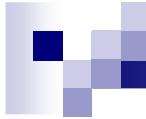


# 1. Elementele de bază ale limbajului Java

Sl. dr. ing. Raul Robu

2022-2023, Semestrul 2



# CUPRINS

- 1.1. Istoricul și caracteristicile limbajului Java
- 1.2. JDK – Java Development Kit
- 1.3. Pachetele Java
- 1.4. Principalele medii integrate de dezvoltare pentru Java
- 1.5 Structura unui program Java
- 1.6 Cuvintele rezervate ale limbajului Java
- 1.7 Tipuri de variabile în Java
- 1.8 Tipul variabilelor în funcție de locul declarării
- 1.9 Clase înfășurătoare
- 1.10 Spațiul ocupat și domeniul de valori al variabilelor primitive
- 1.11 Constante
- 1.12 Tablourile în Java
- 1.13 Sirurile de caractere în Java (clasele *String*, *StringBuffer*, *StringBuilder*)
- 1.14 Date Time API
- 1.15 Operatorii limbajului Java
- 1.16 Instrucțiunile limbajului Java



# 1.1. Istoricul și caracteristicile limbajului Java

## Istoric

- Limbajul Java a fost generat în cadrul proiectului Green al companiei Sun Microsystems, proiect lansat în 1991 și condus de James Gosling
- Prima implementare JDK (Java Development Kit) a fost lansată în 1995 (versiunile alpha și beta) și urmărea principiul "*Write Once, Run Anywhere*". JDK 1.0 a fost lansată în 1996
- Sun Microsystems a fost preluată de Oracle Corporation în 2009
- Platforme Java:
  - Java SE (Java Standard Edition)
  - Java EE (Java Enterprise Edition)
  - Java ME (Java Micro Edition)
- Versiunile Java Standard Edition cu suport pe termen lung (LTS – Long term support) sunt:
  - Java SE 7 (iulie 2011)
  - Java SE 8 (martie, 2014)
  - Java SE 11 (septembrie, 2018)
  - Java SE 17 (septembrie, 2021)

- Începând cu Java SE 9 (Septembrie, 2017) la fiecare 6 luni apare o nouă versiune (în lunile martie și septembrie a fiecărui an)
- Ultima versiune la data curentă este Java SE 19 (Septembrie, 2022), urmează să apară Java SE 20 în martie 2023

## Caracteristici

- Traducere în executabil în 2 faze:
  - Compilarea: text Java -> cod de mașină virtuală Java ("byte code" sau cod intermediu)
  - Interpretarea: cod de mașină virtuală Java -> cod de mașină de execuție efectivă, fizică
- Are un mare grad de independentă de platformă
- Este orientat pe obiecte
- Prezintă câteva simplificări în raport cu C++:
  - nu admite moștenirea multiplă,
  - nu operează cu pointeri, ci cu referințe, asupra cărora se pot efectua doar operații de atribuire, nu și operații aritmetice,
  - Efectuează automat dezalocarea memoriei ocupate de obiecte care nu mai sunt folosite cu ajutorul unei componente numite "Garbage collector"

- Este concurrent
- Este distribuit - Programele Java pot îngloba, alături de obiecte locale, obiecte aflate la distanță, în mod normal în rețea, inclusiv în Internet. În Java, sunt respectate protocoalele de rețea FTP, HTTP, etc
- Este dinamic și robust. În Java, alocarea memoriei este prin excelență dinamică –adică: nu se face la compilare / linkeditare, ci în execuție- și orice alocare este precedată de verificări.
- Este sigur. Înainte ca interpreterul Java să execute codul intermediar, acesta este supus unor verificări de tipul: nedepășirea stivei, etc.

## 1.2. JDK – Java Development Kit

*Kit-ul pentru dezvoltarea aplicațiilor Java conține:*

- compilatorul Java: *javac*
- interpreterul de cod intermediar: *java* (este specific pentru fiecare mașină țintă)
- un depanator: *jdb*
- un generator de documentație: *javadoc*

- un generator de fișiere *header* pentru integrarea unor funcții scrise în C: *javadoc*
- un dezasamblor pentru fișiere .class: *javap*
- un utilitar pentru generarea de arhive: *jar*
- *Java Runtime Environment (JRE)* – conține mașina virtuală Java, biblioteci și alte componente necesare rulării aplicațiilor java
- *JShell: The Java Shell* – JDK 9 introduce *JShell* care este o unealtă de tipul *REPL(Read Evaluate Print Loop)* și rulează din linie de comandă. *JShell* nu se folosește pentru a crea proiecte ci pentru a verifica rapid dacă logica unei bucăți de cod este bună, sau dacă ieșirea unei bucăți de cod este corectă.
- *etc*

## 1.3. Pachete Java

- *java.lang* - conține clasele de bază (clasa *Object*, clasele înfășurătoare, clase pentru tratarea exceptiilor, clase pentru lucru cu fire de execuție, etc). Acest pachet este importat în mod implicit de către compilator
- *java.util* – conține clasele necesare pentru lucrul cu colecții de obiecte și alte clase cu rol utilitar
- *java.io* - conține clase pentru intrări / ieșiri generice și accesarea fișierelor
- *java.math* - conține clase pentru operații matematice
- *java.time* – a fost introdus în Java 8 și conține clase suport pentru lucrul cu date calendaristice, timp, etc.
- *java.sql* - conține clase suport pentru lucrul cu baze de date
- *javax.servlet* – conține clasele suport pentru lucrul cu servleturi
- *javax.xml* – conține clasele suport pentru procesarea documentelor XML
- *java.security* - conține clase suport pentru autentificare, criptare, etc.
- *javax.swing* - conține clase suport pentru dezvoltarea interfețelor grafice
- §.a

## 1.4. Principalele medii integrate de dezvoltare pentru Java

- *IntelliJ* (dezvoltat de JetBrains)
- *Eclipse* (proiectat inițial de IBM, dar aflat acum în stadiu de *open source*)
- *NetBeans* (inițial dezvoltat de Charles University in Prague)

**La dezvoltarea aplicațiilor practice vom folosi:**



**IntelliJ IDEA Community Edition 2022.3.2**



**Eclipse 2022-09 IDE for enterprise Java and Web Developers**

## 1.5 Structura unui program Java

- În general programele Java conțin mai multe clase și interfețe în care se scrie codul *java*
- Clasele și interfețele, se plasează în pachete, care nu sunt altceva decât directoare. Dacă în denumirea pachetului se folosește punct se creează subdirectoare, a căror denumire este delimitată de acest simbol
- În pachete se amplasează clase care deservesc același scop. Dacă ne gândim la clasele din *API-ul Java*, avem pachetul *java.io* care conține clase și interfețe cu ajutorul cărora se fac operații de intrare / ieșire. Pe de altă parte pachetul *javax.swing* conține clase cu ajutorul cărora se realizează aplicații cu interfață grafică
- Pachetele se vor găsi în directorul *src* al proiectului având în ele fișiere cu extensia *java* ce conțin codul sursă (în cazul proiectelor dezvoltate în *IntelliJ* și *Eclipse*) și în directorul *bin* (în cazul proiectelor dezvoltate în *Eclipse*) respectiv în directorul *out/production* (în cazul *IntelliJ*) și vor conține fișiere cu extensia *class*. Aceste fișiere conțin byte code-ul (codul de mașină virtuală rezultat prin compilare)
- În general programatorul trebuie să creeze o clasă care conține funcția ***public static void main(String[] args)*** cu care își începe execuția programul. Excepție de la aceasta regulă o reprezintă proiectele web dinamice cu servlet-uri și JSP-uri și aplicațiile Java mobile realizate în *Android Studio*, în care programatorul nu trebuie să creeze o astfel de clasă.

- Funcția *main*, cu care își începe execuția un program, are o structură fixă, doar numele vectorului de *string*-uri care este parametru de intrare poate să fie modificat
- Un prim program în Java este exemplificat mai jos

```
package capitolul1.hello;

import java.util.Scanner;

public class MainApp {
    public static void main(String[] args) {
        System.out.print("a=");
        Scanner scanner=new Scanner(System.in);
        int a=scanner.nextInt();
        System.out.println("Ati introdus valoarea="+a);
        scanner.close();
    }
}
```

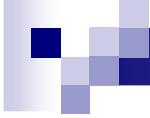
The screenshot shows a Java application window with the following details:

- Toolbar tabs: @ Javadoc, Declaration, Console, Progress.
- Status bar: <terminated> MainApp (41) [Java Application] D:\kituri\
- Console output:
  - a=3
  - Ati introdus valoarea=3

- Programul începe cu cuvântul cheie *package* care specifică numele pachetului (directorului care conține acest cod sursă). Într-un proiect se poate să nu se creeze nici un pachet, în acest caz cuvântul *package* va lipsi.
- Urmează o zonă în care se importă clasele, interfețele utilizate în program, în cazul de față clasa *Scanner* din pachetul *java.util*. Se poate importa tot conținutul unui pachet printr-o sintaxă precum cea de mai jos, dar întotdeauna se recomandă să se importe doar ceea ce se utilizează

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

- Apoi urmează clasa care conține funcția *public static void main(String[] args)*.
- *System.out* este *streamul* de ieșire standard și este asociat monitorului. Apelul metodei *print* sau *println* prin intermediul acestui *stream* va determina scriere pe ecran. Dacă se utilizează un obiect de tip *PrintStream* asociat unui fișier și se apelează metoda *print* pentru acel obiect se va face scriere în fișier
- *System.in* reprezintă *streamul* de intrare standard asociat tastaturii
- Clasa *Scanner* dispune de metode care permit citirea directă a tipurilor primitive.
- La instanțierea obiectului *Scanner* prin constructor i s-a transmis *stream-ul* asociat tastaturii, astădată metoda va citi de la tastatură. Dacă s-ar fi transmis *stream-ul* asociat unui fișier citirea s-ar fi realizat din fișier
- În exemplul precedent operatorul *+* este folosit ca și operator de concatenare a *string-urilor*



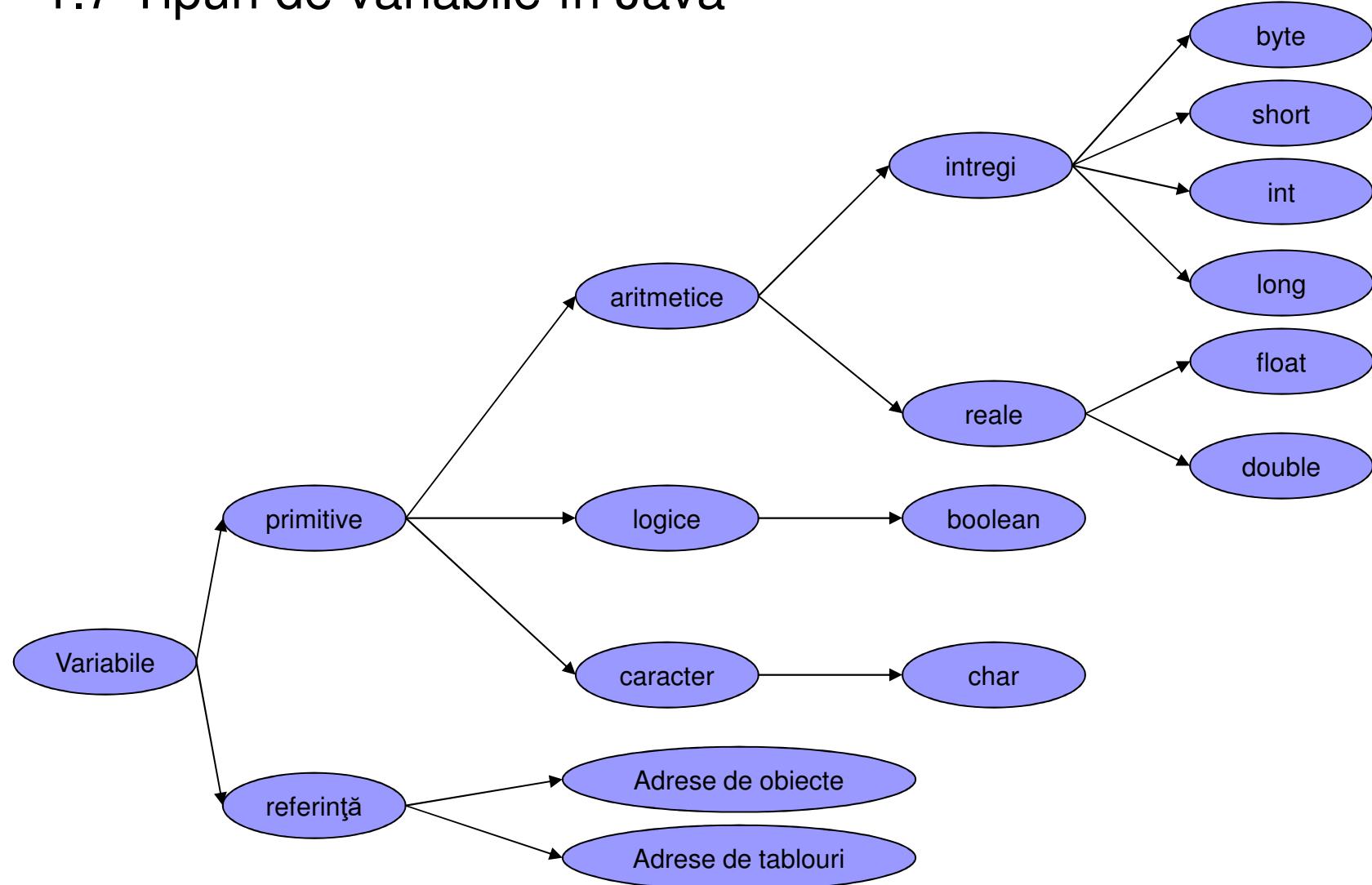
## ■ Recomandări de urmat privind alegerea denumirilor:

- denumirile pachetelor sunt formate din litere mici (ex: *java.util*, *java.sql*, etc)
- denumirile de clase și interfețe, în general denumirile tipurilor de date care nu sunt primitive, încep cu literă mare. Dacă aceste denumiri sunt formate din alipirea mai multe cuvinte, atunci prima litera a fiecărui cuvânt va fi mare (ex. *BufferedReader*, *InputStreamReader*, etc).
- Denumirile de variabile primitive și obiecte încep cu literă mică
- Denumirile de constante se aleg cu toate litere mari și separate prin *underscore* atunci când sunt formate din mai multe cuvinte
- Nu sunt premise spații în denumiri

## 1.6 Cuvintele rezervate ale limbajului Java

abstract	do	import	protected	throws
boolean	double	instanceof	public	transient
break	else	int	return	try
byte	extends	interface	short	void
case	final	long	static	volatile
catch	finally	native	super	while
char	float	new	switch	
class	for	null	synchronized	
continue	if	package	this	
default	implements	private	throw	

## 1.7 Tipuri de variabile în Java





## 1.8 Tipul variabilelor în funcție de locul declarării

- *variabile locale* – declarate în funcții, accesibile doar în interiorul acestora, distruse după ce funcția își încheie execuția
- *variabile de intrare ale metodelor*, sau *parametri formali*, declarați între parantezele rotunde ale metodelor
- *variabile instanță - variabile membre* ale clasei (atributele sau proprietățile clasei) declarate în clasă, în afara metodelor
- *variabile clasă* – variabile declarate în afara metodelor și statice (se vor discuta în detaliu în capitolul următor)

## 1.9 Clasele Înfășurătoare în Java

- Clasele înfășurătoare (*wrapper classes*) sunt clase asociate tipurilor primitive.
- Clasele înfășurătoare sunt: *Byte, Short, Integer, Long, Float, Double, Boolean, Character*
- Obiectul clasei înfășurătoare conține o valoare a tipului primitiv asociat și dispune de metode care permit diverse prelucrări asupra valorii primitive asociate
- În continuare se exemplifică pentru clasa înfășurătoare *Integer*:
- Instantierea:

```
Integer x=new Integer(3);
Integer y=new Integer(3);

//SAU

Integer x=3; //autoboxing
int z=3;
Integer y=z; //autoboxing
int w=x; //unboxing
```

- Conversia automată pe care compilatorul Java o face între tipurile primitive și obiectele corespunzătoare claselor lor înfășurătoare se numește *autoboxing*
- Conversia unui obiect al clasei înfășurătoare la tipul primitiv asociat se numește *unboxing*
- În continuare se exemplifică o parte din metodele care pot fi utilizate în operații cu obiecte *Integer*

```
System.out.println( x.compareTo(y) ); // afiseaza 0. Returneaza 1 cand x>y si -1 cand x<y  
System.out.println( x.equals(y) ); // afiseaza true
```

```
System.out.println( Integer.toString(15) ); // afiseaza 15. Conversie din intreg in String
```

```
System.out.println( Integer.toBinaryString(15) ); // afiseaza 1111. returneaza un String cu  
// reprezentarea binara a intregului transmis ca si parametru
```

```
Integer intVar2;  
intVar2=Integer.valueOf("123"); //conversie din String in Integer
```

```
int intVar3;  
intVar3=Integer.parseInt("123"); //conversie din String in int
```

## 1.10 Spațiul ocupat și domeniul de valori al variabilelor primitive

- Informațiile privind spațiul de memorie ocupat de către varaiabilele primitive precum și valorile minime și maxime ale acestora se pot afla cu ajutorul constantelor *SIZE*, *MIN\_VALUE* și *MAX\_VALUE* din clasele înfășurătoare corespunzătoare

```
System.out.printf("%-6s %-10s %-22s %s\n","Tip","Numar biti","Valoarea minima","Valoarea maxima");
System.out.printf("%-6s %-10s %-22s %s\n",Byte.TYPE, Byte.SIZE, Byte.MIN_VALUE, Byte.MAX_VALUE);
System.out.printf("%-6s %-10s %-22s %s\n",Short.TYPE, Short.SIZE, Short.MIN_VALUE, Short.MAX_VALUE);
System.out.printf("%-6s %-10s %-22s %s\n",Integer.TYPE, Integer.SIZE, Integer.MIN_VALUE, Integer.MAX_VALUE);
System.out.printf("%-6s %-10s %-22s %s\n",Long.TYPE, Long.SIZE, Long.MIN_VALUE, Long.MAX_VALUE);

System.out.printf("%-6s %-10s %-22s %s\n",Float.TYPE, Float.SIZE, Float.MIN_VALUE, Float.MAX_VALUE);
System.out.printf("%-6s %-10s %-22s %s\n",Double.TYPE, Double.SIZE, Double.MIN_VALUE, Double.MAX_VALUE);

System.out.printf("%-6s %-10s %-22s %s\n",Character.TYPE, Character.SIZE, (int) Character.MIN_VALUE, (int) Character.MAX_VALUE);
```

Tip	Numar biti	Valoarea minima	Valoarea maxima
byte	8	-128	127
short	16	-32768	32767
int	32	-2147483648	2147483647
long	64	-9223372036854775808	9223372036854775807
float	32	1.4E-45	3.4028235E38
double	64	4.9E-324	1.7976931348623157E308
char	16	0	65535

- Caracterul se reprezintă pe 2 octeți

## 1.11 Constante în limbajul Java

- Așa cum numele sugerează, o constantă este o entitate dintr-un program care este imutabilă, cu alte cuvinte care are o valoare ce nu poate fi schimbată
- Se declară cu ajutorul cuvântului cheie **final** amplasat înaintea tipului

```
final int a=3;  
//a=4; //eroare de compilare
```

- Constantele pot fi:
  - numerice
    - întregi
    - reale
  - booleene
  - caracter
  - sir
- Constantele întregi pot fi:
  - Octale – precedate de cifra 0, formate din cifre de la 0 la 7
  - Zecimale – prima cifra diferită de 0, iar restul cifre de la 0 la 9
  - Hexazecimale – precedate de grupul de caractere *0x*, formate din cifre de la 0 la 9 și/sau litere de la ‘a’ la ‘f’
  - Dacă înșiruirile respective nu sunt urmate de niciun sufix, atunci constantele în cauză se reprezintă intern pe 4 octeți. Dacă înșiruirile respective sunt urmate de sufixul “L”, atunci constantele în cauză se reprezintă intern pe 8 octeți. În ambele cazuri, reprezentarea internă este “complement de doi”.

## ■ Exemple de valori constante întregi

023	notație octală, reprezentând constanta 19, exprimată intern pe 4 octeți
23	notație zecimală, reprezentând constanta 23, exprimată intern pe 4 octeți
0x15	notație hexazecimală, reprezentând constanta 21, exprimată intern pe 4 octeți
0xffffffff	notație hexazecimală, reprezentând constanta -1, exprimată intern pe 4 octeți
023L	notație octală, reprezentând constanta 19, exprimată intern pe 8 octeți
23L	notație zecimală, reprezentând constanta 23, exprimată intern pe 8 octeți
0x15L	notație hexazecimală, reprezentând constanta 21, exprimată intern pe 8 octeți
0xffffffffL	notație hexazecimală, reprezentând constanta -1, exprimată intern pe 8 octeți

## ■ Exemple de valori constante reale

$$\left\{ \begin{array}{l} [+] \\ [-] \end{array} \right\} \left[ \text{partea intreaga} \right] \left[ \begin{array}{l} \cdot \text{partea fractionara} \\ [.] \end{array} \right] \left\{ \begin{array}{l} e[+] \text{modulul exponentului} \\ e - \text{modulul exponentului} \end{array} \right\} \left\{ \begin{array}{l} f \\ [d] \end{array} \right\}$$

1.4f – dacă se pune sufixul f variabila de reprezintă pe 4 octeți

1. - dacă se pune sufixul d sau nu se pune nici un sufix variabila se reprezintă pe 8 octeți, deci trebuie declarată de tip double

.723

1.2e15f

1.e-7

-27e+12

-43e-6d

```
final float f=2.3;          //GRESIT! Eroare de compilare, pentru ca se incercă atribuirea unei valori  
// double unei variabile float  
  
final float f=2.3f;          //CORECT  
final float f=(float)2.3;    //CORECT
```

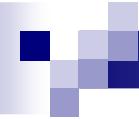
## ■ Valori constante caracter

- Se pun intre apostrofi
- System.out.println('A');
- System.out.println('\u0041'); //severta escape
- System.out.println('\t');//severta escape speciala caraterul t reprezintă tab, alte caractere posibile sunt:
  - b - pentru backspace
  - n - pentru line feed
  - f - pentru form feed
  - r - pentru carriage return
  - " - pentru double quote
  - ' - pentru single quote
  - \ - pentru backslash

## ■ Valori constantele sir

- Se pun intre ghilimele
- Exemplu:
  - System.out.println("Automatica si Calculatoare");
  - System.out.println("Vechiul nume: Universitatea \"Politehnica\" din Timisoara");

## ■ Valori constante booleene {true,false}



## 1.12 Tablourile în Java

- Crearea și inițializarea tablourilor

- `int[] alfa=new int[5];`
  - `alfa[0]=100;`
  - `alfa[1]=200;`
  - `....`
  - `alfa[4]=500;`

*SAU*

- `int[] alfa={100,200,300,400,500};`

- În Java se verifică depășirea capacitatei tablourilor. Încercarea de a accesa un element din afara spațiului alocat va genera excepția *ArrayIndexOutOfBoundsException*
- Lungimea unui tablou → `alfa.length`

## ■ Copierea tablourilor

- Copierea tablourilor se poate realiza cu ajutorul metodei statice *arraycopy()* din clasa *System* sau cu ajutorul metodei statice *copyOf()* din clasa *Arrays*

```
public static void arraycopy (Object referințăTablouSursă,
                            int indiceÎnSursă,
                            Object referințăTablouDestinație,
                            int indiceÎnDestinație,
                            int numărElementeDeCopiat)
```

```
package capitolul1.tablouri1;

public class MainApp {
    public static void main(String[] args) {
        int []a= {1,2,3};
        int []b= {4,5,6};

        b=a; //b indica spre aceeasi referinta ca si a. Nu se face copiere de elemente
        a[0]=99;
        System.out.println("a[0]="+a[0]+ " b[0]="+b[0]); //Output a[0]=99 b[0]=99
```

```

int []c= {1,2,3};
int []d=new int[c.length];
System.arraycopy(c, 0, d, 0, c.length);
//valorile din zona de memorie a lui c se copiaza in zona de memorie a lui d
c[0]=99;
System.out.println("c[0]="+c[0]+" d[0]="+d[0]);
}
}

```

`a[0]=99 b[0]=99  
c[0]=99 d[0]=1`

## ■ Ordonarea tablourilor

- Ordonarea tablourilor de tipuri primitive se poate realiza cu ajutorul metodei statice sort() din clasa Arrays, care utilizează pentru ordonare algoritmul Quicksort
- *Metoda poate fi utilizată și pentru ordonarea vectorilor de obiecte în ordine crescătoare naturală. Dacă obiectele nu au o ordine crescătoare naturală trebuie implementat un comparator*

```

package capitolul1.tablouri2;

import java.util.Arrays;

public class MainApp {
    public static void afis(int []v) {
        for(int i=0;i<v.length;i++)
            System.out.print(v[i]+" ");
        System.out.println();
    }
    public static void main(String[] args) {
        int []a= {5,2,1,7,9,3,0};
        afis(a);
        Arrays.sort(a);
        afis(a);

        int poz=Arrays.binarySearch(a, 5); //doar ptr tablou ordonat
        System.out.println(poz>=0?"Gasit pe pozitia "+poz:"Nu se gaseste!");
        int []b=Arrays.copyOf(a, a.length);
        afis(b);
    }
}

```

```

5 2 1 7 9 3 0
0 1 2 3 5 7 9
Gasit pe pozitia 4
0 1 2 3 5 7 9

```

## ■ Tablouri bidimensionale

- Tablourile de tablouri (tablouri bidimensionale) pot avea un număr diferit de elemente pe fiecare rând

```
String[][] persoane = { { "Dl. ", "Dna. ", "Dsra. " }, { "Popescu", "Georgescu" } };

for (int i = 0; i < persoane.length; i++) { //accesam randurile
    for (int j = 0; j < persoane[i].length; j++){ //accesam elementele de pe randuri
        System.out.print(persoane[i][j] + " ");
    }
    System.out.println();
}
```

---

Dl. Dna. Dsra.  
Popescu Georgescu

## 1.13 Sirurile de caractere în Java

- Java oferă mai multe clase pentru lucru cu siruri de caractere
- **Clasa String** – este cea mai utilizată clasă pentru lucruri cu siruri de caractere. Sirurile de caractere declarate ca și *String*-uri nu pot fi modificate. *Stringul* este imutabil. Toate metodele din clasa *String* care operează asupra sirului de caractere (cum ar fi *trim()*, *toUpperCase()*, *toLowerCase()*, *replace()*, etc) returnează un *String* nou cu valorile modificate, lăsându-l pe cel vechi nemodificat.
- **Clasa StringBuilder** – dispune de metode care permit modificarea sirului de caractere, dar are metode nesincronizate (nu sunt *thread safe*, utilizarea acestora nefiind recomandată în aplicații cu fire de execuție)
- **Clasa StringBuffer** – dispune de metode care permit modificarea sirului de caractere, are metode sincronizate (este *thread safe*, poate să fie utilizata în aplicații cu fire de execuție)

# Clasa *String*

## ■ Declararea și instantierea

```
String sir;    //sau: String sir="hello!";
sir="hello!";

//sau:
String sir=new String("hello!");

//sau:
char[] helloArray={'h','e','l','l','o','!'};
String sir=new String(helloArray);
System.out.println(sir); // afișează: "hello!"
```

## ■ Compararea sirurilor

```
String s1 = "abc";
String s2 = "abc";
System.out.println(s1==s2);
System.out.println(s1.equals(s2));

String s3 = new String("abc");
String s4 = new String("abc");
System.out.println(s3==s4);
System.out.println(s3.equals(s4));
```

```
@ Javadoc D
<terminated> M
true
true
false
true
```

- În cazul stringurilor care sunt instantiatе **fără** a utiliza operatorul *new* (prin atribuirea unei valori), se parcurge containerul de *string*-uri al mașinii virtuale și dacă se găsește un *string* cu aceeași valoare se returnează o referință către acesta, iar dacă nu se găsește atunci se introduce *stringul* în container și apoi se returnează o referință către el
- Astfel când *s2* primește o valoare, după ce *s1* a primit aceeași valoare, cu siguranță există acea valoare în containerul de stringuri și referința din *s2* va fi aceeași cu cea din *s1*. De aceea comparația *s1==s2* returnează *true*
- Comparatia *s3==s4* în schimb returnează *false*, pentru că *s3* și *s4* au fost alocate folosind operatorul *new* în memoria *heap* la adrese diferite ca au același conținut.
- Comparațiile *o1==o2* compară referințe de obiecte și pentru a compara conținutul obiectelor se folosește metoda *equals*. În cazul *string*-urilor metoda *equals* compara conținutul unul sir de caractere cu cel al altui sir de caractere. Metoda este *case-sensitive*. Pentru o comparație *case-insensitive* se folosește metoda *equalsIgnoreCase()*
- Lungimea unui *String*

```
String s="hello!";
System.out.println(sir.length()); // afișează:6
```

## ■ Concatenarea sirurilor

- cu operatorul +
- cu metoda concat() a clasei *String*

```
String s1="afara";
String s2="ploua";

System.out.println(s1+" "+s2); // afisează: afara ploua
System.out.println(s1.concat(" ").concat(s2)); // afisează: afara ploua
```

## ■ Convertirea sirurilor în numere

- Fiecare clasă înfăşurătoare dispune de o metodă *valueOf* care convertește un sir într-un obiect din clasa respectivă
- Fiecare clasă înfăşurătoare dispune de o metodă *parse...*, care convertește un sir într-o variabilă de tipul elementar corespunzător clasei înfăşurătoare în cauză (vezi subcapitolul 1.9).

## ■ Convertirea numerelor în siruri

- Fiecare clasă înfăşurătoare dispune de o metodă numită *toString()*, care convertește tipul primitiv corespunzător într-un sir

```
int i=3;
double d=7;
String s1=Integer.toString(i);
String s2=Double.toString(d);
```

- Obținerea caracterului dintr-o anumită poziție a unui sir

```
String alfa="Luna tu stapană marii, pe a lumii bolta luneci";  
char a=alfa.charAt(9); // variabila a va lua valoarea "t"
```

**Observație:** nu există o metoda *setCharAt()* care să permită modificarea valorii dintr-o anumită poziție a unui sir pentru ca *string-urile* sunt imutabile!

- Utilizarea unui delimiter pentru extragerea de subșiruri

```
String alfa="Exemplu cu split";  
String[] s=alfa.split(" ");  
System.out.println(s[0]); //afiseaza Exemplu
```

- O alternativă la utilizarea metodei *split()* este utilizarea clasei  *StringTokenizer* care permite specificarea delimiterului ori la instanțierea obiectului, ori ca și parametru a metodei *nextToken()*

```
StringTokenizer st = new StringTokenizer("Exemplu cu StringTokenizer", " ");  
while (st.hasMoreTokens()) {  
    System.out.println(st.nextToken());  
}
```

- Obținerea de subșiruri din șiruri de date

```
String alfa="Luna tu stăpană marii, pe a lumii bolta luneci ";
System.out.println(alfa.substring(8,15)); //Afisează stăpană
System.out.println(alfa.substring(8)); //Afisează stăpană marii, pe a lumii bolta luneci
```

- Generarea șirului de minuscule, respectiv a șirului de majuscule corespunzătoare unui șir dat

```
System.out.println(alfa.toUpperCase());
System.out.println(alfa.toLowerCase());
```

- Căutarea unui caracter într-un șir dat

```
System.out.println(alfa.indexOf('a'));    //Afiseaza 3
System.out.println(alfa.indexOf('a',4));    //Afiseaza 10
System.out.println(alfa.lastIndexOf('a'));//Afiseaza 38
System.out.println(alfa.lastIndexOf('a',37)); //Afiseaza 26
```

## ■ Căutarea unui subșir într-un sir dat

```
System.out.println(alfa.indexOf("un"));      //Afiseaza 1  
System.out.println(alfa.indexOf("un",2));    //Afiseaza 41  
System.out.println(alfa.lastIndexOf("un"));   //Afiseaza 41  
System.out.println(alfa.lastIndexOf("un",40)); //Afiseaza 1  
System.out.println(alfa.contains("un"));       //Afiseaza true
```

## ■ Înlocuirea unui caracter într-un sir dat

```
System.out.println(alfa.replace ('a', 'A'));
```

//Afisează: LunA tu stApAnA mArII, pe A lumii boltA luneci

## ■ Înlocuirea unui subșir într-un sir dat

```
System.out.println(alfa.replaceFirst ("un", "UN"));
```

//LUNA tu stapană marii, pe a lumii bolta luneci

```
System.out.println(alfa.replaceAll ("un", "UN"));
```

//LUNA tu stapană marii, pe a lumii bolta IUNeci

## ■ Compararea sirurilor și porțiunilor de siruri

```
System.out.println(alfa.startsWith("Luna")); //true  
System.out.println(alfa.startsWith("tu",5)); //true  
System.out.println(alfa.endsWith("luneci")); //false
```

```
String beta="abcd";  
  
System.out.println(beta.compareTo("abce")); // -1  
System.out.println(beta.compareToIgnoreCase("ABCD")); // 0  
System.out.println(beta.equals("ABCD")); // false  
System.out.println(beta.equalsIgnoreCase("ABCD")); // true  
System.out.println(beta.regionMatches(1, "aaaBCd", 3, 2)); // false  
System.out.println(beta.regionMatches(true, 1, "aaaBCd", 3, 2)); // true
```

- Eliminarea spațiilor de la începutul și finalul sirului se poate realiza cu funcția *trim()*
- În septembrie 2018, a fost lansat Java 11 care introduce noi metode în clasa String. Aceste sunt:
  - **repeat(nr)** – repetă string-ul de un număr de ori dat ca și parametru de intrare și returnează noul string obținut prin concatenare
  - **isBlank()** – determină dacă string-ul este gol sau are doar spații albe, returnează boolean
  - **strip()** – returnează stringul cu spațiile albe de la început și sfârșit eliminate

- ***stripLeading()*** – returnează string-ul cu spațiile albe de la început eliminate
  - ***stripTrailing()*** – returnează string-ul cu spațiile albe de la sfârșit eliminate
  - ***lines()*** – returnează un *stream* care conține liniile extrase din *string*, separate prin terminatorul de linie
- Exemplul următor arată cum pot fi utilizate aceste metode

```
package capitolul1.stringuri;

public class MainApp {
    public static void main(String []args) {
        String s = " test ";
        System.out.println("1:"+s.repeat(3));
        System.out.println("2:"+s.isBlank());
        System.out.println("3:"+"".isBlank());
        System.out.println("4:"+" ".isBlank());
        System.out.println("5:"+s.strip()+".");
        System.out.println("6:"+s.stripLeading()+".");
        System.out.println("7:"+s.stripTrailing()+".");
        s = "aaa\nbbb\nccc";
        s.lines().forEach(System.out::println);
    }
}
```

@ Javadoc Declaration

<terminated> MainApp (26) [

1:	test test test
2:	false
3:	true
4:	true
5:	test.
6:	test .
7:	test.
	aaa
	bbb
	ccc

- Java 12, lansat în martie 2019, aduce îmbunătățiri string-urilor prin introducerea următoarelor metode în clasa *String*:

- **metoda *indent(nr)***, metoda returnează un nou String în care se ajustează indentarea fiecărei linii a stringului pentru care se apelează, în funcție de parametrul transmis la apel. Dacă nr este pozitiv la începutul fiecărei linii se introduc *nr* spații. Dacă nr este negativ se șterg de la începutul fiecărei linii *nr* spații (dacă nu sunt suficiente se șterg cele existente). Dacă nr este 0, nu se fac adăugări sau ștergeri de spații
- **metoda *transform()*** care transformă un *string* într-un nou *string* cu ajutorul unei funcții transmisă ca și argument
- **metoda *describeConstable()*** care returnează un obiect *Optional* care descrie string-ul
- Exemplul următor ilustrează cum pot fi utilizate aceste metode

```
package capitolul1.stringuri;

import java.util.Optional;

public class MainApp2 {
    public static void main(String[] args) {
        String s1="linia 1\nlinia 2";
        System.out.println(s1.indent(4));
```

```

        String s2="    linia 1\n    linia 2"; //4 spatii
        System.out.println(s2.indent(-1));

        String s3="Oana";
        String s4=s3.transform(String::toUpperCase);
        System.out.println("s3="+s3+"\ns4="+s4);

        Optional<String> optional = s4.describeConstable();
        optional.ifPresent(System.out::println);
    }
}

```

- Ieșirea programului este următoarea:

```

@ Javadoc Declaration Console ✎ Progress
<terminated> MainApp (35) [Java Application] D:\kituri
|   linia 1
|   linia 2

linia 1
linia 2

s3=Oana
s4=OANA
OANA

```

## ■ Clasele *StringBuffer* și *StringBuilder*

- Obiectele *StringBuffer* permit modificarea sirului de caractere. De exemplu modificarea unui caracter se poate realiza cu ajutorul metodei *setCharAt()*. Stergerea unui caracter cu ajutorul metodei *deleteCharAt()*

```
String s="abc";
StringBuffer sb=new StringBuffer(s);
sb.setCharAt(1, 'Z');
System.out.println(sb); //output: aZc
sb.deleteCharAt(0);
System.out.println(sb); //output: Zc
```

- Clasa *StringBuffer* dispune de metode pentru:
  - Adăugarea unui tip primitiv la un obiect *StringBuffer* – metoda *append()*
  - Stergerea unei bucati dintr-un sir de caractere – metoda *delete()*
  - Inserarea unui subsir intr-un sir – metoda *insert()*
  - Inversarea literelor sirului de caractere – metoda *reverse()*
  - *Etc*
- Metodele publice ale clasei *StringBuffer* sunt sincronizate, ceea ce înseamnă că sunt sigure în aplicații cu fire de execuție, dar costul pentru acest lucru este performanta.
- Clasa ***StringBuilder*** dispune de metode similare cu *StringBuffer*, doar ca metodele sale nu sunt sincronizate și aceasta aduce un plus de viteza de execuție.
- *StringBuilder* este recomandat în aplicații fără fire de execuție, iar *StringBuffer* în aplicații cu fire de execuție

## 1.14 Date Time API

- De la prima versiune de Java și până în prezent API-ul pentru date calendaristice și timp a evoluat. Primul tip de data calendaristică introdus încă din prima versiune Java a fost tipul *Date*. Utilizarea acestuia este posibilă și în prezent dar nerecomandată pentru că este învechită (*deprecated*)
- Următorul tip de dată calendaristică introdus a fost tipul *Calendar* (dezavantaj, lunile pornesc de la zero)

```
Calendar c=Calendar.getInstance(); //modificare cu settere
System.out.println(c.get(Calendar.DAY_OF_MONTH)+". "+(c.get(Calendar.MONTH)+1)+". "
    +c.get(Calendar.YEAR));
System.out.println(c.get(Calendar.HOUR_OF_DAY)+":" +c.get(Calendar.MINUTE)+":"
    +c.get(Calendar.SECOND));
```

@ Javadoc   
<terminated>  
23.9.2022  
13:58:8

- În Java 8 au fost introduse tipurile *LocalDate*, *LocalTime*, *LocalDateTime*, etc care sunt ușor de utilizat și recomandate

```
LocalDateTime dt1 =LocalDateTime.now();
System.out.println("dt1: " + dt1);

Month luna = dt1.getMonth();
int a_cata_luna=dt1.getMonthValue();
System.out.println(Luna+ " month = month "+a_cata_Luna);
```

```
dt1: 2023-02-27T10:47:37.612350900
FEBRUARY month = month 2
```

```

LocalDateTime dt2 =dt1.withMonth(2).withDayOfMonth(2);
System.out.println("dt2: "+dt2);

LocalDate d1 = dt1.toLocalDate();
System.out.println("d1: " + d1+" data extrașă din dt1");

LocalDate d2 =LocalDate.now();
System.out.println("d2: " + d2);

LocalDate d3 = LocalDate.of(2012, Month.FEBRUARY, 14);
System.out.println("d3: " + d3);                                dt2: 2023-02-02T11:16:26.592126600
                                                               d1: 2023-02-27 data extrașă din dt1
                                                               d2: 2023-02-27
                                                               d3: 2012-02-14
                                                               t1: 21:30
                                                               t2: 20:15:30
                                                               d4: 2017-11-20
                                                               d5: 2023-06-07 data după 100 de zile de la data 2023-02-27
                                                               Urmatoarea zi de luni de după data 2023-02-27 este în data: 2023-03-06

LocalTime t1 = LocalTime.of(21, 30);
System.out.println("t1: " + t1);

LocalTime t2 = LocalTime.parse("20:15:30");
System.out.println("t2: " + t2);                                d5: 2023-06-07 data după 100 de zile de la data 2023-02-27
                                                               Urmatoarea zi de luni de după data 2023-02-27 este în data: 2023-03-06

LocalDate d4 = LocalDate.parse("2017-11-20");
System.out.println("d4: " + d4);

LocalDate d5 = d1.plus(100, ChronoUnit.DAYS);
System.out.println("d5: " + d5+" data după 100 de zile de la data "+d1);

LocalDate luna_viitoare =d1.with(TemporalAdjusters.next(DayOfWeek.MONDAY));
System.out.println("Luna viitoare este în : " + luna_viitoare);

```

```

long nr_zile1=Duration.between(d1.atStartOfDay(), d5.atStartOfDay()).toDays();
System.out.println("Intre " + d1 + " si " + d5 + " sunt: " + nr_zile1 + " zile");

long nr_zile2=ChronoUnit.DAYS.between(d1, d5);
System.out.println("Intre " + d1 + " si " + d5 + " sunt: " + nr_zile2 + " zile");

long nr_ani=ChronoUnit.YEARS.between(d3, d4);
System.out.println("Intre " + d3 + " si " + d4 + " sunt: " + nr_ani + " ani");

String s="20.03.2022";
LocalDate d6=LocalDate.parse(s, DateTimeFormatter.ofPattern("dd.MM.yyyy"));
System.out.println("d6: " + d6);

System.out.println("d6: " + d6.format(DateTimeFormatter.ofPattern("dd.MM.yyyy")));

```

```

Intre 2023-02-27 si 2023-06-07 sunt: 100 zile
Intre 2023-02-27 si 2023-06-07 sunt: 100 zile
Intre 2012-02-14 si 2017-11-20 sunt: 5 ani
d6: 2022-03-20
d6: 20.03.2022

```

# 1.15 Operatorii limbajului Java

## Operatorii unari:

+	Operatorul unar plus; indică o valoare pozitivă (sunt pozitive, de asemenea, numerele neprecedate de niciun operator)
-	Operatorul unar minus, neagă o expresie;
++	Operatorul de incrementare; incrementează o valoare cu 1
--	Operatorul de decrementare; decrementează o valoare cu 1
!	Operatorul de complementare logică; complementează o valoare booleană
~	Operatorul de complementare la nivel de bit

## Operatorii aritmetici:

*	Operatorul de înmulțire
/	Operatorul de împărțire
%	Modulo
+	Operatorul de adunare (folosit, de asemenea, pentru concatenarea sirurilor)
-	Operatorul de scădere

## Operatorii relationali și de egalitate

>	Operatorul mai mare
>=	Operatorul mai mare sau egal
<	Operatorul mai mic
<=	Operatorul mai mic sau egal
==	Operatorul de verificare a egalității
!=	Operatorul de verificare a inegalității

## Operatorii la nivel de bit

<<	Operatorul de deplasare la stânga cu semn
>>	Operatorul de deplasare la dreapta cu semn
>>>	Operatorul de deplasare la dreapta fără semn
&	Operatorul AND bit la bit
	Operatorul OR bit la bit
^	Operatorul XOR bit la bit
~	Operatorul unar NOT la nivel de bit

## Operatorii condiționali

<code>&amp;&amp;</code>	Operatorul logic AND
<code>  </code>	Operatorul logic OR
<code>?:</code>	Operatorul ternar (if-then-else)

## Operatorii de asignare

<code>=</code>	Operatorul de asignare simplă
<code>+=</code>	Operatorul de asignare postadunare ( $x+=y \Leftrightarrow x=x+y$ )
<code>-=</code>	Operatorul de asignare postscădere ( $x-=y \Leftrightarrow x=x-y$ )
<code>*=</code>	Operatorul de asignare postînmulțire ( $x*=y \Leftrightarrow x=x*y$ )
<code>/=</code>	Operatorul de asignare postîmpărțire ( $x/=y \Leftrightarrow x=x/y$ )
<code>%=</code>	Operatorul de asignare postmodulo ( $x\%=y \Leftrightarrow x=x \% y$ )
<code>&amp;=</code>	Operatorul de asignare postAND ( $x\&=y \Leftrightarrow x=x\&y$ )
<code> =</code>	Operatorul de asignare postOR ( $x =y \Leftrightarrow x=x y$ )
<code>^=</code>	Operatorul de asignare postXOR ( $x^=y \Leftrightarrow x=x^y$ )
<code>&lt;&lt;=</code>	Operatorul de asignare posteplasare_la_stânga_cu semn ( $x<<=y \Leftrightarrow x=x<<y$ )
<code>&gt;&gt;=</code>	Operatorul de asignare posteplasare_la_dreapta_cu semn ( $x>>=y \Leftrightarrow x=x>>y$ )
<code>&gt;&gt;&gt;=</code>	Operatorul de asignare posteplasare_la_dreapta_fără semn ( $x>>>=y \Leftrightarrow x=x>>y$ )

## Operatorul de verificare

<code>instanceof</code>	Operatorul de verificare a apartenenței unui obiect la un anumit tip
-------------------------	--

## Operatorii limbajului Java, în ordinea puterii de precedență

Denumirea operatorilor	Notația operatorilor
operatorii unari de postfixare	<i>expr++</i> , <i>expr--</i>
operatorii unari de prefixare	<i>++expr</i> , <i>--expr</i> , <i>+expr</i> , <i>-expr</i> , <i>~</i> , <i>!</i>
operatorii multiplicativi	<i>*</i> , <i>/</i> , <i>%</i>
operatorii aditivi	<i>+</i> , <i>-</i>
operatorii de deplasare	<i>&lt;&lt;</i> , <i>&gt;&gt;</i> , <i>&gt;&gt;&gt;</i>
operatorii relationali	<i>&lt;</i> , <i>&gt;</i> , <i>&lt;=</i> , <i>&gt;=</i> , <i>instanceof</i>
operatorii de egalitate	<i>==</i> , <i>!=</i>
operatorul bit la bit AND	<i>&amp;</i>
operatorul bit la bit XOR	<i>^</i>
operatorul bit la bit OR	<i> </i>
operatorul logic AND	<i>&amp;&amp;</i>
operatorul logic OR	<i>  </i>
operatorul ternar	<i>? :</i>
operatorii de asignare	<i>=</i> , <i>+=</i> , <i>-=</i> , <i>*=</i> , <i>/=</i> , <i>%=</i> , <i>&amp;=</i> , <i>^=</i> , <i> =</i> , <i>&lt;&lt;=</i> , <i>&gt;&gt;=</i> , <i>&gt;&gt;&gt;=</i>

- Pentru stabilirea ordinii dorite de realizare a operațiilor în expresii, se recomandă utilizarea parantezelor rotunde

## 1.16 Instrucțiunile limbajului Java

### ■ if-then-else

```
if(expresieLogica){ //secvența de instrucții dintre aceste accolade se execută doar dacă expresieLogica are valoarea true  
...  
}  
else{ //secvența de instrucții dintre aceste accolade se execută doar dacă expresieLogica are valoarea false  
...  
}
```

### ■ switch – varianta clasică

```
String anotimpul;  
String luna= "octombrie";  
switch (luna.toLowerCase()) {  
    case "decembrie":  
    case "ianuarie":  
    case "februarie":anotimpul = "iarna";break;  
    case "martie":  
    case "aprilie":  
    case "mai":anotimpul = "primavara";break;  
    case "iunie":  
    case "iulie":  
    case "august":anotimpul = "vara";break;  
    case "septembrie":  
    case "octombrie":  
    case "noiembrie":anotimpul = "toamna";break;  
    default: anotimpul=""; break;  
}  
System.out.println("Luna "+luna+" este in anotimpul "+anotimpul);
```

## ■ switch cu expresii

- A fost introdus ca și o caracteristică cu previzualizare în Java 12 (martie, 2019) și inclus ca și caracteristică permanentă în Java 14 (martie, 2020). Noul *switch* permite scrierea mai compactă dar și mai lizibilă a codului
- Switch-ul poate fi utilizat în continuare în forma nouă, ca o expresie sau în forma clasică

```
String luna= "octombrie";
String anotimpul=switch (luna.toLowerCase()) {
    case "decembrie","ianuarie","februarie"-> "iarna";
    case "martie","aprilie","mai"->"primavara";
    case "iunie","iulie","august"-> "vara";
    case "septembrie","octombrie","noiembrie"-> "toamna";
    default -> "";
};
System.out.println("Luna "+luna+" este in anotimpul "+anotimpul);
```

- După cum se observă noul *switch* permite scrierea mai compactă a codului și oferă posibilitatea de a scrie mai puțin cod pentru a realiza același lucru
- caracteristicile switch-ului cu expresii sunt următoarele:
  - case-urile pot enumera mai multe constante separate prin virgule
  - Nu se mai folosește break
  - *default* este obligatoriu
  - Case-urile sunt urmate de -> după care se specifică o valoare pe care switch-ul o va returna în cazul în care se intră pe acea ramură

## ■ while

```
while(expresieLogica){ // verifică dacă expresieLogica=true și execută instrucțiile dintre  
// acolade cât timp această condiție este îndeplinită  
...  
}
```

## ■ do..while

```
do{ // execută instrucțiile dintre acolade și apoi verifică dacă expresieLogica=true;  
// în caz afirmativ reia totul, altfel treci la instrucția următoare  
...  
} while(expresieLogica);
```

## ■ for

```
class MainApp{  
    public static void main(String[] args){  
        for(int i=1; i<11; i++){  
            System.out.println("Contorul este: " + i);  
        }  
    }  
}
```

```
class MainApp{  
    public static void main(String[] args){  
        int[] numbers={1,2,3,4,5,6,7,8,9,10};  
        for(int item:numbers){  
            System.out.println("Count is: " + item);  
        }  
    }  
}
```

## ■ break fără etichetă

- este utilizat în case-urile *switch*-ului clasic sau pentru a termina forțat o buclă de program atunci când aceasta și-a îndeplinit sarcina

```
class MainApp{  
    public static void main(String[] args){  
        int[] arrayOfInts={32,87,3,589,12,1076,2000,8,622,127};  
        int searchfor=12;  
        int i;  
        boolean foundIt=false;  
        for(i=0;i<arrayOfInts.length;i++){  
            if(arrayOfInts[i]==searchfor){  
                foundIt=true;  
                break;  
            }  
        }  
        if(foundIt){  
            System.out.println("Found "+searchfor+" at index "+i);  
        }  
        else{  
            System.out.println(searchfor+" not in the array");  
        }  
    }  
}
```

## ■ break cu etichetă

- Este utilizat pentru a încheia mai multe bucle. De exemplu dacă se dorește ca printr-o comandă break să se iasă din 2 for-uri imbricate, se pune o etichetă deasupra for-ului exterior și se folosește comanda *break eticheta*. Un break simplu, fără eticheta ar determina ieșirea doar din for-ul interior.

```
package capitolul1.break_cu_eticheta;
public class MainApp {
    public static void main(String[] args) {
        int [][]matrice= {{1,2,3},{4,5,6},{7,8,9}};
        int x=3;
        boolean gasit=false;
        int i=0;
        int j=0;
        eticheta:
        for (i=0;i<matrice.length;i++) {//for 1
            System.out.println("Se cauta pe Linia "+i);
            for(j=0;j<matrice[i].length;j++) { //for 2
                if(matrice[i][j]==x) {
                    gasit=true;
                    break eticheta;
                    //break;
                }
            }
        }//end for2
```

```

        //if (gasit) break;
    } //end for1
    if(gasit)
        System.out.println("Elementul "+x+" a fost gasit pe randul "+i
            +" si colana "+j);
    else
        System.out.println("Elementul "+x+" nu a fost gasit in matrice");
} //end main
} //end class

```

- În exemplul precedent dacă se folosește instrucțiunea *break* fără etichetă (se pune în comentariu *break*-ul cu etichetă și se scoate din comentariu *break*-ul fără etichetă) pentru ca programul să funcționeze corect trebuie făcută încă o testare a variabilei *gasit* imediat după *for*-ul al doilea (scos din comentariu *if (gasit) break;*)
- *Break*-ul cu eticheta se poate utiliza și pentru a ieși dintr-un bloc *catch*. Acest aspect va fi discutat în capitolul de tratare a excepțiilor

## ■ continue fără etichetă

- Determină trecerea la următoarea iterare (sare peste iterare curentă) în cadrul unui `for`, `while` sau `do...while`

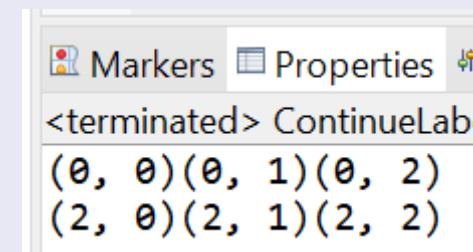
```
class ContinueDemo{  
    public static void main(String[] args){  
        String searchMe="Un vultur sta pe pisc c-un pix in plisc";  
        int max=searchMe.length();  
        int numPs=0;  
  
        for(int i=0;i<max;i++){  
            if(searchMe.charAt(i)!='p')  
                continue;  
            numPs++;  
        }  
        System.out.println("Found "+numPs+" p's in the string.");  
    }  
}
```

## ■ continue cu etichetă

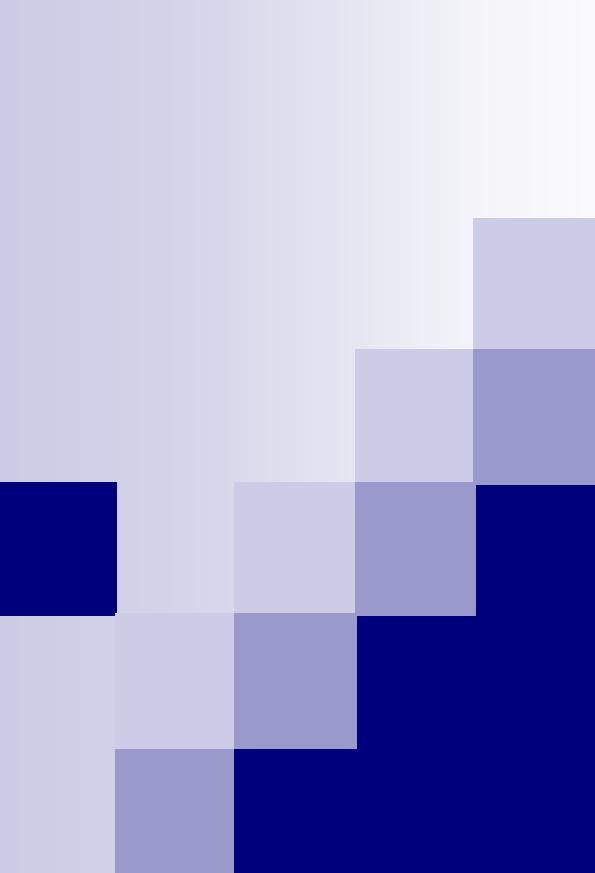
- Determină trecerea la următoarea iterație a buclei exterioare, care este marcată cu eticheta aleasă

```
package capitolul1.continue_cu_eticheta;

public class ContinueLabelDemo {
    public static void main(String[] args) {
        eticheta: for (int i = 0; i < 3; i++) {//for 1
            for (int j = 0; j < 3; j++) {//for 2
                if (i == 1) {
                    continue eticheta;
                } //end if
                System.out.print("(" + i + ", " + j + ")");
            } //end for 2
            System.out.println();
        } //end for 1
    } //end main
} //end class
```



- Exemplul precedent generează și afișează indicii i și j care ar putea fi utilizați pentru a accesa elementele de pe prima și ultima linie a unei matrici de 3 x 3.
  - Se utilizează instrucția continue cu etichetă pentru a sări peste elementele care indică linia 1. Dacă i are valoarea 1, se executa continuă etichetă, care determină for-ul 1 (for-ul exterior) să treacă la următoarea iteratie
- 
- *return*
    - În metodele de tip *void* se folosește *return*; care asigura ieșirea din metodă
    - În metodele de alt tip decât *void* se folosește *return expresie*, unde *expresie* trebuie să fie în consonanță cu tipul folosit la declararea metodei



## 2. Orientarea pe obiecte

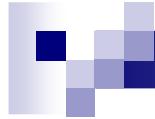
Sl. dr. ing. Raul Robu

2022-2023, Semestrul 2



# CUPRINS

- 2.1 Noțiuni generale
- 2.2 Moștenirea
- 2.3 Variabile și metode statice. Secvențe de cod statice. Importuri statice
- 2.4 Constructori
  - 2.4.1 Notiuni generale
  - 2.4.2 Şablonul de proiectare Singleton
  - 2.4.2 Legarea dinamică și constructorii
- 2.5 Modificatori de acces
- 2.6 Distrugerea obiectelor
- 2.7 Mostenire și polimorfism
- 2.8 Enumerări
- 2.9 Serializarea obiectelor
- 2.10 Generarea documentației
- 2.11 Redefinirea metodelor
- 2.12 Clase generice
- 2.13 Metode generice
- 2.14 Clase imbricate
  - 2.14.1 Noțiuni generale
  - 2.14.2 Clasă interioară membră
  - 2.14.3 Clasă interioară locală



# CUPRINS

- 2.14.4 Clasă interioară anonimă
- 2.14.5 Clase interioare statice
- 2.14.6 Facilități pentru operarea cu clase imbricate (Java 11)
- 2.15 Clase abstracte
- 2.16 Interfețe
  - 2.16.1 Noțiuni generale
  - 2.16.2 Metode implicite, metode statice (Java 8) și metode private în interfețe (Java 9)
  - 2.16.3 Interfețe funcționale (Java 8)
- 2.17 Expresii Lambda
- 2.18 Referințe de metode
- 2.19 Stream API
- 2.20 Clasa *Optional*
- 2.21 Inferență tipului la varibilele locale (local variable type inference)
- 2.22 Tipul înregistrare (record)
- 2.23 Design patterns (Factory, Abstract Factory, Facade, Command, Model View Controller)

## 2.1 Noțiuni generale

- Un program este format din una sau mai multe **clase**.
- O clasă reprezintă descrierea unei mulțimi de obiecte care au aceeași structură și același comportament. Ca urmare, într-o clasă vom găsi definițiile datelor și ale operațiilor ce caracterizează obiectele clasei respective
- Clasa reprezintă un tipar după care se creează un obiect prin instanțiere
- Declararea unei clase este similară cu declararea unui nou tip de date
- Majoritatea conceptelor legate de orientarea pe obiecte sunt similare cu cele din C++:
  - **Încapsularea** – accesul la variabilele membre ale unui obiect se poate realiza doar cu ajutorul metodelor obiectului
  - **Supraîncărcarea** – mai multe metode pot avea același nume dar o semnătură diferită (numărul și tipul parametrilor diferă de la o metodă la alta)
  - **Moștenirea** - constă în extinderea comportamentului unei clase existente prin definirea unei clase noi, care moștenește conținutul primei clase, adăugând la acesta elementele specifice ei
  - **Legarea dinamică** - asocierea unui apel de metodă cu funcția ce trebuie apelată se numește "legare" ("Binding"). Legarea se poate face la compilare ("legare timpurie" sau statică) sau la execuție ("legare târzie" sau "legare dinamică"). În Java pentru metodele finale sau statice legarea este timpurie iar pentru celelalte legarea este dinamică

- Orice construcție are un tip stabilit la momentul compilării
- Nu există tipuri implicate pentru metode și nici posibilitatea ca la apelul unei metode să fie omisi anumiți parametri ("strong typed")
- Polimorfismul este abilitatea unui obiect de a lua mai multe forme. Una din cele mai frecvente utilizări ale polimorfismului în POO este când o referință a unei clase de bază este utilizată pentru a referi un obiect al clasei derivate
- O funcție sau operator este ***suprîncărcat*** dacă execută operații diferite în contexte diferite (de exemplu operatorul + care poate fi adunare sau concatenare de siruri)
- O **clăsă** poate fi ***extinsă*** prin *redefinirea unor metode și / sau adăugarea unor metode noi*
- O metodă care redefineste o alta din clasa de bază are același nume, același tip de rezultat și aceeași listă de parametri
- În cazul în care o clasă derivată are variabile membre cu același nume ca și clasa de bază nu este vorba de o redefinire, cei doi parametri coexistă și pot să referi distinct
- ***final*** – blochează redefinirea. O metodă finală nu mai poate fi redefinită, iar o clasă finală nu mai poate fi extinsă

- Moștenirea
  - Permisă doar moștenirea simplă, spre deosebire de C++
  - Clasa moștenită – **superclasă** sau **supraclasă** sau **clasă de bază**
  - Clasa care realizează extinderea - **subclasă** sau **clasă derivată**
- Relația de moștenire între clase este o relație de formă "**is A**" (este un fel de) adică "subclasa este un fel de superclasa". Dacă considerăm clasa de bază *Vapor* și clasa drivată *VasDeCroaziera*, putem spune că vasul de croazieră este tot un tip de vapor
- Agregarea și compozitia sunt relații de asociere între clase de formă "**has A**" (are o). Agregarea indică o asociere slabă între clase iar compozitia o asociere puternică între clase (*belongs-to* sau *part-of*)
- În cazul agregării distrugerea obiectului principal nu implică distrugerea obiectelor de legătură, acestea continuă să existe (legătură slabă).
- În cazul componenței distrugerea obiectului principal implică distrugerea obiectului de legătură, clasa obiectului principal fiind cea care creează instanța obiectului de legătură (legătură puternică).
- O firmă are mai multe departamente și mai mulți angajați. Închiderea firmei duce la desființarea departamentelor, deci acestea au o legătură puternică cu firma (compoziție). Pe de altă parte angajații continuă să existe și se pot angaja la alte companii (legătură slabă - agregare).
- În anumite situații o legătură poate fi considerată puternică sau slabă în funcție de context sau în funcție de modul în care este privită acea legătură

- Un constructor este o metodă care același nume cu clasa și este apelată implicit la crearea obiectului
- Un obiect reprezintă **încapsularea** unor date asupra cărora nu se pot executa decât anumite operații numite **metode**
- **Signatura unei metode** – reprezintă tipul returnat de metoda, numele acesteia și lista sa de parametri
- **Interfață unui obiect** – modul în care un obiect este cunoscut din exterior
- În momentul în care o clasă este necesară (pentru că se instanțiază un obiect de tipul respectiv sau pentru că este apelată o metodă statică din clasa respectivă) aceasta se va încărca în mașina virtuală. Încărcarea este realizată de către un obiect numit „*class loader*”.
- Clasele formează o ierarhie, care are la rădăcină clasa *Object*
- Rădâcina ierarhiei de clase, clasa **Object** conține o serie de **metode** care vor fi moștenite de orice clasă definită în Java. Unele din aceste metode pot fi redefinite, altele nu pot fi redefinite fiind finale

- Definiția clasei **Object** este conținută în biblioteca **java.lang** și va fi încărcată de pe mașina pe care se execută programul care o utilizează.

