

Design patterns

THE 23 GANG OF FOUR DESIGN PATTERNS

C	Abstract Factory	S	Facade	S	Proxy
S	Adapter	C	Factory Method	B	Observer
S	Bridge	S	Flyweight	C	Singleton
C	Builder	B	Interpreter	B	State
B	Chain of Responsibility	B	Iterator	B	Strategy
B	Command	B	Mediator	B	Template Method
S	Composite	B	Memento	B	Visitor
S	Decorator	C	Prototype		

Introducere

Design pattern-urile reprezintă soluții generale și reutilizabile ale unei probleme comune în design-ul software. Un design pattern este o descriere a soluției sau un template ce poate fi aplicat pentru rezolvarea problemei, nu o bucată de cod ce poate fi aplicată direct. În general pattern-urile orientate pe obiect arată relațiile și interacțiunile dintre clase sau obiecte, fără a specifica însă forma finală a claselor sau a obiectelor implicate.

Se consideră că există aproximativ **2000** de design patterns, iar principalul mod de a le clasifica este următorul:

Clasificare

- **Creational Patterns** - definesc mecanisme de creare a obiectelor
 - Singleton, Factory etc.
- **Structural Patterns** - definesc relații între entități
 - Decorator, Adapter, Facade, Composite, Proxy etc.
- **Behavioural Patterns** - definesc comunicarea între entități
 - Visitor, Observer, Command, Mediator, Strategy etc.

Important

Design pattern-urile nu trebuie privite drept niște rețete care pot fi aplicate direct pentru a rezolva o problemă din design-ul aplicației, pentru că de multe ori pot complica inutil arhitectura. Trebuie întâi înțeles dacă este cazul să fie aplicat un anumit pattern, și de-abia apoi adaptat pentru situația respectivă. Este foarte probabil chiar să folosiți un pattern (sau o abordare foarte similară acestuia) fără să vă dați seama sau să îl numiți explicit. Ce e important de reținut după studierea acestor pattern-uri este un mod de a aborda o problemă de design.

Creational Patterns

Crearea obiectelor este una din cele mai răspîndite sarcini a programatorului. Aceste tipuri de patternuri sunt destinate pentru crearea obiectelor, permitînd sistemului să rămînă independent atît de procesul de creare al obiectelor cît și de tipul obiectelor create.

EXEMPLU

Structural Patterns

Acest tip de patterne se bazează pe crearea sistemului în baza claselor și obiectelor. Aici pot fi utilizate 2 mecanisme:

- **Moștenire** – clasa de bază definește interfața iar clasele extinse realizarea. Aceste structuri vor fi statice
- **Compoziție** – obiectele diferitor clase sunt unite în structuri. Aceste structuri sunt dinamice și pot fi modificate în timpul execuției

Behavioural Patterns

Acest tip de pattern-uri gestionează legăturile dintre obiecte și repartizează responsabilitățile între ele.

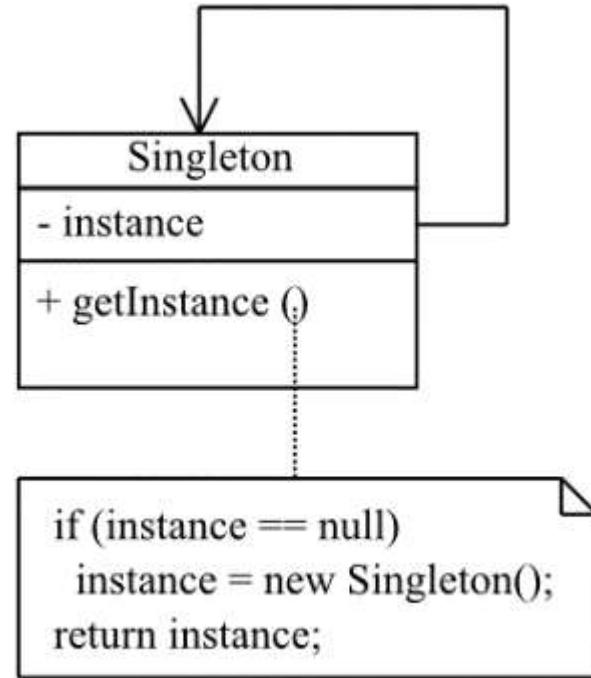
La realizarea acestui tip de pattern poate fi utilizată **moștenirea** sau **compoziția**.

Creational Patterns

Crearea obiectelor este una din cele mai răspîndite sarcini a programatorului. Aceste tipuri de patternuri sunt destinate pentru crearea obiectelor, permitînd sistemului să rămînă independent atît de procesul de creare al obiectelor cît și de tipul obiectelor create.

EXEMPLU

Singleton Pattern



Singleton Pattern

Pattern-ul Singleton este utilizat pentru a restricționa numărul de instanțieri ale unei clase la un singur obiect, deci reprezintă o metodă de a folosi o singură instanță a unui obiect în aplicație.

Utilizari ale Singleton Pattern

- a un subansamblu al altor pattern-uri:
 - împreună cu pattern-urile Abstract Factory, Builder, Prototype etc. De exemplu, în aplicație dorim un singur obiect factory pentru a crea obiecte de un anumit tip.
- obiectele care reprezintă stări
- în locul variabilelor globale. Singleton este preferat variabilelor globale deoarece, printre altele, nu poluează namespace-ul global cu variabile care nu sunt necesare.

Utilizari ale Singleton Pattern

este utilizat des în situații în care avem obiecte care trebuie accesate din mai multe locuri ale aplicației:

- obiecte de tip logger
- obiecte care reprezintă resurse partajate (conexiuni, sockets etc.)
- obiecte ce conțin configurații pentru aplicație
- pentru obiecte de tip *Factory*.

Utilizăm Singleton?

Din punct de vedere al design-ului și testării unei aplicații de multe ori se evită folosirea acestui pattern, în test-driven development fiind considerat un ***anti-pattern***.

A avea un obiect Singleton a carei referință o folosim peste tot prin aplicație introduce multe dependențe între clase și îngreunează testarea individuală a acestora.

În general, codul care folosește stări globale este mai dificil de testat pentru că implică o cuplare mai strânsă a claselor, și împiedică izolarea unei componente și testarea ei individuală. Dacă o clasă testată folosește un obiect singleton, atunci trebuie testat și singleton-ul. Soluția este simularea *mock-up* a singleton-ului în teste. Încă o problemă a acestei cuplări mai strânse apare atunci când două teste depind unul de celălalt prin modificarea singleton-ului, deci trebuie impusă o anumită ordine a rulării testelor.

Implementare Singleton

Aplicarea pattern-ului Singleton constă în implementarea unei metode ce permite crearea unei noi instanțe a clasei dacă aceasta nu există, și întoarcerea unei referințe către aceasta dacă există deja

Implementare Singleton

În cazul Singeton, clasa este instanțiată lazy(lazy instantiation). Astfel, memoria este utilizată doar în caz de necesitate, fiind apelată metoda *getInstance*

Crearea Singleton

```
1  <?
2  class Singleton
3  {
4      protected static $instance = null;
5
6      protected function __construct()
7      {
8          //Constructorul nu este necesar!
9      }
10
11     protected function __clone()
12     {
13         //Interzicem crearea clonelor
14     }
15
16     public static function getInstance()
17     {
18         if (!isset(self::$instance)) {
19             self::$instance = new Singleton();
20         }
21         return self::$instance;
22     }
23 }
```

Important

- Instanța instance este *private*
- Constructorul este privat ca să nu poată fi apelat decât din clasa respectivă
- Instanța este inițial nulă
- Instanța este creată la prima rulare a *getInstance()*

De ce Singleton și nu clase cu membri statici?

O clasă de tip Singleton poate fi extinsă, iar metodele ei suprascrise, însă într-o clasă cu metode statice acestea nu pot fi suprascrise (*overriden*)

Multiton Pattern

Singleton Pull

Multiton Pattern

Este asemănător cu patternul Singleton, fiind un registru de singleton-uri, care oferă posibilitatea creării unui număr determinat de exemplare.

Proprietățile Multiton Pattern

- poate fi utilizat cu un număr fixat de exemplare sau create la cerere
- dacă numărul este fixat, toate exemplarele pot fi initializate la inițializarea aplicației
- în cazul accesării unui exemplar inexistent fie este generată o eroare fie este generat un nou exemplar cu identificatorul respectiv
- Dezavantajul principal este că pot apărea multe părți ale aplicații dependente de el

Multiton Pattern

```
<?
class Database {
    private static $instances = array();

    private function __construct() {}

    public static function getInstance($key) {
        if (!array_key_exists($key, self::$instances)) {
            self::$instances[$key] = new self();
        }
        return self::$instances[$key];
    }

    private function __clone() {}
}

$master = Database::getInstance('master');
var_dump($master); // object(Database)#1 (0) {}

$logger = Database::getInstance('slave');
var_dump($logger); // object(Database)#2 (0) {}

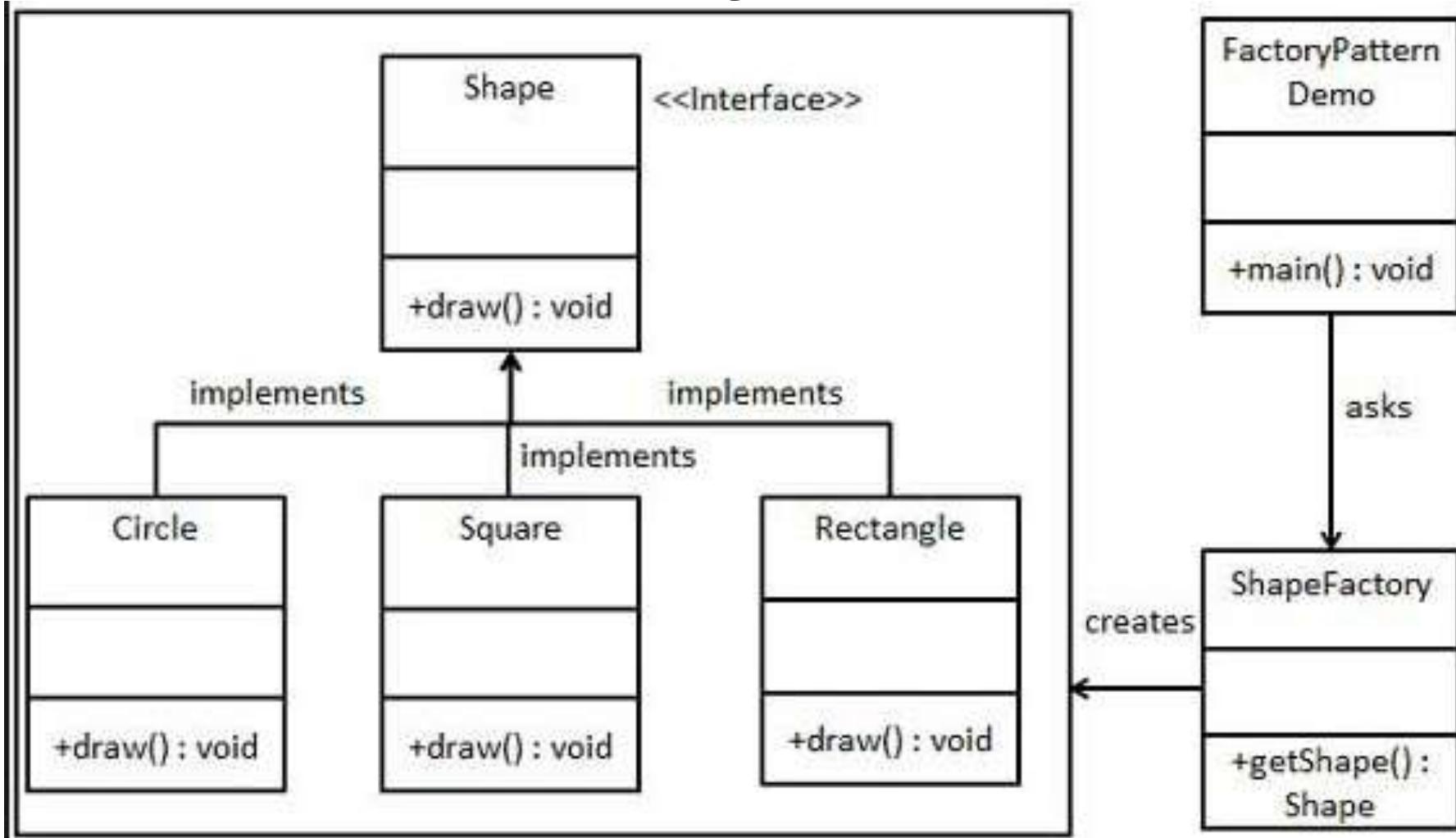
|
```

Object Pull Pattern

Object Pull Pattern

Este foarte asemănător cu patternul Multiton, doar că exemplarele create nu sunt multitonuri.

Factory Pattern



Factory Pattern

Patternurile de tip Factory sunt folosite pentru obiecte care generează instanțe de clase înrudite.

Factory Pattern

Interfețe

```
<?

interface Shape{
    public function draw();
}

class Circle implements Shape{
    public function draw(){
        {
            echo "This is a circle";
        }
    }
}

class Square implements Shape{
    public function draw(){
        {
            echo "This is a square";
        }
    }
}
```

Clase abstracte

```
abstract class Shape{
    abstract public function draw();
}

class Circle extends Shape{
    public function draw(){
        {
            echo "This is a circle";
        }
    }
}

class Square extends Shape{
    public function draw(){
        {
            echo "This is a square";
        }
    }
}
```

Factory Pattern

Clasa ShapeFactory

```
abstract class ShapeFactory
{
    public static function getShape($shapeName) {
        $className = ucfirst($shapeName);
        if(class_exists($className)) {
            return new $className;
        } else {
            return null;
        }
    }
}
```

Factory Pattern

Crearea obiectelor

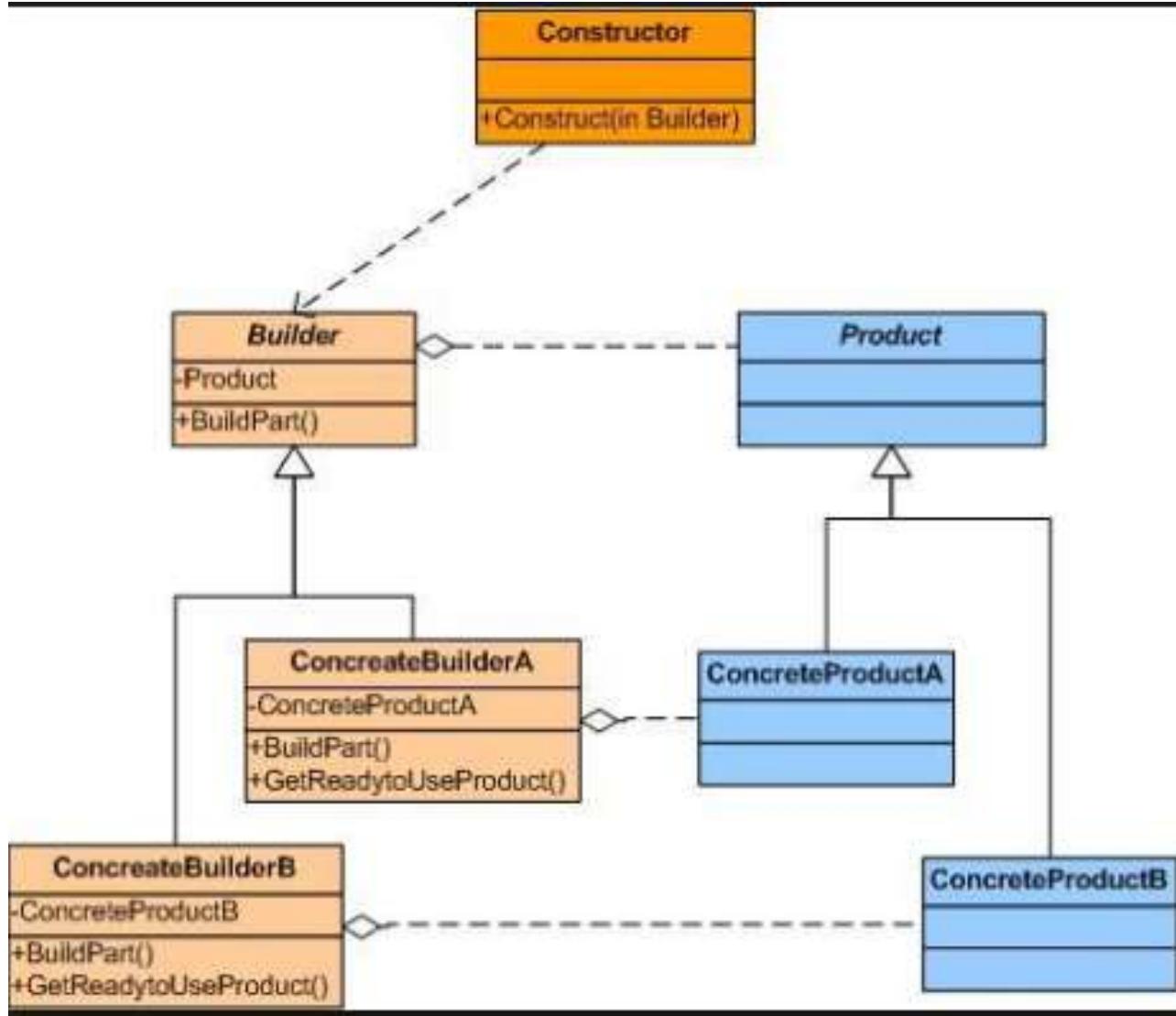
```
$circle = ShapeFactory::getShape("circle");
$circle->draw();

$circle = ShapeFactory::getShape("square");
$circle->draw();
```

Rezultatul va fi

This is a circleThis is a square

Builder Pattern

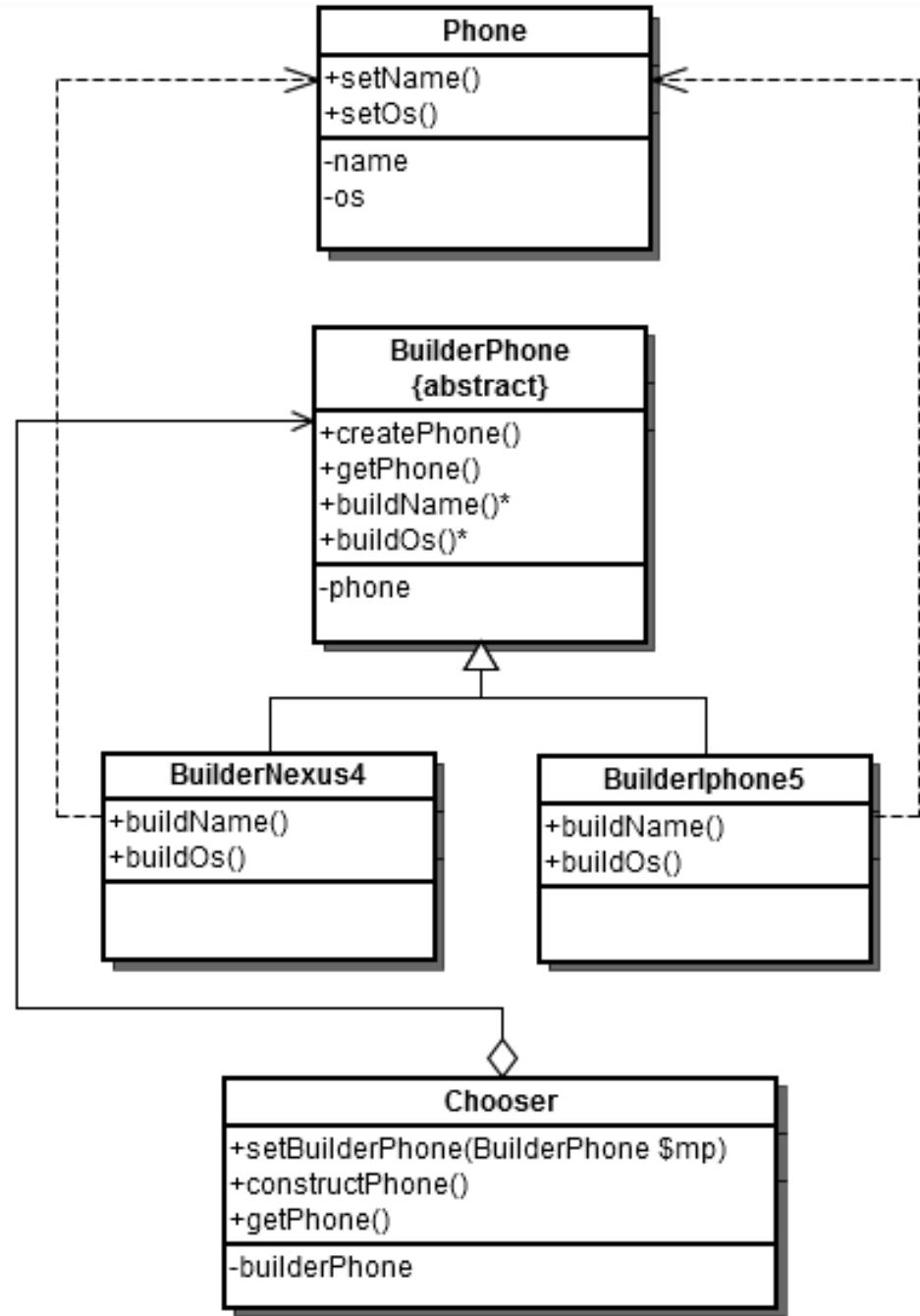


Builder Pattern

Este asemănător cu Factory, principala diferență fiind aceea că Builder conține în interior toate operațiile complexe pentru crearea obiectului. Programatorul îi dă comanda, doresc să construiesc un iPhone, iar constructorul(builder) execută toate operațiile pentru a construi un iPhone. Dacă este nevoie de un alt tip de telefon, este suficient să fie indicat tipul, iar constructorul va efectua toate operațiile necesare.

[Exemplu în fișierul Builder.php](#)

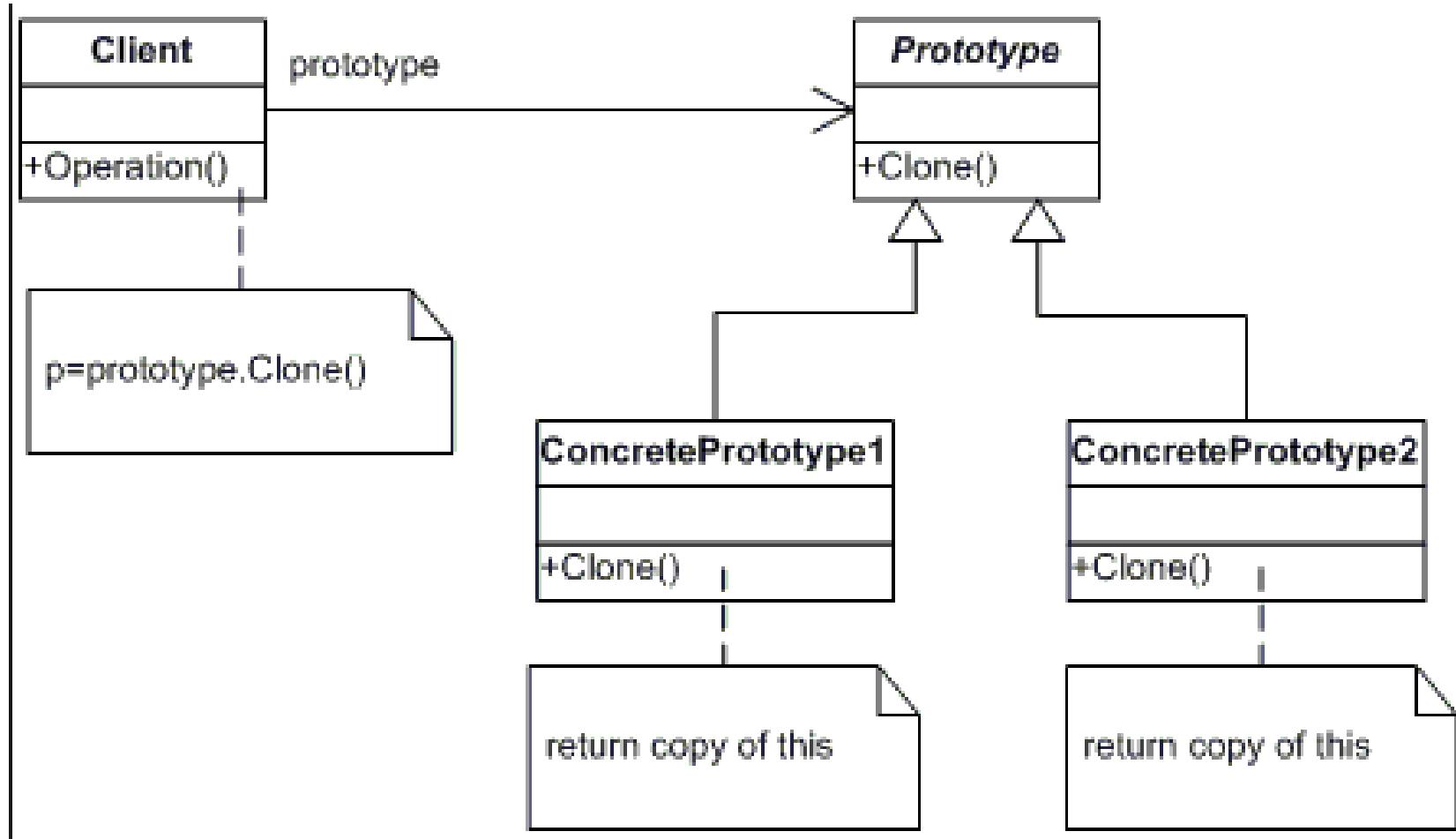
Builder Pattern



Diferența dintre Factory și Builder

- Factory – Telefonul este deja produs, iar programatorul îl utilizează
- Builder – telefonul este produs la cerere. El poate crea din mai multe obiecte simple(componente) un obiect complex(produsul final).

Prototype Pattern



Prototype Pattern

Permite crearea obiectelor noi în baza obiectelor deja existente. Nu este necesar de a cunoaște din ce este compus obiectul prototip.

[Exemplu în fișierul Prototype.php](#)

Structural Patterns

Acest tip de patterne se bazează pe crearea sistemului în baza claselor și obiectelor. Aici pot fi utilizate 2 mecanisme:

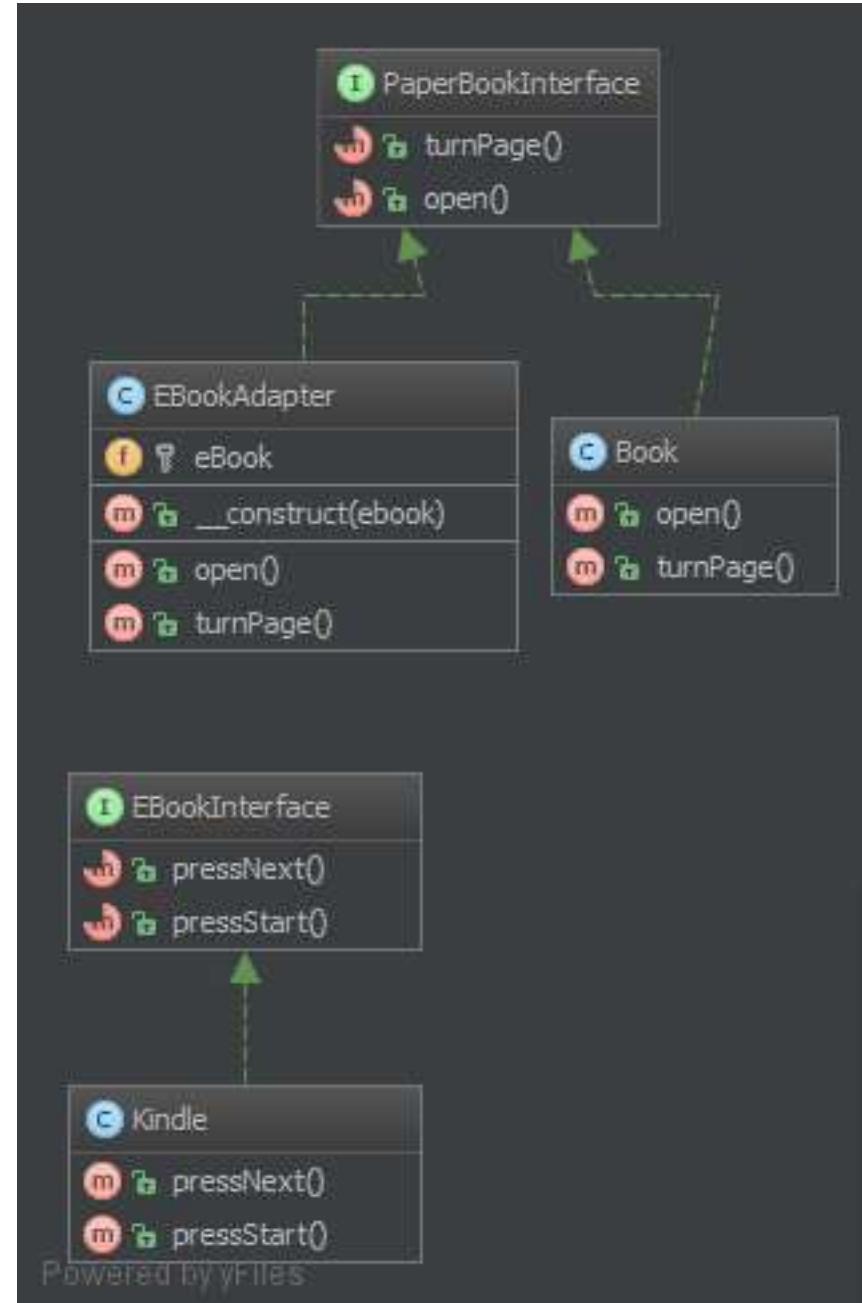
- **Moștenire** – clasa de bază definește interfața iar clasele extinse realizarea. Aceste structuri vor fi statice
- **Compoziție** – obiectele diferitor clase sunt unite în structuri. Aceste structuri sunt dinamice și pot fi modificate în timpul execuției

Adapter pattern

Scopul acestui pattern este de a transforma o interfață incompatibilă și incomodă în una compatibilă pentru aplicație.

El permite claselor să lucreze împreună, chiar dacă au interfețe diferite.

Adapter pattern

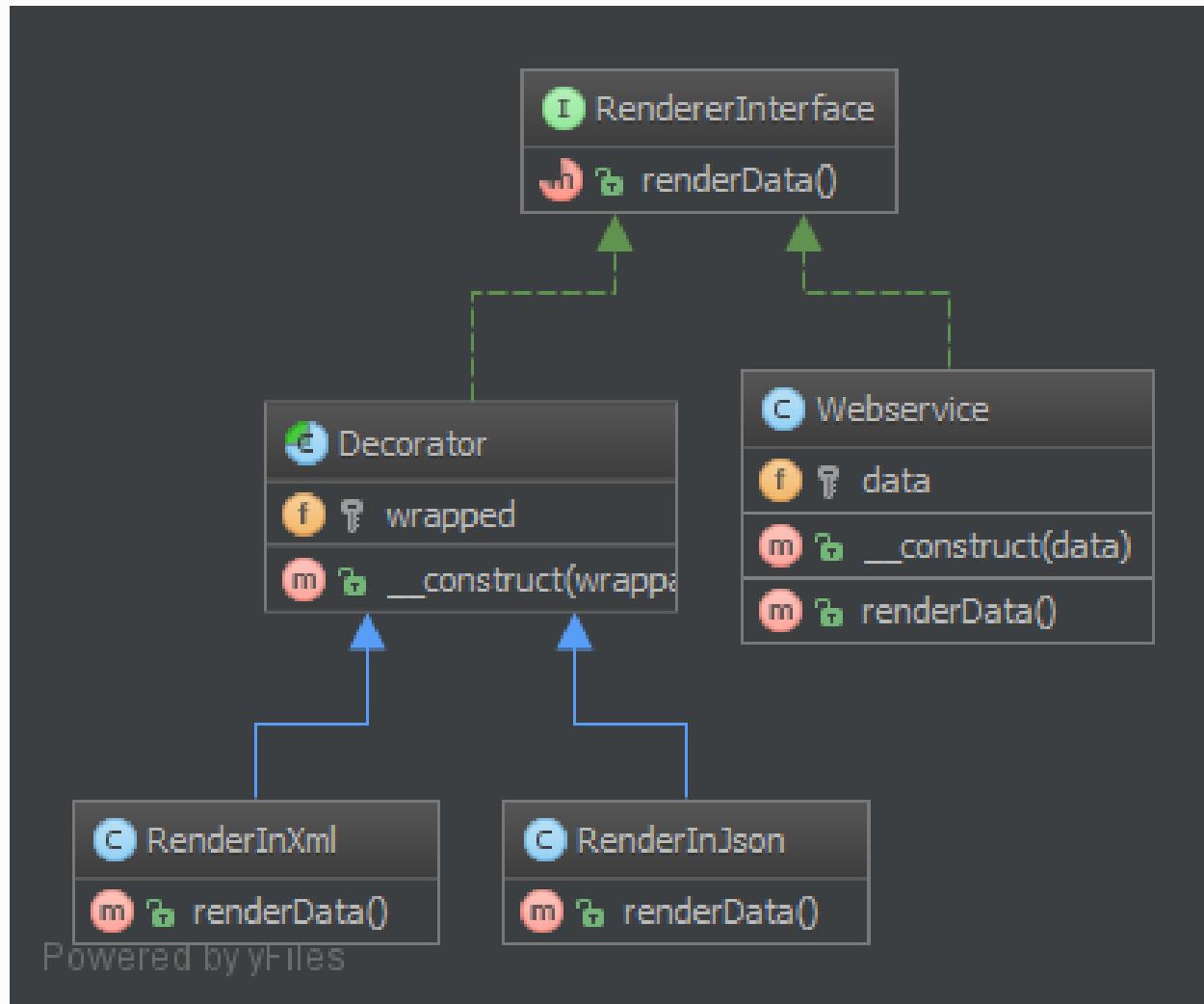


[Exemplu](#)

Decorator pattern

Scopul este de a adăuga dinamic noi posibilități clasei de bază.

