dinamic pe bază de date preluate din alte surse)
 - Nu întorc un rezultat (orice instrucțiune SQL cu excepția SELECT)
- În această situație se utilizează obiecte de tip **Statement** și metoda **executeUpdate**
 - Valoarea returnată reprezintă numărul de înregistrări procesate

```
String url = "jdbc:sqlite:date\\studenti.db";
try (Connection connection = DriverManager.getConnection(url);
     Statement statement = connection.createStatement()) {

    int numarRanduri = statement.executeUpdate("UPDATE Studenti SET Cod = Cod + 100");
    System.out.printf("Au fost modificate %d înregistrări.%n", numarRanduri);
}
```

JDBC – Cereri cu parametri

- Sunt utilizate atunci când cererea SQL trebuie să includă date care nu sunt constante
- Se utilizează obiecte de tip ***PreparedStatement***:
 - Expresia SQL este furnizată la construirea obiectului (metoda *prepareStatement*)
 - În interiorul frazei SQL se marchează folosind caracterul “? “ locurile unde trebuie inserate valorile
 - Înainte de execuție se furnizează valorile pentru fiecare parametru prin intermediu metodelor *setTIP(index, valoare)*

```
var studenti = List.of(new Student(1, "Popescu Maria"), new Student(2, "Marinescu Ion"));

try (Connection connection = DriverManager.getConnection(url);
     PreparedStatement statement = connection.prepareStatement(
         "INSERT INTO Studenti(Cod, Nume) VALUES (?, ?)");) {
    for (var student : studenti){
        statement.setInt(1, student.getCod());
        statement.setString(2, student.getNume());
        statement.executeUpdate();
    }
}
```

JDBC – Citirea datelor

- Se realizează prin intermediul obiectelor de tip ***ResultSet*** obținute în urma execuției cererilor prin metoda *executeQuery* (valabil atât pentru *Statement*, cât și pentru *PreparedStatement*)
- Un obiect ***ResultSet*** încapsulează un pointer către înregistrarea curentă din setul de date generat de expresia SQL de tip SELECT. Poziția inițială este *înainte* de prima înregistrare.
- Principalele metode disponibile:
 - *bool next()*: avansează la înregistrarea următoare și întoarce *true* dacă operația s-a executat cu succes
 - metode de forma *TIP getTIP(int index / string numeColoană)* care citesc și convertesc valoarea din coloana specificată a rândului curent
- Se recomandă **eliberarea resurselor** prin utilizarea unei instrucțiuni de tip ***try-with-resources*** – varianta recomandată – sau prin utilizarea metodei *close()* într-un bloc *finally*

JDBC – Citirea datelor

- Execuția unei comenzi SELECT și parcurgerea datelor:

```
// 1. Declarație listă studenți
List<Student> studenți = new ArrayList<>();
// 2. Citire listă studenți din baza de date
try (Connection connection = DriverManager.getConnection(url);
     PreparedStatement statement = connection.prepareStatement(
         "SELECT Cod, Nume FROM Studenți WHERE Nume LIKE ? || '%'"));
{
    statement.setString(1, "Pop"); // setare valoare parametru

    try (ResultSet date = statement.executeQuery()) {
        // trebuie avansat cursorul până la prima înregistrare
        while (date.next()){
            studenți.add(new Student(date.getInt(1), date.getString(2)));
            // sau: date.getInt("Cod"), date.getString("Nume")
        }
    }
    // 3. Afisare lista citită
    studenți.stream().forEach(student -> System.out.println(student));
}
```

JDBC – Gestiunea tranzacțiilor

- Tranzacția este o unitate logică de prelucrare a datelor unei baze de date cu următoarele caracteristici:
 - *Atomicitate*: modificările executate în cadrul unei tranzacții sunt indivizibile (se execută toate sau niciuna)
 - *Consistență*: o tranzacție transformă baza de date dintr-o stare consistentă în altă stare consistentă
 - *Izolare*: un set de tranzacții executate concurent sau secvențial lasă baza de date în aceeași stare
 - *Durabilitate*: dacă o tranzacție s-a încheiat cu succes atunci datele trebuie să rămână salvate chiar și în cazul apariției unei erori de sistem
- Tipuri de anomalii (ANSI/ISO SQL):
 - *dirty read*: o tranzacție poate citi datele intermediare generate de altă tranzacție
 - *non-repeatable read*: valorile citite dintr-o înregistrare sunt diferite pentru două citiri în cadrul aceleiași tranzacții
 - *phantom-read*: similar cu non-repeatable read, dar pentru un set de înregistrări; setul de înregistrări obținut la două momente de timp în cadrul unei tranzacții diferă deși este utilizat același criteriu de selecție