- Metodele **finale** ale clasei *Object* sunt următoarele:

- `public final Class getClass ()`
- `public final void notify () throws IllegalMonitorStateException`
- `public final void notifyAll () throws IllegalMonitorStateException`
- `public final void wait (long timeout) throws InterruptedException`
- `public final void wait() throws InterruptedException`

- Metodele clasei *Object* care pot fi redefinite sunt următoarele:

- `public boolean equals (Object obj)`
- `public int hashCode()`
- `public String toString ()`
- `protected void finalize() throws Throwable`
- `protected Object clone() throws CloneNotSupportedException`

## 2.2 Moștenirea

- Extinderea comportamentului unei clase existente prin definirea unei clase noi, care moștenește conținutul primei clase, adăugând la acesta elementele specifice ei
- Clasa moștenită se numește clasă de bază, supraclasă sau superclasă, iar clasa care realizează extinderea se numește subclasă, clasă derivată, sau clasă descendentală.
- Relația de moștenire între clase este o relație de forma "**is A**" (este un fel de). Dacă considerăm clasa de bază *Poligon* și clasa derivată *Dreptunghi*, putem spune că dreptunghiul este un fel de poligon
- Implementarea moștenirii se realizează cu ajutorul cuvântului cheie **extends** și nu este permisă moștenirea multiplă

```
class Nume_subclasa extends Nume_supraclasa{  
    //conținut specific subclasei  
}  
  
package capitolul2.mostenire;  
  
class Poligon {  
    protected double[] laturi;  
    public Poligon(int n) {  
        laturi = new double[n];  
    }  
    public double perimetru( ) {  
        double s=0;  
        for(int i=0;i<laturi.length;i++) s+=laturi[i];  
        return s;  
    }  
}
```

```

final class Dreptunghi extends Poligon {
    public Dreptunghi(double L, double h){
        super(4);
        laturi[0] = laturi[2] = L;
        laturi[1] =laturi[3] = h;
    }
    public double aria( ){
        return laturi[0]*laturi[1];
    }
}

class MainApp {
    public static void main(String []args) {
        Dreptunghi d=new Dreptunghi(3, 2);
        System.out.println("Perimetru dreptunghiului este "+d.perimetru());
        System.out.println("Aria dreptunghiului este "+d.aria());
    }
}

```

@ Javadoc Declaration Console Progress  
 <terminated> MainApp2 (1) [Java Application] D:\kitur  
 Perimetru dreptunghiului este 10.0  
 Aria dreptunghiului este 6.0

- Clasa *Dreptunghi* este finală, aşadar nu mai poate fi extinsă
- Nefiind permisă moştenirea multiplă, clasa *Dreptunghi* nu poate extinde şi alte clase
- Se pot crea şi alte clase derivate din clasa *Poligon* precum *Triunghi*, *Patrat*, *Romb*, etc

## *Clase derivate din clasa Object*

*în mod explicit:*

```
class Chitara extends Object{  
    String culoare;  
    int nr_corzi;  
}
```

*în mod implicit:*

```
class Chitara{  
    String culoare;  
    int nr_corzi;  
}
```

```
Chitara c; //variabilă initializată cu null  
//...  
c=new Chitara();
```

*sau*

```
Chitara c=new Chitara();
```

*Accesul la câmpurile obiectului creat, în interiorul pachetului*

```
c.culoare="neagra";  
c.nr_corzi=12;
```

## 2.3 Variabile și metode statice. Secvențe de cod statice. Importuri statice

- Variabilele statice ale unei clase sunt *variabilele globale ale clasei*, adică variabile a căror valoare poate să fie referată de orice obiect din clasa respectivă
- Pentru o variabilă membru statică se *alocă memorie o singură* dată, indiferent de numărul de obiecte de acel tip
- Variabilele și metodele declarate static se consideră că *aparțin clasei* nu obiectelor clasei respective, de aceea ele se *accesează de obicei utilizând numele clasei*:

```
NumeClasa.nume_variabila_statica  
NumeClasa.nume_metoda_statica()
```

- Variabilele statice se mai numesc variabile clasă (*class variables*)
- Variabilele locale NU pot fi statice
- O metodă statică poate să refere doar variabile sau metode statice, dar poate să fie apelată de orice metodă a clasei
- *Metodele statice sunt similare funcțiilor obișnuite din limbajul C*

```
Class Aplicatie{  
    static final int VERSIUNE=3; //constanța  
    static int numarObiecte;  
    ...  
}
```

- *VERSIUNE* este similară unei constante întregi și globale din limbajul C
- *numarObiecte* poate fi actualizată de orice obiect creat pentru a contoriza numărul total de obiecte de tip *Aplicatie*
- exemplul de mai jos arată că pentru o variabilă membru statică se *alocă memorie o singură* dată indiferent de numărul de obiecte de acel tip

```

package capitolul2.statice;

class DespreStatic {
    int x;
    static int y;
    DespreStatic(){
        x=0;
        y=0;
    }
}
class MainApp1{
    public static void main(String args[]){
        DespreStatic t1=new DespreStatic();
        DespreStatic t2=new DespreStatic();
        t1.x=10;
        t1.y=10;

        t2.x=20;
        t2.y=20;
        System.out.println("t1.x="+t1.x+" t1.y="+t1.y);
        System.out.println("t2.x="+t2.x+" t2.y="+t2.y);

        System.out.println();    //modul de acces recomandat
        System.out.println("t1.x="+t1.x+" DespreStatic.y="+DespreStatic.y);
        System.out.println("t2.x="+t2.x+" DespreStatic.y="+DespreStatic.y);
    }
}

```

The screenshot shows the Eclipse IDE interface with the 'Console' tab selected. The output window displays the following text:

```

Problems @ Javadoc Console X
<terminated> Test [Java Application] C:\Program
t1.x=10 t1.y=20
t2.x=20 t2.y=20

t1.x=10 DespreStatic.y=20
t2.x=20 DespreStatic.y=20

```

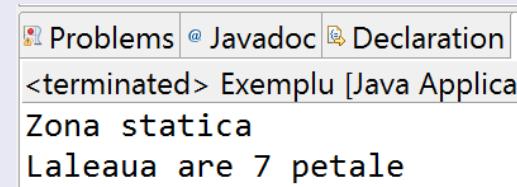
## Secvențe de cod statice

```
static{
    //secvența de cod
}
```

- Secvențe de cod static se pot declara numai în afara metodelor
- Aceste secvențe se execută în momentul în care se referă clasa care le conține

```
package capitolul2.statice;

class Floare {
    protected int numarPetale;
    protected String nume;
    Floare(int numarPetale, String nume){
        this.numarPetale = numarPetale;
        this.nume = nume;
    }
    public void afiseaza(){
        System.out.println(nume + " are " + numarPetale + " petale ");
    }
    static{
        System.out.println("Zona statica");
    }
}
class MainApp2{
    static{
        Floare g = new Floare(7, "Laleaua");
        g.afiseaza();
    }
    public static void main(String args[]){
    }
}
```



# Importuri statice

- Importurile statice permit accesul la variabilele statice ale unei clase fără a mai fi necesară specificarea numelui clasei
- De exemplu pentru apelul metodelor statice *random()*, *sqrt()*, *pow()* din clasa *Math* este necesară specificarea numelui clasei

```
package capitolul2.statice;

class MainApp3 {
    public static void main(String[] args) {
        System.out.println(Math.random());
        System.out.println(Math.sqrt(4));
        System.out.println(Math.pow(2, 4));
    }
}
```

- Dacă se folosește import static metodele pot să fie apelate direct, fără nume clasă

```
package capitolul2.statice;

import static java.lang.Math.random;
import static java.lang.Math.sqrt;
import static java.lang.Math.pow;

class MainApp4 {
    public static void main(String[] args) {
        System.out.println(random());
        System.out.println(sqrt(4));
        System.out.println(pow(2, 4));
    }
}
```

## 2.4 Constructori

### 2.4.1 Noțiuni generale

- Metode speciale apelate la crearea obiectelor
- Au același nume ca și numele claselor în care se găsesc și nu returnează nimic
- Un constructor nu poate fi apelat din alte metode, cu excepția altui constructor
- În mod implicit prima instrucțiune dintr-un constructor este apelul constructorului fără parametri ai superclasei. Singura abatere de la această regulă are loc atunci când prima instrucțiune din constructor este un apel explicit la alt constructor al clasei sau la un constructor al supraclasei.
- Dacă într-o clasă nu se definește nici un constructor, atunci este furnizat automat un constructor fără argumente (numit și constructor implicit) al cărui corp conține numai un apel al constructorului implicit al superclasei
- Dacă într-o clasă se definește cel puțin un constructor, constructorul implicit fără argumente nu mai este furnizat automat. Constructorii nu pot avea atributele *abstract*, *native*, *static*, *synchronized* sau *final*

- În general modifierul de acces al constructorului este *public*, dar sunt situații de excepție în care constructorul este *privat* (design pattern-ul *singleton*)

```
class Chitara {  
    private String culoare;  
    private int nr_corzi;  
    public Chitara(){  
        culoare="Neagra";  
        nr_corzi=6;  
    }  
    public Chitara(String culoare,int nr_corzi){  
        this.culoare=culoare;  
        this.nr_corzi=nr_corzi;  
    }  
}
```

SAU:

```
class Chitara {  
    private String culoare;  
    private int nr_corzi;  
    public Chitara(String culoare,int nr_corzi){  
        this.culoare=culoare;  
        this.nr_corzi=nr_corzi;  
    }  
    public Chitara(){  
        this ("Neagra",6);  
    }  
}
```

- Apelul constructorului supraclasselor se poate face în felul următor:

**super(lista\_de\_parametrii);**

## 2.4.2 Şablonul de proiectare *Singleton*

- *Singleton* este un şablon de proiectare (design pattern) care este utilizat pentru a restricţiona numărul de instanţieri ale unei clase la un singur obiect
- La baza pattern-ului Singleton stă o metodă ce permite crearea unei noi instanţe a clasei dacă aceasta nu există deja. Dacă instanţă există deja, atunci întoarce o referinţă către obiectul existent. Pentru a asigura o singură instanţiere a clasei, constructorul trebuie făcut *private*
- Uneori este important să avem doar un singur obiect pentru o clasă. De exemplu, într-un sistem ar trebui să existe un singur manager de ferestre (sau doar un sistem de fişiere). De obicei, *singletons* sunt folosite pentru managementul centralizat al resurselor interne sau externe
- Crearea unui *Singleton* presupune crearea unui clase cu:
  - Un constructor *privat* care nu va permite crearea de obiecte din afara clasei
  - O metodă statică de obicei numită *getInstance()* care returnează un obiect de tip *Singleton*
  - O variabilă membru privată şi statică de tip *Singleton* care se poate instanţia când clasa este încărcată (*early instantiation*) sau la cerere atunci când crearea instanţei este cerută (*lazy instantiation*)

```

package capitolul2.construtori.singleton_early;//early instantiation

class Singleton {
    //creaza un obiect al clasei Singleton
    private static Singleton instance = new Singleton();

    //Constructorul este privat deci nu se poate apela din alta clasa pentru instantiere
    private Singleton() {}

    //Returneaza singurul obiect disponibil
    public static Singleton getInstance(){
        return instance;
    }

    public void showMessage(){
        System.out.println("Ok!");
    }
}

class MainApp {
    public static void main(String[] args) {

        //Instantierea de mai jos da eroare. Constructorul privat nu este vizibil.
        //Singleton object = new Singleton();

        //Preia singurul obiect disponibil
        Singleton object = Singleton.getInstance();

        //Afiseaza mesajul
        object.showMessage();
    }
}

```

```

package capitolul2.construtori.singleton_lazy;//lazy instantiation

class Singleton {
    //creaza un obiect al clasei Singleton
    private static Singleton instance;

    //Constructorul este privat deci nu se poate apela din alta clasa pentru instantiere
    private Singleton() {}

    //Returneaza singurul obiect disponibil
    public static Singleton getInstance(){
        if (instance==null)
            instance=new Singleton();
        return instance;
    }

    public void showMessage(){
        System.out.println("Ok!");
    }
}

class MainApp {
    public static void main(String[] args) {

        //Instantierea de mai jos da eroare. Constructorul privat nu este vizibil.
        //Singleton object = new Singleton();

        //Preia singurul obiect disponibil
        Singleton object = Singleton.getInstance();

        //Afiseaza mesajul
        object.showMessage();
    }
}

```

## 2.4.3 Legarea dinamică și constructorii

- Deoarece în Java la apelul metodelor non-statice se aplică legarea dinamică, pe de o parte, și ținând cont de modul în care se apelează constructorii într-o ierarhie de clase, pe de altă parte, trebuie să avem grijă cum proiectăm constructorii dacă aceștia apelează la rândul lor metode ale claselor respective.
- *Fișierul SuperClasa.java*

```
package capitolul2.constructori.legarea_dinamica;

class SuperClasa {
    protected int a;
    private int x;

    public SuperClasa() {
        a = 2;
        x = calcul();
    }

    public int calcul() {
        return 2 * a;
    }

    @Override
    public String toString() {
        return a+", "+x;
    }
}
```

## ■ Fișierul SubClasa.java

```
package capitolul2.constructori.legarea_dinamica;

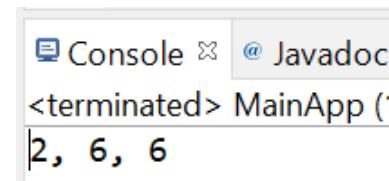
class SubClasa extends SuperClasa{
    private int y;
    public SubClasa() {
        y = calcul();
    }
    @Override
    public int calcul() {
        return 3 * a;
    }
    @Override
    public String toString() {
        return super.toString() + ", "+y;
    }
}
```

## ■ Fișierul MainApp.java

```
package capitolul2.constructori.legarea_dinamica;

class MainApp {
    public static void main(String[] a) {
        SubClasa ob=new SubClasa();
        System.out.println(ob);
    }
}
```

- Utilizarea anotăției `@Override` nu este obligatorie dar este recomandată pentru că aceasta asigură că metoda de dedesubtul ei redefineste o altă metodă din clasa de bază
- La instanțierea obiectului *ob* în programul principal se apelează constructorul fără parametri din *Subclasa*.
- Prima linie din acesta, este una implicită care realizează apel la constructorul fără parametri din *SuperClasa*. Astfel înainte de a se executa *y = calcul();* se vor executa liniile de cod din constructorul superclasei.
- Astfel a primește valoarea 2, iar *x* va primi valoarea returnată de metoda *calcul()*. Întrucât există două metode *calcul*, una în clasa de bază și una în clasa derivată (care o redefineste pe cea din clasa de bază) se pune problema care din cele două metode se va apela în linia *x=calcul();* Valorile variabilelor *x* și *y* depind de care din metodele de *calcul* se vor executa la apel. Legarea dinamică este asocierea dintre un apel de metodă și metoda care se va executa efectiv la rulare.
- Apelul metodei *calcul()* din constructorul clasei de bază determină execuția metodei *calcul* din clasa obiectului în curs de instanțiere, aceasta este metoda *calcul* din *SubClasa* și în acest fel ***x va primi valoarea 6***.
- Constructorul din *SuperClasa* se încheie, execuția codului continuă cu liniile de cod din constructorul clasei deriveate, mai exact cu instrucțiunea *y = calcul();* și în acest caz se va apela metoda *calcul* din clasa obiectului în curs de instanțiere, deci din *SubClasa*, *y primind valoarea 6*.
- În acest fel ieșirea programului va fi



```
Console  @ Javadoc
<terminated> MainApp (1)
2, 6, 6
```

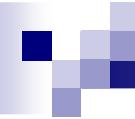
## 2.5 Modificatori de acces

- Principiul încapsulării presupune ca variabilele membre ale claselor să fie ascunse și modificarea lor să se poată face numai cu ajutorul metodelor definite în cadrul claselor respective, de exemplu:

```
package capitolul2.modificatori_acces;

class Chitara {
    private String culoare;
    private int nr_corzi;
    public Chitara(String culoare,int nr_corzi){
        this.culoare=culoare;
        this.nr_corzi=nr_corzi;
    }
    public Chitara() {
        this ("Neagra",6);
    }
    public String getCuloare () {
        return culoare;
    }
    public void setCuloare (String culoare) {
        this.culoare=culoare;
    }
    //...
}
```

- În *IntelliJ* *getterele* și *setterele* se generează apăsând *Alt + Insert* atunci când cursorul se află în interiorul clasei de interes și alegând comanda *Generate getter and setter*
- În Eclipse metodele de tip get set pot fi generate alegând opțiunea de meniu **Source** și apoi **Generate Getters and Setters...** atunci când cursorul se află în interiorul clasei de interes



### ■ ***public***

- se poate aplica constructorilor, metodelor, variabilelor membre și claselor
- o clasă publică poate fi importată într-un alt pachet decât cel în care este declarată și utilizată acolo
- o clasă care nu are atributul public este vizibilă doar la nivelul pachetului în care se află
- o clasă publică trebuie amplasată într-un fișier *java* cu aceeași denumire ca și clasa

### ■ ***private***

- se poate aplica constructorilor, variabilelor membre, metodelor și claselor interioare
- restrâne nivelul de vizibilitate al elementelor care au acest atribut doar la nivelul clasei

### ■ ***protected***

- se poate aplica constructorilor, variabilelor membre, metodelor și claselor interioare
- metodele și variabilele membre *protected* sunt accesibile din metodele clasei și din metodele subclaselor
- un obiect al unei clase care conține variabile și metode *protected* nu dispune de acces la acestea din afara pachetului (accesul este permis doar dacă obiectul se găsește în același pachet cu clasa)

```
package capitolul2.modificatori_acces;

class Test {
    protected String camp_protected;
}

class MainApp1 {
    public static void main(String []args) {
        Test t=new Test();
        t.camp_protected="acces permis";
    }
}
```

## ■ Modificatorul de acces implicit (package)

- Când nu se specifică un modificador de acces, accesul este limitat la nivelul pachetului. Variabilele membre sau metodele pentru care nu se specifică un modificador de acces sunt accesibile din alte clase ale aceluiași pachet
- În exemplul de mai jos clasa *Test* și clasa *MainApp* sunt în același pachet (*exemplu2*)

```
package capitolul2.modificatori_acces;

public class Testul {
    String camp;
}

class MainApp2 {
    public static void main(String []args) {
        Testul t=new Testul();
        t.camp="Acces permis";
    }
}
```

- Pentru ca o clasă să poată fi utilizată într-un alt pachet decât cel în care este definită ea trebuie să fie publică și apoi trebuie importată. În continuare, clasa *OClasa* din pachetul *capitolul2.modificatori\_acces.test*, importă clasa *Testul* pentru a putea declara obiecte de tip *Testul*. Accesul la variabilele membre implicate nu este permis în acest caz

```
package capitolul2.modificatori_acces.test;

import capitolul2.modificatori_acces.Testul;

class OClasa {
    void metoda() {
        Testul t=new Testul();
        //t.camp="Acces interzis";

    }
}
```

- În ceea ce privește vizibilitatea pe relație de moștenire, variabilele membre și metodele *private* ale unei clase de bază nu sunt vizibile în clasele derivate, cele *protected* și *public* sunt vizibile, iar cele fără modificador de acces sunt vizibile doar dacă cele 2 clase sunt în același pachet

## 2.6 Distrugerea obiectelor

- Nu cade în sarcina programatorului, ci în sarcina unei componente a mașinii virtuale numita *Garbage Collector*
- Dacă spre un anumit obiect nu mai există nici o referință externă, în nici o funcție activă, acel obiect devine candidat la eliminarea din memorie
- Înainte de a elimina un obiect din memorie este apelată metoda *finalize*  

```
protected void finalize() throws Throwable;
```
- În metoda *finalize* pot fi scrise instrucțiuni care să reactiveze obiectul, sau poate fi afișat un mesaj care precizează că obiectul urmează să fie distrus
- La crearea unui obiect se apelează o ierarhie de constructori începând cu constructorul clasei *Object*, iar la distrugerea lui se apelează un lanț de metode *finalize*
- Dacă există un cod care trebuie să se execute când un obiect nu mai este util atunci fie se scrie codul într-o metodă care se apelează în mod explicit, fie se apelează în mod explicit *Garbage Collector*, pentru că altfel nu există garanția că obiectul va fi distrus până se încheie programul

- Exemplul de mai jos arată cum se poate utiliza metoda *finalize*

```
package capitolul2.gc;

class Test {
    private int x;

    public Test(int x) {
        this.x = x;
    }

    public int getX() {
        return x;
    }

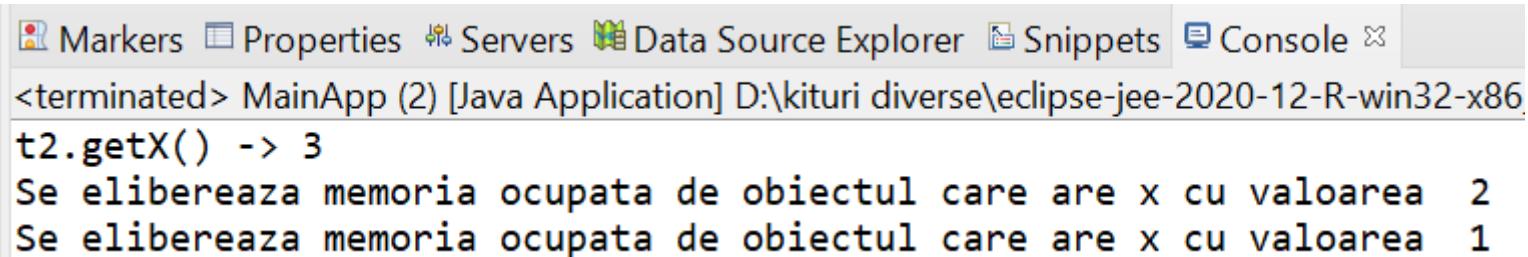
    @Override
    protected void finalize() throws Throwable {
        System.out.println("Se elibereaza memoria ocupata de obiectul "
            + "care are x cu valoarea " + this.x);
    }
}
```

```
package capitolul2.gc;

public class MainApp {
    public static void main(String[] args) {
        Test t1 = new Test(1);
        t1 = null;

        Test t2 = new Test(2);
        Test t3 = new Test(3);
        t2 = t3;

        System.out.println("t2.getX() -> "+t2.getX());
        System.gc();
    }
}
```



```
<terminated> MainApp (2) [Java Application] D:\kituri diverse\eclipse-jee-2020-12-R-win32-x86
t2.getX() -> 3
Se elibereaza memoria ocupata de obiectul care are x cu valoarea 2
Se elibereaza memoria ocupata de obiectul care are x cu valoarea 1
```

- Apelul metodei statice **System.gc()** determină rularea *Garbage Collector* în mașina virtuală *java*. Fără acest apel *garbage collector* nu rulează până programul își încheie execuția
- Rularea *garbage collector* determină eliberarea zonelor de memorie alocate spre care nu mai sunt referințe active.
- În exemplul precedent, atribuirea *t1=null* determină pierderea referinței către zona de memorie rezervată pentru obiectul *t1* (care are *x* cu valoarea 1). De asemenea atribuirea *t2=t3* face ca referința către zona de memorie spre care indică *t2* (cea care conține *x* cu valoare 2) să se piardă
- *garbage collector* eliberează memoria ocupată de obiectele menționate mai sus și execută pentru fiecare obiect dezalocat metoda *finalize*

## 2.7 Moștenire și polimorfism

- Polimorfismul este abilitatea unui obiect de a lua mai multe forme. Una din cele mai frecvente utilizări ale polimorfismului în POO este când o referință a unei clase de bază este utilizată pentru a referi un obiect al clasei derivate
- Relația de moștenire dintre clase, oferă suport pentru polimorfism
- Relația de moștenire între clase este o relație de forma "*is A*" (este un fel de) adică "subclasa este un fel de superclasă". Dacă considerăm clasa de bază *Vapor* și clasa derivată *VasDeCroaziera*, putem spune că un vas de croazieră este tot un tip de vapor. Datorită acestui lucru putem face o referință a clasei de bază *Vapor* să refere un obiect al clasei derivate *VasDeCroaziera*

```
Vapor v=new VasDeCroazaiera();
```

- Polimorfismul mai poate fi implementat folosind interfețele și clasele care le implementează, acest lucru va fi discutat în subcapitolul dedicat interfețelor

```
package capitolul2.conversii;

class Vapor{
    private String denumire;
    private int nr_membrii_echipaj;
    public Vapor(String denumire, int nr_membrii_echipaj) {
        this.denumire = denumire;
        this.nr_membrii_echipaj = nr_membrii_echipaj;
    }
    public String getDenumire() {
        return denumire;
    }
    @Override
    public String toString() {
        return denumire + ", " + nr_membrii_echipaj;
    }
}
class VasDeCroaziera extends Vapor{
    private int nr_restaurante;
    private int nr_piscine;
    VasDeCroaziera(String denumire,int nr_membrii_echipaj,int nr_restaurante,int nr_piscine){
        super(denumire,nr_membrii_echipaj);
        this.nr_restaurante=nr_restaurante;
        this.nr_piscine=nr_piscine;
    }
    public int getNr_restaurante() {
        return nr_restaurante;
    }
}
```

```

@Override
public String toString() {
    return super.toString() + ", " + nr_restaurante + ", " + nr_piscine;
}

class MainApp {
    public static void main(String[] args) {
        Vapor a = new Vapor("Pescarus", 10);
        System.out.println(a);

        VasDeCroaziera b = new VasDeCroaziera("Harmony of the Seas", 100, 10, 7);
        System.out.println(b);

        a = b;
        System.out.println(a.toString());

        b = (VasDeCroaziera)a;
        System.out.println(b.toString());
    }
}

```

```

Console × Markers Properties Servers
<terminated> MainApp (146) [Java Application] D:\...
Pescarus, 10
Harmony of the Seas, 100, 10, 7
Harmony of the Seas, 100, 10, 7
Harmony of the Seas, 100, 10, 7

```

- În cazul în care un obiect primește o referință la un obiect al subclasei, nu este necesară o conversie explicită prin *cast*

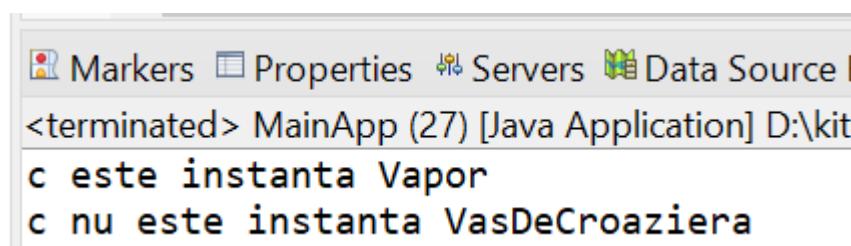
- Conversia explicită prin cast este necesară dacă un obiect primește o referință a superclasei sale
- Conversia prin operație de cast de la un obiect al superclasei la un obiect al subclasei este posibilă doar dacă în momentul execuției superclasa referă un obiect al subclasei.
- Dacă obiectul referit nu este de tipul corespunzător se va genera o excepție *ClassCastException*
- După ce un obiect al superclasei a obținut ca valoare o referință la un obiect al subclasei, în cazul în care în clasa de bază există o metoda redefinită de clasa derivată, atunci cea din clasa derivată va fi cea apelată
- *Utilizarea mecanismului de conversie pe bază de cast se poate face numai între o superclasa și subclasele sale (sau subclasele subclaselor sale)*
- În general, deoarece *Object* este rădăcina ierarhiei de clase se pot executa conversii precum în secvența următoare

```
Object o;  
Ceva c=new Ceva();  
  
o=c;  
...  
c=(Ceva) o;
```

- Pentru a asigura corectitudinea castului se recomandă verificarea apartenenței obiectului la clasa respectivă cu ajutorul operatorului *instanceof*

```
Vapor c=new Vapor("Pescarus",10);
if (c instanceof Vapor)
    System.out.println("c este instanta Vapor");
else
    System.out.println("c nu este instanta Vapor");

if (c instanceof VasDeCroaziera)
    System.out.println("c este instanta VasDeCroaziera");
else
    System.out.println("c nu este instanta VasDeCroaziera");
```



The screenshot shows the Eclipse IDE interface with the 'Markers' tab selected. Below the tabs, the status bar displays the message '<terminated> MainApp (27) [Java Application] D:\kitu'. The main workspace area contains two lines of text output from the console:

```
c este instanta Vapor
c nu este instanta VasDeCroaziera
```

- Operatorul *instanceof* de regulă este utilizat într-un *if* prin care se verifică dacă un obiect este o instanță a unei anumite clase. Dacă obiectul este o instanță a acelei clase, pentru a putea avea acces la metodele clasei, în varianta clasică de utilizare a lui *instanceof* se impune o conversie de tip (vezi primul if din exemplul de mai jos).
- În versiunile Java 12, 14 și 15 a fost introdus ca și o caracteristică cu previzualizare (preview feature) operatorul *pattern matching instanceof*, iar începând cu versiunea Java 16 (martie, 2021) acest operator a fost introdus ca și o caracteristică permanentă
- Utilizând operatorul *patter matching instanceof* se poate declara un obiect care va prelua caracteristicile obiectului verificat, obiect care poate fi utilizat în acel *if*. În acest fel nu mai este necesară operația de cast (vezi al doilea if din exemplul de mai jos).

```
Vapor d=new VasDeCroaziera("Seadream", 95, 5, 4);

if (d instanceof VasDeCroaziera) {
    System.out.println(d.getDenumire()
        + " are "+((VasDeCroaziera)d).getNr_restaurante()+" restaurante");
}

if (d instanceof VasDeCroaziera x) {
    System.out.println(x.getDenumire()
        + " are "+x.getNr_restaurante()+" restaurante");
}
```

## 2.8 Enumerări

- Enumerările se folosesc pentru variabile care au o listă finită de valori cunoscută la momentul compilării
- Tipul enumerare deși NU trebuie instantiat folosind *new*, are capabilități similare unei clase (poate avea constructor, metode, variabile membre, etc).
- Enumerările nu pot extinde alte clase și nici nu pot fi extinse (pentru că extind deja clasa *java.lang.Enum* și nu este permisă moștenirea multiplă) dar pot implementa interfețe
- Declararea unei enumerări se face cu ajutorul cuvântului cheie *enum*.
- Enumerarea trebuie să înceapă cu o listă de constante și apoi pot să urmeze metode, variabile sau constructori
- Potrivit convenției de nume din *Java* este important să se denumească constantele cu toate literele mari
- Fiecare constantă *enum* reprezintă un obiect care are atributele *public static final* și poate fi accesat utilizând numele enumerării
- Tipurile enumerare pot fi utilizate în *switch*

- Inclusiv funcția *main()* poate fi scrisă într-o enumerare
- Metoda *toString()* din clasa *Object* este redefinită în clasa *java.lang.Enum* astfel încât returnează numele constantei
- Alte metode importante din clasa *java.lang.Enum* sunt *values()*, *ordinal()* și *valueOf()*
- Metoda *values()* returnează un vector cu toate valorile pe care le poate lua o enumerare
- Metoda *ordinal()* returnează poziția pe care se află o anumită constantă în enumerare
- Metoda *valueOf()* returnează constanta corespunzătoare *string-ului* specificat ca și parametru de intrare dacă aceasta există (dacă nu există metoda aruncă excepția *java.lang.IllegalArgumentException*)
- Enumărările pot conține un constructor privat care se va executa separat pentru fiecare constantă
- Următoarele două exemple ilustrează cum se poate utiliza tipul enumerare

```
package capitolul2.enumerari1;
import java.util.Scanner;

enum Anotimp{
    PRIMAVARA,
    VARA,
    TOAMNA,
    IARNA
}
class MainApp {
    public static void main(String[] args) {
        Anotimp a=Anotimp.VARA;
        System.out.println(a);

        boolean ok=false;
        Scanner scanner=new Scanner(System.in);
        do {
            try {
                System.out.print("Introduceti anotimpul preferat:");
                String s=scanner.next();
                a=Anotimp.valueOf(s.toUpperCase());
                ok=true;
            }
            catch(IllegalArgumentException e) {
                System.out.println(e);
            }
        }while(!ok);
        scanner.close();
    }
}
```

```

String cuvant=switch(a) {
    case PRIMAVERA->"ghiocei";
    case VARA->"soare";
    case TOAMNA->"culoare";
    case IARNA->"zapada";
    default->throw new IllegalStateException();
};

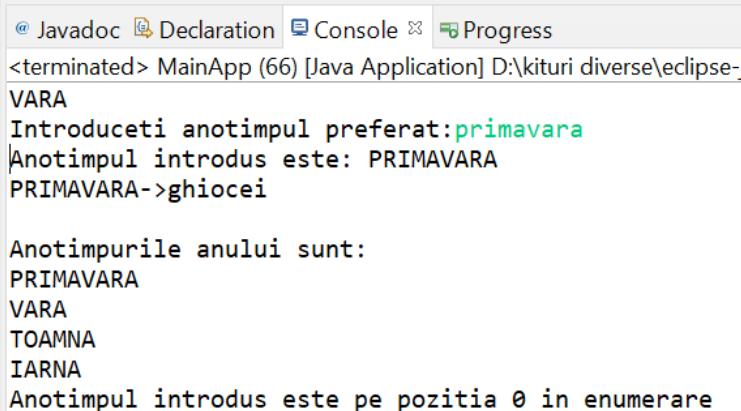
System.out.println("Anotimpul introdus este: "+a);
System.out.println(a+"->"+cuvant+"\n");

System.out.println("Anotimpurile anului sunt:");

Anotimp [] anotimpuri=Anotimp.values();
for (Anotimp b:anotimpuri) {
    System.out.println(b);
}

System.out.println("Anotimpul introdus este pe pozitia "+a.ordinal()+" in enumerare");
}
}

```


 The screenshot shows the Eclipse IDE interface. The top part contains the Java code provided in the slide. Below it is the 'Console' view, which displays the program's output. The output consists of several lines of text: 'VARA', 'Introduceti anotimpul preferat:primavara', 'Anotimpul introdus este: PRIMAVERA', 'PRIMAVERA->ghiocei', 'Anotimpurile anului sunt:', followed by the four seasons: 'PRIMAVERA', 'VARA', 'TOAMNA', and 'IARNA'. At the bottom of the console, it says 'Anotimpul introdus este pe pozitia 0 in enumerare'.

- În exemplul următor tipul enumerare are pe lângă constante și un constructor, o variabilă membră și o metodă. Constructorul unei enumerărări trebuie să aibă modificatorul de acces *private*

```
package capitolul2.enumerari2;
enum Anotimp{
    PRIMAVARA,
    VARA,
    TOAMNA,
    IARNA;

    private int a=7;

    private Anotimp() {
        System.out.println("Contractor apelat pentru fiecare constanta");
    }

    void metoda() {
        System.out.println("Metoda in enumerare. a="+a);
    }
}

class MainApp {
    public static void main(String[] args) {
        Anotimp a=Anotimp.VARA;
        System.out.println(a);
        a.metoda();
    }
}
```

@ Javadoc Declaration Console Progress  
<terminated> MainApp (67) [Java Application] D:\kituri dive  
Contractor apelat pentru fiecare constanta  
VARA  
Metoda in enumerare. a=7

## 2.9 Serializarea obiectelor

- Serializarea obiectelor presupune descompunerea acestora într-o înșiruire de octeți și salvarea lor într-un flux de date. Procesul opus, de recomponere a obiectelor din sirul de octeți se numește deserializare
- Pot să fie serializate obiectele oricărei clase care implementează interfața *Serilizable*
- O clasă care implementează interfața *Serilizable* poate avea variabile membre obiecte ale altor clase, care la rândul lor trebuie să implementeze interfața *Serilizable* pentru a putea fi salvate
- Câmpurile ale căror valoare nu se dorește să fie serializată trebuie să fie precedate de cuvântul cheie *transient*
- Serializarea / deserializarea obiectelor se poate realiza cu ajutorul claselor *ObjectOutputStream* / *ObjectInputStream* iar a tipurilor primitive cu ajutorul claselor *DataOutputStream* / *DataInputStream*
- Metoda *writeObject()* din clasa *ObjectOutputStream* realizează serializarea, iar metoda *readObject()* din clasa *ObjectInputStream()* realizează deserializarea

- Exemplul de mai jos serializează o colecție de obiecte de tip *List* în fișierul *pers.bin*
- Fișierul *Persoana.java*:

```
package capitolul2.serializare;

import java.io.Serializable;

class Persoana implements Serializable{
    private String nume;
    private transient int varsta;
    private Adresa adresa;

    public Persoana(String nume, int varsta,Adresa adresa) {
        this.nume = nume;
        this.varsta = varsta;
        this.adresa=adresa;
    }
    @Override
    public String toString() {
        return nume + " " + varsta+" "+adresa;
    }
}
```

- Fișierul *Adresa.java*:

```
package capitolul2.serializare;

import java.io.Serializable;

class Adresa implements Serializable{
    private String localitate;
    private String strada;
    private int nr;

    public Adresa(String localitate, String strada, int nr) {
        this.localitate = localitate;
        this.strada = strada;
        this.nr = nr;
    }
    @Override
    public String toString() {
        return localitate + ", " + strada + ", " + nr;
    }
}
```

- Fișierul *MainApp.java*:

```
package capitolul2.serializare;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

class MainApp {
    static void scrie(Object o, String fis) {
        try {
            FileOutputStream f = new FileOutputStream(fis);
            ObjectOutputStream oos = new ObjectOutputStream(f);
            oos.writeObject(o);
            oos.close();
            f.close();
        }
        catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

```

static Object citeste(String fis) {
    try {
        FileInputStream f = new FileInputStream(fis);
        ObjectInputStream ois = new ObjectInputStream(f);
        Object o=ois.readObject();
        ois.close();
        f.close();
        return o;
    }
    catch (IOException | ClassNotFoundException e) {
        e.printStackTrace();
    }
    return null;
}

public static void main(String[]args) {
    List<Persoana> persoane = new ArrayList<Persoana>();
    persoane.add(new Persoana("Ion", 20, new Adresa("Timisoara", "Parvan", 1)));
    persoane.add(new Persoana("Ana", 21, new Adresa("Timisoara", "Republicii", 1)));

    scrie(persoane, "pers.bin");//varsta nu a fost serializata, utilizare transient

    List<Persoana> q;
    q = (List<Persoana>) citeste("pers.bin");
    for (Persoana p : q)
        System.out.println(p);
}
}

```

- Câmpul varsta fiind precedat de cuvântul cheie transient nu va fi serializat, își va pierde valoarea
- Pentru ca obiectele clasei *Persoana* să poată să fie serializate este necesar ca aceasta să implementeze interfața *Serilizable*. Clasa *Persoana* are un câmp de tip *Adresa*. *Pentru ca acel câmp să poată fi serializat este necesar ca și clasa Adresa să implementeze interfața Serilizable. În caz contrar se va produce NotSerializableException*

## 2.10 Generarea documentației

- *javadoc* este generatorul de documentație al proiectelor și este inclus în JDK (Java Development Kit)
- *javadoc* generază documentația în format *html*, similar cu documentația Java API, aceasta putând fi urcată pe un server
- Documentația se introduce în proiect utilizând comentarii de documentație (doc comments) care sunt în formatul de mai jos. Acestea se amplasează imediat deasupra claselor, metodelor, constructorilor, interfețelor, etc pe care le documentează

```
/**  
 *  
 */
```

- În prima parte a comentariului de documentație se introduce o descriere generală iar în a doua parte se introduc tag-uri prin care se poate specifica cine este autorul secvenței documentate, versiunea acesteia, data dezvoltării, etc. În cazul documentării metodelor se pot introduce tag-uri prin care se precizează ce reprezintă parametrii de intrare, ce reprezintă valoarea returnată, respectiv în ce situații aruncă metoda anumite exceptii.

- Exemplu de clasă documentată, fișierul *Patrat.java*:

```
package capitolul12.documentatie;  
/**  
 * Patratul este un patrulater cu laturile egale si cu unghiurile drepte.  
 * @author student  
 * @version 1  
 * @since 2023  
 */  
  
public class Patrat {  
    private final int a;  
    /**  
     * Constructorul clasei Patrat  
     * @param a latura patratului  
     */  
    public Patrat(int a) {  
        super();  
        this.a = a;  
    }  
    /**  
     * Getter care da acces de citire a variabilei membre care contine dimensiunea  
     * laturii patratului.  
     * @return Latura patratului  
     */  
    public int getA() {  
        return a;  
    }  
}
```

```
/*
 * Calculeaza perimetrul patratului.
 * @return Returneaza perimetrul patratului obtinut inmultind
 * latura patratului cu patru.
 */
public int perimetru() {
    return 4*a;
}
/**Calculeaza aria patratului
 * @return Returneaza aria patratului, calculata facand produsul a doua laturi.
 */
public int aria() {
    return a*a;
}
/*
 * Diagonala patratului este ipotenuza unui triunghi dreptunghic isoscel ale carui
 * catete sunt egale cu latura patratului.
 * @return Dimensiunea diagonalei patratului calculata cu teorema lui Pitagora.
 */
public double diagonala() {
    return a*Math.sqrt(2);
}
}
```

- Clasa *MainApp* în fișierul *MainApp.java*:

```
package capitolul2.documentatie;

class MainApp {
    public static void main(String[] args) {
        Patrat p1=new Patrat(2);
        System.out.println("Patratul cu latura " + p1.getA()
            + " are perimetru " + p1.perimetru()
            + ", aria " + p1.aria()
            + " și diagonala " + String.format("%.2f", p1.diagonala()));
    }
}
```

- În exemplul precedent s-au utilizat tag-urile @author, @version, @since, @param și @return
- Documentația introdusă va fi afișată în IDE-uri precum *IntelliJ* sau *Eclipse* atunci când se vor declara și instanța obiecte de tip *Patrat* sau se vor apela metodele documentate
- Generarea documentației se poate face din linie de comandă (poate fi utilizat terminalul din *IntelliJ* sau din *Eclipse*) sau cu ajutorul opțiunilor de meniu pentru generare. În *IntelliJ* se alege opțiunea de meniu **Tools > Generate JavaDoc...**, iar în *Eclipse* se alege **Project > Generate Javadoc...**

- În linie de comandă documentația aferentă clasei *Patrat* poate fi generată printr-o comandă precum cea de mai jos

```
javadoc -d d:\doc src/capitolul2/documentatie/Patrat.java
```

- Comanda va genera documentația clasei *Patrat* în directorul *d:\doc*. Comanda a fost rulata fiind poziționată în directorul proiectului
- Documentația generată poate fi vizualizată în browser accesând fișierul ***index.html***

#### *Method Summary*

All Methods	Instance Methods	Concrete Methods
Modifier and Type	Method and Description	
int	<a href="#">aria()</a>	Calculeaza aria patratului
double	<a href="#">diagonala()</a>	Diagonala patratului este ipotenuza unui triunghi dreptunghic isoscel ale căruia catete sunt egale cu latura patratului.
int	<a href="#">getA()</a>	Getter care da acces de citire a variabilei membre care contine dimensiunea laturii patratului.
int	<a href="#">perimetru()</a>	Calculeaza perimetrul patratului.

## 2.11 Redefinirea metodelor

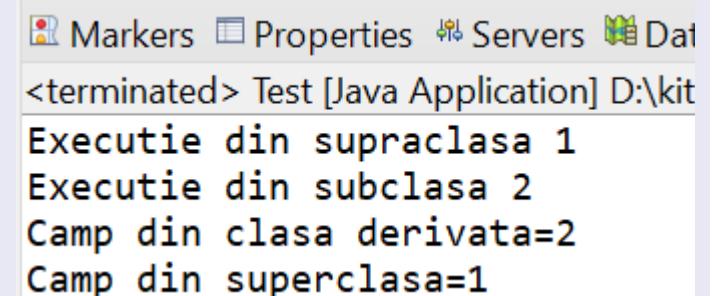
- dacă într-o clasă derivată se definește o metodă cu același nume și signatură cu o metodă din clasa de bază, spunem că metoda din clasa derivată o redefineste pe cea din clasa de bază (*overriding*)
- Din subclasă pot fi apelate ambele metode:
  - exemplu() – apelează metoda din clasa derivată
  - super.exemplu() –apelează metoda din clasa de bază

```
package capitolul2.redefinirea;
class ClasaDeBaza {
    int camp;
    public ClasaDeBaza(){
        camp=1;
    }
    void afisareCamp(){
        System.out.println("Executie din supraclasa "+camp);
    }
    final void fa_ceva() {
        System.out.println("Metoda finala, nu poate fi redefinita"
            +" in clasele deriveate");
    }
}
```

```

class ClasaDerivata extends ClasaDeBaza{
    int camp;
    void setCamp(int camp){
        this.camp=camp;
    }
    @Override
    void afisareCamp(){
        System.out.println("Executie din subclasa "+camp);
    }
    void afis(){
        super.afisareCamp();
        afisareCamp();
        System.out.println("Camp din clasa derivata="+this.camp
                           +"\nCamp din superclasa="+super.camp);
    }
    //void fa_ceva() {
    //    System.out.println("Eroare de compilare, nu este permisa redefinirea"
    //                      +" metodelor finale");
    //}
}
class Test{
    public static void main(String args[]){
        ClasaDerivata f=new ClasaDerivata();
        f.setCamp(2);
        f.afis();
    }
}

```


 Markers Properties Servers Data  
 <terminated> Test [Java Application] D:\kit  
 Executie din supraclasa 1  
 Executie din subclasa 2  
 Camp din clasa derivata=2  
 Camp din superclasa=1

- În exemplul precedent metoda `void afisareCamp()` din *ClasaDerivata* redefineste metoda `void afisareCamp()` din *ClasaDeBaza*
- Utilizarea adnotatiei `@Override` nu este obligatorie, dar asigura ca se face redefinire. Daca se foloseste adnotatia `@Override` si nu se face redefinire se produce eroare de compilare
- Metoda `fa_ceva()` din *ClasaDeBaza* este metoda finala, deci nu poate fi redefinita. Incercarea de a crea o metoda in *ClasaDerivata* cu aceeasi signatura produce eroare de compilare
- Din clasa derivata pot sa fie accesate cele 2 metode `afisareCamp()` (se foloseste `super` pentru a o accesa pe cea din clasa de baza).

## 2.12 Clase generice

- O clasă generică este o clasă care are parametri generici, enumerați între parantezele unghiulare care însotesc definiția clasei.
- Valorile din interiorul parantezelor unghiulare reprezintă tipuri de date care vor fi stabilite când se declară un obiect de tipul clasei respective.
- Operatorul `<>` este cunoscut sub denumirea operatorul *diamond*

```
package capitolul2.generice1;

class Persoana<T,S>{
    T nume;
    S varsta;
    public Persoana(T nume, S varsta) {
        this.nume = nume;
        this.varsta = varsta;
    }
    @Override
    public String toString() {
        return nume + ", " + varsta;
    }
}
```

```
class MainApp {  
    public static void main(String[] args) {  
        String nume1="Popescu Ion";  
        int varsta1=23;  
        Persoana<String, Integer> p1=new Persoana<String, Integer>(nume1, varsta1);  
        System.out.println(p1);  
  
        StringBuilder nume2=new StringBuilder("Popescu Ion");  
        byte varsta2=23;  
        Persoana<StringBuilder,Byte> p2=new Persoana<StringBuilder,Byte>(nume2, varsta2);  
        System.out.println(p2);  
    }  
}
```

## 2.13 Metode generice

- Metodele generice sunt metode care pot fi apelate cu argumente de diferite tipuri
- Metodele generice au operatorul *diamond* înaintea tipului returnat. În interiorul acestuia se enumera tipurile generice separate prin virgule

```
package capitolul2.generice2;

public class MainApp {
    public static <T> void afiseazaVector(T[] vector) {
        for(T v : vector) {
            System.out.print(v+" ");
        }
        System.out.println();
    }

    public static void main(String[] args) {
        Integer[] v1 = { 1, 2, 3, 4, 5 };
        Character [] c={'t','e','s','t'};

        afiseazaVector(v1);
        afiseazaVector(c);
    }
}
```

## 2.14 Clase imbricate

### 2.14.1 Noțiuni generale

- O clasă poate fi declarată în interiorul altei clase. Scopul este de a grupa clasele într-un singur loc, obținând astfel un cod mai ușor de citit și întreținut
- Clasa interioară poate fi de mai multe tipuri:
  - Clasă interioară non-statică
    - Clasa interioară membră – se declară direct în clasa exterioară (în afara oricărei metode) și are acces la variabilele membre ale acesteia chiar private fiind.
    - Clasa interioară locală – se declara într-un bloc de cod care de obicei este o metodă
    - Clasa interioară anonimă – permite declararea unei clase și instanțierea unui obiect în același timp. De obicei se plasează în alte metode, dar expresia clasei anonte poate fi plasata și direct în clasa exterioară
  - Clasă interioara statică – se declara direct în clasa exterioară (în afara oricărei metode) și are acces la membri statici ai clasei exterioare, chiar privați fiind

## 2.14.2 Clasă interioară membră

- Clasa interioară membră se declară direct în clasa exterioară (în afara oricărei metode) și are acces la variabilele membre ale acesteia chiar private fiind
- Clasa interioară membru poate fi însotită de modificatorii de acces *private*, *protected*, *public*
- Într-o a treia clasă se poate declara obiect de tipul clasei interioare (dacă modificatorii de acces asigura vizibilitate către aceasta) cu ajutorul unui obiect al clasei exterioare
- Fișierul *ClasaExterioara.java*

```
package capitolul2.clase_imbricate1;

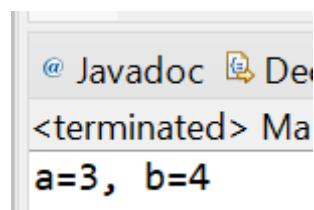
public class ClasaExterioara {//Outer class
    private int a=3;

    class ClasaInterioara{//Inner class
        private int b=4;
        public void afisare(){
            System.out.println("a="+a+", b="+b);
        }
    }
}
```

- Fișierul *MainApp.java*

```
package clase_imbricate;

public class MainApp {
    public static void main(String[] args) {
        ClasaExterioara o1=new ClasaExterioara();
        ClasaExterioara.ClasaInterioara o2=o1.new ClasaInterioara();
        o2.afisare();
    }
}
```



## 2.14.3 Clasă interioară locală

- Se declară într-un bloc de cod care de obicei este o metodă
- Nu poate avea modificatori de acces și este vizibilă doar în acel bloc de cod
- Fișierul *ClasaExterioara.java*

```
package capitolul2.clase_imbricate2;

class ClasaExterioara {//Outer class
    private int a=3;

    public void metoda() {
        final int b=4;
        class ClasaInterioaraLocala{//Local Inner class
            public void afisare(){
                System.out.println("a="+a+", b="+b);
            }
        }

        ClasaInterioaraLocala o=new ClasaInterioaraLocala();
        o.afisare();
    }
}
```

```
class MainApp {  
    public static void main(String[] args) {  
        ClasaExterioara o1=new ClasaExterioara();  
        o1.metoda();  
    }  
}
```

## 2.14.4 Clasă interioară anonimă

- Clasele anonte permit:
  - scrierea comasată a codului
  - declararea unei clase și instanțierea ei în același timp
- O clasă anonimă este o expresie. Sintaxa unei clase anonte este asemănătoare cu apelul unui constructor însotit de definiția unei clase
- Expresia unei clase anonte conține următoarele:
  - Operatorul *new*
  - Numele unei interfețe de implementat sau a unei clase de extins
  - Parantezele care contin argumentele unui constructor
  - Corpul declarării unei clase
- Fișierul *Ordonare.java*

```
package capitolul2.clase_imbricate3;

interface Ordonare {
    public void metoda();
}
```

- Fișierul *MainApp.java*:

```
package capitolul2.clase_imbricate3;

public class MainApp {
    public static void main(String []args){
        Ordonare t=new Ordonare(){
            public void metoda(){
                System.out.println("Ordoneaza prin HeapSort");
            }
        };
        t.metoda();
    }
}
```

## 2.14.5 Clase interioare statice

- Clasele interioare pot fi statice. În acest caz instantierea unui obiect al clasei interioare nu mai necesită un obiect al clasei exterioare (vezi exemplul următor). Din clasa interioară pot fi accesati membrii statici ai clasei exterioare

```
package capitolul2.clase_imbricate4;

class ClasaExterioara {//Outer class
    private static int a=3;

    static class ClasaInterioaraStatica{
        private int b=4;
        public void afisare(){
            System.out.println("a="+a+", b="+b);
        }
    }
}

class MainApp {
    public static void main(String[] args) {
        ClasaExterioara.ClasaInterioaraStatica o=new ClasaExterioara.ClasaInterioaraStatica();
        o.afisare();
    }
}
```

## 2.14.6 Facilități pentru operarea cu clase imbricate (Java 11)

- În Java 11 (versiune Java cu suport pe termen lung, lansată în septembrie 2018) a fost introdus conceptul de cuib (nest) care cuprinde clasele imbricate și metode care ajută la operarea asupra unor clase imbricate, precum:
  - `getNestHost()` – returnează clasa exterioară, cea care găzduiește clasele interioare
  - `getNestMembers()` – returnează clasa exterioară și clasele interioare
  - `isNestmateOf()` – verifică dacă clasa pentru care se apelează se găsește în același cuib ca și clasa specificată ca și parametru de intrare
- Exemplul următor arată cum pot fi utilizate aceste metode (a fost importată `ClasaExterioara` din pachetul `capitolul2.clase_imbricate1`):

```
package capitolul2.clase_imbricate5;

import java.util.List;
import capitolul2.clase_imbricate1.ClasaExterioara;

public class MainApp {
    public static void main(String[] args) {
        Class<ClasaExterioara.ClasaInterioara> ci=ClasaExterioara.ClasaInterioara.class;
        Class<ClasaExterioara> ce=ClasaExterioara.class;

        System.out.println(ci.getNestHost());
        System.out.println(ce.getNestHost());
```

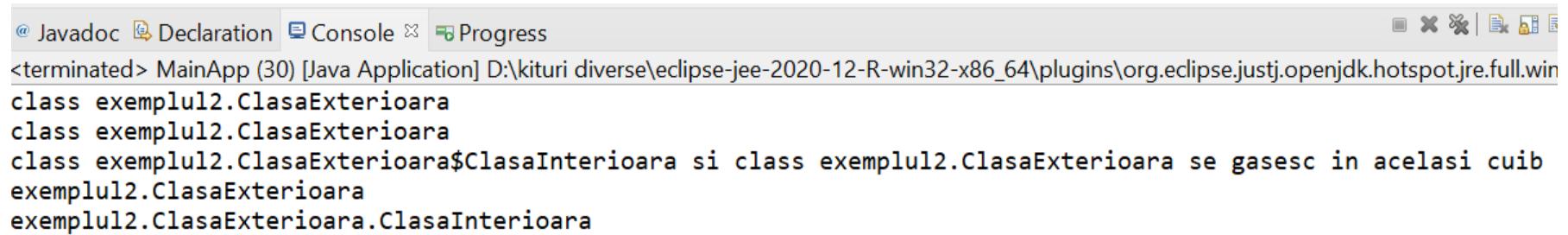
```

        if(ci.isNestmateOf(ce)) {
            System.out.println(ci + " si "+ce+" se gasesc in acelasi cuib");
        }
        else {
            System.out.println(ci + " si "+ce+" se gasesc in cuib-uri diferite");
        }

        List.of(ci.getNestMembers())
            .stream()
            .map(Class::getCanonicalName)
            .forEach(System.out::println);
    }
}

```

- Ieșirea programului precedent este următoarea:



The screenshot shows the Eclipse IDE interface with the 'Console' tab selected. The console window displays the following output:

```

@ Javadoc Declaration Console Progress
<terminated> MainApp (30) [Java Application] D:\kituri diverse\eclipse-jee-2020-12-R-win32-x86_64\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64\lib\jre\bin\java.exe
class exemplul2.ClasaExterioara
class exemplul2.ClasaExterioara
class exemplul2.ClasaExterioara$ClasaInterioara si class exemplul2.ClasaExterioara se gasesc in acelasi cuib
exemplul2.ClasaExterioara
exemplul2.ClasaExterioara.ClasaInterioara

```

- Exemplul precedent conține două clase imbricate. Din clasa interioară se pot accesa inclusiv membrii privați ai clasei exterioare

- Metoda `getNestHost()` returnează numele clasei exterioare, indiferent pentru care clasă se apelează
- Metoda `isNestmate()` permite verificarea dacă cele două clase se găsesc în același cuib
- În ultimul exemplu s-a creat o listă care conține toți membri din cuib. Toate elementele din `stream-ul` asociat listei au fost mapate cu ajutorul metodei `getCanonicalName`, care returnează numele clasei împreună cu pachetul în care se găsește. Apoi `stream-ul` a fost afișat

## 2.15 Clase abstracte

- O clasă abstractă este o supraclasă pentru o ierarhie de clase
- O clasă abstractă conține câmpuri și metode normale (pentru care se specifică o implementare dar și modele de metode (metode abstracte) care urmează să fie implementate în mod obligatoriu de clasele normale care extind clasa abstractă)
- O clasa normală poate extinde o clasă abstractă doar dacă implementează metodele abstracte. Altfel clasa în cauză va conține metode abstracte care vin pe relația de moștenire și trebuie făcută abstractă
- O metoda abstractă nu are implementare.