Decorator pattern



[Exemplu](#)

Dependency injection

Principiul de lucru al acestui pattern constă în encapsularea unui obiect în alt obiect, astfel permîțînd utilizarea de către obiectul principal a funcționalităților obiectului encapsulat.

Dependency injection

Principiul de lucru al acestui pattern constă în encapsularea unui obiect în alt obiect, astfel permîțînd utilizarea de către obiectul principal a funcționalităților obiectului encapsulat.

Behavioural Patterns

Acest tip de pattern-uri gestionează legăturile dintre obiecte și repartizează responsabilitățile între ele.

La realizarea acestui tip de pattern poate fi utilizată **moștenirea** sau **compoziția**.

Высшее образование дает
возможность нам заработать на
хлеб, а самообразование - на икру
и на масло.

Programarea orientată pe obiect II

- **30 ore PRELEGERI (15 pr. – 2 h)**
- ?? ore LABORATOR ... Depinde de grupă
 - Cel puțin 3 note... Detalii la lecții de laborator!

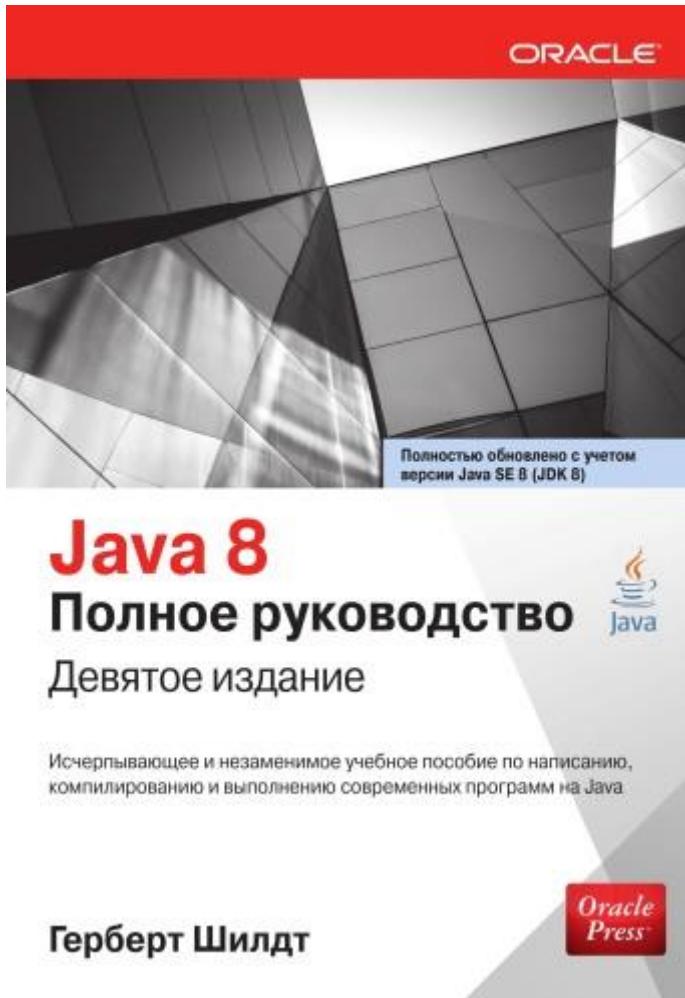
Cum primim notele?

- Notă la prelegerere:
 - $PR = (LC1 + LC2) / 2$
- Nota la laborator:
 - $LLab = (LP1 + LP2 + LP3) / 3$
- Nota la evaluarea curentă:
 - $N_{ev_cur} = (PR + LLab) / 2, PR \geq 5, LLab \geq 5$
- Nota finală:
 - $N_{finală} = 0,6 \times N_{ev_cur} + 0,4 \times N_{examen}$

Principiile de lucru în cadrul disciplinei

- Nu este salutată întârzierea la ore.
- Este salutată poziția activă a studentului.
 - care studiază din propria inițiativă noi conținuturi
 - formulează întrebări în cadrul prelegerilor și a orelor practice.
- Prezentarea unor soluții a sarcinilor, preluate de la colegi va fi considerată plagiat și va fi sancționată prin note de „1”.

Surse bibliografice



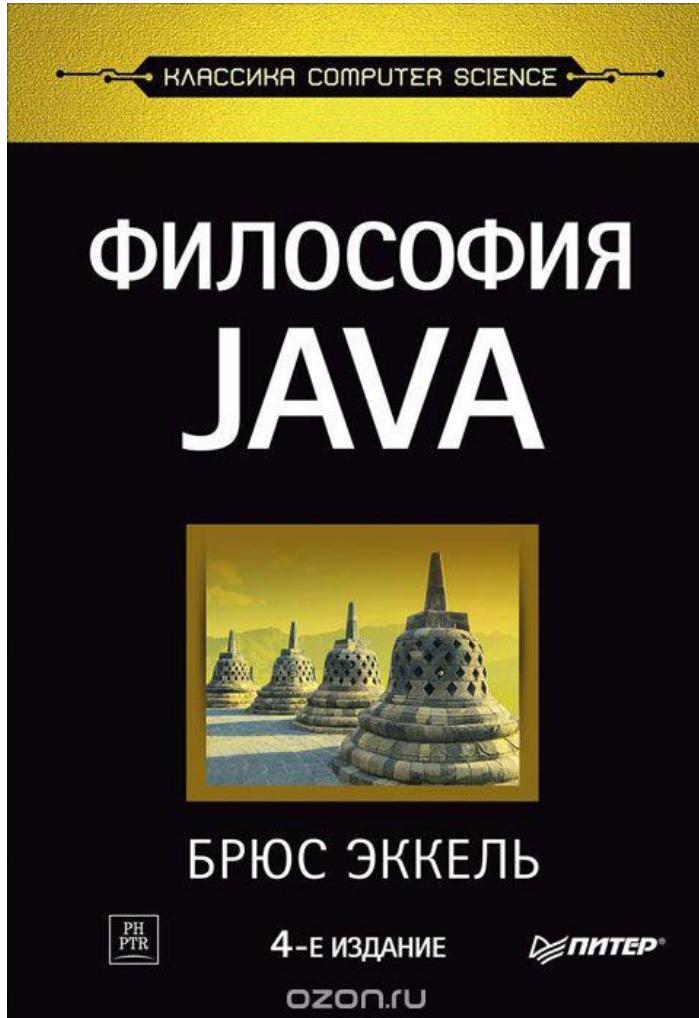
**Java 8. Полное
руководство
9-е издание**
Герберт Шилдт
Oracle Press
2015

Surse bibliografice



**Java 8. Полное
руководство
10-е издание**
Герберт Шилдт
Oracle Press
2018

Surse bibliografice



Философия Java
Thinking in Java (4th
Edition)
Питер
2016

Resursele informaționale la unitatea de curs

- **Cărți – în română, rusă și engleză**
- **Surse internet**
- **Surse video**
- **Prezentările de la prelegeri**

Despre mine



- 2006 – licență specialitatea *Informatica și limba engleză aplicată*
- 2012 – doctorat specialitatea *Bazele teoretice ale informaticii; programarea calculatoarelor*
- 2014 – mobilitate postdoctorală domeniul *Procesare a limbajului natural*
- 2014 – proiectul *GeDeRO*
- 2015 – proiectul *SoftCrates*
- 2018 – proiectul Conferința *MITI2018*
- 2019 – Institutul de Matematică și Informatică „Vladimir Andrunachievici”
- 2019 – ACETI...

Programarea orientată pe obiect II

Prelegerea nr. 1

Tehnologii Java: Java 8/9/10...13,
prezentare. Interfețe Java.

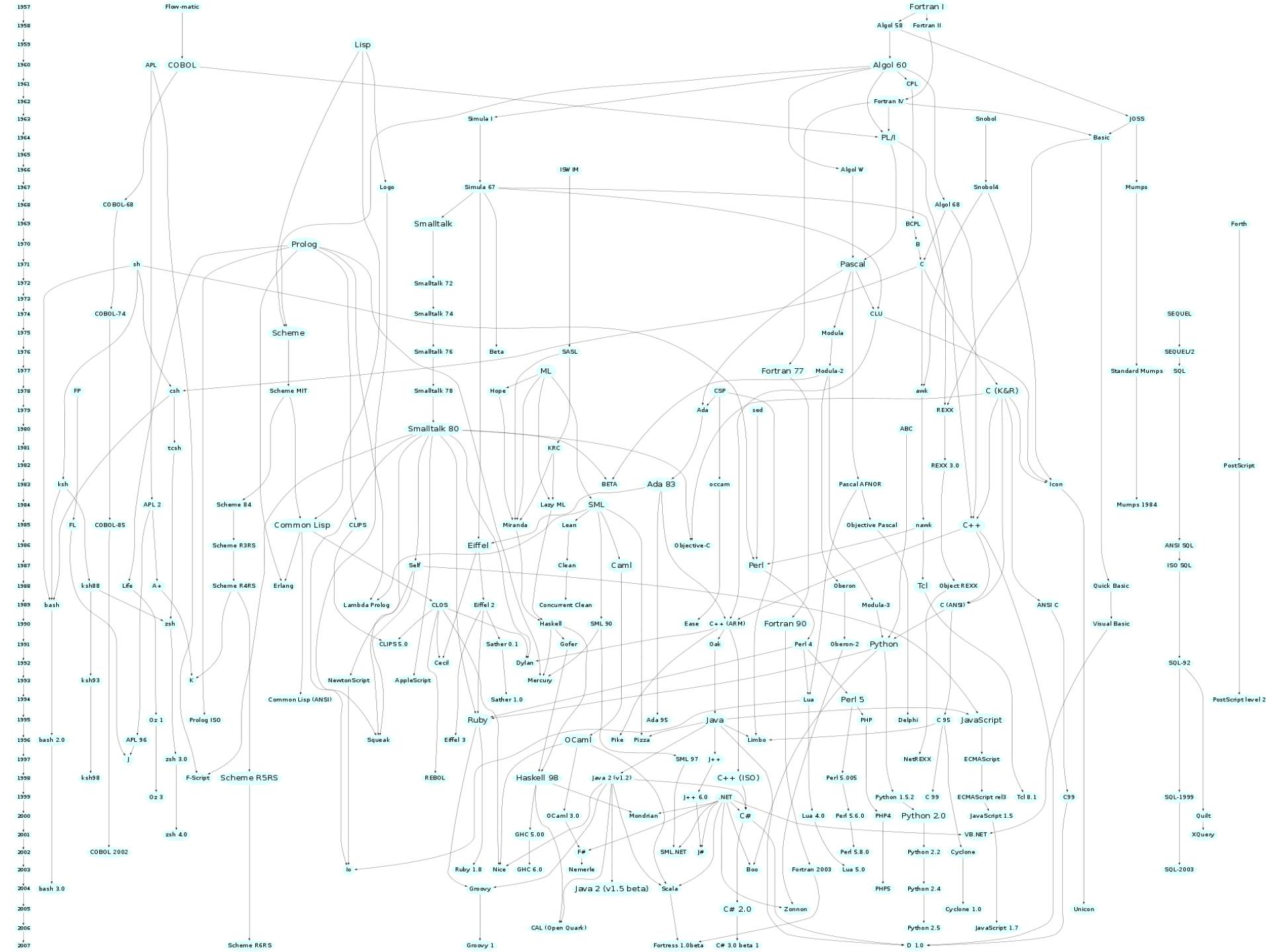
Petic Mircea

petic.mircea@yahoo.com

www.facebook.com/mircea.petic

Limbajele folosite deja cunoscute





Motivele apariției limbajelor noi de programare

- *Adaptarea la medii și domeniului de aplicare a limbajului;*
- *Realizarea îmbunătățirii și perfecționării domeniului programării.*

**Pentru a învăța un limbaj de programare
în profunzime este necesar să înțelegem:**

- *Motivul apariției; factorii care au stat la baza apariției; particularitățile moștenite de la alte limbaje.*

Originea limbajului de programare Java

- **Limbajul C** – un limbaj scris de programatori pentru programatori! (înc. '70)
 - *Limbaj structurat!*
- **Limbajul C++** - un limbaj care extinde limbajul C adăugînd lucru cu clase! (anul 1979)
 - Prima denumire a sa este “*C with Classes*”
- **Dezvoltarea Internetului!**

Apariția limbajului Java

- În 1991 – în compania Sun Microsystems - **Oak**
 - Mai mulți programatori au dorit să elaboreze un limbaj pentru a programa dispozitive electronice independent de platformă!
- În 1993 – se încearcă să fie posibil de a elabora aplicații pentru Internet.
- În 1995 limbajul Oak se transformă în Java.
- **Popularitatea lui demonstrează apariția LP C#**

Tipuri de aplicații Java (MS-DOS)

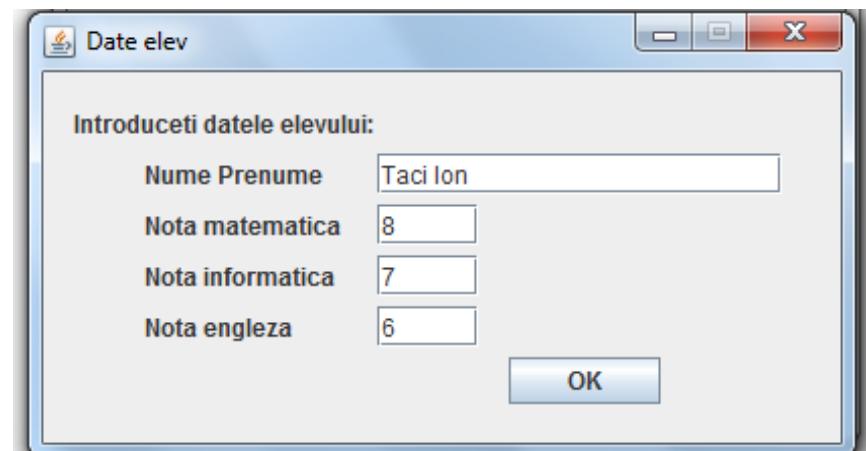
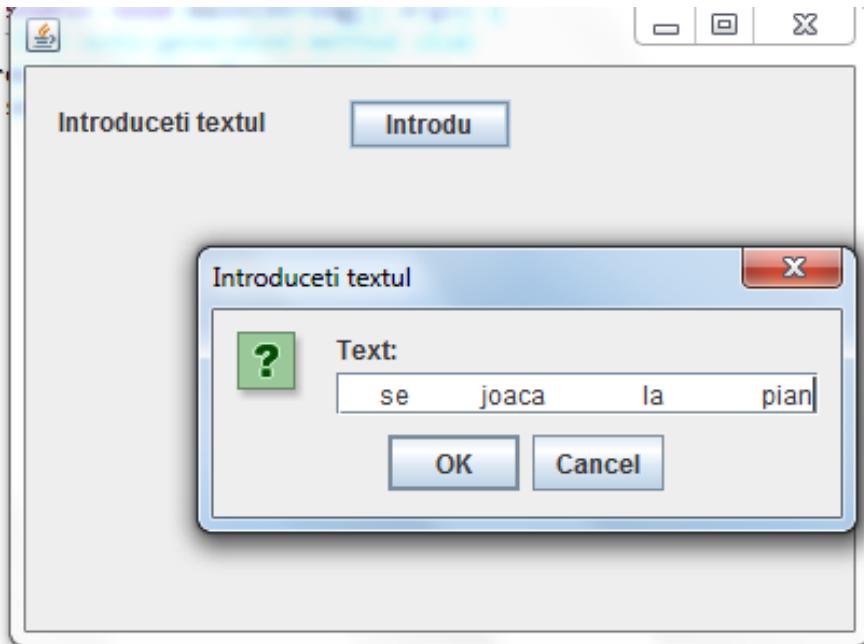
The image shows a Windows operating system interface with two windows open. The top window is a Notepad titled "test.java - Notepad" containing the following Java code:

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

The bottom window is a Command Prompt titled "Command Prompt" with the following text displayed:

```
D:\>javac test.java
D:\>java test
Start Java!
D:\>
```

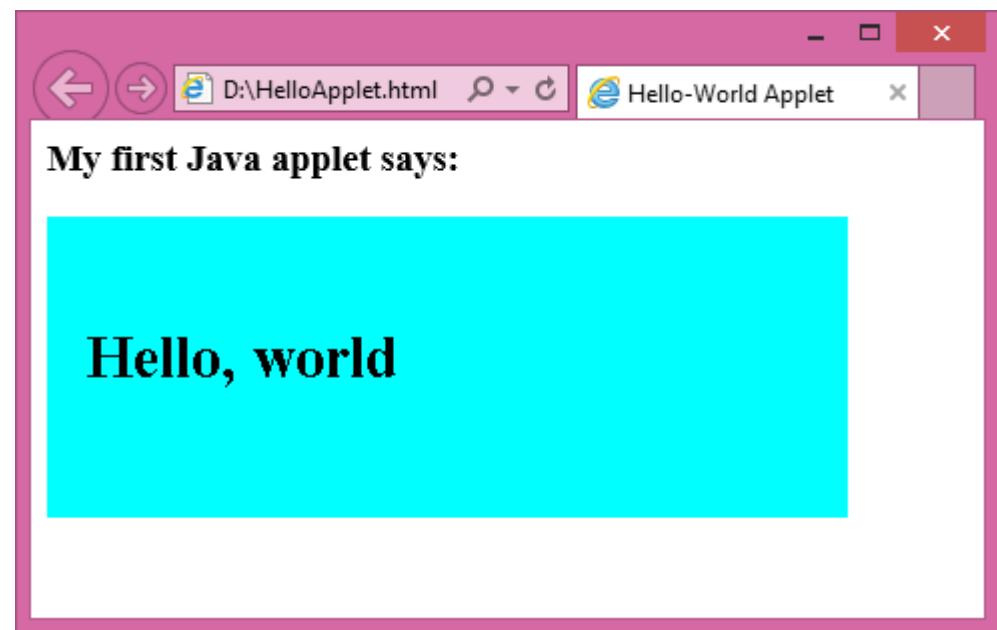
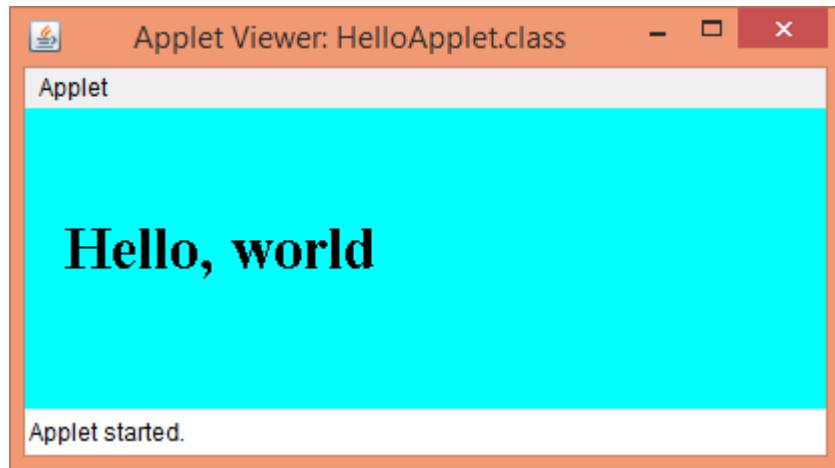
Tipuri de aplicații Java (Desktop-GUI)



The image shows a Java desktop application window titled "Adauga elev". It features a table with five columns: "Nume Prenume", "Mat", "Inf", "Eng", and "media". The table contains five rows of data:

Nume Prenume	Mat	Inf	Eng	media
Petrache Vasile	8	7	4	6,33
Gavrilieuc Ion	9	6	7	7,33
Bostan Aurica	8	5	6	6,33
Colibaba Vasile	9	7	8	8,00

Tipuri de aplicații Java (Applets)



Cum a schimbat LP Java Internetul?

- **Applet-ul** – un tip special de aplicație Java destinat transmiterii prin Internet și îndeplinirea automată într-un browser compatibil cu Java.
- 1 iulie 1997 – a fost declarat că cercetătorii NASA cu ajutorul appleturilor Java controlează robotul care studiază paneta Marte.
- Caracteristicile applet-ului
 - *Securitatea* – acțiunea appletului nu se referă la accesul la alte părți ale sistemelor de operare.
 - *Transferabilitatea* - ... Byte-codul

Byte-codul

- Compilatorul Java ca rezultat nu crează un fișier executabil, dar aşa numitul byte-cod.
- Byte-cod-ul este un grup de instrucțiuni optimizate, care sănt necesare pentru a fi executate în mediu de execuție a sistemului Java, numit **Java Virtual Machine**.
- **Mecanismul asigură executarea pe mai multe platforme la fel!**

Java Virtual Machine

- JVM – un program executat pe un anumit sistem de operare și care oferă aplicațiilor Java toate posibilitățile necesare.
- Codul sursă a programului Java este un fișier text obișnuit, care are în denumire extensia .java.
- Fișierul se transmite compilatorului, care transformă într-un format special Java-byte code.
- Rezultatul compilării este fișierul cu extensia .class
- Aplicația Java constând din aceste fișiere este transmisă JVM pentru execuție.

Tipuri de aplicații Java (JSP)

The screenshot shows a web browser window with the URL `localhost:8080/laborator4/index.jsp`. The page title is "Lista Angajatilor". Below the title is a table listing 22 employees. The table has columns: Nr., Nume prenume, Data nașterii, functia, and two buttons (delete, edit). At the bottom of the table are buttons for Import data and Adauga angajat. Below the table is a file input field with placeholder text "Fisier Alegeți fisierul Nu ati ales niciun fisier Încarcă Resetări".

Nr.	Nume prenume	Data nașterii	functia	
2	Petrenco Ivan	17.02.1996	tractorist	delete edit
3	Burlacu Virginia	11.12.1985	designer	delete edit
5	Bostan Marina	14.02.1992	designer	delete edit
7	Zasadnii Dumitru	28.06.1993	inginer	delete edit
8	Teleba Ana	14.02.1990	economist	delete edit
9	Grosu Mihail	14.02.1990	laborant	delete edit
10	Anton Ionescu	10.10.2000	furnizor	delete edit
11	Bunescu Mariana	12.05.1998	operator	delete edit
12	Bostan Marin	01.05.1994	liberal	delete edit
13	Tirbu Mariana	06.02.1982	inginer	delete edit
14	German Alexandra	09.08.1996	vinzator	delete edit
15	Patrache Marin	05.08.1992	aviator	delete edit
16	Grumeza Aliona	02.12.1998	sora medicala	delete edit
17	Guzun Marina	15.02.1991	Educatoare	delete edit
18	Cojoc Marin	12.08.1991	profesor	delete edit
19	Bursuc Ana	12.09.1997	secretar	delete edit
20	Caldare Vanea	12.11.1994	inginer	delete edit
21	Bufer Ion	15.08.1991	disigner	delete edit
22	Gandrabura Kiril	12.02.1981	prim Ministru	delete edit

Servlet-urile & Paginile JSP

- Servletul este un mic program, care se execută pe partea de server.
- Servlet-ele la fel se compilează în byte-code și se execută în mașina virtuală.
- Unul și același servlet poate fi îndeplinit pe mai multe medii de server.
- Condiția este ca pe server să fie instalat JVM și containere pentru servlete.

Caracteristicile deosebite ale limbajului Java

- *Simplitate*
- *Securizare*
- *Transferabilitate*
- *Orientare pe obiect*
- *Siguranță*
- *Multithreading*
- *Independență de arhitectură*
- *Interpretare*
- *Productivitate ridicată*
- *Distribuire*
- *Dinamicitate*

Java Development Kit (JDK)

- 23 mai 1995 – prima versiune!

JDK este instrument de dezvoltare care conține utilitare de bază, bibliotecile standarde de clase și exemple demonstrative.

- JDK nu conține nici un mijloc de elaborare a aplicațiilor. Nu are nici un redactor textual și operează doar cu fișierele java.
- Compilatorul java este **javac (java compiler)**.
- Mașina virtuală este reprezentat prin **java**.
- Pentru vizualizarea appleturilor este **appletviewer**.

Primele 8 biblioteci standard

- *java.lang* – clasele de bază, necesare pentru orice aplicație
- *java.util* – multe clase suplimentare
- *java.applet* – clase pentru crearea appleturilor
- *java.awt, java.awt.peer* – biblioteca pentru crearea interfeței grafice utilizator
- *java.awt.image* – clasă suplimentară pentru lucru cu imagini
- *java.io* – lucru cu fluxuri de date (streams) și fișiere
- *java.net* – lucru cu rețea

Java 2

- 15 iunie 1999
- Java 2 Platform, Standard Edition (J2SE) – pentru aplicații de birou, ca și în JDK standard.
- Java 2 Platform, Enterprise Edition (J2EE) – pentru elaborarea aplicațiilor complexe, sigure și pe servere distribuite.
- Java 2 Platform, Micro Edition (J2ME) – reprezintă versiune redusă a JDK, suficientă pentru dispozitivele mici și portabile.



- Transferarea a avut loc
 - APRILIE 2009 – IANUARIE 2010
- Prima nouă versiune Oracle a fost Java SE 7.
 - возможность управления инструкцией `switch` с помощью объектов класса `String`;
 - двоичные целочисленные литералы;
 - символы подчеркивания в числовых литералах;
 - расширенная инструкция `try`, называемая инструкцией `try с ресурсами` и поддерживающая автоматическое управление ресурсами;
 - выведение типов (через ромбовидный оператор) при создании обобщенного экземпляра объекта;
 - усовершенствованная обработка исключений, благодаря которой несколько исключений могут быть перехвачены в одном (групповом) блоке `catch`, а также улучшенный контроль типов для повторно генерируемых исключений.

Java SE 8

- Expresiile Lamda și tot ce ține de asta... ->
- Susținerea JavaFX

Java SE 9

- Apariția de **module**. Pachetele sunt organizate în module.
- Interpretatorul JShell.

Java SE 10 primăvara 2018

- Declararea variabilelor cu ajutorul cuvântului **var**.
- Apariția unui singur repozitoriu pentru JDK.

Java Platform, Standard Edition Documentation

Java Platform, Standard Edition (Java SE) helps you develop and deploy Java applications on desktops and servers. Java offers the rich user interface, performance, versatility, portability, and security that today's applications require.

Latest Release

[JDK 13](#)

Previous Releases

[JDK 12](#)

[JDK 11](#)

[JDK 10](#)

[JDK 9](#)

[JDK 8](#)

[JDK 7](#)

Programarea obiect orientată

Practic toate programele Java sănt
obiect orientate!

Două metodologii de programare:

- ***Codul care acționează asupra datelor*** (Model, orientat pe procese) – program ca consecutivitate de pași liniari (de cod).
- ***Datele, care gestionează cu accesul la cod*** (Programare obiect orientată) – organizarea programului în jurul datelor (adică obiectelor) și colecție de interfețe cu aceste date.

Abstractizare

- Operație a gândirii prin care se desprind și se rețin unele dintre caracteristicile și relațiile esențiale ale obiectului cercetării.
- Un mod eficient de aplicare a abstractizării reprezintă clasificarea ierarhică.
- Grație abstractizării datele programului obișnuit orientat pe procese, poate fi transformat în obiecte constitutive, și etapele procesului în mulțimea de mesaje cu care comunică obiectele.
- Fiecare obiect descrie comportamentul său.
- Obiectele pot fi considerate noțiuni esențiale, care reacționează la mesaje, care prescriu îndeplinirea anumitei acțiuni.

Principiile programării obiect orientate

- **Încapsulare** – mecanismul, care leagă codul și datele, cu care se manipulează, protejînd ambele aceste componente de la amestecul extern.
- **Moștenire** – proces, în rezultatul căruia un obiect primește proprietățile altuia.
- **Polimorfism** – principiul POO, care permite folosirea unuia și aceleleași interfețe pentru mai multe tipuri de acțiuni.

Clasă & Obiect

- Clasa definește structura și comportamentul (datele și codul), care va fi folosit cu colecția de obiecte.
- Clasa reprezintă o descriere a unui grup de obiecte.
- Clasa este o construcție logică.
- Obiectul este o construcție fizică.
- *În programele bine scrise în Java, metodele definesc modul în care vor fi folosite variabilele clasei. - Încapsulare*

Exemplu de obiecte și clasă



Exemplu de obiecte și clasă



Primul exemplu de program simplu

```
/*
 este un program simplu
*/
class Example {
// programul începe cu apelul metodei main()
    public static void main (String args[]) {
        System.out.println("Program simplu Java");
    }
}
```

Culegerea codului programului

- Fișierul care va conține codul programului Java va avea extensia **.java**.
- Denumirea fișierului care conține programul cules în slidul precedent va fi **Example.java**.
- Codul programului Java este un fișier textual.
- În Java tot codul trebuie să fie conținut într-o clasă.
- Numele clasei principale, trebuie să coincidă cu numele fișierului care conține codul programului.
- În Java se deosebesc literele mari de litere mici.

Compilarea programului

- Pentru a compila programul Example, trebuie lansat compilatorul (**javac**), și de indicat numele fișierului sursă în linia de comandă:
C:\>javac Example.java
- Compilatorul javac va crea fișierul Example.class, care conține byte-codul pentru JVM.
- Pentru a lansa programul în execuție trebuie de utilizat încăr cătorul aplicațiilor Java (java):
C:\>java Example
- În rezultat va apărea rezultatul execuției:
Program simplu Java

Al doilea exemplu de program

```
/*
 Încă un program java
*/
class Example2 {
    public static void main(String args[]) {
        int num;
        num=100;
        System.out.println("Variabila num:" + num);
        num=num*2;
        System.out.print("Valoarea variabile num*2 este egal cu ");
        System.out.println(num);
    }
}
```

```
/*
```

Демонстрация использования переменных.

Присвойте файлу с исходным кодом имя Example2.java.

```
*/
```

```
class Example2 {
```

```
    public static void main(String args[]) {
```

```
        int var1; // объявляется переменная
```

```
        int var2; // объявляется еще одна переменная
```

```
        var1 = 1024; // переменной var1 присваивается значение 1024
```

```
        System.out.println("Переменная var1 содержит " + var1);
```

```
        var2 = var1 / 2;
```

```
        System.out.print("Переменная var2 содержит var1 / 2: ");
```

```
        System.out.println(var2);
```

```
}
```

```
}
```

Объявление переменных

Присваивание
значения
переменной

Operatorul condițional **if**

- Sintaxa acestui operator este similară operatorilor if în limbajele de programare C, C++ și C#.

`if (condiție) operator`

Exemplu:

`if (num < 100)`

`System.out.println("num < 100");`

Operatorul repetitiv **for**

- Formatul operatorului

for(initializare; condiție; iterare)
operator

- Exemplu:

```
for(int x=0;x<10;x=x+1)
```

```
System.out.println("Valoarea x: "+x);
```

```
/*
 * Демонстрация применения цикла for.
 *
 Присвойте файлу с исходным кодом имя ForDemo.java.
 */
class ForDemo {
    public static void main(String args[]) {
        int count;

        for(count = 0; count < 5; count = count + 1) ←————— Этот цикл выполняет
            System.out.println("Значение счетчика: " + count);

        System.out.println("Готово!");
    }
}
```

Utilizarea blocurilor de cod

- Două și mai multe instrucțiuni pot fi grupate în blocuri de cod.
- Blocurile de cod se iau în paranteze figurate {
 - Instrucțiunea 1;
 - Instrucțiunea 2;
 - ...}
- Deseori blocurile de cod se utilizează cu instrucțiunile compuse if și for.

```
/*
```

Демонстрация применения блоков кода.

Присвойте файлу с исходным кодом имя BlockDemo.java

```
*/
```

```
class BlockDemo {
```

```
    public static void main(String args[]) {
```

```
        double i, j, d;
```

```
        i = 5;
```

```
        j = 10;
```

```
        // Телом этой инструкции if является целый блок кода
```

```
        if(i != 0) {
```

```
            System.out.println("i не равно нулю");
```

```
            d = j / i;
```

```
            System.out.print("j / i равно " + d);
```

```
}
```

```
}
```



Телом оператора if
является весь блок

Programarea orientată pe obiect II

Prelegerea nr. 2

Elemente de bază ale limbajului Java.