JDBC – Gestiuinea tranzacțiilor

- Nivelurile de izolare ale tranzacțiilor sunt definite sub formă de constante în interfața *Connection*:

Constanta	Suportă tranzacții	Dirty Read	Non-repetable Read	Phantom Read
<code>TRANSACTION_NONE</code>	NU	-	-	-
<code>TRANSACTION_READ_UNCOMMITTED</code>	DA	DA	DA	DA
<code>TRANSACTION_READ_COMMITTED</code>	DA	NU	DA	DA
<code>TRANSACTION_REPEATABLE_READ</code>	DA	NU	NU	DA
<code>TRANSACTION_SERIALIZABLE</code>	DA	NU	NU	NU

- Metodele interfeței *Connection* utilizate pentru controlul nivelului de izolare:
 - `void setTransactionIsolation(int level) throws SQLException`
 - `int getTransactionIsolation() throws SQLException`

JDBC – Gestiuinea tranzacțiilor

- JDBC suportă două moduri de gestiune a tranzacțiilor:
 - Auto-commit (default): fiecare apel de metodă *execute* este considerat o tranzacție (se aplică automat COMMIT după fiecare apel)
 - Explicit: toate apelurile execute sunt considerate ca făcând parte din aceeași tranzacție până la întâlnirea unui apel *commit()* sau *rollback()*
- Metodele interfeței Connection utilizate pentru controlul tranzacțiilor:
 - *void setAutoCommit(boolean autoCommit)*: setează modul de gestiune (*true* pentru auto-commit sau *false* pentru control explicit)
 - *boolean getAutoCommit()*: întoarce modul curent de gestiune
 - *void commit()*: salvează modificările efectuate de tranzacție în baza de date și încheie tranzacția curentă (doar pentru modul explicit)
 - *void rollback()* : anulează modificările efectuate de tranzacție și încheie tranzacția curentă (doar pentru modul explicit)

JDBC – Gestiunea tranzacțiilor

- Exemplu de utilizare tranzacție explicită:

```
try (Connection connection = DriverManager.getConnection(url)) {  
    connection.setAutoCommit(false);  
    try {  
        // Operația 1: Ștergem datele existente  
        try (var deleteStatement = connection.createStatement()) {  
            deleteStatement.executeUpdate("DELETE FROM Studenti");  
        }  
  
        // Operația 2: Adăugăm înregistrările din listă  
        try (var insertStatement =  
                connection.prepareStatement("INSERT INTO Studenti(Cod, Nume) VALUES (?, ?)")) {  
            for (var student : studenti) {  
                insertStatement.setInt(1, student.getCod());  
                insertStatement.setString(2, student.getNume());  
                insertStatement.executeUpdate();  
            }  
        }  
        connection.commit();  
    } catch (Exception e) { connection.rollback(); throw e; }  
}
```

JDBC - Metadate

- **Interfața *Connection*** – furnizează informații despre baza de date (structura, tabele, catalog, ...):
 - ***DatabaseMetaData getMetaData()*** – informații despre baza de date curentă
 - *ResultSet getTables(String catalog, String schemaPattern, String tableNamePattern, String[] types)* – lista de tabele filtrate în funcție de criteriile furnizate
 - *ResultSet getColumns(String catalog, String schemaPattern, String tableNamePattern, String columnNamePattern)* – lista de coloane filtrate în funcție de criteriile furnizate
 - *ResultSet getPrimaryKeys(String catalog, String schema, String table)* – lista de coloane care compun cheia primară în tabela specificată

JDBC - Metadate

- **Interfața *ResultSet*** – furnizează informații despre rezultatul interogării (coloane, tipuri, ...):
 - ***ResultSetMetaData getMetaData()*** – informații despre interogarea curentă
 - *int getColumnCount()* – numărul de coloane
 - *String getColumnName(int column)* – denumirea coloanei
 - *int getColumnType(int column)* – tipul coloanei
 - *String getColumnTypeName(int column)* – denumirea tipului coloanei
 - *int getColumnDisplaySize(int column)* – dimensiunea coloanei în caractere