```
abstract class ClasaAbstracta{  
    abstract void metoda();  
}
```

- O metodă finală nu mai poate să fie redefinită
- O metodă cu atributul abstract trebuie să fie redefinită pentru a fi utilizată printr-o metodă care nu are același atribut

- O clasă abstractă poate să fie extinsă de o altă clasă abstractă sau de o clasă normală (dacă aceasta implementează metodele abstracte)

```
class ClasaNormala extends ClasaAbstracta{  
    @Override  
    void metoda() {  
        System.out.println("S-a executat metoda");  
    }  
}
```

- O metodă abstractă se află în mod obligatoriu într-o clasă abstractă sau într-o interfață
- Clasele normale nu pot contine metode abstracte
- Deoarece metodele abstracte reprezintă numai modele de metode nu se pot face instanțieri de obiecte pentru clasele abstracte
- Dacă prin extinderea clasei *ClasaAbstracta* se definește o clasă care nu mai este abstractă se poate face și o instanțiere

```
//ClasaAbstracta o=new ClasaAbstracta(); //eroare de compilare  
ClasaAbstracta o=new ClasaNormala();
```

## 2.16 Interfețe

### 2.16.1. Noțiuni generale

- Interfețele sunt un mecanism de abstractizare
- O interfață este un model al unei clase.
- O interfață este un tip abstract utilizat pentru a specifica comportamentul unei clase (care implementează interfața)
- Dacă o clasă care implementează o interfață nu specifică o implementare tuturor metodelor abstrakte atunci clasa trebuie făcută abstractă
- O clasă care implementează interfața trebuie să specifice codul corespunzător metodelor din interfață dar poate să declare variabile și metode care nu apar în interfață

- O interfață poate conține:
  - Constante (chiar dacă se fac declarații de variabile compilatorul le atașează atributelor *public static final*, transformându-le în acest fel în constante)
  - Metode abstracte
  - Metode implicite (începând cu Java 8) – metode cu cod care au atributul *default*
  - Metode statice (începând cu Java 8)
  - Metode private în interfețe (începând cu Java 9) – apelate din metodele implicite
- Variabilele de tip interfață pot fi utilizate ca argumente ale metodelor și pot fi obținute ca rezultat, se pot crea ierarhii de interfețe similare ierarhiilor de clase
- *Variabile referință la o interfață* – valoarea unei astfel de variabile poate fi o referință la un obiect al unei clase care implementează interfața sau a unei clase care extinde o clasă ce implementează interfața
- Metodele care apar într-o interfață sunt în mod implicit publice
- O interfață poate fi extensia unei alte interfețe, adică poate să adauge noi modele de metode la o interfață care există
- O interfață se declară cu ajutorul cuvântului cheie *interface*

- Implementarea uneia sau mai multor interfețe se face cu ajutorul cuvântului cheie *implements*

```
package capitolul2.interfete1;

interface Figura{
    void deseneaza();
}

class Triunghi implements Figura{
    @Override
    public void deseneaza() {
        System.out.println("Deseneaza un triunghi!");
    }
}

class Cerc implements Figura{
    @Override
    public void deseneaza() {
        System.out.println("Deseneaza un cerc!");
    }
}
```

```
class MainApp {  
    public static void deseneaza(Figura f) {  
        f.deseneaza();  
    }  
    public static void main(String[] args) {  
        deseneaza(new Triunghi());  
        deseneaza(new Cerc());  
    }  
}
```

- În exemplul precedent s-a creat interfața *Figura*, care conține o metodă abstractă *deseneaza()* (atributul *abstract* poate fi scris explicit sau dedus)
- Pornind de la aceasta interfață se poate realiza o familie de clase care implementează interfața *Figura*, o familie de clase care știu să deseneze diferite figuri geometrice.
- În programul principal s-a creat metoda statică *desenează* (era necesar să fie statică pentru că este apelată din *main* care este o funcție statică), metodă care are parametrul de intrare un obiect de tip *Figura*. Metoda apelează metoda *deseneaza()* a aceluui obiect. La apel se pot transmite instanțe ale unei clase care implementează direct sau indirect interfața. Direct este exemplificat în exemplu, iar indirect ar fi fost dacă se transmitea o instanță a unei clase care extinde o clasă ce implementează interfața *Figura*.

- În funcție de instanță transmisă la apel se apelează o anumită metodă de desenare
- Exemplul utilizează polimorfismul, abilitatea unui obiect de a lua mai multe forme
- API-ul Java oferă interfața *Comparable*, care dispune de metoda abstractă *compareTo()* prin care se poate specifica cum să se compare un obiect al unei clase cu un altul, care a fost dat ca și parametru. Modul concret în care se face comparația se specifică la nivelul claselor care implementează această interfață

```
package capitolul2.interfete2;
class Persoana implements Comparable<Persoana>{
    private String nume;
    private Integer varsta;

    public Persoana(String nume, Integer varsta) {
        this.nume = nume;
        this.varsta = varsta;
    }
    public String getNume() {
        return nume;
    }
    @Override
    public String toString() {
        return nume + ", " + varsta;
    }
    @Override
    public int compareTo(Persoana o) {
        return this.varsta.compareTo(o.varsta);
    }
}
```

```

class MainApp {
    public static void main(String[] args) {
        Persoana p1=new Persoana("Ionel",20);
        Persoana p2=new Persoana("Ana",23);

        //switch cu expresii caracteristica permanenta din Java 14
        String mesaj=switch(p1.compareTo(p2)) {
            case 0->p1.getNume()+" are aceeasi varsta ca si "+p2.getNume();
            case 1->p1.getNume()+" este mai in varsta decat "+p2.getNume();
            case -1->p1.getNume()+" este mai tanar decat "+p2.getNume();
            default->"valoare eroanata";
        };
        System.out.println(mesaj);
    }
}

```

- Notația generică <Persoana> determină tipul parametrului din metoda de comparare. Fără aceasta notație parametrul metodei *compareTo* ar fi fost de tip *Object* și era necesară operația de cast pentru a accesa caracteristicile din clasa derivată

## 2.16.2 Metode **implicite**, metode **staticice** (Java 8) și metode **private** în interfețe (Java 9)

- Posibilitatea de a adăuga metode **implicite** în interfețe (metode cu cod care au atributul *default*) a fost introdusă în Java 8. La fel și posibilitatea de a adăuga metode **staticice** în interfețe
- Posibilitatea de a adăuga metode **private** în interfețe a fost introdusă în Java 9
- Metodele **implicite** se definesc cu ajutorul cuvântului cheie ***default***. Se utilizează când se dorește să se adauge o nouă funcționalitate unei interfețe și în același timp să se păstreze compatibilitatea cu clasele care deja implementează interfața
- Exemplul următor ilustrează cum se pot utiliza aceste metode în interfețe
- Interfața *Figura* din exemplul precedent, a fost extinsă cu 2 metode **implicite**, *culoare\_contur()* și *culare\_umplere()*. Clasele care implementează această interfață nu trebuie să își schimbe structura (să implementeze metodele **implicite**, dar pot face aceasta dacă e necesar). Clasa *Triunghi* redefineste metoda implicită *culoare\_contur()*, iar clasa *Cerc* folosește metoda implicită *culoare\_contur()* din interfață. Ambele clase utilizează metoda *culoare\_umplere()* din interfață

```
package capitolul2.interfete3;

interface Figura{
    int grosimea_liniei_de_desenare=10;
    abstract void deseneaza();
    static void metoda_statica () {//metode statice in interfete incepand cu Java 8
        System.out.println("Exemplu metoda statica.");
    }
    private String culoare() {//metode private incepand cu Java 9
        return "rosie";
    }
    default String culoare_umplere() { //metode implicite incepand cu Java 8
        return " Culoarea de umplere "+culoare()+"." ;
    }
    default String culoare_contur() {
        return " Culoarea conturului "+culoare()+"." ;
    }
}
class Triunghi implements Figura{
    @Override
    public String culoare_contur() {
        return "Culoarea conturului neagra.";
    }
    @Override
    public void deseneaza() {
        System.out.println("Deseneaza un triunghi! "+culoare_contur()+culoare_umplere());
    }
}
```

```

class Cerc implements Figura{
    @Override
    public void deseneaza() {
        System.out.println("Deseneaza un cerc!" + culoare_contur() + culoare_umplere());
    }
}

public class MainApp {
    public static void deseneaza(Figura f) {
        f.deseneaza();
    }
    public static void main(String[] args) {
        deseneaza(new Triunghi());
        deseneaza(new Cerc());
        Figura.metoda_statica();
        System.out.println("Grosimea liniei de desenare este:"
                +Figura.grosimea_liniei_de_desenare);

        //eroare de compilare daca seincearca modificarea unei constante (public static final)
        //Figura.grosimea_liniei_de_desenare=4;
    }
}

```

Deseneaza un triunghi! Culoarea conturului neagra. Culoarea de umplere rosie.  
 Deseneaza un cerc! Culoarea conturului rosie. Culoarea de umplere rosie.  
 Exemplu metoda statica.  
 Grosimea liniei de desenare este: 10

- Interfața *Figura* din pachetul *capitolul2.interfete1* a fost extinsă cu diferite elemente prezentate în continuare, ajungând la forma din pachetul *capitolul2.interfete3*
- Interfața *Figura* a fost extinsă cu 2 metode implicite, *culoare\_contur()* și *culare\_umplere()*. Clasele care implementează această interfață nu trebuie să își schimbe structura (să implementeze metodele隐式的, dar pot face aceasta dacă este necesar). Clasa *Triunghi* redefineste metoda implicită *culoare\_contur()*, iar clasa *Cerc* folosește metoda implicită *culoare\_contur()* din interfață. Ambele clase utilizează metoda implicită *culoare\_umplere()* din interfață.
- Cele 2 metode implicite din interfață apeleză metoda privată a interfeței numită *culoare()*, metodele private pot să fie introduse în interfețe începând cu Java 9
- Interfața *Figura* a fost extinsă și cu o metodă statică, care se apeleză cu *Nume\_interfata.nume\_metoda\_statica*, în programul principal
- De asemenea interfața *Figura* a fost extinsă cu constantă întreagă numită *grosimea\_liniei\_de\_desenare*. Cu toate că declarația acesteia este similară cu unei variabile, ea nu este o variabilă pentru că, compilatorul îi adaugă atributele *public static final*. Fiind statică, poate fi accesată pentru a fi afișată în programul principal cu *Nume\_interfata.nume\_constanta*, iar încercarea de a-i modifica valoarea eșuează cu mesajul ***The final field Figura.grosimea\_liniei\_de\_desenare cannot be assigned***, demonstrând astfel că este vorba despre o constantă

## 2.16.3 Interfețe funcționale (Java 8)

- O interfață este funcțională dacă are o singură metodă abstractă. Metode implicate, statice și private pot să fie oricâte
- Java 8 introduce adnotarea `@FunctionalInterface` și pachetul `java.util.functions` care conține un număr important de interfețe funcționale.
- Amplasarea adnotării `@FunctionalInterface` deasupra unei interfețe asigură că acea interfață este într-adevăr una funcțională (dacă nu este funcțională se produce eroare de compilare). Interfețele se află în ierarhii de interfețe, deci pot avea metode abstracte moștenite de la super interfețe
- Interfețele funcționale de obicei sunt implementate prin expresii *Lamda*, de asemenea introduse în Java 8, care vor fi discutate în cadrul unui subcapitolul următor
- În exemplul următor se consideră clasa *Persoana* cu variabilele membre *nume* și *varsta*. Datele mai multor persoane vor fi adăugate unei colecții de obiecte de tip *List* și apoi vor fi afișate filtrat. Se construiește și se utilizează interfața funcțională *Filtru* cu o metodă abstractă, numită *test()* și se utilizează interfața `java.util.functions.Predicate`, din API-ul Java 8, care de asemenea are o metodă pentru testarea unei condiții

```
package capitolul2.interfete_functionale;

import java.util.ArrayList;
import java.util.List;
import java.util.function.Predicate;

@FunctionalInterface
interface Filtru<T>{
    public boolean test(T p);
}

class Persoana{
    private String nume;
    private int varsta;
    public Persoana(String nume, int varsta) {
        this.nume = nume;
        this.varsta = varsta;
    }
    public String getNume() { return nume; }
    public int getVarsta() { return varsta; }
    @Override
    public String toString() {
        return "Persoana [nume=" + nume + ", varsta=" + varsta + "]";
    }
}
```

```

class MainApp {
    static void afisare_filtrata1(List<Persoana> pers, Filtru<Persoana> f) {
        for(Persoana p:pers)
            if(f.test(p))
                System.out.println(p);
    }
    static void afisare_filtrata2(List<Persoana> pers, Predicate<Persoana> f) {
        for(Persoana p:pers)
            if(f.test(p))
                System.out.println(p);
    }

    public static void main(String[] args) {
        List<Persoana> lista=new ArrayList<Persoana>();

        lista.add(new Persoana("Vladut",23));
        lista.add(new Persoana("Oana",19));
        lista.add(new Persoana("Iulia",22));
        afisare_filtrata1(lista, new Filtru<Persoana>() {
            @Override
            public boolean test(Persoana p) {
                return p.getVarsta()<20;
            }
        });
    }
}

```

```

afisare_filtrata2(Lista, new Predicate<Persoana>() {
    @Override
    public boolean test(Persoana p) {
        return p.getVarsta()<20;
    }
});

afisare_filtrata2(Lista, p->p.getVarsta()<20); //cu expresie Lambda
}
}

```

- *Filtru* este o interfață funcțională și generică, tipul parametrului de intrare al metodei abstracte *test()* va fi stabilit de fiecare clasă care implementează interfața
- Implementarea interfeței *Filtru* se face prin expresia unei clase anonime, împreună cu instantierea unui obiect și transmiterea acestuia ca și parametru către metoda *afisare\_filtrata1()*
- Metoda *afisare\_filtrata2()*, realizează același lucru ca și metoda *afisare\_filtrata1()* (și la apel primește un obiect similar), doar că utilizează interfața funcțională *Predicate*, din API-ul Java. Interfața funcțională *Filtru* a fost creată pentru a exemplifica cum se poate crea o interfață funcțională, dar de foarte multe ori se găsesc interfețele funcționale potrivite în pachetul *java.util.functions* și se recomandă utilizarea acestora în locul scrierii noile și similare, pentru a avea un cod mai concis

- Ultima afişare filtrată foloseşte o expresie *Lambda*, este cea mai concisă, expresiile *Lambda* vor fi detaliate în subcapitolul următor
- O mică parte din interfeţele funcţionale ale pachetului *java.util.functions sunt următoarele*:
- **Predicate<T>** reprezintă o operaţie care acceptă un singur parametru de intrare şi returnează un *boolean*
- **Consumer<T>** reprezintă o operaţie care acceptă un singur parametru de intrare si nu produce nici un rezultat
- **BiConsumer<T,U>** are o funcţie care acceptă două argumente şi nu returnează nimic
- **Function<T,R>** reprezintă o funcţie care are un parametru de intrare si produce un rezultat
- **BiFunction<T,U,R>** are o funcţie care are doi parametri de intrare şi produce un rezultat
- **UnaryOperator<T>** reprezintă un o operaţie cu un singur operand care produce un rezultat de acelaşi tip ca operandul
- **BinaryOperator<T>** are o funcţie care are doi parametri de intrare de acelaşi tip si produce un rezultat de acelaşi tip ca şi parametrii de intrare
- etc

## 2.17 Expresii *Lambda*

- Expresiile *Lambda* au fost introduse în **Java 8**
- O expresie *Lambda* este caracterizată prin următoarea sintaxă:

```
parametri -> corpul expresiei
```

- Expresiile *Lambda* sunt utilizate în principal pentru a defini implementarea inline (printr-o linie de cod) a unei interfețe cu o singură metodă abstractă (interfață funcțională)
- Expresia *Lambda* elimină necesitatea unei clase anonime și oferă o capacitate de programare funcțională foarte simplă, dar puternică pentru Java.
- Caracteristicile expresiilor *Lambda* sunt următoarele:
  - Nu este necesar să se declare tipul unui parametru. Compilatorul poate să deducă tipul lui
  - Parantezele rotunde sunt necesare doar pentru cel puțin doi parametri. Pentru un parametru sunt opționale
  - În corpul expresiilor sunt necesare paranteze accolade doar dacă acestea conțin două sau mai multe instrucțiuni
  - Cuvântul *return* este optional – compilatorul automat returnează valoarea dacă corpul are o singură expresie care să returneze valoarea

- Fișierul *Persoana.java* are conținutul de mai jos:

```
package capitolul2.lambda;

class Persoana{
    private String nume;
    private int varsta;
    public Persoana(String nume, int varsta) {
        this.nume = nume;
        this.varsta = varsta;
    }
    public String getNume() {
        return nume;
    }
    public int getVarsta() {
        return varsta;
    }
    @Override
    public String toString() {
        return nume + " " + varsta;
    }
}
```

- Fisierul *MainApp1.java* are conținutul de mai jos

```
package capitolul2.lambda;

import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;

class MainApp1 {
    public static void main(String[] args) {
        List<Persoana> pers=new ArrayList<Persoana>();
        pers.add(new Persoana("Maria",23));
        pers.add(new Persoana("Ana",24));
        pers.add(new Persoana("Oana",22));

        Collections.shuffle(pers);
        System.out.println("Colectia aleatoare: "+pers);

        Collections.sort(pers, new Comparator<Persoana>() {
            @Override
            public int compare(Persoana o1, Persoana o2) {
                return o1.getNume().compareToIgnoreCase(o2.getNume());
            }
        });
        System.out.println("Colectia ordonata dupa nume in Java 7 style: "+pers);
```

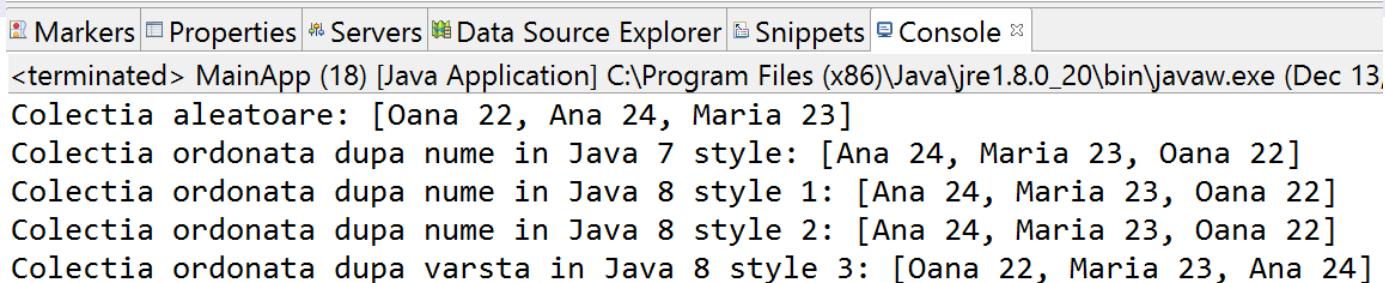
```

Collections.shuffle(pers);
Collections.sort(pers,(Persoana a,Persoana b)->a.getNume().compareToIgnoreCase(b.getNume()));
System.out.println("Colectia ordonata dupa nume in Java 7 style: "+pers);

Collections.shuffle(pers);
Collections.sort(pers,(a,b)->{
    return a.getNume().compareToIgnoreCase(b.getNume());
});
System.out.println("Colectia ordonata dupa nume in Java 8 style 1: "+pers);

Collections.shuffle(pers);
Collections.sort(pers,(Persoana a,Persoana b)->{
    if (a.getVarsta()<b.getVarsta()) return -1;
    else
        if(a.getVarsta()>b.getVarsta()) return 1;
        else return 0;
});
System.out.println("Colectia ordonata dupa varsta in Java 8 style 3: "+pers);
}
}

```



The screenshot shows the Eclipse IDE interface with the 'Console' tab selected. The output window displays the results of the three sorting operations:

- Colectia aleatoare: [Oana 22, Ana 24, Maria 23]
- Colectia ordonata dupa nume in Java 7 style: [Ana 24, Maria 23, Oana 22]
- Colectia ordonata dupa nume in Java 8 style 1: [Ana 24, Maria 23, Oana 22]
- Colectia ordonata dupa nume in Java 8 style 2: [Ana 24, Maria 23, Oana 22]
- Colectia ordonata dupa varsta in Java 8 style 3: [Oana 22, Maria 23, Ana 24]

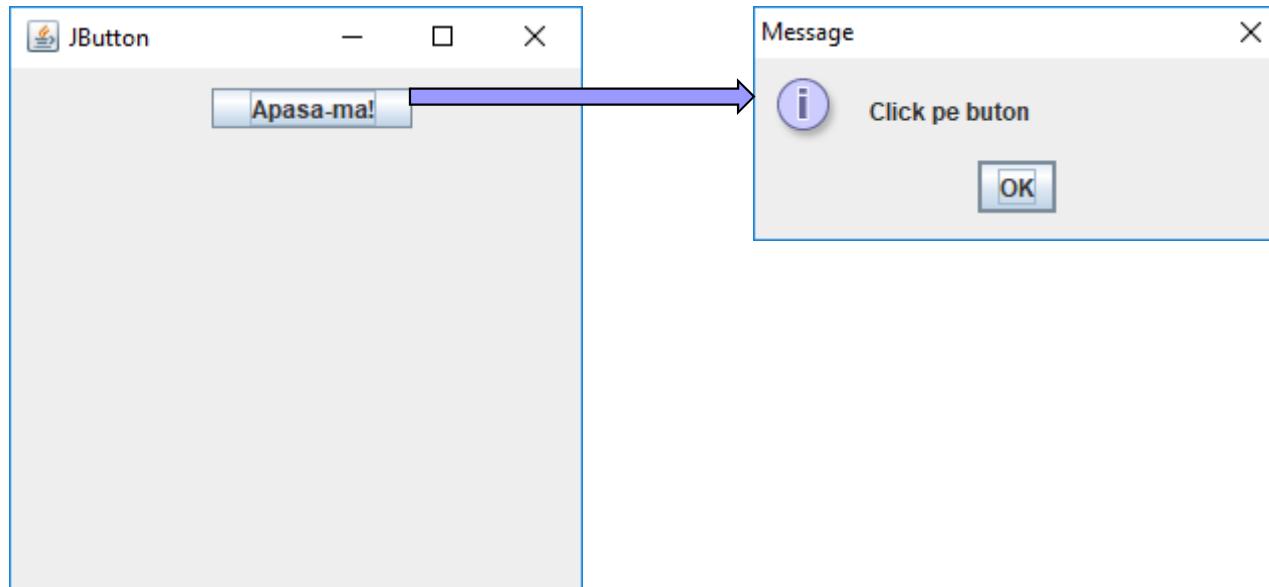
- În Java 7 ordonarea unei colecții se face cu ajutorul unei clase care implementează interfața de comparare
- În Java 8 această problemă poate fi rezolvată cu ajutorul notațiilor *Lambda*
- Parametrii care apar în expresia Lambda sunt parametrii metodei de comparare a interfeței. Tipul acestor parametrii poate să fie specificat (Persoana a, Persoana b) sau nu (a,b) (vezi slide-ul precedent).
- Corpul expresiei *Lambda* conține codul din metoda de comparare, care este automat returnat
- În cazul în care corpul expresiei Lambda conține mai multe linii de cod, este necesar să se utilizeze paranteze accolade și să se returneze prin utilizarea lui *return* valorile dorite (vezi exemplul de ordonare după vârsta din slideul precedent)

```
package capitolul2.lambda;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JOptionPane;

class MainApp2 {
    public static void main(String[] args) {
        JFrame myFrame = new JFrame("JButton");
        myFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        myFrame.setSize(300, 300);
        myFrame.getContentPane().setLayout(null);
```

```
JButton myButton=new JButton("Apasa-ma!");
myButton.setBounds(100, 10, 100, 20);
myButton.addActionListener(e->JOptionPane.showMessageDialog(null,"Click pe buton"));
myFrame.add(myButton);
myFrame.setVisible(true);
}
}
```

- Exemplul de mai sus ilustrează cum se poate adăuga un listener pe un buton în stilul Java 8, cu ajutorul expresiilor Lambda



## 2.18 Referințe de metode

- O referință de metodă este o construcție Java 8 care poate fi folosită pentru a referi o metodă fără a o apela
- O referință de metodă poate fi identificată cu ajutorul notatiei :: care separă o clasă sau un obiect de numele unei metode
- Obținerea unei referințe de constructor se poate realiza în felul următor:

```
String::new
```

- Obținerea unei referințe către o metodă statică:

```
String::valueOf
```

- Obținerea unei referințe către o metodă legată de un obiect:

```
str::toString
```

- Obținerea unei referințe către o metodă nelegată de un obiect:

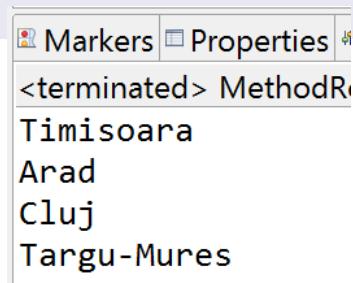
```
String::toString
```

- Fișierul *MainApp.java* are conținutul de mai jos:

```
package capitolul2.referinte_metode;

import java.util.List;

class MainApp {
    public static void main(String args[]) {
        //metoda List.of() introdusa in Java 9
        List<String> orase = List.of("Timisoara", "Arad", "Cluj", "Targu-Mures");
        orase.forEach(System.out::println);
    }
}
```



- *forEach* este un operator introdus în Java 8 pentru a traversa colecțiile. Acesta traversează colecția și realizează pentru fiecare element acțiunea transmisă ca și parametru de intrare
- O referință către metoda *println* a fost transmisă ca și parametru de intrare a metodei *forEach* care traversează colecția și realizează acțiunea dată de parametrul de intrare pentru fiecare element al colecției, lucru care va determina afișarea colecției în consolă

## 2.19 Stream API

- Introdus în Java 8
- Un *stream* este un iterator al cărui rol este de a accepta un set de acțiuni să fie aplicate fiecărui din elementele pe care le conține
- Un *stream* reprezintă o secvență de obiecte dintr-o sursă cum ar fi o colecție de obiecte, un vector sau mai multe elemente individuale.
- *Stream*-urile au fost proiectate cu scopul de a face procesarea colecțiilor mai simplă și concisă.
- Spre deosebire de colecții cu ajutorul *stream*-urilor se pot face o serie de prelucrări încă din etapa de declarare
- Operațiile *stream*-urilor pot să fie ori intermediare ori terminale. Operațiile terminale returnează un rezultat de un anumit tip, iar cele intermediare returnează însăși *stream*-ul, în acest fel se pot înlăntui mai multe operații
- Operațiile *stream*-urilor se aplică fiecărui element din *stream* și pot să fie executate secvențial sau paralel
- Colectiile în Java 8 au fost extinse deci se poate crea un *Stream* foarte ușor apelând una din metodele *Collection.stream()* or *Collection.parallelStream()*
- *Stream*-ul paralel împarte taskul furnizat în mai multe taskuri pe care le rulează în diferite fire de execuție.

- Fisierul *MainApp.java*:

```
package capitolul2.streamuri;

import java.util.List;

class MainApp {
    public static void main(String[] args) {
        //metoda List.of() introdusa in Java 9, creeaza o colectie imutabila de obiecte
        List<String> orase = List.of("Timisoara", "Arad", "Cluj", "Targu-Mures");
        orase.forEach(System.out::println);
        //orase.forEach((o)->System.out.println(o));

        System.out.println("Orase care incep cu litera 'T':");
        orase
            .stream()
            .filter(s -> s.startsWith("T"))
            .forEach(System.out::println);

        System.out.println("\nLista ordonata a oraselor, scrisa cu litere mari: ");
        orase
            .stream()
            .map(String::toUpperCase)
            //.map((o)->o.toUpperCase())
            .sorted((a, b) -> a.compareTo(b))
            .forEach(System.out::println);
    }
}
```

The screenshot shows the Eclipse IDE interface with the 'Console' tab selected. The output window displays the following text:

```
Markers Properties Servers Data Source Explorer Snippets Console
<terminated> MainApp (19) [Java Application] C:\Program Files (x86)\Java\jre1.8.0
Orase care incep cu litera 'T'
Timisoara
Targu-Mures

Lista ordonata a oraselor, scrisa cu litere mari:
ARAD
CLUJ
TARGU-MURES
TIMISOARA
```

- În primul exemplu din programul precedent apelând metoda *stream()* se creează un *stream* secvențial pentru colecția de obiecte de tip *List* ce conține denumirile orașelor. În continuare *stream*-ul se filtrează pentru a rămâne în el doar orașele a căror denumire începe cu litera ‘T’, iar apoi *stream*-ul filtrat se afișează
- Cel de-al doilea exemplu utilizează *stream-uri* pentru a afișa elementele din colecție ordonat și cu litere mari
- Metoda *forEach* aplică metoda transmisă ca și parametru de intrare fiecărui element din colecție sau fiecărui element din *stream*, în funcție de modul de apel al acesteia.
- Întrucât în exemplele precedente parametrul de intrare al metodei *forEach* este o referință a metodei *println*, pentru *stream*-ul asociat monitorului, această metodă se va aplica fiecărui element afișând-ul pe ecran
- Parametrul de intrare al metodei *forEach* este un *Consumer* (interfață funcțională cu o metodă care are un parametru de intrare și nu returnează nimic), aceasta putând fi implementat printr-o expresie *Lambda*. Parametrul de intrare al expresiei *Lambda* reprezintă elementul asupra căruia se va aplica acțiunea din corpul expresiei. Acesta se va aplica rând pe rând fiecărui element din *stream*
- Metoda *filter()* are un parametru de intrare de tip *Predicate* (interfață funcțională cu o metodă care are un parametru de intrare asupra căruia se realizează un test, rezultatul testului fiind returnat sub forma unui *boolean*). În exemplul precedent implementarea acestui *Predicate* s-a făcut printr-o expresie *Lambda* care are un parametru de intrare (elementul din *stream*) iar corpul expresiei *Lambda* reprezintă testul care se realizează asupra parametrului de intrare și care este automat returnat. Metoda *filter()* se va aplica fiecărui element din *stream*, lăsând în *stream*-ul de ieșire doar elementele care trec testul

- Metoda *map()* are un parametru de intrare de tip *Function* (interfață funcțională cu o metodă abstractă care are un parametru de intrare și produce un rezultat). În exemplul precedent, metoda *map* primește ca și parametru de intrare o referință către metoda *toUpperCase* din clasa *String*. Aceasta metoda va fi apelata pentru fiecare element din stream-ul de intrare transformand-ul în majuscule în stream-ul de ieșire.
- Operația de mapare pune în corespondență fiecare element din stream-ul de intrare către un element din stream-ul de ieșire cu ajutorul unei funcții
- Metoda *sorted()* are un parametru de intrare de tip *Comparator* (interfață funcțională cu o metodă de comparare). Metoda de comparare a fost implementată printr-o expresie *Lambda*, prin care se specifică cum să se compare elementele din *stream* pentru a-l ordona. Corpul expresiei Lambda trebuie să returneze o valoare pozitivă când primul parametru al expresiei este mai mare decât al doilea după criteriul de comparare, valoarea zero când cei doi parametri sunt egali și valoare negativă când al doilea parametru este mai mare decât primul
- În cele două exemple cu stream-uri din programul precedent operația terminală este *forEach* care afișează elementele stream-ului pe ecran.
- Operația terminală poate fi *collect(Collectors.toList())* care creează o nouă colecție cu elementele din *stream*
- Exemplul următor creează o listă în care pune numerele naturale extrase dintr-o listă de întregi

```

List<Integer> intregi=List.of(-4,3,-2,5,-8,9);
List<Integer> nr_naturale=intregi
    .stream()
    .filter((nr)->nr>=0)
    .collect(Collectors.toList());

System.out.println("Numere naturale extrase din lista de intregi:");
nr_naturale.forEach(System.out::println);

```

- În exemplul următor operația terminală este *count()* care returnează numărul de elemente din *stream*. Exemplul creează un *stream* paralel pentru o colecție de stringuri, se pune un filtru care lasă în *stream* doar elementele necomplete și apoi se determină numărul acestora cu ajutorul metodei *count*

```

List<String> strings = Arrays.asList("aaa", "", "zz", "bbb", "eee", "    ", "ddd");

int count = (int) strings
    .parallelStream()
    .filter(string -> string.isBlank())
    .count();

```

- Metoda *isBlank()* a fost introdusă în Java 11 și returnează *true* când string-ul conține sirul vid sau doar spații albe

## 2.20 Clasa *Optional*

- Clasa *Optional* oferă suport pentru evitarea excepției *NullPointerException* care apare frecvent în dezvoltarea programelor
- Clasa *Optional* este disponibilă în API-ul Java începând cu Java 8 și a continuat să fie îmbunătățită prin adăugarea de noi metode în Java 9 (*or*, *ifPresentOrElse*, *stream*), în Java 10 (prin introducerea metodei *orElseThrow*) în Java 11 (prin introducere metodei *isEmpty*), etc
- Evitarea excepției *NullPointerException* implică de obicei multe verificări ale *null*-ului. Clasa *Optional* oferă facilități pentru scrierea unui cod fără multe teste ale *null*-ului.
- Prin clasa *Optional* se pot specifica valori alternative care să fie returnate sau cod alternativ care să fie rulat când se întâlnește *null*
- Utilizarea clasei *Optional* crește lizibilitatea codului
- Metoda de mai jos afișează numărul de cuvinte al unui propoziție dată ca și parametru de intrare al funcției, printr-un *String* ale cărui cuvinte sunt separate prin spații

```
public static void nr_cuvinte(String s) {  
    System.out.println("Propozitia: " + s + " are " + s.split(" ").length + " cuvinte");  
}
```

- Dacă la afelul funcției se transmite un *String* care are valoarea *null*, aceasta determină producerea excepției *NullPointerException*
- Pentru evitarea excepției se poate testa *null-ul* precum în exemplul următor:

```

public static void nr_cuvinte(String s) {
    if(s!=null)
        System.out.println("Propozitia: " + s + " are " + s.split(" ").length + " cuvinte");
    else
        System.out.println("Stringul e null");
}

```

- Lucrând cu clasa *Optional*, acest test se poate face în felul următor:

```

public static void nr_cuvinte_cu_optional(String s) {
    Optional<String> opt = Optional.ofNullable(s);
    if (opt.isPresent())
        System.out.println("Propozitia: " + opt.get() + " are " + opt.get().split(" ").length +
                           " cuvinte");
    else
        System.out.println("Stringul e null");
}

```

- *Optional* este un container care poate să conțină o valoare sau nu. Dacă valoarea este prezentă funcția *get()* o va returna
- Metoda *ofNullable()* returnează un obiect *Optional* care va descrie valoarea specificată ca și parametru, dacă aceasta nu este *null*, în caz contra va returna un *Optional* gol (*Optional.empty()*)
- API-ul oferă metode adiționale care depind de prezența sau absența valorii cum ar fi *orElse()* metodă care returnează o valoare implicită în caz că valoarea inspectată nu e prezentă sau metoda *ifPresent()* care execută un bloc de cod dacă valoarea este prezentă

- Un exemplu de utilizare a acestora este următorul:

```
package capitolul2.optional1;

import java.util.Optional;

class MainApp {
    public static void fara_optional_uppercase(String s) {
        if(s!=null) {
            System.out.println(s.toUpperCase());
        }
        else {
            System.out.println("valoare lipsa".toUpperCase());
        }
    }
    public static void cu_optional_uppercase(String s) {
        Optional<String> opt=Optional.empty();
        opt=Optional.ofNullable(s);
        System.out.println(opt.orElse("valoare lipsa").toUpperCase());
    }

    public static void main(String[]args) {
        String s=null;
        //String s="test";
        fara_optional_uppercase(s);
        cu_optional_uppercase(s);
    }
}
```

- În exemplul precedent metoda `orElse()` va returna valoarea din container dacă aceasta este prezentă sau parametrul de intrare în caz contrar. Valoarea returnată de funcție este apoi transformate în majuscule
- Un exemplu de utilizare a metodei `ifPresent()` este următorul:

```
opt.ifPresent(value ->System.out.println(value.toUpperCase()) );
```

- Metoda primește ca și parametru de intrare un obiect care implementează interfața funcțională `Consumer`, interfață ce conține o metodă care primește un singur parametru de intrare și care nu returnează nici un rezultat
- Parametrul de intrare al funcției `ifPresent()` a fost transmis printr-o expresie *Lambda* care afișează cu majuscule valoarea din containerul `Optional`
- O situație frecventă de producere a excepției `NullPointerException` se întâlnește în aplicațiile cu baze de date în care apar situații de genul:

```
Persoana p=findPersoanaById(2);
System.out.println(p.getNume());
```

- În exemplul de mai sus, dacă persoana căutată nu se găsește, rândul următor produce `NullPointerException`
- Situația se poate rezolva lucrând cu `Optional` și cu metoda `ifPresent()` în felul următor

```
Optional<Persoana> optional = findPersoanaById(2);

optional.ifPresent(p -> {
    System.out.println("Numele persoanei este " + p.getNume());
});
```

- Metoda *findPersoanaById()* de mai sus returnează un obiect de tip *Optional* asociat unui obiect de tip *Persoana*
- Metoda *or()* disponibilă începând cu Java 9, returnează obiectul de tip *Optional* care descrie valoarea dacă aceasta este prezentă în container, iar în caz contrar returnează un obiect de tip *Optional* implicit

```
String sir = "o valoare nenula";
//String sir=null;

Optional<String> optional = Optional.ofNullable(sir);
Optional<String> optDefault = Optional.ofNullable("o valoare implicita");
System.out.println(optional.or(() -> optDefault));
```

- Parametrul de intrare al metodei *or()* este de tip *Supplier* (interfață funcțională cu o metodă care nu are nici un parametru de intrare și care returnează un rezultat) aşadar poate fi implementată printr-o expresie *Lambda*
- Metoda *ifPresentOrElse()* disponibilă începând cu Java 9, realizează o anumită acțiune specificată asupra valorii din container dacă aceasta este prezentă, în caz contrar realizează o acțiune „empty based” cum ar fi afișarea unui mesaj

```
// String sir="un sir de caractere";
String sir=null;

Optional<String> optional = Optional.ofNullable(sir);
optional.ifPresentOrElse(System.out::println, ()->System.out.println("Valoare lipsa"));
```

- Metoda *orElseThrow()* este disponibilă începând cu Java 10. Metoda returnează valoarea din containerul *Optional* dacă aceasta este prezentă, iar dacă nu este prezentă aruncă excepția *java.util.NoSuchElementException: No value present*. Metoda este similară cu metoda *get* și începând cu Java 10 se recomandă utilizarea acesteia în locul metodei *get()*

```
try {
    //Integer a = 1;
    Integer a=null;
    Optional<Integer> optional =  Optional.ofNullable(a);
    Integer b  = optional.orElseThrow();
    System.out.println("b="+b);
}
catch(NoSuchElementException ex) {
    System.out.println(ex);
}
```

- În exemplul de mai sus, metoda *orElseThrow()* va face ca *b* să primească valoarea lui *a*, dacă *a* nu este *null*. Dacă *a* este *null* metoda va genera excepția *java.util.NoSuchElementException*

```

try {
    List<Integer> intregi=List.of(1,3,5);
    //List<Integer> intregi=List.of(1,3,5,8);
    int primulPar = intregi.stream()
        .filter(i -> i % 2 == 0)
        .findFirst()
        .orElseThrow();
    System.out.println("Primul intreg par din lista este: "+primulPar);
}
catch(NoSuchElementException ex) {
    System.out.println(ex);
}

```

- În exemplul precedent, metoda *findFirst()* va returna un *Optional*. Acesta poate să conțină sau nu un element în container în funcție de situație. Metoda *orElseThrow()* va returna elementul din containerul *Optional*, dacă există un astfel de element, în caz contrar va arunca excepție
- Metoda *isEmpty* este disponibilă în clasa *Optional* începând cu Java 11. Metoda verifică dacă containerul *Optional* este gol sau nu. Metoda a fost introdusă ca o alternativă la utilizarea metodei *isPresent* cu negație

```
package capitolul2.optional3;

import java.util.Optional;

class MainApp {
    public static void main(String[] args) {
        String s = null;
        System.out.println(!Optional.ofNullable(s).isPresent());
        System.out.println(Optional.ofNullable(s).isEmpty());
        s = "test";
        System.out.println(!Optional.ofNullable(s).isPresent());
        System.out.println(Optional.ofNullable(s).isEmpty());
    }
}
```

```
true  
true  
false  
false
```

## 2.21 Inferența tipului la variabilele locale (local variable type inference)

- Inferența tipului la variabilele locale a fost introdusă în Java 10
- Termenul de inferență desemnează o „operație logică de trecere de la un enunț la altul și în care ultimul enunț este dedus din primul” (dex). Inferența tipului se referă la deducerea tipului de către compilator. Tipul poate fi dedus doar pentru variabile locale, iar notația folosită este **var**
- Exemplul următor arată cum poate fi utilizat **var**:

```
package capitolul2.inferenta_tipului;

import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class MainApp {
    public static void main(String[] args) {
        System.out.println("Înainte de Java 10:");
        int nr1=3;
        String s1="abc";

        System.out.println("nr="+nr1);
        System.out.println("Sirul introdus:"+s1+" are "+s1.length()+" caractere");

        System.out.println("\nCu Java 10:");
        var nr2=3;
        var s2="abc";
```

```

System.out.println("nr="+nr2);
System.out.println("Sirul introdus: "+s2+" are "+s2.length()+" caractere");

System.out.println("\nInainte de Java 10:");
Map<String,List<String>> orase_in_judete1=new HashMap<String,List<String>>();
orase_in_judete1.put("Arad", List.of("Ineu", "Sebis", "Arad"));
orase_in_judete1.put("Timis", List.of("Timisoara", "Lugoj", "Faget"));

for(Map.Entry<String,List<String>> o:orase_in_judete1.entrySet()) {
    System.out.println(o.getKey()+": "+o.getValue());
}

System.out.println("\nCu Java 10:");
var orase_in_judete2=new HashMap<String,List<String>>();
orase_in_judete2.put("Arad", List.of("Ineu", "Sebis", "Arad"));
orase_in_judete2.put("Timis", List.of("Timisoara", "Lugoj", "Faget"));

for(var o:orase_in_judete2.entrySet()) {
    System.out.println(o.getKey()+": "+o.getValue());
}
}

```

- După cum se observă în exemplul de mai sus, începând cu Java 10, la declararea unei variabile locale se poate pune cuvântul *var* în locul specificării tipului concret, compilatorul fiind cel care deduce tipul efectiv. Utilizarea lui *var* este avantajoasă în special în cazul unor tipuri lungi (vezi ultimul exemplu)

- După cum se observă în exemplul de mai sus, începând cu Java 10, la declararea unei variabile locale se poate utiliza cuvântul *var* în locul specificării tipului concret, compilatorul fiind cel care deduce tipul efectiv. Utilizarea lui *var* este avantajoasă în special în cazul unor tipuri lungi (vezi ultimul exemplu)
- În exemplul de mai sus s-a creat o colecție de tip *Map*, care pune în corespondență chei unice către anumite valori. Cheile au fost alese *String-uri* și reprezintă denumiri de județe, iar valorile sunt colecții de tip *list* imutabile care conțin orașe din județele alese.
- **Utilizarea lui var are anumite restricții. var NU poate fi utilizat pentru:**
  - Variabile locale neinitializate

```
var a; //error: Cannot use 'var' on variable without initializer
```

- Variabile locale initializate cu *null*

```
var a=null; //error: Cannot infer type for local variable initialized to 'null'
```

- Parametri de intrare ai unor metode

```
void afisare(var a) { //error: 'var' is not allowed here  
    System.out.println(x);  
}
```

- Tipul returnat de o metodă

```
var calculeaza() { //error: 'var' is not allowed here  
    return 2*3*4;  
}
```

- Parametri de intrare ai unui constructor

```
class Punct{  
    private int x,y;  
    public Punct(var x, var y) { //error: 'var' is not allowed here  
        this.x=x;  
        this.y=y;  
    }  
    //...  
}
```

- Variabile membre în clase

```
class Punct{  
    private var x,y; //error: 'var' is not allowed here  
    public Punct() {}  
    //...  
}
```

- Vectori inițializați

```
var v = { "a", "b", "c" }; //error: Array initializer needs an explicit target-type
```

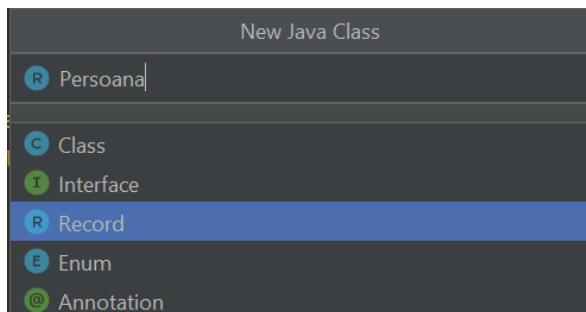
- Expresii lambda

```
Predicate<Integer> predicate1 = n -> n%2 == 0; //ok  
int x=5;  
System.out.print(predicate1.test(x)?x+" este par":x+" este impar");
```

```
var predicate2 = n -> n%2 == 0; //error: Lambda expression needs an explicit target-type
```

## 2.22 Tipul înregistrare (record)

- A fost introdus în Java 14 ca și caracteristică cu previzualizare și a devenit o caracteristică permanentă începând cu Java 16 (martie, 2021)
- Tipul record este un tip special de clasă care ajută la reducerea codului standard.(boilerplate code).
- Definirea unei înregistrări este o modalitate concisă de definire a unui obiect imutabil care deține date.
- Crearea unei înregistrări se face cu ajutorul cuvântului cheie *record*
- În *IntelliJ* crearea unei înregistrări se realizează cu ajutorul comenzi *New Class*, apoi se introduce numele clasei și se alege tipul *Record*. În *Eclipse* comanda pentru crearea unei înregistrări este *New Record*. În continuare se creează în *IntelliJ* înregistrarea cu numele *Persoana*



- În fișierul Persona.java a fost creată înregistrarea *Persoana* cu următorul conținut

```
package capitolul2.record;

public record Persoana() {}
```

- Se completează între parantezele rotunde lista de componente ale record-ului

```
package capitolul2.record;

public record Persoana(String nume, int varsta) {}
```

- *record-ul* construit mai sus reprezintă o clasă finală care are:
  - variabile membre *nume* și *varsta* finale și private
  - constructor cu parametri
  - *gettere* care asigură acces de citire a variabilelor membre private și care au aceeași denumire cu variabilele membre
  - metoda *toString()* care returnează un *String* ce conține valorile variabilelor membre
  - Metoda *hashCode()* care returneză *Object.hashCode(lista\_componentelor)*; în cazul de față *Object.hashCode(nume,varsta)*;
  - Metoda *equals* care redefinește metoda *equals* din clasa *Object* astfel încât cele două obiecte comparate nu sunt egale doar în situația în care indică spre aceeași referință ci și dacă variabilele membre ale celor două obiecte comparate au aceleași valori

- Tipul record nu poate extinde alte clase încă dintrucât implicit extinde clasa `java.lang.Record` și în Java nu este permisă moștenirea multiplă
- O clasă normală nu poate extinde o clasă de tip record pentru că aceasta este finală
- O clasă de tip record poate implementa interfețe
- Orice alt câmp care se declară într-un *record* în afara listei de componente (în afara parantezelor rotunde și în interiorul parantezelor accolade) trebuie să fie declarat static
- O clasă de tip record poate conține metode normale
- În programul principal din fișierul *MainApp.java* se declară și se instanțiază două obiecte de tip record

```
package capitolul2.record;

class MainApp {
    public static void main(String[] args) {
        Persoana p1 = new Persoana("Oana", 23);
        Persoana p2 = new Persoana("Oana", 23);
        System.out.println(p1);
        System.out.println("Persoana cu numele " + p1.nume() + " are varsta " + p1.varsta());
        System.out.println(p1.equals(p2));
    }
}
```

- Se observă existența constructorului cu parametri, a metodei *toString()*, a getterelor care dau acces de citire a câmpurilor nume și varsta și a metodei *equals* care returnează *true* cu toate că cele două obiecte comparate nu indică spre aceeași referință dar au același conținut

## 2.23 Design patterns

- Dacă o problema apare de mai multe ori, soluția ei este descrisă ca un şablon (pattern).
- Un şablon de proiectare descrie o problemă care se întâlnește în mod repetat în proiectarea programelor și soluția generală pentru problema respectivă
- Un şablon este o soluție a unei probleme, într-un anumit context
- Soluția este exprimată folosind clase și obiecte. Atât descrierea problemei cât și a soluției sunt abstrakte astfel încât să poată fi folosite în multe situații diferite.
- Folosind şabloanele de proiectare putem face codul mai flexibil, reutilizabil și ușor de întreținut.
- Cartea de referință pentru şabloane de proiectare este “**Design Patterns: Elements of Reusable Object-Oriented Software**”, având ca autori pe Erich Gamma, Richard Helm, Ralph Johnson și John Vlissides. Ea este adesea numită “Gang of Four Book” (GoF). Apărută în 1994, ea s-a bucurat de un mare succes și a activat subiectul şabloanelor în dezvoltarea software, în ultimii ani acesta fiind tratat în numeroase conferințe, articole și cărți.

- Erich Gamma: “*Proiectarea unui software orientat pe obiecte este grea, iar proiectarea unui software orientat pe obiecte reutilizabil este și mai grea*”.
- Designerii experimentați reutilizează soluțiile cu care au lucrat în trecut.
- Cunoștința şabloanelor cu care a lucrat în trecut, îi permite unui proiectant să fie mai productiv, iar designurile rezultate să fie mai flexibile și reutilizabile
- Şabloanele de proiectare sunt soluții demonstre și testate de către programatori cu experiență. Ele nu sunt o soluție absolută la o problemă, dar ele furnizează claritate în arhitectura sistemului și posibilitatea de a construi un sistem mai bun.
- Şabloanele de proiectare se împart în următoarele **categorii**:
  - **Şabloane creaționale (Creational Patterns)** sunt pattern-uri ce implementează mecanisme de creare a obiectelor. În această categorie se încadrează pattern-urile **Singleton**, **Factory** și **AbstractFactory**
  - **Şabloane structurale (Structural Patterns)** sunt pattern-uri ce simplifică design-ul aplicației prin găsirea unei metode de a defini relațiile dintre entități. În această categorie se încadrează pattern-ul **Facade**.
  - **Şabloane comportamentale (Behavioral Patterns)** sunt pattern-uri ce definesc modul în care obiectele comunică între ele. În această categorie se încadrează pattern-ul **Command**

## Şablonul de proiectare *Singleton*

- Şablonul *Singleton* este utilizat pentru a restricţiona numărul de instanţieri ale unei clase la un singur obiect. Pentru a asigura o singură instanţiere a clasei, constructorul trebuie făcut *private*. A fost discutat în cadrul subcapitolului *2.4 Constructori - 2.4.2 Şablonul de proiectare Singleton*

## Şablonul de proiectare *Factory*

- Patternul *Factory* face parte din categoria *Şabloanelor Creaţionale* şi ca atare rezolvă problema creării unui obiect fără a specifica exact clasa obiectului ce urmează a fi creat. Acest lucru este implementat prin definirea unei metode al cărei scop este crearea obiectelor.
- Metoda va avea specificat ca parametru de returnat în antet un obiect de tip părinte, urmând ca, în funcţie de alegerea programatorului, aceasta să creeze şi să întoarcă obiecte noi de tip copil.
- Situaţia cea mai întâlnită în care se potriveşte acest pattern este aceea când trebuie instantiată multe clase care implementează o anumită interfaţă sau extind o altă clasă (eventual abstractă), ca în exemplul de mai jos. Clasa care foloseşte aceste subclase nu trebuie să „ştie” tipul lor concret ci doar pe al părintelui.

```
package capitolul2.design_patterns.factory;

interface Forma {
    public void deseneaza();
}

class Dreptunghi implements Forma{
    @Override
    public void deseneaza() {
        System.out.println("Deseneaza dreptunghi");
    }
}
class Triunghi implements Forma{
    @Override
    public void deseneaza() {
        System.out.println("Deseneaza triunghi");
    }
}
class Cerc implements Forma{
    @Override
    public void deseneaza() {
        System.out.println("Deseneaza cerc");
    }
}

class FormaFactory {
    public Forma getForma(String tipulFormei){
        if(tipulFormei == null) {
            return null;
        }
        if(tipulFormei.equalsIgnoreCase("cerc")) {
            return new Cerc();
        }
    }
}
```

```

    else
        if(tipulFormei.equalsIgnoreCase("dreptunghi")){
            return new Dreptunghi();
        }
        else
            if(tipulFormei.equalsIgnoreCase("triunghi")){
                return new Triunghi();
            }
        return null;
    }
}

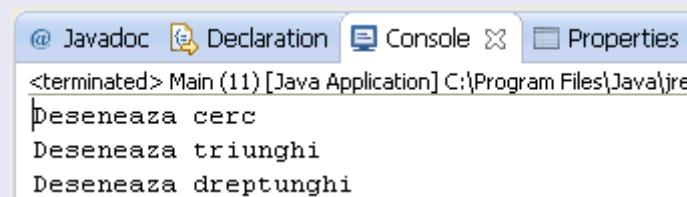
class MainApp {
    public static void main(String[] args) {
        FormaFactory factory = new FormaFactory();

        Forma forma1 = factory.getForma("cerc");
        forma1.deseneaza();

        Forma forma2 = factory.getForma("triunghi");
        forma2.deseneaza();

        Forma forma3 = factory.getForma("dreptunghi");
        forma3.deseneaza();
    }
}

```

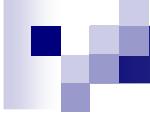


The screenshot shows a Java application running in an IDE. The code in the editor matches the one above. Below it, a console window displays the output of the application's execution:

```

@ Javadoc Declaration Console Properties
<terminated> Main (11) [Java Application] C:\Program Files\Java\jre...
Deseneaza cerc
Deseneaza triunghi
Deseneaza dreptunghi

```



## Şablonul de proiectare *Abstract Factory*

- *Abstract Factory* face parte din rândul şabloanelor creaţionale
- Acest şablon funcţionează ca o *fabrică de fabrici*.
- Sistemul este configurat să lucreze cu mai multe familii de produse
- Oferă o interfaţă pentru crearea unei familii de obiecte corelate, fără a specifica explicit clasele acestora
- În şablonul *Abstract Factory* o interfaţă este responsabilă pentru crearea unei fabrici de obiecte relaţionate fără a le specifica clasele – Fiecare fabrică generată poate să creeze obiecte precum *factory pattern*

```
package capitolul2.design_patterns.abstract_factory;

interface Forma {
    public void deseneaza();
}

class Dreptunghi implements Forma{
    @Override
    public void deseneaza() {
        System.out.println("Deseneaza dreptunghi");
    }
}

class Triunghi implements Forma{
    @Override
    public void deseneaza() {
        System.out.println("Deseneaza triunghi");
    }
}

class Cerc implements Forma{
    @Override
    public void deseneaza() {
        System.out.println("Deseneaza cerc");
    }
}

interface Culoare {
    void umple();
}
```

```
class Rosu implements Culoare {
    @Override
    public void umple() {
        System.out.println("-Rosu");
    }
}

class Verde implements Culoare {
    @Override
    public void umple() {
        System.out.println("-Verde");
    }
}

class Albastru implements Culoare {
    @Override
    public void umple() {
        System.out.println("-Albastru");
    }
}

abstract class AbstractFactory {
    abstract Culoare getCuloare(String culoare);
    abstract Forma getForma(String forma) ;
}

class FormaFactory extends AbstractFactory{
    public Forma getForma(String tipulFormei){
        if(tipulFormei == null){
            return null;
        }
        if(tipulFormei.equalsIgnoreCase("cerc")){
            return new Cerc();
        }
    }
}
```

```

        else
            if(tipulFormei.equalsIgnoreCase("dreptunghi")){
                return new Dreptunghi();
            }
            else
                if(tipulFormei.equalsIgnoreCase("triunghi")){
                    return new Triunghi();
                }
        return null;
    }

    Culoare getCuloare(String culoare) {
        return null;
    }
}

class CuloareFactory extends AbstractFactory {
    @Override
    public Forma getForma(String tipulFormei){
        return null;
    }
    @Override
    Culoare getCuloare(String culoare) {
        if(culoare == null){
            return null;
        }
        if(culoare.equalsIgnoreCase("rosu")){
            return new Rosu();
        }
        else if(culoare.equalsIgnoreCase("verde")){
            return new Verde();
        } else if(culoare.equalsIgnoreCase("albastru")){
            return new Albastru();
        }
        return null;
    }
}

```

```

class FactoryProducer {
    public static AbstractFactory getFactory(String choice) {
        if(choice.equalsIgnoreCase("forma")){
            return new FormaFactory();
        } else if(choice.equalsIgnoreCase("culoare")){
            return new CuloareFactory();
        }
        return null;
    }
}

class MainApp {
    public static void main(String[] args) {
        AbstractFactory shapeFactory = FactoryProducer.getFactory("forma");
        Forma f1 = shapeFactory.getForma("cerc");
        f1.deseneaza();

        Forma f2 = shapeFactory.getForma("dreptunghi");
        f2.deseneaza();

        Forma f3 = shapeFactory.getForma("triunghi");
        f3.deseneaza();

        AbstractFactory colorFactory = FactoryProducer.getFactory("culoare");
        Culoare c1 = colorFactory.getCuloare("rosu");
        c1.umple();

        Culoare c2 = colorFactory.getCuloare("verde");
        c2.umple();

        Culoare c3 = colorFactory.getCuloare("albastru");
        c3.umple();
    }
}

```

## Şablonul de proiectare *Facade*

- Facade ascunde complexitatea sistemului și furnizează o interfață către client prin care clientul poate accesa sistemul. Face parte din rândul patternurilor structurale și aduce o interfață simplificată sistemelor existente pentru a le ascunde complexitatea
- Acest pattern implică o singură clasă care oferă metode simplificate care sunt cerute de client și delegă apelurile către metodele existente
- Se recomandă utilizarea acestui design pattern atunci când se dorește atribuirea unei interfețe simple unui subsistem complex.