Petic Mircea

petic.mircea@yahoo.com

www.facebook.com/mircea.petic

Таблица 1.1. Ключевые слова Java

abstract	assert	boolean	break	byte	case
catch	char	class	const	continue	default
do	double	else	enum	exports	extends
final	finally	float	for	goto	if
implements	import	instanceof	int	interface	long
module	native	new	open	opens	package
private	protected	provides	public	requires	return
short	static	strictfp	super	switch	synchronized
this	throw	throws	to	transient	transitive
try	uses	void	volatile	while	which

Таблица 2.1. Встроенные примитивные типы данных Java

Тип	Описание
boolean	Представляет логические значения true и false
byte	8-разрядное целое число
char	Символ
double	Числовое значение с плавающей точкой двойной точности
float	Числовое значение с плавающей точкой одинарной точности
int	Целое число
long	Длинное целое число
short	Короткое число

Тип	Разрядность, бит	Диапазон допустимых значений
byte	8	от -128 до 127
short	16	от -32768 до 32767

Тип	Разрядность, бит	Диапазон допустимых значений
int	32	от -2147483648 до 2147483647
long	64	от -9223372036854775808 до 9223372036854775807

```
/*
    Расчет числа кубических дюймов в кубе объемом в 1 куб. милю
*/
class Inches {
    public static void main(String args[]) {
        long ci;
        long im;

        im = 5280 * 12;

        ci = im * im * im;

        System.out.println("В одной кубической миле содержится " +
                           ci + " кубических дюймов");
    }
}
```

```
/*
    Определение длины гипотенузы исходя из длины катетов,
    по теореме Пифагора
*/
class Hypot {
    public static void main(String args[]) {
        double x, y, z;

        x = 3;
        y = 4;
        z = Math.sqrt(x*x + y*y);
        System.out.println("Длина гипотенузы: " +z);
    }
}
```

Обратите внимание на вызов метода sqrt().
Перед именем метода указывается имя
класса, членом которого он является.

```
// С символьными переменными можно обращаться
// как с целочисленными
class CharArithDemo {
    public static void main(String args[]) {
        char ch;

        ch = 'X';
        System.out.println("ch содержит " + ch);

        ch++; // инкрементировать переменную ch ← Переменную типа char
        System.out.println("теперь ch содержит " + ch); ← можно инкрементировать

        ch = 90; // присвоить переменной ch значение 'Z' ← Переменной
        System.out.println("теперь ch содержит " + ch); ← типа char
    } ← можно присвоить
}
```

ch содержит X

теперь ch содержит Y

теперь ch содержит Z

```
// Демонстрация использования логических значений
class BoolDemo {
    public static void main(String args[]) {
        boolean b;

        b = false;
        System.out.println("Значение b: " + b);
        b = true;
        System.out.println("Значение b: " + b);

        // Логическое значение можно использовать для
        // управления условной инструкцией if
        if(b) System.out.println("Эта инструкция выполняется");

        b = false;
        if(b) System.out.println("Эта инструкция не выполняется");

        // В результате сравнения получается логическое значение
        System.out.println("Результат сравнения 10 > 9: " + (10 > 9));
    }
}
```

Результат выполнения данной программы будет таким.

```
Значение b: false
Значение b: true
Эта инструкция выполняется
Результат сравнения 10 > 9: true
```

Таблица 2.2. Управляющие последовательности символов

Управляющая последовательность	Описание
\'	Одинарная кавычка
\"	Двойная кавычка
\\"	Обратная косая черта
\r	Возврат каретки
\n	Перевод строки
\f	Перевод страницы
\t	Горизонтальная табуляция
\b	Возврат на одну позицию
\ddd	Восьмеричная константа (где <i>ddd</i> – восьмеричное число)
\uxxxx	Шестнадцатеричная константа (где <i>xxxx</i> – шестнадцатеричное число)

```
// Демонстрация управляющих последовательностей в
// символьных строках
class StrDemo {
    public static void main(String args[]) {
        System.out.println("Первая строка\nВторая строка");
        System.out.println("A\tB\tC");
        System.out.println("D\tE\tF");
    }
}
```

Используйте табуляцию
для выравнивания вывода

Используйте последовательность \n
для вставки символа перевода строки

Первая строка
Вторая строка
A B C
D E F

```
int a, b = 8, c = 19, d; // инициализация переменных b и с
```

```
// Демонстрация динамической инициализации
class DynInit {
    public static void main(String args[]) {
        double radius = 4, height = 5;
        // Переменная volume инициализируется динамически
        // во время выполнения программы
        double volume = 3.1416 * radius * radius * height;
        System.out.println("Объем: " + volume);
    }
}
```

Переменная volume
динамически
инициализируется
во время выполнения

```
// Демонстрация области действия блока кода
class ScopeDemo {
    public static void main(String args[]) {
        int x; // Эта переменная доступна для всего кода в
               // методе main

        x = 10;
        if(x == 10) { // Начало новой области действия

            int y = 20; // Эта переменная доступна только
                         // в данном блоке

            // Обе переменные, "x" и "y", доступны в данном блоке кода

            System.out.println("x и y: " + x + " " + y);
            x = y * 2;
        }

        // y = 100; // Ошибка! В этом месте переменная "y" недоступна ←
                  // Здесь переменная у
                  // находится вне своей
                  // области действия

        // А переменная "x" по-прежнему доступна
        System.out.println("x - это " + x);
    }
}
```

```
class NestVar {
    public static void main(String args[]) {
        int count; ←

        for(count = 0; count < 10; count = count+1) {
            System.out.println("Значение count: " + count);

            int count; // Недопустимо!!! ←
            for(count = 0; count < 2; count++)
                System.out.println("В этой программе есть ошибка!");
        }
    }
}
```

Нельзя объявлять переменную count, поскольку ранее она уже была объявлена

Ниже перечислены операторы сравнения.

Оператор	Значение
<code>==</code>	Равно
<code>!=</code>	Не равно
<code>></code>	Больше
<code><</code>	Меньше
<code>>=</code>	Больше или равно
<code><=</code>	Меньше или равно

Ниже перечислены логические операторы.

Оператор	Значение
<code>&</code>	И
<code> </code>	ИЛИ
<code>^</code>	Исключающее ИЛИ
<code> </code>	Укороченное ИЛИ
<code>&&</code>	Укороченное И
<code>!</code>	НЕ

```
// Демонстрация использования укороченных логических операторов
class SCops {
    public static void main(String args[]) {
        int n, d, q;

        n = 10;
        d = 2;
        if(d != 0 && (n % d) == 0)
            System.out.println(d + " является делителем " + n);

        d = 0; // установить для d нулевое значение

        // Второй operand не вычисляется, поскольку значение
        // переменной d равно нулю
        if(d != 0 && (n % d) == 0) ←
            System.out.println(d + " является делителем " + n);

        // А теперь те же самые действия выполняются без
        // использования укороченного логического оператора.
        // В результате возникнет ошибка деления на нуль.
        if(d != 0 & (n % d) == 0) ←
            System.out.println(d + " является делителем " + n);
    }
}
```

Укороченная
операция
предотвращает
деление на нуль

Теперь вычисляются
оба выражения,
в результате чего
будет выполняться
деление на нуль

```
// Демонстрация приведения типов
class CastDemo {
    public static void main(String args[]) {
        double x, y;
        byte b;

        int i;
        char ch;

        x = 10.0;
        y = 3.0;
        i = (int) (x / y); // привести тип double к типу int
        System.out.println("Целочисленный результат деления x / y: " + i);

        i = 100;
        b = (byte) i; // Представление символа X в коде ASCII
        System.out.println("Значение b: " + b);

        b = 257;
        b = (byte) i; // Явное приведение несовместимых типов
        System.out.println("Значение b: " + b);

        b = 88; // Представление символа X в коде ASCII
        ch = (char) b;
        System.out.println("ch: " + ch);
    }
}
```

В данном случае теряется дробная часть числа

А в этом случае информация не теряется.
Тип byte может содержать значение 100.

На этот раз информация теряется. Тип
byte не может содержать значение 257.

```
// Игра в угадывание букв, третья версия
class Guess3 {
    public static void main(String args[])
        throws java.io.IOException {

        char ch, answer = 'K';

        System.out.println("Задумана буква из диапазона A-Z.");
        System.out.print("Попытайтесь ее угадать: ");

        ch = (char) System.in.read(); // чтение символа с клавиатуры

        if(ch == answer) System.out.println("** Правильно! **");
        else {
            System.out.print("...Извините, нужная буква находится ");
            Вложенная
            инструкция if
                // вложенная инструкция if
                if(ch < answer) System.out.println("ближе к концу алфавита");
                else System.out.println("ближе к началу алфавита");
            }
        }
    }
}
```

```
switch(ch1) {  
    case 'A':  
        System.out.println("Это ветвь внешней инструкции switch");  
        switch(ch2) {  
            case 'A':  
                System.out.println("Это ветвь внутренней  
                    инструкции switch");  
                break;  
            case 'B': // ...  
        } // конец внутренней инструкции switch  
        break;  
    case 'B': // ...
```

```
// Выполнение цикла до тех пор, пока с клавиатуры
// не будет введена буква S
class ForTest {
    public static void main(String args[])
        throws java.io.IOException {

        int i;

        System.out.println("Для остановки нажмите клавишу S");

        for(i = 0; (char) System.in.read() != 'S'; i++)
            System.out.println("Проход #" + i);
    }
}
```

Bazele claselor

- Clasa este elementul de bază al Java.
- ***Clasa definește un nou tip de date.***
- **Clasa** este **un şablon** pentru crearea **obiectului**, iar **obiectul** este un **exemplu de clasă**.
- Forma simplă de clasă:

```
class nume_clasă {  
    tip variabilă_object_i;  
    tip nume_metoda_i (lista parametrilor formali) {  
        // corpul metodei  
    }  
}
```

Exemplu nr. 1

```
class Box {  
    double width;  
    double height;  
    double depth;  
}  
  
class BoxDemo {  
    public static void main(String args[]) {  
        Box mybox=new Box();  
        double vol;  
        mybox.width=10; mybox.height=20; mybox.depth=15;  
  
        vol = mybox.width*mybox.height*mybox.depth;  
        System.out.println("Volumul egal cu: " + vol);  
    }  
}
```

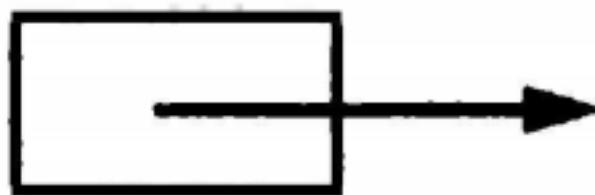
Volumul egal cu: 3000.0

Declararea obiectelor

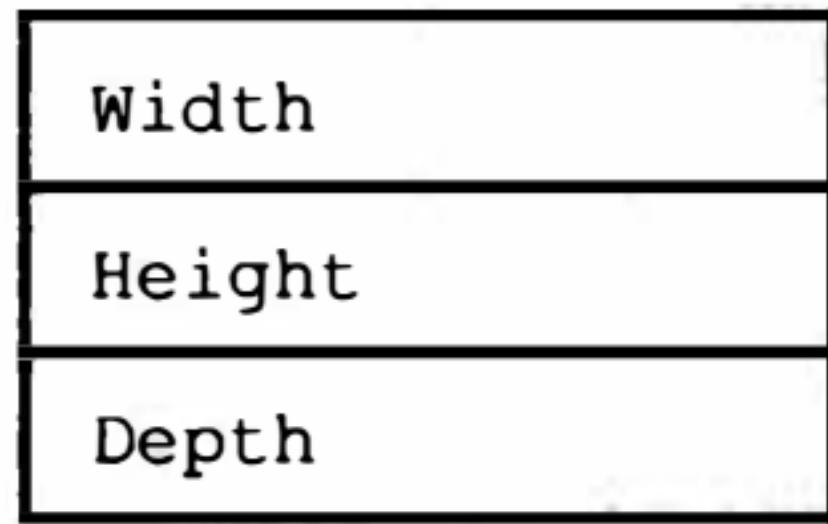
- Declararea obiectelor este un proces în două etape:
 - *Se definește variabila de tipul clasei;*
 - *Se obține copia fizică a obiectului și se atribuie acestei variabile.*
- A doua etapă se realizează cu ajutorul operatorului **new** care:
 - *Rezervează spațiu de memorie pentru obiect;*
 - *Returnează referință la el.*
- Referința reprezintă adresa obiectului în memorie, rezervată cu operatorul **new**.



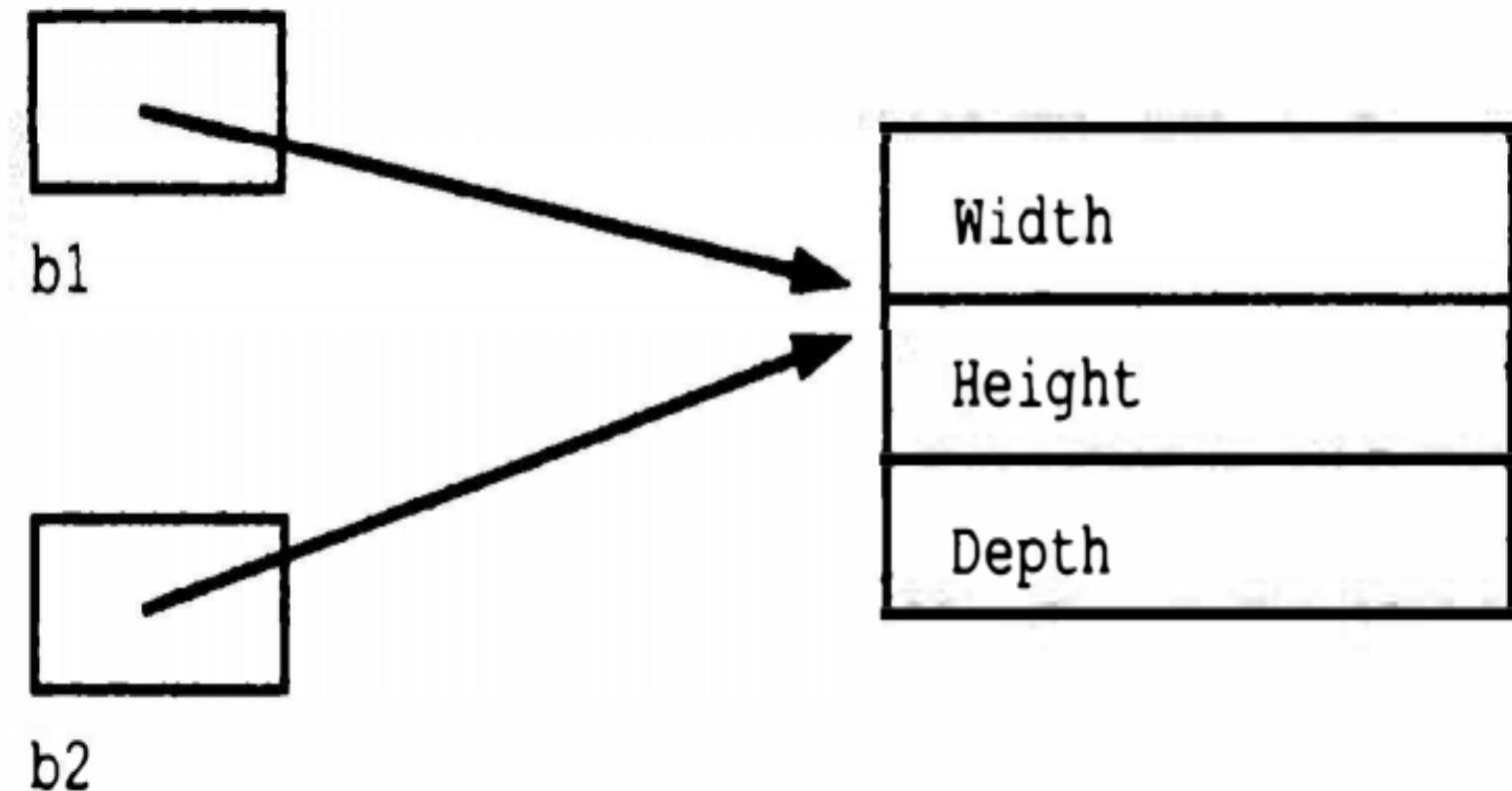
mybox



mybox



```
Box b1 = new Box();  
Box b2 = b1;  
// ...  
b1 = null;
```



Introducere în metode

- Forma generală de definire a metodelor este:
`tip nume(lista parametrilor formali)
{
 //corpu metodei
}`
- Metodele care au tipul diferit de void returnează valoarea indicată după operatorul return:
`return valoare;`

Implementarea metodei în clasa Box –

Varianta 1

```
class Box {  
    double width;  
    double height;  
    double depth;  
  
    void volume() {  
        System.out.println("Volumul egal cu");  
        System.out.println(width * height * depth);  
    }  
}  
  
class BoxDemo {  
    public static void main(String args[]) {  
        Box mybox1=new Box(); Box mybox2=new Box();  
        mybox1.width=10; mybox1.height=20; mybox1.depth=15;  
        mybox2.width=3; mybox2.height=6; mybox2.depth=9;  
  
        mybox1.volume();  
        mybox2.volume();  
    }  
}
```

Volumul egal cu 3000.0
Volumul egal cu 162.0

Implementarea metodei în clasa Box – Varianta 2

```
class Box {  
    double width;  
    double height;  
    double depth;  
  
    double volume() {  
        return width * height * depth;  
    }  
}  
  
class BoxDemo {  
    public static void main(String args[]) {  
        Box mybox1=new Box(); Box mybox2=new Box();  
        mybox1.width=10; mybox1.height=20; mybox1.depth=15;  
        mybox2.width=3; mybox2.height=6; mybox2.depth=9;  
        double vol;  
        vol=mybox1.volume(); System.out.println("Volum egal cu " + vol);  
        vol=mybox2.volume(); System.out.println("Volum egal cu " + vol);  
    }  
}
```

Volumul egal cu 3000.0

Volumul egal cu 162.0

Observații

- La folosirea metodelor cu returnare de valori trebuie să ținem cont de următoarele:
 - Tipul de date, care este returnat de metodă, trebuie să fie compatibil cu tipul de date indicat în metodă.
 - Variabila, care primește valoarea returnată de metodă, trebuie să fie compatibilă cu tipul indicat în metodă.
- `System.out.println("..." + mybox1.volume());`

Parametrii metodelor

- Metodă fără parametri:

```
int square() {  
    return 10 * 10;  
}
```

- Metodă cu parametri:

```
int square (int i) {  
    return i * i;  
}
```

Noțiuni

- **Parametru** – *variabila definită în metodă, care primește valoarea transmisă la apelul metodei.*

`int square (int i) { ... }` – i este parametru

- **Argument** – *este valoarea, care este transmisă la apelul metodei.*

`square(24)` – 24 este argumentul.

Implementarea metodei în clasa Box – Varianta 3

```
class Box {  
    double width;  
    double height;  
    double depth;  
    double volume() {  
        return width * height * depth;  
    }  
    void setDim (double w, double h, double d) {  
        width=w; height=h; depth=d;  
    }  
}  
  
class BoxDemo {  
    public static void main(String args[]) {  
        Box mybox1=new Box(); Box mybox2=new Box();  
        mybox1.setDim(10,20,15); mybox2.setDim(3,6,9);  
        double vol;  
        vol=mybox1.volume(); System.out.println("Volum egal cu " + vol);  
        vol=mybox2.volume(); System.out.println("Volum egal cu " + vol);  
    }  
}
```

Volumul egal cu 3000.0
Volumul egal cu 162.0

Noțiune de constructor

- Constructorul este o metodă care initializează obiectul nemijlocit la crearea lui.
- Numele lui coincide cu numele clasei în care este declarat.
- Constructorul nu returnează nici o valoare, nici nu are tip.
- Tipul evident al constructorului este tipul clasei din care face parte.

Implementarea metodei în clasa Box – Varianta 4

```
class Box {  
    double width;  
    double height;  
    double depth;  
    Box () {  
        width=10; height=10; depth=10;  
    }  
    double volume() {  
        return width * height * depth;  
    }  
}  
  
class BoxDemo {  
    public static void main(String args[]) {  
        Box mybox1=new Box(); Box mybox2=new Box();  
        double vol;  
        vol=mybox1.volume(); System.out.println("Volum egal cu " + vol);  
        vol=mybox2.volume(); System.out.println("Volum egal cu " + vol);  
    }  
}
```

Volumul egal cu 1000.0
Volumul egal cu 1000.0

Implementarea metodei în clasa Box – Varianta 5

```
class Box {  
    double width;  
    double height;  
    double depth;  
    Box (double w, double h, double d) {  
        width=w; height=h; depth=d;  
    }  
    double volume() {  
        return width * height * depth;  
    }  
}  
  
class BoxDemo {  
    public static void main(String args[]) {  
        Box mybox1=new Box(10,20,15); Box mybox2=new Box(3,6,9);  
        double vol;  
        vol=mybox1.volume(); System.out.println("Volum egal cu " + vol);  
        vol=mybox2.volume(); System.out.println("Volum egal cu " + vol);  
    }  
}
```

Volumul egal cu 3000.0

Volumul egal cu 162.0

Cuvântul cheie **this**

- **Varianta 1**

```
Box (double w, double h, double d) {  
    this.width=w;  
    this.height=h;  
    this.depth=d;  
}
```

- **Varianta 2**

```
Box (double width,double height,double depth) {  
    this.width=width;  
    this.height=height;  
    this.depth=depth;  
}
```

Garbage collector

- Aşa cum sănătate create multe obiecte dinamic (în timpul execuției) este nevoie de a elibera memoria.
- Acest lucru este realizat AUTOMAT de Garbage Collector.
- Mecanismul este următor:
 - La lipsa oricărora referințe la obiect se consideră că acest obiect nu trebuie și spațiul de memorie ocupat de el trebuie eliberat.

Metoda `finalize()`

- Uneori obiectul care este distrus trebuie să îndeplinească careva acțiuni.
- În aşa situație este pornit mecanismul de **finalizare absolută**, pentru care se poate descrie acțiunile concrete care se vor executa înainte apelului garbage collector.
- Finalizarea absolută poate fi descrisă în metoda **finalize()**:

```
protected void finalize() {  
    //codul finalizării absolute  
}
```
- Similar aproape destructorilor din alte limbaje de programare.

Suprăîncărcarea

- Suprăîncărcarea este o modalitate de implementarea polimorfismului în Java.
- Suprăîncărcarea este posibilitatea de a denumi mai multe metode cu același nume, dacă diferă:
 - *Tipul parametrilor*
 - *Numărul parametrilor*
 - *Rezultatul returnat*

Exemplu 1 (Clasa OverloadDemo)

```
class OverloadDemo  {
    void test() {
        System.out.println("Parametrii lipsesc");
    }
    void test(int a) {
        System.out.println("a: " +a);
    }
    void test(int a, int b) {
        System.out.println("a și b " +a + " " + b);
    }
    double test(double a) {
        System.out.println("double a: " +a);
        return a*a;
    }
}
```

Exemplu 1 (Clasa Overload)

```
class Overload {  
    public static void main(String args[]) {  
        OverloadDemo ob=new OverloadDemo();  
        double result;  
        ob.test();  
        ob.test(10);  
        ob.test(10,30);  
        result=ob.test(123.25);  
        System.out.println("Rezultatul apelului ");  
        System.out.println("ob.test(123.25) este");  
        System.out.println(result);  
    }  
}
```

Exemplu 1 (Rezultatul executiei)

Parametrii lipsesc

a: 10

a și b: 10 20

double a: 123.25

Rezultatul apelului

ob.test(123.25) este

15190.5625

Exemplu 2 – Supraîncărcarea constructorilor

```
class Box {  
    double width;  
    double height;  
    double depth;  
    Box (double w, double h, double d) {  
        width=w; height=h; depth=d;  
    }  
    Box () { width=-1; height=-1; depth=-1}  
    Box (double len) { width=height=depth=len}  
    double volume() {  
        return width*height*depth;  
    }  
}
```

Obiecte ca parametri (Exemplu 3)

```
class Test {  
    int a, b;  
    Test (int i, int j){ a=i; b=j }  
    boolean equals(Test o){  
        if (o.a==a && o.b==b) return true;  
        else return false;  
    }  
}  
class Pass0b {  
    public static void main(String args[]) {  
        Test ob1=new Test(100,22);  
        Test ob2=new Test(100,22);  
        System.out.println(ob1.equals(ob2));  
    }  
}
```

Constructori cu parametri de tip Obiect

```
class Box{  
    double width;  
    double height;  
    double depth;  
  
...  
  
Box (Box ob) {  
    width=ob.width;  
    height=ob.height;  
    depth=ob.depth;  
}  
  
...  
}
```

```
class Lucru{  
    public static void main  
        (String args[]) {  
        Box mybox1=new Box();  
        Box mybox2=new Box (mybox1)  
    }  
}
```

Transmiterea parametrilor

- Transmitere prin valoare
 - *în cazul tipurilor de date primitive*
- Transmiterea prin referință
 - *în cazul claselor și tipurilor de date structurate.*

Exemplu transmitere parametru Obiect

```
Class Test {  
    int a, b;  
    Test (int i, int j) { a=i; b=j }  
    void meth(Test o) {  
        o.a *=2;  
        o.b /=2;  
    }  
}  
class PassObjRe{  
    public static void main (String args[]){  
        Test ob=new Test(15, 20);  
        ob.meth(ob);  
        System.out.println(ob.a+ " "+ob.b)  
    }  
}
```

Returnarea rezultatului de tip Obiect

```
class Test {  
    int a;  
    Test (int i) { a=i; }  
    Test incrByTen() {  
        Test temp=new Test(a+10);  
        return temp;  
    }  
}  
class RetOb {  
    public static void main(String args[]) {  
        Test ob1=new Test(2);  
        Test ob2;  
        ob2=ob1.incrByTen();  
        System.out.println(ob2.a);  
    }  
}
```

Exemplu de metodă recursivă

- Fiind date 2 numere a și b se cere să se calculeze cel mai mare divizor comun al lor.

```
public class program47 {  
    static int cmmdc (int a, int b)  
    {  
        if (a==b) return a;  
        else if (a>b) return cmmdc(a-b, b);  
        else return cmmdc(a, b-a);  
    }  
    public static void main (String[] args) {  
        int a=24, b=36;  
        System.out.println(cmmdc(a,b));  
    }  
}
```

Modificatori

- Modificatori de clasa
- Modificatori date membru
- Modificatori metode

Modificatori de clasa

- **public** – clasa respectivă poate fi accesată de orice cod Java care accesează pachetul în care se găsește acea clasă.
- **abstract** – marchează o clasă abstractă.
- **final** – se folosește atunci când se dorește ca respectiva clasă să fie extinsă, caz în care această operatie este interzisă.

Modificatori date membru

- **public** – data membru poate fi accesată de oriunde este accesibilă și poate fi moștenită.
- **protected** – data membru poate fi accesată de codul aflat în același pachet și poate fi moștenită.
- **private** – data membru poate fi accesată numai din clasa în care se află și nu este moștenită.
- **final** – data membru este considerată constantă, deci nu poate fi modificată.
- **static** – data membru nu este memorată de fiecare obiect al clasei respective.

Modificatori de metode

- **public** – metoda poate fi accesată de oriunde este accesibilă și poate fi moștenită.
- **protected** – metoda poate fi accesată de codul aflat în același pachet și poate fi moștenită.
- **private** – metoda poate fi accesată numai din clasa în care se află și nu este moștenită.
- **final** – specifică faptul că într-o clasă derivată, metoda nu poate fi redefinită.
- **static** – are ca efect faptul că metoda nu este reținută de fiecare obiect al clasei respective.

Exemplu – modificatori de acces

```
class Test {  
    int a;  
    public int b;  
    private int c;  
    void setc (int i) { c=i; }  
    int getc() { return c; }  
}  
  
class AccessTest {  
    public static void main (String args[]) {  
        Test ob=new Test();  
        ob.a = 10;  
        ob.b = 20;  
        // ob.c= 100 // Eroare!!!  
        ob.setc(100); // CORECT!  
        System.out.println("a, b și c:" + ob.a + " " + ob.b  
+ " " + ob.getc());  
    }  
}
```

Cuvîntul cheie **static**

- Se folosește atunci când este nevoie de a folosi un membru al clasei fără a crea obiectul acelei clase.
- Variabila definită ca fiind **static**, de fapt este global.
- Metodele definite ca **static** au următoarele restricții:
 - Ele pot apela implicit doar alte metode statice;
 - În cadrul unor astfel de metode sînt accesibile doar variabile statice;
 - Ele nu pot face referință de tipul **this** sau **super**.

Exemplu:

```
class StaticDemo {  
    static int a = 42;  
    static int b = 99;  
    static void callme() {  
        System.out.println("a = " + a);  
    }  
}  
  
class StaticByName {  
    public static void main(String args[]) {  
        StaticDemo.callme();  
        System.out.println("b = " + StaticDemo.b);  
    }  
}
```


Din lecția precedentă ...

- Când a apărut prima versiune oficială Java?

Din lecția precedentă ...

- Când a apărut prima versiune oficială Java?
 - 23 mai 1995

Din lecția precedentă ...

- Când a apărut prima versiune oficială Java?
 - 23 mai 1995
- Cum se numește un program executat de sistem de operare și care oferă aplicațiilor Java toate posibilitățile necesare?

Din lecția precedentă ...

- Când a apărut prima versiune oficială Java?
 - 23 mai 1995
- Cum se numește un program executat de sistem de operare și care oferă aplicațiilor Java toate posibilitățile necesare?
 - Java Virtual Machine (JVM)

Din lecția precedentă ...

- Când a apărut prima versiune oficială Java?
 - 23 mai 1995
- Cum se numește un program executat de sistem de operare și care oferă aplicațiilor Java toate posibilitățile necesare?
 - Java Virtual Machine (JVM)
- Ce reprezintă o aplicație Java?

Din lecția precedentă ...

- Când a apărut prima versiune oficială Java?
 - 23 mai 1995
- Cum se numește un program executat de sistem de operare și care oferă aplicațiilor Java toate posibilitățile necesare?
 - Java Virtual Machine (JVM)
- Ce reprezintă o aplicație Java?
 - Orice aplicație Java este o colecție de clase, care descrie noi tipuri de obiecte.

Programarea orientată pe obiect 2

Prelegerea nr. 3

Elemente de bază ale limbajului Java.

Tipuri de date structurate.

Petic Mircea

petic.mircea@yahoo.com

www.facebook.com/mircea.petic

**Lucru cu
numere mari**

- Prin numere mari vom înțelege acele valori numerice care depășesc limitele de memorare ale tipurilor primitive.
- Operațiile cu astfel de numere sănt anevoieioase prin faptul că numărul trebuie memorat pe cifre (într-un vector).
- În Java există două clase care permit lucrul cu numere mari: clasa BigInteger și BigDecimal

Clasa BigInteger (1) - metode

- **BigInteger(String s)** – constructor
- **String toString();** - convertește numărul reținut de obiectul curent către un sir de caractere.
- **BigInteger add(BigInteger n)** – adună la numarul reținut de obiect
- **BigInteger subtract (BigInteger n)** – scade din numărul reținut de obiect
- **BigInteger multiply (BigInteger n)** – înmulțește numărul reținut de obiect

Clasa BigInteger (2) - metode

- **BigInteger divide (BigInteger n)** – împărțire întreagă a numărului reținut de obiect
- **BigInteger remainder (BigInteger n)** – returnează un obiect care reține restul împărțirii numărului reținut de obiectul curent la numărului obiectului transmis ca parametru.
- **BigInteger pow(BigInteger n)** – dacă obiectul curent reține m și obiectul transmis ca parametru valoarea n, atunci metoda returnează obiectul care reține m^n .

Clasa BigInteger (3) -metode

- **BigInteger negate();** - dacă obiectul curent reține m, atunci metoda returnează obiectul care reține $-m$.
- **int signum();** - dacă obiectul curent reține m se returnează obiectul care reține semnul lui m.
- **int CompareTo(BigInteger n);** - dacă obiectul curent reține m, iar obiectul referit reține n, atunci metoda returnează -1 dacă $m < n$, dacă $m = n$, 1 dacă $m > n$

Exemplu: se cere să se calculeze 2^{1000} .

Numărul depășește cu mult capacitatea de memorare a tipului long. Prin urmare se va utiliza clasa BigInteger.

```
import java.math.*;
class t{
    public static void main(String[] args){
        BigInteger n1=new BigInteger("1");
        BigInteger n2=new BigInteger("2");
        for(int i=1;i<=1000;i++)
            n1=n1.multiply(n2);
        System.out.println(n1.toString());
    }
}
```

Clasa BigDecimal (1) - metode

- **BigDecimal(String s)** – constructor
- **BigDecimal(BigInteger val)** – constructor
- **BigDecimal(double val)** - constructor
- **BigDecimal abs()** – returnează obiectul care reține modulul numărului reținut de obiectul curent.
- **String toString();** - convertește numărul reținut de obiectul curent către un sir de caractere.
- **BigDecimal add(BigDecimal n)** – adună la numarul reținut de obiect
- **BigDecimal subtract (BigDecimal n)** – scade din numărul reținut de obiect
- **BigDecimal multiply (BigDecimal n)** – înmulțește numărul reținut de obiect

Clasa BigDecimal (2) - metode

- **BigDecimal divide (BigDecimal n, int rotunjire)** – împărțire celor doi operanzi cu rotunjirea indicată
- **int scale();** - returnează numărul zecimalelor ale numărului reținut de obiectul curent.
- **BigDecimal setScale(int nrzec);** - returnează un obiect care memorează numărul reținut de obiectul curent cu nrzec zecimale.

Clasa BigDecimal (3) -metode

- **BigDecimal negate();** - dacă obiectul curent reține m, atunci metoda returnează obiectul care reține $-m$.
- **int signum();** - dacă obiectul curent reține m se returnează obiectul care reține semnul lui m.
- **int CompareTo(BigDecimal val);** - dacă obiectul curent reține m, iar obiectul referit reține n, atunci metoda returnează -1 dacă $m < n$, dacă $m = n$, 1 dacă $m > n$

Tipuri de date structurate

- Tablouri unidimensionale
- Tablouri bidimensionale

тип имя_массива[] = new тип[размер];

```
int sample[] = new int[10];
```

```
int sample[];  
sample = new int[10];
```

Masive - declarare

- Se declară o variabilă care poate reține o referință către vector/matrice:
 - int[] V; sau ... int V[];
 - int [][] mat; ... int mat[][]
- alocă spațiu de memorie pentru masiv
 - V=new int[4];
 - mat=new int[4][3];

```

// Демонстрация одномерного массива
class ArrayDemo {
    public static void main(String args[]) {
        int sample[] = new int[10];
        int i;

        for(i = 0; i < 10; i = i+1) ←
            sample[i] = i;
        ←
        for(i = 0; i < 10; i = i+1) ←
            System.out.println("Элемент[" + i + "]: " + sample[i]);
    }
}

```

Индексация массивов начинается с нуля

Элемент sample[0]: 0
 Элемент sample[1]: 1
 Элемент sample[2]: 2
 Элемент sample[3]: 3
 Элемент sample[4]: 4
 Элемент sample[5]: 5
 Элемент sample[6]: 6
 Элемент sample[7]: 7
 Элемент sample[8]: 8
 Элемент sample[9]: 9

Sample [0]	Sample [1]	Sample [2]	Sample [3]	Sample [4]	Sample [5]	Sample [6]	Sample [7]	Sample [8]	Sample [9]
0	1	2	3	4	5	6	7	8	9

Masive – exemple de inițializare

- `int[] v; v=new int[4];
for(int i=0;i<4;i++) v[i]=i+1;`
- `int[] v={1,2,3,4};`
- `int[][] mat={{1,2,3,4},{3,4,5,6}}`

Declarare într-un singur pas

- `float[] v=new float[4];`
- `System.out.println(v.length); -- 4`

```
// Поиск минимального и максимального значений в массиве
class MinMax {
    public static void main(String args[]) {
        int nums[] = new int[10];
        int min, max;

        nums[0] = 99;
        nums[1] = -10;
        nums[2] = 100123;
        nums[3] = 18;
        nums[4] = -978;
        nums[5] = 5623;
        nums[6] = 463;
        nums[7] = -9;
        nums[8] = 287;
        nums[9] = 49;

        min = max = nums[0];
        for(int i=1; i < 10; i++) {
            if(nums[i] < min) min = nums[i];
            if(nums[i] > max) max = nums[i];
        }
        System.out.println("min и max: " + min + " " + max);
    }
}
```

```
// Пример реализации алгоритма пузырьковой сортировки
for(a=1; a < size; a++)
    for(b=size-1; b >= a; b--) {
        if(nums[b-1] > nums[b]) { // если требуемый порядок следования
                                // не соблюдается, поменять элементы
                                // местами
            t = nums[b-1];
            nums[b-1] = nums[b];
            nums[b] = t;
        }
    }
}
```

```

// Демонстрация использования двумерного массива
class Twod {
    public static void main(String args[]) {
        int t, i;
        int table[][] = new int[3][4];

        for(t=0; t < 3; ++t) {
            for(i=0; i < 4; ++i) {
                table[t][i] = (t*4)+i+1;
                System.out.print(table[t][i] + " ");
            }
            System.out.println();
        }
    }
}

```

3 ← правый индекс

	0	1	2	3
0	1	2	3	4
1	5	6	7	8
2	9	10	11	12

левый индекс

table[1][2]

Tablouri bidimensionale iregulare

```
int table[][] = new int[3][];
table[0] = new int[4];
```

```
int riders[][] = new int[7][];
riders[0] = new int[10];
riders[1] = new int[10];
riders[2] = new int[10];
riders[3] = new int[10];
riders[4] = new int[10];
riders[5] = new int[2];
riders[6] = new int[2];
```

Для первых пяти элементов длина массива по второму индексу равна 10

Для остальных двух элементов длина массива по второму индексу равна 2

Masive - length

- ```
int mat[][]=new int [2][]
mat[0]=new int[3]; mat[1]=new int[2];
mat[0][0]=1;mat[0][1]=2;mat[1][0]=3;
System.out.println(mat[1].length) // 2
```
- ```
int mat[][]=new int[4][5];
System.out.println(mat.length); // 4
System.out.println(mat[2].length); // 5
```

```
// Инициализация двумерного массива
class Squares {
    public static void main(String args[]) {
        int sqrs[][] = {
            { 1, 1 },
            { 2, 4 },
            { 3, 9 },
            { 4, 16 },
            { 5, 25 },
            { 6, 36 },
            { 7, 49 },
            { 8, 64 },
            { 9, 81 },
            { 10, 100 } }
        ;
        int i, j;

        for(i=0; i < 10; i++) {
            for(j=0; j < 2; j++)
                System.out.print(sqrs[i][j] + " ");
            System.out.println();
        }
    }
}
```

Обратите внимание на то, что у каждой строки свой набор инициализаторов

```
// Пример использования переменной length для копирования массивов
class ACopy {
    public static void main(String args[]) {
        int i;
        int nums1[] = new int[10];
        int nums2[] = new int[10];

        for(i=0; i < nums1.length; i++)
            nums1[i] = i;

        // Копирование массива nums1 в массив nums2
        if(nums2.length >= nums1.length) ← Использование переменной length
            for(i = 0; i < nums2.length; i++) для сравнения размеров массивов
                nums2[i] = nums1[i];

        for(i=0; i < nums2.length; i++)
            System.out.print(nums2[i] + " ");
    }
}
```

Citirea datelor de la tastatură

```
import java.util.Scanner;
class citire {
public static void main(String[] args) {

    Scanner scan=new Scanner(System.in);
    int k=scan.nextInt();

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

Instrucțiunea repetitivă for-each

- **Declararea unui tablou unidimensional**

```
double[] V; V=new double[4];
```

```
for(int i=0;i<4;i++) V[i]=i+1;
```

- **Afișarea la ecran a elementelor tabloului**

a) *for(int i=0;i<4;i++) {*

System.out.print(V[i]+” ”); }

b) *for(double i: V) {*

System.out.print(i+” ”); }

Alt exemplu for-each

```
int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int sum = 0;

for(int i=0; i < 10; i++) sum += nums[i];
```

```
int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int sum = 0;

for(int x: nums) sum += x;
```

Exemplu de program

```
// Использование цикла типа for-each
class ForEach {
    public static void main(String args[]) {
        int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
        int sum = 0;

        // Использование цикла типа for-each для
        // суммирования и отображения значений
        for(int x : nums) { ←———— Цикл типа for-each
            System.out.println("Значение: " + x);
            sum += x;
        }

        System.out.println("Сумма: " + sum);
    }
}
```

Tablou bidimensional & for-each

```
// Использование расширенного цикла for
// для обработки двумерного массива
class ForEach2 {
    public static void main(String args[]) {
        int sum = 0;
        int nums[][] = new int[3][5];

        // Ввод ряда значений в массив nums
        for(int i = 0; i < 3; i++)
            for(int j=0; j < 5; j++)
                nums[i][j] = (i+1)*(j+1);

        // Использование цикла типа for-each для
        // суммирования и отображения значений.
        for(int x[] : nums) { ←———— Обратите внимание на способ объявления переменной x
            for(int y : x) {
                System.out.println("Значение: " + y);
                sum += y;
            }
        }
        System.out.println("Сумма: " + sum);
    }
}
```

Programare orientată pe obiect 2

Prelegerea nr. 4

Elemente de bază ale limbajului Java.

Lucrul cu sirurile de caractere.

Petic Mircea

petic.mircea@yahoo.com

www.facebook.com/mircea.petic

Clasa String

```
// Знакомство с классом String
class StringDemo {
    public static void main(String args[]) {
        // Различные способы объявления строк
        String str1 = new String("В Java строки - это объекты.");
        String str2 = "Их можно создавать разными способами.";
        String str3 = new String(str2);

        System.out.println(str1);
        System.out.println(str2);
        System.out.println(str3);
    }
}
```

В результате выполнения этой программы будет получен следующий результат:

В Java строки - это объекты.
Их можно создавать разными способами.
Их можно создавать разными способами.

```
public static void main(String args[]) {  
    String strs[] = { "Эта", "строка", "является", "тестом." };  
  
    System.out.println("Исходный массив: ");  
    for(String s : strs)  
        System.out.print(s + " ");  
    System.out.println("\n");  
  
    // Изменение строки  
    strs[2] = "также является";  
    strs[3] = "тестом!";  
  
    System.out.println("Измененный массив: ");  
    for(String s : strs)  
        System.out.print(s + " ");  
}  
}
```

В результате выполнения этого фрагмента кода будет получен следующий результат.

Исходный массив:
Эта строка является тестом.

Измененный массив:
Эта строка также является тестом!

Constructori unui sir de caractere

- Un obiect al clasei String reține un sir de caractere.
- Clasa String este înzestrată cu doi constructori:
 - String() – inițializarea unui obiect String cu un sir vid.

```
String s=new String();
s="un sir";
System.out.println(s);
```

- String(String s) – inițializarea unui obiect String cu un sir de caractere dat.

```
String s=new String("un sir");
System.out.println(s);
```

Explicație

In primul caz obiectul este alocat in String Pool, acesta fiind parte componenta a memoriei heap.

De fiecare data cand un string este creat prin atribuire - JVM cauta intai sa vada daca acea valoare nu este deja in String Pool, daca da noua variabila va referi acea valoare din String Pool, daca nu, va fi creata una noua.

In al doilea caz, de fiecare data in memoria JVM va fi creata un nou obiect de tip String, deci, de fiecare data va fi referita alta zona de memorie.

Exemplu

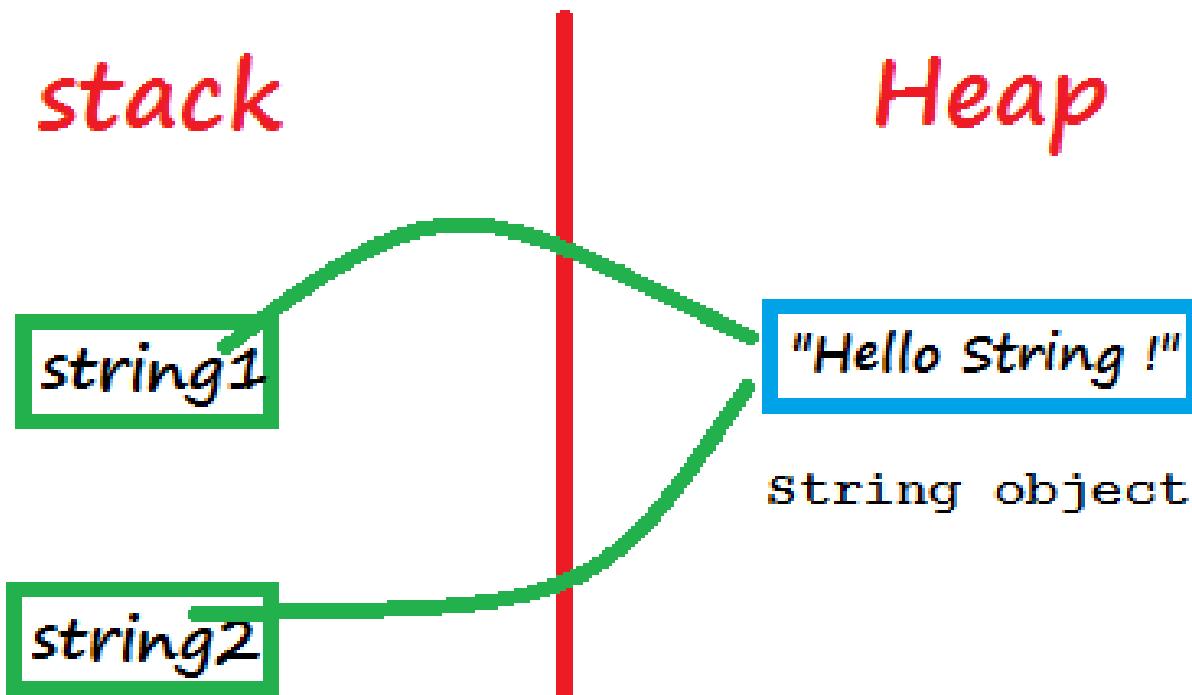
- În Java, sirurile de caractere sunt obiecte. Atunci cand se utilizeaza = intre 2 referinte String veti copia valoarea referintei si nu valoarea obiectului.

Urmatorul exemplu:

```
String string1 = "Hello Strings";
        String string2 = string1;
if ( string1 == string2 )
        System.out.println("egale");
else
        System.out.println("NU sunt egale");
```

va afisa la consola ***egale***, deoarece referintele au aceeasi valoare/adresa si genereaza in memorie aceasta situatie:

Exemplu de reprezentare



String references © 2011 www.itcsolutions.eu

Ce inseamna ca String este imutabil (immutable)

- Un obiect imuabil (immutable) este un obiect care odata ce este creat nu isi poate modifica valoarea. String este o clasa imuabila si se bazandu-ne pe exemplul anterior, vedem ce se intampla atunci cand facem ceva de genul:

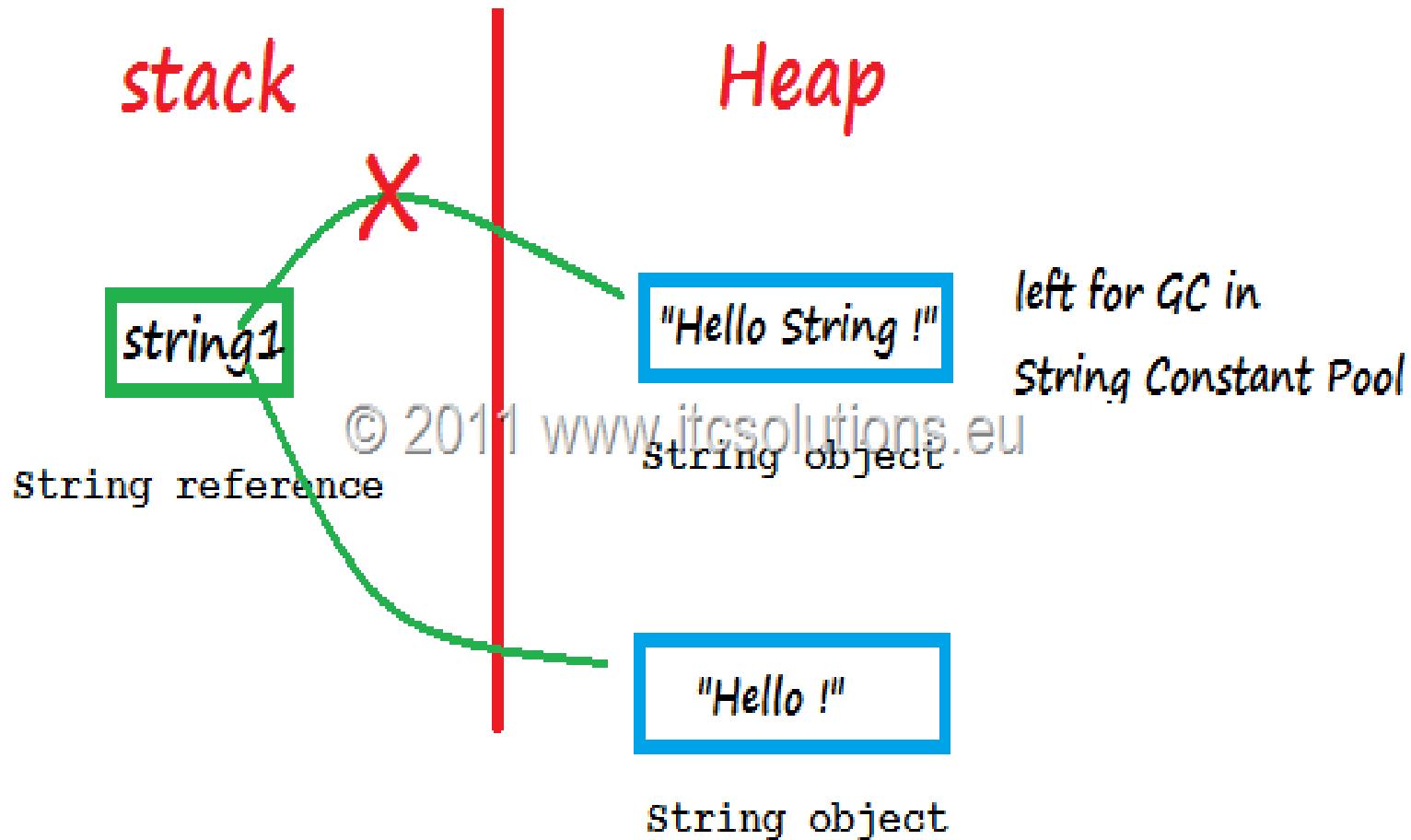
```
String string1 = "Hello String!";
System.out.println(string1);
string1 = "Hello !";
System.out.println(string1);
```

Daca rulati acest exemplu, veti obtine

Hello String!

Hello Java!

Exemplu de reprezentare



```
String string3 = "Hello String !" ;
String string4 = "Hello String !" :
// compara referinte de obiecte
String, NU valorile lor
if(string3 == string4)
    System.out.println("EGALE");
else
    System.out.println("NU sunt
egale");
```

va printa EGALE deoarece datele sunt prelucrate astfel:

stack

Heap

string3

string4

(1)

(2)

String constant pool

"Hello String!"

String object

String references

© 2011 www.itcsolutions.eu

Lungimea unui sir de caractere

- int length() – întoarce numărul de caractere din sir.

Exemplu:

```
String s=new String("un sir");
```

```
System.out.println(s.length());
```

Adresarea la un caracter al șirului

- În Java nu este permisă adresarea unui caracter al șirului prin indice, pentru exemplul precedent `s[k]`.
- **Metoda specială!!!**
- **char charAt(int i)** – returnează caracterul de pe poziția *i*.

```
String s=new String("un sir");
```

```
System.out.println(s.charAt(1));
```

Cum se poate prelucra o valoare String

Metoda	Descriere
<code>charAt()</code>	intoarce caracterul aflat la un anumit indice; indicele ia valori de la 0 la <code>length()-1</code> ;
<code>concat()</code>	concateneaza un String la sfarsitul celui existent; la fel ca <code>+</code> ;
<code>equals()</code>	compara la nivel de caracter 2 valori String; face diferenta intre litere mici si mari
<code>length()</code>	returneaza numarul de caractere; NU ESTE atributul length al unui vector. Este o metoda.
<code>replace()</code>	inlocuieste aparitiile unui caracter cu unul primit
<code>substring()</code>	returneaza un subsir
<code>toLowerCase()</code>	converteste toate caractere la litere mici
<code>toString()</code>	returneaza valoarea obiectului String
<code>toUpperCase()</code>	converteste toate caractere la majuscule
<code>trim ()</code>	elimina spatiul de la sfarsitul sirului de caractere

Alte metode de prelucrare String

Metoda	Descriere
append()	adauga argumentul la sfarsitul obiectul curent
delete()	sterge caractere intre un index de inceput si indicele de sfarsit
insert()	introduce un sir de caractere la un anumit offset
reverse()	inverseaza valoarea obiectului curent
toString()	returneaza valoarea obiectului StringBuilder sau StringBuffer

Compararea șirurilor de caractere

- Șirurile de caractere reținute de obiectele clasei *String* pot fi comparate din punct de vedere lexicografic.
- `int compareTo(String s)` – compară șirul de caractere reținut de obiectul curent cu șirul de caractere reținut s. Metoda returnează:
 - O valoare negativă, dacă șirul reținut de obiectul curent este situat înaintea șirului reținut de obiectul s.
 - 0 (zero), dacă șirurile reținute de cele două obiecte coincid.
 - O valoare pozitivă, dacă șirul reținut de obiectul curent este situat după șirul reținut de obiectul s.

Compararea sirurilor de caractere - exemplu

- Secvența de mai jos va afișa -2 0 2

```
String s=new String("abc");
String s1=new String("cd");
System.out.println(s.compareTo(s1)+" "+
s.compareTo(s)+" "+s1.compareTo(s));
```

```
int compareTolgnoreCase(String s)
```

- La fel precum compareTo(), numai că nu se face diferența între literele mari și mici.
- Exemplu: secvența de mai jos afișează 0 (zero).

```
String s=new String("AB");
```

```
String s1=new String("ab");
```

```
System.out.println(s.compareTolgnoreCase(s1));
```

Compararea sirurilor de caractere (3)

- **boolean equals(String s)** – returnează true dacă sirurile de caractere reținute de obiectul curent și cel transmis ca parametru sunt identice.
- **boolean equalsIgnoreCase(String s)** – la fel ca mai sus numai că nu se face distincție între literele mari și cele mici.

Subșiruri (1)

- Vom înțelege prin subșir al unui sir de caractere, mai multe caractere consecutive ale acestuia. De exemplu, "text" are ca sușir "ex".
- **boolean startsWith(String s)** – returnează true dacă sirul reținut de s precede sirul reținut de obiectul curent.
- **boolean endsWith(String s)** – returnează true dacă sirul reținut de s se află la sfîrșitul sirului reținut de obiectul curent.

Exemple pentru **startsWith** și **endsWith**

- Secvența de mai jos afișează de două ori true.

```
String s=new String("123");
```

```
String s1=new String("12");
```

```
String s3=new String("23");
```

```
System.out.println(s.startsWith(s1));
```

```
Systme.out.println(s.endsWith(s3));
```

```
boolean regionMatches(int i1, String s, int i2, int l);
```

- compară două subșiruri de lungime l. Primul subșir este al obiectului curent și începe de pe poziția i1, al doilea subșir este al sirului s și începe pe poziția i2.
- În caz de egalitate, metoda returnează **true**, contrar returnează **false**.

Exemplu: Se afișează sirul **567 coincid**:

```
if (s.regionMatches(4,s1,2,3))  
    System.out.println("567 coincid");
```

String substring(int index)

- metoda returnează subșirul șirului curent care este între poziția index și sfîrșitul șirului.
- Exemplu: secvența de mai jos afișează 456789.

```
String s=new String("123456789");
```

```
System.out.println(s.substring(3));
```

String substring(int index, int sf)

- metoda returnează subșirul șirului curent care este între poziția index și sf-1.
- Exemplu: secvența de mai jos afișează 45.

```
String s=new String("123456789");
```

```
System.out.println(s.substring(3,5));
```

String replace (char c1, char c2)

- Metoda returnează sirul obținut dacă se înlocuiesc în sirul reținut de obiectul curent toate aparițiile caracterului c1 cu caracterul c2.
- Exemplu: secvența de mai jos afișează tata.

```
String s=new String("mama");
```

```
System.out.println(s.replace('m', 't'));
```

String replaceAll (String s1, String s2)

- Metoda returnează sirul obținut dacă se înlocuiesc în sirul reținut de obiectul curent toate aparițiile subșirului s1 cu subșirul s2.
- Exemplu: secvența de mai jos afișează:
*** doi *** doi trei

```
String s=new String("unu doi unu doi trei");
System.out.println(s.replaceAll("unu","***"));
```

String replaceFirst(String s1, String s2)

- La fel ca mai sus, doar că se înlocuiește numai prima apariție a subșirului.
- Exemplu: secvența de mai jos afișează:
 *** doi unu doi trei

```
String s=new String("unu doi unu doi trei");
System.out.println(s.replaceFirst("unu","***"));
```

Subșiruri (5)

- **String trim()** – întoarce subșirul reținut de obiectul curent, subșir obținut prin eliminarea spațiilor de la început și de la sfîrșit.
- **int indexOf(String s)** – returnează indicele primei apariții a subșirului s în sirul reținut de obiectul curent. În cazul în care s nu este găsit ca subșir al sirului referit de obiectul curent, metoda returnează -1.
- Exemplu: secvența de mai jos afișează 4.

```
String s1=new String("un exemplu");
String s2=new String("xe");
System.out.println(s1.indexOf(s2));
```

Concatenarea șirurilor (1)

- Concatenarea cu ajutorul operatorului +
- ***Varianta A***

```
String s=new String ("un sir");
```

```
String s1=new String(" Alt sir");
```

```
s=s+s1;
```

- ***Varianta B***

```
String s=new String("un sir "+" alt sir");
```

```
System.out.println(s);
```

Concatenarea șirurilor (2)

- String concat(String s) – returnează șirul obținut prin concatenarea șirului reținut de obiectul curent s.
- Exemplu: secvența următoare afișează

Un sir alt sir

```
String s=new String("Un sir ");
String s1=new String("alt sir");
System.out.println(s.concat(s1));
```

Parametrii metodei main()

```
public static void main(String[] args)
```

- antetul conține un vector cu elemente de tip String.
- Numele vectorului este args.
- Numărul de elemente ale vectorului este:
args.length
- Sirurile vectorului sănt introduse în linia de comandă și sănt separate prin spațiu.

Exercițiu: programul de mai jos calculează suma valorilor introduse ca parametri.

- Dacă apelul este **java t 3 4** se va afișa 7
- Dacă apelul este **java t 1 2 3 4** se va afișa 10

```
class t{  
    public static void main(String[] args){  
        int s=0;  
        for(int i=0;i<args.length;i++)  
            s+=Integer.parseInt(args[i]);  
        System.out.println("suma argumentelor este= "+s);  
    }  
}
```

Exemplu: Fie 2 șiruri de caractere. Să se listeze indicii fiecărei apariții a primului șir ca subșir al celui de-al doilea.

```
mama
tata mama tata mama tata mama
5 15 25
public static void main(String[] args){
    String subsir="mama";
    String sir="tata mama tata mama tata mama";
    int poz=0, ind=0, lsubs=subsir.length();
    while (poz!=-1) {    poz=sir.indexOf(subsir);
        if (poz!=-1) {    System.out.print(ind+poz);
            ind+=poz+lsubs;
            sir=sir.substring(poz+lsubs); }
    }
}
```

```
// Отображение всех данных, указываемых в командной строке
class CLDemo {
    public static void main(String args[]) {
        System.out.println("Программе передано " + args.length +
                           " аргумента командной строки.");
        System.out.println("Список аргументов: ");
        for(int i=0; i<args.length; i++)
            System.out.println("arg[" + i + "]: " + args[i]);
    }
}
```

Допустим, программа CLDemo была запущена из командной строки следующим образом:

```
java CLDemo one two three
```

Тогда результат ее выполнения будет следующим.

Программе передано 3 аргумента командной строки.

Список аргументов:

```
arg[0]: one
arg[1]: two
arg[2]: three
```

Clasa StringTokenizer

Clasa StringTokenizer

- Clasa StringTokenizer se găsește în pachetul java.util.
- Clasa StringTokenizer conține metode care permit extragerea unităților lexicale.
- Implicit unitățile lexicale se consideră separate prin caracterele albe – ‘ ’, ‘\n’, ‘\t’, ‘\r’, ‘\f’.
- **StringTokenizer(String s)** – constructor, creează obiectul care tratează sirul de caractere referit de s.
- **StringTokenizer(String s, String delim)** – constructor, creează obiectul care tratează sirul de caractere referit de s, dar tokenurile sănt separate de caracterele care alcătuiesc delim.

Metodele StringTokenizer

- **boolean hasMoreTokenizer()** – returnează true dacă sirul mai conține tokenuri necitite și false în caz contrar.
- **String nextToken()** – returnează sirul de caractere care alcătuiesc următorul token (citește un token);
- **int countTokens()** – returnează numărul de token-uri rămase necitite (cu nextToken()).

Exemplu: programul avînd o linie de sir de caractere, afișează token-urile întîlnite.

- De exemplu: mama are 3500 lei
- Afișează:

mama
are
3500
lei

```
import java.util.*;  
class t{  
    public static void main(String[] args) {  
        String s="mama are 3500 lei";  
        StringTokenizer tk=new StringTokenizer(s);  
        while(tk.hasMoreTokens())  
            System.out.println(tk.nextToken());  
    }  
}
```

```

// Простейший автоматизированный телефонный справочник
class Phone {
    public static void main(String args[]) {
        String numbers[][] = {
            { "Tom", "555-3322" },
            { "Mary", "555-8976" },
            { "John", "555-1037" },
            { "Rachel", "555-1400" }
        };
        int i;

        // Для того чтобы воспользоваться программой,
        // ей нужно передать один аргумент командной строки
        if(args.length != 1) ←
            System.out.println("Использование: java Phone <имя>");
        else {
            for(i=0; i<numbers.length; i++) {
                if(numbers[i][0].equals(args[0])) {
                    System.out.println(numbers[i][0] + ": " +
                        numbers[i][1]);
                    break;
                }
            }
            if(i == numbers.length)
                System.out.println("Имя не найдено.");
        }
    }
}

```

Для выполнения программы
нужен как
минимум один
аргумент
командной
строки

Выполнение этой программы может дать, например, следующий результат.

```
C>java Phone Mary
Mary: 555-8976
```

**Clase
înfășurătoare**

Clase înfășurătoare

- Pentru fiecare tip primitiv s-a construit cîte o clasă înfășurătoare.
- Un obiect al clasei înfășurătoare poate reține o valoare a tipului primitiv.
- Fiecare clasă înfășurătoare este înzestrată cu o metodă constructor.
- Constructorii acestor clase sunt supraîncărcați, în sensul că există și constructori care au ca parametru de intrare referințe către obiecte de tip String.

Corespondență

tip primitiv – clasă infășurătoare

int – *Integer*

short – *Short*

Long – *Long*

byte – *Byte*

char – *Character*

float – *Float*

double – *Double*

boolean – *Boolean*

Exemplu utilizare constructori

```
int n=4;
```

```
Integer nr=new Integer(n);
```

```
Double x=new Double(-12.34);
```

```
Character c=new Character('e');
```

```
Boolean este=new Boolean(true);
```

```
Integer nr=new Integer("10");
```

```
Double x=new Double("-12.34");
```

```
Boolean este=new Boolean("true");
```

Conversia către un tip primitiv

- Fiecare clasă înfășurătoare conține o metodă care returnează valoarea reținută de obiectul instanțiat de ea:

```
int n1=nr.intValue();
```

```
double x1=x.doubleValue();
```

```
char c1=c.charValue();
```

```
boolean este1=este.booleanValue()
```

Fiecare clasă care reține o valoare numerică conține câte o metodă statică pentru conversia către un tip primitiv a unui obiect de tip String.

- int parseInt(String s);
int t=Integer.parseInt("10");
- float parseFloat(String s);
float b=Float.parseFloat("-12.34");
- Double parseDouble(String s);
double b=Double.parseDouble("-12.34");

Conversii

- Dacă sirul de caractere nu poate fi convertit către o valoare numerică de tipul dorit, se generează o excepție în urma căreia executarea programului se întrerupe.
- Conversia inversă, de la un tip primitiv către un sir se poate realiza ușor concatenând un sir cu o valoare numerică:

```
int i=10;  
String s=new String(i+'');  
System.out.println(s);
```

Constanțele MIN_VALUE și MAX_VALUE

- Clasele înfășurătoare conține constantele MIN_VALUE și MAX_VALUE care rețin cea mai mică și cea mai mare valoare a tipului respectiv.

```
System.out.println(Integer.MIN_VALUE);
```

```
System.out.println(Integer.MAX_VALUE);
```

```
System.out.println(Double.MIN_VALUE);
```

```
System.out.println(Double.MAX_VALUE);
```

Constanta NaN (Not a Number)

- Valoarea NaN se obține dacă, într-o expresie de tip real,
 - se împarte 0 la 0,
 - se extrage radical (indice 2) dintr-o valoare negativă
 - se aplică logaritmul unui număr negativ ... Etc
- a. `double x=0, y=0, z;`
`z=x/y; System.out.println(z);`
- b. `int x=-2;`
`System.out.println(Math.sqrt(x));`

Metoda isNaN

- Metoda returnează true dacă se obține NaN.
- boolean isNaN(double a);
- boolean isNaN(float a);
- Exemplu: Se va afișa Ambii operanzi sănt 0.

```
int x; double y=0;
```

```
if (Double.isNaN(x/y))
```

```
System.out.println("Ambii operanzi sănt 0");
```

Constancele

POSITIVE_INFINITY și NEGATIVE_INFINITY

- Se obțin într-o expresie de tip real, cînd se împarte o valoare (pozitivă sau negativă) la 0.
- **double x=2; System.out.println(x/0);**
- **double x=-2; System.out.println(x/0);**
- **boolean isInfinite(double/float x);** - metoda prin care se testează dacă am obținut una din constantele de mai sus.

```
double x=2;
```

```
if (Double.isInfinite(x/0))
```

```
    System.out.println("Împartire la 0");
```

Extinderea claselor (Moștenire)

- Moștenirea este unul din cele trei principii POO;
- Pornind de la o clasă dată se poate crea o altă clasă, care o extinde (**moștenește**);
- Clasa care este extinsă se numește **superclasa**;
- Noua clasă se numește **subclasă**;
- *Subclasa* este o versiune specială a superclasei, care moștenește toate variabile și metodele din superclasă completându-se cu elementele proprii.

```
class имя_подкласса extends имя_суперкласса {  
    // тело класса  
}
```

```
// Простая иерархия классов

// Класс, описывающий двумерные объекты
class TwoDShape {
    double width;
    double height;

    void showDim() {
        System.out.println("Ширина и высота - " + width + " и " +
                           height);
    }
}
```

```
// Подкласс для представления треугольников,
// производный от класса TwoDShape
```

```
class Triangle extends TwoDShape {
```

```
    String style;
```



Класс Triangle наследует класс TwoDShape

```
    double area() {
```

```
        return width * height / 2; ← Из класса Triangle можно обращаться к
```

```
    }
```

членам класса TwoDShape так, как если бы это были его собственные члены

```
    void showStyle() {
```

```
        System.out.println("Треугольник " + style);
```

```
}
```

```
}
```

```
class Shapes {
    public static void main(String args[]) {
        Triangle t1 = new Triangle();
        Triangle t2 = new Triangle();

        t1.width = 4.0;
        t1.height = 4.0; ←————— Объектам типа Triangle доступны все члены
        t1.style = "закрашенный";
        класса Triangle, даже те, которые унаследованы
        от класса TwoDShape

        t2.width = 8.0;
        t2.height = 12.0;
        t2.style = "контурный";

        System.out.println("Информация о t1: ");
        t1.showStyle();
        t1.showDim();
        System.out.println("Площадь - " + t1.area());

        System.out.println();

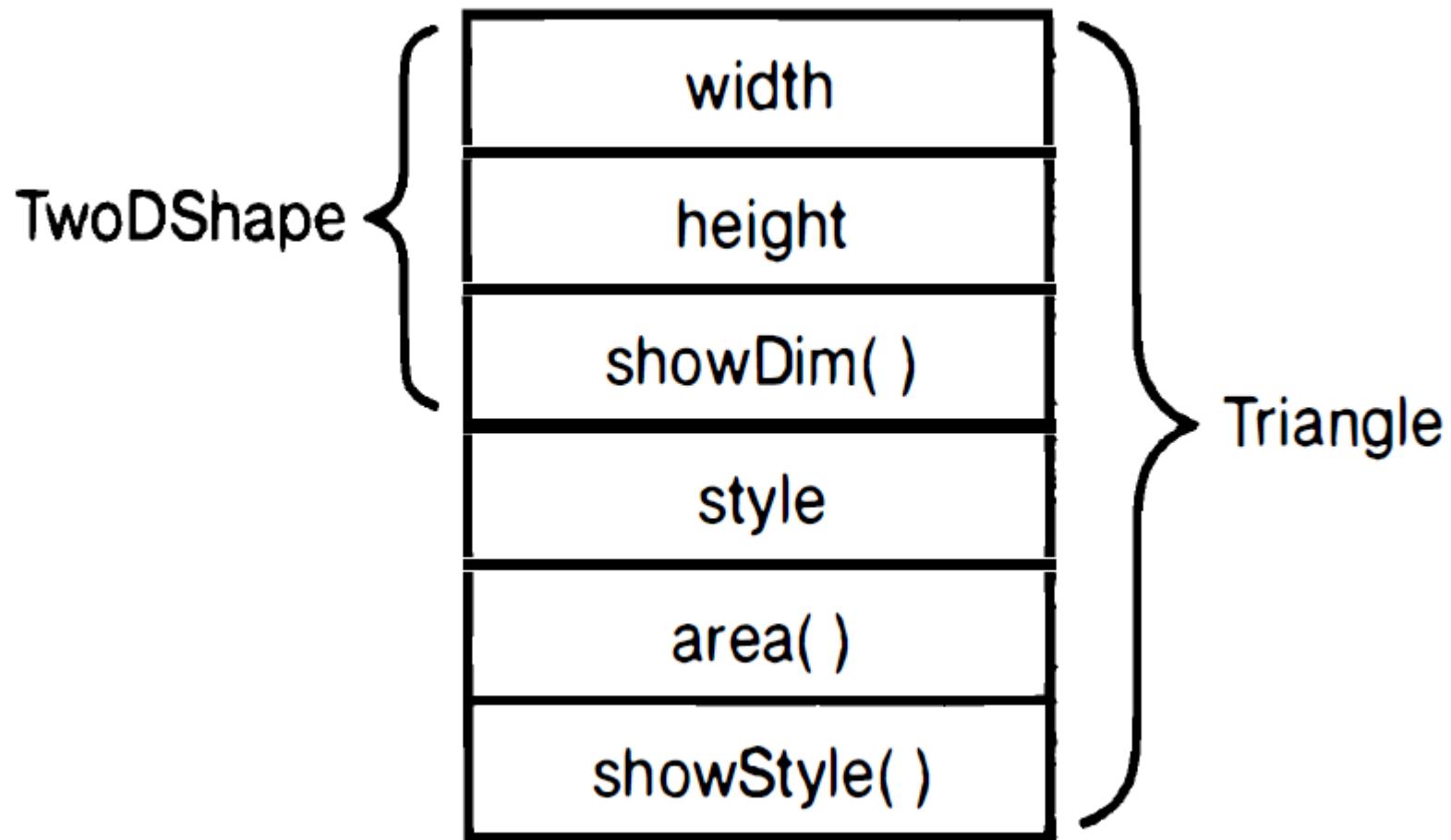
        System.out.println("Информация о t2: ");
        t2.showStyle();
        t2.showDim();
        System.out.println("Площадь - " + t2.area());
    }
}
```

Выполнение этой программы дает следующий результат.