```
package capitolul2.design_patterns.facade;
interface Forma {
    public void deseneaza();
}
class Dreptunghi implements Forma{
    @Override
    public void deseneaza() {
        System.out.println("Deseneaza dreptunghi");
    }
}
class Triunghi implements Forma{
    @Override
    public void deseneaza() {
        System.out.println("Deseneaza triunghi");
    }
}
class Cerc implements Forma{
    public void deseneaza() {
        System.out.println("Deseneaza cerc");
    }
}
```

```

class CreatorForme {
    private Forma dreptunghi;
    private Forma triunghi;
    private Forma cerc;

    public CreatorForme() {
        dreptunghi = new Dreptunghi();
        triunghi = new Triunghi();
        cerc = new Cerc();
    }

    public void deseneazaCerc(){
        cerc.deseneaza();
    }
    public void deseneazaDreptunghi(){
        dreptunghi.deseneaza();
    }
    public void deseneazaTriunghi(){
        triunghi.deseneaza();
    }
}

class MainApp {
    public static void main(String[] args) {
        CreatorForme shapeMaker = new CreatorForme();

        shapeMaker.deseneazaCerc();
        shapeMaker.deseneazaDreptunghi();
        shapeMaker.deseneazaTriunghi();
    }
}

```

## Şablonul de proiectare *Command*

- Command pattern este un şablon comportamental. O cerere este încapsulată într-un obiect sub forma de comandă şi este pasată unui obiect invocator. Obiectul invocator caută un obiect potrivit să execute comanda şi pasează cererea aceluia obiect
- Un avantaj a acestui design pattern este că separă obiectul care invocă o operaţie de obiectul care execută operaţia.

```
package capitolul2.design_patterns.command;

class Comutator {
    private EchipamentElectric echipament;

    public EchipamentElectric getEquipment() {
        return echipament;
    }
    public void setEquipment(EchipamentElectric echipament) {
        this.echipament = echipament;
    }
    public void on() {
        System.out.println("Comutatorul a fost pornit!");
        echipament.porneste();
    }
    public void off() {
        System.out.println("Comutatorul a fost oprit!");
        echipament.opreste();
    }
}
```

```
class ComutatorCuFir extends Comutator {}

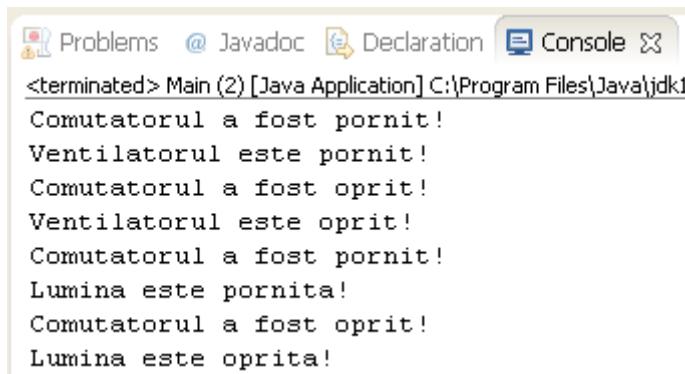
class ComutatorFaraFir extends Comutator {}

interface EchipamentElectric {
    void porneste();
    void opreste();
}

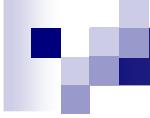
class Ventilator implements EchipamentElectric {
    public void porneste() {
        System.out.println("Ventilatorul este pornit!");
    }
    public void opreste() {
        System.out.println("Ventilatorul este oprit");
    }
}

class Candelabru implements EchipamentElectric {
    public void porneste() {
        System.out.println("Lumina este pornita!");
    }
    public void opreste() {
        System.out.println("Lumina este oprita!");
    }
}
```

```
class MainApp {  
    public static void main(String[] args) {  
        EchipamentElectric ventilator = new Ventilator();  
        EchipamentElectric candelabru = new Candelabru();  
  
        Comutator comutator_cu_fir = new ComutatorCuFir();  
        Comutator comutator_fara_fir = new ComutatorFaraFir();  
  
        comutator_cu_fir.setEquipment(ventilator);  
        comutator_cu_fir.on();  
        comutator_cu_fir.off();  
  
        comutator_fara_fir.setEquipment(candelabru);  
        comutator_fara_fir.on();  
        comutator_fara_fir.off();  
    }  
}
```

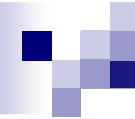


Problems @ Javadoc Declaration Console X  
<terminated> Main (2) [Java Application] C:\Program Files\Java\jdk1  
Comutatorul a fost pornit!  
Ventilatorul este pornit!  
Comutatorul a fost oprit!  
Ventilatorul este oprit!  
Comutatorul a fost pornit!  
Lumina este pornita!  
Comutatorul a fost oprit!  
Lumina este oprita!



## Şabonul de proiectare *Model-View-Controller (MVC)*

- MVC este un şablon de proiectare care are rolul de a separa componentele aplicaţiei
- Utilizarea acestui şablon de proiectare presupune divizarea aplicaţiei în trei componente (Modelul, View-ul si Controller-ul)
- **Modelul** reprezintă datele aplicaţiei şi este un obiect
- **View-ul** reprezintă vizualizarea datelor conţinute în model (interfaţa cu utilizatorul)
- **Controllerul** acţionează şi asupra view-ului şi asupra modelului. El urmăreşte modelul şi actualizează datele din view atunci când modelul se schimbă. Controllerul păstrează view-ul si modelul separate. Controller-ul se ocupa cu comunicarea dintre utilizatori şi model
- Separarea conceptelor face ca testarea şi mențenanţa aplicaţiei să fie mai uşoare



## ■ Fisierul *Persoana.java*

```
package capitolul2.design_patterns.mvc;

class Persoana {
    private int id;
    private String nume;
    private int varsta;

    public Persoana(){}
    public Persoana(int id, String nume, int varsta) {
        this.id = id;  this.nume = nume;  this.varsta = varsta;
    }
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getNume() {
        return nume;
    }
    public void setNume(String nume) {
        this.nume = nume;
    }
    public int getVarsta() {
        return varsta;
    }
    public void setVarsta(int varsta) {
        this.varsta = varsta;
    }
}
```

- Fisierul **PersoanaView.java**

```
package capitolul2.design_patterns.mvc;

class PersoanaView {
    public void printPersoana(int id, String nume, int varsta){
        System.out.println(id+", "+nume+", "+varsta);
    }
}
```

- Fisierul **PersoanaController.java**

```
package capitolul2.design_patterns.mvc;

class PersoanaController {
    private Persoana model;
    private PersoanaView view;

    public PersoanaController(Persoana model, PersoanaView view){
        this.model = model;
        this.view = view;
    }
    public void setPersoanaId(int id){
        model.setId(id);
    }
    public int getPersoanaId(){
        return model.getId();
    }
    public void setPersoanaNume (String nume){
        model.setNume(nume);
    }
}
```

```

public String getPersoanaName(){
    return model.getNume();
}
public void setPersoanaVarsta(int varsta){
    model.setVarsta(varsta);
}
public int getPersoanaVarsta(){
    return model.getVarsta();
}
public void updateView(){
    view.printPersoana(model.getId(), model.getNume(),model.getVarsta() );
}
}

```

## ■ Fisierul **MainApp.java**

```

package capitolul2.design_patterns.mvc;
class MainApp {
    public static void main(String[] args) {
        Persoana model = getPersoanaFromDB();
        PersoanaView view = new PersoanaView();
        PersoanaController controller = new PersoanaController(model, view);
        controller.updateView();

        controller.setPersoanaVarsta(21);
        controller.updateView();
    }
    private static Persoana getPersoanaFromDB(){
        Persoana p = new Persoana();
        p.setId(1);
        p.setNume("Maria");
        p.setVarsta(20);
        return p;
    }
}

```

### **3. Colecții de obiecte**

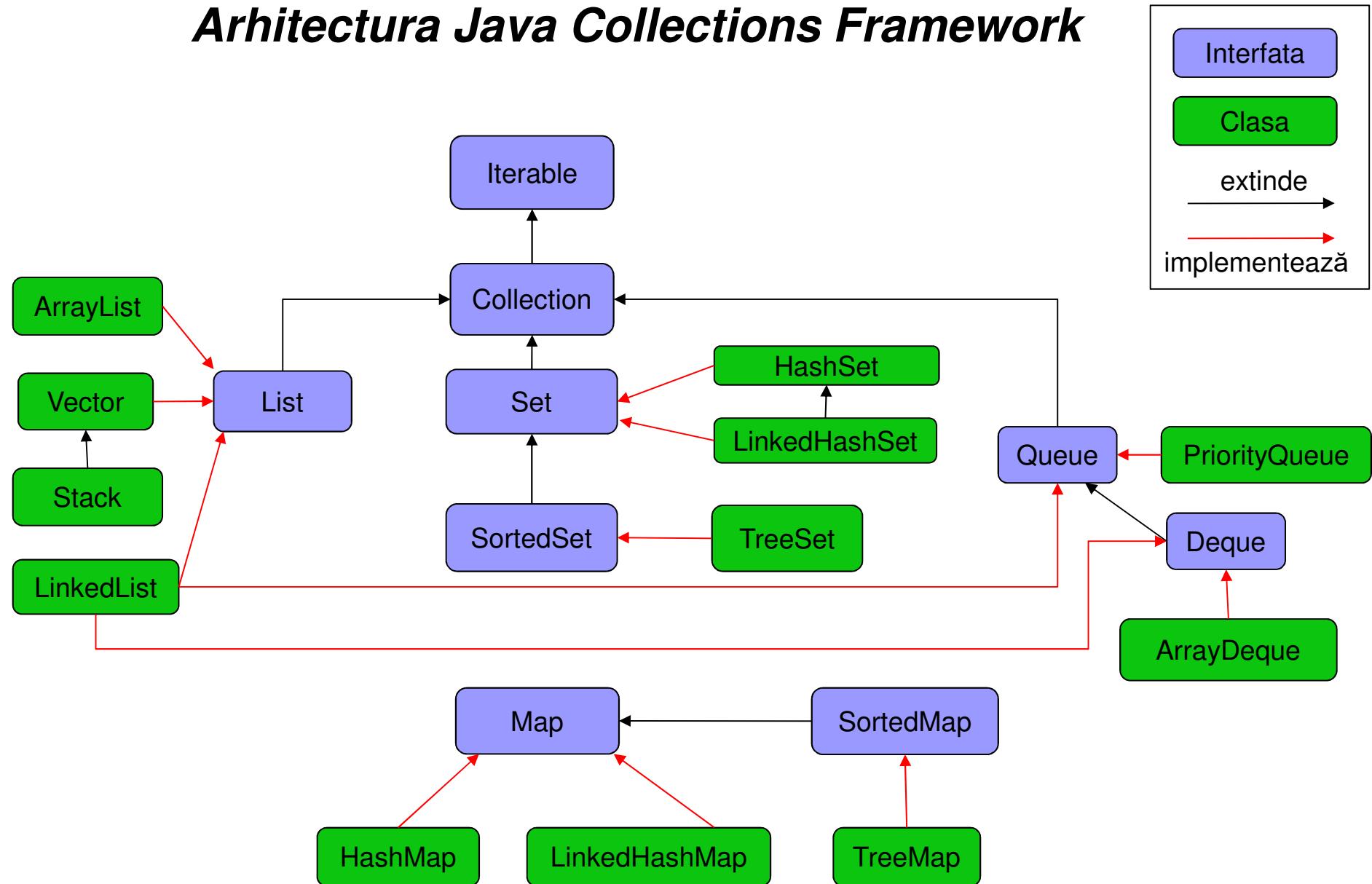
Sl. dr. ing. Raul Robu

2022-2023, Semestrul 2

## 3.1 Noțiuni generale

- “The Collections Framework “ constituie o arhitectură unificată pentru reprezentarea și manipularea colecțiilor, permitând manipularea acestora independent de detaliile implementării lor.
- Framework-ul este bazat pe **interfețe** de colecții. El include **implementări** ale acestor interfețe și **algoritmi** pentru a le manipula
- Avantajele colecțiilor de obiecte:
  - reduc efortul de programare și cresc performanță – prin faptul că furnizează structuri de date și algoritmi performanți care altfel ar trebui scriși de programator
  - cresc gradul de reutilizare a codului – prin faptul că o furnizează o interfață standard pentru colecții și algoritmi pentru a le manipula
- Algoritmii sunt furnizați de clasa *Collections*, sub forma unor metode statice, a căror prim argument este colecția de elemente asupra căreia se va aplică operația. Majoritatea algoritmilor operează asupra unor instanțe *List*.
- Algoritmii permit ordonarea colecțiilor, căutarea unui element, determinarea valorii maxime, a celei minime, etc

# Arhitectura Java Collections Framework



## Interfete

- **Iterable** – este interfața aflată la rădăcina ierarhiei de interfețe. Interfața *Collection* extinde interfața *Iterable* și toate clasele care implementează interfețele derivate din interfața *Collection* o implementează și pe aceasta și pe *Iterable*. De exemplu clasa *ArrayList* implementează interfața *List* care extinde interfața *Collection*, care la rândul ei extinde interfața *Iterable*, aşadar clasa *ArrayList* implementează interfețele *List*, *Collection* și *Iterable*
- **Collection** – un grup de obiecte. Este utilizată când se dorește maximul de generalitate.
- Interfețele ***List***, ***Set*** și ***Queue*** extind interfața ***Collection***.
- ***List*** – colecție de asemenea cunoscută ca secvență. Elementele se regăsesc în ordinea adăugării. Duplicatele sunt permise. Permite accesul la elemente cu ajutorul unui indice
- ***Set*** - nu permite elemente dupăcat. Elementele nu pot fi accesate prin indice
- ***SortedSet*** – extinde interfața *Set* moștenind comportamentul acestuia și în plus având elementele automat ordonate, fie în ordine naturală fie printr-un obiect de tip *Comparator* specificat când instanța de tip *SortedSet* este creată
- ***Queue*** - cozile în mod tipic ordonează elementele în stil *FIFO* (first-in-first-out). Printre excepții sunt cozile prioritare care ordonează elementele pe baza valorii lor (valoările sunt mai importante decât ordinea de adăugare). Indiferent de stilul de ordonare, elementul care se va șterge atunci când se apelează metoda *remove()* este cel din capul cozii.

- **Deque** – Extinde interfața *Queue*. Suportă adăugarea și eliminarea de elemente de la ambele capete a structurii de date, de aceea poate fi utilizată ca și o coadă (FIFO) sau ca o stivă (LIFO)
- **Map** – o punere în corespondență cheile către valorile (keys to values). Fiecare cheie corespunde unei valori, cheile sunt unice iar valorile se pot repeta
- **SortedMap** – un *Map* a cărui elemente sunt ordonate automat prin cheie fie în ordine naturală fie printr-un comparator transmis instanței *SortedMap* atunci când aceasta este creată

## **Implementări**

- **ArrayList** – o implementare a interfeței *List* printr-un vector care își ajustează dimensiunea (cea mai bună implementare a interfeței *List*), conține metode nesincronizate.
- **Vector** - vector folosește pentru a stoca elemente un vector care își ajustează dimensiunea. Este similar cu *ArrayList* dar conține metode sincronizate și multe metode care nu fac parte din interfețele Java *Collections framework*
- **Stack** – subclasă a lui *Vector* care implementează structura de date **LIFO (Last In First Out)**. Pe lângă metodele clasei *Vector* mai conține metode **push()**, **peek()** care realizează operații specifice unei stive
- **LinkedList** – o implementare a interfeței *List* cu listă dublu înlanțuită. Poate furniza o performanță mai bună decât implementările cu *ArrayList* dacă elementele sunt inserate sau șterse frecvent, conține metode nesincronizate. Implementează și interfețele *Queue* și *Deque*.

- **HashSet** - implementarea printr-un tabel de dispersie (*HashTable*) a interfeței *Set* (cea mai bună implementare a interfeței *Set*)
- **LinkedHashSet** – implementarea interfeței *Set* cu ajutorul unui tabel de dispersie și a unei liste înlăntuite (*linked list*). O implementare care face inserare în ordinea adăugării și care rulează aproape la fel de rapid ca un *HashSet*.
- **TreeSet** – implementarea printr-un arbore roșu – negru (*Red-black tree*) a interfeței *SortedSet*. Elementele sunt ordonate pe baza valorii lor
- **PriorityQueue** – implementează interfața *Queue*. Elementele unei cozi cu prioritate sunt ordonate fie în ordine naturală fie cu ajutorul unui comparator furnizat în momentul construirii cozii
- **ArrayDeque** - implementează interfața *Deque*. Este un tip special de vector crescător care permite adăugarea și ștergerea elementelor la ambele capete
- **HashMap** – Implementarea interfeței *Map* cu ajutorul unui tabel de dispersie (în mod esențial un tabel de dispersie nesincronizat care suportă *null* pentru chei și valori (cea mai bună implementare a interfeței *Map*)
- **LinkedHashMap** – implementarea cu tabel de dispersie și listă înlăntuită a interfeței *Map*. O implementare bazată pe inserare ordonată care lucrează aproape la fel de repede ca un *HashMap*
- **TreeMap** - implementarea printr-un arbore roșu - negru a interfeței *SortedMap* .

## 3.2 Interfețe

### ***Interfața Iterable***

- este interfața aflată la rădăcina ierarhiei de interfețe. Interfața *Collection* extinde interfața *Iterable* și toate clasele care implementează interfețe derivate din interfața *Collection* o implementează și pe aceasta
- Începând cu Java 8 conține metoda *forEach*, care permite aplicarea unei acțiuni fiecărui element din colecție
- Conține metoda *iterator()* care creează un iterator peste elementele colecției. Interfața *Iterator* oferă facilități de traversare a elementelor din colecție într-o singura direcție. Metodele interfeței sunt:
  - *hasNext()* - returnează boolean dacă există un element următor sau nu,
  - *next()* – returneză următorul element din colecție
  - *remove()* - șterge ultimul element furnizat de iterator
- Exemplul următor creează cu ajutorul metodei *List.of*, introdusă în Java 9, o colecție imutabilă de obiecte și apoi colecția este traversată și afișată întâi cu *forEach*, apoi cu *Iterator* și apoi cu *for*. Toate aceste moduri de traversare se transferă prin moștenire la subinterfețele sale, respectiv la clasele care le implementează

```
package capitolul3.iterator;

import java.util.Iterator;
import java.util.List;

class MainApp {
    public static void main(String[] args) {
        Iterable<String> colectie=List.of("Arad","Timisoara");
        colectie.forEach(System.out::println);

        Iterator<String> iterator=colectie.iterator();
        while(iterator.hasNext())
            System.out.println(iterator.next());

        for(String s:colectie)
            System.out.println(s);
    }
}
```

## **Interfața Collection**

- O colecție este formată dintr-un grup de obiecte. Interfața *Collection* este folosită pentru transmiterea colecțiilor de obiecte în situații în care se dorește maximul de generalitate. Este interfața de bază pentru interfețele *List*, *Set* și *Queue*
- Pe lângă metodele moștenite de la superinterfața sa *Iterable*, interfața *Collection* oferă metode precum:
  - ***size()*** – returnează numărul de elemente care fac parte din colecție
  - ***isEmpty ()*** - verifică dacă colecția este goală
  - ***contains ()***- verifică dacă un anumit obiect se găsește în colecție
  - ***add(), remove()*** - adaugă / elimină un obiect din colecție
  - ***toArray()*** – returnează un vector care conține elementele colecției
  - ***removeIf()*** – primește ca și parametru un *Predicate* și elimină elementele din colecție care respectă condiția acestuia (metodă disponibilă începând cu Java 8)
  - ***stream*** – returnează un *stream* secvențial care conține elementele colecției (metodă disponibilă începând cu Java 8)

- ***containsAll()*** — returnează *true* dacă colecția ţintă conține toate elementele din colecția specificată
  - ***addAll()*** - adaugă toate elementele din colecția specificată la colecția ţintă
  - ***removeAll()*** – șterge din colecția ţintă toate elementele care sunt de asemenea conținute în colecția specificată
  - ***retainAll()*** – șterge din colecția ţintă toate elementele care nu sunt conținute în colecția specificată. Reține în colecția ţintă doar elementele care sunt și în colecția specificată
  - ***clear()*** – șterge toate elementele din colecție
- Metoda de adăugare este definită destul de general încât să poată fi folosită în colecții de elemente care permit duplicatele respectiv în cele care nu permit acest lucru. Metodele *add ()* și *remove()* returnează *true* dacă adăugarea / ștergerea a reușit și *false* dacă nu

## Interfața List

- O listă este o colecție în care elementele se găsesc în ordinea adăugării. Listele pot conține duplicate. Interfața *List* este implementată de clasele *ArrayList* (cea mai utilizată implementare), *LinkedList*, *Vector* și *Stack*. Pe lângă operațiile moștenite de la interfața *Collection*, interfața *List* dispune de următoarele operații:
  - **Acces pozitional** – manipulează obiectele pe baza poziției lor numerice în listă (metoda *get(pozitie)*, *remove(pozitie)*, *add(pozitie, obiect)*, *set(pozitie, obiect\_nou)*, etc)
  - **Căutare** – cauță după un anumit obiect și îi returnează poziția în listă (metodele *indexOf()* și *lastIndexOf()*)
  - **Vizualizarea unui domeniu** – permite realizarea unor operații asupra elementelor care se află într-un domeniu al listei (metoda *subList()*)
- Nu se pot crea liste de tipuri primitive, trebuie să se folosească clasele înfășurătoare
- Exemplul următor ilustrează cum pot fi realizate mai multe operații asupra listei
- În clasa *Persoana* s-a redefinit metoda *equals* din clasa *Object* astfel încât să returneze *true* dacă obiectul transmis ca și parametru indică spre aceeași referință ca și obiectul curent sau are aceleași caracteristici ca și obiectul curent

```
package capitolul3.liste;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;
import java.util.Stack;
import java.util.Vector;
import java.util.stream.Collectors;

class Persoana{
    private String nume;
    private int varsta;

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

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

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

```

public int getVarsta(){
    return varsta;
}

public String toString(){
    return nume+" "+varsta;
}

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

```

- În clasa *Persoana* s-a redefinit metoda *equals* din clasa *Object* astfel încât să returneze *true* dacă obiectul transmis ca și parametru indică spre aceeași referință ca și obiectul curent sau are aceleași caracteristici ca și obiectul curent. Metoda *equals* este utilizată de metodele *contains()* și *indexOf()*, respectiv *lastIndexOf()* care caută un element în colecție. Implementarea implicită a metodei *equals()* din clasa *Object* compară doar referinta obiectului curent cu cea a obiectului transmis ca și parametru, fără să țină cont de valori

```

class ComparaNume implements Comparator{
    @Override
    public int compare(Object o1, Object o2) {
        return (((Persoana)o1).getNume().compareToIgnoreCase(((Persoana) o2).getNume()));
    }
}

class ComparaVarsta implements Comparator<Persoana>{
    @Override
    public int compare(Persoana p1, Persoana p2) {
        if (p1.getVarsta()>p2.getVarsta()) return 1;
        else
            if (p1.getVarsta()<p2.getVarsta()) return -1;
            else return 0;
    }
}

```

- Clasele *ComparaNume* și *ComparaVarsta* sunt utilizate pentru ordonarea crescătoare a colecției de obiecte după numele persoanelor, respectiv după vîrstă acestora. Cele două clase implementează interfața *Comparator* care dispune de metoda abstractă *compare()* care trebuie să returneze valoarea 1 când primul obiect este mai mare decât al doilea după criteriul de comparare, -1 când al doilea obiect este mai mare decât primul și 0 când sunt egale

```

class MainApp {
    public static void afisari(List<Persoana> pers) {
        System.out.println("----Afisare1---");
        System.out.println(pers);

        System.out.println("----Afisare2---");
        for (int i=0;i<pers.size();i++)
            System.out.println(pers.get(i).toString() );

        System.out.println("----Afisare3---");
        for (Persoana p:pers)
            System.out.println(p);

        System.out.println("----Afisare4---");
        Iterator<Persoana> it=pers.iterator();
        while (it.hasNext()){
            Persoana p=it.next();
            System.out.println(p.toString());
        }
        System.out.println("----Afisare5---");
        pers.forEach(System.out::println);
    }

    public static void amestecare(List<Persoana> pers) {
        System.out.println("----Colectie neordonata---");
        Collections.shuffle(pers);
        System.out.println(pers);
    }
}

```

```

Dimensiunea initiala a listei=4 elemente
----Afisare1---
[Maria 30, Denisa 28, Ela 20, Oana 18]
----Afisare2---
Maria 30
Denisa 28
Ela 20
Oana 18
----Afisare3---
Maria 30
Denisa 28
Ela 20
Oana 18
----Afisare4---
Maria 30
Denisa 28
Ela 20
Oana 18
----Afisare5---
Maria 30
Denisa 28
Ela 20
Oana 18

```

```

public static void ordonari(List<Persoana> pers) {
    amestecare(pers);
    System.out.println("----Ordonare varsta---");
    Collections.sort(pers, new ComparaVarsta());
    System.out.println(pers);

    amestecare(pers);
    System.out.println("----Ordonare nume, varianta 1---");
    Collections.sort(pers, new ComparaNume());
    System.out.println(pers);

    amestecare(pers);
    System.out.println("----Ordonare nume, varianta 2---");
    pers.sort(new ComparaNume());
    System.out.println(pers);

    amestecare(pers);
    System.out.println("----Ordonare nume, varianta 3---");
    pers.sort((a,b)->a.getNume().compareToIgnoreCase(b.getNume()));
    System.out.println(pers);
}

public static void cautare_in_colectie_ordonata(List<Persoana> colectie_ordonata, String nume){
    System.out.println("----Cautare in colectie ordonata---");

    int x=Collections.binarySearch(colectie_ordonata, new Persoana(nume),
                                    (a,b)->a.getNume().compareToIgnoreCase(b.getNume()));
    if (x>=0)
        System.out.println(nume+" gasita pe pozitia "+x);
    else
        System.out.println(nume+" nu a fost gasita");
}

```

----Colectie neordonata---  
[Oana 18, Denisa 28, Maria 30, Ela 20]  
----Ordonare varsta---  
[Oana 18, Ela 20, Denisa 28, Maria 30]  
----Colectie neordonata---  
[Oana 18, Denisa 28, Ela 20, Maria 30]  
----Ordonare nume, varianta 1---  
[Denisa 28, Ela 20, Maria 30, Oana 18]  
----Colectie neordonata---  
[Denisa 28, Oana 18, Maria 30, Ela 20]  
----Ordonare nume, varianta 2---  
[Denisa 28, Ela 20, Maria 30, Oana 18]  
----Colectie neordonata---  
[Maria 30, Ela 20, Denisa 28, Oana 18]  
----Ordonare nume, varianta 3---  
[Denisa 28, Ela 20, Maria 30, Oana 18]

----O noua colectie ordonata dupa nume---  
[Denisa 28, Ela 20, Maria 30, Oana 18]  
----Cautare in colectie ordonata---  
Ela gasita pe pozitia 1

```

public static void cautari_in_colecție_neordonata(List<Persoana> pers, Persoana p) {
    amestecare(pers);
    System.out.println("----Cautare in colecție neordonata---");
    if(pers.contains(p)){//utilizeaza metoda equals din clasa Persoana
        System.out.println("Persoana "+p+" se gaseste in colecție");
    }
    else {
        System.out.println("Persoana "+p+" NU se gaseste in colecție");
    }
    int pozitie=pers.indexOf(p);//utilizeaza metoda equals din clasa Persoana
    if(pozitie>=0)
        System.out.println("Persoana "+p+" se gaseste in colecție, pe pozitia "+pozitie);
    else
        System.out.println("Persoana "+p+" NU se gaseste in colecție");
}

public static void main(String[] args) {
    List<Persoana> pers = new ArrayList<Persoana>();
    //List<Persoana> pers = new Vector<Persoana>();
    //List<Persoana> pers = new LinkedList<Persoana>();
    //List<Persoana> pers = new Stack<Persoana>();

    pers.add(new Persoana("Maria",30));
    pers.add(new Persoana("Ela",20));
    pers.add(new Persoana("Oana",18));
    pers.add(1,new Persoana("Denisa",28)); //adauga pe pozitia 1

    System.out.println("Dimensiunea initiala a listei="+pers.size()+" elemente");
}

```

----Colectie neordonata---  
 [Maria 30, Ela 20, Oana 18, Denisa 28]  
 ----Cautare in colecție neordonata---  
 Persoana Denisa 28 se gaseste in colecție  
 Persoana Denisa 28 se gaseste in colecție, pe pozitia 3

```

afisari(pers);
ordonari(pers);
cautari_in_colecție_neordonata(pers,new Persoana("Denisa",28));

var colecție_ordonata=pers
    .stream()
    .sorted((a,b)->a.getNumă().compareToIgnoreCase(b.getNumă()))
    .collect(Collectors.toList());

System.out.println("----0 nouă colecție ordonată după nume---");
System.out.println(colecție_ordonata);

cautare_in_colecție_ordonata(colecție_ordonata, "Ela");

System.out.println("---Stergere---");
//pers.remove(0);//sterge element accesat prin indice
pers.removeIf(p->p.getNumă().equalsIgnoreCase("denisa"));
System.out.println(pers);                                ---Stergere---
                                                       [Maria 30, Ela 20, Oana 18]
                                                       ---Setarea unui element---
                                                       [Maria 30, Ela 20, Roxana 50]
                                                       ---Stergerea unei bucăți din lista---
                                                       [Roxana 50]

System.out.println("---Setarea unui element---");
pers.set(2,new Persoana("Roxana",50));
System.out.println(pers);

System.out.println("---Stergerea unei bucăți din lista---");
pers.subList(0, 2).clear();
System.out.println(pers);

} //funcția main
} //end clasa Mainapp

```

- La instanțierea colecției de obiecte de tip *List* se pot utiliza oricare din cele 4 clase care implementează interfața *List* (*ArrayList*, *Vector*, *LinkedList*, *Stack*)
- Varianta 3 de ordonare după nume implementează interfața funcțională *Comparator* prin intermediul unei expresii *Lambda*, fiind varianta cea mai concisă și recomandată
- Căutarea binară funcționează corect doar în colecții ordonate după câmpul utilizat în căutare. În exemplul precedent, în programul principal s-a creat o colecție ordonată folosind stream-uri, colectori și expresii lambda (disponibile începând cu Java 8) și inferența tipului pentru variabile locale (disponibilă începând cu java 10)
- Căutarea în colecție neordonată se poate realiza cu ajutorul metodelor *contains()*, *indexOf()* care au fost exemplificate mai sus, metode care utilizează metoda *equals()* pentru a realiza comparațiile, sau se poate traversa colecția prin multitudinea de variante disponibile și se scrie un cod de căutare liniară
- Interfața funcțională *Predicate*, care este parametru de intrare a metodei *removeIf*, a fost implementată printr-o expresie *Lambda* care returnează boolean dacă numele persoanei este cel exemplificat
- Metoda *subList()* returnează o listă care conține elementele extrase din lista inițială de la limita inferioară specificată inclusiv și până la cea exterioară specificată exclusiv

## **Interfața Set**

- Un set este o colecție de elemente care nu poate conține duplicate
- Elementele din colecție nu pot fi accesate prin indice, nu există posibilitatea de acces pozitional. Colecția poate fi traversată într-un bloc *for each* sau cu ajutorul unui iterator
- O interfață *Set* conține doar metodele moștenite de la interfața *Collection* și adaugă restricția că duplicatele sunt interzise
- Java oferă 3 implementări ale interfeței *Set*: *HashSet*, *TreeSet* și *LinkedHashSet* (vezi subcapitolul 3.1 - Implementări)
- Presupunând că avem o colecție *col* și vrem să creem o altă colecție care să conțină aceleași elemente dar toate duplicatele eliminate, putem scrie următorul cod:

```
Collection<Type> faraDuplicate = new HashSet<Type>(col);
```

- Un exemplu de eliminare a dupliilor dar care păstrează ordinea elementelor din colecție în urma eliminării este prezentat în continuare:

```

package capitolul3.set;

import java.util.Collection;
import java.util.LinkedHashSet;
import java.util.List;

class MainApp {
    public static void main(String[] args) {
        List<Integer> intregi=List.of(1,2,3,1,2,3);

        Collection<Integer> fara_duplicate=new LinkedHashSet<Integer>(intregi);
        fara_duplicate.forEach(System.out::println);
    }
}

```

```

@ Javadoc
<terminated>
1
2
3

```

- Următorul exemplu subliniază că într-un *Set* duplicatele sunt eliminate și că elementele sunt ordonate diferit în funcție de implementarea aleasă

```

package capitolul3.set;

import java.util.HashSet;
import java.util.Iterator;
import java.util.LinkedHashSet;
import java.util.Set;
import java.util.TreeSet;

class MainApp2 {
    public static void main(String[] args) {
        Set<String> s=new HashSet<String>();
        //Set<String> s=new TreeSet<String>();
        //Set<String> s=new LinkedHashSet<String>();

        s.add("b");
        s.add("a");
        s.add("b");
        s.add("c");

        Iterator<String> iterator=s.iterator();
        while(iterator.hasNext())
            System.out.println(iterator.next());
    }
}

```

Console Problems  
<terminated> SetExample [Java Application]  
Value :b  
Value :c      HashSet  
Value :a

Console Problems  
<terminated> SetExample [Java Application]  
Value :a  
Value :b      TreeSet  
Value :c

Console Problems  
<terminated> SetExample [Java Application] C:\Program File  
Value :b  
Value :a      LinkedHashSet  
Value :c

- În situația în care se utilizează o implementare a interfeței *Set* cu table de dispersie (*HashTable*) cum ar fi *HashSet* sau *LinkedHashSet* pe o colecție de obiecte în care obiectele nu au o ordine naturală, duplicatele nu sunt automat eliminate. Vezi exemplul următor:

```
package capitolul3.set;
import java.util.HashSet;
import java.util.Set;

class Carte{
    private String titlu;
    private String autor;
    public Carte(String titlu, String autor) {
        this.titlu = titlu; this.autor = autor;
    }
    public String toString() {
        return titlu + " " + autor;
    }
}
class MainApp3 {
    public static void main(String[] args) {
        Set<Carte> carti = new HashSet<Carte>();//sau new LinkedHashSet<Carte>();
        carti.add(new Carte("carte1","autor1"));
        carti.add(new Carte("carte2","autor2"));
        carti.add(new Carte("carte3","autor3"));
        carti.add(new Carte("carte1","autor1"));
        System.out.println(carti);
    }
}
```

[carte1 autor1, carte3 autor3, carte2 autor2, carte1 autor1]

- Acest lucru se întamplă pentru ca nu fost redefinite metodele *equals()* și *hashCode()* din clasa *Object* și se folosesc implementările implicite ale acestora
- Implementarea implicită a metodei *equals()* din clasa *Object* compară doar referinta obiectului curent cu cea a obiectului transmis ca și parametru, fără să țină cont de valori
- Tot din acest motiv codul de mai jos afiseaza mesajul *Diferite*, pentru că referințele celor două obiecte sunt diferite, chiar daca ele au același continut.

```
package capitolul13.set;

class MainApp4 {
    public static void main(String[] args) {
        Carte c1 = new Carte("carte1", "autor1");
        Carte c2 = new Carte("carte1", "autor1");

        if(c1.equals(c2))
            System.out.println("Egale");
        else
            System.out.println("Diferite");
    }
}
```

@ Javadoc   
<terminated>  
**Diferite**

- Pentru ca metoda *equals()* să compare câmpurilor obiectului current cu câmpurile obiectului dat ca și parametru, ea trebuie redefinită în clasa *Carte* și specificată o implementare corespunzătoare

- Metoda *hashCode()* returneaza valoarea hash a obiectului care este folosită ca și cheie în tabele de dispersie (hash tables)
- **Pentru două obiecte care sunt egale potrivit metodei *equals()* metoda *hashCode()* trebuie să returneze aceeași valoare hash**
- Acest lucru se poate asigura și dacă metoda *hashCode()* returnează valoarea 1. Totuși pentru o funcționare eficientă se recomandă ca pentru obiecte diferite conform *equals()* valorile hash să fie diferite.
- IDE-urile *IntelliJ* și *Eclipse* dispun de facilități de generare a metodelor *equals* și *hashCode*
- În *IntelliJ* metodele cele două metode se pot genera lansând comanda **Generate (Alt + Insert)** și alegând **equals() and hashCode()**
- În *Eclipse* pot fi generate ambele metode din opțiunea de meniu **Source > Generate hashCode() and equals(...)**
- După ce se generează metodele *hashCode()* și *equals()* în clasa *Carte*, duplicatele nu vor mai fi permise
- Clasa *Carte* după generarea metodelor *equals* și *hashCode* în IntelliJ va avea următorul conținut

```

class Carte{
    private String titlu;
    private String autor;
    public Carte(String titlu, String autor) {
        this.titlu = titlu; this.autor = autor;
    }
    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Carte carte = (Carte) o;
        return titlu.equals(carte.titlu) && autor.equals(carte.autor);
    }

    @Override
    public int hashCode() {
        return Objects.hash(titlu, autor);
    }
    public String toString() {
        return titlu + " " + autor;
    }
}

```

- În cazul în care se utilizează pentru interfața *Set* o implementare prin clasa *TreeSet*, va trebui specificat un comparator. Acesta va fi folosit atât pentru ordonare și pentru eliminarea dupliilor (vezi exemplul următor)

```
package capitolul3.set;

import java.util.Set;
import java.util.TreeSet;

class Angajat implements Comparable<Angajat>{
    private String nume;
    private String firma;
    public Angajat(String nume, String firma) {
        this.nume = nume;
        this.firma = firma;
    }
    @Override
    public String toString() {
        return nume + ", " + firma;
    }
    @Override
    public int compareTo(Angajat a) {
        if(this.nume.equalsIgnoreCase(a.nume)==false)
            return this.nume.compareToIgnoreCase(a.nume);
        else
            return this.firma.compareToIgnoreCase(a.firma);
    }
}
```

```

class MainApp5 {
    public static void main(String[] args) {
        Set<Angajat> angajati=new TreeSet<Angajat>();
        //SortedSet<Angajat> angajati=new TreeSet<Angajat>();
        angajati.add(new Angajat("Tudor","Firma1"));
        angajati.add(new Angajat("Vlad","Firma3"));
        angajati.add(new Angajat("Vlad","Firma2"));
        angajati.add(new Angajat("Vlad","Firma2"));
        angajati.add(new Angajat("Bogdan","Firma4"));
        angajati.forEach(System.out::println);
    }
}

```

@ Javadoc Declaration  
 <terminated> MainApp5 (1)  
 Bogdan, Firma4  
 Tudor, Firma1  
 Vlad, Firma2  
 Vlad, Firma3

- Clasa *Angajat* implementează interfața *Comparable* și astfel metoda *compareTo* care va fi utilizată pentru ordonarea elementelor din colecție și pentru eliminarea dupliilor.
- Implementarea metodei *compareTo()* determină ordonarea după nume a persoanelor din colecție. Dacă sunt persoane cu același nume la firme diferite atunci se face ordonare după firmă. Se consideră că este dupliat dacă se adaugă două persoane cu același nume la aceeași firmă (două obiecte cu aceleași caracteristici). În colecția *Set* dupliile nu sunt permise
- Printr-o implementare corespunzătoare a metodei *compareTo()* se poate considera o altă situație
- Întrucât clasa *TreeSet* implementează și interfața *SortedSet*, care este o extensie a interfeței *Set*, în exemplul precedent poate utiliza această interfață

## Interfața SortedSet

- Un obiect *SotedSet* este un set care menține elementele în ordine crescătoare, sortate pe baza ordonării naturale sau în acord cu un obiect de tip *Comparator* furnizat *SortedSet-ului* în momentul creării. Pe lângă operatorii oferiti de interfața *Set*, interfața *SortedSet* mai aduce operatori pentru:
  - Vizualizarea unui domeniu de valori
  - Determinarea primului și ultimului element din *SortedSet*
  - Returnează comparatorul utilizat pentru a ordona setul

```
package capitolul3.sortedset;
import java.util.SortedSet;
import java.util.TreeSet;

class MainApp {
    public static void main(String[] args) {
        SortedSet<String> orase=new TreeSet<String>();
        orase.add("Timisoara");
        orase.add("Arad");
        orase.add("Ineu");
        orase.add("Sebis");
        orase.forEach(System.out::println);

        System.out.println("\nPrimul oras din colectia ordonata alfabetic este "+orase.first());
        System.out.println("Ultimul oras din colectia ordonata alfabetic este "+orase.last());
        System.out.println("-----");
        orase.subSet("Arad", "Sebis").forEach(System.out::println);
    }
}
```

Arad  
Ineu  
Sebis  
Timisoara

Primul oras din colectia ordonata alfabetic este Arad  
Ultimul oras din colectia ordonata alfabetic este Timisoara  
-----  
Arad  
Ineu

## **Interfața Queue**

- O coadă este o colecție pentru păstrarea elementelor înainte de procesare. Pe lângă operațiile de bază ale interfeței *Collection* (interfața *Queue* extinde interfața *Collection*) cozile dispun de operații de inserare, stergere și inspectare adiționale
- Cozile în mod tipic ordonează elementele în stil *FIFO* (first-in-first-out),
- Interfața *Queue* este implementată de clasele *LinkedList* și *PriorityQueue*
- Implementarea *LinkedList* adaugă elementele la sfârșitul cozii și le elimină de la început, respectând integral principiul *FIFO*
- Implementarea *PriorityQueue* adaugă elementele în coadă funcție de prioritate (se face adăugare ordonată cu ajutorul unui comparator) și le elimină de la începutul cozii
- Metoda *remove()* din interfața *Queue*, nu are parametri, ea elimina mereu elementul din vârful cozii
- Metodele *element()* și *peek()* returneză elementul din vârful cozii fără să-l elimine, diferența între cele două metode fiind că metoda *peek()* returnează *null* când coada este goală iar metoda *element()* aruncă excepție dacă coada este goală
- Elementele nu pot fi accesate prin indice

```
package capitolul3.queue;

import java.util.Collections;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;
import java.util.PriorityQueue;
import java.util.Queue;

class Persoana implements Comparable<Persoana>{
    private String nume;
    private int varsta;

    public Persoana(String nume,int varsta){
        this.nume=nume;
        this.varsta =varsta;
    }
    public Persoana(String nume){
        this.nume=nume;
    }
    public String getNume(){
        return nume;
    }
    public int getVarsta(){
        return varsta;
    }
    public String toString(){
        return nume+" "+varsta;
    }
    @Override
    public int compareTo(Persoana o) {
        return this.nume.compareToIgnoreCase(o.nume);
    }
}
```

```

class MainApp{
    public static void main(String []args){
        Queue<Persoana> coada = new LinkedList<Persoana>();
        //Queue<Persoana> coada = new PriorityQueue<Persoana>();

        coada.add(new Persoana("Bogdan",30));
        coada.add(new Persoana("Vlad",20));
        coada.add(new Persoana("George",18));
        coada.add(new Persoana("Dan",28));

        System.out.println("La coada stau "+coada.size()+" persoane");
        System.out.println("----Coada initiala---");
        System.out.println(coada);

        System.out.println("Prima persoana din coada este "+coada.peek());
        System.out.println("Elementul din varful cozii este "+coada.element());

        System.out.println("Persoana "+coada.remove()+" a iesit din coada (a fost stearsa)");

        System.out.println("In coada au ramas persoanele:");
        coada.forEach(System.out::println);

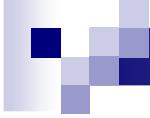
        //ordonare doar pentru coada neordonata de tip LinkedList
        System.out.println("---Ordinea elementelor din coada poate fi modificata prin ordonare,
pentru implementari LinkedList ---");
        Collections.sort((List<Persoana>) coada);
    }
}

```

```
Iterator<Persoana> it=coada.iterator();
while (it.hasNext()){
    Persoana p=it.next();
    System.out.println(p);
}
System.out.println(coada);
for (Persoana pe:coada)
    if (pe.getNume().equals("Vlad"))
        System.out.println("Vlad Gasit!");
}
```

---

```
La coada stau 4 persoane
---Coada initiala---
[Bogdan 30, Vlad 20, George 18, Dan 28]
Prima persoana din coada este Bogdan 30
Elementul din varful cozii este Bogdan 30
Persoana Bogdan 30 a iesit din coada (a fost stearsa)
In coada au ramas persoanele:
Vlad 20
George 18
Dan 28
---Ordinea elementelor din coada poate fi modificata prin ordonare, pentru implementari LinkedList ---
Dan 28
George 18
Vlad 20
[Dan 28, George 18, Vlad 20]
Vlad Gasit!
```



## **Interfața Deque**

- Extinde interfața *Queue*. Suportă adăugarea și eliminarea de elemente de la ambele capete a structurii de date, de aceea poate fi utilizată ca și o coadă (**FIFO**) sau ca o stivă (**LIFO**).
- *Deque* provine de la "**double ended queue**". Pe lângă metodele interfeței *Queue* interfața *Deque* introduce următoarele metode:
- *peekFirst()* – metoda returneză primul element din colecție, fără să-l elimine. Dacă colecția este goală metoda returnează *null*
- *peekLast()* – metoda returneză ultimul element din colecție, fără să-l elimine. Dacă colecția este goală metoda returnează *null*
- *offerFirst()* - Inserează un element la începutul colecției
- *offerLast()* - Inserează un element la sfârșitul colecției

```
package capitolul3.dequeue;

import java.util.ArrayDeque;
import java.util.Deque;

public class MainApp {
    public static void main(String[] args) {
        Deque<Integer> deque=new ArrayDeque<Integer>();
        deque.add(1);
        deque.add(2);
        deque.add(3);

        deque.forEach(System.out::println);

        System.out.println("Primul element din colectie este "+deque.peekFirst());
        System.out.println("Ultimul element din colectie este "+deque.peekLast());

        deque.offerFirst(0);
        deque.offerLast(4);

        System.out.println(deque);
    }
}
```

1  
2  
3  
Primul element din colectie este 1  
Ultimul element din colectie este 3  
[0, 1, 2, 3, 4]

## Interfața Map

- Un *Map* este un obiect care pune în corespondență cheile către anumite valori. Un *Map* nu poate să conțină duplicate de chei. Fiecare cheie corespunde unei singure valori
- Interfața *Map* are trei implementări *HashMap*, *TreeMap* și *LinkedHashMap*
- Operațiile de bază cu *Map* sunt *put*, *get*, *containsKey*, *containsValue*, *size*, *isEmpty*, *putIfAbsent*
- Metodele de explorare a colecției permit *Map-ului* să fie vizualizat în 3 moduri:
  - **keySet** — setul de chei conținut în *Map*
  - **values** — colecția de valori conținute în *Map*. Această colecție nu este de tip *Set* pentru că diferite chei pot să corespundă aceleiași valori
  - **entrySet** — Set-ul de perechi *key-value* conținut în *Map*.

```
for (KeyType key : m.keySet())
    System.out.println(key);
```

- Exemplul următor arată cum se poate utiliza colecția *Map*

```
package capitolul3.map;

import java.util.Map;
import java.util.Comparator;
import java.util.HashMap;
import java.util.Iterator;
import java.util.LinkedHashMap;
import java.util.Map.Entry;
import java.util.Set;
import java.util TreeMap;

class Persoana{
    private String nume;
    private int varsta;

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

    public String getNume() {
        return nume;
    }

    @Override
    public String toString() {
        return nume + ", " + varsta;
    }
}
```

```

class MainApp{
    public static void main(String []args){
        Map<String,Persoana> map=new HashMap<String,Persoana>();
        //Map<String,Persoana> map=new TreeMap<String,Persoana>();
        //Map<String,Persoana> map=new LinkedHashMap<String,Persoana>();

        map.put("cheia1",new Persoana("Maria",20) );
        map.put("cheia2",new Persoana("Ileana",10) );
        map.put("cheia3",new Persoana("Oana",20) );
        map.put("cheia4",new Persoana("Denisa",30));
        map.put("cheia1",new Persoana("Maia",30));

        //Set<Entry<String, Persoana>> entryset=map.entrySet();
        var entryset=map.entrySet();

        //Iterator<Entry<String, Persoana>> it=entryset.iterator();
        var it=entryset.iterator();

        while(it.hasNext())
        {
            //Entry<String, Persoana> m =it.next();
            var m =it.next();

            String key=m.getKey();
            Persoana value=m.getValue();
            System.out.println("Cheie :" +key+ " Valoare :" +value.toString());

            if (key.equalsIgnoreCase("cheia3"))
                it.remove();
        }
    }
}

```

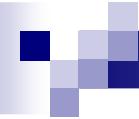
```

map.putIfAbsent("cheia1", new Persoana("Zaza",30));
System.out.println(map);

entryset
.stream()
.sorted(Map.Entry.comparingByKey(Comparator.reverseOrder()))
//.sorted(Map.Entry.comparingByValue((a,b)->a.getNume().compareToIgnoreCase(b.getNume())))
.forEach(System.out::println);
}
}                                Cheie :cheia1 Valaore :Maia, 30
                                    Cheie :cheia2 Valaore :Ileana, 10
                                    Cheie :cheia3 Valaore :Oana, 20
                                    Cheie :cheia4 Valaore :Denisa, 30
{cheia1=Maia, 30, cheia2=Ileana, 10, cheia4=Denisa, 30}
cheia4=Denisa, 30
cheia2=Ileana, 10
cheia1=Maia, 30

```

- Apelul metodei *put* pentru o cheie existentă dar o nouă valoare determină înlocuirea vechii valori
- Exemplul utilizează inferență tipului pentru variabilele locale care ajută la scrierea a mai puțin cod pentru a realiza același acțiuni (vezi liniile comentate)
- Iteratorul permite eliminarea unui element din colecție în timpul parcurgerii acesteia
- Metoda *putIfAbsent* pună cheia și valoarea specificată în colecția *Map*, dacă cheia nu se găsește. În exemplul considerat „cheia1” era introdusă deja, aşadar valoarea nu a fost înlocuită
- Ultimul exemplu afișează elementele din *map* ordonate descrescător după cheie cu ajutorul *stream*-urilor. Se poate decommenta apelul al doilea al lui *sorted* și comentă primul apel al lui *sorted* în felul acesta afișându-se elementele din colecție ordonate crescător după valoare (nume)



### 3.3 Clasa *Vector*

- Clasa *Vector* folosește pentru a stoca elemente într-un vector care își ajustează dimensiunea
- Clasa *Vector* implementează interfața *List*, deci dispune de toate metodele acesteia dar are și multe metode care nu sunt în interfața *List* cum ar fi *addElement()*, *firstElement()*, *lastElement()*, *capacity()*, *elementAt()*, *removeElementAt()*, *insertElementAt()*, *removeElement()*, etc
- Capacitatea unui Vector (*capacity*) este întotdeauna cel puțin la fel de mare ca și dimensiunea vectorului (*size*). Capacitatea unui vector se poate modifica în funcție de conținutul acestuia. (De exemplu, capacitatea inițială este 10 și rămâne atât până se adaugă primele 10 obiecte. Înainte să se adauge elementul al 11-lea se produce o reallocare și capacitatea vectorului se dublează, ajungând la 20). Dispune de un constructor prin care se poate specifica capacitatea inițială, care va fi și cea utilizată la fiecare relocare.
- Clasa *Vector* este sincronizată, ceea ce înseamnă că metodele care modifică structura vectorului cum ar fi *add ()*, *remove ()* sunt sincronizate și pot fi folosite cu încredere în aplicațiile cu fire de execuție (sunt thread safe). *ArrayList* nu este thread safe deci nu este recomandată în fire de execuție. Pe de alta parte *ArrayList* este mai rapidă, deci recomandată în programele fără fire de execuție

```

package capitolul3.vector;

import java.util.Collections;
import java.util.Vector;

class MainApp {
    public static void main(String[] args) {
        Vector<Integer> vector=new Vector<Integer>();
        vector.add(0);
        System.out.println("Capacitate vector="+vector.capacity());
        System.out.println("Numar elemente in vector="+vector.size());

        for (int i=1;i<11;i++) {
            vector.addElement(i);
        }

        vector.insertElementAt(99, 5);

        System.out.println("Capacitate vector="+vector.capacity());
        System.out.println("Numar elemente in vector="+vector.size());

        System.out.println("---Elementele vectorului---");
        for (int i=0;i<vector.size();i++)
            System.out.println(vector.elementAt(i));
    }
}

```

|Capacitate vector=10  
 Numar elemente in vector=1  
 Capacitate vector=20  
 Numar elemente in vector=12  
 ---Elementele vectorului---  
 0  
 1  
 2  
 3  
 4  
 99  
 5  
 6  
 7  
 8  
 9  
 10

```
System.out.println("");
vector.removeElementAt(0);
vector.forEach(System.out::println);

System.out.println("Primul element este "+vector.firstElement());
System.out.println("Ultimul element este "+vector.lastElement());

System.out.println("");
Collections.sort(vector);
System.out.println(vector);
}
```

---Dupa stergerea primului element---

1  
2  
3  
4  
99  
5  
6  
7  
8  
9  
10|

Primul element este 1  
Ultimul element este 10

---Ordonare---

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 99]

### 3.4 Clasa Properties

- Clasa Properties este utilizată la colecții de obiecte formate din perechi chei-valoare la care atât cheia cât și valoarea sunt striguri. Clasa nu face parte din Java collections framework
- Proprietățile pot să fie salvate într-un stream sau încărcate dintr-un stream cu ajutorul metodelor load() și store()

```
package capitolul3.properties;

import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.util.Iterator;
import java.util.Properties;
import java.util.Set;

class MainApp {
    public static void main(String[] args) throws IOException {
        Properties judete = new Properties();
        judete.setProperty("TM", "Timis");
        judete.setProperty("AR", "Arad");
        judete.setProperty("HD", "Hunedoara");

        System.out.println(judete);
    }
}
```

{TM=Timis, AR=Arad, HD=Hunedoara}

```

//Set<Object> jud = judete.keySet();
var jud = judete.keySet();

//Iterator<Object> itr = jud.iterator();
var itr = jud.iterator();

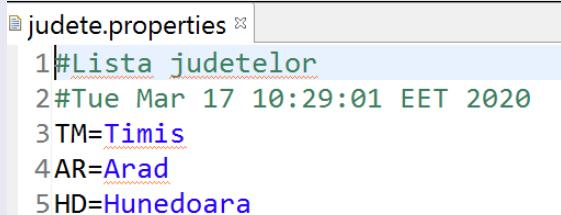
while (itr.hasNext()) {
    String s = (String) itr.next();
    System.out.println("Judetul " + judete.getProperty(s) + " are codul " + s + ".");
}

scrie(judete, "judete.properties");
System.out.println("citire din fisier");
Properties jud_in = new Properties();
citeste(jud_in, "judete.properties");
}

public static void scrie(Properties p, String fis) {
    FileWriter writer;
    try {
        writer = new FileWriter(fis);
        p.store(writer, "Lista judezelor");
        writer.close();
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
}

```

Judetul Timis are codul TM.  
 Judetul Arad are codul AR.  
 Judetul Hunedoara are codul HD.



```

public static void citeste(Properties p, String fis) {
    FileReader reader;
    try {
        reader = new FileReader(fis);
        p.load(reader);
        p.list(System.out);
        reader.close();
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
}

```

citire din fisier  
 -- listing properties --  
 TM=Timis  
 AR=Arad  
 HD=Hunedoara

- Introducerea elementelor în colecție se poate face cu ajutorul metodelor *setProperty()* sau *put()*
- Setul de chei poate fi obținut cu ajutorul metodei *keySet()*
- Valoarea unei chei se poate obține cu ajutorul metodei *getProperty()*
- Metoda *list()* permite afișarea colecției în streamul dat ca și parametru de intrare, care poate fi asociat monitorului, unui fișier, etc
- Metoda *remove()* permite eliminarea unui element din colecție

## 4. Tratarea excepțiilor

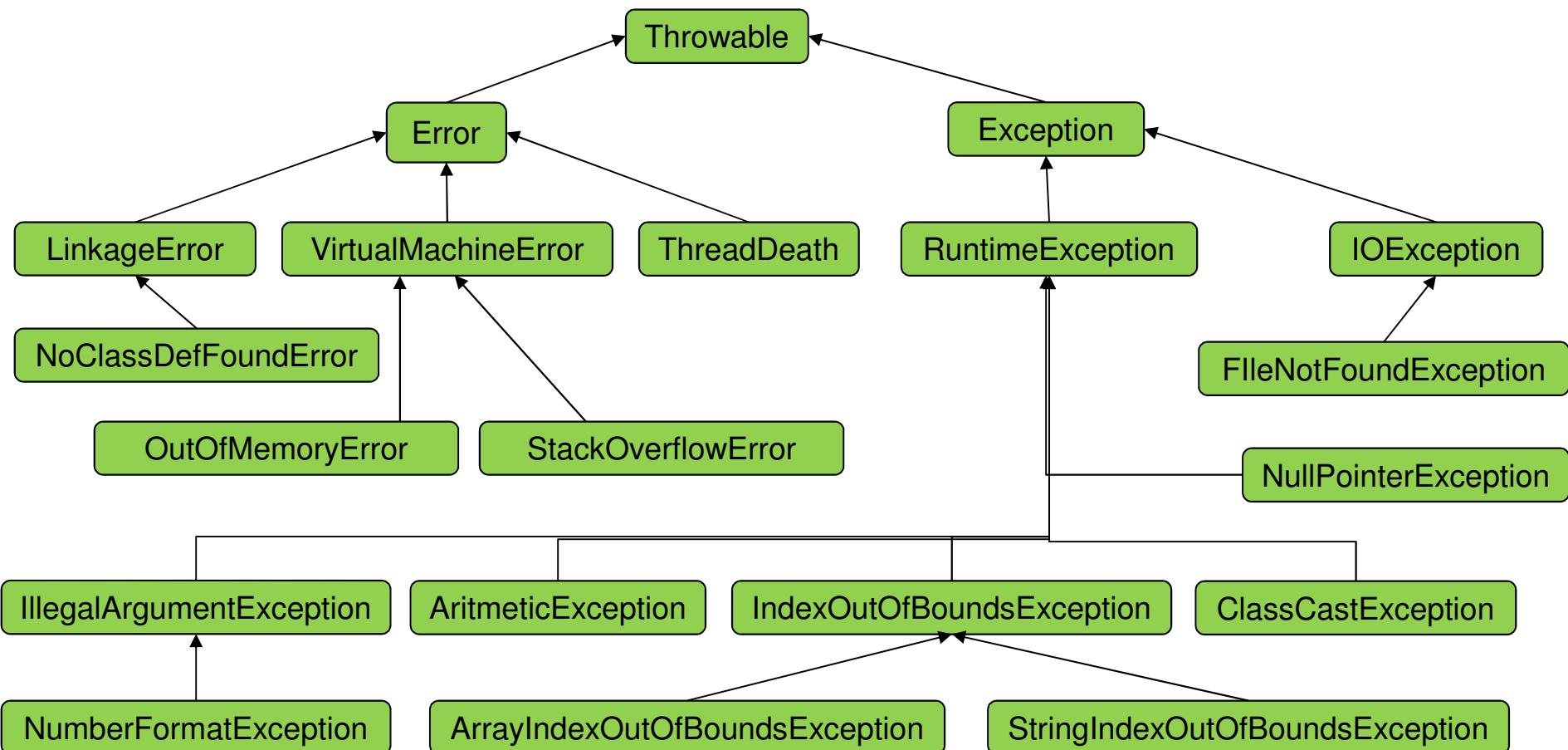
Sl. dr. ing. Raul Robu

2022-2023, Semestrul 2

## 4.1 Tratarea excepțiilor

- O excepție este un eveniment care nu permite continuarea normală a execuției programului
- Exemple de excepții
  - Încercarea de a accesa un element aflat dincolo de limitele unui vector (**ArrayIndexOutOfBoundsException**)
  - Încercarea de a accesa un fișier de intrare care nu există (**FileNotFoundException**)
  - Încercarea de a realiza o împărțire la 0 (**ArithmetricException**)
  - Încercarea de a accesa cu metoda `charAt()` un caracter din afara unui String (**StringIndexOutOfBoundsException**)
  - Încercarea de a converti în întreg cu ajutorul metodei `Integer.parseInt()` un String necorespunzător (**NumberFormatException**)
- Utilizarea excepțiilor este un mecanism elegant de a renunța la execuția secvenței de cod care a provocat excepția, semnalarea acesteia și eventual execuția unei secvențe de tratare corespunzătoare
- O excepție este un obiect ce aparține casei **Throwable** sau unei clase derivate din aceasta
- Java oferă o ierarhie de clase predefinite pentru tratarea excepțiilor, care se găsesc în pachetul **java.lang**. La rădăcina ierarhiei se află clasa **Throwable**