```
Информация о t1:
Треугольник закрашенный
Ширина и высота - 4.0 и 4.0
Площадь - 8.0
```

```
Информация о t2:
Треугольник контурный
Ширина и высота - 8.0 и 12.0
Площадь - 48.0
```

Structura clasei Triangle



Moștenirea multiplă interzisă în Java!

- În comparație cu LP C++, unde e posibilă moștenirea multiplă, în LP Java este interzisă moștenirea multiplă!
- Totuși este posibilă o structură ierarhică de moștenire!
- De exemplu, o subclasă poate fi superclasă pentru altă subclasă!
- O clasă nu poate fi superclasă pentru ea însăși!

Moștenire – exemplul 1

```
class c1 { int x,y;}  
class c2 extends c1 { int z;  
    void afis() { System.out.println(x+" "+y+"  
"+z); }  
}  
public class c {  
    public static void main(String[] args){  
        c2 ref=new c2();  
        ref.x=3; ref.y=4;ref.z=5;  
        ref.afis();  
    }  
}
```

Moștenire – exemplul 2

```
class unu {  
    unu() {System.out.println("Constructorul  
1");}  
}  
class doi extends unu {  
    doi() {System.out.println("Constructorul  
2");}  
}  
public class c {  
    public static void main(String[] args) {  
        doi x=new doi();  
    }  
}
```

Moștenire – exemplul 3

```
class c1 { int x=1,y; }
class c2 extends c1 {
    int x;
    void afis() {System.out.println(super.x+" "+y+
"+x);}
}
class c {
    public static void main (String[] args) {
        c2 pt=new c2();
        pt.x=3; pt.y=2;
        pt.afis();
    }
}
```

Se va afisa: 1 2 3

Moștenire – exemplul 4

```
class elev {  
    String nume, prenume;  
    elev(String nume, String prenume) {  
        this.nume=new String(nume);  
        this.prenume=new String(prenume);  
    }  
}  
class olimpic extends elev {  
    String materia;  
    olimpic (String nume, String prenume, String materia) {  
        super(nume, prenume);  
        this.materia=new String(materia);  
    }  
    void afis() {System.out.println(nume+" "+prenume+" "+materia);}  
}  
public class test {  
    public static void main(String[] args) {  
        olimpic a=new olimpic("Pascu", "Marius", "informatica");  
        String n=new String("Grosu"); String p=new String("Bogdan"); String mat=new  
        String("matematica");  
        olimpic b=new olimpic(n,p,mat); a.afis(); b.afis();  
    }  
}
```

Moștenirea și acces la membrii clasei

- Moștenirea claselor nu neagă restricțiile impuse de modificatori;

```
// Закрытые члены класса не наследуются

// Этот код не пройдет компиляцию

// Класс, описывающий двумерные объекты
class TwoDShape {
    private double width; // теперь эти переменные
    private double height; // объявлены как закрытые

    void showDim() {
        System.out.println("Ширина и высота - " + width + " и " +
                           height);
    }
}

// Подкласс для представления треугольников,
// производный от класса TwoDShape
class Triangle extends TwoDShape {
    String style;

    double area() {
        return width * height / 2; // Ошибка: доступ запрещен!
    }

    void showStyle() {
        System.out.println("Треугольник " + style);
    }
}
```

Доступ к членам суперкласса, объявленным как закрытые, невозможен

```
// Использование методов доступа для установки и
// получения значений закрытых членов.

// Класс, описывающий двумерные объекты
class TwoDShape {
    private double width; // теперь эти переменные
    private double height; // объявлены как закрытые

    // Методы доступа к закрытым переменным экземпляра width и height
    double getWidth() { return width; }
    double getHeight() { return height; } ← Методы доступа к переменным
    void setWidth(double w) { width = w; }
    void setHeight(double h) { height = h; }

    void showDim() {
        System.out.println("Ширина и высота - " + width + " и " +
                           height);
    }
}

// Подкласс для представления треугольников,
// производный от класса TwoDShape
class Triangle extends TwoDShape {
    String style;
    ↓
    Использование методов доступа,
    предоставляемых суперклассом

    double area() {
        return getWidth() * getHeight() / 2;
    }

    void showStyle() {
        System.out.println("Треугольник " + style);
    }
}
```

În ce situație este necesar de a ascunde o variabilă a clasei?

- Nu există reguli strict formulate!
- Există totuși 2 principii:
 - Dacă variabila este folosită doar de metodele clasei, atunci ea trebuie ascunsă în clasa dată;
 - Dacă valoarea variabilei trebuie nu să fie utilizată în afara clasei atunci la fel variabila trebuie să fie ascunsă.

Constructori și moștenire

- Există posibilitate ca superclasele și subclasele să aibă constructori proprii.
- Care este acel constructor care are grijă de subclasă? – cel al *superclasei* sau al *subclasei*?
- Constructorul superclasei construiește partea moștenită, constructorul subclasei se referă la partea proprie a subclasei.

```
// Добавление конструктора в класс Triangle

// Класс, описывающий двумерные объекты
class TwoDShape {
    private double width; // теперь эти переменные
    private double height; // объявлены как закрытые

    // Методы доступа к переменным экземпляра width и height
    double getWidth() { return width; }
    double getHeight() { return height; }
    void setWidth(double w) { width = w; }
    void setHeight(double h) { height = h; }

    void showDim() {
        System.out.println("Ширина и высота - " + width + " и " +
                           height);
    }
}

// Подкласс для представления треугольников,
// производный от класса TwoDShape
class Triangle extends TwoDShape {
    private String style;

    // Конструктор
    Triangle(String s, double w, double h) {
        setWidth(w);
        setHeight(h); ←———— Инициализация части объекта,
        style = s;           соответствующей классу TwoDShape
    }

    double area() {
        return getWidth() * getHeight() / 2;
    }

    void showStyle() {
        System.out.println("Треугольник " + style);
    }
}
```

Utilizarea cuvântului **super**

- Există 2 situații în care se folosește cuvântul **super**:
 - *Apelul constructorului superclasei*
 - *Acces la membrii superclasei, ascunși de membrii subclasei.*

Exemplu super – apel al constructorului superclasei.

```
// Добавление конструкторов в класс TwoDShape
class TwoDShape {
    private double width;
    private double height;

    // Параметризованный конструктор
    TwoDShape(double w, double h) { ← Конструктор класса TwoDShape
        width = w;
        height = h;
    }

    // Методы доступа к переменным экземпляра width и height
    double getWidth() { return width; }
    double getHeight() { return height; }
    void setWidth(double w) { width = w; }
    void setHeight(double h) { height = h; }

    void showDim() {
        System.out.println("Ширина и высота - " + width + " и " +
                           height);
    }
}

// Подкласс для представления треугольников,
// производный от класса TwoDShape
class Triangle extends TwoDShape {
    private String style;

    Triangle(String s, double w, double h) {
        super(w, h); // вызов конструктора суперкласса
        ↑
        style = s;
    }                                | Использование оператора super() для вызова
                                    | конструктора класса TwoDShape

    double area() {
        return getWidth() * getHeight() / 2;
    }

    void showStyle() {
        System.out.println("Треугольник " + style);
    }
}
```

```
// Добавление дополнительных конструкторов в класс TwoDShape
class TwoDShape {
    private double width;
    private double height;

    // Конструктор, заданный по умолчанию
    TwoDShape() {
        width = height = 0.0;
    }

    // Параметризованный конструктор
    TwoDShape(double w, double h) {
        width = w;
        height = h;
    }

    // Создание объекта с одинаковыми значениями
    // переменных экземпляра width и height
    TwoDShape(double x) {
        width = height = x;
    }

    // Методы доступа к переменным экземпляра width и height
    double getWidth() { return width; }
    double getHeight() { return height; }
    void setWidth(double w) { width = w; }
    void setHeight(double h) { height = h; }

    void showDim() {
        System.out.println("Ширина и высота - " + width + " и " +
                           height);
    }
}
```

```
// Подкласс для представления треугольников,
// производный от класса TwoDShape
class Triangle extends TwoDShape {
    private String style;

    // Конструктор по умолчанию
    Triangle() {
        super(); // вызов конструктора суперкласса по умолчанию
        style = "none";
    }

    // Конструктор
    Triangle(String s, double w, double h) {
        super(w, h); // вызов конструктора суперкласса с
                     // двумя аргументами

        style = s;
    }

    // Конструктор с одним аргументом
    Triangle(double x) {
        super(x); // вызов конструктора суперкласса
                  // с одним аргументом

        style = "закрашенный";
    }

    double area() {
        return getWidth() * getHeight() / 2;
    }

    void showStyle() {
        System.out.println("Треугольник " + style);
    }
}
```

Использование метода super ()
для вызова разных форм
конструктора TwoDShape ()

```
// Использование ключевого слова super
// для предотвращения скрытия имен
class A {
    int i;
}

// Создание подкласса, расширяющего класс A
class B extends A {
    int i; // эта переменная i скрывает переменную i из класса A

    B(int a, int b) {
        super.i = a; // переменная i из класса A ←———— Здесь super.i ссылается
        i = b;          // переменная i из класса B на переменную i из класса A
    }

    void show() {
        System.out.println("i в суперклассе: " + super.i);
        System.out.println("i в подклассе: " + i);
    }
}

class UseSuper {
    public static void main(String args[]) {
        B subOb = new B(1, 2);

        subOb.show();
    }
}
```

Выполнение этой программы даст следующий результат.

```
i в суперклассе: 1
i в подклассе: 2
```

Crearea ierarhiilor de clase

- Structura ierarhică se potrivește pentru situații în care o subclasă este o superclasă pentru alte subclase.
- Dacă clasa A este superclasă pentru B, și B este superclasă pentru C, atunci C poate moșteni totul ce este în B și A.

```
// Многоуровневая иерархия
class TwoDShape {
    private double width;
    private double height;

    // Конструктор по умолчанию
    TwoDShape() {
        width = height = 0.0;
    }

    // Параметризованный конструктор
    TwoDShape(double w, double h) {
        width = w;
        height = h;
    }

    // Создание объекта с одинаковыми значениями
    // переменных экземпляра width и height
    TwoDShape(double x) {
        width = height = x;
    }

    // Методы доступа к переменным экземпляра width и height
    double getWidth() { return width; }
    double getHeight() { return height; }
    void setWidth(double w) { width = w; }
    void setHeight(double h) { height = h; }

    void showDim() {
        System.out.println("Ширина и высота - " + width + " и " +
                           height);
    }
}
```

```
// Расширение класса TwoDShape
class Triangle extends TwoDShape {
    private String style;

    // Конструктор по умолчанию
    Triangle() {
        super();
        style = "none";
    }

    Triangle(String s, double w, double h) {
        super(w, h); // вызов конструктора суперкласса

        style = s;
    }

    // Конструктор с одним аргументом для построения треугольника
    Triangle(double x) {
        super(x); // вызов конструктора суперкласса

        style = "закрашенный";
    }

    double area() {
        return getWidth() * getHeight() / 2;
    }

    void showStyle() {
        System.out.println("Треугольник " + style);
    }
}
```

```
// Расширение класса Triangle
class ColorTriangle extends Triangle {
    private String color;
    ColorTriangle(String c, String s, double w, double h) {
        super(s, w, h);
        color = c;
    }
    String getColor() { return color; }
    void showColor() {
        System.out.println("Цвет - " + color);
    }
}
```

Класс ColorTriangle наследует класс Triangle, производный от класса TwoDShape, и поэтому включает все члены классов Triangle и TwoDShape

Ordinea în care sunt apelați constructorii.

- Constructorii sunt apelați în ordinea moștenirii începând cu superclase și terminând cu subclase.
- Metoda super() trebuie să fie primul în constructorul subclasei.
- Dacă metoda super() lipsește atunci constructorul superclasei se apelează automat.

```
// Демонстрация очередности вызова конструкторов

// Создание суперкласса
class A {
    A() {
        System.out.println("Конструктор A");
    }
}

// Создание подкласса в результате расширения класса A
class B extends A {
    B() {
        System.out.println("Конструктор B");
    }
}

// Создание подкласса в результате расширения класса B
class C extends B {
    C() {
        System.out.println("Конструктор C");
    }
}

class OrderOfConstruction {
    public static void main(String args[]) {
        C c = new C();
    }
}
```

Результат выполнения данной программы выглядит следующим образом.

Конструктор А
Конструктор В
Конструктор С

Referințe

```
// Этот код не пройдет компиляцию
class X {
    int a;

    X(int i) { a = i; }
}

class Y {
    int a;

    Y(int i) { a = i; }
}

class IncompatibleRef {
    public static void main(String args[]) {
        X x = new X(10);
        X x2;
        Y y = new Y(5);

        x2 = x; // допустимо, поскольку обе переменные одного типа

        x2 = y; // ошибка, поскольку переменные разных типов
    }
}
```

```

// Обращение к объекту подкласса по ссылочной
// переменной суперкласса
class X {
    int a;

    X(int i) { a = i; }
}

class Y extends X {
    int b;

    Y(int i, int j) {
        super(j);
        b = i;
    }
}

class SupSubRef {
    public static void main(String args[]) {
        X x = new X(10);
        X x2;
        Y y = new Y(5, 6);

        x2 = x; // допустимо, поскольку обе переменные одного типа
        System.out.println("x2.a: " + x2.a);      Класс Y является подклассом X, поэтому
                                                    переменные x2 и у могут ссылаться на
                                                    один и тот же объект производного класса
        ↓
        x2 = y; // по-прежнему допустимо по указанной выше причине
        System.out.println("x2.a: " + x2.a);

        // В классе X известны только члены класса X
        x2.a = 19; // допустимо
        // x2.b = 27; // ошибка, так как переменная b не
                      // является членом класса X
    }
}

```

```

// Переопределение метода
class A {
    int i, j;

    A(int a, int b) {
        i = a;
        j = b;
    }

    // Отображение переменных i и j
    void show() {
        System.out.println("i и j: " + i + " " + j);
    }
}

class B extends A {
    int k;

    B(int a, int b, int c) {
        super(a, b);
        k = c;
    }

    // Отображение переменной k - переопределение метода show() в A
    void show() { ← Метод show() в B переопределяет
        System.out.println("k: " + k);
    }
}

class Override {
    public static void main(String args[]) {
        B subOb = new B(1, 2, 3);

        subOb.show(); // вызов метода show() из класса B
    }
}

```

Suprascriere

```
class B extends A {  
    int k;  
  
    B(int a, int b, int c) {  
        super(a, b);  
        k = c;  
    }  
  
    void show() {  
        super.show();  
        System.out.println("k: " + k);  
    }  
}
```

С помощью ключевого слова **super** вызывается версия метода **show()**, определенная в суперклассе **A**

Suprăîncărcarea

- Suprăîncărcarea este o modalitate de implementarea polimorfismului în Java.
- Suprăîncărcarea este posibilitatea de a denumi mai multe metode cu același nume, dacă diferă:
 - *Tipul parametrilor*
 - *Numărul parametrilor*
 - *Rezultatul returnat*

Exemplu 1 (Clasa OverloadDemo)

```
class OverloadDemo  {
    void test() {
        System.out.println("Parametrii lipsesc");
    }
    void test(int a) {
        System.out.println("a: " +a);
    }
    void test(int a, int b) {
        System.out.println("a și b " +a + " " + b);
    }
    double test(double a) {
        System.out.println("double a: " +a);
        return a*a;
    }
}
```

Exemplu 1 (Clasa Overload)

```
class Overload {  
    public static void main(String args[]) {  
        OverloadDemo ob=new OverloadDemo();  
        double result;  
        ob.test();  
        ob.test(10);  
        ob.test(10,30);  
        result=ob.test(123.25);  
        System.out.println("Rezultatul apelului ");  
        System.out.println("ob.test(123.25) este");  
        System.out.println(result);  
    }  
}
```

Exemplu 1 (Rezultatul executiei)

Parametrii lipsesc

a: 10

a și b: 10 20

double a: 123.25

Rezultatul apelului

ob.test(123.25) este

15190.5625

Exemplu 2 – Supraîncărcarea constructorilor

```
class Box {  
    double width;  
    double height;  
    double depth;  
    Box (double w, double h, double d) {  
        width=w; height=h; depth=d;  
    }  
    Box () { width=-1; height=-1; depth=-1}  
    Box (double len) { width=height=depth=len}  
    double volume() {  
        return width*height*depth;  
    }  
}
```

Programarea orientată pe obiect 2

Prelegherea nr. 5

Clase incluse. Ascunderea și
încapsularea datelor. Clase abstracte.
Interfețe.

Petic Mircea

petic.mircea@yahoo.com

www.facebook.com/mircea.petic

Clase interioare

- În Java există posibilitatea ca o clasă să includă una sau mai multe clase.
- Ca și date membru sau metodele unei clase, clasele interne pot fi:
 - Statice
 - Nestatice

Clase interioare - nestatiche

- Se includ în structura de clasă.
- Pentru a fi activate, clasa mamă va conține o referință către un obiect al clasei interne.
- Prin intermediul acestei referințe se pot accesa datele membru și metodele clasei interne.

Clase interioare – nestatice. Exemplu

```
class A {  
    I ref=new I();  
    int a=1;  
    void M1() {System.out.println("a="+a);}  
    class I{  
        int b=2;  
        void M2() {System.out.println("Metoda M2");}  
    }  
}  
public class C{  
    public static void main(String[] args){  
        A x=new A();  
        x.M1();  
        x.ref.M2();  
        System.out.println(x.ref.b);  
    }  
}
```

Clase interioare - statice

```
class A {  
    int a=1;  
    void M1() {System.out.println("a="+a);}  
    static class I{  
        static int b=2;  
        static void M2() {System.out.println("Metoda M2");}  
    }  
}  
public class C{  
    public static void main(String[] args){  
        A x=new A();  
        x.M1();  
        x.I.M2();  
        System.out.println(x.I.b);  
    }  
}
```

Clase interioare. Observații

- Dacă o clasă conține o clasă interioară, atunci după compilare se obțin două clase (două fișiere cu extensia .class – *A.class* și *A\$I.class*).
- Dintr-o clasă interioară se pot accesa datele membru ale clasei care o conține.
- Pentru clasele interioare nestatice, datele membru și metodele claselor interioare se pot accesa și fără a utiliza acea referință către clasa interioară.

```
A x=new A();
```

```
A.I y=x.new I();  
y.M2();
```

Observații

- Clasa interioară nu poate exista fără clasa în care se conține.
- Domeniul de vizibilitate al clasei este limitată de clasa sa externă.
- Dacă clasa internă este definită în limitele domeniului de vizibilitate a clasei el devine membru clasei din care face parte.

```

// Применение внутреннего класса
class Outer {
    int nums[];

    Outer(int n[]) {
        nums = n;
    }

    void Analyze() {
        Inner inOb = new Inner();

        System.out.println("Минимум: " + inOb.min());
        System.out.println("Максимум: " + inOb.max());
        System.out.println("Среднее: " + inOb.avg());
    }
}

// Внутренний класс
class Inner { ← Внутренний класс
    int min() {
        int m = nums[0];

        for(int i=1; i < nums.length; i++)
            if(nums[i] < m) m = nums[i];
        return m;
    }

    int max() {
        int m = nums[0];
        for(int i=1; i < nums.length; i++)
            if(nums[i] > m) m = nums[i];

        return m;
    }

    int avg() {
        int a = 0;
        for(int i=0; i < nums.length; i++)
            a += nums[i];
        return a / nums.length;
    }
}

class NestedClassDemo {
    public static void main(String args[]) {
        int x[] = { 3, 2, 1, 5, 6, 9, 7, 8 };
        Outer outOb = new Outer(x);

        outOb.Analyze();
    }
}

```

Минимум: 1
 Максимум: 9
 Среднее: 5

```

// Применение класса ShowBits в качестве локального
class LocalClassDemo {
    public static void main(String args[]) {

        // Внутренняя версия класса ShowBits
        class ShowBits { ← Локальный класс, вложенный в метод
            int numbits;

            ShowBits(int n) {
                numbits = n;
            }

            void show(long val) {
                long mask = 1;

                // Сдвиг влево для установки единицы в нужной позиции
                mask <<= numbits-1;

                int spacer = 0;
                for(; mask != 0; mask >>>= 1) {
                    if((val & mask) != 0) System.out.print("1");
                    else System.out.print("0");

                    spacer++;
                    if((spacer % 8) == 0) {
                        System.out.print(" ");
                        spacer = 0;
                    }
                }
                System.out.println();
            }
        }
    }

    for(byte b = 0; b < 10; b++) {
        ShowBits byteval = new ShowBits(8);

        System.out.print(b + " в двоичном представлении: ");
        byteval.show(b);
    }
}
}

```

Выполнение этой программы дает следующий результат.

0 в двоичном представлении:	00000000
1 в двоичном представлении:	00000001
2 в двоичном представлении:	00000010
3 в двоичном представлении:	00000011
4 в двоичном представлении:	00000100
5 в двоичном представлении:	00000101
6 в двоичном представлении:	00000110
7 в двоичном представлении:	00000111
8 в двоичном представлении:	00001000
9 в двоичном представлении:	00001001

Lista argumentelor de lungime variabilă

```
// Демонстрация использования метода
// с переменным числом аргументов
class VarArgs {

    // Метод vaTest() допускает переменное число аргументов
    static void vaTest(int ... v) {
        System.out.println("Количество аргументов: " + v.length);
        System.out.println("Содержимое: ");

        for(int i=0; i < v.length; i++)
            System.out.println(" arg " + i + ": " + v[i]);

        System.out.println();
    }

    public static void main(String args[])
    {
        // Метод vaTest() может вызываться с
        // переменным числом аргументов
        vaTest(10);           // 1 аргумент
        vaTest(1, 2, 3);      // 3 аргумента
        vaTest();              // без аргументов
    }
}
```

Вызовы метода с указанием различного числа аргументов

```
// Использование массива аргументов переменной длины
// наряду с обычными аргументами
class VarArgs2 {

    // Здесь msg - обычный параметр,
    // а v - массив параметров переменной длины
    static void vaTest(String msg, int ... v) { ← "Обычный" параметр
        System.out.println(msg + v.length);           и параметр в виде массива
        System.out.println("Содержимое: ");           переменной длины

        for(int i=0; i < v.length; i++)
            System.out.println(" arg " + i + ": " + v[i]);

        System.out.println();
    }

    public static void main(String args[])
    {
        vaTest("Один аргумент в массиве: ", 10);
        vaTest("Три аргумента в массиве: ", 1, 2, 3);
        vaTest("Отсутствуют аргументы в виде массива: ");
    }
}
```

Clase abstracte

- O clasă abstractă conține date membru (nu obligatoriu) și mai multe metode, dar unele pot fi, la rîndul lor abstracte.
- Metodă abstractă, în clasă se trece doar prin antetul ei.
- Metodele abstracte urmează să fie redefinite în subclase.
- Clasa abstractă este precedată de cuvîntul cheie abstract, ca și metodele abstracte.
- Clasele abstracte se utilizează pentru a fi extinse.

Exemplu – clasă abstractă

```
abstract class Ex {  
    abstract void afis1();  
    abstract void afis2();  
    void decid(int b) { if (b>0) afis1(); else afis2(); }  
}  
class Ex1 extends Ex  
{  
    void afis1() { System.out.println("Afis 1"); }  
    void afis2() { System.out.println("Afis 2"); }  
}  
public class test {  
    public static void main(String[] args) {  
        Ex1 a=new Ex1();  
        a.decid(1);  
        a.decid(-1);  
    }  
}
```

Методы *finale*

```
class A {  
    final void meth() {  
        System.out.println("Это метод final.");  
    }  
}  
  
class B extends A {  
    void meth() { // Ошибка: этот метод не может быть переопределен!  
        System.out.println("Недопустимо!");  
    }  
}
```

Clase finale

```
final class A {  
    // ...  
}
```

```
// Следующее определение класса недопустимо  
class B extends A { // Ошибка: класс A не может иметь подклассов!  
    // ...  
}
```

- Imposibilă moștenirea!
- Nu se folosesc final împreună cu abstract

Constante finale

- Cuvîntul final marchează declararea unei constante
- Denumirile constantelor de regulă cu litere mari.

```
final int OUTERR = 0;  
final int INERR = 1; ← Объявление констант final  
final int DISKERR = 2;  
final int INDEXERR = 3;
```

Clasa Object

- Clasa Object reprezintă implicit superclasa tuturor claselor în Java.
- Variabila de referință de tip Object poate să se refere la obiect al oricărei clase.
- Variabila de referință de tip Object poate să se refere la variabilele de tip tablou aşa cum acestea sunt organizate în formă de clasă.

Metodele clasei Object

Метод	Назначение
Object clone()	Создает новый объект, аналогичный клонируемому объекту
boolean equals(Object объект)	Определяет равнозначность объектов
void finalize()	Вызывается перед тем, как неиспользуемый объект будет удален сборщиком мусора (не рекомендуется в JDK 9)
Class<?> getClass()	Определяет класс объекта во время выполнения
int hashCode()	Возвращает хеш-код, связанный с вызывающим объектом
void notify()	Возобновляет работу потока, ожидающего уведомления от вызывающего объекта
void notifyAll()	Возобновляет работу всех потоков, ожидающих уведомления от вызывающего объекта
String toString()	Возвращает символьную строку, описывающую объект
void wait()	Ожидает выполнения другого потока
void wait(long миллисекунды)	
void wait(long миллисекунды, int наносекунды)	

Interfețe

- Prin interfață vom înțelege un proiect de clasă.
- O interfață poate conține doar constante și antete de metode.
- Interfețele vor fi implementate de una sau mai multe clase.
- O interfață arată ca o clasă, numai că, în locul cuvântului `class`, se folosește cuvântul `interface`
- Atunci când scriem o clasă care implementează o anumită interfață – `implementation`.

Exemplu – interfață

```
interface InterfA {  
    int i=5;  
    void afis();  
}  
class A implements InterfA {  
    int i=6;  
    public void afis() {  
        System.out.println("Afis din clasa A="+i+" "+InterfA.i);}  
}  
class B implements InterfA {  
    int i=7;  
    public void afis() {  
        System.out.println("Afis din clasa A="+i+" "+InterfA.i);}  
}  
  
public class test {  
    public static void main(String[] args) {  
        A x=new A(); x.afis();  
        B y=new B(); y.afis();  
    }  
}
```

O interfață poate extinde mai multe interfețe

interface A

```
{ void metA();}
```

interface B

```
{ void metB();}
```

interface C

```
{ void metC();}
```

interface extinsa extinde A, B, C

```
{ void metE(); }
```

A implementa o interfata extinsa inseamna a implementa toate metodele interfetei care au stat la baza extinderii plus metodele a caror antet se gaseste in interfata extinsa.

```
class ex implements extinsa {  
    public void metA() { System.out.println("metA"); }  
    public void metB() { System.out.println("metB"); }  
    public void metC() { System.out.println("metC"); }  
    public void metD() { System.out.println("metD"); }  
}  
public class test {  
    public static void main(String[] args)  
    {  
        ex x=new ex();  
        x.metA(); x.metB(); x.metC(); x.metD();  
    }  
}
```

O clasa poate implementa oricite interfete.

```
interface A { void metA(); }
interface B { void metB(); }
interface C { void metC(); }
class ex implements A, B, C {
    public void metA() { System.out.println("metA"); }
    public void metB() { System.out.println("metB"); }
    public void metC() { System.out.println("metC"); }
}
public class test {
    public static void main(String[] args){
        ex x=new ex();
        x.metA(); x.metB(); x.metC();
    }
}
```

Cu ajutorul interfetelor se simuleaza mostenirea multipla

```
interface A { void metA();}  
class Aclass implements A { public void metA() { System.out.println("metA");} }  
interface B { void metB();}  
class Bclass implements B { public void metB() { System.out.println("metB");} }  
interface C { void metC();}  
class Cclass implements C { public void metC() { System.out.println("metC");} }  
clasa mosten implements A, B, C {  
    public void metA() { A x=new Aclass(); x.metA; }  
    public void metB() { B x=new Bclass(); x.metB; }  
    public void metC() { C x=new Cclass(); x.metC; }  
}  
public class test {  
    public static void main (String[] args){  
        mosten x=new mosten();  
        x.metA(); x.metB(); x.metC();  
    }  
}
```

Se pot mosteni metode ale claselor care implementeaza o aceeasi interfata.

```
interface interf { void met();}  
class A implements interf  
{ public void met() { System.out.println("metoda din A");} }  
class B implements interf  
{ public void met() { System.out.println("metoda din B");} }  
class C implements interf  
{ public void met() { System.out.println("metoda din C");} }  
clasa mosten {  
    interf x;  
    mosten (interf x) { this.x=x;}  
    void met() {x.met();}  
}  
public class test {  
    public static void main (String[] args) {  
        mosten v1=new mosten (new A()); v1.met();  
        mosten v2=new mosten (new B()); v2.met();  
        mosten v3=new mosten (new C()); v3.met();  
    }  
}
```

Observații

- Prin susținerea interfețelor în Java este implementată ideea polimorfismului: "O interfață – mai multe metode!"
- Până în versiunea JDK8 în interfețe nu era posibilă implementarea implicită.
- Începînd cu JDK9 e posibilă utilizarea metodelor înclose.

Referințe la variabile ale interfeței

```
// Использование интерфейсных ссылок
class ByTwos implements Series {
    int start;
    int val;

    ByTwos() {
        start = 0;
        val = 0;
    }

    public int getNext() {
        val += 2;
        return val;
    }

    public void reset() {
        start = 0;
        val = 0;
    }

    public void setStart(int x) {
        start = x;
        val = x;
    }
}
```

```
class ByThrees implements Series {
    int start;
    int val;

    ByThrees() {
        start = 0;
        val = 0;
    }

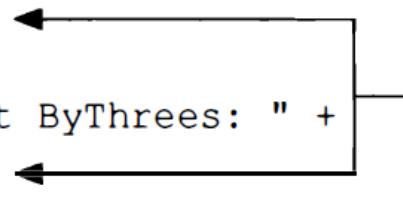
    public int getNext() {
        val += 3;
        return val;
    }

    public void reset() {
        start = 0;
        val = 0;
    }

    public void setStart(int x) {
        start = x;
        val = x;
    }
}
```

Referințe la variabile ale interfeței

```
class SeriesDemo2 {  
    public static void main(String args[]) {  
        ByTwos twoOb = new ByTwos();  
        ByThrees threeOb = new ByThrees();  
        Series ob;  
  
        for(int i=0; i < 5; i++) {  
            ob = twoOb;  
            System.out.println("Следующее значение ByTwos: " +  
                               ob.getNext());  
            ob = threeOb;  
            System.out.println("Следующее значение Next ByThrees: " +  
                               ob.getNext());  
        }  
    }  
}
```



Доступ к объекту с помощью
интерфейсной ссылки

Constante în interfețe

```
// Интерфейс, содержащий только константы
interface IConst {
    int MIN = 0;
    int MAX = 10;
    String ERRORMSG = "Ошибка диапазона";
}

class IConstD implements IConst {
    public static void main(String args[]) {
        int nums[] = new int[MAX];

        for(int i=MIN; i < 11; i++) {
            if(i >= MAX) System.out.println(ERRORMSG);
            else {
                nums[i] = i;
                System.out.print(nums[i] + " ");
            }
        }
    }
}
```



Это константы

Metodele folosite implicit

```
public interface MyIF {  
    // Объявление обычного метода интерфейса, которое НЕ включает  
    // определение реализации по умолчанию  
    int getUserId();  
  
    // Объявление метода по умолчанию, включающее его реализацию  
    default int getAdminID() {  
        return 1;  
    }  
}  
  
// Реализация интерфейса MyIF  
class MyIFImp implements MyIF {  
    // Реализации подлежит лишь метод getUserId() интерфейса MyIF.  
    // Делать это для метода getAdminID() необязательно, поскольку  
    // при необходимости может быть использована его реализация,  
    // заданная по умолчанию.  
  
    public int getUserId() {  
        return 100;  
    }  
}
```

Folosirea clasei

```
// Использование интерфейсного метода по умолчанию
class DefaultMethodDemo {
    public static void main(String args[]) {

        MyIFIImpl obj = new MyIFIImpl();

        // Вызов метода getUserId() возможен, поскольку он явно
        // реализован классом MyIFIImpl
        System.out.println("Идентификатор пользователя " +
                           obj.getUserId());

        // Вызов метода getAdminID() также возможен, поскольку
        // предоставляется его реализация по умолчанию
        System.out.println("Идентификатор администратора: " +
                           obj.getAdminID());
    }
}
```

В результате выполнения программы будет получен следующий результат.

```
Идентификатор пользователя: 100
Идентификатор администратора: 1
```

Modificatori date membru

- **public** – data membru poate fi accesată de oriunde este accesibilă și poate fi moștenită.
- **protected** – data membru poate fi accesată de codul aflat în același pachet și poate fi moștenită.
- **private** – data membru poate fi accesată numai din clasa în care se află și nu este moștenită.
- **final** – data membru este considerată constantă, deci nu poate fi modificată.
- **static** – data membru nu este memorată de fiecare obiect al clasei respective.

Încapsularea

- **Încapsularea** (engleza: *encapsulation*) este proprietatea obiectelor de a-si ascunde o parte din date si metode.
- Din exteriorul obiectului sunt accesibile ("vizibile") numai datele si metodele *publice*.
- Datorită acestei proprietăți este posibilă modificare logicii clasei, fară a schimba logica altor clase.
- Accesul la datele membru ale claselor se realizează de metodele (*accessors*) pentru citire (**getters**) și de scriere (**setters**).

Exemplu 1

```
public class Human {  
    private int age;  
    public int getAge() {  
        return age;  
    }  
    public void setAge(int a) {  
        age=a;  
    }  
}
```

Exemplu 2

```
public class Human {  
    private double age;  
  
    public int getAge() {  
        return (int)Math.round(age);  
    }  
    public void setAge(int a) {  
        age=a;  
    }  
    public double getExactAge() {  
        return age;  
    }  
    public void setExactAge(double a) {  
        age=a;  
    }  
}
```

Exemplu 3

- Getter-ii asigura asignarea valorilor corecte pentru datelor membru ale clasei.

```
public void setAge(int a)
{
    if (a>=0) { age=a; }
}
```

- Age nu va primi niciodată o valoare negativă.

Variabile shadow

- **Shadowing** se produce atunci cand se defineste o metoda cu parametri de intrare care au nume identice cu variabile statice sau variabile de instanta (atribute).
- este importanta utilizarea referintei **this**, pentru a indica care variabila este atributul.

Variabile shadow

```
public class A{
```

```
    int x;
```

```
    public A() { /*Constructor*/ }
```

```
}
```

```
public class B extends A {
```

```
    int x;
```

```
    public B() { /*Constructor*/ }
```

```
}
```

```
x / this.x / super.x / (A this.x) / super.super.x
```

Exemplu 4

```
public class Human {  
    private double age;  
  
    public int getAge() {  
        return (int)Math.round(age);  
    }  
    public void setAge(int age) {  
        this.age=age;  
    }  
    public double getExactAge() {  
        return age;  
    }  
    public void setExactAge(double age) {  
        this.age=age;  
    }  
}
```

Exemplu de acces la valorile datelor membru private.

```
public class test {  
    public static void main(String[] args) {  
        Human x=new Human();  
        x.setExactAge(31);  
        System.out.println("Vîrstă este "+x.getExactAge());  
    }  
}
```

Obscuring

- Poate apărea cînd un nume simplu poate fi în același timp ca variabilă, tip sau nume de pachet.

```
import java.awt.*;  
public class Obscuring {  
    static Point Test = new Point(3,2);  
    public static void main (String s[]) {  
        print(Test.x);  
    }  
}  
class Test { static int x = -5; }
```

Explicație

- numele Test este în același timp data membru pentru clasa Obscuring și nume unui tip care se află în același pachet Test.
- Cu ajutorul acestui nume are loc accesul la câmpul x, care este definit și în clasa java.awt.Point și Test.
- Rezultatul exemplului va fi 3, variabila are o prioritate mai mare.
- În același timp tipul are o prioritate mai mare ca nume de pachet.

- Un suport software care trebuie sa fie instalat pe calculatoarele care rulează fișierele .class se numește _____.

- Un suport software care trebuie să fie instalat pe calculatoarele care rulează fișierele .class se numește **Java Virtual Machine**.

- Tipurile de date primitive în Java sănt
-

- Tipurile de date primitive în Java sănt **byte**, **short**, **int**, **long**, **float**, **double**.

Tipuri de date primitive

Tip	Octeți ocupăți	Numere întregi din intervalul
byte	1	[-128, 127]
short	2	[-32768, 32768]
int	4	[-2147483648, +2147483647]
long	8	[-9223372036854775808, 9223372036854775807]

Tip	Octeți ocupăți	Modulul valorilor între
float	4	[3.4×10^{-38} , 3.4×10^{38}]
double	8	[1.7×10^{-308} , 1.7×10^{308}]

```
char a='m', b='\u006d', c=109;  
System.out.println(a); System.out.println(b);  
System.out.println(c);
```

- Dacă dorim să utilizăm unele metode din una sau mai multe clase atunci se vor elabora

- Dacă dorim să utilizăm unele metode din una sau mai multe clase atunci se vor elabora **pachete**.

- Dacă o clasă conține date membru și mai multe metode unele din ele au doar antet atunci aşa clasă se numește _____.

- Dacă o clasă conține date membru și mai multe metode unele din ele au doar antet atunci aşa clasă se numește **clasă abstractă**.

- Structura care grupează datele și unitățile de prelucrare a acestora într-un modul, unindu-le astfel într-o entitate naturală se numește
-

- Structura care grupează datele și unitățile de prelucrare a acestora într-un modul, unindu-le astfel într-o entitate naturală se numește **clasă**.

- `Math.random()` returnează o valoare aleatoare
în intervalul _____.

- `Math.random()` returnează o valoare aleatoare în intervalul **[0, 1)**.

- Metoda specială a clasei respective care are rolul de a aloca în memorie spațiul necesar obiectului dar și se a inițializa datele membru ale acestuia se numește _____.

- Metoda specială a clasei respective care are rolul de a aloca în memorie spațiul necesar obiectului dar și se a inițializa datele membru ale acestuia se numește **constructor**.

- O clasă care conține doar constante și antete de metode se numește _____.

- O clasă care conține doar constante și antete de metode se numește **interfata**.

- Numărul parametrilor formali trebuie să coincidă cu _____.

- Numărul parametrilor formali trebuie să coincidă cu **numarul parametrilor actuali/efectivi**.

- Instrucțiunea break se folosește pentru instrucțiunile _____.

- Instrucțiunea break se folosește pentru instrucțiunile **for, while, do while și switch**.

¹

Clasa care este extinsă se numește...

a. *constructor*

b. *clasă*

c. *metodă*

d. *obiect*

¹

Clasa care este extinsă se numește...

- a. *constructor*
- b. *clasă*
- c. *metodă*
- d. *obiect*

2 Fie dată următoarea definiție de clasă

```
class complex {int a; int b; void afis() {  
System.out.println(a+" "+b);}}}
```

Cum trebuie numit fișierul în care vom scrie această clasă?

- a.** class
- b.** class.java
- c.** complex
- d.** complex.java

2 Fie dată următoarea definiție de clasă

```
class complex {int a; int b; void afis() {  
System.out.println(a+" "+b);}}}
```

Cum trebuie numit fișierul în care vom scrie această clasă?

- a. class**
- b. class.java**
- c. complex**
- d. complex.java**

3 Un constructor al clasei este apelat de operatorul ...

- a.** this
- b.** new
- c.** class
- d.** super

3 Un constructor al clasei este apelat de operatorul ...

- a. this
- b. new**
- c. class
- d. super

4 Clasa care este extinsă se numește...

- a. *supraclasa*
- b. *superclasa*
- c. *subclasa*
- d. *clasa abstractă*

4 Clasa care este extinsă se numește...

- a. *supraclasa*
- b. ***superclasa***
- c. *subclasa*
- d. *clasa abstractă*

- 5 Posibilitatea ca atât superclasa cât și subclasa să aibă metode cu același nume se numește ...
- a. supraîncărcare b. moștenire c. polimorfism d. recursie

- 5 Posibilitatea ca atât superclasa cât și subclasa să aibă metode cu același nume se numește ...
- a. supraîncărcare b. moștenire **c. polimorfism** d. recursie

6 Cuvîntul cheie care înseamnă referință către obiectul current este ...

- a.** this **a.** new **a.** class **a.** super

6 Cuvîntul cheie care înseamnă referință către obiectul current este ...

- a. this**
- a. new**
- a. class**
- a. super**

7 O clasă care conține doar constante și antete de metode se numește?

- a. *interfață*
- b. *clasă abstractă*
- c. *superclasă*
- d. *obiect*

7 O clasă care conține doar constante și antete de metode se numește?

- a. *interfață*
- b. *clasă abstractă*
- c. *superclasă*
- d. *obiect*

8 Care din următoarele comentarii sănt greșite?

- a. `String s = "text/*just text*/";`
- b. `circle.get/*comment*/Radius();`
- c. `circle./*comment*/getRadius();`
- d. `int/*comment*/x=1;`

8 Care din următoarele comentarii sănt greșite?

- a. `String s = "text/*just text*/";`
- b. `circle.get/*comment*/Radius();`
- c. `circle./*comment*/getRadius();`
- d. `int/*comment*/x=1;`

9 De care modificatori nu există?

- a. *date membru*
- b. *metode*
- c. *clase*
- d. *obiecte*

9 De care modificatori nu există?

- a. *date membru*
- b. *metode*
- c. *clase*
- d. *obiecte*

10 Rezultatul compilării este fișierul cu extensia ...

- a. *.java
- b. *.javac
- c. *.class
- d. *.exe

10 Rezultatul compilării este fișierul cu extensia ...

- a. *.java
- b. *.javac
- c. *.class
- d. *.exe

```
class Complex {  
    double x,y;  
    void ___() {System.___.println(x+“ ”+y);}  
}  
  
_____ class test{  
    public static void main (String[] ____){  
        Complex z1;  
        z1=new ____();  
        z1.x=3; z1.y=-4.7;  
        z1.afis();  
    }  
}
```

```
class Complex {  
    double x,y;  
    void afis() {System.out.println(x+" "+y);}  
}  
  