# Ierarhia de clase pentru tratarea excepțiilor



Clasă  
extinde

**Notă:** pachetul **java.lang** mai conține încă multe clase pentru tratarea excepțiilor pe lângă cele exemplificate în schema de mai sus

- Din clasa **Throwable** sunt derivate în mod direct următoarele două ierarhii de clase:
  - **Error** – pentru erorile nerecuperabile. De obicei aceste erori se referă la mașina virtuală sau la editarea legăturilor. Aceste erori nu trebuie tratate pentru că în cazul apariției unor astfel de erori nu mai poate continua execuția. În astfel de situații se afișează un mesaj și programul se închide
  - **Exception** se împarte în două ierarhii:
    - O ierarhie care pornește din clasa *RuntimeException* (se referă în general la greșeli de programare cum ar fi depășirea vectorilor sau utilizarea proastă a tipurilor)
    - O altă ierarhie care poate să fie creată de către programator
- Programatorul poate utiliza clasele predefinite de excepții sau poate să-și definească propriile clase pentru tratarea excepțiilor, pentru situațiile de excepție care pot apărea în dezvoltarea programelor și care nu sunt prevăzute în API-ul Java (De exemplu se citește codul numeric personal al unei persoane de la tastatură și dacă este introdus greșit se produce excepția `CNPInvalid`)
- Clasele definite de programator pentru tratarea excepțiilor trebuie să fie derivate direct sau indirect din **Throwable** (prin convenție se derivează din clasa **Exception**)

- Mecanismele de emitere-captare a excepțiilor:
  - permit separarea explicită a secvențelor de cod care tratează situațiile normale față de cele care tratează excepțiile.
  - oferă suportul necesar pentru a „forța“ o funcție să returneze, în situații deosebite, valori ale altui tip decât cel specificat în antet ca tip returnat.
- Tratarea unei excepții are două componente:
  - Componenta care semnalează excepția (*throws*)
  - Componenta care tratează excepția (prinde excepția)
- Java pune la dispoziție următoarele construcții de bază pentru lucrul cu excepții
  - instrucțiunea **throw** pentru emiterea excepțiilor;
  - Clauza **throws** prin care o funcție precizează că în anumite situații poate arunca o excepție
  - blocul **try** pentru delimitarea secvențelor de cod sensibile la apariția excepțiilor
  - blocul **catch** pentru definirea secvenței de captare și tratare a unei excepții;
  - Blocul **finally** care conține cod care se va executa indiferent dacă excepția se produce sau nu

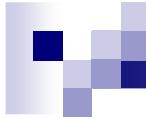
## Instrucțiunea throw

throw obiect;

- obiect trebuie să fie o referință la un obiect al unei clase care face parte din ierarhia ce are la bază clasa *Throwable*
- O instrucțiune **throw** poate să apară într-o funcție numai dacă:
  - Se găsește într-un bloc *try - catch* care captează tipul de excepție returnat de expresia din *throw* (sau o superclasă a acestuia)
  - Definiția funcției este însorită de o clauză *throws* în care apare tipul de excepție respectiv (sau o superclasă al acestuia)
  - Excepția generată aparține claselor *RuntimeException* sau *Error* sau descendenților acestora, în acest caz nu este neceasă plasarea acesteia într-un bloc *try* și nici utilizarea lui *throws*

## Clauza throws

- Dacă într-o metodă este posibil să apară o excepție fără ca aceasta să fie tratată, atunci în linia de declarație a metodei trebuie specificat tipul excepției respective (pentru excepții care nu aparțin claselor *RuntimeException*, *Error* sau subclaselor acestora)
- O metodă trebuie să declare că poate să arunce o excepție pentru ca metodele care o apelează să prevadă tratarea excepției respective



```
tip_returnat nume_functie(lista_param) throws tip_ex1, tip_ex2,...,tip_exn
```

- Această clauză se atașează la antetul unei funcții și ea precizează tipurile de excepții care pot fi generate pe parcursul execuției funcției respective
- Dacă în lista de tipuri din clauza *throws* există cel puțin două tipuri, *tip\_exi* și *tip\_exj*, deriveate dintr-o superclasă comună *tip\_exsup*, limbajul permite înlocuirea celor două cu *tip\_exsup*.
- Dacă în listă se trece doar *Exception*, informația este foarte vagă pentru cititorul programului
- Dacă programatorul nu prevede clauza *throws*, și nici secvențe *try-catch* într-o funcție dezvoltată de el, care apelează funcții predefinite dotate cu clauze *throws*, care aruncă excepții nederivate din *Error* sau *RuntimeException* compilatorul va sesiza acest lucru ca eroare.
- În continuare este prezentat un exemplu în care se utilizează clauza *throws* și instrucțiunea *throw*

```

package capitolul4.exceptii1;

import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

class MainApp {
    public static void main(String[] args) throws FileNotFoundException {
        Scanner scanner=new Scanner(new File("src/capitolul4/exceptii/in.txt"));
        while(scanner.hasNext())
            System.out.println(scanner.nextLine());
        scanner.close();

        throw new ArithmeticException("Exceptie la impartirea cu zero");
    }
}

```

- În exemplul precedent constructorul clasei *Scanner* care primește ca și parametru un obiect de tip *File* și aruncă excepția *FileNotFoundException*

`java.util.Scanner.Scanner(File source) throws FileNotFoundException`

- Excepția trebuie ori tratată (pus apelul constructorului în *try*) ori aruncată mai departe (funția *main* se completează cu clauza *throws*) altfel programul va avea eroare de compilare *Unhandled exception type FileNotFoundException*. Acest lucru este necesar deoarece clasa *FileNotFoundException* nu este derivată din *Error* sau *RuntimeException*)

- Pe de altă parte, ultima linie de cod din funcția *main*, aruncă o excepție de tip *ArithmeticException*, clasă care este derivată din clasa *RuntimeException* iar codul nici nu o tratează, nici nu o aruncă mai departe, acest lucru nu cauzează eroare de compilare

## **Blocurile try -catch**

- Acestea indică zona de instrucțiuni în care pot să apară excepțiile, tipul excepțiilor și modul în care se face tratarea acestora

```
try {  
    //secventa obisnuită de operatii, care poate fi intreruptă de apariția unei exceptii  
}  
catch(tip_ex1 e){  
    //tratare excepție de tipul tip_ex1  
}  
catch(tip_ex2 e){  
    //tratare excepție de tipul tip_ex2  
}  
//...  
catch(tip_exn | tip_exm e){  //incepând cu java 7  
    //tratare excepție de tipul tip_exn și tip_exm  
}  
finally {  
    //secventa optională  
}
```

- Un bloc *try* este urmat de 0, 1 sau mai multe blocuri *catch* și eventual de un bloc *finally*
- Dacă pe parcursul executării instrucțiunilor din blocul *try* este emisă o excepție, secvența se întrerupe și se baleiază blocurile *catch* pentru a-l găsi pe cel corespunzător excepției emise
- Dacă un astfel de bloc există se execută instrucțiunile din el. După aceea execuția programului continuă cu linia care urmează după ultimul bloc *catch* sau, dacă blocul opțional *finally* există atunci cu instrucțiunile din el
- Prin bloc *catch* corespunzător unei excepții se înțelege un bloc *catch* care are tipul specificat ca parametru identic cu tipul excepției, sau o supraclasă a tipului excepției.
- Dacă nu se găsește nici un catch corespunzător unei excepții se caută un try înconjurător și catch-ul corespunzător (blocurile *try-catch* pot fi imbricate, aşadar se repetă procesul)
- Dacă se ajunge cu propagarea până în main excepția este preluată de handler-ul de excepții al mașinii virtuale care determină abandonul programului, cu afișarea unui mesaj corespunzător
- Un bloc *catch* se poate termina normal (adică prin epuizarea instrucțiunilor care îl compun) sau printr-un *throw* care să emită o nouă excepție sau tot pe cea primită. În acest din urmă caz are loc retransmisia excepției.
- Un bloc *catch* poate prinde mai multe tipuri de excepții care sunt separate printr-o bară verticală

```
catch (NumberFormatException | ArrayIndexOutOfBoundsException e){}
```

## **Secvența finally**

- Secvența finally reprezintă un mecanism prin care se forțează execuția unei porțiuni de cod indiferent dacă o excepție a apărut sau nu.

```
package capitolul4.exceptii2;

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

class MainApp {
    public static boolean cautare(FileReader f, String cuvant) throws IOException{
        BufferedReader reader=null;
        try{
            reader = new BufferedReader(f);
            String linie;
            while ((linie = reader.readLine()) != null)
                if(linie.contains(cuvant))
                    return true;
            return false; //nu s-a gasit cuvantul in fisier
        }
        finally {
            System.out.println("Secventa finally");
            if(reader != null) reader.close();
        }
    }
}
```

```
public static void main(String[] args) throws IOException {
    String cuvant_cautat="linia";
    boolean gasit=cautare(new FileReader("src/capitolul4/exceptii2/in.txt")
                           ,cuvant_cautat);

    if(gasit)
        System.out.println("Cuvantul "+cuvant_cautat+" se gaseste in fisier");
    else
        System.out.println("Cuvantul "+cuvant_cautat+" nu se gaseste in fisier");
}
```

---

Secventa finally  
Cuvantul linia se gaseste in fisier

- Aşa cum arată captura de ecran de la rulare, cu toate că în *try* se returnează valoare, lucru care determină ieşirea din funcţie, codul din secventa *finally* se execută şi afişează pe ecran mesajul *Secventa finally*

## **Blocuri try cu resurse**

- O resursă este un obiect care trebuie închis după ce programul a terminat de lucrat cu el (vezi obiectul numit *reader* din exemplul precedent)
- Într-ul bloc *try* cu resurse, resursele sunt închise automat la finalul blocului (nu este necesară închiderea lor explicită, precum în exemplul precedent)
- Orice obiect care implementează interfața *java.lang.AutoCloseable*, poate fi folosit ca o resursă
- Blocurile try cu resurse au fost introduse în Java 7
- Codul din exemplul precedent poate fi rescris mai comasat dacă se folosește un bloc *try* cu resurse, precum în exemplul următor:

```
public static boolean cautare2(FileReader f, String cuvant) throws IOException {
    try(BufferedReader reader=new BufferedReader (f)){
        String linie;
        while ((linie = reader.readLine()) != null)
            if(linie.contains(cuvant))
                return true;
        return false; //nu s-a gasit cuvantul in fisier
    }
}
```

- Dacă sunt mai multe resurse acestea se separă prin simbolul punct și virgulă

- Blocurile try cu resurse din Java 7 au limitarea că resursa trebuie să fie declarată în cadrul blocului *try*. Java 9 permite ca resursa să fie declarată în afara blocului *try*, vezi exemplul următor:

```
package capitolul4.exceptii3;

import java.io.FileNotFoundException;
import java.io.PrintStream;

class MainApp {
    public static void pana_la_java_9() {
        try(PrintStream ps=new PrintStream("out1.txt")) {
            ps.print("Scrie ceva in fisier");
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }

    public static void de_la_java9() throws FileNotFoundException {
        PrintStream ps=new PrintStream("out2.txt");
        try(ps) {
            ps.print("Scrie ceva in fisier");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public static void main(String []args) throws FileNotFoundException {
        pana_la_java_9();
        de_la_java9();
    }
}
```

- Clasa *PrintStream* implementează mai multe interfețe, printre care și interfața *AutoCloseable*, deci obiectul de tip *PrintStream* poate fi utilizat în blocuri *try* cu resurse pentru a închide automat stream-ul corespunzător, astfel nu mai este necesar apelul explicit al metodei *close()*
- Obiectul *PrintStream* permite crearea și scrierea într-un *stream* de ieșire. În exemplul de mai sus se creează fișierele *out1.txt* și *out2.txt* și se scrie câte un text în ambele fișiere
- Din motive de compatibilitate în Java 9, stream-ul poate fi declarat în interiorul sau în afara blocului *try*, dar în versiunile mai vechi decât 9, nedeclararea stream-ului în interiorul blocului *try* produce eroare de compilare
- Dacă sunt mai multe resurse se separă prin punct și virgulă
- În următorul exemplu sunt produse și prinse exceptiile *ArrayIndexOutOfBoundsException*, *NumberFormatException*, *NullPointerException*

```
package capitolul4.exceptii4;

public class MainApp {
    public static void exceptie_array() {
        int [] tab = new int[10];
        int i;
        try {
            for(i=0;;)
                tab[i++]=i;
        }
        catch(ArrayIndexOutOfBoundsException e) { //cand i >=tab.length
            System.out.println(e);
            //System.out.println(e.getLocalizedMessage());
        }
    }

    public static void exceptie_conversie() {
        try {
            int x=Integer.parseInt("zzz");
            System.out.println("x="+x);
        }catch(NumberFormatException e) {
            System.out.println("Conversie nereusita! "+e.getMessage());
        }
    }
}
```

```

public static void exceptie_null() {
    try {
        String s=null;
        System.out.println(s.toUpperCase());
    }catch(NullPointerException e) {
        e.printStackTrace();
    }
}
public static void main(String[] args) {
    exceptie_array();
    exceptie_conversie();
    exceptie_null();
}
}

```

- Ieșirea programului este afișată în captura de ecran de mai jos. După cum se vede, excepțiile determinate au fost prinse și afișate prin mesajele corespunzătoare. În fiecare catch s-a utilizat alt mod de a afișa mesajul aferent excepției

---

```

java.lang.ArrayIndexOutOfBoundsException: Index 10 out of bounds for length 10
Conversie nereusita! For input string: "zzz"
java.lang.NullPointerException: Cannot invoke "String.toUpperCase()" because "s" is null
    at capitolul4.exceptii4.MainApp.exceptie_null(MainApp.java:26)
    at capitolul4.exceptii4.MainApp.main(MainApp.java:34)

```

- Funcția `exceptie_array()` încearcă să pună un element într-un vector folosind un indice care determină accesarea unui element care este în afara spațiului rezervat pentru vector, lucru care produce excepția `ArrayIndexOutOfBoundsException`

- Funcția `exceptie_conversie()` încearcă să convertească în întreg un sir de caractere care nu conține doar cifre, lucru care nu este posibil și se produce excepția `NumberFormatException`
- Funcția `exceptie_null()` încearcă să afișeze pe ecran cu majuscule un sir de caractere care este `null`, lucru care nu este posibil și se produce excepția `NullPointerException`. Apelul unei metode a unui obiect care este null, produce excepția corespunzătoare

## ***Crearea unor noi clase de excepții***

- În situația în care excepțiile predefinite ale limbajului nu sunt suficiente pentru situațiile de excepție care pot să apară în dezvoltarea unui program, se pot crea propriile clase de excepții. Acestea trebuie să fie derivate direct sau indirect din clasa `Throwable`, prin convenție se derivează din clasa `Exception`
- De exemplu dacă într-un program vrem să citim codul numeric personal al unei persoane și pornind de la acesta să calculăm data nașterii și să aflăm vârstă, putem crea o clasă `CNPException`, pentru situațiile de excepție care pot apărea la introducerea unui CNP (nu se introduc 13 caractere, nu se introduc numai cifre, cifrele care codifică sexul, luna nașterii sau ziua nașterii nu sunt valide, etc).
- Exemplul următor schizează un model de program care poate fi adaptat pentru diferite situații de excepții care pot apărea în dezvoltarea programelor

```
package capitolul4.exceptii5;

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

class MainApp {
    public static void main(String[] args) {
        boolean sw=false;

        do {
            //preia datele de intrare
            try {
                calculeaza_date_iesire(date_intrare);
                afiseaza_date_iesire();
                sw=true;
            }
            catch(MyException e) {
                System.out.println(e);
            }
        }while(!sw);
    }
}
```

- Clasa *MyException* are un constructor care primește ca și parametru de intrare un sir de caractere pe care îl transmite către constructorul superclasei (clasei *Exception*). Acest sir de caractere reprezintă mesajul care va fi afișat atunci când excepția se produce, iar conținutul acestui mesaj se va specifica în instrucțiunea *throw* atunci când se va arunca un obiect de tip *MyException*.
- În *do...while* se vor prelua anumite date de intrare de la tastatură, care vor fi transmise către funcția *calculeaza\_date\_iesire(date\_intrare)* care pe baza acestora va trebui să calculeze și să returneze datele de ieșire. Funcția *calculeaza\_date\_iesire(date\_intrare)* nu va putea să-și îndeplinească scopul dacă datele de intrare sunt într-un format necorespunzător. În această situație funcția va arunca o excepție de tipul *MyException*, însotită de un mesaj corespunzător.
- Excepția aruncată va fi prinsă în blocul *catch*, se va afișa mesajul excepției și tot procesul se va relua cu preluarea din nou a datelor de intrare de la tastatură
- Dacă datele de intrare se introduc corect, atunci se calculează cu succes datele de ieșire, se afișează și variabila booleană se pune pe true, fapt care va determina oprirea cilului repetitiv
- **Tema:** Să se realizeze un program care aplică modelul precedent în problema determinării vârstei unei persoane pe baza CNP-ului. Programul va citi CNP-ul persoanei de la tastatură, va determina data nașterii persoanei și ținând cont de data curentă citită din sistem va determina și afișa vârstă. Se va realiza o metodă care primește la intrare CNP-ul persoanei, calculează și returnează vârstă. Dacă CNP-ul este introdus greșit de la tastatură metoda va arunca o excepție de tip CNPEronat însotită de o descriere a problemei. Procesul se va relua până la introducerea corectă a CNP-ului



## **5. Java Database Connectivity**

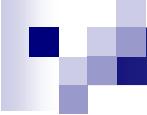
Sl. dr. ing. Raul Robu

2022-2023, Semestrul 2



# CUPRINS

- 5.1 Caracteristicile generale ale JDBC
- 5.2 Caracteristicile și utilizarea MySQL
- 5.3 Crearea unui proiect Maven care utilizează JDBC
- 5.4 Preluarea datelor din baza de date
  - 5.4.1 Stabilirea unei conexiuni către baza de date,
  - 5.4.2 Execuția interogărilor SQL
  - 5.4.3 Procesarea rezultatelor
  - 5.4.4 Închiderea conexiunii
- 5.5 Rularea comenziilor SQL cu parametri
- 5.6 Actualizarea conținutului unui tabel cu ajutorul comenziilor SQL
- 5.7 Actualizarea conținutului unui tabel cu ajutorul actualizațiilor programatice
- 5.8 Apelul procedurilor stocate
- 5.9 Gestionația tabelelor (creare, ștergere, modificare)
- 5.10 Determinarea denumirii coloanelor unei tabele și tipul lor
- 5.11 JDBC și Oracle



## 5. Java Database Connectivity (JDBC)

### 5.1 Caracteristicile generale ale JDBC

- JDBC este o tehnologie care permite conectarea la baze de date și oferă metode pentru interogarea și actualizarea acestora.
- Tehnologia JDBC este orientată către baze de date relaționale (*Oracle, MySql, Microsoft Sql Server, etc*), dar permite conectarea și la fișiere tabelare (*Microsoft Excel spreadsheets*) sau fișierele plate (*flat files*).
- Conectarea la un anumit tip de bază de date se face cu ajutorul driver-ului aferent bazei de date respective
- Clasele JDBC sunt conținute în pachetele ***java.sql*** și ***javax.sql***.
- Clasa DriverManager este folosită pentru a crea conexiunile JDBC.
- Conexiunile JDBC suportă crearea și executarea comenziilor SQL. Acestea pot fi comenzi de manipulare a datelor (*Data Manipulation Language -DML*) precum *INSERT, UPDATE, DELETE, SELECT* (manipulează date read-only) sau pot fi comenzi destinate definirii structurilor de stocare a datelor (*Data Definition Language - DDL*) precum *CREATE TABLE, ALTER TABLE, DROP TABLE, etc.* JDBC execută comenziile sql cu ajutorul metodelor oferite de una din următoarele interfețe:

- Statement* – este folosită pentru trimiterea comenziilor *SQL* simple fără parametri
  - PreparedStatement* – permite folosirea instrucțiunilor *SQL* precompilate și a parametrilor de intrare în interogări.
  - CallableStatement* – permite executarea unor proceduri stocate pe baza de date.
- Comenziile *SQL* precum *INSERT*, *UPDATE* și *DELETE* returnează un contor care reprezintă numărul de rânduri care au fost afectate de comandă
- Interogările *SQL* (*SELECT*) returnează un obiect *JDBC*, de tip ***ResultSet***, care este organizat pe rânduri și coloane. Coloanele pot să fie accesate fie cu ajutorul numelui, fie cu ajutorul indexului coloanei. Un *ResultSet* conține metadate care descriu numele coloanelor și tipul acestora

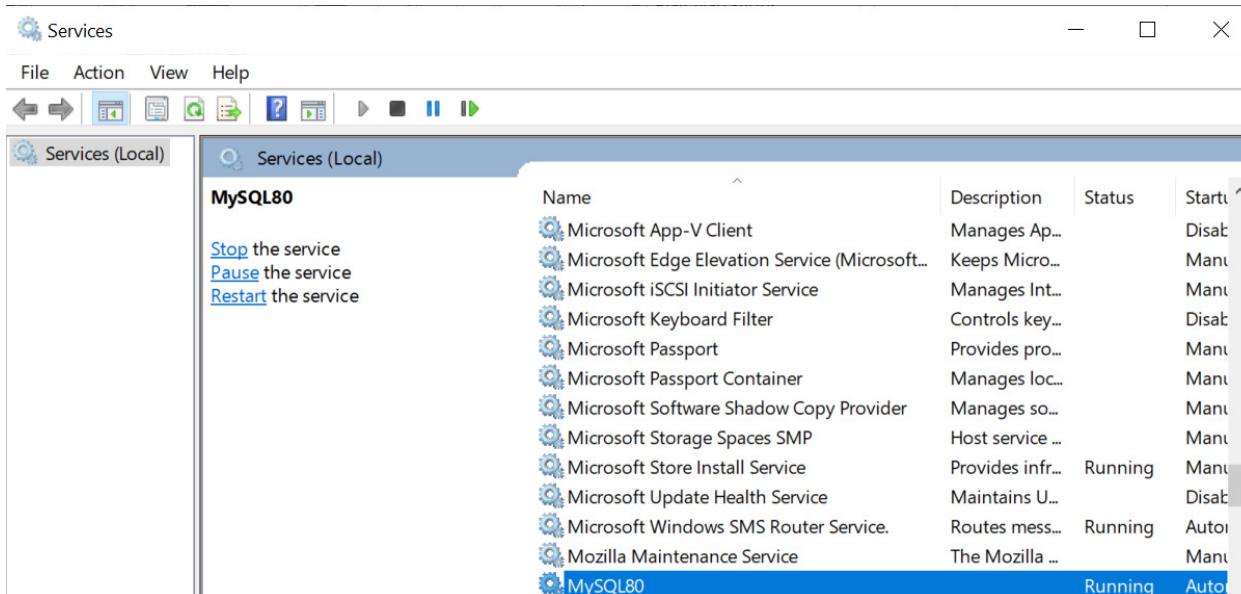
## 5.2 Caracteristicile și utilizarea MySQL

- **Caracteristici**

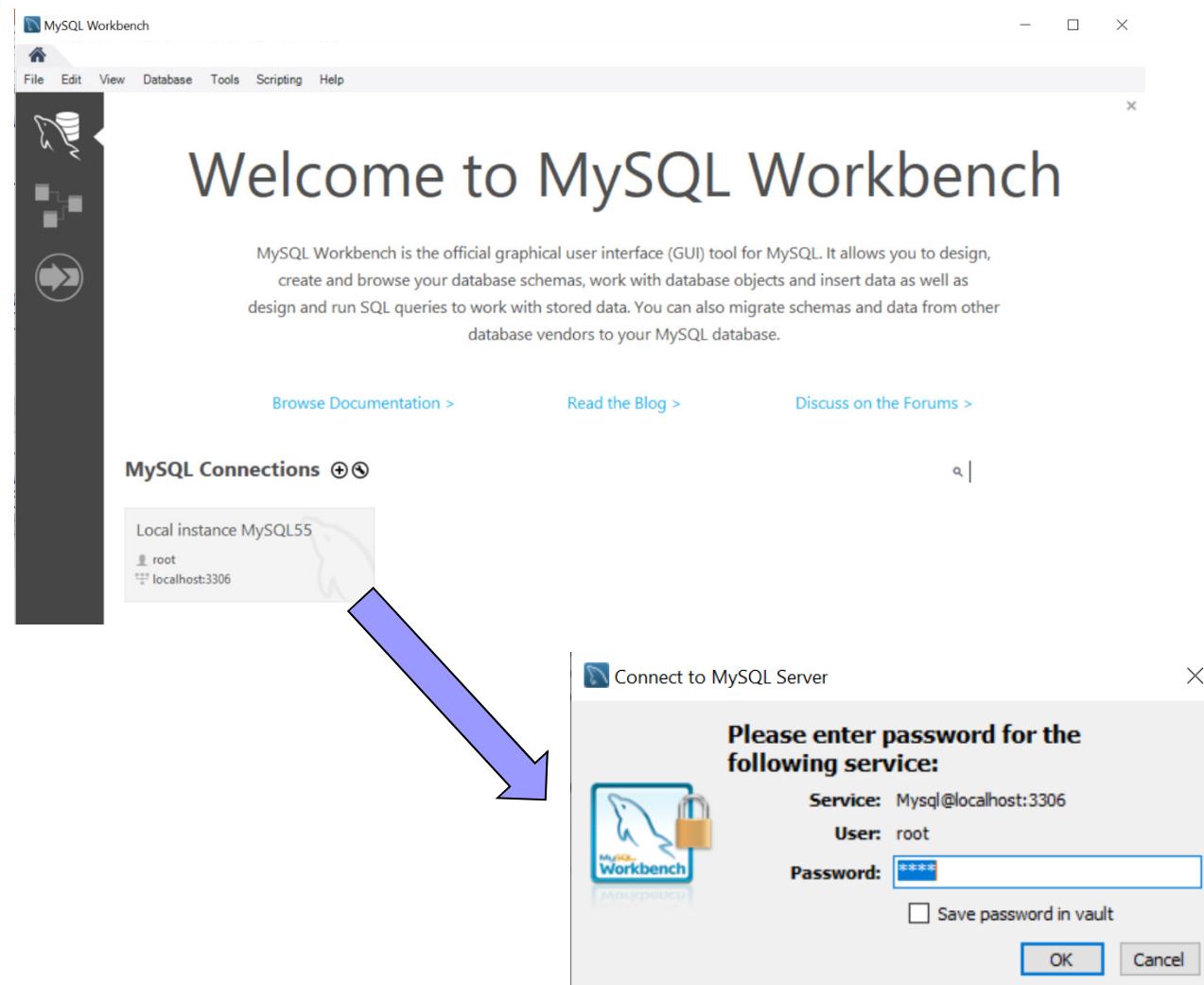
- Unul din cele mai populare SGBD-uri (Sisteme de Gestire a Bazelor de Date) din lume
  - Scris în C și C++
  - Testat cu o arie largă de compilatoare
  - Funcționează pe multe platforme (*Linux*, *Mac OS*, *Windows*, *Solaris*, etc)

- Testat cu *Purify* (un instrument software care ajută la buna gestiune a memoriei)
  - Utilizează un mod de proiectare cu module independente
  - Proiectat astfel încât să poată fi rulat cu ușurință pe mai multe procesoare
  - Furnizează motoare de stocare tranzacționale sau ne-tranzacționale
  - Folosește un sistem de alocare bazat pe fire de execuție foarte rapid
  - Execută foarte rapid comenzi de tip *join*
  - Implementează tabele *hash*, care sunt folosite ca tabele temporare
  - Server-ul poate fi tratat ca un program separat și poate fi utilizat în aplicații de tip client-server în rețea, dar poate fi tratat și ca o librărie care poate fi încorporată în aplicații de sine stătătoare. Aceste aplicații pot fi utilizate în medii în care nu este disponibilă nici o rețea
- 
- **Descărcare și instalare**
    - <http://dev.mysql.com/downloads/installer/>

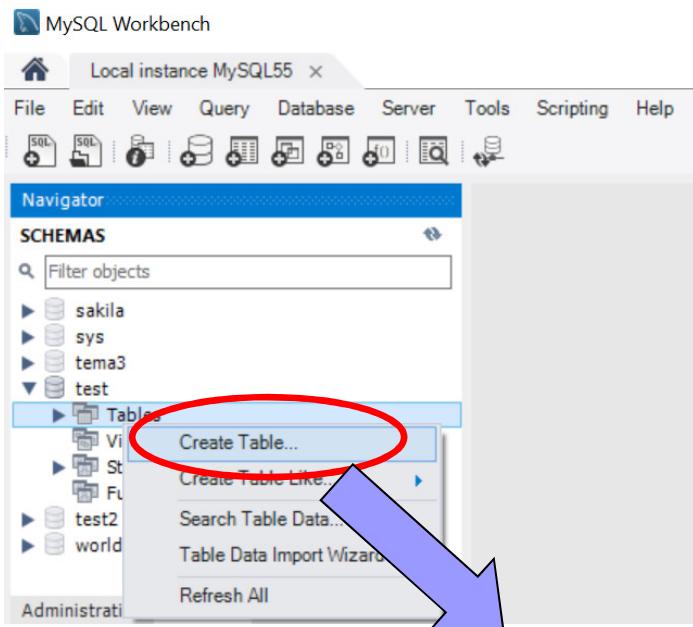
- La instalare alegeți opțiunea *Default* care instalează următoarele instrumente
  - *MySQL Server*
  - *MySQL Workbench* – este un instrument vizual care permite administrarea bazelor de date
  - Conectorii care permite accesarea bazelor de date *MySQL* din *Java*, *C*, *C++*, *.Net*, *ODBC*, *Excel*
  - *MySQL Notifier* care permite monitorizarea și schimbarea stării serverului *MySQL*, cu ajutorul unui indicator care este plasat în *system tray*
  - Documentație si exemple
- **Utilizarea**
  - În sistemul de operare a fost instalat un serviciu care se lansează automat atunci când pornim calculatorul. Acest serviciu pornește serverul *MySQL*.
  - Serverul pornit ascultă și acceptă, pe bază de utilizator și parolă, cererile de conectare



- Conectarea la server, crearea bazelor de date (schemelor) și a tabelelor, popularea acestora cu date, etc se realizează cu ajutorul instrumentului *MySQL Workbench*
- Înțâi se realizează conectarea la serverul MySQL cu ajutorul parolei stabilite la instalare



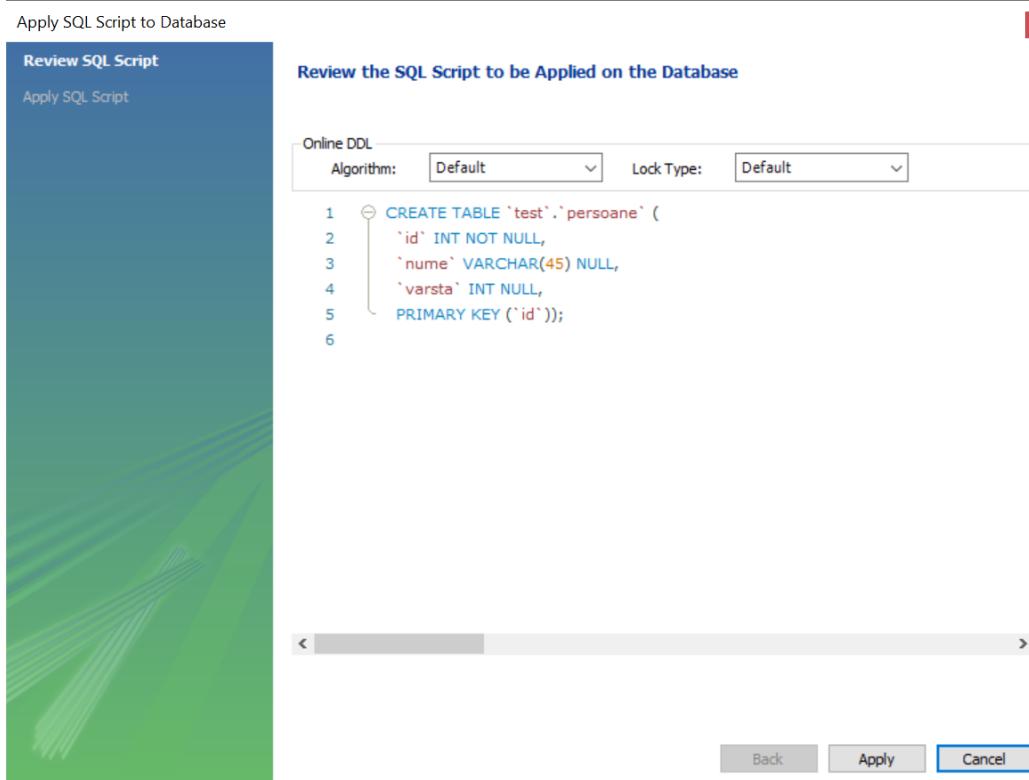
- O bază de date poate conține tabele, vederi, proceduri stocate sau funcții
- Se deschide baza de date în care se dorește să se creeze tabelul (sau se creează o nouă bază de date)
- Se face click dreapta pe secțiunea Tables și se alege comanda Create Table



The screenshot shows the MySQL Workbench interface with the 'Local instance MySQL55' connection selected. In the Navigator pane, under the 'SCHEMAS' section, the 'test' schema is selected. In the main workspace, a 'Table Name' dialog is open, showing 'persoane' as the table name and 'test' as the schema. The dialog displays the structure of the 'persoane' table:

Column Name	Datatype	PK	NN	UQ	B	UN	ZF	AI	G	Default/Expr
id	INT	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>						
nume	VARCHAR(45)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
varsta	INT	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	

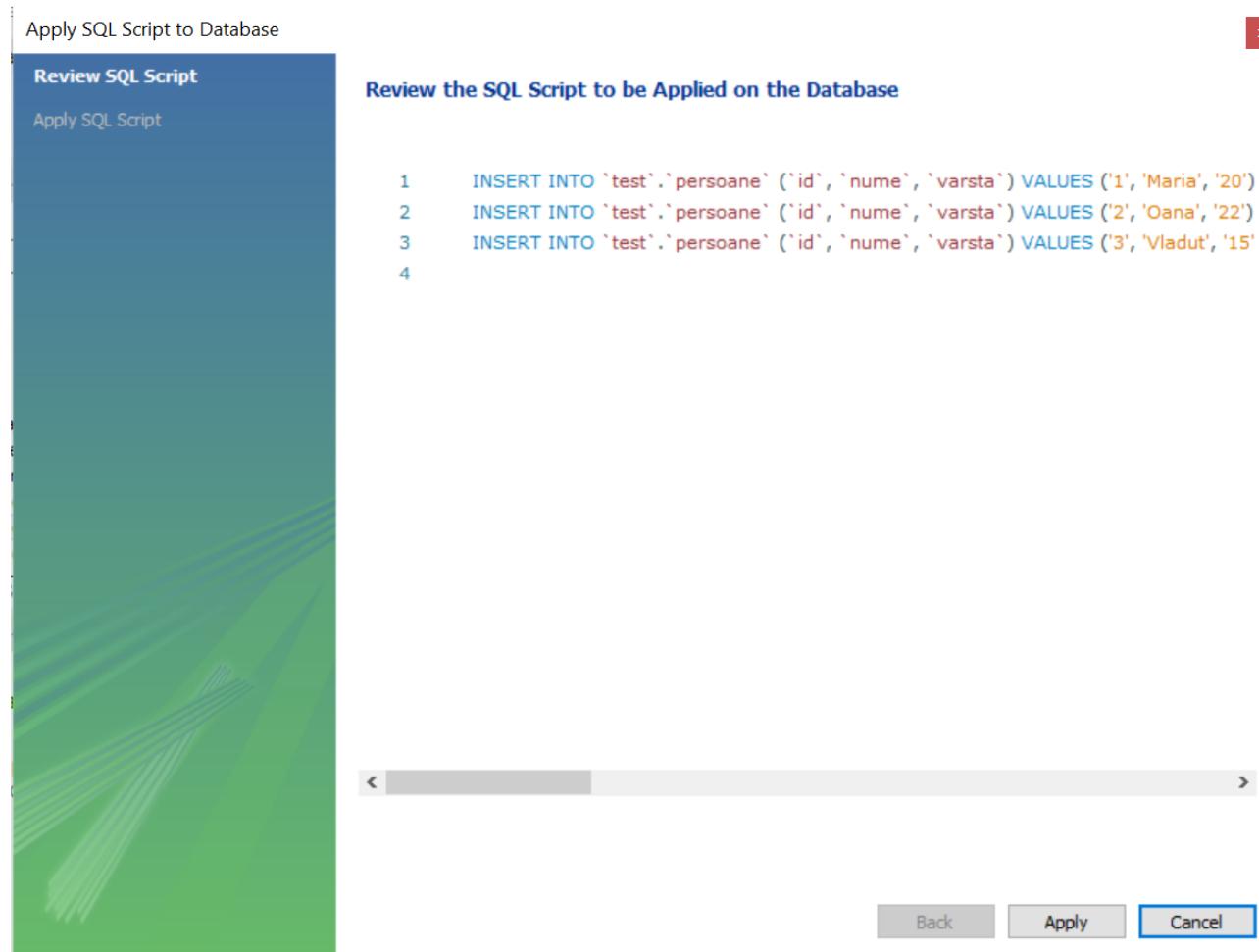
- Se generează automat comanda SQL de creare a tabelului în conformitate cu cele menționate în interfață grafică



	id	nume	varsta
1	Maria	20	
2	Oana	22	
3	Vladut	15	
*	NULL	NULL	NULL

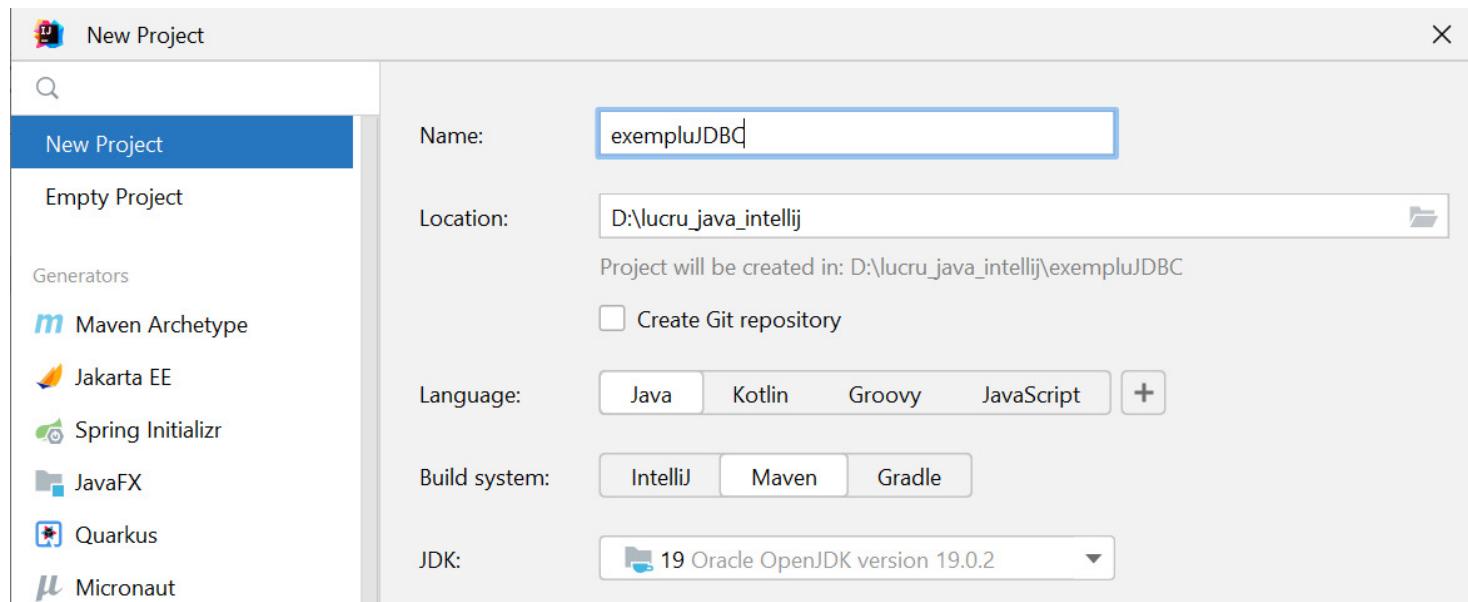
- Popularea cu date a tabelului creat se face prin click dreapta pe numele tabelului și alegând comanda *Select Rows – Limit 1000*

- După introducerea datelor se apasă comanda Apply, în urma căreia se generează automat comenzi SQL - INSERT pentru introducerea datelor în tabel

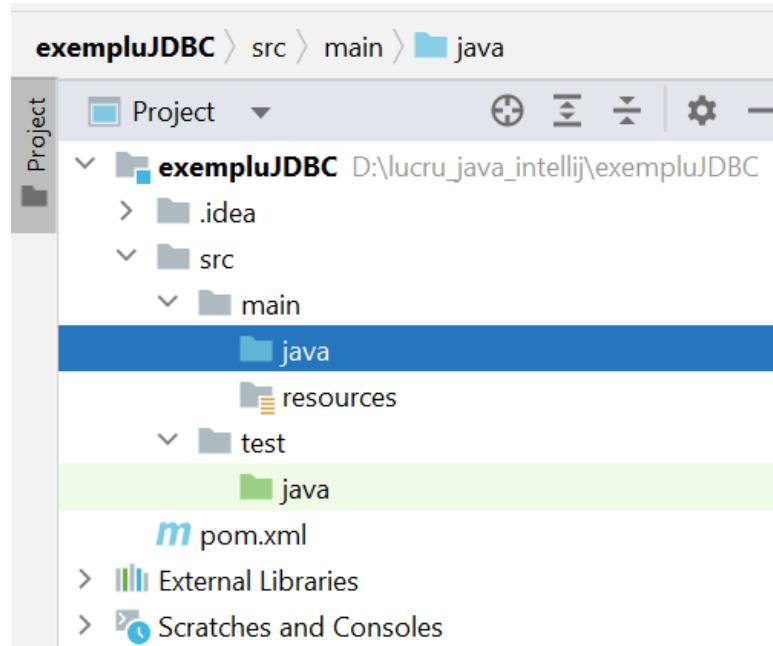


## 5.3 Crearea unui proiect Maven care utilizează JDBC

- Apache Maven este un instrument care gestionează toate fazele de construcție ale proiectelor software. De asemenea el permite crearea unor proiecte pe baza unor modele de proiecte (arhetipuri) și gestionează dependențele proiectelor software (bibliotecile pe care acestea le utilizează).
- Un proiect Maven primește instrucțiunile cu privire la dependențele necesare pentru rularea lui, instrucțiunile de compilare, build, raportare și documentare prin unul sau mai multe fișiere XML numite *pom.xml* (*Project Object Model*).
- Crearea unui proiect Maven în IntelliJ se realizează alegând comanda *File > New Project*. Se introduce numele proiectului și se alege sistemul de build Maven



- Proiectul realizat are următoarea structură:



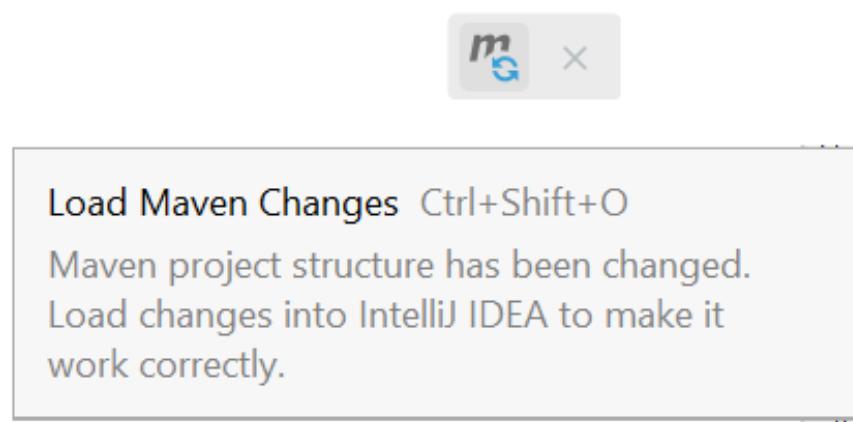
- În directorul *src/main/java* se introduce codul sursă al proiectului.
- În directorul *src/main/resources* se introduc resursele proiectului, cum ar fi imagini, fișiere *JSON* sau *XML* de intrare.
- În *src/test/java* se scriu unități de testare, de exemplu *JUNIT*.
- Fisierul *pom.xml* (*Project Object Model*) este fișierul utilizat de *Maven* pentru configurarea proiectului.

- În fișierul *pom.xml* a fost generat tag-ul `<properties>` cu subtagurile `<maven.compiler.source>` și `<maven.compiler.target>` care permit modificarea versiunii de Java utilizată de proiect.
- De asemenea au fost generate tag-urile:
  - `groupId` – se completează cu numele organizației sau echipei care dezvoltă proiectul
  - `artifactId` - reprezintă numele proiectului. În cadrul organizației sau echipei artifactId ar trebui să reprezinte în mod unic un proiect specific
  - `version 0.0.1 SNAPSHOT` - reprezintă un proiect în dezvoltare
- Dependențele (artefactele) de care are nevoie proiectul pentru a rula trebuie introduse în fișierul *pom.xml* în interiorul tag-ului `<dependencies></dependencies>`
- Pentru fiecare artefact trebuie introdus id-ul grupului, id-ul artefactului și versiunea. Determinarea acestor informații se face accesând depozitul *Maven central*: <https://mvnrepository.com> și căutând după artefactul dorit.
- Într-un proiect *Maven* care lucrează cu o bază de date *MySQL* folosind tehnologia *JDBC (Java Database Connectivity)* este necesar să se introducă o dependență față de connectorul *MySQL* (driverul de conectare la baza de date *MySQL*)
- Această dependență se obține din depozitul *Maven central* <https://mvnrepository.com> căutând după “mysql connector”. Din lista rezultată în urma căutării se alege conectorul corespunzător versiunii de *MySQL* instalată (pe calculatoarele din laborator aceasta este 8.0.19) și se copiază tag-ul `dependency` care conține tagurile `groupId`, `artifactId` și `version`

- În fișierul *pom.xml*, în interiorul tagului *project* se completează tagul *dependencies* și în acesta tagul *dependency*

```
<dependencies>
    <!-- https://mvnrepository.com/artifact/mysql/mysql-connector-java -->
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>8.0.19</version>
    </dependency>
</dependencies>
```

- În *IntelliJ*, după ce se modifică și se salvează fișierul *pom.xml*, trebuie dată comanda **Load Maven Changes (Ctrl + Shift + O)**. Dependentele aduse în proiect pot să fie vizualizate în secțiunea *External Libraries*.



- În urma acestei comenzi dependențele completeate în fișierul *pom.xml*, se aduc în proiect din depozitul *Maven* local care se găsește în c:\users\nume\_user\.m2\repository. Dacă dependențele nu se regăsesc în depozitul *Maven* local atunci se vor descărca automat din depozitul central în cel local și apoi se vor aduce în proiect.
- Dacă în fișierul *pom.xml* se pun în comentariu sau se șterg rândurile care solicită dependențele și apoi se salvează fișierul *pom.xml* și se actualizează proiectul, se observă ca acestea sunt eliminate din proiect.
- În **Eclipse**, după ce se modifică și se salvează fișierul *pom.xml*, se dă click dreapta pe proiect și se rulează comanda *Maven > Update project (Alt + F5)*. Dependentele aduse în proiect pot să fie vizualizate în secțiunea *Maven Dependencies*.

## 5.4 Preluarea datelor din baza de date

- Presupune parcurgerea următorilor pași:
  - Stabilirea unei conexiuni către baza de date
  - Execuția interogărilor *SQL*
  - Procesarea rezultatelor
  - Închiderea conexiunii cu baza de date

### 5.4.1 Stabilirea unei conexiuni către baza de date

- Odată ce s-a încărcat un driver putem să-l folosim pentru stabilirea unei conexiuni către baza de date. O conexiune *JDBC* este identificată printr-un *URL JDBC* specific
- Sintaxa standard pentru *URL*-ul unei baze de date este cea de mai jos. *Driver* este un nume de driver valid, *server* este numele sau adresa *IP* a calculatorului care găzduiește serverul *MySQL*, *port* este numărul unui port deschis pe care se poate face conectarea

```
jdbc:driver://server:port/baza_de_date
```

- Pentru baza de date *test*, în care a fost creat tabelul *persoane*, din exemplul precedent *URL*-ul este următorul

```
String url = "jdbc:mysql://localhost:3306/test";
```

- Pentru stabilirea unei conexiuni se folosește metoda statică `getConnection()` din clasa `DriverManager`

```
Connection connection = DriverManager.getConnection (url, "root", "root");
```

#### 5.4.2 Execuția interogărilor SQL

- După ce s-a stabilit conexiunea cu baza de date se pot trimite comenzi SQL. Aceste comenzi SQL pot fi comenzi *DML (Data Manipulation Language)* sau *DDL (Data Definition Language)*, cu sau fără parametri. În cadrul acestui subcapitol se va discuta despre rularea unor interogări simple fără parametri care aduc date din baza de date în program
- execuția instrucțiunilor SQL neparametrizate se realizează cu ajutorul unui obiect de tip `Statement` care se instanțiază prin intermediul metodei `createStatement()` din interfața `Connection`

```
Statement statement;  
statement = con.createStatement();
```

- Metoda `executeQuery()` din interfața `Statement` este utilizată pentru rularea interogărilor (*SELECT*) care returnează o mulțime rezultat
- Pentru a prelua datele din tabelul persoane executăm o interogare *SELECT* și preluăm datele într-un obiect de tip `ResultSet`

```
ResultSet rs;  
rs = statement.executeQuery("select * from persoane");
```

### 5.4.3 Procesarea rezultatelor

- Datele preluate din baza de date se găsesc sub formă tabelară într-un obiect de tip *ResultSet*. Acesta conține rezultatul unei interogări
- Parcurea înregistrărilor din *ResultSet* se realizează cu ajutorul cursorului. Acesta este poziționat inițial înaintea primei linii
- Metodele *first()*, *previous()*, *next()*, *last()*, *absolute()* permit deplasarea cursorului
- Metodele *getInt()*, *getString()*, *getBoolean()*, etc permit obținerea valorilor câmpurilor specificate ca și parametru de pe rândul indicat de cursor. Parametrul de intrare al acestor metode poate să fie denumirea câmpului sau index-ul acestuia (primul câmp are index-ul 1)
- Pentru afișarea pe ecran a datelor din tabelul *persoane* se poate utiliza codul de mai jos

```
while (rs.next())
    System.out.println("id=" + rs.getInt("Id") + ", nume= " + rs.getString("nume") + ",  

                       varsta=" + rs.getInt("varsta"));
```

- Determinarea liniei curente spre care indică cursorul se poate face cu ajutorul metodei *getRow()*
- Pentru a determina dacă cursorul este la sfârșit sau început se pot utiliza metode: *isFirst()*, *isLast()*, *isBeforeFirst()*, *isAfterLast()*

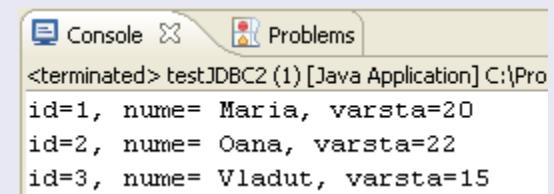
#### 5.4.4 Închiderea unei conexiuni la baza de date

- Colectorul de reziduuri nu știe dacă este cazul sau nu să elibereze resursele exterioare, de aceea este bine ca programatorii să închidă explicit conexiunile către baza de date

```
connection.close();  
statement.close();  
rs.close();
```

- **Exemplu**, preluarea și afișarea în program a datelor din tabelul *persoane*:

```
package capitolul6.exemplul1;  
  
import java.sql.Connection;  
import java.sql.DriverManager;  
import java.sql.ResultSet;  
import java.sql.SQLException;  
import java.sql.Statement;  
  
class MainApp {  
    public static void main(String[] args) throws SQLException {  
        String url = "jdbc:mysql://localhost:3306/test";  
        String sql="select * from persoane";  
  
        Connection connection= DriverManager.getConnection(url, "root", "root");  
        Statement statement = connection.createStatement();  
        ResultSet rs = statement.executeQuery(sql);  
        while (rs.next())  
            System.out.println("id=" + rs.getInt("Id") + ", nume= "  
                + rs.getString("nume") + ", varsta=" + rs.getInt("varsta"));  
    }  
}
```



```
        connection.close();
        statement.close();
        rs.close();
    }
}
```

## 5.5 Rularea comenziilor SQL cu parametri

- Interfața *PreparedStatement* permite rularea comenziilor *SQL* cu parametri. Comanda *SQL* se transmite metodei *prepareStatement()* din interfața *Connection*. Metoda returnează un obiect de tip *PreparedStatement*.
- Stabilirea valorilor parametrilor se face cu ajutorul unor metode precum *setString()*, *setInt()* care primesc ca și parametri indexul parametrului care va fi setat și valoarea acestuia
- Exemplul următor, extrage date din tabelul persoane, filtrând doar persoanele care au un nume precizat și o vârstă mai mică decât o valoare precizată. Exemplul rulează o interogare *select* cu parametri

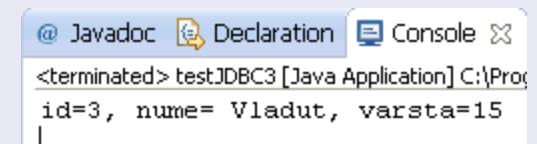
```
package capitolul6.exemplul2;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
```

```

class MainApp {
    public static void main(String[] args) {
        String url = "jdbc:mysql://localhost:3306/test";
        String sql="select * from persoane where nume=? and varsta<?";
        try {
            Connection connection = DriverManager.getConnection (url, "root", "root");
            PreparedStatement ps=connection.prepareStatement(sql);
            ps.setString(1, "Vladut");
            ps.setInt(2, 18);
            ResultSet rs = ps.executeQuery();
            while (rs.next())
                System.out.println("id="+rs.getInt(1)+", nume= " + rs.getString(2)
                    + ", varsta="+rs.getInt(3));
            connection.close();
            ps.close();
            rs.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}

```



## 5.6 Actualizarea bazei de date cu ajutorul comenziilor SQL

- Următorul exemplu arată cum pot fi rulate comenzi *DML* (*Data Manipulation Language*) pentru a adăuga, modifica și șterge date în/din tabelul personae. Exemplul rulează comenzi *SQL* cu parametri cu ajutorul obiectului *PreparedStatement*

```
package capitolul6.exemplul3;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

class MainApp {
    public static void afisare_tabela(Statement statement, String mesaj)  {
        String sql="select * from persoane";
        System.out.println("\n---"+mesaj+"---");
        try(ResultSet rs=statement.executeQuery(sql)) {
            while (rs.next())
                System.out.println("id=" + rs.getInt(1) + ", nume=" + rs.getString(2) + ", varsta="
                    + rs.getInt(3));
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

```
public static void adaugare(Connection connection, int id, String nume, int varsta) {  
    String sql="insert into persoane values (?,?,?,?)";  
    try(PreparedStatement ps=connection.prepareStatement(sql)) {  
        ps.setInt(1, id);  
        ps.setString(2, nume);  
        ps.setInt(3, varsta);  
  
        int nr_randuri=ps.executeUpdate();  
        System.out.println("\nNumar randuri afectate de adaugare="+nr_randuri);  
    } catch (SQLException e) {  
        System.out.println(sql);  
        e.printStackTrace();  
    }  
}  
  
public static void actualizare(Connection connection,int id,int varsta){  
    String sql="update persoane set varsta=? where id=?";  
    try(PreparedStatement ps=connection.prepareStatement(sql)) {  
        ps.setInt(1, varsta);  
        ps.setInt(2, id);  
        int nr_randuri=ps.executeUpdate();  
        System.out.println("\nNumar randuri afectate de modificare="+nr_randuri);  
    } catch (SQLException e) {  
        System.out.println(sql);  
        e.printStackTrace();  
    }  
}
```

```
public static void stergere(Connection connection,int id){
    String sql="delete from persoane where id=?";
    try(PreparedStatement ps=connection.prepareStatement(sql)) {
        ps.setInt(1, id);
        int nr_randuri=ps.executeUpdate();
        System.out.println("\nNumar randuri afectate de modificare="+nr_randuri);
    }
    catch (SQLException e) {
        System.out.println(sql);
        e.printStackTrace();
    }
}

public static void main(String[] args){
    String url = "jdbc:mysql://localhost:3306/test";

    try {
        Connection connection = DriverManager.getConnection(url, "root", "root");
        Statement statement = connection.createStatement();

        afisare_tabela(statement, "Continut initial");

        adaugare(connection,4,"Dana",23);
        afisare_tabela(statement, "Dupa adaugare");

        actualizare(connection,4,24);
        afisare_tabela(statement, "Dupa modificare");
    }
}
```

```

        stergere(connection,4);
        afisare_tabela(statement, "Dupa stergere");

        statement.close();
        connection.close();
    } catch (SQLException e) {
        e.printStackTrace();
    }
}
}

```

```

---Continut initial---
id=1, nume=Maria,varsta=20
id=2, nume=Oana,varsta=22
id=3, nume=Vladut,varsta=15

Numar randuri afectate de adaugare=1

---Dupa adaugare---
id=1, nume=Maria,varsta=20
id=2, nume=Oana,varsta=22
id=3, nume=Vladut,varsta=15
id=4, nume=Dana,varsta=23

Numar randuri afectate de modificare=1

---Dupa modificare---
id=1, nume=Maria,varsta=20
id=2, nume=Oana,varsta=22
id=3, nume=Vladut,varsta=15
id=4, nume=Dana,varsta=24

Numar randuri afectate de modificare=1

---Dupa stergere---
id=1, nume=Maria,varsta=20
id=2, nume=Oana,varsta=22
id=3, nume=Vladut,varsta=15

```

- Interfețele *Statement*, *PreparedStatement*, *Connection*, *ResultSet* implementează interfața *AutoCloseable*, putând fi folosite în blocuri try cu resurse, pentru ca resursa să fie închisă automat
- În blocurile catch se afișează pe lângă mesajul exceptiei și comanda sql care a determinat-o
- Metoda *executeUpdate()* este utilizată pentru a rula comenzi SQL insert, update și delete. Metoda returnează numărul de rânduri afectate de comanda SQL. Comenzi update și delete pot afecta de la nici un rând până la toate rândurile din tabelă, în funcție de condiția din clauza *where*
- Valorile efective ale parametrilor au fost stabilite prin metodele *setInt(nr\_parametru, valoare)* sau *setString(nr\_parametru, valoare)*
- Accesarea coloanelor din *ResultSet* s-a făcut de această dată precizând index-ul coloanei, nu denumirea acesteia

## 5.7 Actualizarea bazei de date cu ajutorul actualizărilor programatice

- Baza de date se poate actualiza prin rularea unor interogări SQL de tip *INSERT*, *UPDATE*, *DELETE* (precum în subcapitolul precedent) dar și cu ajutorul unor **actualizări programatice**.
- **Actualizările programatice** sunt actualizări aplicate direct *ResultSet-ului* care sunt automat efectuate și asupra bazei de date.
- Actualizările programatice nu se pot executa decât dacă obiectul de tip *Statement* a fost creat cu parametrii de mai jos

```
Statement statement = connection.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,  
                                              ResultSet.CONCUR_UPDATABLE);
```

- Constanta *TYPE\_SCROLL\_SENSITIVE* determină crearea unui obiect *ResultSet* care este explorabil (scrollable) și în general sensitiv la schimbările datelor care privesc *ResultSet*-ul
- Constanta *CONCUR\_UPDATABLE* –indică modul de concurență pentru un obiect *ResultSet* care poate fi actualizat
- Actualizări programatice pentru inserarea unui rând:

```
rs.moveToInsertRow(); //se muta cursorul pe un rand nou  
rs.updateInt("nume_coloana", valoare_coloana); //se introduc datele  
rs.updateString(numar_colonan, valoare_colonan);  
rs.insertRow(); //se introduce randul in BD
```

- Actualizări programatice pentru modificarea datelor:

```
rs.first(); //se pozitioneaza cursorul pe randul de modificat  
rs.updateInt("nume_coloana", noua_valoare);  
rs.updateString(numar_coloana, noua_valoare);  
rs.updateRow(); //se actualizeaza randul in BD
```

- Actualizări programatice pentru ștergerea unui rând:

```
rs.first(); //se pozitioneaza cursorul pe randul de sters  
rs.deleteRow(); //se sterge randul din rs si din BD
```

- Următorul exemplu realizează operațiile elementare adăugare, modificare și stergere asupra tabelei *persoane* lucrând cu actualizări programatice

```
package capitolul6.exemplul4;  
  
import java.sql.Connection;  
import java.sql.DriverManager;  
import java.sql.ResultSet;  
import java.sql.SQLException;  
import java.sql.Statement;  
  
class MainApp {  
    public static void afisare_tabela(ResultSet rs, String mesaj) {  
        System.out.println("\n---"+mesaj+"---");  
        try {  
            rs.beforeFirst();
```

```

        while (rs.next())
            System.out.println("id=" + rs.getInt(1) + ", nume=" + rs.getString(2) + ", varsta="
                + rs.getInt(3));
    } catch (SQLException e) {
        e.printStackTrace();
    }
}

public static void adaugare(ResultSet rs, int id, String nume, int varsta) {
    try {
        rs.moveToInsertRow();
        rs.updateInt("id", id);
        rs.updateString("nume", nume);
        rs.updateInt("varsta", varsta);
        rs.insertRow();
    } catch (SQLException e) {
        e.printStackTrace();
    }
}

public static void actualizare(ResultSet rs,int id,int varsta){
    boolean modificat=false;
    try {
        rs.beforeFirst();
        while (rs.next())
            if(rs.getInt("id")==id) {
                rs.updateInt("varsta", varsta);
                rs.updateRow();
                modificat=true;
                break;
            }
    }
}

```

```

    if(modificat)
        System.out.println("\nVarsta persoanei "+rs.getString("nume")
            +" a fost actualizata cu succes!");
    else
        System.out.println("Nu se gaseste nici o persoana cu id-ul specificat");
} catch (SQLException e) {
    e.printStackTrace();
}
}

public static void stergere(ResultSet rs,int id){
    boolean sters=true;
    try {
        rs.beforeFirst();
        while (rs.next())
            if(rs.getInt("id")==id) {
                rs.deleteRow();
                sters=true;
                break;
            }
        if(sters)
            System.out.println("\nPersoana cu id-ul "+id+" a fost stearsa cu succes!");
        else
            System.out.println("Nu se gaseste nici o persoana cu id-ul specificat");
    } catch (SQLException e) {
        e.printStackTrace();
    }
}

```

```

public static void main(String[] args){
    String url = "jdbc:mysql://localhost:3306/test";
    String sql="select * from persoane";
    try {
        Connection connection = DriverManager.getConnection(url, "root", "root");
        Statement statement = connection.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                                                          ResultSet.CONCUR_UPDATABLE);

        ResultSet rs=statement.executeQuery(sql);
        afisare_tabela(rs, "Continut initial");

        adaugare(rs,4,"Dana",23);
        afisare_tabela(rs, "Dupa adaugare");

        actualizare(rs,4,24);
        afisare_tabela(rs, "Dupa modificare");

        stergere(rs,4);
        afisare_tabela(rs, "Dupa stergere");
        statement.close();
        connection.close();
    } catch (SQLException e) {
        e.printStackTrace();
    }
}

```

```

---Continut initial---
id=1, nume=Maria,varsta=20
id=2, nume=Oana,varsta=22
id=3, nume=Vladut,varsta=15

---Dupa adaugare---
id=1, nume=Maria,varsta=20
id=2, nume=Oana,varsta=22
id=3, nume=Vladut,varsta=15
id=4, nume=Dana,varsta=23

Varsta persoanei Dana a fost actualizata cu succes!

---Dupa modificare---
id=1, nume=Maria,varsta=20
id=2, nume=Oana,varsta=22
id=3, nume=Vladut,varsta=15
id=4, nume=Dana,varsta=24

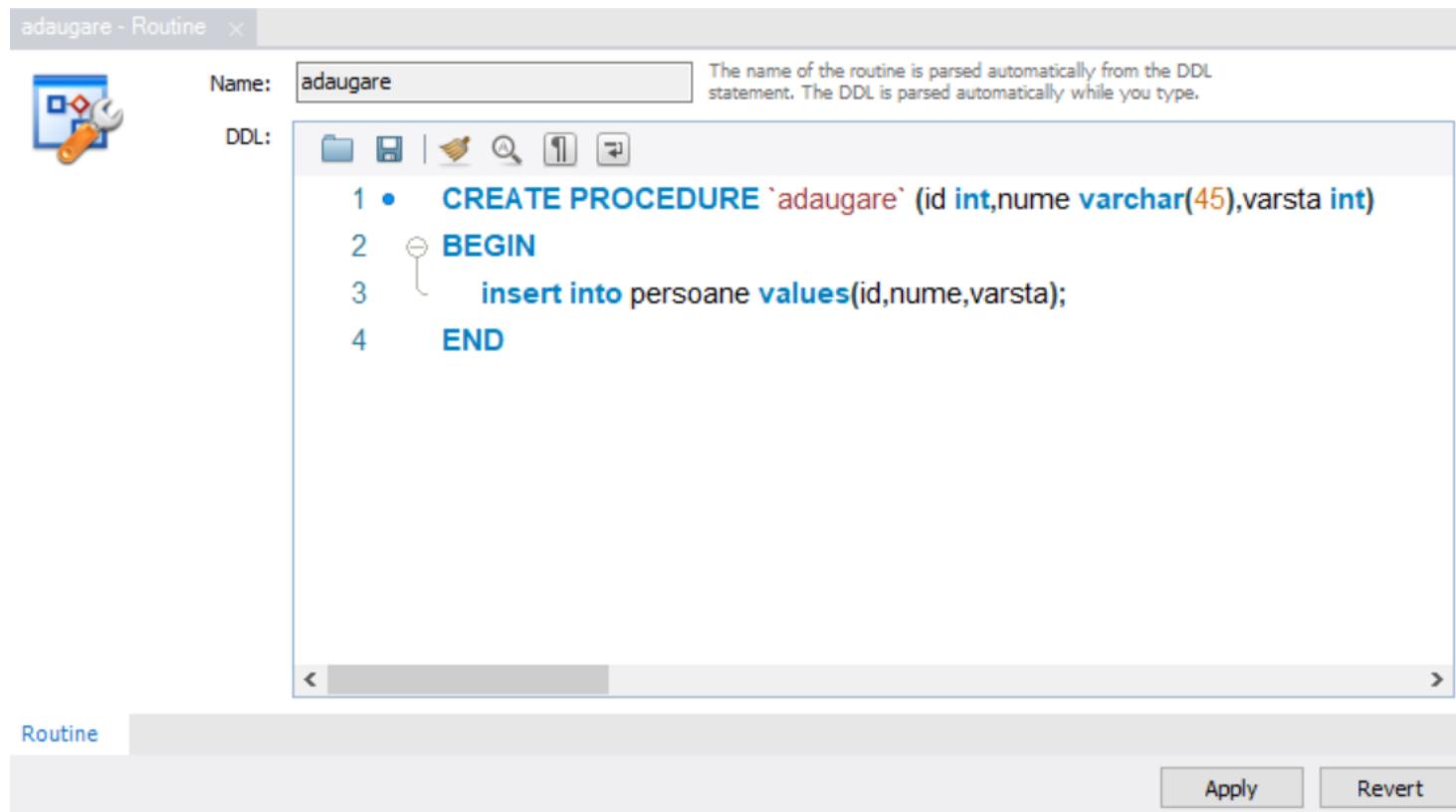
Persoana cu id-ul 4 a fost stearsa cu succes!

---Dupa stergere---
id=1, nume=Maria,varsta=20
id=2, nume=Oana,varsta=22
id=3, nume=Vladut,varsta=15

```

## 5.8 Apelul procedurilor stocate

- Cu ajutorul utilitarului **MySQL Workbench** se creează o procedură stocată pe server, în baza de date **test**, tabela **persoane**, prin click dreptă pe **Stored Procedures** și apoi alegând comanda **Create Stored Procedure...**
- Procedura creată va permite adăugarea unei noi înregistrări în tabela persoane și va avea conținutul de mai jos



The screenshot shows the 'adaugare - Routine' editor window in MySQL Workbench. The 'Name:' field contains 'adaugare'. The 'DDL:' pane displays the following SQL code:

```
1 • CREATE PROCEDURE `adaugare` (id int, nume varchar(45), varsta int)
2   BEGIN
3     insert into persoane values(id, nume, varsta);
4   END
```

At the bottom, there are 'Routine' and 'Script' tabs, and buttons for 'Apply' and 'Revert'.

- Exemplul de mai jos arată cum se poate apela procedura stocată și afișează conținutului tablei:

```
package capitolul6.exemplul5;
import java.sql.CallableStatement;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

class MainApp {
    public static void main(String[]args) {
        String url = "jdbc:mysql://localhost:3306/test";
        try {
            Connection connection = DriverManager.getConnection (url, "root", "root");

            CallableStatement cs=connection.prepareCall("{call adaugare(?, ?, ?)}");
            cs.setInt(1,4);
            cs.setString(2, "Dana");
            cs.setInt(3,23);
            cs.execute();
            cs.close();

            Statement statement = connection.createStatement();
            ResultSet rs = statement.executeQuery("select * from persoane");
            while (rs.next())
                System.out.println("id=" + rs.getInt(1) + ", nume= " + rs.getString(2)
                    + ", varsta=" + rs.getInt(3));
        }
    }
}
```

```
        connection.close();
        statement.close();
        rs.close();
    } catch (SQLException e) {
        e.printStackTrace();
    }
}
```

```
id=1, nume= Maria, varsta=20
id=2, nume= Oana, varsta=22
id=3, nume= Vladut, varsta=15
id=4, nume= Dana, varsta=23
```

## 5.9 Gestiunea tabelelor (creare, stergere, modificare)

- Comenzile *DDL* (*Data Definition Language*) cum sunt *create table*, *drop table*, *alter table* pot fi rulate cu ajutorul unui obiect *Statement* și a metodei *executeUpdate()*.
- Metoda *executeUpdate()* permite executarea unor comenzi *DML* (*Data Manipulation Language*) cum sunt *insert*, *update* sau *delete* în acest caz metoda returnează numărul de rânduri afectate de comandă și de asemenea permite executarea unor comenzi *DDL* (*Data Definition Language*) în acest caz returnează valoarea 0
- Exemplul următor afișează toate tabelele din baza de date *test* (rulând interogarea *show tables*), apoi șterge tabela *persoane* dacă aceasta există, creează tabela *persoane* cu câmpurile *id* (intreg, cheie primară), *nume* (*varchar(20)*) și *vârstă* (*intreg*), apoi modifică structura tabelei adăugând o nouă coloană (*email* – *varchar(50)*)

```
package capitolul6.exemplul6;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

class MainApp {
    public static void main(String[] args) throws SQLException {
        String url = "jdbc:mysql://localhost:3306/test";
        String sql="show tables";
```

```

Connection connection = DriverManager.getConnection(url, "root", "root");
Statement statement = connection.createStatement();

ResultSet rs = statement.executeQuery(sql);
System.out.println("---Lista tabelelor din BD test---");
while (rs.next())
    System.out.println(rs.getString(1));

statement.executeUpdate("drop table if exists persoane");

statement.executeUpdate("create table persoane (id integer primary key,"
+ "nume varchar(20), " + "varsta integer)");

statement.executeUpdate("alter table persoane add email varchar(50)");

statement.executeUpdate("insert into persoane values (1,'Oana',20,'oana@gmail.com')");
System.out.println("---Continutul tablei persoane---");
rs = statement.executeQuery("select * from persoane");

while (rs.next())
    System.out.println(rs.getInt(1)+", "+rs.getString(2)+", "+rs.getInt(3)+", "
    +rs.getString(4));

connection.close();
statement.close();
rs.close();
}
}

```

## 5.10 Determinarea denumirii coloanelor unei tabele și tipul lor

- În aplicații în care se lucrează cu diferite tabele care au o structură diferită, determinarea unor metadate cum ar fi numărul de coloane, denumirea lor, tipul etc se poate realiza cu ajutorul interfeței **ResultSetMetaData**:
- Exemplul de mai jos ilustrează cum pot fi determinate denumirile coloanelor, numărul lor, tipul și dimensiunea.

```
package capitolul6.exemplul7;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.sql.SQLException;
import java.sql.Statement;

class MainApp {
    public static void main(String[] args) throws SQLException {
        String url = "jdbc:mysql://localhost:3306/test";
        String sql="select * from persoane";

        Connection connection = DriverManager.getConnection(url, "root", "root");
        Statement statement = connection.createStatement();
        ResultSet rs = statement.executeQuery(sql);
```

```
ResultSetMetaData meta = rs.getMetaData();

for (int i=1;i<=meta.getColumnCount();i++){
    System.out.println(meta.getColumnName(i)
+ ", "+meta.getColumnTypeName(i)
+ ", "+meta getColumnDisplaySize(i));
}
connection.close();
statement.close();
rs.close();
}
}
```

```
id, INT, 10
nume, VARCHAR, 20
varsta, INT, 10
email, VARCHAR, 50
```

## 5.11 JDBC și Oracle

- Exemplul utilizează Oracle Database 11g Express Edition Release 11.2.0.2.0 - Production
- Se creează în *Oracle* o tabelă persoane (cu câmpurile id, nume și varsta, similară cu cea creată în *MySQL* și utilizată în exemplele din acest capitol). Tabela se creează folosind clientul *SQL Plus* și contul de student

Run SQL Command Line

```
SQL*Plus: Release 11.2.0.2.0 Production on Mi Noi 9 20:10:40 2022

Copyright (c) 1982, 2010, Oracle. All rights reserved.

SQL> conn student
Enter password:
Connected.
SQL> set autocommit on;
SQL>
SQL> create table persoane (id int primary key, nume varchar(20), varsta int);

Table created.

SQL>
SQL> insert into persoane values (1,'Maria',20);

1 row created.

Commit complete.
SQL> insert into persoane values (2,'Oana',22);

1 row created.

Commit complete.
SQL> insert into persoane values (3,'Vladut',15);

1 row created.

Commit complete.
SQL>
SQL> select * from persoane;

      ID  NUME          VARSTA
-----  -----
        1 Maria           20
        2 Oana            22
        3 Vladut          15

SQL>
```

- Se completează fișierul *pom.xml* cu dependența față de driver-ul de conectare la Oracle, obținută din depozitul *Maven* central. În *IntelliJ* se salvează fișierul *pom.xml* și se dă comanda *Load Maven Changes (Ctrl + Shift + O)*

```
<!-- https://mvnrepository.com/artifact/com.oracle.database.jdbc/ojdbc8 -->
<dependency>
    <groupId>com.oracle.database.jdbc</groupId>
    <artifactId>ojdbc8</artifactId>
    <version>12.2.0.1</version>
</dependency>
```

- Exemplul de mai jos preia datele din tabela persoane și le afișeză în consolă

```
package capitolul6.exemplul8;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

class MainApp {
    public static void main(String[] args) throws SQLException {
        String url="jdbc:oracle:thin:@localhost:1521:XE";
        String sql="select * from persoane";

        Connection connection = DriverManager.getConnection(url, "student", "student2");
        Statement statement = connection.createStatement();
```

```
ResultSet rs = statement.executeQuery(sql);
while (rs.next())
    System.out.println("id=" + rs.getInt("Id") + ", nume= " + rs.getString("nume")
    + ", varsta=" + rs.getInt("varsta"));

connection.close();
statement.close();
rs.close();
}

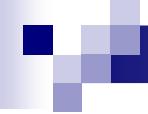
}
```



## **6. Dezvoltarea proiectelor web dinamice**

Sl. dr. ing. Raul Robu

2022-2023, Semestrul 2



# CUPRINS

- 6.1 Noțiuni generale
- 6.2 Structura de bază a unui servlet
- 6.3 Ciclul de viață al unui servlet
- 6.4 Clasa HttpServlet
- 6.5 Punerea în funcțiune a serverului web Apache Tomcat
- 6.6 Crearea unui proiect web dinamic
- 6.7 Rularea proiectului web
- 6.8 Annotarea @WebServlet
- 6.9 Utilizarea sesiunilor
- 6.10 Lucru cu obiectele de tip *ServletContext*
- 6.11 Utilizarea cookie-urilor în cadrul servelturilor
- 6.12 Servelturi și JDBC
- 6.13 Crearea și rularea fișierului war
- 6.14 Crearea unui proiect web Maven

## 6.1 Noțiuni generale

- IDE-ul utilizat în acest capitol este **Eclipse IDE for Enterprise Java and Web Developers**, care poate fi descărcat de pe link-ul  
<https://www.eclipse.org/downloads/packages/release/2022-06/r>
- Proiectele web dinamice care vor fi realizat vor conține alături de fișiere *html*, *xml* și *servleturi java*
- Un **Servlet** este o componentă *software* pe partea de server, scrisă în Java și care extinde funcționalitatea unui server (de obicei server *HTTP*)
- Un **applet** este o componentă *software* pe partea clientului care rulează în cadrul unui browser *Web*
- Spre deosebire de *applet-uri*, *servleturile* nu afișează o interfață grafică către utilizator ci returnează doar rezultatele către client (de obicei sub forma unui document *HTML*)
- *Servleturile* sunt clase *java* care se conformează unei interfețe specifice ce poate fi apelată de către server
- *Servleturile* furnizează un cadru pentru crearea de aplicații care implementează paradigma cerere – răspuns
- Când un *browser* trimite o cerere către *server*, *serverul* o trimite mai departe unui *servlet*. *Servletul* procesează cererea și construiește un răspuns (în *HTML*) care este returnat clientului
- Pachetele pentru *servleturi* sunt **javax.servlet** și **javax.servlet.http**

- Facilitățile servleturilor:
  - Construiesc dinamic și returnează un document *HTML* pe baza cererii clientului
  - Procesează datele complete de utilizatori în formularele *HTML* și furnizează un răspuns
  - Furnizează suport pentru autentificarea utilizatorilor și alte mecanisme de securitate
  - Interacționează cu resursele serverului cum ar fi baze de date, fișiere cu informații utile pentru clienți
  - Procesează intrările de la mai mulți clienți pentru aplicații cum ar fi jocurile în rețea
  - Permit serverului să comunice cu *appleturile* client printr-un protocol specific și păstrează conexiunea în timpul conversației
  - Atașează automat elemente de design pentru pagini *web*, cum ar fi antete sau note de subsol, pentru toate paginile returnate de server
  - Redirectează cererile de la un server la altul cu scop de echilibrare a încărcării
  - Partiționează un serviciu logic între *servleturi* sau între servere pentru a procesa eficient o problemă

## 6.2 Structura de bază a unui servlet

- Un servlet se poate defini în două moduri:
  - prin extinderea uneia din cele două clase de bază referitoare la servleturi: **GenericServlet** și **HttpServlet**
  - Implementarea interfeței **Servlet**
- Metoda care este apelată automat ca răspuns la cererea fiecărui client se numește *service()*. Această metodă poate fi suprascrisă pentru a furniza funcționarea dorită
- Servleturile care extind *HTTPServer* pot să nu suprascrie *service()*, pentru că implementarea implicită a acestei metode va apela automat una din metodele *doGet()* sau *doPost()* (în funcție de tipul cererii *HTTP*) pentru a răspunde cererii clientului
- Alte metode apelate implicit de către majoritatea servleturilor sunt *init()* și *destroy()*. Metoda *destroy* este apelată când servletul este descărcat eliberând resursele servletului

## 6.3 Ciclul de viață al unui servlet

- Procesul apelării de către server a unui servlet se poate împărții în următorii pași:
  - Serverul încarcă servletul când acesta este cerut de client sau la pornirea serverului, dacă aşa impune configurația. Servletul poate fi încărcat local sau dintr-o locație la distanță
  - Serverul creează o instanță a clasei servletului pentru deservirea tuturor cererilor
  - Serverul apelează metoda *init()* a servletului care se va executa complet înainte ca servletul să primească cereri
  - În momentul primirii unei cereri pentru un *servlet* serverul construiește un obiect de tip *ServletRequest* sau *HttpServletRequest* din datele introduse în cererea clientului. De asemenea acesta construiește un obiect de tip *ServletResponse* sau *HttpServletResponse* care furnizează metode pentru furnizarea răspunsului
  - Serverul apelează metoda *service ()* care pentru servleturi HTTP poate apela o metodă specifică cum ar fi *doGet()* sau *doPost()*
  - Metoda *service()* procesează cererea clientului prin evaluarea obiectului *ServletRequest* sau *HttpServletRequest*. Apoi acesta răspunde utilizând obiectul *ServletResponse* sau *HttpServletResponse*
  - Dacă serverul primește o nouă cerere pentru acest servlet procesul începe din nou de la apelul metodei *service()*
  - De fiecare dată când containerul servletului determină că un servlet trebuie descărcat, servletul apelează metoda *destroy*

## 6.4 Clasa HttpServlet

- Servleturile *HTTP*, cele care extind clasa *HTTP* sunt utilizate pentru construirea de aplicații Web care întorc de obicei documente *Web* (*HTML*, *XML*, etc) ca răspuns la cererile navigatorilor
- Clasa *HttpServlet* extinde clasa *GenericServlet* deci moștenește de la aceasta toate funcționalitățile. În plus *HttpServlet* adaugă funcționalitatea specifică protocolului *HTTP* și furnizează un cadru în care să construim aplicații *HTTP*
- Clasa *HttpServlet* este o clasă abstractă prezentă în pachetul *javax.servlet.http*. Când se construiește un servlet *HTTP* se creează o clasă care extinde *HTTPServlet*. Un *Servlet HTTP* funcțional trebuie să implementeze una din metodele *service()*, *doGet()*, *doHead()*, *doPost()*, *doPut()*, *doDelete()* corespunzătoare metodelor protocolului *HTTP*
- Metodele *init()*, *destroy()* și *getServletInfo()* aparțin clasei *GenericServlet* din care se derivează clasa *HttpServlet*
- metoda *service()* are prototipul de mai jos. Primul argument al metodei *service()* este un obiect care implementează interfața *HttpServletRequest* și este reprezentarea cererii clientului. Al doilea parametru este un obiect care implementează interfața *HttpServletResponse* și este reprezentarea răspunsului serverului către client

```
protected void service(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException
```

- Când se utilizează *HTTP* este recomandată utilizarea metodelor *HTTP* specifice *doGet()*, *doPost()* în locul suprascrierii metodei *service()*
- Dacă metoda *service()* nu este suprascrisă, atunci implementarea acesteia este să apeleze metodele *doGet()* sau *doPost()* corespunzătoare. Dacă se suprascrige rămâne în sarcina programatorului să apeleze metodele *doGet()* sau *doPost()* corespunzătoare, după modelul de mai jos

```
public void service(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException{

    if (request.getMethod().equalsIgnoreCase("Get"))
        doGet(request, response);
}
```

- Metodele *service()*, *doGet()*, *doHead()*, *doPost()*, *doPut()*, *doDelete()* – au aceeași parametri de intrare ca și metoda *service()*
- Metodele ***doGet()*** și ***doPost()*** – sunt apelate de fiecare dată când apar cereri *HTTP* de tip *GET* respectiv *POST* primite de servlet. Aceste metode sunt de obicei utilizate pentru a prelucra informațiile din formularele web. Informația introdusă de utilizator într-un formular HTML este încapsulată într-un obiect de tip *HttpServletRequest* și trimisă metodei corespunzătoare

- **Redirectarea unei cereri** constă în trimiterea unui URL ca răspuns, iar browser-ul Web va face apel la resursa respectivă pentru a afișa un rezultat. Acest lucru se realizează cu ajutorul metodei `sendRedirect()` din interfața `HttpServletResponse`.

```
response.sendRedirect ("http://localhost:8080/Test/PagTinta");
```

- **Returnarea unei erori** – servleurile pot întoarce coduri de eroare conform protocolului HTTP. Fiecare cerere HTTP primește ca răspuns un cod de stare care va indica tipul de răspuns primit.

```
response.sendError(404, "Eroare grava");
```

- Servleurile rulează în cadrul containerului Tomcat
- Serverul web creează obiectele request (de tip `HttpServletRequest`) și response (`HttpServletResponse`) de fiecare dată când un servlet este accesat dintr-un browser
- Obiectele request și response sunt create per acces
- Obiectele servlet nu sunt create per acces ci sunt reutilizate. Diferite cereri au asociate diferite fire de execuție, nu instanțe.

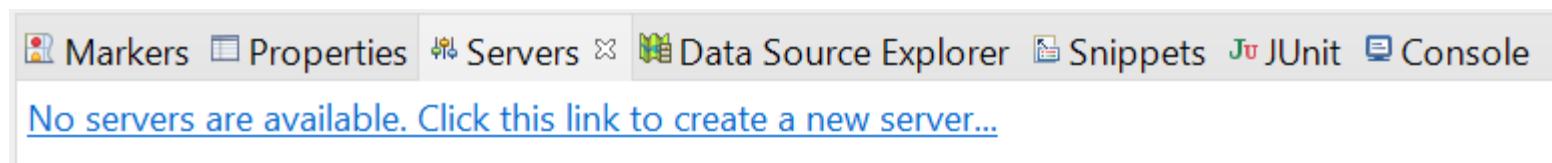
## 6.5 Punerea în funcțiune a serverului web Apache Tomcat

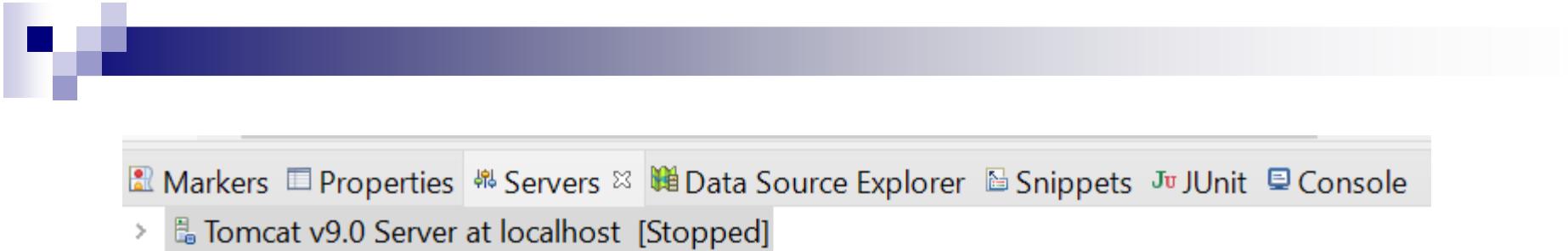
- Execuția proiectului web o va realiza serverului web open-source **Apache Tomcat**. Acesta este o parte a proiectului **Apache Jakarta** și reprezintă o implementare de referință oficială pentru servleurile Java și pentru specificațiile *JSP (Java Server Pages)*
- În acest material a fost utilizat *Apache Tomcat* versiunea 9 care se descarcă de pe link-ul :

<http://tomcat.apache.org/>



- Se asigură o perspectivă în *Eclipse* adecvată unor aplicații *Java Enterprise Edition*. Modificarea perspectivei se poate realiza acționând butonul *Open Perspective* de pe bara de unele sau din obținerea de meniu *Window > Perspective > Open perspective > Other > Java EE*
- În tab-ul **Servers** se introduce server-ul web descărcat

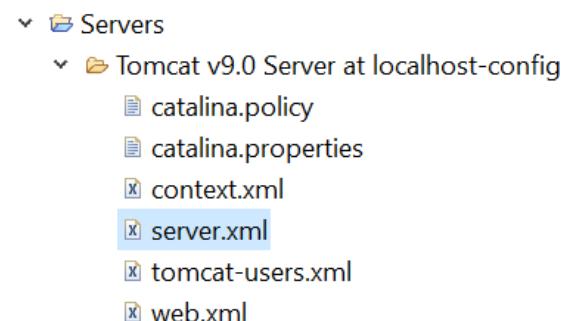




- Pornirea serverului se poate face din tab-ul *Servers*, făcând click dreapta pe server și alegând comanda *Start* sau acționând butonul *Start* precum în captura de mai jos



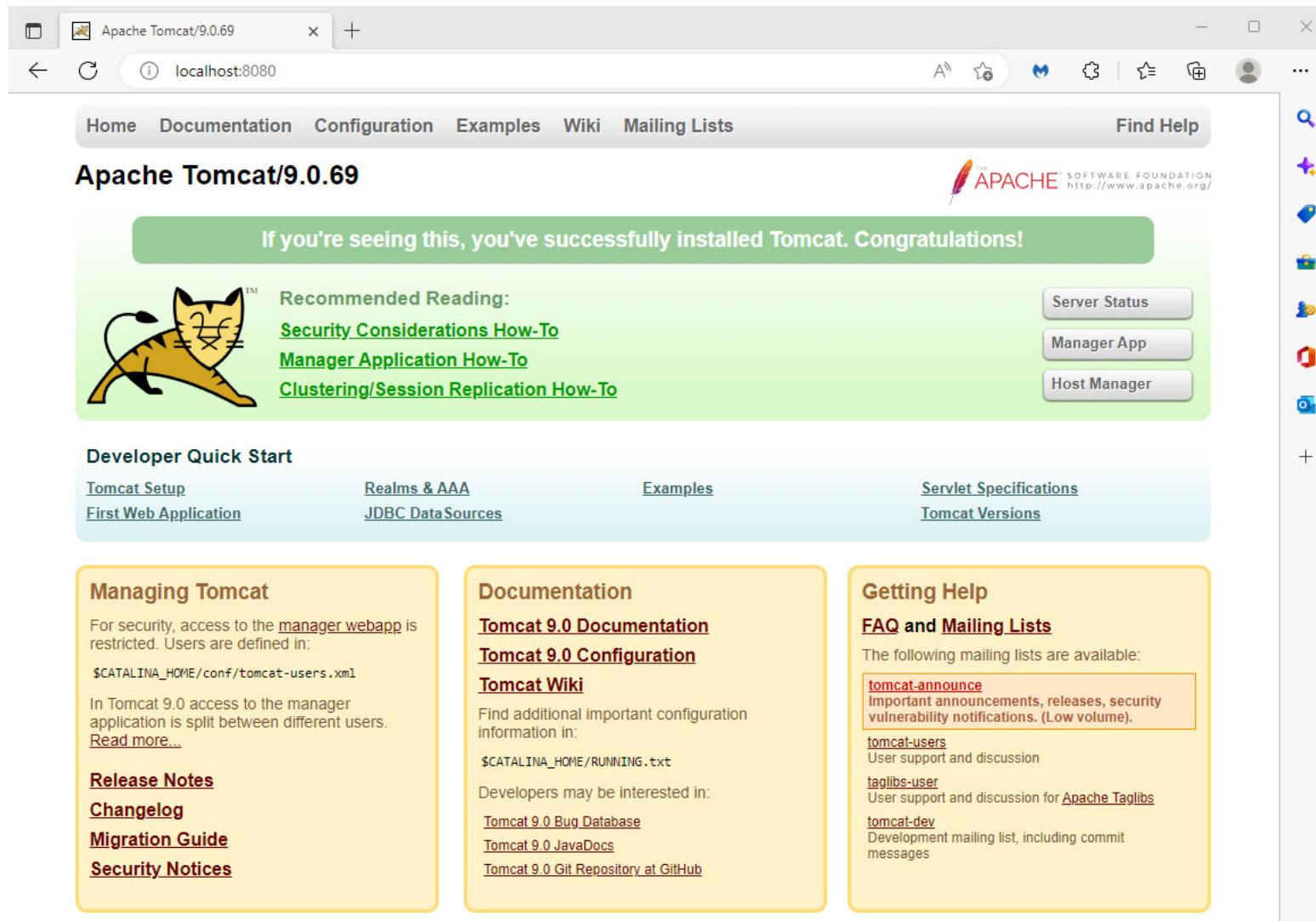
- Dacă portul implicit 8080 care este necesar serverului web *Apache Tomcat* este ocupat, acesta poate fi modificat din fișierul *server.xml* (care poate fi accesat prin *project explorer*) și se poate introduce un alt port care este liber de exemplu 8081, 8082, etc



- Se testează funcționarea corespunzătoare a serverului, pornind serverul și apoi accesând în orice browser URL-ul [http://localhost:nr\\_port](http://localhost:nr_port) unde nr\_port este implicit 8080 (daca a fost schimbat se introduce noul număr al portului)

```
<Connector connectionTimeout="20000" port="8080" protocol="HTTP/1.1" redirectPort="8443"/>
```

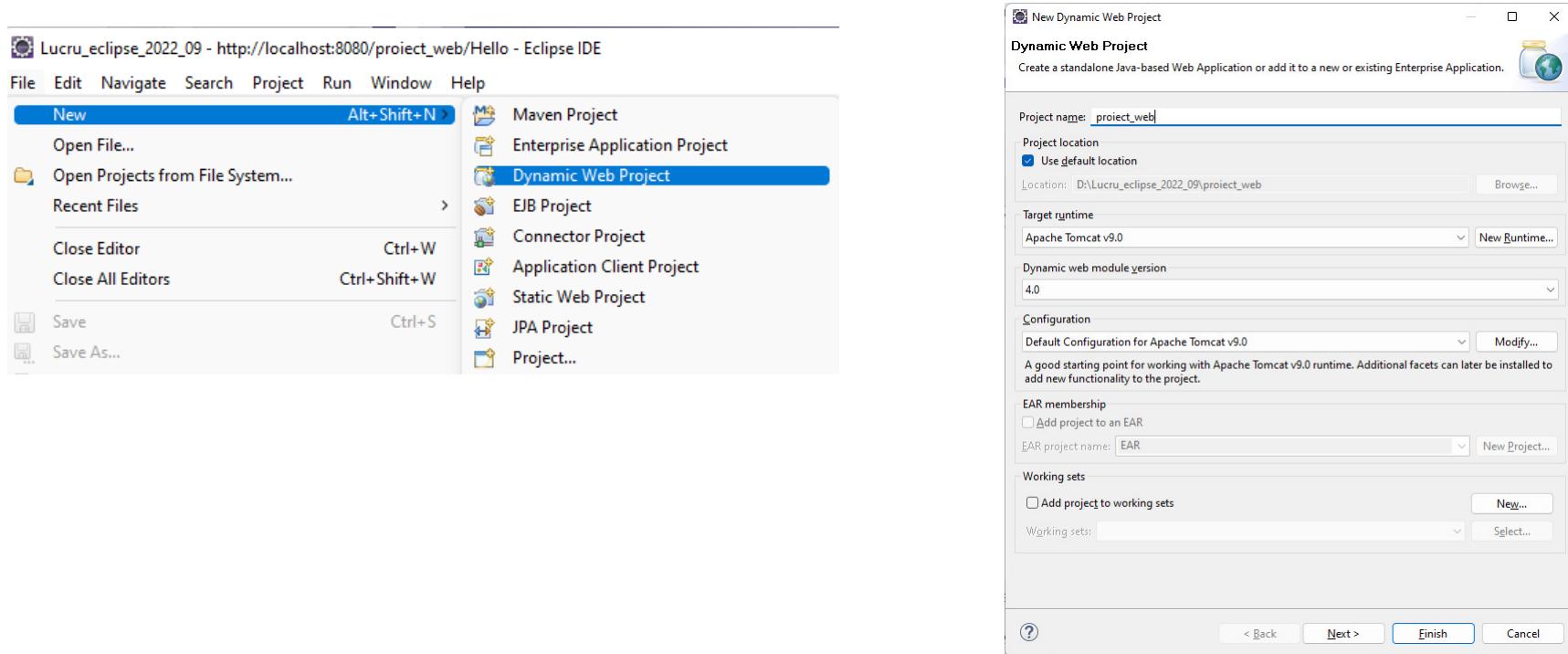
- Dacă Apache Tomcat funcționează corespunzător pagina următoare ar trebui să apară



- Dacă în locul paginii precedente apare eroarea 404 – *The origin server did not find a current representation for the target resource or is not willing to disclose that one exists*- cu toate că serverul este pornit, atunci trebuie realizate următoarele configurații:
  - Se oprește serverul
  - Se face click dreapta pe server și se alege opțiunea **Properties**
  - Dacă în fereastra care se deschide este completată opțiunea **Location: [workspace metadata]** atunci se schimbă locația cu ajutorul comenzi **Switch location** care va face ca noua locație să fie **/Servers/ Tomcat v9.0 Server at localhost.server**. Se aplică configurația făcută și se închide fereastra.
  - Se face dublu click pe server și se schimbă valoarea proprietății **Server locations** din **Use workspace metadata** în **Use Tomcat installation**
  - Se pornește serverul și se testează din nou URL-ul <http://localhost:8080>
- Configurarea serverului web se poate face prin intermediul fișierelor XML de configurare din proiectul **Servers**, proiect care a fost creat în *workspace* odată cu introducerea serverului web sau prin intermediul fișierelor xml de configurație din directorul **apache-tomcat-9.0.34\conf**

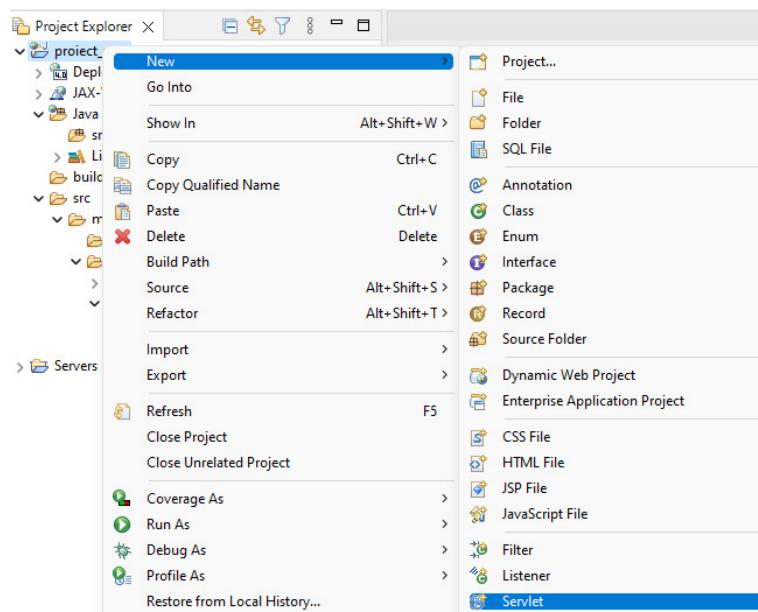
## 6.6 Crearea unui proiect web dinamic

- Având o perspectivă în Eclipse adecvată dezvoltării de aplicații *Java Enterprise Edition*, se alege comanda *File > New > Dynamic Web Project*, se introduce numele proiectului și se apasă *Finish*

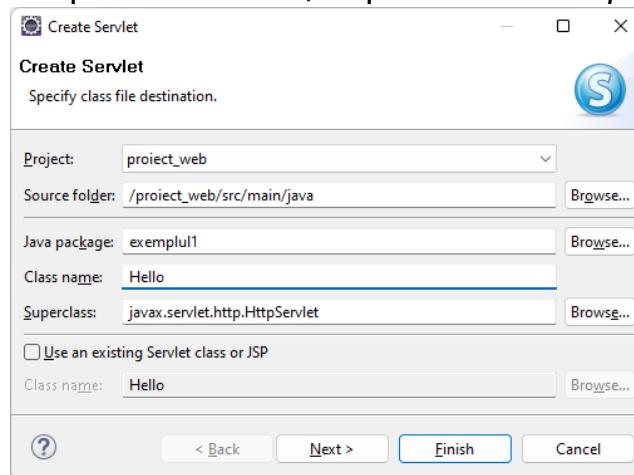


- Se configurează browserul preferat pentru rularea proiectului web, mergând în opțiunea de meniu *Window > Preferences > General > Web Browser*. În acest material s-a utilizat browser-ul încorporat în *Eclipse*, alegând opțiunea *Use internal web browser*

- În continuare se creează un servlet în proiectul web dinamic, făcând click dreapta pe proiect și alegând *New > Servlet*



- Servletul va fi amplasat în proiectul creat, în pachetul *exemplul1*

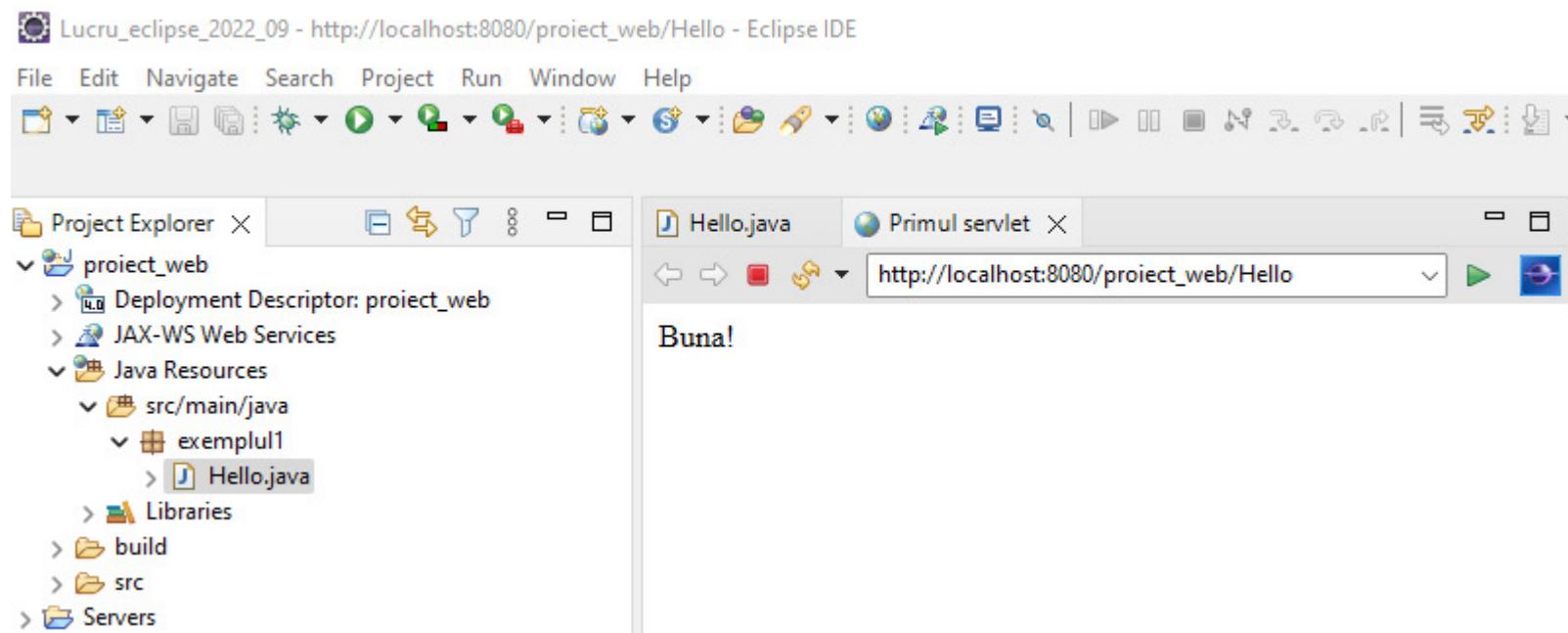


- Se modifică servletul generat încât să conțină următorul cod

```
package exemplu1;
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

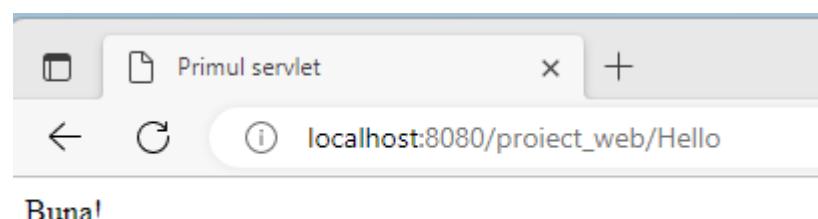
@WebServlet("/Hello")
public class Hello extends HttpServlet {
    private static final long serialVersionUID = 1L;
    public Hello() {
        super();
    }
    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        PrintWriter pw=response.getWriter();
        pw.println("<html><head><title>Primul servlet</title></head><body>" +
        +"Buna!</body></html>");
    }
}
```

- Rularea servletului determină afișarea mesajului “Buna!”, titlul ferestrei va fi “*Primul servlet*”



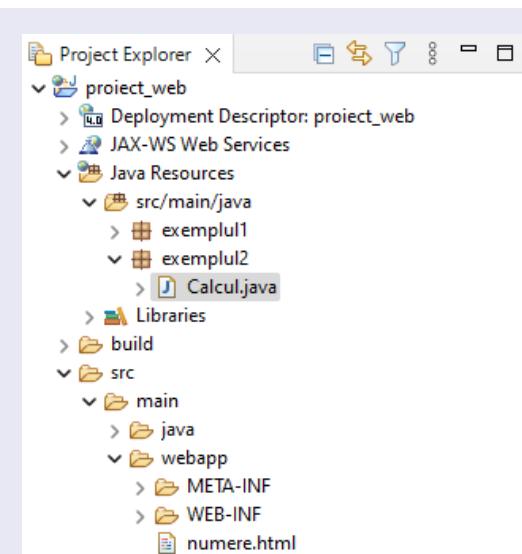
- Servletul poate fi rulat din orice browser dacă se accesează URL-ul de mai jos, cu condiția ca serverul Web să fie pornit

[http://localhost:8080/proiect\\_web/Hello](http://localhost:8080/proiect_web/Hello)



- În continuare se creează în proiectul web dinamic o pagina ***numere.html*** (paginile html, htm și jsp se amplasează în directorul ***webapp***) care afișează un formular cu două casete de text și un buton care apelaază servlețul *Calcul*, din pachetul *exemplul2*. Servlețul va extrage valorile din casetele de text, va calcula suma și va trimite rezultatul către browser

```
<html>
  <head>
    <title>
      Suma a doua numere
    </title>
  </head>
  <body>
    <form method ="GET" action="Calcul">
      <p>Numarul 1:<input type="text" name="nr1"></p>
      <p>Numarul 2:<input type="text" name="nr2"></p>
      <p><input type="submit" value ="Trimite" ></p>
    </form>
  </body>
</html>
```



The screenshot shows a web browser window titled "Suma a doua numere". The address bar shows the URL "http://localhost:8080/proiect\_web/numere.html". The page content consists of two text input fields labeled "Numarul 1:" and "Numarul 2:", and a submit button labeled "Trimite".

- Servletul *Calcul* apelat de către form-ul din fișierul ***numere.html***

```
package exemplul2;

import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet("/Calcul")
public class Calcul extends HttpServlet {
    private static final long serialVersionUID = 1L;

    public Calcul() {
        super();
    }

    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws
        ServletException, IOException {

        String nr1=request.getParameter("nr1");
        String nr2=request.getParameter("nr2");

        PrintWriter out = response.getWriter();
        out.println("<html><head><title>Rezultatul</title></head><body>" );
    }
}
```

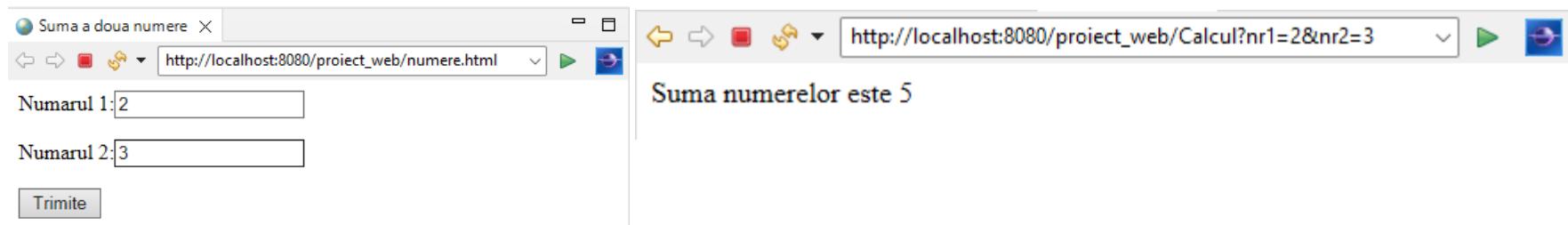
```

        if(nr1==null || nr2==null) {
            out.println("Rulati intai fisierul numere.html</body></html>");
        }
        else {
            try {
                int x=Integer.parseInt(nr1);
                int y=Integer.parseInt(nr2);

                int suma=x+y;
                out.println("Suma numerelor este "+suma+"</body></html>");
            }
            catch(Exception ex) {
                out.println("Valori lipsa sau in format necorespunzator</body></html>");
            }
        }
    }
}

```

- Capturile de ecran de mai jos ilustrează modul de rulare al exemplului 2



- Datorită utilizării metodei *GET* pentru transmiterea datelor din formular către servlet, *URL*-ul servletului include valorile completeate în formular

- Exemplul precedent calculează suma a două numere folosind un fișier html și un servlet.
- Exemplul următor calculează suma a două numere folosind un servlet și metodele *doGet()* și *doPost()* ale acestuia. Metoda *doGet()* va transmite către browser formularul în care utilizatorul poate introduce numerele pe care dorește să le adune. Formularul utilizează metoda *POST* pentru transmiterea datelor, acest lucru va determina ca la apăsarea butonului din formular să se apeleze metoda *doPost()* a servletului care extrage valorile din casetele de text, calculează suma și transmite rezultatul către browser

```
package exemplul3;

import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet("/Suma")
public class Suma extends HttpServlet {
    private static final long serialVersionUID = 1L;

    public Suma() {
        super();
    }

    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        String numar1 = request.getParameter("numar1");
        String numar2 = request.getParameter("numar2");
        int suma = Integer.parseInt(numar1) + Integer.parseInt(numar2);
        out.println("Suma este: " + suma);
    }
}
```

```

protected void doGet(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
PrintWriter out = response.getWriter();
out.println("<html><head><title></title><body>");
out.println("<form method ='POST' action='Suma'>");
out.println("<p>Numarul 1:<input type='text' name='nr1'></p>");
out.println("<p>Numarul 2:<input type='text' name='nr2'></p>");
out.println("<p><input type='submit' value ='Calculeaza suma' ></p>");
out.println("</form>");
out.println("</body>");
out.println("</html>");
}

protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
PrintWriter out = response.getWriter();
String nr1=request.getParameter("nr1");
String nr2=request.getParameter("nr2");

out.println("<html><head><title>Rezultatul</title></head><body>" );
try {
    int x=Integer.parseInt(nr1);
    int y=Integer.parseInt(nr2);

    int suma=x+y;
    out.println("Suma numerelor este "+suma+"</body></html>");
}

```

```

        catch(Exception ex) {
            out.println("Valori lipsa sau in format necorespunzator</body></html>");
        }
    }
}

```

The image shows two side-by-side browser windows. Both have the URL `http://localhost:8080/project_web/Suma` in the address bar.

**Left Browser Window:**

- Address bar: `http://localhost:8080/project_web/Suma`
- Form fields:
  - Numarul 1:
  - Numarul 2:
- Button:

**Right Browser Window:**

- Address bar: `http://localhost:8080/proiect_web/Suma`
- Text displayed: Suma numerelor este 5

- Datorită utilizării metodei *POST* pentru transmiterea datelor din formular către servlet, URL-ul servletului nu va include valorile complete din formular
- Suma a două numere poate fi calculată folosind doar metoda *goGet* a servletului. În acest caz, dacă metoda *doGet* a fost apelată fără ca butonul din formular să fie apăsat, înseamnă că ea a fost apelată datorită rularii servletului și ea va transmite către browser formularul. Altfel dacă metoda a fost apelată ca urmare a apăsării butonului se vor extrage valorile din casetele de text, se va calcula suma și se va transmite către browser (vezi exemplul următor)

```
package exemplu14;
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet("/GetSuma")
public class GetSuma extends HttpServlet {
    private static final long serialVersionUID = 1L;
    public GetSuma() {
        super();
    }
    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws
        ServletException, IOException {
        PrintWriter out = response.getWriter();
        if(request.getParameter("aduna")==null) {
            out.println("<html><head><title></title><body>");
            out.println("<form method ='GET'>");
            out.println("<p>Numarul 1:<input type='text' name='nr1'></p>");
            out.println("<p>Numarul 2:<input type='text' name='nr2'></p>");
            out.println("<p><input type='submit' value = 'Suma' name='aduna'></p>");
            out.println("</form>");
            out.println("</body>");
            out.println("</html>");
        }
    }
}
```

```

else {
    String nr1=request.getParameter("nr1");
    String nr2=request.getParameter("nr2");
    out.println("<html><head><title>Rezultatul</title></head><body>" );

    try {
        int x=Integer.parseInt(nr1);
        int y=Integer.parseInt(nr2);

        int suma=x+y;
        out.println("Suma numerelor este "+suma+"</body></html>");
    }
    catch(Exception ex) {
        out.println("Valori lipsa sau in format necorespunzator</body></html>");
    }
}
}

```

GetSuma.java

http://localhost:8080/proiect\_web/GetSuma

Numarul 1:

Numarul 2:

GetSuma.java

Rezultatul

http://localhost:8080/proiect\_web/GetSuma?nr1=2&nr2=3&aduna=Suma

Suma numerelor este 5

- În exemplul precedent, butonului din formular a primit numele *aduna*, acesta fiind utilizat pentru a verifica dacă metoda *doGet()* este apelată datorită rulării servletului sau datorită apelării servletului ca urmare a acționării acestui buton. Metoda *request.getParameter("aduna")* va returna valoarea *null* atunci când butonul nu a fost acționat și textul de pe buton în caz contrar
- Atributul *action* al tag-ului *form* nu a mai fost completat în exemplul precedent, acțiunea implicită fiind de reapelare a servletului. Atributul *action* putea să fie completat și să se specifice ca și acțiune numele servletului *GetSuma*

## 6.7 Rularea proiectului web

- Servleturile, jsp-urile sau html-urile din proiect pot fi rulate făcând click dreapta pe ele și alegând opțiunea **Run As > Run on server**
- Se poate face click dreapta pe proiectul web și se alege obținerea Run As > Run on server, în acest caz, în mod implicit, execuția proiectului va începe cu una din paginile:
  - index.html, index.htm, index.jsp
  - default.html, default.htm, default.jsp
- Pagina de start a proiectului web dinamic poate fi schimbată. Configurarea paginii de start a proiectului se realizează cu ajutorul fișierului **web.xml**
- Fișierul **web.xml** se găsește în folderul **/webapp / web-inf**, dacă atunci când se creează proiectul web dinamic se merge cu Next până în ultima fereastră și se bifează opțiunea **Generate web.xml deployment descriptor**. Dacă nu s-a parcurs această etapă poate fi generat oricând făcând click dreapta pe proiect și alegând *Java EE Tools > Generate Deployment Descriptor Stub*

- Fișierul **web.xml** are următorul conținut:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://xmlns.jcp.org/xml/ns/javaee"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
  http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd" version="4.0">
  <display-name>proiect_web</display-name>
  <welcome-file-list>
    <welcome-file>index.html</welcome-file>
    <welcome-file>index.htm</welcome-file>
    <welcome-file>index.jsp</welcome-file>
    <welcome-file>default.html</welcome-file>
    <welcome-file>default.htm</welcome-file>
    <welcome-file>default.jsp</welcome-file>
  </welcome-file-list>
</web-app>
```

- Acest fișier se poate edita cu scopul de a asigura începerea execuției proiectului web cu fișierul care se dorește
- Fișierele **html** și **jsp** din proiect trebuie să fie amplasate în folderul **webapp**
- Să se modifice fișierul **web.xml** și în locul fișierului *index.html* să se completeze întâi fișierul *numere.html* și apoi *Suma*. După ce se modifică și se salvează **web.xml**, se rulează proiectul web făcând clic drept pe *proiect\_web* și alegând **Run as > Run on server**. Dacă noua pagină configurată nu apare se dă **refresh** în browser și/sau se restartează serverul web

## 6.8 Adnotăția @WebServlet

- Crearea unui servlet în cadrul unui proiect web dinamic determină crearea clasei servletului și deasupra acesteia amplasarea adnotăției @WebServlet care primește ca și parametru un sir de caractere care are aceeași valoare cu numele clasei servletului
- Acest sir de caractere este atribuit parametrului **urlPatterns** și reprezintă numele care poate fi utilizat pentru a rula servletul
- De exemplu, se creează un pachet numit **exemplu5** și în cadrul lui se creează servletul **UnServlet**:

```
package exemplu5;

import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.*;

@WebServlet("/UnServlet")
public class UnServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;

    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws
        ServletException, IOException {

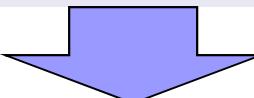
        PrintWriter out=response.getWriter();
        out.print("UnServlet ruleaza");
    }
}
```

- Se execută acest servlet făcând click dreapta pe el și alegând comanda **Run on Server**. Servletul rulează, iar în browser este afișat url-ul:

[http://localhost:8080/proiect\\_web/UnServlet](http://localhost:8080/proiect_web/UnServlet)

- Se modifică parametrul adnotăției @WebServlet astfel încât să nu mai coincidă cu clasa servletului (se alege de exemplu denumirea MyServlet) și se rulează servletul din nou. În URL nu apare denumirea clasei servletului ci valoarea parametrului adnotăției @WebServlet

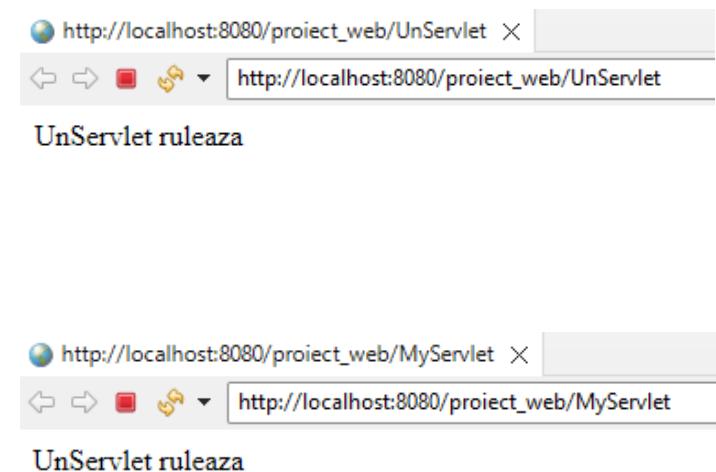
```
@WebServlet("/UnServlet")
public class UnServlet extends HttpServlet {/*...*/}
```



```
http://localhost:8080/proiect\_web/UnServlet
```