public class test{  
    public static void main (String[] args){  
        Complex z1;  
        z1=new Complex();  
        z1.x=3; z1.y=-4.7;  
        z1.afis();  
    }  
}
```

Programarea orientată pe obiect 2

Prelegerea nr. 7

Diagramele UML.

Crearea modelului conceptual.

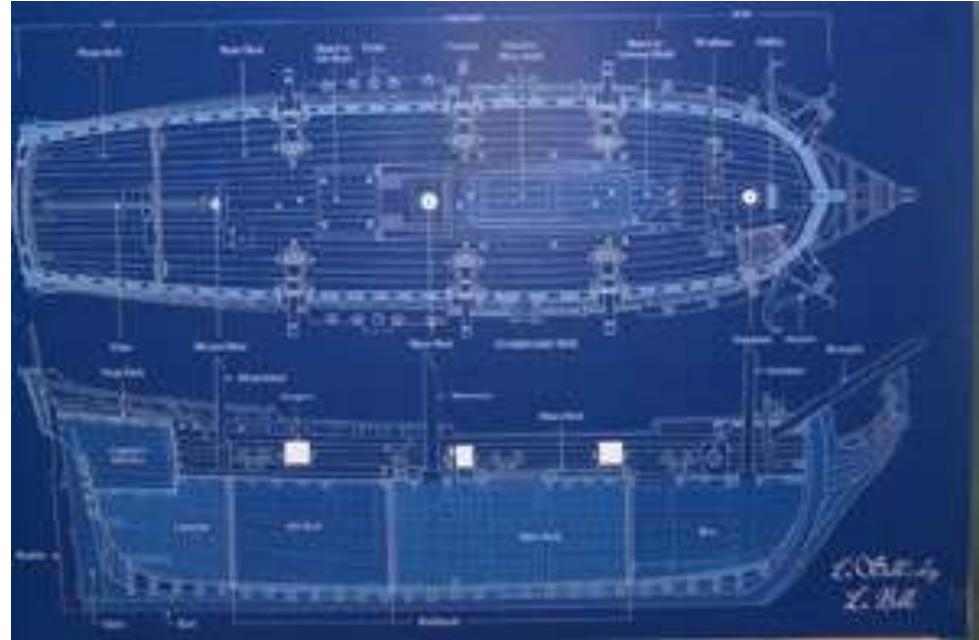
Petic Mircea

petic.mircea@yahoo.com

www.facebook.com/mircea.petic

Modelarea - De ce?

- ▶ Ce este un model?
 - Simplificarea realității
 - Planul detaliat al unui sistem (blueprints)



- ▶ De ce modelăm?
 - Pentru a înțelege mai bine ce avem de făcut
 - Pentru a ne concentra pe un aspect la un moment dat
- ▶ Unde folosim modelarea?

Scopurile Modelării

- ▶ Vizualizarea unui sistem
- ▶ Specificarea structurii sale și/sau a comportării
- ▶ Oferirea unui şablon care să ajute la construcție
- ▶ Documentarea deciziilor luate

Modelarea Arhitecturii

- ▶ Cu ajutorul **Use case**-urilor: pentru a prezenta cerințele
- ▶ Cu ajutorul **Design**-ului: surprindem vocabularul și domeniul problemei
- ▶ Cu ajutorul **Proceselor**: surprindem procesele și thread-urile
- ▶ Cu ajutorul **Implementării**: avem modelarea aplicației
- ▶ Cu ajutorul **Deployment**: surprindem sistemul din punct de vedere ingineresc

Principiile Modelării

- ▶ Modelele influențează soluția finală
- ▶ Se pot folosi diferite niveluri de precizie
- ▶ Modelele bune au corespondent în realitate
- ▶ Nu e suficient un singur model

Limbaje de Modelare

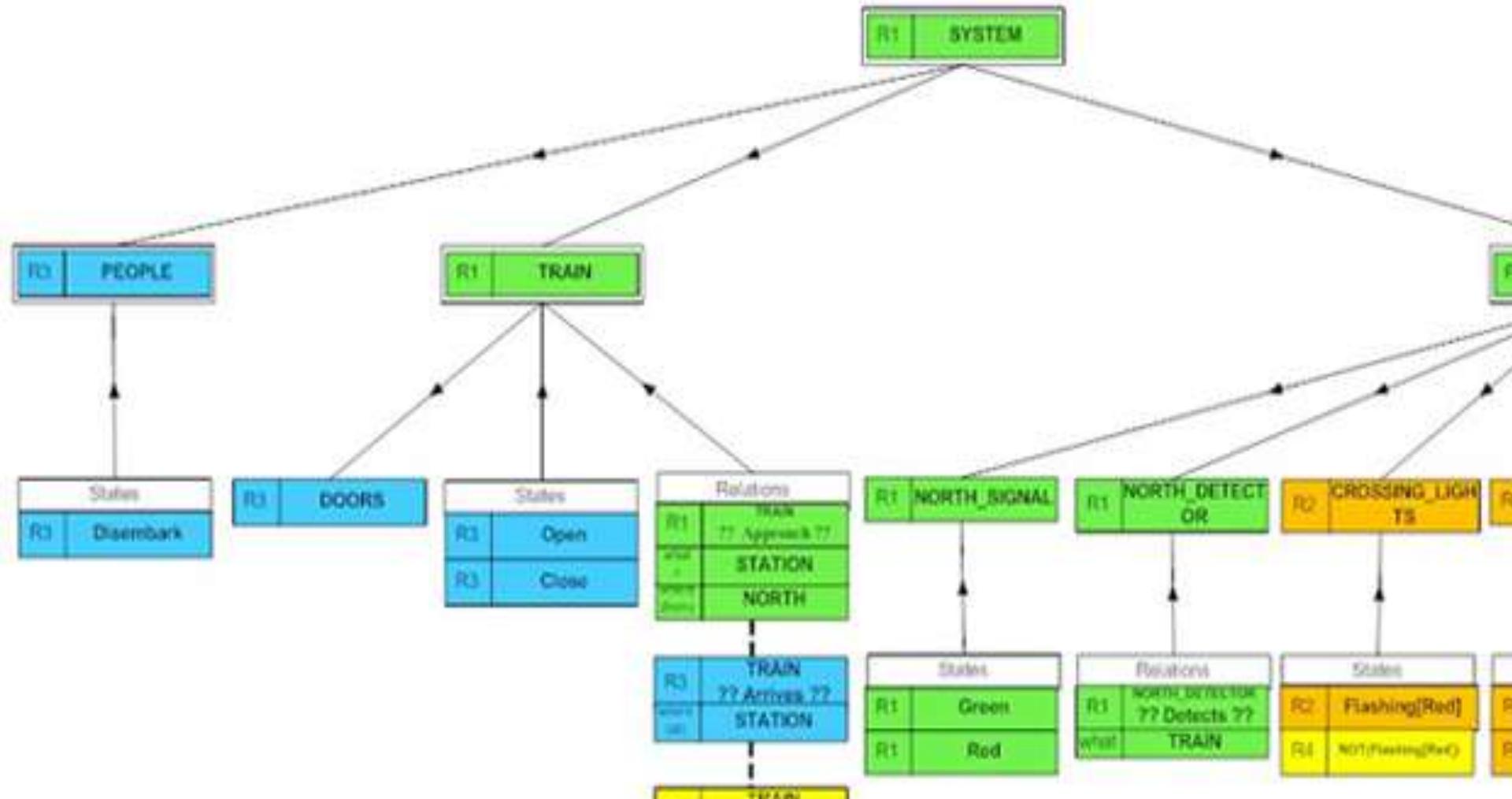
- ▶ Analiza și proiectarea unui proiect trebuie făcute înainte de realizarea codului
- ▶ În prezent, se acordă o atenție deosebită acestei etape, deoarece de ele depind **producerea și refolosirea de software**
- ▶ Pentru analiza și proiectarea programelor s-au creat **limbajele de modelare**
- ▶ **Limbaj de modelare este un limbaj artificial care poate fi folosit să exprime informații sau cunoștere sau sisteme**

Tipuri de Limbaje de Modelare

- ▶ **Limbaje Grafice:** arbori comportamentali, modelarea proceselor de business, EXPRESS (modelarea datelor), flowchart, ORM (modelarea rolurilor), rețelele Petri, **diagrame UML**
- ▶ **Limbaje Specifice:** modelare algebrică (AML) (pentru descrierea și rezolvarea problemelor de matematică ce necesită putere computațională mare), modelarea domeniilor specifice (DSL), modelarea arhitecturilor specifice (FSML), modelarea obiectelor (object modeling language), modelarea realității virtuale (VRML)

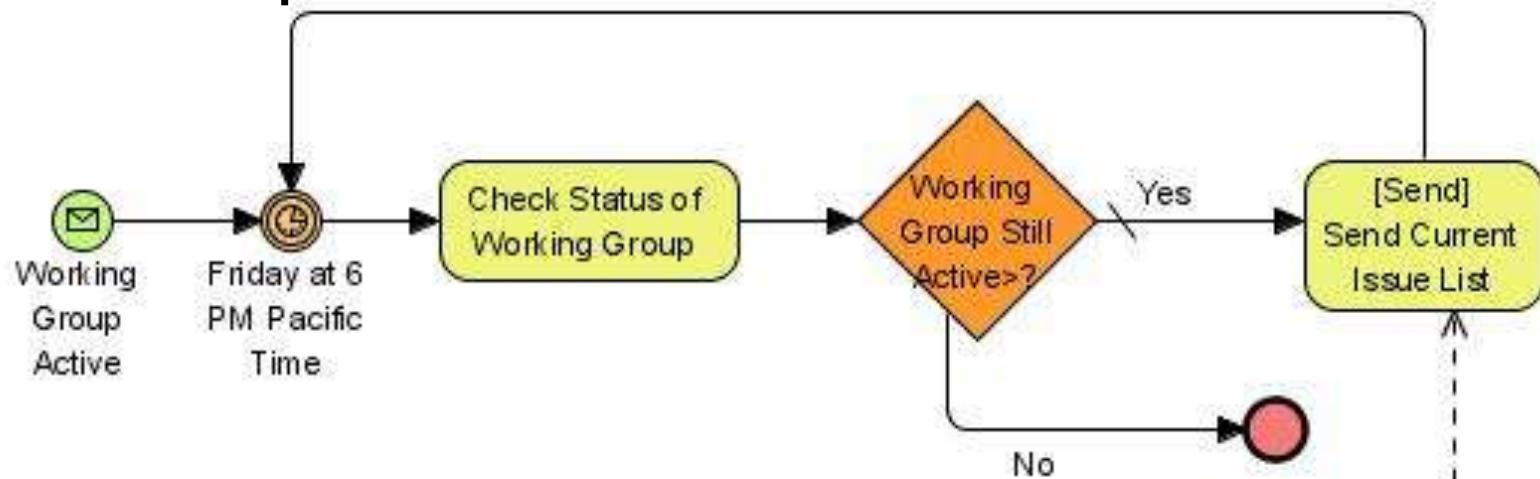
Limbaje Grafice 1

► Arbori comportamentali

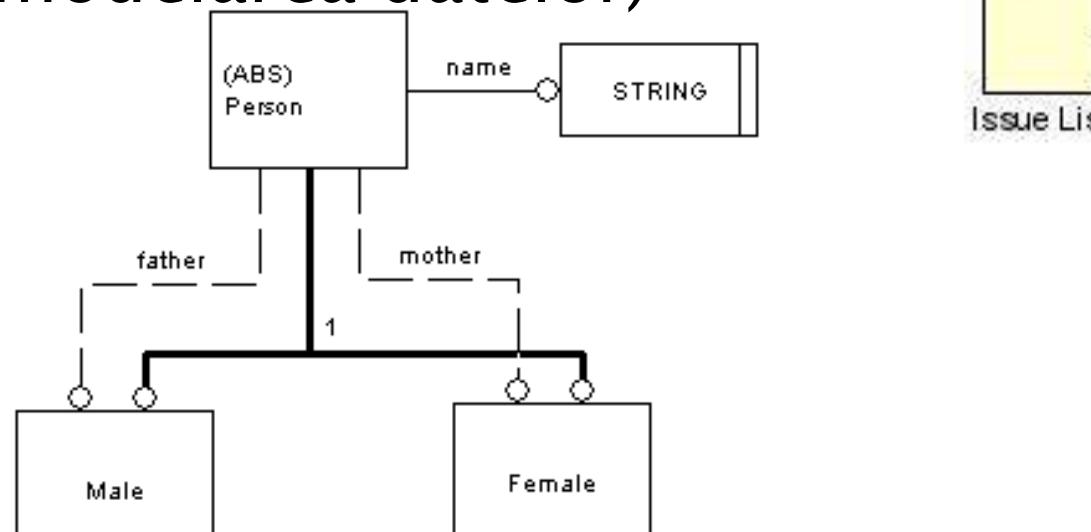


Limbaje Grafice 2

► Modelarea proceselor de business

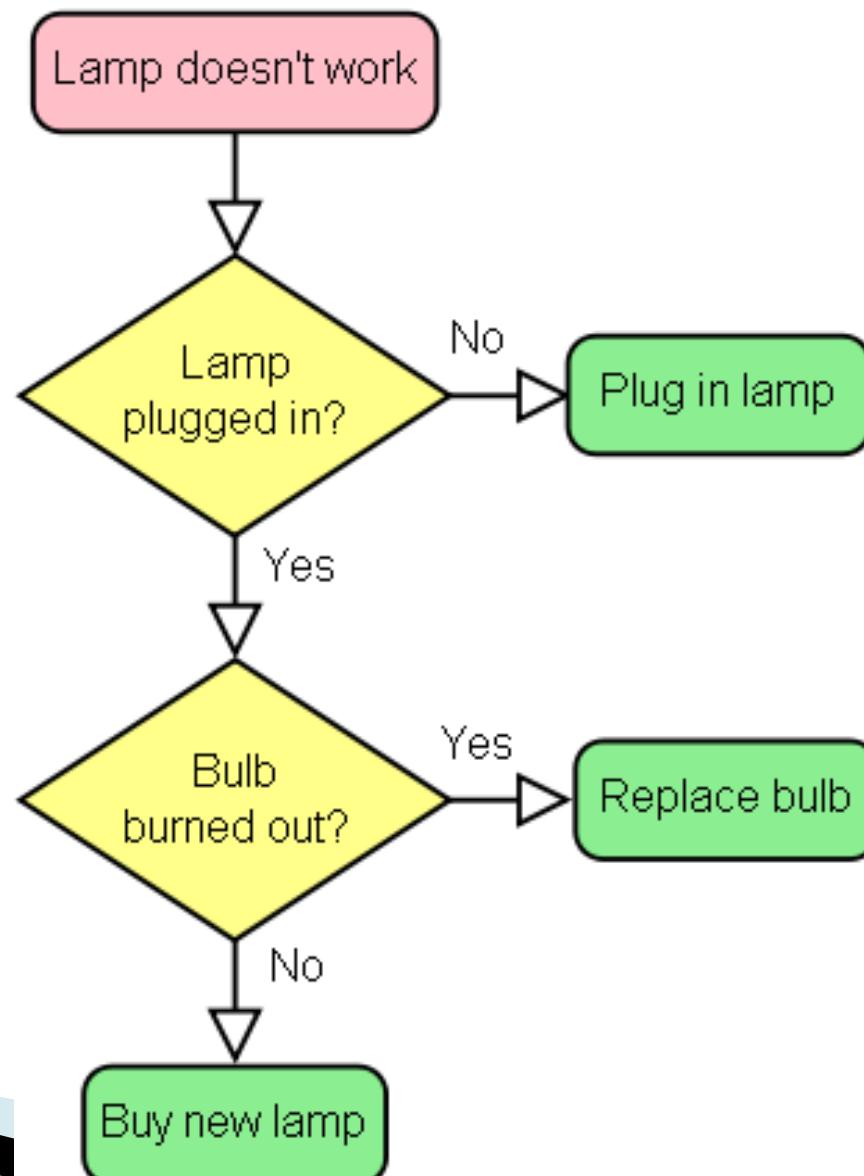


► EXPRESS (modelarea datelor)



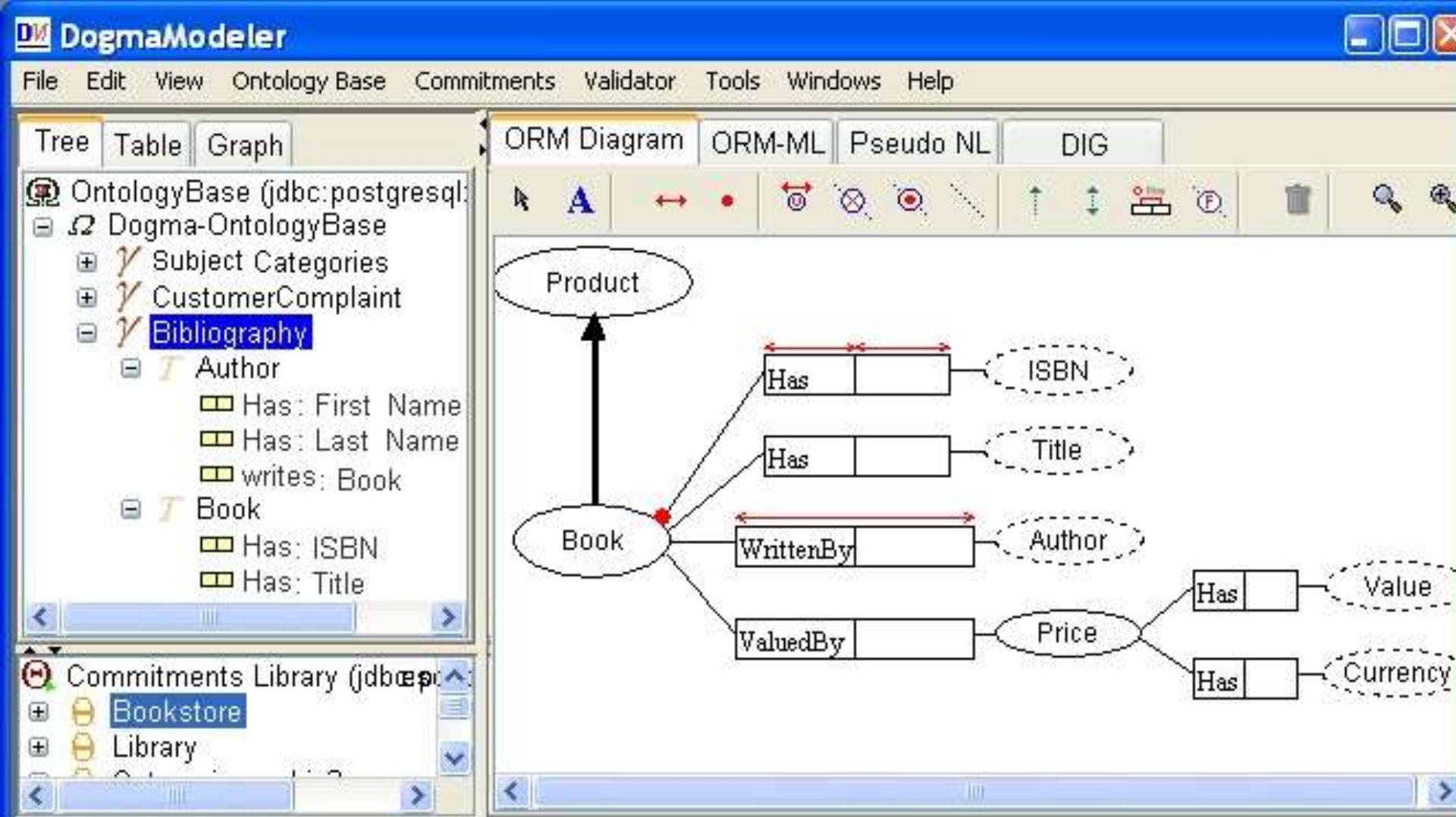
Limbaje Grafice 3

▶ Flowchart



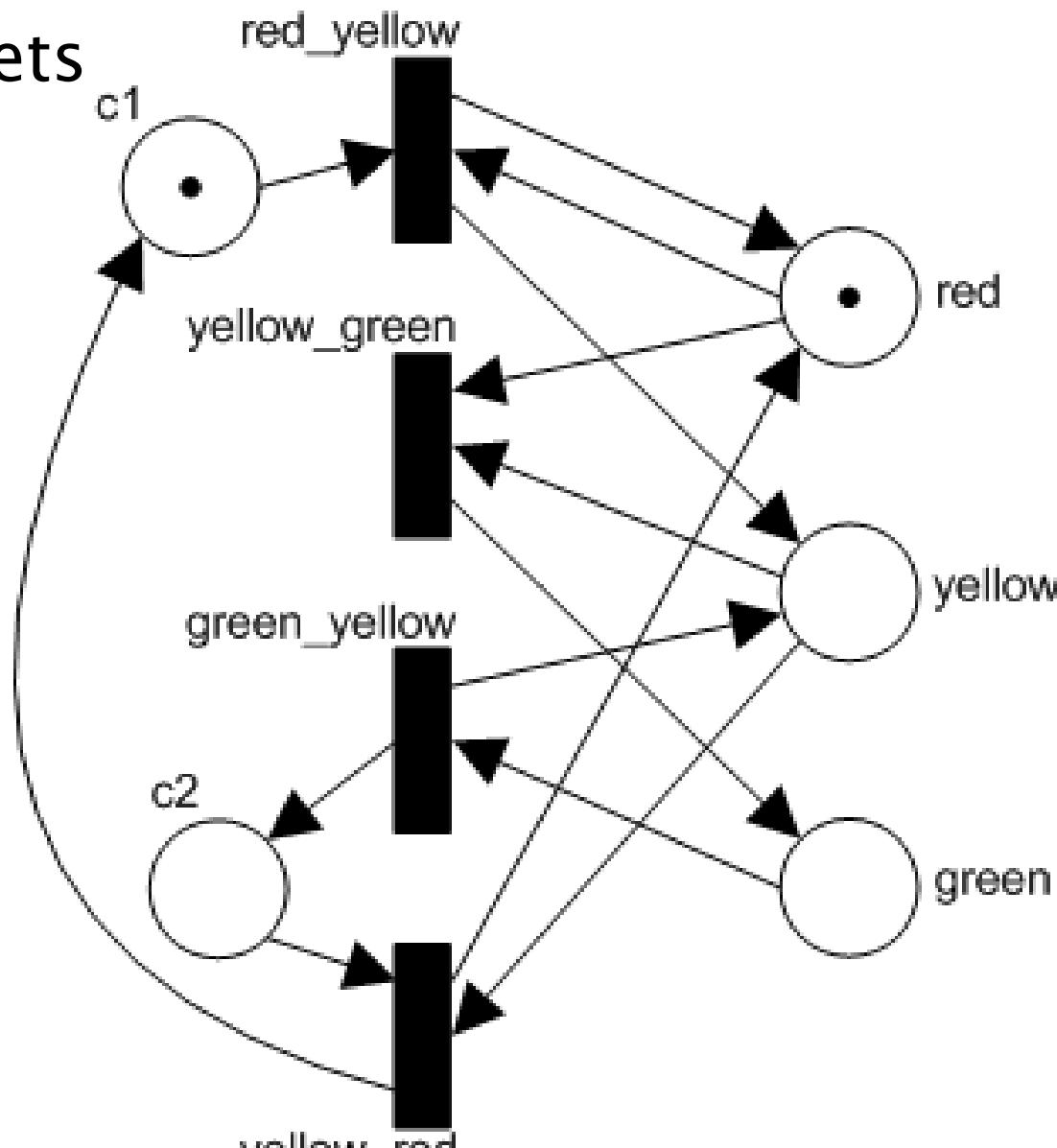
Limbaje Grafice 4

▶ ORM (Object Role Modeling)



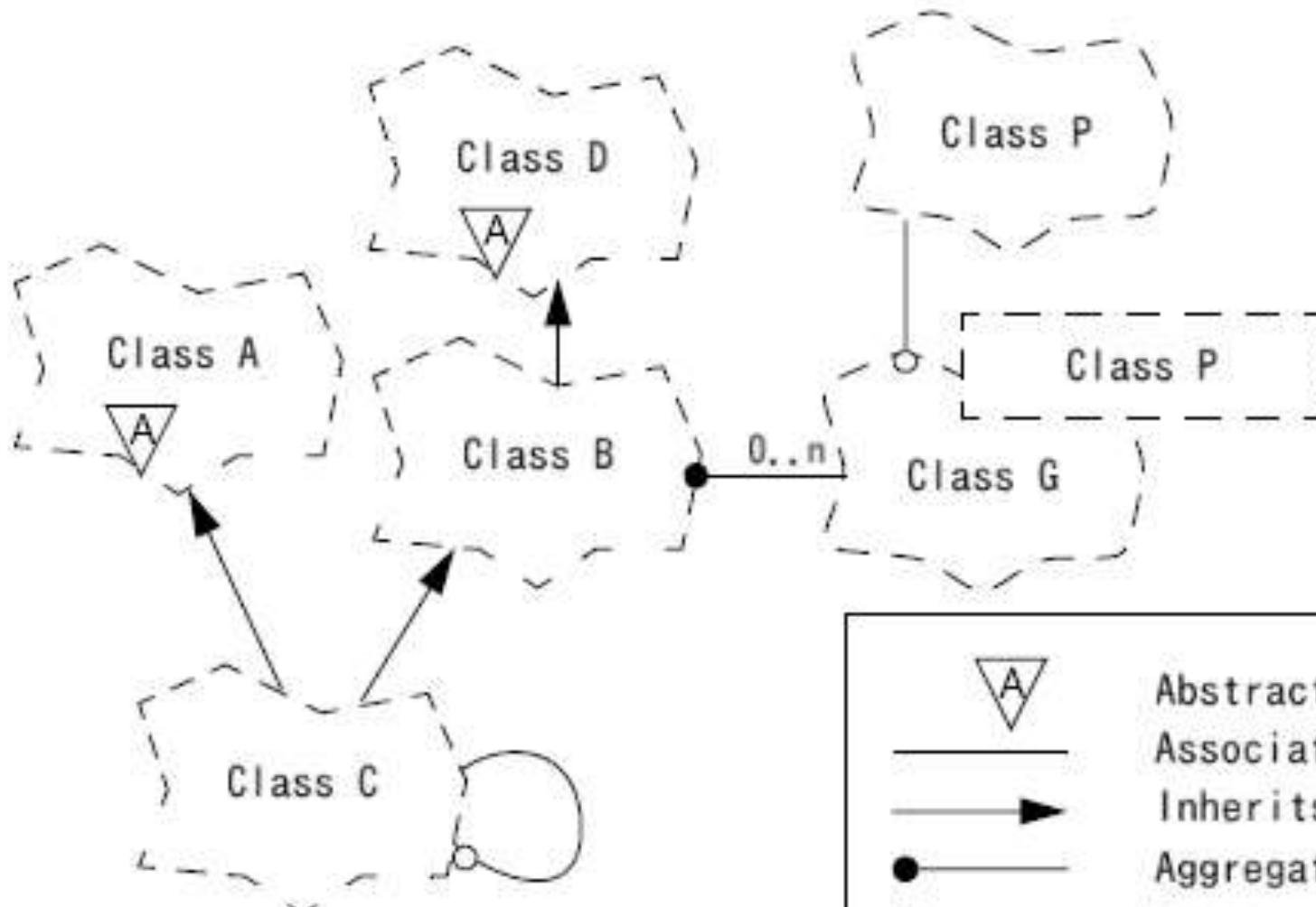
Limbaje Grafice 5

▶ Petri Nets



Limbaje Grafice 6

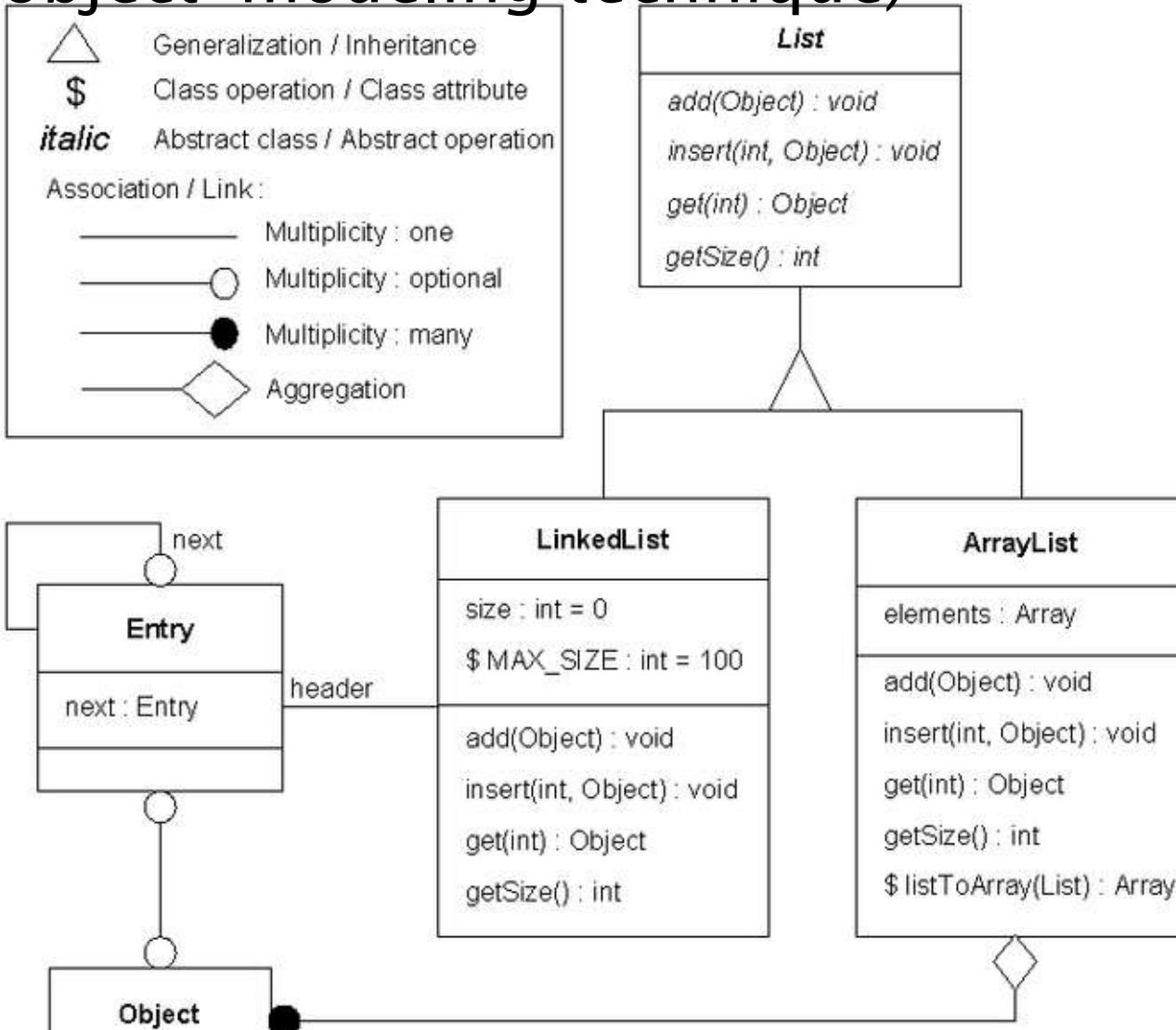
- ▶ Metoda Booch (Grady Booch) – analiza și design oo



	Abstract Class
	Association
	Inherits
	Aggregation
	Uses

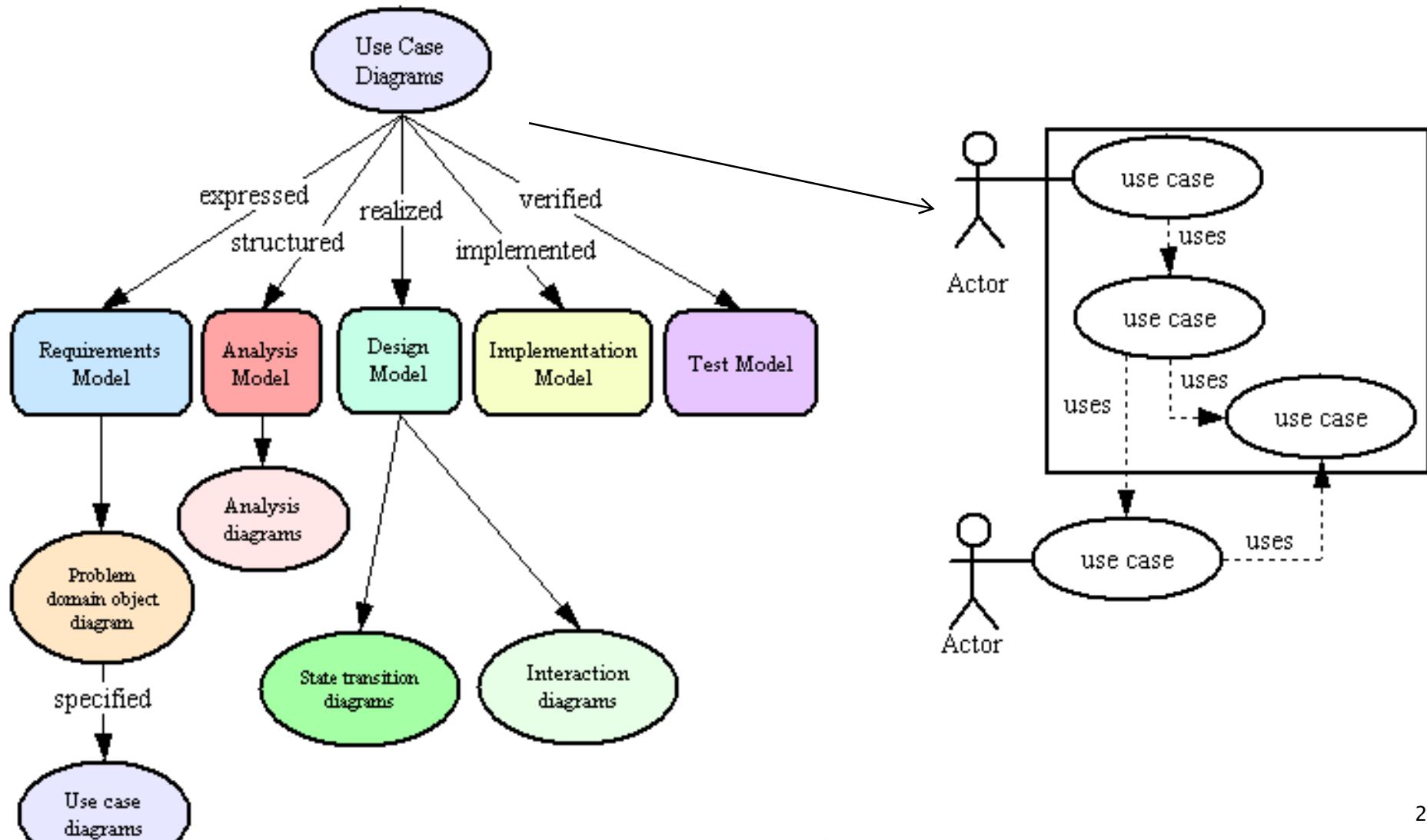
Limbaje Grafice 7

▶ OMT (object-modeling technique)



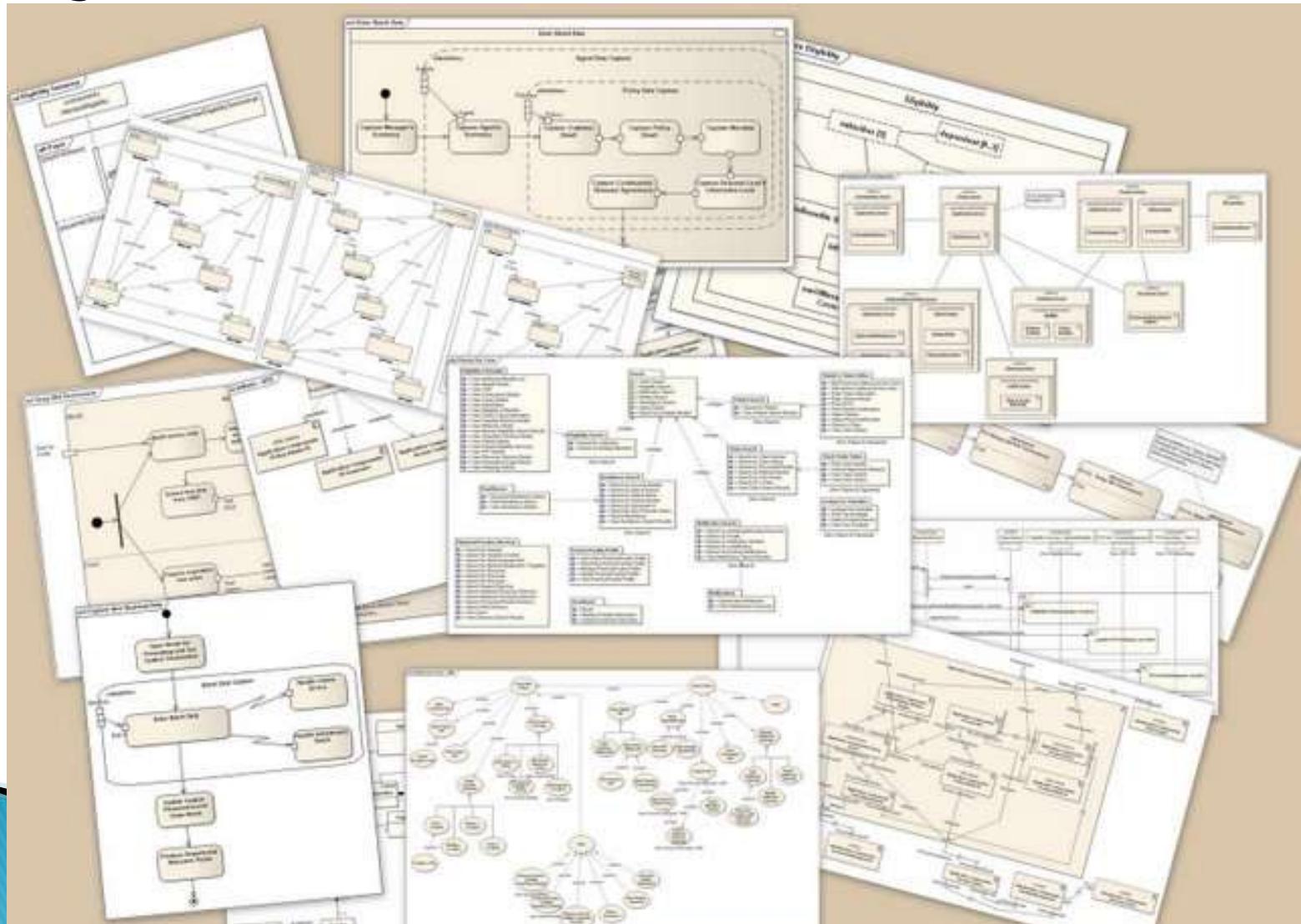
Limbaje Grafice 8

▶ OOSE (Object-oriented software engineering)



Limbaje Grafice 9

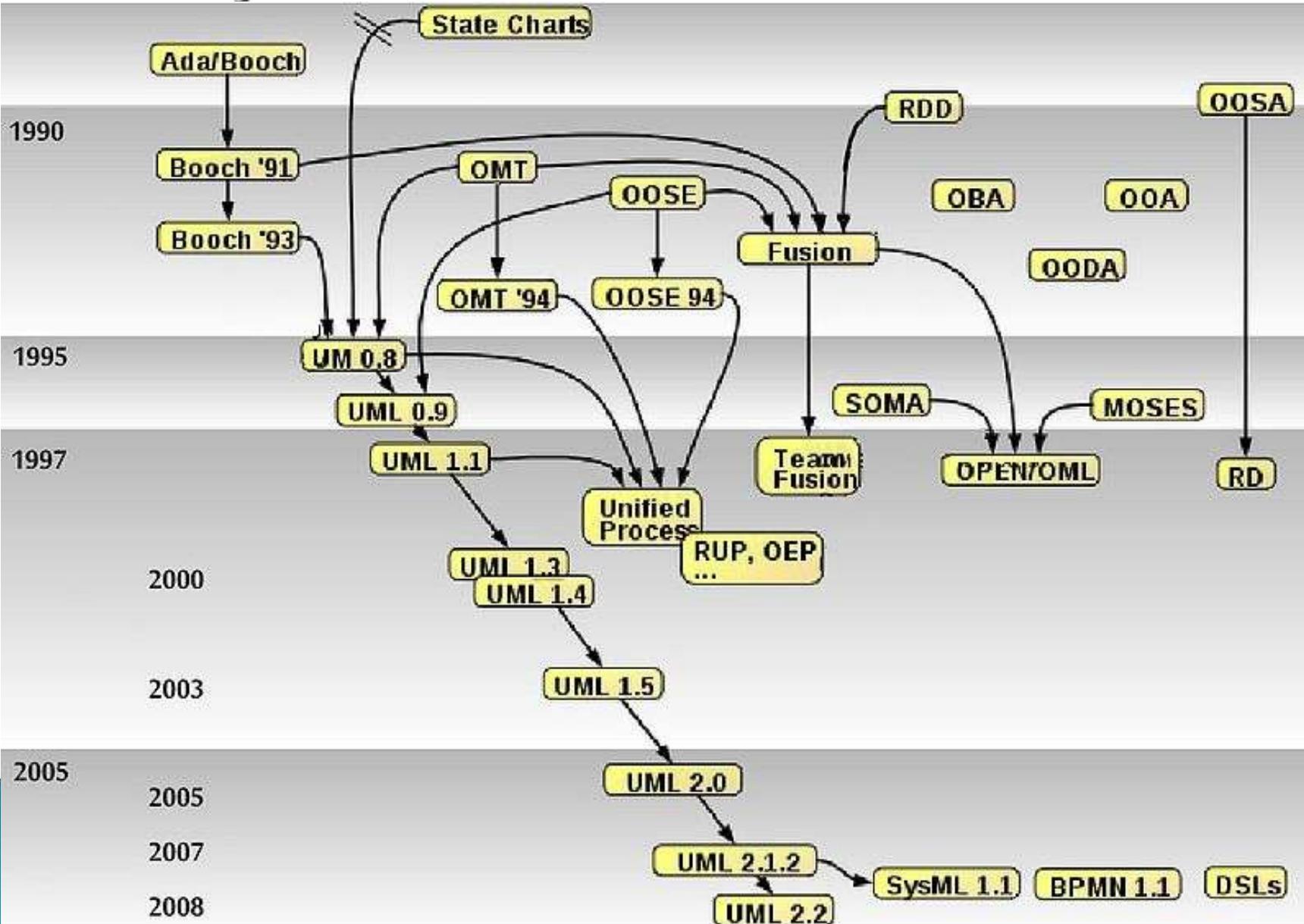
▶ Diagramme UML



UML – Introducere

- ▶ UML (Unified Modeling Language) este succesorul celor mai bune trei limbaje OO de modelare anterioare:
 - Booch (Grady Booch)
 - OMT (Ivar Jacobson)
 - OOSE (James Rumbaugh)
- ▶ UML se constituie din unirea acestor limbaje de modelare și în plus are o expresivitate mai mare

Evoluție UML



UML - Definiție (OMG)

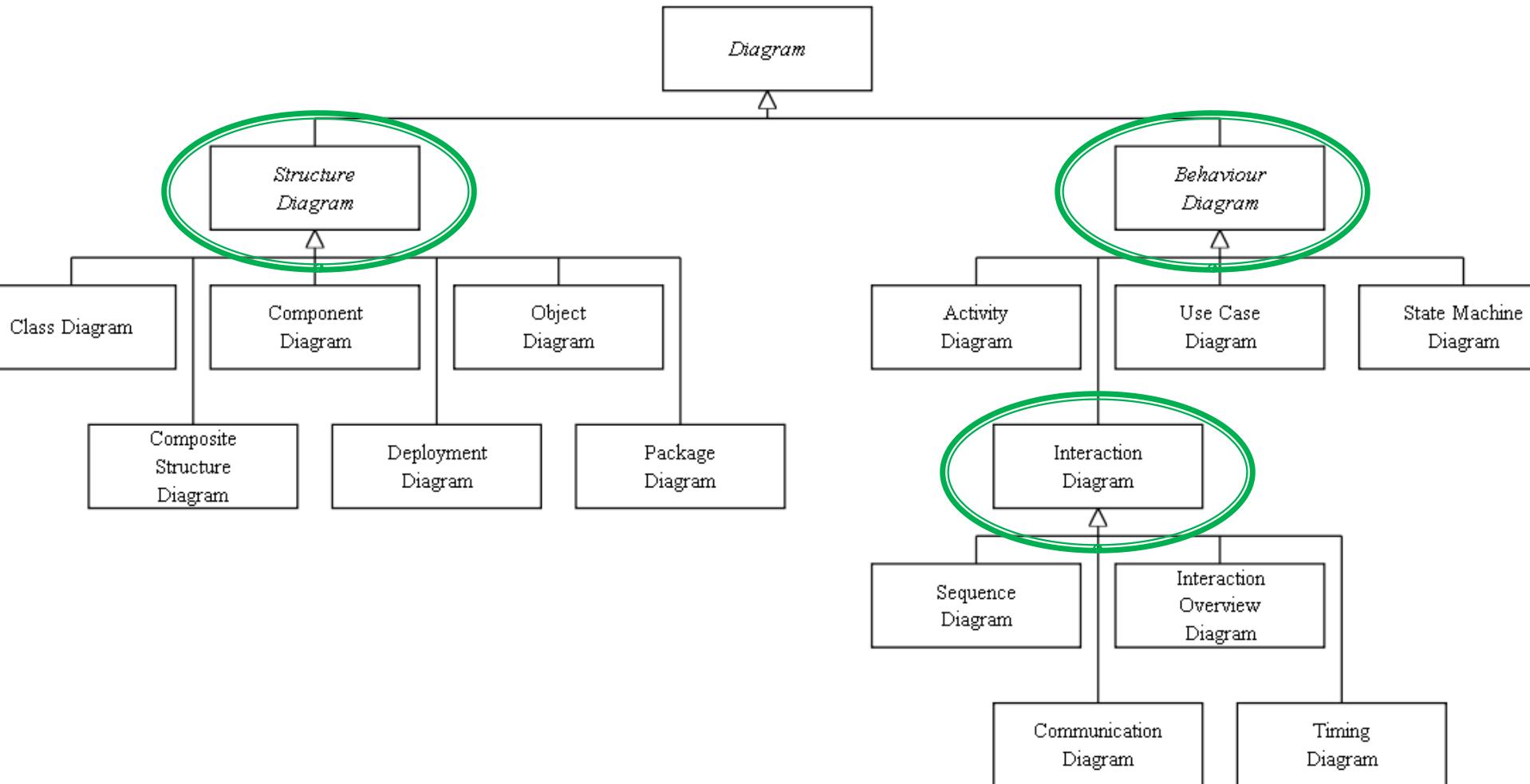
- ▶ "*The Unified Modeling Language (UML) is a graphical language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system.*"
- ▶ *The UML offers a standard way to write a system's blueprints, including conceptual things such as business processes and system functions as well as concrete things such as programming language statements, database schemas, and reusable software components.*"

UML – Standard Internațional

- ▶ Ianuarie 1997 – UML 1.0 a fost propus spre standardizare în cadrul OMG (Object Management Group)
- ▶ Noiembrie 1997 – Versiunea UML 1.1 a fost adoptată ca standard de către OMG
- ▶ Ultima versiune este UML 2.2
- ▶ Site-ul oficial: <http://www.uml.org>

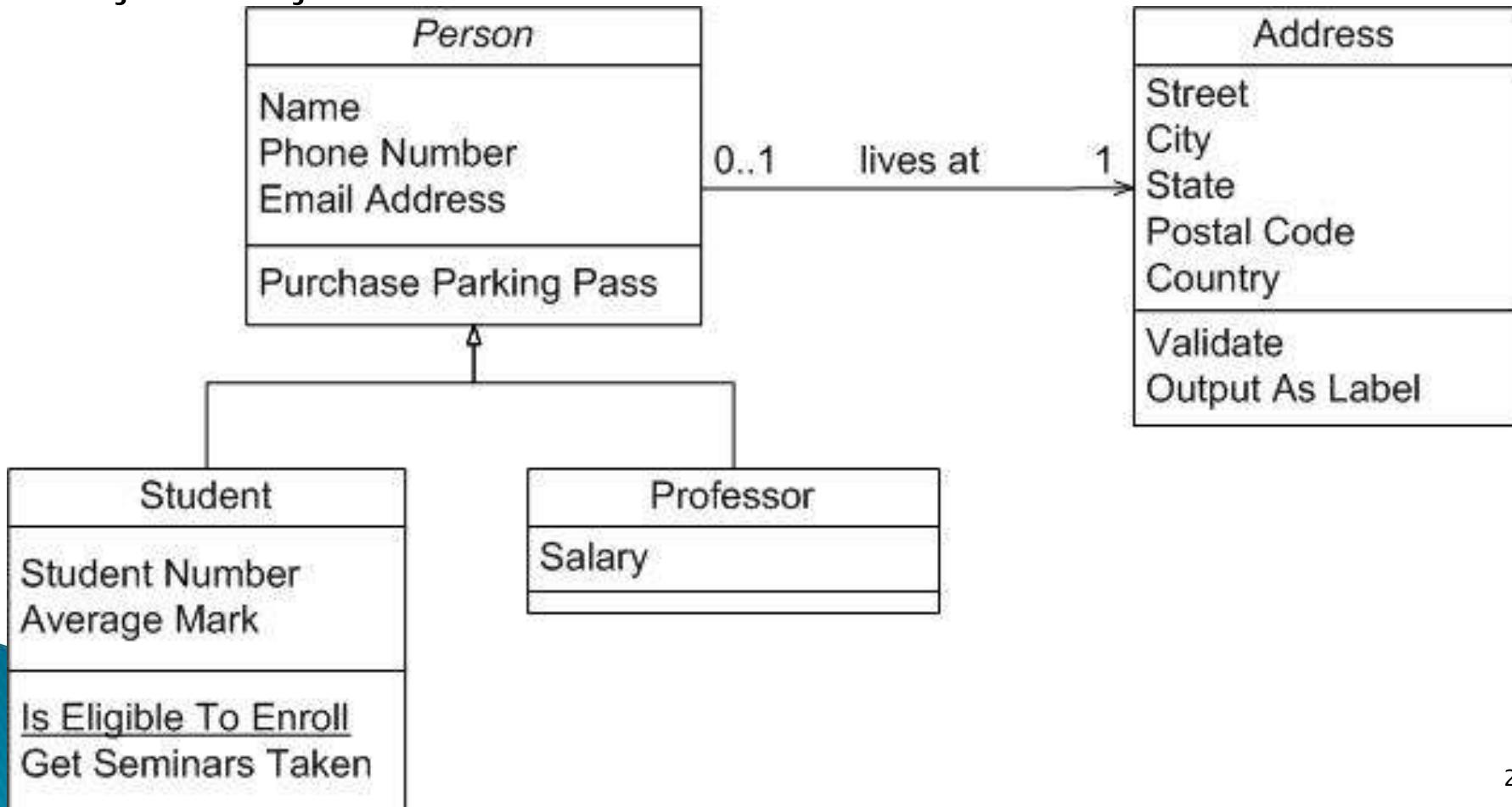