**SAU**

```
@WebServlet(urlPatterns="/MyServlet")
public class UnServlet extends HttpServlet {/*...*/}}
```



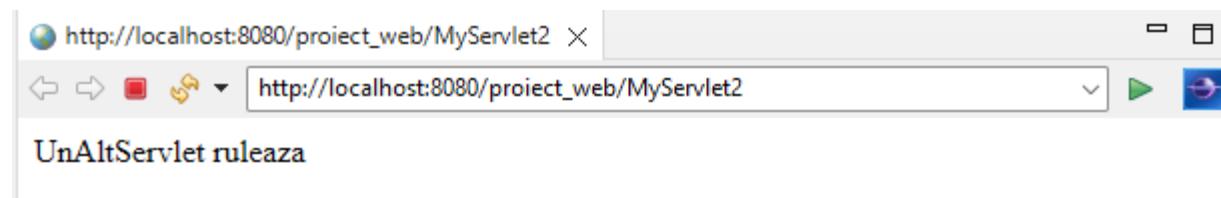
- Stabilirea numelui de apel al servletului se poate realiza și fără a utiliza adnotăția **@WebServlet**, prin realizarea unor configurații în fișierul **web.xml**
- Fișierul web.xml poate fi accesat prin dublu click pe **Deployment descriptor** sau navigând în directorul **src\main\webapp\WEB-INF** și deschizând fișierul
- În pachetul *exemplu15* se creează *UnAltServlet*. Se șterge sau se pune în comentariu adnotăția **@WebServlet**. Încercarea de a rula servletul eșuează cu codul de eroare 404.

```
package exemplu15;
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
//import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
//{@WebServlet("/UnAltServlet")}
public class UnAltServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;
    public UnAltServlet() {
        super();
    }
    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws
    ServletException, IOException {
        PrintWriter out=response.getWriter();
        out.print("UnAltServlet ruleaza");
    }
}
```

- Se adaugă în fișierul web.xml următorul cod:

```
<!-- ... -->

<servlet>
    <servlet-name>UnAltServlet</servlet-name>
    <servlet-class>exemplul5.UnAltServlet</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>UnAltServlet</servlet-name>
    <url-pattern>/MyServlet2</url-pattern>
</servlet-mapping>
```

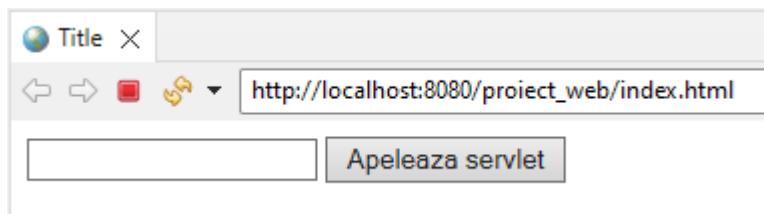


- După completarea și salvarea fișierului web.xml servletul rulează, afișând mesajul din captura de ecran de mai sus
- Tagul **servlet** conține tagurile **servlet-name** și **servlet-class** prin care se specifică numele servletului și clasa acestuia
- Specificarea denumirii care poate fi folosită pentru a rula servletul se realizează în interiorul tagului **servlet-mapping** cu ajutorul tagurilor **servlet-name** și **url-pattern**. Denumirea introdusă în tagul **url-pattern** apare în URL atunci când rulăm servletul cu denumirea din tagul **servlet-name**

## 6.9 Utilizarea sesiunilor

- În situația în care dorim să transmitem valori între diferite accesări ale unui servlet, sau între diferite servleuri, putem lucra cu sesiuni
- Se consideră servleurile **WebServlet1** și **WebServlet2** în pachetul **exemplu6** al proiectului **proiect\_web** și fișierul **index.html** în directorul **webapp** al aceluiași proiect
- Fișierul **index.html**, conține un formular cu o casetă de text și un buton. Apasarea butonului determină apelul servleutului **WebServlet1**

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="ISO-8859-1"><title>Title</title>
    </head>
    <body>
        <form method="get" action="WebServlet1">
            <input type="text" name="txt" />
            <input type="submit" value="Apeleaza servlet" />
        </form>
    </body>
</html>
```



- Fișierul **WebServlet1.java** – conține un servlet care preia în metoda *doGet()* textul introdus în caseta de text și-l afișează în consolă. Apoi se realizează o redirectare către servletul **WebServlet2**

```

package exemplul6;

import java.io.*;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.*;

@WebServlet("/WebServlet1")
public class WebServlet1 extends HttpServlet {
    private static final long serialVersionUID = 1L;

    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws
        ServletException, IOException {

        String s=request.getParameter("txt");

        PrintWriter out=response.getWriter();
        HttpSession session=request.getSession();

        if (!s.isBlank()){
            session.setAttribute("txt", s);
            System.out.println("webServlet1: request.getParameter(\"txt\")="+s);
            response.sendRedirect("WebServlet2");
        }
        else
            out.print("In caseta de text nu a fost introdusa nici o valoare");
    }
}

```

INFO: Server startup in 450 ms  
webServlet1: request.getParameter("txt")=proba

- Fișierul **WebServlet2.java** conține un servlet care afișeaza în browser conținutul casetei de text, transmis cu ajutorul sesiunilor și arată că acesta nu poate fi preluat în cel de-al doilea servlet cu ajutorul parametrului de tip **HttpServletRequest**

```
package exemplul6;

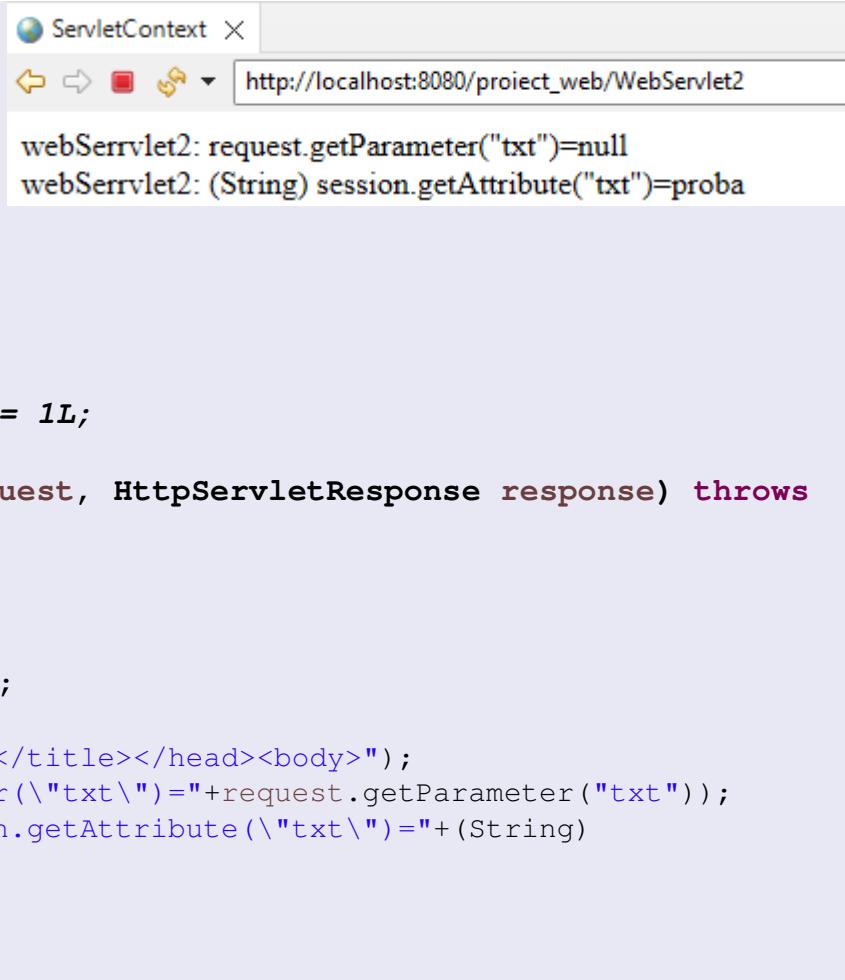
import java.io.*;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.*;

@WebServlet("/WebServlet2")
public class WebServlet2 extends HttpServlet {
    private static final long serialVersionUID = 1L;

    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws
        ServletException, IOException {

        String s=request.getParameter("txt");
        PrintWriter out=response.getWriter();
        HttpSession session=request.getSession();

        out.print("<html><head><title>ServletContext</title></head><body>");
        out.print("webServlet2: request.getParameter(\"txt\")="+request.getParameter("txt"));
        out.print("<br>webServlet2: (String) session.getAttribute(\"txt\")=(String)
            session.getAttribute(\"txt\"))");
        out.print("</body></html>");
    }
}
```



The screenshot shows a browser window titled "ServletContext" with the URL "http://localhost:8080/proiect\_web/WebServlet2". The page content displays two lines of text: "webServlet2: request.getParameter("txt")=null" and "webServlet2: (String) session.getAttribute("txt")=proba". This indicates that the first servlet did not receive the parameter from the session, while the second servlet successfully retrieved it.

- Un obiect de sesiune este creat per utilizator per browser
- Obiectele de sesiune sunt accesibile între diferite accesări ale aceluiași servlet sau între servleuri
- Fiecare obiect request conține un mâner (handler) către un obiect de sesiune

## 6.10 Lucru cu obiecte de tip *ServletContext*

- Obiectele de tip *ServletContext* sunt folosite pentru a accesa date între diferiți utilizatori sau între diferite browsere sau pentru a prelua parametri de context din fișierul *web.xml*
- Dacă se copiază URL-ul [http://localhost:8080/proiect\\_web/WebServlet2](http://localhost:8080/proiect_web/WebServlet2) din exemplul precedent într-un alt browser se va obține valoarea null pentru obiectul de sesiune
- Dacă se adaugă în fișierul **WebServlet1.java** următoarele linii de cod:

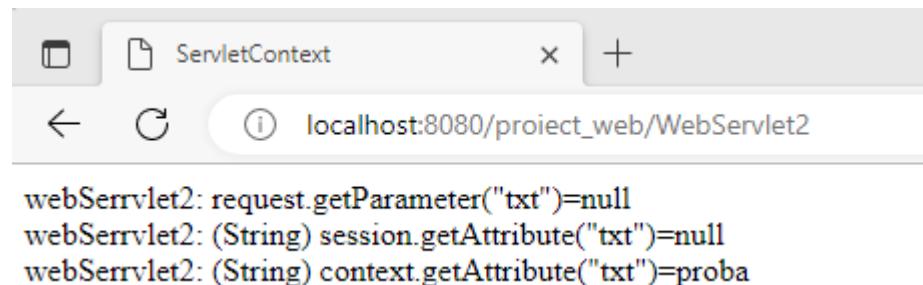
```
//...
    ServletContext context=request.getServletContext();
//...
    context.setAttribute("txt", s);
```

- și dacă se adaugă în fișierul **WebServlet2.java** următoarele linii de cod:

```
//...
ServletContext context=request.getServletContext();

//...
out.print("<br>webSerrvlet2: (String) context.getAttribute(\"txt\")="+(String)
context.getAttribute("txt"));
```

- După ce se rulează exemplul în Eclipse se poate copia url-ul [http://localhost:8080/proiect\\_web/WebServlet2](http://localhost:8080/proiect_web/WebServlet2) în orice alt browser, se va vedea că valoarea introdusă în caseta de text este accesibilă indiferent de browser, datorită utilizării obiectului *ServletContext*



- O altă utilitate a obiectului de tip *ServletContext* este să preia informațiile de configurare din fișierul *web.xml*.
- Informațiile utilizate de servleturile din cadrul proiectului web dinamic (cum ar fi diverse constante) pot fi completate în fișierul *web.xml*, folosind tagul <context-param>

```
<context-param>
    <param-name>...</param-name>
    <param-value>...</param-value>
</context-param>
```

- Exemplul de mai jos preia valorile unor parametri de context din fișierul fișierului *web.xml* cu ajutorul obiectului *ServletContext*
- Se deschide fișierul *web.xml* (prin dublu click pe *Deployment descriptor*) și se completează în fișierul *web.xml* cu următorii parametri de context:

```
<web-app ...>
...
<context-param>
    <param-name>parametrul 1</param-name>
    <param-value>valoare parametrul 1</param-value>
</context-param>

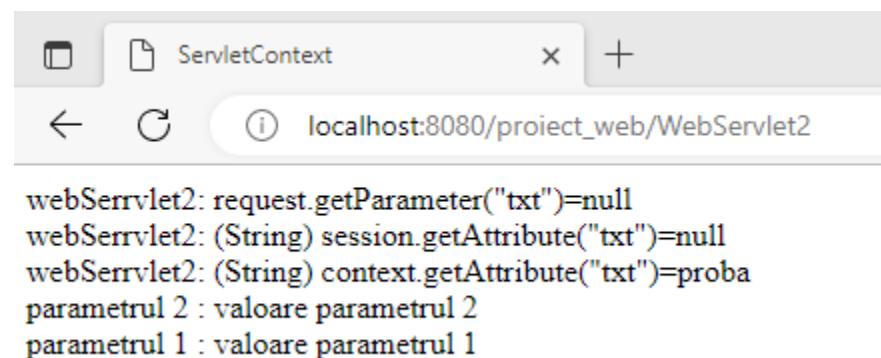
<context-param>
    <param-name>parametrul 2</param-name>
    <param-value>valoare parametrul 2</param-value>
</context-param>
```

- Se completează fișierul **WebServlet2**, din exemplul precedent, cu următoarele linii de cod:

```
Enumeration<String> e=context.getInitParameterNames();

String nume="";
while(e.hasMoreElements()){
    nume=e.nextElement();
    out.print("<br> "+nume+" : "+context.getInitParameter(nume));
}
```

- Output-ul programului este:

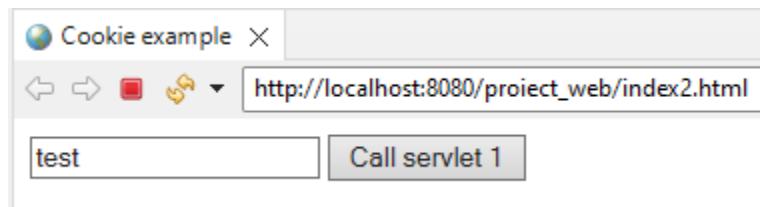


## 6.11 Utilizarea cookie-urilor în cadrul servelturilor

- *cookie-urile reprezintă informații care se transmit între diferite cereri ale clientilor*
- Un cookie are:
  - O denumire
  - O valoare
  - Atribute opționale cum ar fi:
    - durata maximă de viață (în secunde)
    - Versiunea
    - Un comentariu
    - Etc
- Cookie-urile sunt stocate pe partea clientului, în memoria *cache* a browser-ului și se transmit împreună cu răspunsul servletului
- Prin intermediul cookie-urilor se poate transmite doar informație textuală și pot fi utilizate doar dacă browser-ul este configurat să accepte cookie-uri
- Tipuri de cookie-uri:
  - *Nepersistente* – sunt valide doar în cadrul unei sesiuni. Sunt distruse de fiecare dată când utilizatorul închide browser-ul
  - *Persistente* – sunt valide în cadrul mai multor sesiuni. Nu se distrug dacă utilizatorul închide browser-ul.

- Exemplul 7, este compus din fișierele *index2.html*, *Servlet1.java* și *Servlet2.java* și ilustrează cum poate fi transmis un *cookie*
- Fișierul *index2.html* conține un formular care apelează *Servlet1* și are conținutul următor:

```
<html>
  <head>
    <title>Cookie example</title>
  </head>
  <body>
    <form method="post" action="Servlet1">
      <input type="text" name="txt">
      <input type="submit" value="Call servlet 1" />
    </form>
  </body>
</html>
```



- Fișierul *Servlet1.java* are conținutul următor:

```
package exemplul7;

import java.io.IOException;
import java.io.PrintWriter;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.Cookie;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet("/Servlet1")
public class Servlet1 extends HttpServlet {
    private static final long serialVersionUID = 1L;

    public Servlet1() {
        super();
    }

    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
        ServletException, IOException {

        response.setContentType("text/html");
        PrintWriter pw=response.getWriter();
```

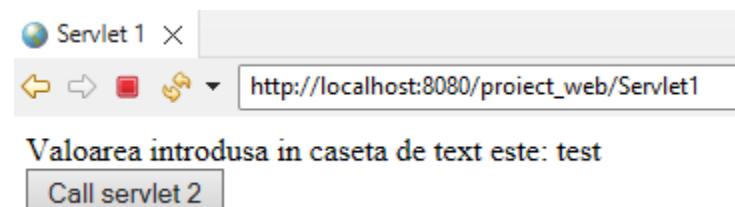
```

String s=request.getParameter("txt");
pw.print("Valoarea introdusa in caseta de text este: "+s);

Cookie ck=new Cookie("cookie_txt",s);
//ck.setMaxAge(2); //setarea unei durate de viata
response.addCookie(ck);

pw.println("<html><head><title>Servlet 1</title></head><body>"
        + "<form method='post' action='Servlet2'>"
        + "<input type='submit' value='Call servlet 2' />"
        + "</form>");
pw.close();
}
}

```

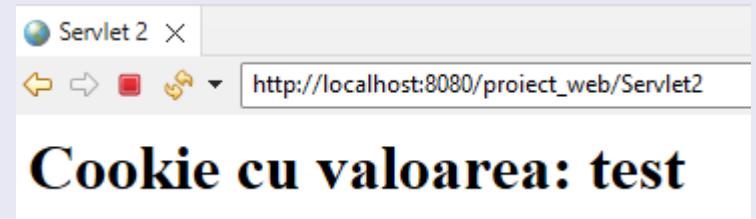


- Servletul 1, creează un cookie cu denumirea “cookie\\_txt” și valoarea egală cu cea introdusă în caseta de text
- Afisează un formular cu un buton care apelează metoda *doPost()* din *Servlet2*

- Fișierul *Servlet2.java* are conținutul de mai jos. Servletul 2, extrage din parametrul request cookie-urile sub forma unui vector și afișează valoarea cookie-ului transmis

```
package exemplul7;
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.Cookie;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet("/Servlet2")
public class Servlet2 extends HttpServlet {
    public Servlet2() {
        super();
    }
    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        Cookie ck[] = request.getCookies();
        out.print("<html><head><title>Servlet 2</title></head><body>" +
                  "<h1>Cookie cu valoarea: " + ck[0].getValue() + "</h1></body><html>");
        out.close();
    }
}
```



## 6.12 Servleturi și JDBC

- Servleturile se pot utiliza împreună cu tehnologia JDBC pentru a realiza aplicații web cu baze de date
- Driverul de conectare la baza de date trebuie încărcat în directorul **webapp / WEB-INF / lib**.
- Pe calculatoarele din laborator acesta se găsește pe calea C:\Program Files (x86)\MySQL\Connector J 8.0\mysql-connector-java-8.0.19.jar sau se poate descărca de pe link-ul <https://downloads.mysql.com/archives/c-j/>
- Se creează pachetul exemplul8 și în acesta servlețul *OperatiiJDBC*
- Servlețul realizează operațiile elementare asupra tabelei MySQL persoane, tabelă cu câmpurile *id*, *nume*, *varsta*

```
package exemplul8;

import java.io.*;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.*;
import java.sql.*;
```



The screenshot shows a web application interface for managing a 'Persoane' database table. The table has columns 'Id', 'Nume', and 'Varsta'. The data is as follows:

Id	Nume	Varsta
1	Ana	19
2	Ionel	33
3	Oana	22

```

@WebServlet("/OperatiiJDBC")
public class OperatiiJDBC extends HttpServlet {
    private static final long serialVersionUID = 1L;
    public OperatiiJDBC() {
        super();
    }
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
        try{
            String url = "jdbc:mysql://localhost:3306/test";
            Statement statement=null;
            ResultSet rs=null;
            Class.forName("com.mysql.cj.jdbc.Driver");
            Connection connection = DriverManager.getConnection(url, "root", "root");
            statement = connection.createStatement();
            if(request.getParameter("btnAdauga")==null
            && request.getParameter("btnModifica")==null
            && request.getParameter("btnSterge")==null)
                rs = statement.executeQuery("select * from persoane");

            PrintWriter out=response.getWriter();
            out.println("<html><head><title>Persoane</title></head><body>");
            out.println("<form method='get'>");
            out.println("<table align='center' width='50%' cellspacing='5'>");

            out.println("<tr><td align='right'>Id</td><td><input type='text' "
                    +" name='txtId' size='5'></td></tr>");

```

```

out.println("<tr><td align='right'>Nume</td><td><input type='text' "
+ " name='txtNume'></td></tr>");
out.println("<tr><td align='right'>Varsta</td><td><input type='text' "
+ " name='txtVarsta' size='5' maxlength='3'></td></tr>");
out.println("<tr><td colspan='2' align='center'><input type='submit' "
+ " name='btnAdauga' value='Adauga' style='width: 110px; height: 25px;'>");
out.println("<input type='submit' name='btnModifica' value='Modifica' "
+ " style='width: 110px; height: 25px;'>");
out.println("<input type='submit' name='btnSerge' value='Serge' "
+ " style='width: 110px; height: 25px;'></td></tr>");
out.println("</table></form>");
out.println("<table align='center' width='50%' border='1'>");
out.println("<tr><th>Id</th><th>Nume</th><th>Varsta</th></tr>");

if (request.getParameter("btnAdauga") !=null) {
    int id=Integer.parseInt(request.getParameter("txtId"));
    String nume=request.getParameter("txtNume");
    int varsta=Integer.parseInt(request.getParameter("txtVarsta"));
    String comanda="insert into persoane values (?,?,?)";
    try {
        PreparedStatement ps=connection.prepareStatement(comanda);
        ps.setInt(1, id);
        ps.setString(2, nume);
        ps.setInt(3, varsta);
        ps.executeUpdate();
        ps.close();
        rs = statement.executeQuery("select * from persoane");
    }
}

```

```

        catch (SQLException e) {
            System.out.println(comanda+"\n"+e);
        }
    }

    if (request.getParameter("btnModifica")!=null) {
        int id=Integer.parseInt(request.getParameter("txtId"));
        String nume=request.getParameter("txtNume");
        int varsta=Integer.parseInt(request.getParameter("txtVarsta"));

        String comanda="update persoane set nume=?,varsta=? where id=?";
        try {
            PreparedStatement ps=connection.prepareStatement(comanda);
            ps.setString(1, nume);
            ps.setInt(2, varsta);
            ps.setInt(3, id);
            ps.executeUpdate();
            ps.close();
            rs = statement.executeQuery("select * from persoane");
        } catch (SQLException e) {
            System.out.println(comanda+"\n"+e);
        }
    }

    if (request.getParameter("btnSterge")!=null) {
        int id=Integer.parseInt(request.getParameter("txtId"));
        String comanda="delete from persoane where id=?";
        try {
            PreparedStatement ps=connection.prepareStatement(comanda);

```

```

        ps.setInt(1, id);
        ps.executeUpdate();
        ps.close();
        rs = statement.executeQuery("select * from persoane");
    } catch (SQLException e) {
        System.out.println(comanda+"\n"+e);
    }
}

while(rs.next())
{
    out.println("<tr><td>" +rs.getInt("Id")+"</td><td>" + rs.getString("nume")
            + "</td><td>" +rs.getInt(3)+"</td><tr>");
    out.println("</table></body></html>");
    rs.close();
    statement.close();
    connection.close();
}
catch(Exception ex){
    System.out.println(ex);
}
}
}
}

```

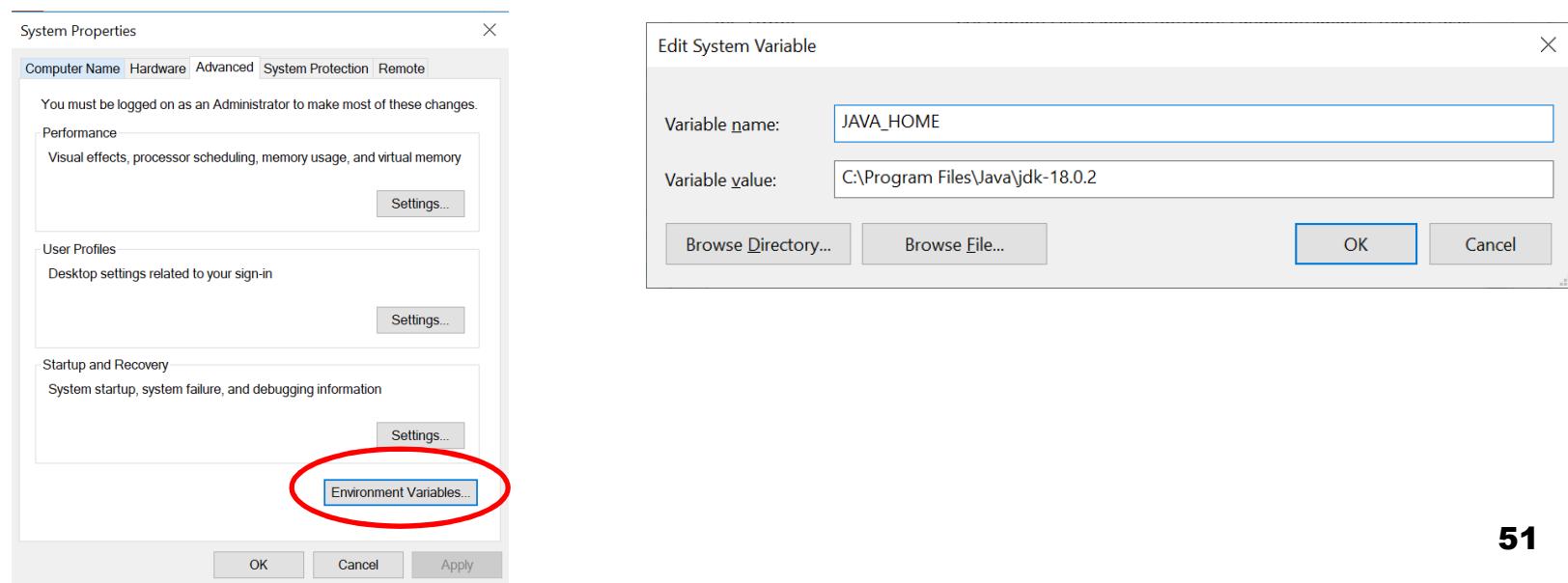
- Aplicația web își începe execuția cu metoda *doGet()* a servletului. În această metodă se încarcă dinamic clasa driverului și driverul este înregistrat de către managerul de drivere
- În continuare se creează o conexiune la baza de date *test*, în care a fost creat tabelul *persoane* și se rulează o comandă SQL care preie în obiectul de tip *ResultSet* toate datele din tabela *persoane*

- Se crează și se instanțiază obiectul de tip *PrintWriter* cu ajutorul căruia se transmite o pagină *html* către browser
- În pagina *html* se creează un formular care apelează servlețul *OperatiiJDBC* și utilizează metoda *get* pentru a transmite datele către acesta (servlețul se autoapeleză atunci când se apasă unul din cele 3 butoane)
- Informația afișată în formular a fost pusă într-un tabel fară borduri, care a fost aliniat la centru și pentru care s-a stabilit o lățime de 50% din lățimea browserului. S-a lasat un spatiu de dimensiune 5 între celule
- Tabelul are 4 linii, primele 3 linii au câte două coloane (pe prima coloană este un text informativ aliniat la dreapta, iar pe a doua coloană sunt cele trei casete de text), iar a patra linie are o singură coloană și conține cele trei butoane. Atributul *colspan* specifică numărul de coloane pe care se va întinde o celulă.
- Sub acest tabel s-a creat un tabel de bordura 1 în care au fost afișate datele preluate din tabela persoane
- Dacă a fost apăsat butonul *btnAdauga*, se preiau datele din cele 3 casete de text și se construiește comanda sql *insert* care se și execută pentru a insera în baza de date persoana dorită. Dacă se produce vreo excepție (duplicat de cheie primară, etc) aceasta este prință în blocul try ...catch interior și se afisează un mesaj corespunzător

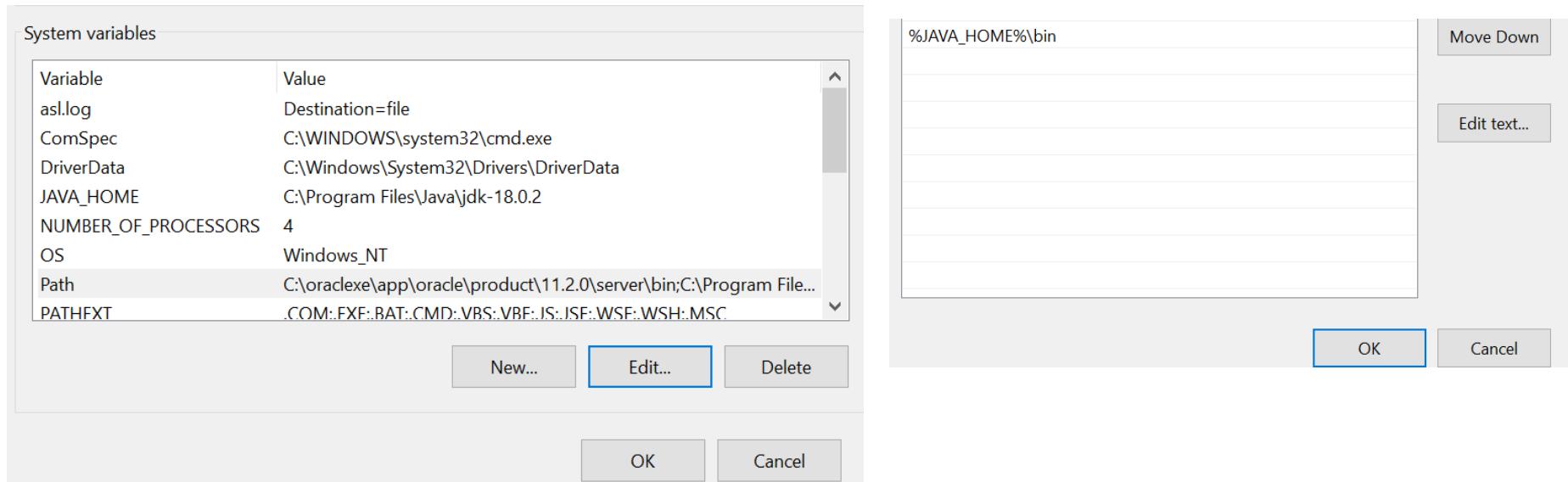
- Tabela a fost interogată din nou pentru a prelua în *ResultSet* inclusiv datele persoanei adăugate și a le afișa în browser
- În mod similar se rulează comenzi *sql* pentru actualizarea datelor unei persoane referită prin *id*, respectiv pentru ștergerea unei persoane identificată prin câmpul *id*

## 6.13 Crearea și rularea fișierului war

- Când se consideră că proiectul web dinamic a fost finalizat se poate construi fișierul war. Pașii care trebuie parcursi în acest scop sunt enumerați în continuare
- Se creează o variabilă de mediu (environment variable) în sistemul de operare, cu denumire **JAVA\_HOME** prin care se specifică calea JDK-ul instalat



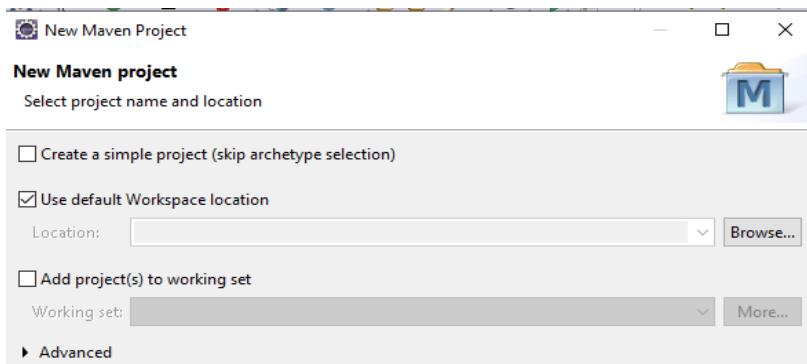
- În continuare se editează **Path** adăugând directorul **bin** al jdk-ului instalat



- În continuare se exportă proiectul web ca și un fișier **WAR**, alegând opțiunea **File > Export > Java > WAR file**
- În fereastra următoare se alege o denumire pentru proiectul web dynamic și se salveaza în folderul **webapps** din **ApacheTomcat**
- Se pornește serverul web lansând **ApacheTomcat \ bin \ startup.bat** din line de comandă
- Se acceseaza în browser URL-ul [http://localhost:8080/Denumire\\_fisier\\_war/](http://localhost:8080/Denumire_fisier_war/)

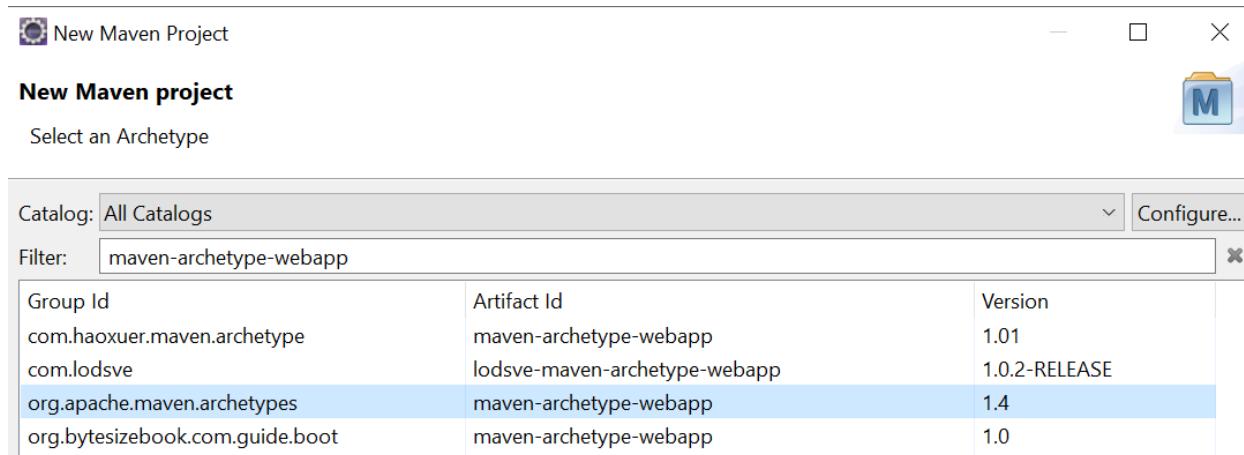
## 6.14 Crearea unui proiect web Maven

- Exemplul următor creează un proiect web cu *Maven* care se conectează la o tabelă *MySQL*, extrage datele din aceasta cu ajutorul *JDBC* și le afișează în browser în format tabelar
- **Versiunea de Eclipse utilizată este Eclipse 2022-06 (4.24.0), Eclipse IDE for Enterprise Java and Web Developers** Aceasta poate fi descărcată de pe linkul: <https://www.eclipse.org/downloads/packages/release/2022-06/r>
- Apache Tomcat a fost configurat să utilizeze portul 8081 (vezi subcapitolul 7.5, pagina 11)
- Se creează un proiect Maven și se lasă **debifată** opțiunea *Create a simple project (Skip archetype selection)*

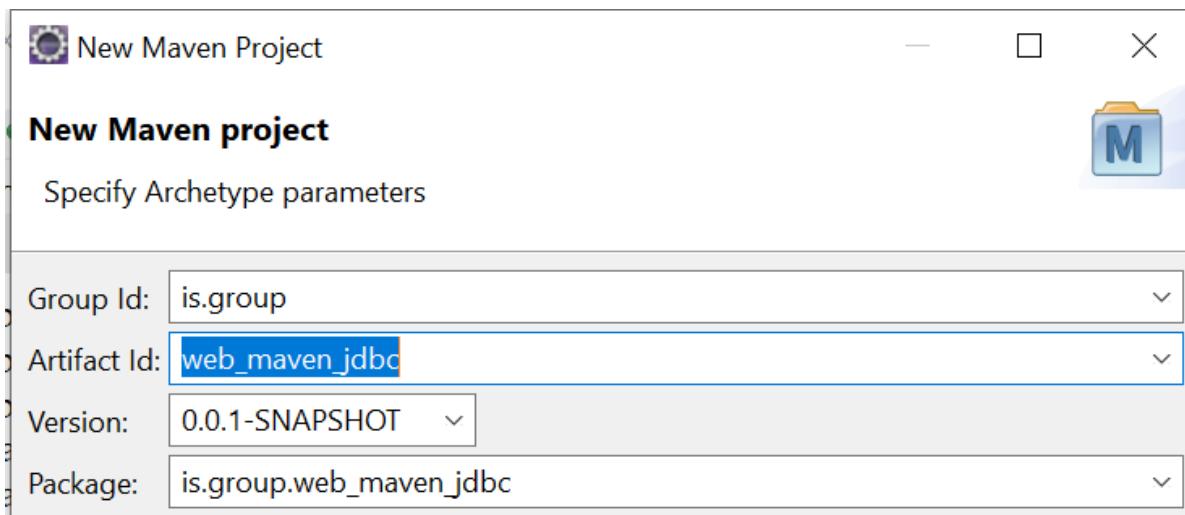


- În fereastra următoare trebuie specificat arhetipul după care se va crea proiectul
- Un arhetip este un model după care se face o lucrare
- În cazul de față prin arhetip se înțelege un model de proiect

- se caută după cuvântul arhetipul *maven-archetype-webapp*



- Se creează un proiect cu caracteristicile din imaginea de mai jos



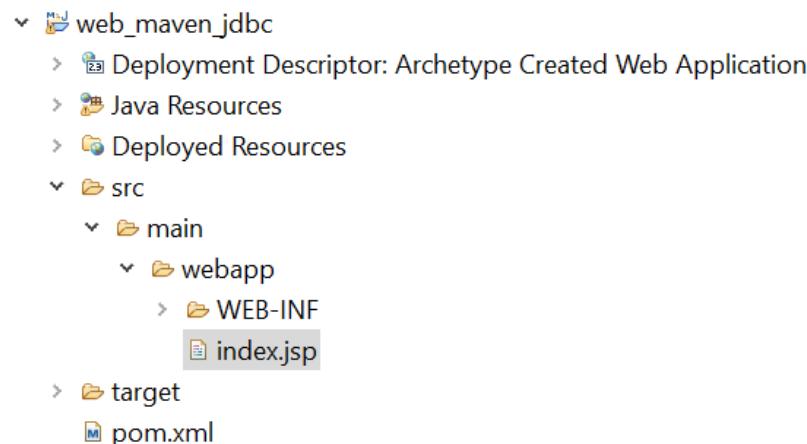
- În pom.xml se elimină dependența de JUnit și se adaugă următoarele două dependente, obținute din depozitul *Maven* central <https://mvnrepository.com>

```
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>javax.servlet-api</artifactId>
    <version>4.0.1</version>
    <scope>provided</scope>
</dependency>

<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.19</version>
</dependency>
```

- În urma salvării fișierului *pom.xml* dependențele ***javax.servlet-api-4.0.1.jar* și *mysql-connector-java-8.0.19.jar*** se descarcă din depozitul maven central în cel local și apoi se aduc în proiect putând fi vizualizate accesând calea Java Resources > Libraries > Maven Dependencies
- După adăugarea dependenței de *Servlet API* și salvarea fișierului *pom.xml*, se actualizează proiectul Maven (click dreapta pe proiect și Maven > Update project), se dă un refresh proiectului (click dreapta pe proiect și refresh) și eroarea *The superclass "javax.servlet.http.HttpServlet" was not found on the Java Build Path* generată de fișierul implicit *index.jsp* va dispărea.
- Browser-ul în care va rula proiectul web dinamic poate fi configurat din meniul ***Window > Preferences > General > Web browser***. În acest material s-a utilizat browser-ul intern din Eclipse fiind aleasă opțiune ***Use internal web browser***

- În continuare poate fi rulat fișierul *index.jsp* pentru a testa funcționarea serverului web (click dreapta pe fișier și *Run as > Run on server*)



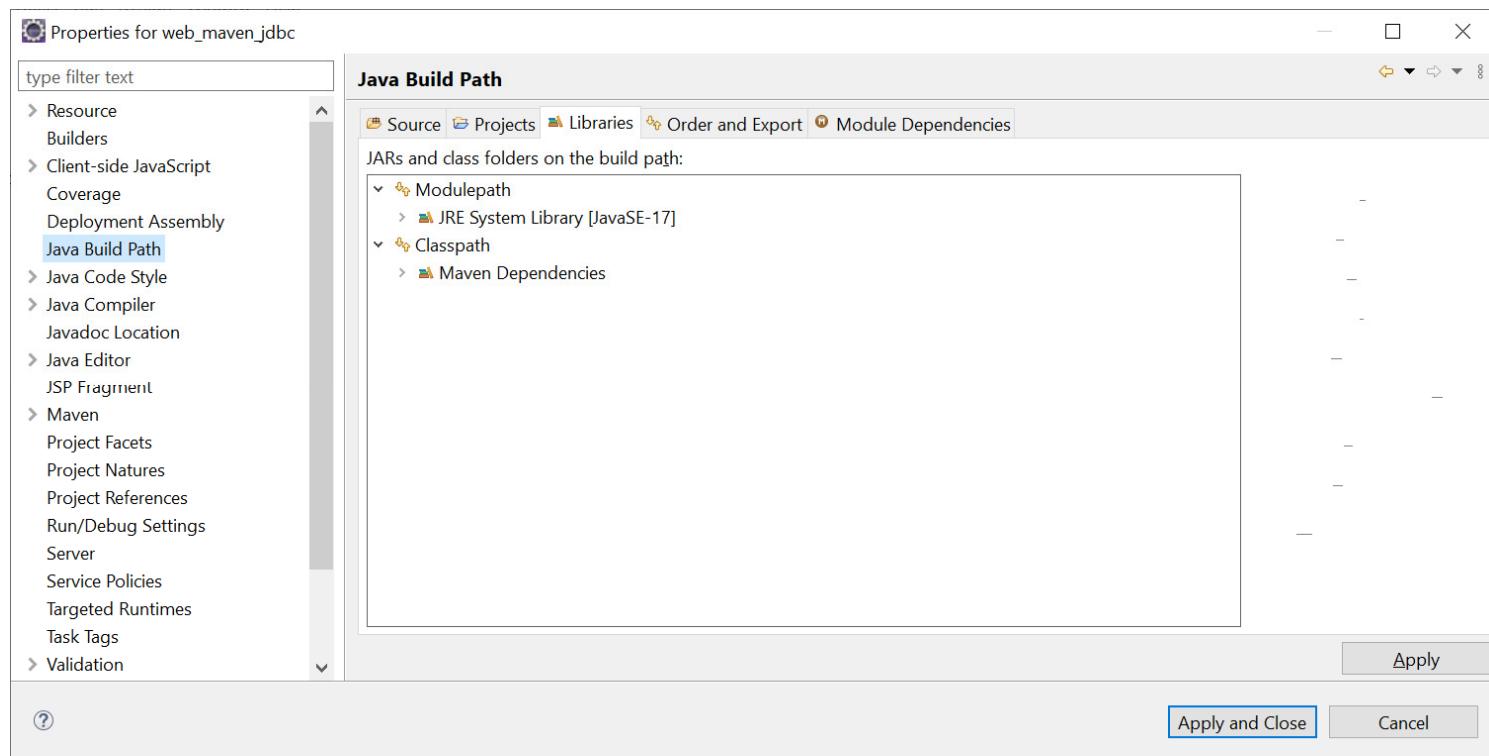
- Conținutul lui *index.jsp*:

```
1<html>
2<body>
3<h2>Hello World!</h2>
4</body>
5</html>
```

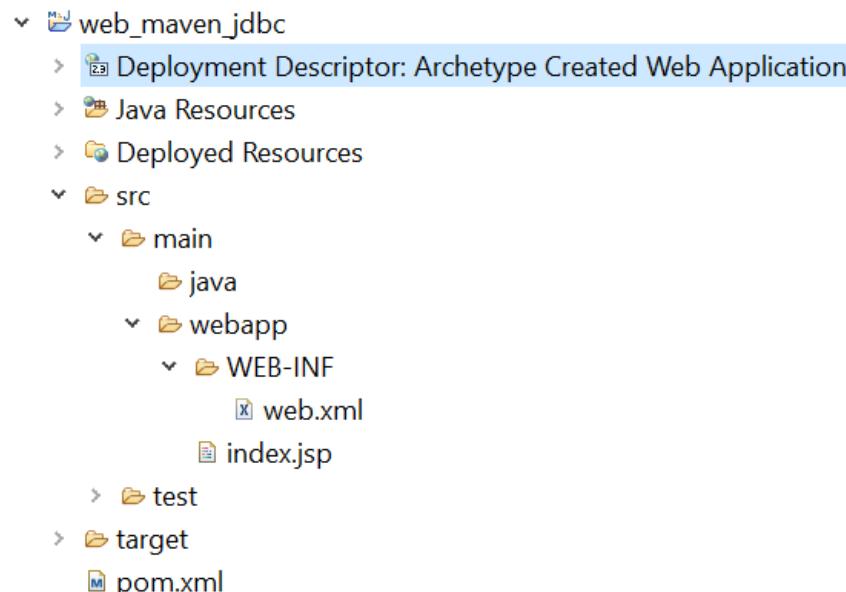
- În eclipse va apărea browser-ul intern care va afișa rezultatul rulării fișierului *index.jsp*



- În fișierul *pom.xml*, în tag-ul *properties* se modifică versiunea de Java utilizată pentru compilarea și interpretarea proiectului și se introduce versiunea instalată (în acest material a fost folosită versiunea 17). După fiecare modificare și salvare a fișierului pom.xml se actualizează proiectul *Maven* (click dreapta pe proiect *Maven > Update Project*)
- Se accesează proprietățile proiectului (prin click dreapta pe proiect și apoi ***Project Properties***) și se verifică ca în secțiunile ***Java Build Path***, ***Java Compiler*** și ***Project Facets*** să fie completată aceeași versiune de Java care a fost completată în fișierul ***pom.xml*** (dacă nu este atunci se modifică)



- Se accesează fișierul **web.xml** din proiect (cel mai simplu prin dublu click pe **Deployment Descriptor**)



- **web.xml** are conținutul de mai jos

```
<!DOCTYPE web-app PUBLIC
"-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd" >

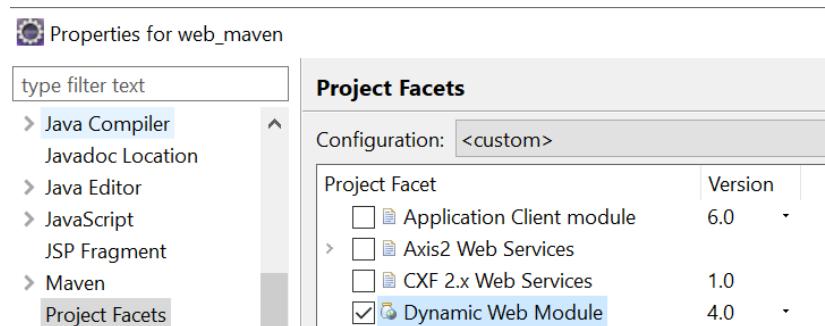
<web-app>
  <display-name>Archetype Created Web Application</display-name>
</web-app>
```

- Se înlocuiește acest conținut cu următorul actualizând în acest fel *Dynamic Web Module* de la versiunea 2.3 la versiunea 4.0

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://xmlns.jcp.org/xml/ns/javaee"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
  http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd"
  id="WebApp_ID" version="4.0">

  <display-name>Archetype Created Web Application</display-name>
</web-app>
```

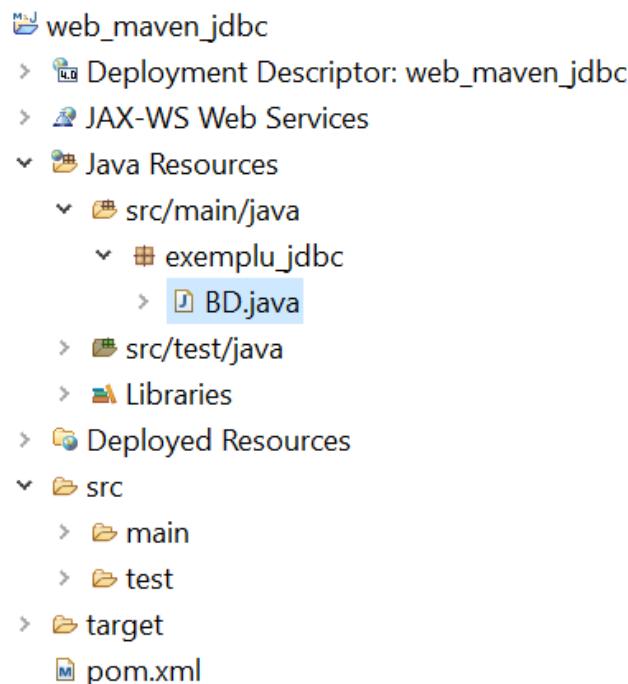
- După ce se fac schimbări asupra fișierelor *pom.xml* și *web.xml* trebuie actualizat proiectul *Maven* (click drepta pe proiect > *Maven* > *Update Project*)
- Se intră în proprietățile proiectului (click dreapta pe proiect și *Project Properties*) iar în secțiunea *Project Facets* se actualizează *Dynamic Web Module* la versiunea 4.0



- În versiuni mai vechi de Eclipse este posibil să nu se permite schimbarea versiunii la 4.0, atunci se deschide *Windows explorer* și se intră în workspace, nume proiect, subdirectorul **.settings** unde se găsește fișierul de mai jos

### **web\_maven\.settings\ org.eclipse.wst.common.project.facet.core.xml**

- Se modifică versiunea 2.3 în 4.0 și se salvează fișierul xml. În urma acestei modificări și secțiunea *Project Facets* din *Eclipse* trebuie aibă versiunea 4 pentru dacă *Dynamic web module*
- Se creează un servlet în proiect (click drepta pe scrc/main/java) și se alege New > Servlet. Clasa servletului se denumește *BD* și pachetul care o conține *exemplu\_jdbc*



- Servletul BD se conectează la baza de date test, preia datele din tabelul persoane și le afișează în format tabelar în browser. Servletul are următorul cod:

```
package exemplu_jdbc;

import java.io.IOException;
import java.io.PrintWriter;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet("/BD")
public class BD extends HttpServlet {
private static final long serialVersionUID = 1L;

    public BD() {
        super();
    }
}
```

```
protected void doGet(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
    try {
        String url = "jdbc:mysql://localhost:3306/test";
        Class.forName("com.mysql.cj.jdbc.Driver");
        Connection connection = DriverManager.getConnection(url, "root", "root");
        Statement statement = connection.createStatement();
        ResultSet rs = statement.executeQuery("select * from persoane");

        PrintWriter out=response.getWriter();
        out.println("<html><head><title>JDBC</title></head><body>");
        out.println("<table border='1' align='center' width='50%'>");
        out.println("<tr><th>Id</th><th>Nume</th><th>Varsta</th></tr>");

        while(rs.next()) {
            out.println("<tr><td>" +rs.getString(1)+"</td><td>" +rs.getString(2)
            +"</td><td>" +rs.getString(3)+"</td><tr>");
        }
        out.println("</table></body></html>");
        rs.close();
        statement.close();
        connection.close();
    } catch (SQLException | ClassNotFoundException e) {
        e.printStackTrace();
    }
}
```

<b>Id</b>	<b>Nume</b>	<b>Varsta</b>
1	Ana	19
2	Ionel	33
3	Oana	22

- În continuare se specifică în fișierul *web.xml* pagina de start a proiectului web ca fiind servleul BD. Se salvează fișierul *web.xml*, se actualizează proiectul *Maven* (click dreapta pe proiect *Maven* > *Update Project*) și se verifică rulând proiectul prin click dreapta pe proiect *Run as > Run on Server*

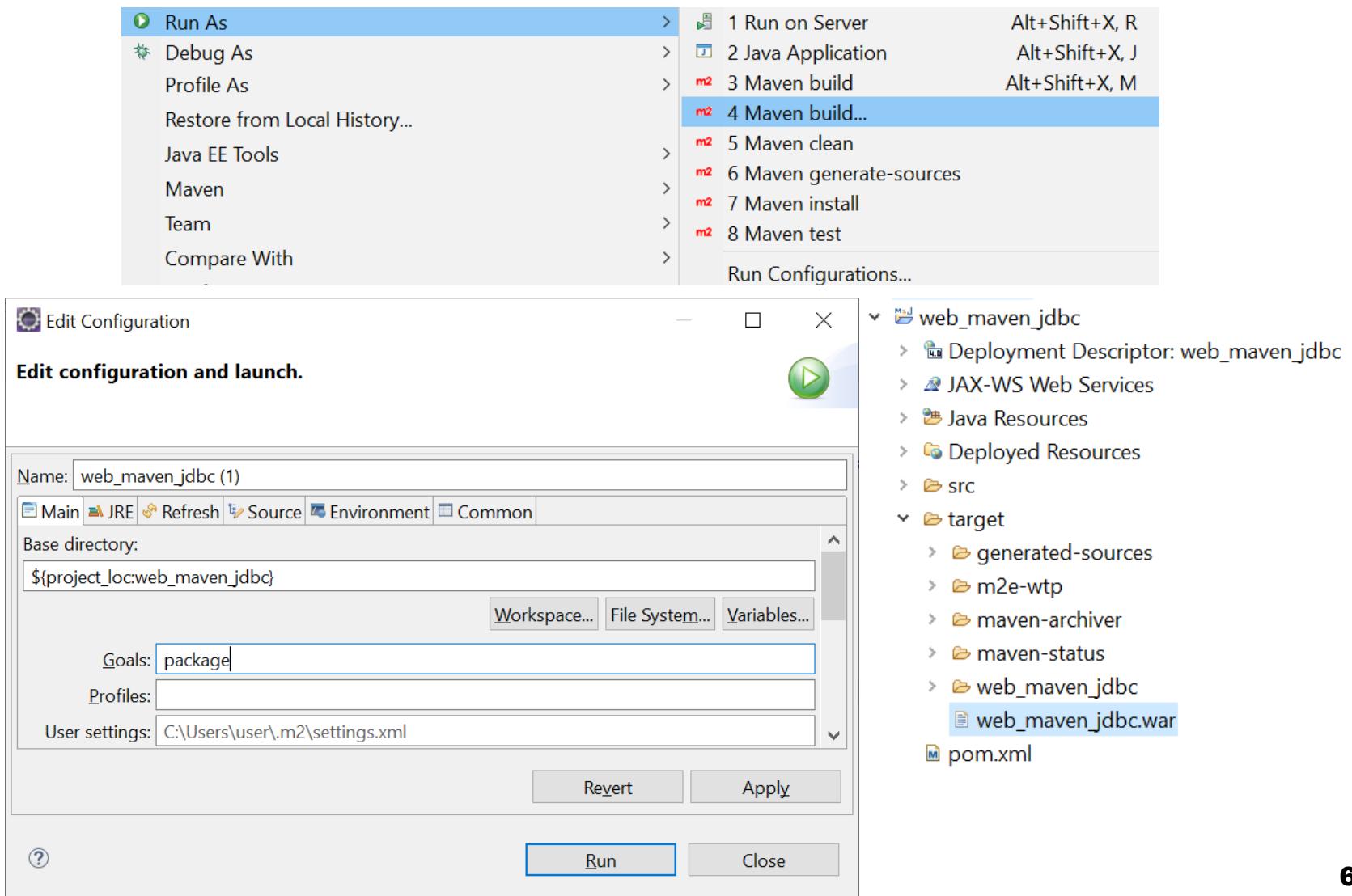
```

<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://xmlns.jcp.org/xml/ns/javaee"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
  http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd"
  id="WebApp_ID" version="4.0">

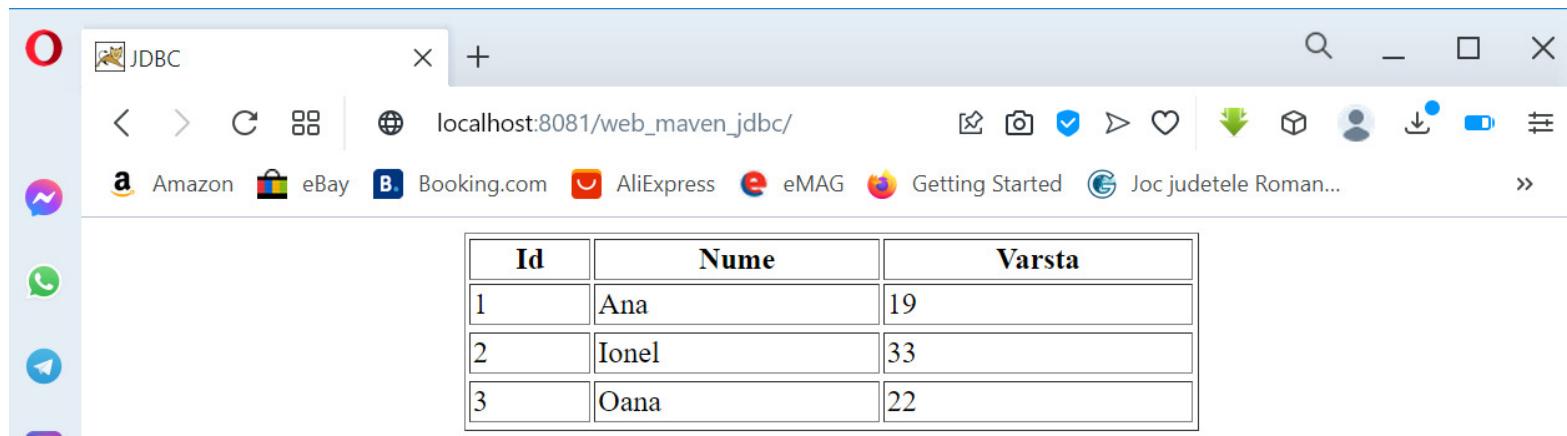
  <display-name>Archetype Created Web Application</display-name>
  <welcome-file-list>
    <welcome-file>BD</welcome-file>
  </welcome-file-list>
</web-app>

```

- În continuare se creează fișierul war. Într-un proiect Maven dezvoltat în Eclipse acesta se creează prin click dreapta pe proiect *Run as > Maven build...* și introducând la *Goals* valoarea **package**. Apoi se face click dreapta pe proiect și Refresh



- Fișierul *war* creat se copiază în directorul **webapps** din **ApacheTomcat**
- Se închide *Eclipse* și se pornește serverul web lansând **ApacheTomcat \ bin \ startup.bat** din line de comandă
- Se deschide un browser și se rulează fișierul war.  
[http://localhost:numar\\_port/Denumire\\_fisier\\_war/](http://localhost:numar_port/Denumire_fisier_war/)
- În captura de ecran de mai jos a fost rulat proiectul *web\_maven\_jdbc* care a fost configurat să-și inceapă execuția cu servletul *BD*

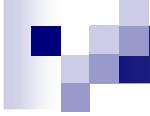




# 7. Spring Framework

Sl. dr. ing. Raul Robu

2022-2023, Semestrul 2



# CUPRINS

7.1 Noțiuni generale

7.2 Prințipiu de injectare al dependentelor

7.3 Injectarea dependentelor cu Spring framework

    7.3.1 Injectarea dependentelor cu ajutorul setterelor

    7.3.2 Injectarea dependentelor cu ajutorul constructorului

    7.3.3 Injectarea obiectelor

    7.3.4 Injectarea colecțiilor de obiecte

    7.3.5 Utilizarea atributului *scope*

    7.3.6 Relația de moștenire din cadrul definiției beanurilor

    7.3.7 Ciclul de viață al unui bean

    7.3.8 Utilizarea interfeței *BeanPostProcessor*

    7.3.9 Utilizarea *PropertyPlaceholderConfigurer*

    7.3.10 Utilizarea interfețelor

    7.3.11 Utilizarea anotațiilor `@Autowired` și `@Qualifier`

    7.3.12 Utilizarea anotațiilor Spring JSR-250

    7.3.13 Utilizarea anotațiilor `@Component` și `@Value`

7.4 Spring JDBC Framework

## 7.1 Noțiuni generale

- Spring framework este o platformă Java Open Source, care permite dezvoltarea ușoară și rapidă a unor aplicații Java robuste
- Spring framework a fost inițial scris de Rod Johnson în 2003
- Spring framework furnizează un număr mare de module care pot fi folosite în funcție de cerințele aplicațiilor, dar el este popular pentru facilitățile de injectare a dependențelor (dependency injection)
- Termenul “dependență” desemnează o asociere dintre două clase. Clasa A este dependență de clasa B dacă în clasa A avem o referință la un obiect de tip B. Termenul “Injectare” presupune că obiectele clasei B vor fi injectate în clasa A. Injectarea se poate realiza prin intermediul constructorului sau cu ajutorul setterelor.
- Când scriem aplicații Java complexe, clasele aplicației ar trebui să fie cât de independente se poate de alte clase Java pentru a crește posibilitatea de a refolosi aceste clase și de a le testa independent de alte clase atunci când facem unități de testare

## 7.2 Principiul de injectare a dependentelor

- În cazul în care se dorește realizarea unui program care calculează aria unor figuri geometrice se poate crea o clasă pentru fiecare figură. În acea clasă se introduce o metodă `aria()` care calculează aria figurii

```
package ex1;

class Cerc {
    public void aria() {
        System.out.println("Aria cercului");
    }
}
class Dreptunghi {
    public void aria() {
        System.out.println("Aria dreptunghiului");
    }
}

class MainApp{
    public static void main(String[]args){
        Cerc c=new Cerc();
        c.aria();

        Dreptunghi d=new Dreptunghi();
        d.aria();
    }
}
```

- În încercarea de a obține independentă față de clasele unor tipuri particulare de figuri, putem utiliza polimorfismul

```
package ex2;

abstract class Figura {
    public abstract void aria();
}

class Cerc extends Figura{
    @Override
    public void aria() {
        System.out.println("Aria cercului");
    }
}

class Dreptunghi extends Figura{
    @Override
    public void aria() {
        System.out.println("Aria dreptunghiului");
    }
}

class MainApp{
    public static void main(String[] args) {
        Figura c=new Cerc();
        c.aria();

        Figura d=new Dreptunghi();
        d.aria();
    }
}
```

- În continuare clasa aplicației este legată de clasele Cerc și Dreptunghi.

- Metoda **calculeaza\_aria()** este independentă de tipul figurii:

```
package ex3;

abstract class Figura {
    public abstract void aria ();
}

class Cerc extends Figura{
    @Override
    public void aria() {
        System.out.println("Aria cercului");
    }
}

class Dreptunghi extends Figura{
    @Override
    public void aria() {
        System.out.println("Aria dreptunghiului");
    }
}

class MainApp{
    public static void main(String[]args){
        calculeaza_aria(new Cerc());
        calculeaza_aria(new Dreptunghi());
    }

    public static void calculeaza_aria(Figura f){
        f.aria();
    }
}
```

- Clasa **ex4.Aria** este independentă de tipul figurii pentru care calculează aria
- Clasa Aria poate calcula și afișa aria oricărei figuri care extinde clasa abstractă *Figura*, fără să existe nici o dependență între această clasă și un tip particular de figură
- Obiectul corespunzător figurii pentru care se va calcula aria este “injectat” în clasă cu ajutorul setterului. Acesta este principiul injectării dependențelor
- Injectarea dependențelor are ca scop decuplarea obiectelor, limitarea legăturilor dintre ele

```
package ex4;

abstract class Figura {
    public abstract void aria ();
}

class Aria{
    private Figura f;
    public void setFigura(Figura f){
        this.f=f;
    }
    public void calculeaza_aria(){
        this.f.aria();
    }
}

class Cerc extends Figura{
    @Override
    public void aria() {
        System.out.println("Aria cercului");
    }
}
```

```
class Dreptunghi extends Figura{
    @Override
    public void aria() {
        System.out.println("Aria dreptunghiului");
    }
}

class MainApp{
    public static void main(String[]args){
        Aria a=new Aria();
        a.setFigura(new Cerc());
        a.calculeaza_aria();
    }
}
```

- Clasa aplicației este cea care injectează dependențele

## 7.3 Injectarea dependentelor cu Spring Framework

### 7.3.1 Injectarea dependentelor su ajutorul setterelor

- Nucleul lui *Spring framework* este reprezentat de ***Spring container***. Containerul are rolul de a crea obiecte, de a le interconecta, de a le configura, de a gestiona întreg ciclul lor de viață de la creare la distrugere
- Containerul Spring folosește injectarea dependentelor pentru a gestiona componentele care compun o aplicație. Aceste obiecte se numesc *Spring beans*
- Containerul își preia instrucțiunile cu privire la ce obiecte să instanțieze citind metadatele de configurare furnizate. Aceste metadate pot fi furnizate cu ajutorul fișierului XML, cu ajutorul adnotățiilor sau a codului Java
- Spring furnizează următoarele două tipuri de containere:
  - ***BeanFactory*** – este cel mai simplu container care frunizeaza suport pentru injectarea dependentelor și este definit de interfața *org.springframework.beans.factory.BeanFactory*. Cea mai comună implementare a acestei interfețe este *XmlBeanFactory* care preia metadatele de configurare dintr-un fișier XML
  - ***ApplicationContext*** – include toata funcționalitatea lui *BeanFactory* și alte funcționalități în plus, cum ar fi abilitatea de a rezolva mesajele textuale dintr-un fișier de proprietăți și abilitatea de a publica evenimentele aplicației către ascultătorii corespunzători. Acest container este definit de interfața *org.springframework.context.ApplicationContext*

- Un **Spring bean** este un obiect care este instantiat, asamblat și gestionat de un container Spring. Containerul creează astfel de obiecte pe baza metadatelor de configurare furnizate printr-un fișier XML, prin intermediul tagului *bean*
- În continuare se creează un proiect *Maven* și se completează în fișierul *pom.xml* versiunea de Java utilizată pentru compilarea proiectului și dependențele necesare rulării proiectului. Dependențele s-au obținut din depozitul Maven central <https://mvnrepository.com>.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>siaps.group</groupId>
    <artifactId>spring_maven</artifactId>
    <version>0.0.1-SNAPSHOT</version>

    <properties>
        <maven.compiler.target>15</maven.compiler.target>
        <maven.compiler.source>15</maven.compiler.source>
    </properties>
```

```
<dependencies>
    <!-- https://mvnrepository.com/artifact/org.springframework/spring-core -->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-core</artifactId>
        <version>5.3.24</version>
    </dependency>
    <!-- https://mvnrepository.com/artifact/org.springframework/spring-context -->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>5.3.24</version>
    </dependency>
</dependencies>
</project>
```

- Fișierul *pom.xml* se salvează și apoi se actualizează proiectul *Maven* (în *Eclipse* click dreapta pe proiect și apoi *Maven > Update Project...*, iar în *IntelliJ Load Maven Changes (Ctrl + Shift +O)*)
- Dependențele necesare rulării proiectului sunt aduse din depozitul *Maven* central în cel local și apoi încărcate în proiect

- Se creează clasele *Dreptunghi* și *MainApp* în pachetul *exemplul01* și fișierul *beans01.xml*

```

package exemplul01;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.context.support.ClassPathXmlApplicationContext;

class Dreptunghi {
    private int lungime;
    private int latime;
    public void aria() {
        System.out.println("Aria dreptunghiului: "+(lungime*latime));
    }
    public int getLungime() { return lungime; }
    public void setLungime(int lungime) {
        this.lungime = lungime;
    }
    public int getLatime() { return latime; }
    public void setLatime(int latime) {
        this.latime = latime;
    }
}

class MainApp{
    public static void main(String []args){
        BeanFactory factory=new ClassPathXmlApplicationContext("beans01.xml");
        Dreptunghi d=(Dreptunghi) factory.getBean("dreptunghi");
        d.aria();
    }
}

```

Aria dreptunghiului: 6

- Fișierul **beans01.xml**, amplasat în directorul **src/main/resources**:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

    <bean id="dreptunghi" class="exemplul01.Dreptunghi">
        <property name="Lungime" value="2"/>
        <property name="Latime" value="3"/>
    </bean>
</beans>
```

- Tagul *beans* se amplasează la începutul fișierului spring de configurare. Acesta are attribute care îi specifică parserului XML din java de unde să își ia informațiilele necesare pentru a valida fisierul XML.
- Parserul XML din Java va citi valoarea atributului schemaLocation și va încerca să acceseze schema de pe Internet cu scopul de a valida fișierul XML. Spring va intercepta această cerere de accesare și va servi informatiile necesare din interiorul jar-urilor sale.
- Tagul bean are atributul *class*, care reprezintă clasa obiectelor create de container și atributul *id*, prin care poate fi identificat beanul

- Prin intermediul celor două proprietăți *lungime* și *latime* se dă valori variabilelor membre cu aceeași denumire din clasa *Dreptunghi*
- Valorile specificate pentru lungime și latime sunt injectate în clasa dreptunghi cu ajutorul setterelor
- În programul principal se folosește metoda *getBean()* a interfeței *BeanFactory* pentru a crea un obiect de tip *Dreptunghi* instantiat prin intermediul *setterelor* cu valorile complete din fișierul XML
- Obiectul de tip ***Dreptunghi*** din programul principal a fost instantiat fără a folosi operatorul ***new***
- În mod similar se pot crea obiecte instantiate cu aceleasi valori în oricâte fișiere

## 7.3.2 Injectarea dependențelor cu ajutorul constructorului

```
package exemplul02;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

class Dreptunghi {
    private int lungime;
    private int latime;

    public Dreptunghi(int lungime, int latime) {
        this.lungime = lungime;
        this.latime = latime;
    }
    public void aria() {
        System.out.println("Aria dreptunghiului: "+(lungime*latime));
    }
}

class MainApp {
    public static void main(String []args) {
        ApplicationContext context=new ClassPathXmlApplicationContext("beans02.xml");
        Dreptunghi d=(Dreptunghi) context.getBean("dreptunghi");
        d.aria();
    }
}
```

- Fișierul **beans02.xml**, amplasat în directorul **src/main/resources**:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

    <bean id="dreptunghi" class="exemplul02.Dreptunghi">
        <constructor-arg value="2"/>
        <constructor-arg value="3"/>
    </bean>
</beans>
```

- În programul principal a fost creat un bean folosind containerul *ApplicationContext*
- Pentru ca injectarea dependențelor să se facă prin constructor în fișierul *xml* a fost folosit tagul *constructor-arg*
- Tagul *constructor-arg* fost scris de două ori pentru a da valori celor 2 parametri
- Valorile au fost completeate ca siruri de caractere (punându-le între ghilimele), frameworkul fiind cel care le convertește la întregi
- Dacă în clasă am avea încă un constructor care primește ca și parametri două stringuri, frameworkul nu ar stii pe care din cei doi constructori să-i apeleze

- Această situație se poate rezolva utilizând atributul *type*
- În exemplul de mai jos apelăm constructorul cu 2 parametri întregi

```
//...
<constructor-arg type="int" value="2"/>
<constructor-arg type="int" value="3"/>
```

- În situația în care se dorește precizarea poziției parametrului care primește o anumită valoare se poate utiliza atributul *index*
- Valoarea unui parametru a constructorului poate fi precizată cu ajutorul atributului *value* sau a tagului *value*

```
//...
<constructor-arg index="0">
    <value>2</value>
</constructor-arg>
<constructor-arg index="1" value="3"/>
```

### 7.3.3 Injectarea obiectelor

- Fișierul **Persoana.java**

```
package exemplul03;

public class Persoana {
    private String nume;
    private int varsta;
    private Adresa adresa;

    public String getNume() { return nume; }
    public void setNume(String nume) { this.nume = nume; }

    public int getVarsta() {return varsta;}
    public void setVarsta(int varsta) {this.varsta = varsta;}

    public Adresa getAdresa() {return adresa;}
    public void setAdresa(Adresa adresa) {this.adresa = adresa; }

    @Override
    public String toString() {
        return nume + ", " + varsta + ", "+adresa.toString();
    }
}
```

- Fișierul **Adresa.java**

```
package exemplul03;
public class Adresa {
    private int nr;
    private String strada;
    private String localitatea;
    public Adresa(int nr, String strada, String localitatea) {
        this.nr = nr; this.strada = strada; this.localitatea = localitatea;
    }
    public String toString() {
        return nr + ", " + strada + ", " + localitatea;
    }
}
```

- Fișierul **MainApp.java**

```
package exemplul03;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
class MainApp {
    public static void main(String []args) {
        ApplicationContext context=new ClassPathXmlApplicationContext("beans03_1.xml");
        //ApplicationContext context=new ClassPathXmlApplicationContext("beans03_2.xml");
        //ApplicationContext context=new ClassPathXmlApplicationContext("beans03_3.xml");
        Persoana p=(Persoana)context.getBean("pers");
        System.out.print(p);
    }
}
```

- Fișierul **beans03\_1.xml**

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

    <bean id="pers" class="exemplul03.Persoana">
        <property name="nume" value="Popescu"/>
        <property name="varsta" value="20"/>
        <property name="adresa" ref="adr"/>
    </bean>

    <bean id="adr" class="exemplul03.Adresa">
        <constructor-arg value="2"/>
        <constructor-arg value="Venus"/>
        <constructor-arg value="Timisoara"/>
    </bean>
</beans>
```

- A fost creat câte un *bean*, atât pentru obiectul de tip *Adresa* cât și pentru cel de tip *Persoana*
- *numele*, *varsta* și *adresa* au fost injectate în obiectul de tip *Persoana* prin intermediul setterelor

- *numarul, starda și localitatea* au fost injectate în obiectul de tip *Adresa* prin intermediul constructorului
- Injectarea obiectului *adresa* în obiectul *persoana* s-a realizat specificand prin intermediul atributului *ref*, referință către bean-ul *adr*
- O alternativă la utilizarea atributului *ref* o constituie lucru cu *bean-uri interioare*. Fișierul ***beans03\_2.xml***:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

    <bean id="pers" class="exemplul03.Persoana">
        <property name="nume" value="Popescu"/>
        <property name="varsta" value="20"/>
        <property name="adresa">
            <bean class="exemplul03.Adresa">
                <constructor-arg value="2"/>
                <constructor-arg value="Venus"/>
                <constructor-arg value="Timisoara"/>
            </bean>
        </property>
    </bean>
</beans>
```

- În exemplul precedent atributul *ref* a proprietății *adresa* a fost șters, iar *Beanul* corespunzător adresei a fost introdus în proprietatea *adresa*
- Întrucât atributul *id* al bean-ului corespunzator adresei nu mai este utilizat, se poate renunța la acest atribut
- Dacă pentru bean-uri se folosesc aceleași denumiri pe care le au variabilele membre din clasă, se poate configura atributul **autowire** astfel încât *Spring* va face singur legătura între variabilele membre și beanuri:
- Fișierul **beans03\_3.xml**:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

    <bean id="pers" class="exemplul03.Persoana" autowire="byName">
        <property name="nume" value="Popescu"/>
        <property name="varsta" value="20"/>
    </bean>
    <bean id="adresa" class="exemplul03.Adresa">
        <constructor-arg value="2"/>
        <constructor-arg value="Venus"/>
        <constructor-arg value="Timisoara"/>
    </bean>
</beans>
```

- În fișierul **beans03\_3.xml**, pentru bean-ul *pers* a fost configurat atributul autowire la valoarea **byName**
- În consecință, în situația în care nu se folosesc configurații explicate, *Spring* va lega automat variabilele membre din clasă de beanurile cu aceleași nume. Astfel beanul “adresa” va fi folosit pentru a injecta obiectul cu același nume în clasa *Persoana*

### 7.3.4 Injectarea colecțiilor de obiecte

- În fișierul XML, se foloșete unul din tagurile **list**, **set**, **map** sau **props** în funcție de tipul colecției
- În interiorul tagului colecției pot fi introduse:
  - Taguri **ref**, pentru a face referire la bean-uri din afara colecției
  - Bean-uri interioare, corespunzătoare obiectelor care alcătuiesc colecția
  - Taguri value, pentru situația în care colecția este de Stringuri sau de valori numerice
- Se consideră exemplul precedent, cu mențiunea că o persoană poate avea mai multe adrese
- Pentru a reține adresele s-a ales o colecție de tip **Set**

- Fișierul **Persoana.java**

```
package exemplul04;
import java.util.Set;
import exemplul03.Adresa;

public class Persoana {
    private String nume;
    private int varsta;
    private Set<Adresa> adrese;

    public String getNume() { return nume; }
    public void setNume(String nume) { this.nume = nume; }

    public int getVarsta() { return varsta; }
    public void setVarsta(int varsta) { this.varsta = varsta; }

    public Set<Adresa> getAdrese() { return adrese; }
    public void setAdrese(Set<Adresa> adrese) { this.adrese = adrese; }
    @Override
    public String toString() {
        return nume + ", " + varsta + ", "+adrese.toString();
    }
}
```

- Clasa *Persoana* reutilizează clasa *Adresa* din pachetul *exemplul03* care este importată. Fișierul **MainApp.java** este similar cu cel de la exemplul precedent, cu mențiunea că își extrage metadatele de configurare din fișierul *beans04.xml*

- Fișierul **beans04.xml**:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

    <bean id="pers" class="exemplul04.Persoana">
        <property name="nume" value="Popescu"/>
        <property name="varsta" value="20"/>
        <property name="adrese">
            <set>
                <ref bean="adr1"/>
                <ref bean="adr2"/>
            </set>
        </property>
    </bean>
    <bean id="adr1" class="exemplul03.Adresa">
        <constructor-arg value="2"/>
        <constructor-arg value="Venus"/>
        <constructor-arg value="Timisoara"/>
    </bean>
    <bean id="adr2" class="exemplul03.Adresa">
        <constructor-arg value="3"/>
        <constructor-arg value="Uranus"/>
        <constructor-arg value="Timisoara"/>
    </bean>
</beans>
```

- În locul utilizării tagului *ref* ar fi putut fi introduse cele două beanuri corespunzătoare celor două adrese, direct în colecția *Set*

### 7.3.5 Utilizarea atributului **scope**

- În fișierul XML de configurare, putem să utilizăm atributul **scope**, al tagului bean, pentru a determina *Spring* să:
  - Creeze o nouă instanță a obiectului **bean**, oricând este nevoie de una. Acest lucru se va întâmpla când *scope* primește valoarea **prototype**
  - Returneze aceeași instanță a obiectului **bean**, atunci când este nevoie de ea. Acest lucru se întâmplă când atributul *scope* primește valoarea **singleton**
- Fișierul **Mesaj.java**:

```
package exemplul05;

public class Mesaj {
    private String mesaj;

    public String getMesaj() {
        return mesaj;
    }
    public void setMesaj(String mesaj) {
        this.mesaj = mesaj;
    }
    @Override
    public String toString() {
        return mesaj;
    }
}
```

- Fișierul **beans05.xml**:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

    <bean id="mesaj" class="exemplul05.Mesaj" scope="singleton">
        <property name="mesaj" value="Primul mesaj!" />
    </bean>
</beans>
```

- Fișierul **MainApp.java**:

```
package exemplul05;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

class MainApp {
    public static void main(String []args) {
        ApplicationContext context=new ClassPathXmlApplicationContext("beans05.xml");
        Mesaj m1=(Mesaj) context.getBean("mesaj");
        System.out.println("Obiectul m1 dupa creare:"+m1);

        m1.setMesaj("Al doilea mesaj!");
        System.out.println("Obiectul m1 dupa setare:"+m1);

        Mesaj m2=(Mesaj) context.getBean("mesaj");
        System.out.println("Obiectul m2 dupa creare:"+m2);
    }
}
```

- Rularea programului când **scope=“singleton”** (în mod implicit atributul **scope** are valoarea **singleton**)

```
Obiectul m1 dupa creare:Primul mesaj!
Obiectul m1 dupa setare:Al doilea mesaj!
Obiectul m2 dupa creare:Al doilea mesaj!
```

- Rularea programului când **scope=“prototype”**

```
Obiectul m1 dupa creare:Primul mesaj!
Obiectul m1 dupa setare:Al doilea mesaj!
Obiectul m2 dupa creare:Primul mesaj!
```

## 7.3.6 Relația de moștenire din cadrul definiției beanurilor

- Definiția unui bean poate să conțină foarte multe informații de configurare, cum ar fi argumentele ale constructorului sau valorile unor proprietăți
- Un bean copil moștenește metadatele de configurare de la beanul părinte
- Un bean copil poate să suprascrie unele valori sau să adauge altele, după cum este nevoie
- Relația de moștenire din cadrul definiției bean-urilor (bean definition inheritance) nu obligă la existența unei relații de moștenire între clasele corespunzătoare bean-urilor, deși aceasta poate să existe
- Conceptul de moștenire din cadrul definiției beanurilor, este același cu cel de moștenire al claselor din java
- Într-un fișier de configurare xml se poate specifica un bean copil dacă se utilizează atributul *parent* și se specifică cu ajutorul acestuia id-ul bean-ului părinte.

- Fișierul **DouaMesaje.java**:

```
package exemplul06;

class DouaMesaje {
    private String mesaj1;
    private String mesaj2;

    public DouaMesaje() {}

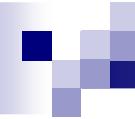
    public String getMesaj1() {
        return mesaj1;
    }

    public void setMesaj1(String mesaj1) {
        this.mesaj1 = mesaj1;
    }

    public String getMesaj2() {
        return mesaj2;
    }

    public void setMesaj2(String mesaj2) {
        this.mesaj2 = mesaj2;
    }

    @Override
    public String toString() {
        return mesaj1 + ", " + mesaj2;
    }
}
```



## ■ Fișierul *TreiMesaje.java*:

```
package exemplul06;

class TreiMesaje {
    private String mesaj1;
    private String mesaj2;
    private String mesaj3;

    public TreiMesaje() {}

    public String getMesaj1() {
        return mesaj1;
    }
    public void setMesaj1(String mesaj1) {
        this.mesaj1 = mesaj1;
    }
    public String getMesaj2() {
        return mesaj2;
    }
    public void setMesaj2(String mesaj2) {
        this.mesaj2 = mesaj2;
    }
    public String getMesaj3() {
        return mesaj3;
    }
    public void setMesaj3(String mesaj3) {
        this.mesaj3 = mesaj3;
    }
    @Override
    public String toString() {
        return mesaj1 + ", " + mesaj2+ ", " + mesaj3;
    }
}
```

- Fișierul **beans06.xml**:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

    <bean id="mes1" class="exemplul06.DouaMesaje">
        <property name="mesaj1" value="1 din 2"/>
        <property name="mesaj2" value="2 din 2"/>
    </bean>

    <bean id="mes2" class="exemplul06.TreiMesaje" parent="mes1">
        <property name="mesaj1" value="1 din 3"/>
        <property name="mesaj3" value="3 din 3"/>
    </bean>
</beans>
```

- Fișierul **MainApp**:

```
package exemplul06;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApp {
    public static void main(String []args){
        ApplicationContext context=new ClassPathXmlApplicationContext("beans06.xml");
        DouaMesaje m1=(DouaMesaje) context.getBean("mes1");
        System.out.println(m1);
        TreiMesaje m2=(TreiMesaje) context.getBean("mes2");
        System.out.println(m2);
    }
}
```

1 din 2, 2 din 2  
1 din 3, 2 din 2, 3 din 3

- Clasele *DouaMesaje* și *TreiMesaje* nu sunt legate prin relație de moștenire
- Bean-ul *mes2* este beanul copil și moștenește metadatele de configurare de la beanul parinte *mes1*
- *Proprietatea mesaj2 primește valoarea stabilită în cadrul beanului mes1*
- *Între clasele Java poate exista relație de moștenire, vezi exemplul următor*

- Fișierul **Persoana.java**:

```
package exemplul07;

class Persoana {
    private String nume;
    private int varsta;

    public Persoana(){}
    public Persoana(String nume, int varsta) {
        this.nume = nume;
        this.varsta = varsta;
    }
    @Override
    public String toString() {
        return nume + ", " + varsta;
    }
}

class Angajat extends Persoana{
    private int vechime;
    public Angajat(){}
    public Angajat(String nume, int varsta, int vechime) {
        super(nume, varsta);
        this.vechime = vechime;
    }
    @Override
    public String toString() {
        return super.toString() + ", "+vechime;
    }
}
```

- Fișierul **beans07.xml**:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

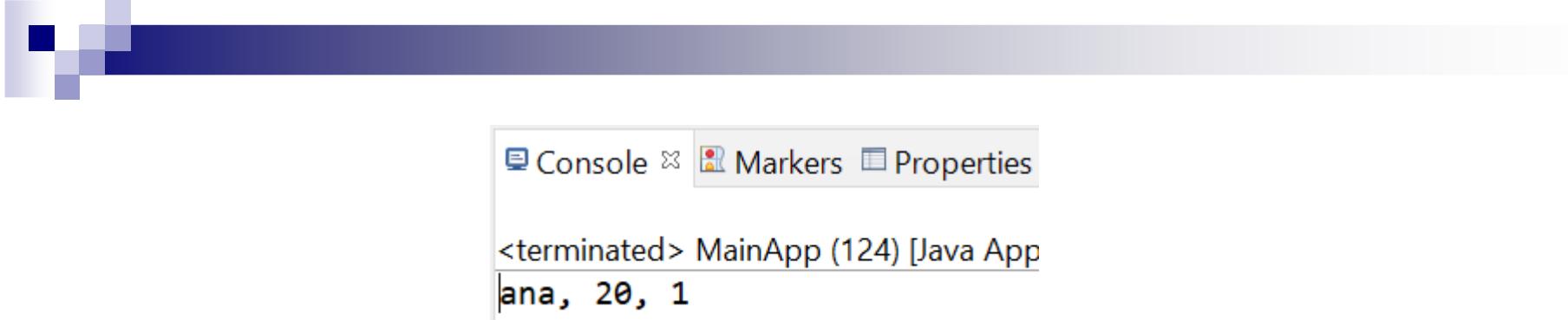
    <bean id="pers" class="exemplul07.Persoana">
        <constructor-arg value="ana"/>
        <constructor-arg value="20"/>
    </bean>

    <bean id="angajat" class="exemplul07.Angajat" parent="pers">
        <constructor-arg value="1"/>
    </bean>
</beans>
```

- Fișierul **MainApp**:

```
package exemplul07;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApp {
    public static void main(String []args) {
        ApplicationContext context=new ClassPathXmlApplicationContext("beans07.xml");
        Persoana p=(Angajat)context.getBean("angajat");
        System.out.print(p);
    }
}
```



The screenshot shows the Eclipse IDE interface with the 'Console' tab selected. The output window displays the text: '<terminated> MainApp (124) [Java App]' followed by the values 'ana, 20, 1'.

- Cele două clase *Angajat* și *Persoana* sunt legate prin relație de moștenire
- Variabilele membru *nume* și *varsta* primesc valori prin constructorul beanului părinte
- Variabila membru *vechime* primește valori prin intermediul constructorului beanului copil

### 7.3.7 Ciclul de viață al unui bean

- Când un bean se instantiază, poate fi necesar, în anumite situații, să facem anumite prelucrări specifice
- În mod similar, pot fi necesare unele prelucrări când beanul nu mai este necesar și este eliminat din container
- Atributele ***init-method*** și ***destroy-method*** ale tagului bean, permit specificarea unor metode din clasa bean-ului care se vor executa imediat ce obiectul este instantiat, respectiv imediat ce obiectul este eliminat din container
- Clasa beanului poate implementa interfețele ***InitializingBean*** (conține metoda ***afterPropertiesSet()*** care se apelează când beanul se initializează) și ***DisposableBean*** (conține metoda ***destroy()*** care se apelează când beanul se distrugă)
- Configurările realizate la nivelul documentului XML, permit stabilirea denumirilor dorite pentru metodele de initializare, respectiv distrugere

- Fișierul **Dreptunghi.java**:

```
package exemplul08;

import org.springframework.beans.factory.DisposableBean;
import org.springframework.beans.factory.InitializingBean;

class Dreptunghi implements InitializingBean,DisposableBean{
    private int lungime;
    private int latime;

    public Dreptunghi(int lungime, int latime) {
        this.lungime = lungime;
        this.latime = latime;
    }
    public void aria() {
        System.out.println("Aria dreptunghiului: "+(lungime*latime));
    }
    public void init(){
        System.out.println("init()");
    }
    public void disp(){
        System.out.println("disp()");
    }
    @Override
    public void afterPropertiesSet() throws Exception {
        System.out.println("InitializingBean - afterPropertiesSet()");
    }
    @Override
    public void destroy() throws Exception {
        System.out.println("DisposableBean - destroy()");
    }
}
```

- Fișierul **Dreptunghi.java**:

```
package exemplul08;

import org.springframework.context.support.AbstractApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

class MainApp {
    public static void main(String []args){
        AbstractApplicationContext context=new ClassPathXmlApplicationContext("beans08.xml");
        Dreptunghi d=(Dreptunghi)context.getBean("dreptunghi");
        d.aria();
        context.registerShutdownHook();
        context.close();
    }
}
```

- Fișierul **beans08.xml**:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

    <bean id="dreptunghi" class="exemplul08.Dreptunghi" init-method="init" destroy-method="disp">
        <constructor-arg value="2"/>
        <constructor-arg value="3"/>
    </bean>
</beans>
```



The screenshot shows the Eclipse IDE's Console view with the following log output:

```
<terminated> MainApp (125) [Java Application] D:\kituri diverse\eclipse-jee-  
InitializingBean - afterPropertiesSet()  
init()  
Aria dreptunghiului: 6  
DisposableBean - destroy()  
disp()
```

- ApplicationContext este interfață
- AbstractApplicationContext este clasă abstractă
- AbstractApplicationContext implementeaza mai multe interfețe printre care și ApplicationContext, BeanFactory, ConfigurableApplicationContext, etc.
- ***ConfigurableApplicationContext*** are metoda ***registerShutdownHook()***, care apelează metodele de distrugere relevante

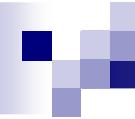
## 7.3.8 Utilizarea interfeței *BeanPostProcessor*

- Dacă vrem ca frameworkul să facă aceleasi prelucrări după ce inițializează fiecare bean, indifferent de tipul acestuia, putem să utilizăm interfața *BeanPostProcessors*
- Metodele interfeței BeanPostProcessor o să ruleze pentru fiecare bean pe care îl avem fără să conteze câte beanuri avem sau care este tipul acestora
- Realizăm o clasă care implementează interfața BeanPostProcessor
- Fișierul ***AnyBeanPostProcessor.java***:

```
package exemplul09;

import org.springframework.beans.BeansException;
import org.springframework.beans.factory.config.BeanPostProcessor;

class AnyBeanPostProcessor implements BeanPostProcessor{
    @Override
    public Object postProcessAfterInitialization(Object bean, String beanName)
        throws BeansException {
        System.out.println("Bean name after initialization:" +beanName);
        return bean;
    }
    @Override
    public Object postProcessBeforeInitialization(Object bean, String beanName)
        throws BeansException {
        System.out.println("Bean name before initialization:" +beanName);
        return bean;
    }
}
```



## ■ Fișierul *Dreptunghi.java*

```
package exemplul09;

class Dreptunghi {
    private int lungime;
    private int latime;

    public Dreptunghi(int lungime, int latime) {
        this.lungime = lungime;
        this.latime = latime;
    }
    public void aria() {
        System.out.println("Aria dreptunghiului: "+(lungime*latime));
    }
}
```

## ■ Fișierul *Triunghi.java*

```
package exemplul09;

class Triunghi {
//...
}
```

- Fișierul **beans09\_1.xml**:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

    <bean id="dreptunghi" class="exemplul09.Dreptunghi">
        <constructor-arg value="2"/>
        <constructor-arg value="3"/>
    </bean>

    <bean id="triunghi" class="exemplul09.Triunghi"/>

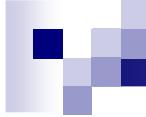
    <bean class="exemplul09.AnyBeanPostProcessor"/>
</beans>
```

- Fișierul **MainApp.java**

```
package exemplul09;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

class MainApp {
    public static void main(String []args) {
        ApplicationContext context=new ClassPathXmlApplicationContext("beans09_1.xml");
        //ApplicationContext context=new ClassPathXmlApplicationContext("beans09_2.xml");
        Dreptunghi d=(Dreptunghi)context.getBean("dreptunghi");
        d.aria();
    }
}
```



```
Console × Markers Properties Servers Data Source Explorer  
<terminated> MainApp (126) [Java Application] D:\kituri diverse\eclipse-jee-  
Bean name before initialization:dreptunghi  
Bean name after initialization:dreptunghi  
Bean name before initialization:triunghi  
Bean name after initialization:triunghi  
Aria dreptunghiului: 6
```

- Aşa cum arată captura de la rulare, cele două metode de initializare se rulează pentru fiecare bean declarat în fişierul XML, chiar dacă în programul principal nu se creează obiecte decât pentru unul din beanuri
- Fişierul **xml** de configurare conține un bean a cărui clasă implementează interfaţa *BeanPostProcessor*
- Pentru acest *bean* nu a fost ales un id, întrucât el nu va fi referit prin *id*
- Oricâte beanuri ar fi în fisierul xml de configurare și oricare ar fi tipul acestor beanuri, după initializarea fiecărui bean se vor apela cele două metode din clasa ce implementează *BeanPostProcessor*
- Metodele din clasa ce implementează interfaţa *BeanPostProcessor* se rulează pentru fiecare bean din fişierul XML în ordinea apariției, la instantierea obiectului de tip *ApplicationContext*

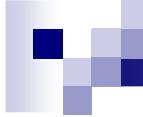
### 7.3.9 Utilizarea *PropertyPlaceholderConfigurer*

- Valorile de configurare pot să nu fie introduse în fișierul XML ci în afara acestuia, într-un fișier de configurare
- În acest caz în fișierul de configurare XML, în locul valorilor se vor utiliza substituenți (placeholders), care vor fi introdusi prin notația “ ***\${placeholder}***”
- Se va crea un fișier de configurare în care se vor introduce pe câte un rând valorile în formatul ***placeholder=valoare***
- Fișierul ***beans09\_2.xml***:

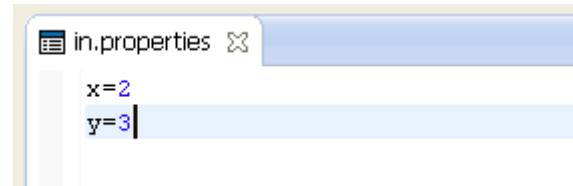
```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

    <bean id="dreptunghi" class="exemplul09.Dreptunghi">
        <constructor-arg value="${x}" />
        <constructor-arg value="${y}" />
    </bean>

    <bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
        <property name="locations" value="in.properties"/>
    </bean>
</beans>
```



- Fișierul ***in.properties***:



- Se va crea un bean a cărui clasă este ***PropertyPlaceholderConfigurer***. Această clasă va inspecta fișierul de configurare, apoi fisierul XML și va înlocui placeholderii cu valorile din fișierul de configurare
- Specificarea fișierului de configurare în care sunt completate valorile se face cu ajutorul proprietății ***locations***, care primește ca și valoare numele fișierului de configurare
- Se modifică clasa *exemplul09.MainApp* încât metadatele de configurare să fie preluate din fișierul *beans09\_2.xml* și se rulează programul

## 7.3.10 Utilizarea interfețelor

- Fișerul ***Figura.java***

```
package exemplul10;

interface Figura {
    public void aria();
}
```

- Fișierul ***Dreptunghi.java***

```
package exemplul10;

class Dreptunghi implements Figura{
    private int lungime;
    private int latime;

    public Dreptunghi(int lungime, int latime) {
        this.lungime = lungime;
        this.latime = latime;
    }
    public void aria() {
        System.out.println("Aria dreptunghiului: "+(lungime*latime));
    }
}
```

## ■ Fișierul **Cerc.java**

```
package exemplul10;

class Cerc implements Figura{
    private double raza;

    public void aria() {
        System.out.println("Aria cercului: "+Math.PI*raza*raza);
    }
    public double getRaza() {
        return raza;
    }
    public void setRaza(double raza) {
        this.raza = raza;
    }
}
```

## ■ Fișierul **beans10.xml**

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

    <bean id="dreptunghi" class="exemplul10.Dreptunghi">
        <constructor-arg value="2"/>
        <constructor-arg value="3"/>
    </bean>

    <bean id="cerc" class="exemplul10.Cerc">
        <property name="raza" value="2"/>
    </bean>
</beans>
```

- Fișierul **MainApp.java**

```
package exemplul10;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

class MainApp {
    public static void main(String []args)
    {
        ApplicationContext context=new ClassPathXmlApplicationContext("beans10.xml");
        //Figura f=(Figura)context.getBean("cerc");
        Figura f=(Figura)context.getBean("dreptunghi");
        f.aria();
    }
}
```

- Prin intermediul fișierului xml de configurare au fost injectate dependențele în clasa Dreptunghi cu ajutorul constructorului, iar în clasa Cerc cu ajutorul setterului
- Programul principal este dependent de interfață, nu de o implementare specifică a acesteia
- În programul principal nu se cunoaște aria cărei figuri se va calcula, se cunoaște doar că se va calcula aria unei figuri care implementează interfața Figura
- Extinderea programului cu facilități de a calcula aria altor figuri este foarte ușoară

### 7.3.11 Utilizarea adnotățiilor @Autowired și @Qalifier

- Injectarea dependențelor poate fi configurată cu ajutorul adnotățiilor
- Legăturile dintre beanuri pot fi specificate în fișierul XML sau în clasele java, cu ajutorul adnotățiilor
- Adnotatiile pot fi plasate deasupra claselor, metodelor sau declarațiilor de câmpuri
- Injectarea adnotățiilor se realizează înaintea injectării XML
- Pentru a putea utiliza adnotățiile este necesară adăugarea dependenței de *spring-aop* în fișierul *pom.xml*

```
<!-- https://mvnrepository.com/artifact/org.springframework/spring-aop -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-aop</artifactId>
    <version>5.3.24</version>
</dependency>
```

- Activarea utilizării adnotățiilor se realizează adăugând tagul **<context:annotation-config>** în fișierul XML de configurare
- Atributele tagului beans trebuie extinse, adăugând și atributul **xmlns:context**, precum în fișierul **beans11.xml** :

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
                           http://www.springframework.org/schema/context
                           http://www.springframework.org/schema/context/spring-context-3.0.xsd">

    <context:annotation-config/>

    <bean id="pers" class="exemplul11.Persoana">
        <property name="nume" value="Dana"/>
    </bean>
    <bean id="jobul" class="exemplul11.Job">
        <property name="firma" value="BestCompany"/>
        <property name="functia" value="sef"/>
    </bean>
</beans>
```

- În clasa `exemplul11.Persoana` adnotația `@Autowired` se amplasează deasupra setterului. Adnotația `@Autowired` poate fi amplasată deasupra *setterelor*, constructorilor sau variabilelor membre ale clasei

```
package exemplul11;

import org.springframework.beans.factory.annotation.Autowired;

class Persoana {
    private String nume;
    private Job job;

    public Persoana(){}

    public String getNume() {
        return nume;
    }
    public void setNume(String nume) {
        this.nume = nume;
    }
    public Job getJob() {
        return job;
    }
    @Autowired
    public void setJob(Job job) {
        this.job = job;
    }

    @Override
    public String toString() {
        return nume + ", " + job;
    }
}
```

- Fișierul **Job.java**:

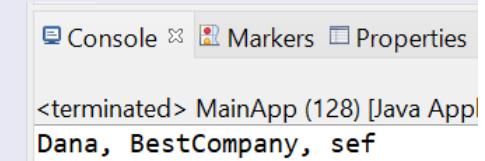
```
package exemplul11;

public class Job {
    private String firma;
    private String functia;
    public String getFirma() {
        return firma;
    }
    public void setFirma(String firma) {
        this.firma = firma;
    }
    public String getFunctia() {
        return functia;
    }
    public void setFunctia(String functia) {
        this.functia = functia;
    }
    public String toString() {
        return firma + ", " + functia;
    }
}
```

- Fișierul **MainApp.java** are următorul conținut:

```
package exemplul11;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

class MainApp {
    public static void main(String []args){
        ApplicationContext context=new ClassPathXmlApplicationContext("beans11.xml");
        Persoana p=(Persoana)context.getBean("pers");
        System.out.println(p);
    }
}
```



- Când Spring găsește o anotăție **@Autowired** deasupra unui setter, încearcă să realizeze legătura **byType**
- În mod implicit, anotația **@Autowired** implică că dependența este cerută. Se poate modifica comportamentul implicit dacă se utilizează opțiunea **@Autowired (required=false)**
- Se poate testa adăugând anotația **@Autowired (required=false)** în clasa **exemplul11.Job** deasupra setterului **setFirma(String firma)** și punând în comentariu în fișierul **beans11.xml** proprietatea prin care se injectează valorarea "BestCompany" în proprietatea firma. Datorită valorii lui *required* nu se generează nici o excepție. Dacă *required* are valoarea *true* se va genera excepție

- **Adnotăția @Qualifier**
- Pot fi situații când în fișierul de configurare XML se creează mai multe beanuri de același tip și se dorește legarea unui singur bean la o proprietate. În acest caz se utilizează adnotăția **@Qualifier** împreună cu adnotăția **@Autowired**, pentru a specifica care este beanul de legătură
- Se consideră fișierul **beans12.xml**, care conține două beanuri de tip **Job**

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
                           http://www.springframework.org/schema/context
                           http://www.springframework.org/schema/context/spring-context-3.0.xsd">

    <context:annotation-config/>

    <bean id="pers" class="exemplul12.Persoana">
        <property name="nume" value="dana"/>
    </bean>

    <bean id="job1" class="exemplul11.Job">
        <property name="firma" value="BestCompany"/>
        <property name="functia" value="sef"/>
    </bean>
    <bean id="job2" class="exemplul11.Job">
        <property name="firma" value="TheOtherBestCompany"/>
        <property name="functia" value="sef"/>
    </bean>
</beans>
```

- În fișierul **Persoana.java** se specifică cu ajutorul anotăției @Qualifier, amplasată deasupra variabilei membru corespunzătoare, care din cele două beanuri va fi injectată în variabila membru job

```
package exemplul12;

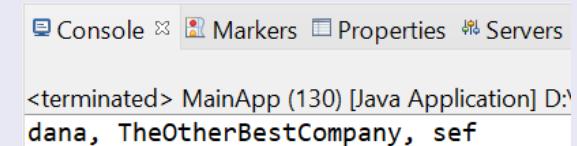
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import exemplul11.Job;

class Persoana {
    private String nume;
    @Autowired
    @Qualifier("job2")
    private Job job;

    public Persoana(){}
    public String getNume() { return nume; }
    public void setNume(String nume) { this.nume = nume; }

    public Job getJob() { return job; }
    public void setJob(Job job) { this.job = job; }

    public String toString() {
        return nume + ", " + job;
    }
}
```



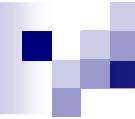
- Clasa Persoana reutilizează clasa exemplul11.Job care este importată, iar clasa **exemplul12.MainApp** este similară cu clasa **exemplul11.MainApp** cu mențiunea că preia metadatele de configurare din fișierul **beans12.xml**

### 7.3.12 Utilizarea adnotățiilor Spring JSR-250

- *JSR-250 (Java Specification Request)* este un standard care definește un set de adnotății care se pot utiliza împreună cu diferite tehnologii și diferite frameworkuri
- *JSR-250* are obiectivul de a defini un set de adnotății care pot fi utilizate de multe componente *Java EE (enterprise edition)*
- *@PostConstruct* – această adnotăție se aplică unei metode pentru a indica că aceasta trebuie apelată după ce injectarea dependentelor este completă
- *@PreDestroy* – aceasta adnotăție se aplică unei metode pentru a indica că acea metodă se va apela înainte ca obiectul *bean* să fie eliminat din containerul Spring (înainte ca acesta să fie distrus)
- *@Resource* – permite specificarea numelui beanului care va fi injectat, reprezentând o alternativă la utilizarea adnotățiilor *@Autowired* și *@Qualifier*
- Adnotăția *@Resource* marchează o resursă care este necesară aplicației. Când adnotăția este aplicată unui câmp sau unei metode, containerul va injecta o instanță a resursei cerute în componenta aplicației, atunci când componenta este inițializată

- Adnotația `@Resource` are atributul `name`, care reprezintă numele beanului care va fi injectat
- Adnotația `@PreDestroy` este utilizată de către metode de notificare, pentru a semnaliza că instanța este în proces de a fi eliminată de către container.
- Adnotăriile `JSR250` se găsesc în pachetul `javax.annotation.*`. Utilizarea acestui pachet necesită adăugarea dependenței de `javax.annotation-api` în fișierul `pom.xml`, salvarea acestui fișier și actualizarea proiectului `Maven`

```
<!-- https://mvnrepository.com/artifact/javax.annotation/javax.annotation-api -->
<dependency>
    <groupId>javax.annotation</groupId>
    <artifactId>javax.annotation-api</artifactId>
    <version>1.3.2</version>
</dependency>
```



## ■ Fișierul *Persoana.java*:

```
package exemplul13;

import javax.annotation.*;
import exemplul11.Job;

class Persoana {
    private String nume;

    @Resource(name="job1")
    private Job job;

    public Persoana() {}

    public String getNume() { return nume; }
    public void setNume(String nume) { this.nume = nume; }

    public Job getJob() { return job; }
    public void setJob(Job job) { this.job = job; }

    public String toString() { return nume + ", " + job; }

    @PostConstruct
    public void init(){
        System.out.println("init()");
    }
    @PreDestroy
    public void destroy(){
        System.out.println("destroy()");
    }
}
```

- Clasa *Persoana* este dependentă de clasa ***exemplul11.Job***, aceasta fiind importată din exemplul precedent
- Fișierul *beans13.xml* diferă de *beans12.xml* doar prin faptul că va injecta valori în obiectele clasei ***exemplul13.Persoana***
- Fișierul ***MainApp.java***:

```
package exemplul13;

import org.springframework.context.support.AbstractApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

class MainApp {
    public static void main(String []args) {
        AbstractApplicationContext context=new ClassPathXmlApplicationContext("beans13.xml");
        Persoana p=(Persoana)context.getBean("pers");
        System.out.println(p);
        context.registerShutdownHook();
        context.close();
    }
}
```

<terminated> MainApp (131) [Java Application] D:\  
init()  
dana, BestCompany, sef  
destroy()

### 7.3.13 Utilizarea adnotățiilor @Component și @Value

- Adnotăția @Component se poate pune deasupra unei clase pentru a marca că acea clasă corespunde unui bean.
- Amplasarea adnotăției @Component deasupra unei clase este echivalentă cu introducere unui bean fără constructor și fără proprietăți în fișierul de configurare XML
- Dacă se lucrează cu fișierul XML de configurare în acesta se pot introduce mai multe beanuri pentru o clasă, lucru care nu se poate realiza cu ajutorul adnotățiilor
- Spring framework trebuie să știe că avem beanuri în cadrul condului, pe care el trebuie să le găsească. Acest lucru se specifică adăugând în XML tagul de mai jos:

```
<context:component-scan base-package="denumire_pachet"/>
```

- Adnotăția @Value se poate utiliza pentru a injecta o valoare într-o variabilă membră

## ■ Fișierul **Job.java**

```
package exemplul14;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

@Component
public class Job {
    private String firma;
    private String functia;

    public String getFirma() {
        return firma;
    }
    @Value("TheBestCompany")
    public void setFirma(String firma) {
        this.firma = firma;
    }
    public String getFunctia() {
        return functia;
    }
    @Value("Boss")
    public void setFunctia(String functia) {
        this.functia = functia;
    }
    @Override
    public String toString() {
        return firma + ", " + functia;
    }
}
```



## ■ Fișierul **Persoana.java**

```
package exemplul14;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

@Component
class Persoana {
    @Value("John")
    private String nume;
    @Autowired
    private Job job;

    public Persoana() {}

    public String getNume() {
        return nume;
    }
    public void setNume(String nume) {
        this.nume = nume;
    }
    public Job getJob() {
        return job;
    }
    public void setJob(Job job) {
        this.job = job;
    }
    public String toString() {
        return nume + ", " + job;
    }
}
```

## ■ Fișierul **MainApp.java**

```
package exemplul14;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

class MainApp {
    public static void main(String []args){
        ApplicationContext context=new ClassPathXmlApplicationContext("beans14.xml");
        Persoana p=(Persoana)context.getBean("persoana");
        System.out.println(p);
    }
}
```

John, TheBestCompany, Boss

## ■ Fișierul **beans14.xml**

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
                           http://www.springframework.org/schema/context
                           http://www.springframework.org/schema/context/spring-context-3.0.xsd">

    <context:annotation-config/>
    <context:component-scan base-package="exemplul14"/>

</beans>
```

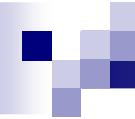
- Atât clasa Persoana cât și clasa Job au fost marcate cu adnotăția @Component,
- În clasa Job, injectarea dependențelor s-a realizat cu ajutorul setterelor și a adnotăției @Value
- În clasa Persoana, injectarea dependențelor s-a realizat cu ajutorul adnotăției @Value și a adnotăției @Autowired, ambele amplasate deasupra variabilelor membru corespunzătoare
- Adnotăția @Autowired a permis realizarea legăturii dintre variabila membru job și beanul job. În mod alternativ, s-ar fi putut utiliza adnotăția @Resource

## 7.4 Spring JDBC Framework

- Permite accesul la funcționalitățile *JDBC* din containerul *Spring*
- Oferă câteva simplificări în comparație cu clasicul *JDBC* în ceea ce privește managementul conexiunilor și tratarea excepțiilor
- În fișierul *pom.xml* trebuie adăugate următoarele dependențe:

```
<!-- https://mvnrepository.com/artifact/mysql/mysql-connector-java -->
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.19</version>
</dependency>

<!-- https://mvnrepository.com/artifact/org.springframework/spring-jdbc -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-jdbc</artifactId>
    <version>5.3.24</version>
</dependency>
```



## ■ Fișierul *Persoana.java*

```
package exemplul15;

class Persoana {
    private int id; //in BD se seteaza sa fie autoincrement
    private String nume;
    private int varsta;

    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getNume() {
        return nume;
    }
    public void setNume(String nume) {
        this.nume = nume;
    }
    public int getVarsta() {
        return varsta;
    }
    public void setVarsta(int varsta) {
        this.varsta = varsta;
    }
    @Override
    public String toString() {
        return id + ", " + nume + ", " + varsta;
    }
}
```

- Fișierul **PersoanaMapper.java**

```
package exemplul15;

import java.sql.ResultSet;
import java.sql.SQLException;
import org.springframework.jdbc.core.RowMapper;

class PersoanaMapper implements RowMapper<Persoana> {
    public Persoana mapRow(ResultSet rs, int rowNum) throws SQLException {
        Persoana persoana = new Persoana();
        persoana.setId(rs.getInt("id"));
        persoana.setNume(rs.getString("nume"));
        persoana.setVarsta(rs.getInt("varsta"));
        return persoana;
    }
}
```

- Datele preluate din baza de date se găsesc sub formă tabelară într-un obiect de tip *ResultSet*
- Parcurgerea înregistrărilor din *ResultSet* se realizează cu ajutorul cursorului. Acesta este poziționat inițial înaintea primei linii
- Metodele *first()*, *previous()*, *next()*, *last()*, *absolute()* permit deplasarea cursorului
- Metodele ***getInt()***, ***getString()*** permit obținerea valorilor câmpurilor specificate ca și parametru de pe rândul indicat de cursor
- Metoda ***mapRow()*** din interfața ***RowMapper***, mapează un rând din ***ResultSet*** indicat de parametrul ***rowNumber*** la un obiect de tip ***Persoana***

- Fișierul ***OperatiiBD.java***

```
package exemplul15;

import java.util.List;
import javax.sql.DataSource;
import org.springframework.jdbc.core.JdbcTemplate;

class OperatiiBD {
    private DataSource dataSource;
    private JdbcTemplate jdbcTemplateObject;

    public void setDataSource(DataSource dataSource) {
        this.dataSource = dataSource;
        this.jdbcTemplateObject = new JdbcTemplate(dataSource);
    }

    public void insert(String nume, int varsta) {
        String SQL = "insert into persoane (nume, varsta) values (?, ?)";
        jdbcTemplateObject.update( SQL, nume, varsta);
    }

    public void update(int id, int varsta){
        String SQL = "update persoane set varsta = ? where id = ?";
        jdbcTemplateObject.update(SQL, varsta, id);
    }

    public void delete(int id){
        String SQL = "delete from persoane where id = ?";
        jdbcTemplateObject.update(SQL, id);
    }
}
```

```

public Persoana getPersoana(int id) {
    String SQL = "select * from persoane where id = ?";
    Persoana pers = jdbcTemplateObject.queryForObject(SQL, new Object[]{id},
        new PersoanaMapper());
    return pers;
}
public List<Persoana> getListaPersoane() {
    String SQL = "select * from persoane";
    List <Persoana> pers = jdbcTemplateObject.query(SQL, new PersoanaMapper());
    return pers;
}
}

```

- Un obiect de tip ***DataSource*** permite obținerea unei conexiuni la BD. Parametrii conexiunii sunt injectați în parametrul `dataSource` cu ajutorul setterelor (vezi fișierul ***beans15.xml***). Aceștia sunt clasa driverului, url-ul de conectare la baza de date, utilizatorul și parola
- Clasa ***JdbcTemplate*** este clasa centrală din *Spring JDBC*. Ea simplifică utilizarea *JDBC*-ului și ajută la evitarea erorilor comune. Clasa execută interogările SQL sau update-urile, inițîind iteratîii asupra *ResultSet*-ului, prinzând excepîile *JDBC* și translatîndu-le în excepîi generice, mult mai informative
- Sursa de date este transmisă ca și parametru prin constructor, la instanîierea obiectului de tip ***JdbcTemplate***

- Metoda update din clasa JdbcTemplate, permite rularea unor comenzi SQL non-query
- Interogări SQL pot fi rulate cu ajutorul metodelor *query()* și *queryForObject()*. Metoda *queryForObject()* permite rularea unor interogări care produc un singur rezultat
- Maparea rezultatului obținut la un obiect de tip Persoana se face cu ajutorul clasei *PersoanaMapper*
- Fișierul ***beans15.xml***

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

    <bean id="operatiJDBC" class="exemplul15.OperatiBD">
        <property name="dataSource" ref="dataSource" />
    </bean>

    <bean id="dataSource"
          class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <property name="driverClassName" value="com.mysql.cj.jdbc.Driver" />
        <property name="url" value="jdbc:mysql://localhost:3306/TEST" />
        <property name="username" value="root" />
        <property name="password" value="root" />
    </bean>
</beans>
```

- Fișierul ***beans15.xml*** conține metadatele de configurare necesare pentru injectarea parametrilor necesari pentru realizarea unei conexiuni cu baza de date în obiectul *dataSource* cu ajutorul setterelor

- Fișierul **MainApp.java**

```
package exemplul15;

import java.util.List;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApp {
    public static void main(String[] args) {
        ApplicationContext context =new ClassPathXmlApplicationContext("Beans20.xml");

        OperatiibD operatiibD =(OperatiibD) context.getBean("operatiijDBC");

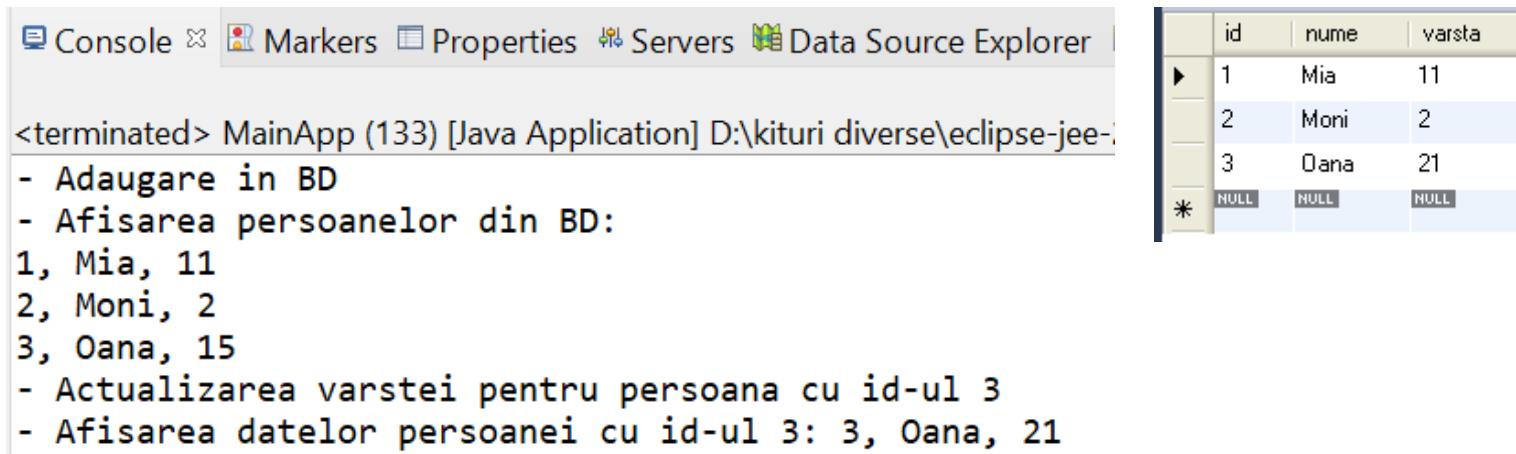
        System.out.println("- Adaugare in BD" );
        operatiibD.insert("Mia", 11);
        operatiibD.insert("Moni", 2);
        operatiibD.insert("Oana", 15);

        System.out.println("- Afisarea persoanelor din BD:" );
        List<Persoana> persoane = operatiibD.getListaPersoane();
        for (Persoana p : persoane) {
            System.out.println(p);
        }

        System.out.println("- Actualizarea varstei pentru persoana cu id-ul 3" );
        operatiibD.update(3, 21);

        System.out.print("- Afisarea datelor persoanei cu id-ul 3: " );
        Persoana p = operatiibD.getPersoana(3);
        System.out.println(p);
    }
}
```

- În programul principal se creează beanul *operatiiBD* și cu ajutorul lui se adaugă date în BD, se modifică aceste date și se afișează conținutul tabelei persoane



The screenshot shows the Eclipse IDE interface with the Data Source Explorer view open. A table titled 'persoane' is displayed with columns 'id', 'nume', and 'varsta'. The data is as follows:

	id	nume	varsta
▶	1	Mia	11
▶	2	Moni	2
▶	3	Oana	21
*	HULL	NULL	NULL

The Eclipse Console output window shows the following log entries:

```
<terminated> MainApp (133) [Java Application] D:\kituri diverse\eclipse-jee-eclipselink-2.7.2\plugins\org.eclipse.persistence_2.7.2.v20150604-1258\lib\jpa\org.eclipse.persistence.jpa.PersistenceProvider.jar  
- Adaugare in BD  
- Afisarea persoanelor din BD:  
1, Mia, 11  
2, Moni, 2  
3, Oana, 15  
- Actualizarea varstei pentru persoana cu id-ul 3  
- Afisarea datelor persoanei cu id-ul 3: 3, Oana, 21
```