The screenshot shows the homepage of the OMG website. At the top, there's a navigation bar with links for "About Us", "Press Room", "Calendar", "Documents", "Members Only", "Technology", and "Industries". Below the navigation bar, the "OMG" logo is displayed with the tagline "WE SET THE STANDARD". To the right of the logo, the "UNIFIED MODELING LANGUAGE™" logo is shown. At the bottom of the page, there's a section titled "UML® Resource Page" with a horizontal menu containing links for "Introduction to UML", "UML Success Stories", "UML Certification Program", and "Vendor Directory".

UML2.0 – 13 Tipuri de Diagrame



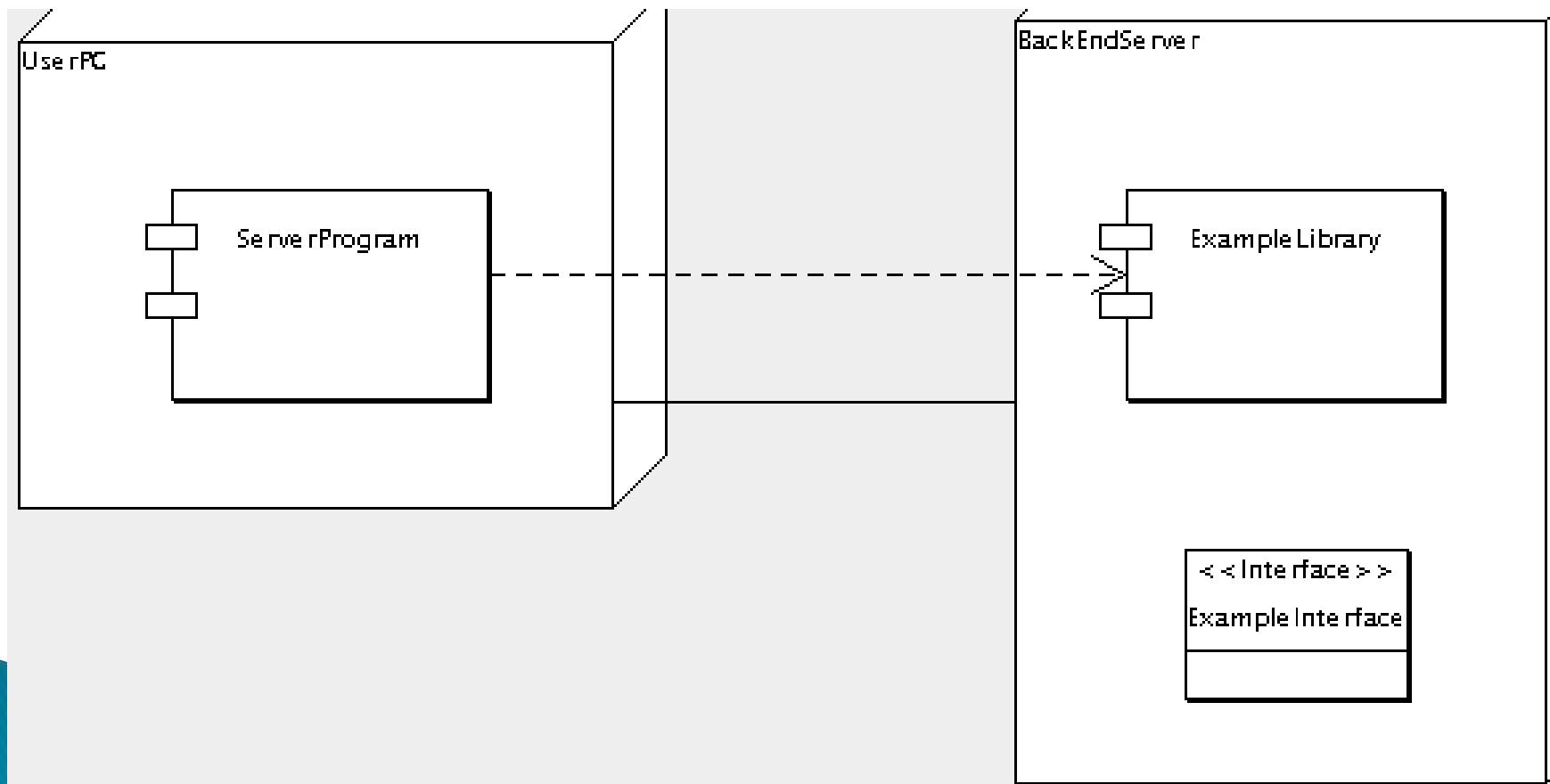
UML2.0 – Diagramme de Structură 1

- ▶ **Diagramme de Clasă:** clasele (atributele, metodele) și relațiile dintre clase



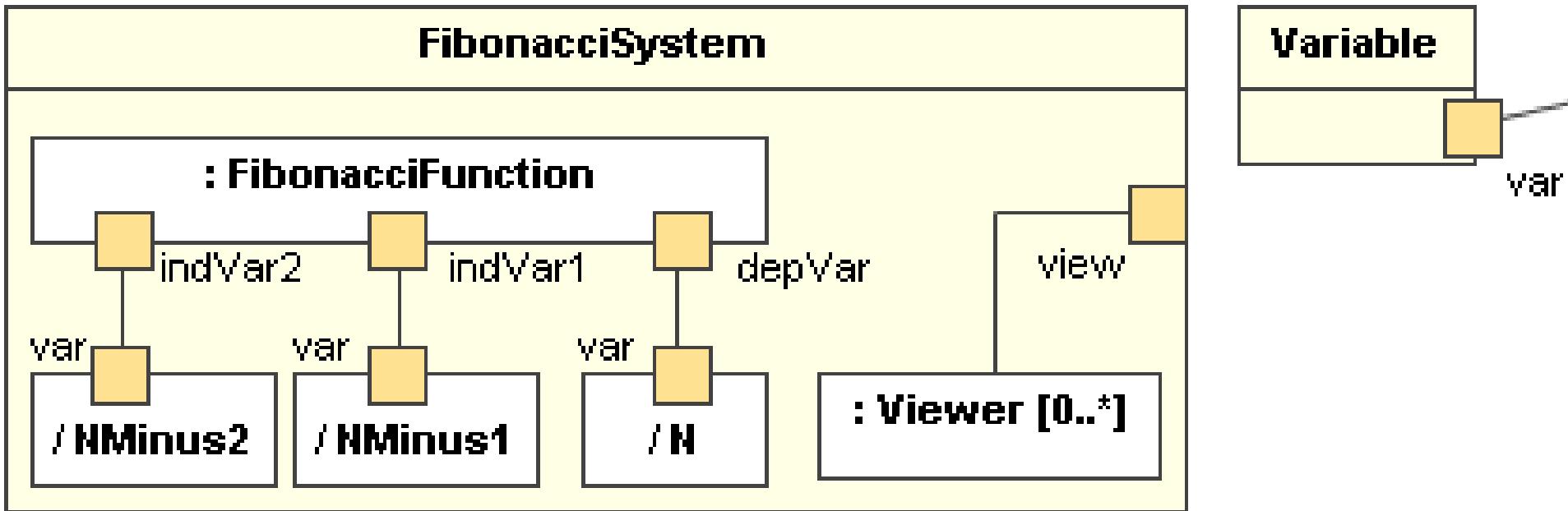
UML2.0 – Diagramme de Structură 2

- ▶ **Diagramă de Componente:** componentele sistemului și legăturile între componente



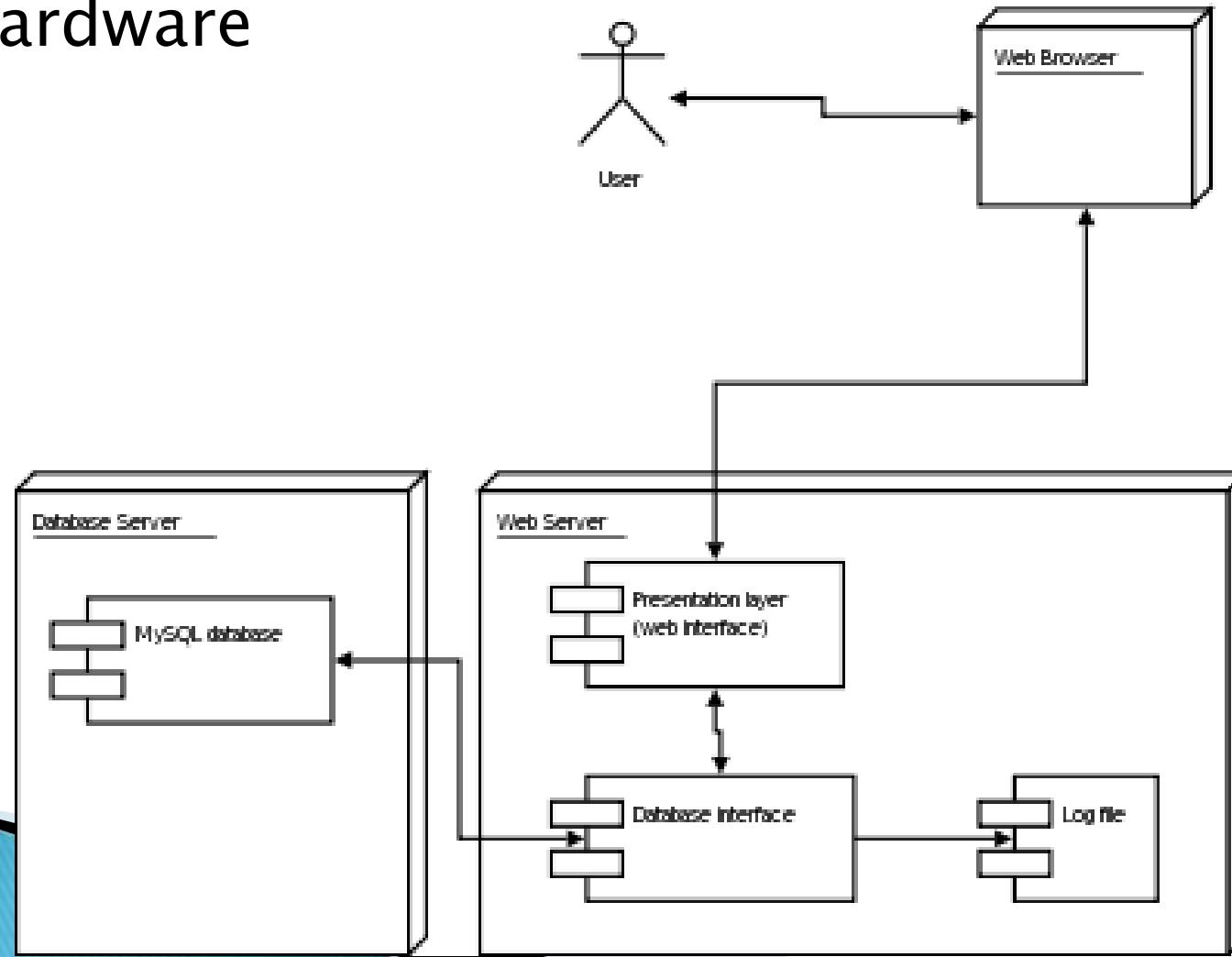
UML2.0 - Diagrame de Structură 3

- ▶ Diagrame structură composită: structura internă



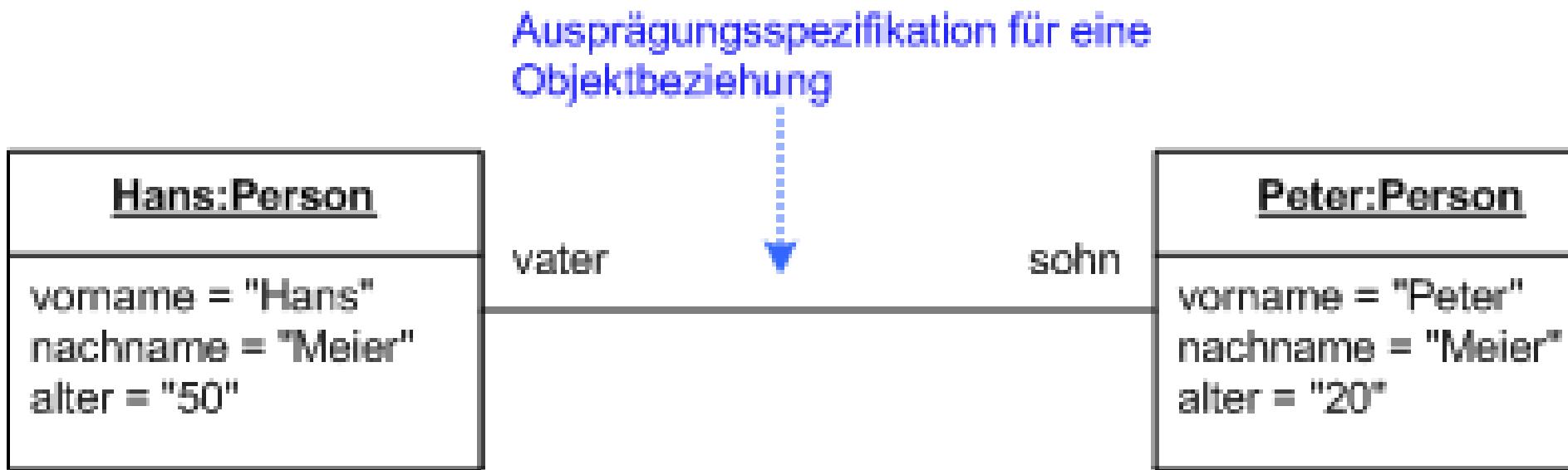
UML2.0 - Diagramă de Structură 4

- ▶ Diagramă de Deployment: modelarea structurii hardware



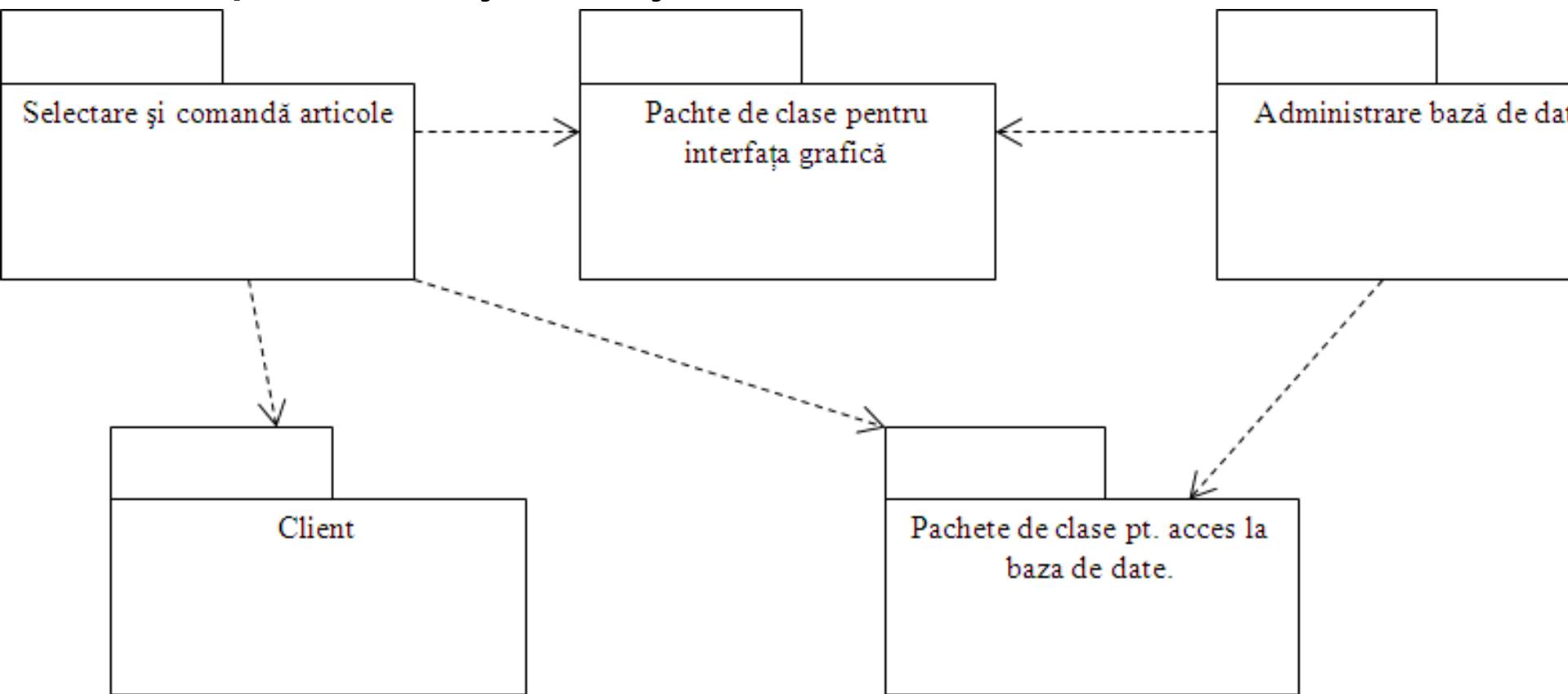
UML2.0 – Diagramme de Structură 5

- ▶ Diagramă de obiecte: structura sistemului la un moment dat



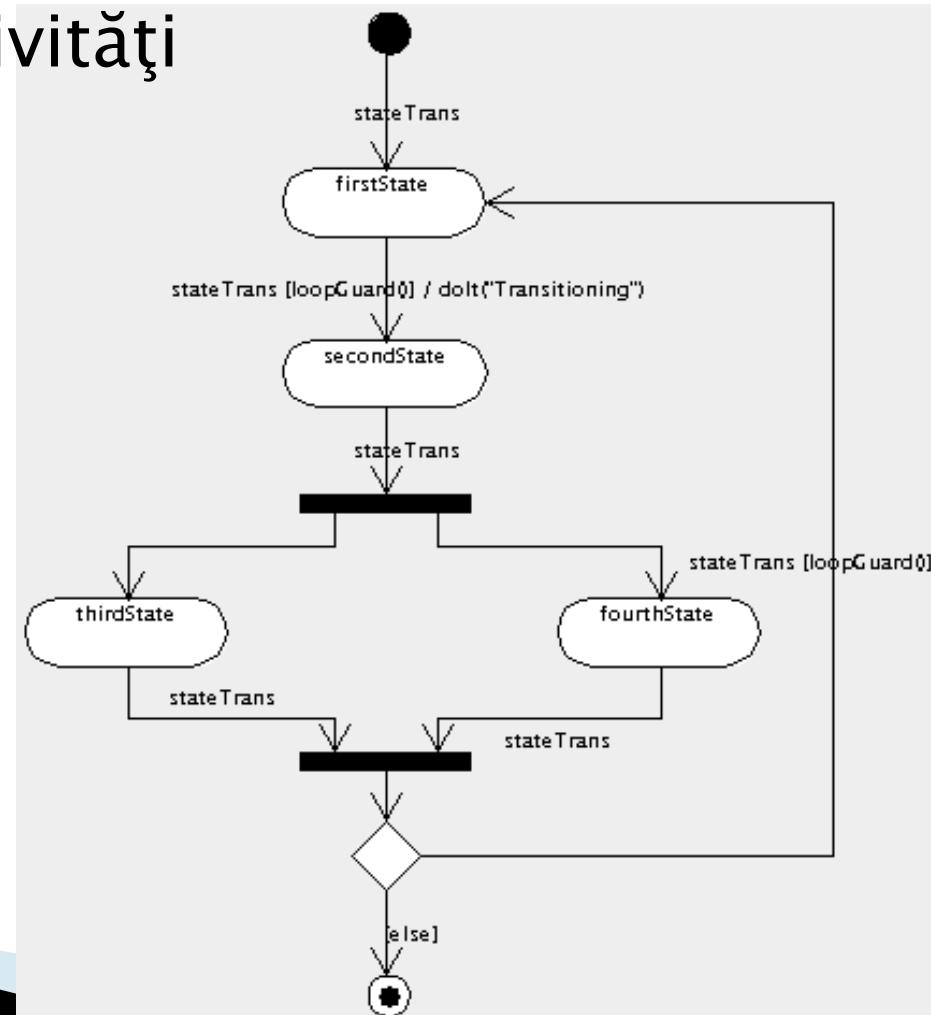
UML2.0 - Diagramme de Structură 6

- ▶ **Diagramă de pachete:** împărțirea sistemului în pachete și relațiile dintre ele



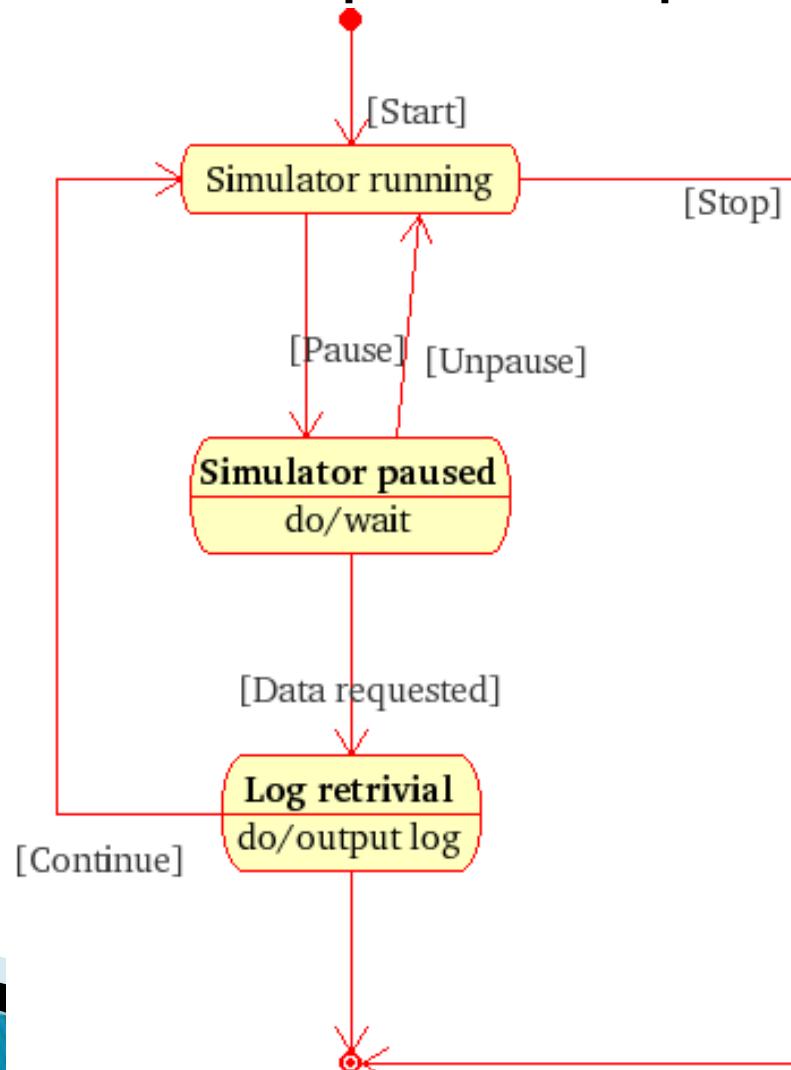
UML 2.0 – Diagrame Comportamentale 1

- ▶ **Diagrame de activitate:** prezentare business și a fluxului de activități



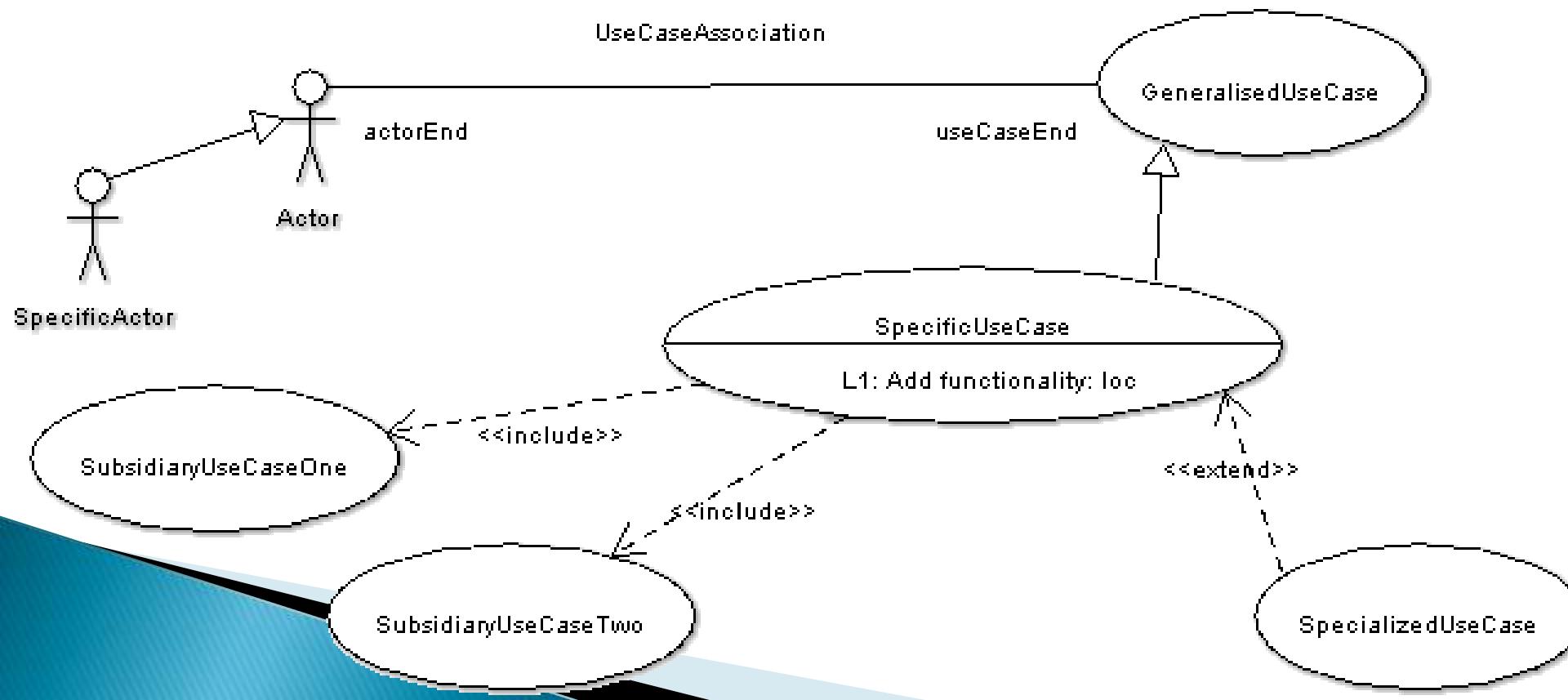
UML 2.0 – Diagrame Comportamentale 2

- ▶ Diagrame de stare: pentru a prezenta mai multe sisteme



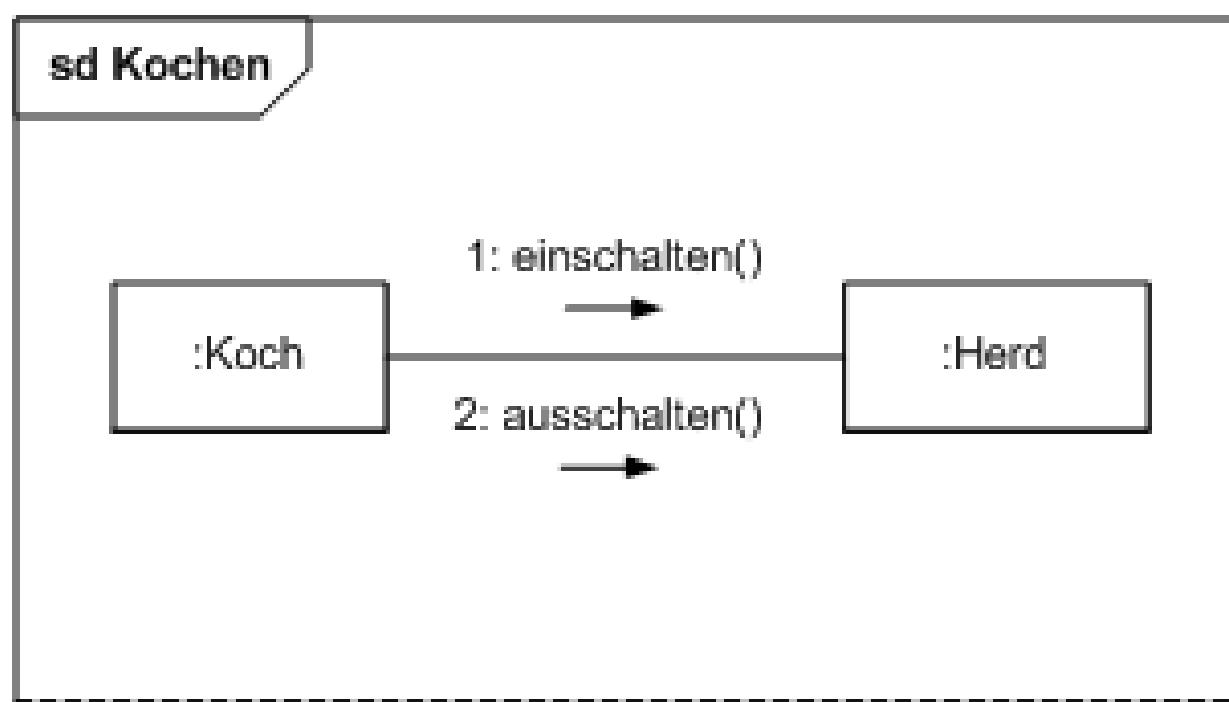
UML 2.0 – Diagrame Comportamentale 3

- ▶ **Diagrame Use Case:** prezintă funcționalitățile sistemului folosind actori, use case-uri și dependențe între ele



UML 2.0 – Diagrame de interacțiuni 1

- ▶ **Diagrama de comunicare:** arată interacțiunile între obiecte (comportamentul dinamic al sistemului) (actori: bucătar, aragaz, acțiuni: gătirea, aprinderea, deconectarea)



UML 2.0 – Diagrame de interacțiuni 2

- ▶ Diagramă de sevență: prezintă modul în care obiectele comunică între ele din punct de vedere al trimiterii de mesaje

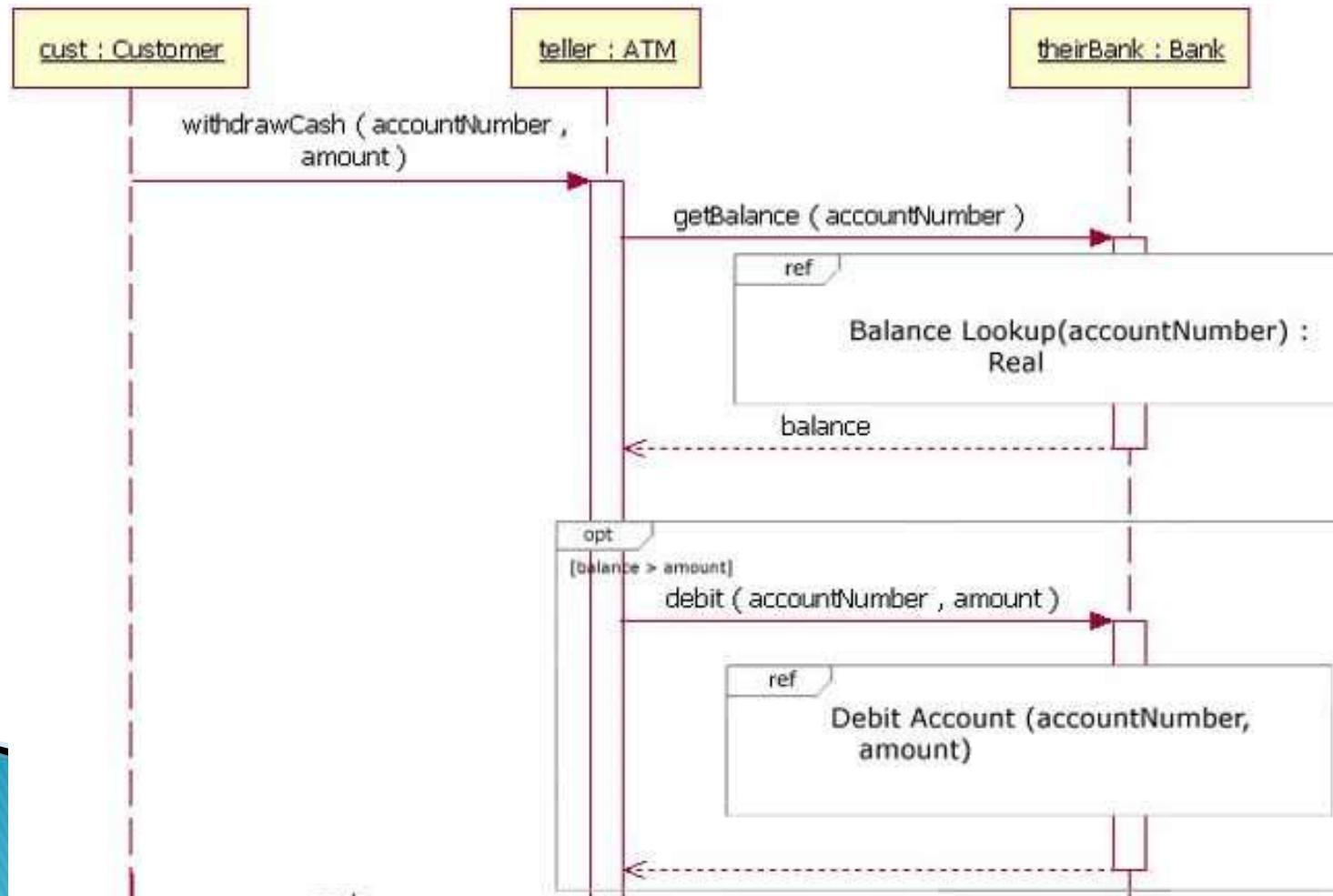


Diagrama cazurilor de utilizare (*Use Case Diagram*)

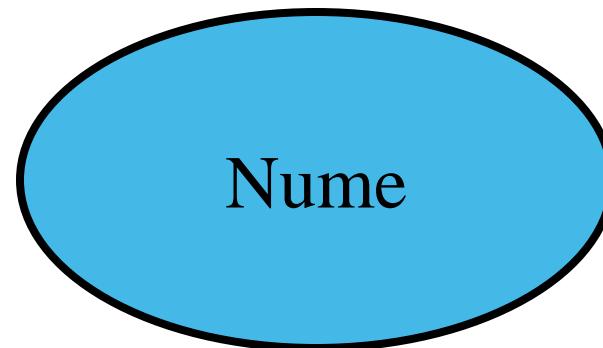
- ▶ Este o diagramă comportamentală care captează cerințele sistemului
 - ▶ Delmitează granițele sistemului
 - ▶ Punctul de plecare îl constituie scenariile de folosire a sistemului din fișa cerințelor
 - ▶ Poate prezenta:
 - specificarea cerințelor (externe) din punctul de vedere al utilizatorului
 - specificarea funcționalității sistemului din punctul de vedere al sistemului
 - ▶ Conține:
 - **UseCase**-uri = funcționalități ale sistemului
 - **Actori** = entități externe cu care sistemul interacționează
- Relatii**

UseCase

- ▶ Este o descriere a unei mulțimi de secvențe de acțiuni (incluzând variante) pe care un program le execută atunci când interacționează cu entitățile din afara lui (*actori*) și care conduc la obținerea unui rezultat observabil
- ▶ Poate fi un sistem, un subsistem, o clasă, o metodă
- ▶ Reprezintă o funcționalitate a programului
- ▶ Precizează ce face un program sau subprogram
- ▶ Nu precizează cum se implementează o funcționalitate
- ▶ Identificarea UseCase-urilor se face pornind de la cerințele clientului și analizând descrierea problemei.

UseCase – Reprezentare

- ▶ Notație
- ▶ Atribute
 - Nume = fraza verbală ce denumește o operatie sau un comportament din domeniul problemei.
- ▶ Restricții
 - Numele este unic

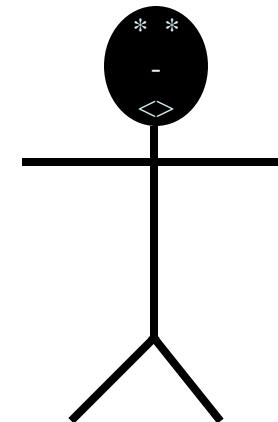


Actor

- ▶ Reprezintă un rol pe care utilizatorii unui UseCase îl joacă atunci când interacționează cu acesta
- ▶ Este o entitate exterioara sistemului
- ▶ Interacționează cu sistemul:
 - Inițiază execuția unor cazuri de utilizare
 - Oferă funcționalitate pentru realizarea unor cazuri de utilizare
- ▶ Poate fi:
 - Utilizator (uman)
 - Sistem software
 - Sistem hardware

Actor – Reprezentare

- ▶ Notație
- ▶ Atribute
- ▶ Nume = **indica rolul pe care actorul îl joacă în interacțiunea cu un UseCase**
- ▶ Restricții
 - Numele este unic



Relații

- ▶ Se stabilesc între două elemente
- ▶ Tipuri de relații:
 - **Asociere:** Actor – UseCase, UseCase – UseCase
 - **Generalizare:** Actor – Actor, UseCase – UseCase
 - **Dependență:** UseCase – UseCase
(<<include>>, <<extend>>)

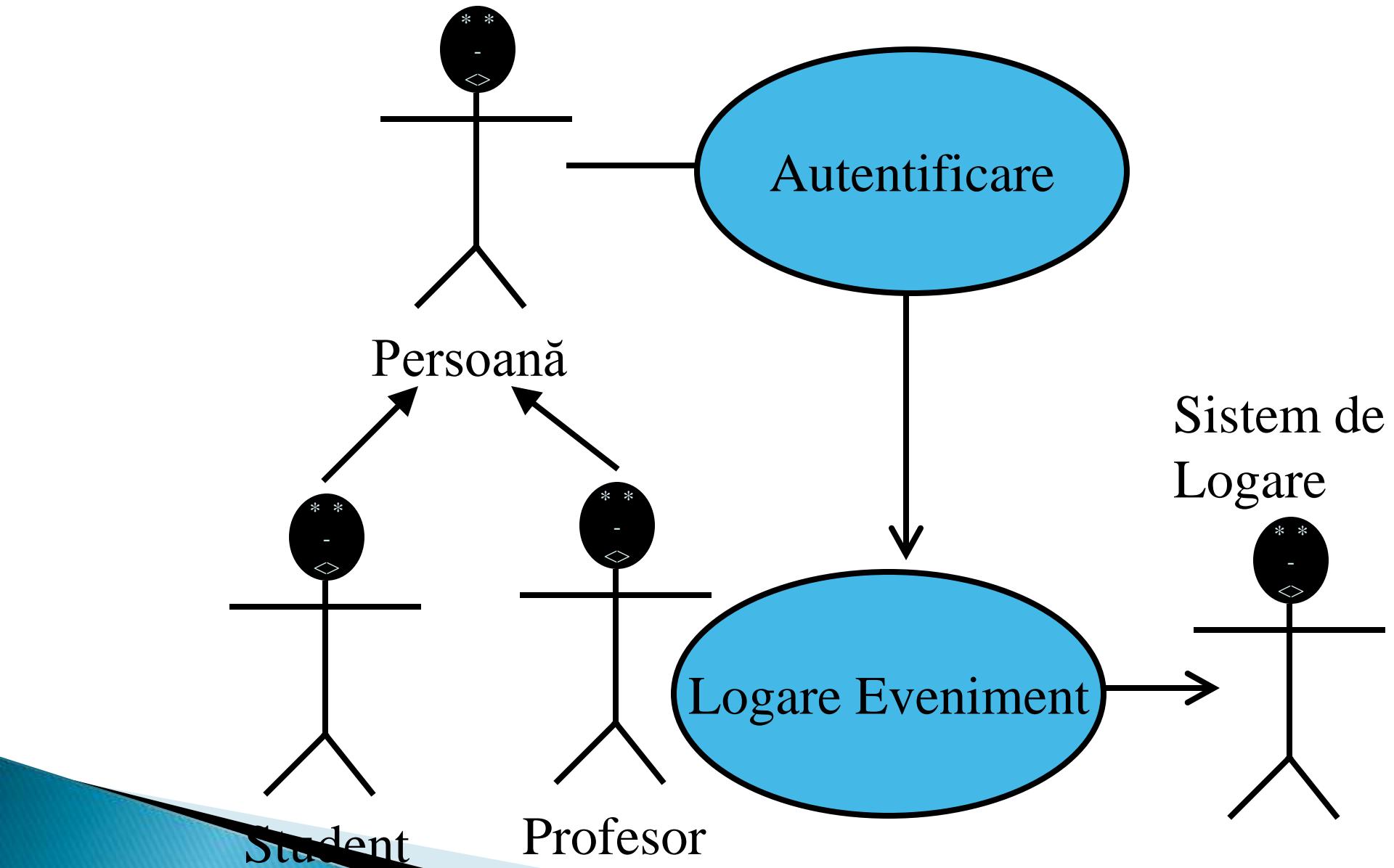
Relația de Asociere

- ▶ Modelează o comunicare între elementele pe care le conectează
- ▶ Poate să apară între
 - un actor și un UseCase (actorul inițiază execuția cazului de utilizare sau oferă funcționalitate pentru realizarea acestuia)
 - două UseCase-uri (transfer de date, trimitere de mesaje/semnale)
- ▶ Notație

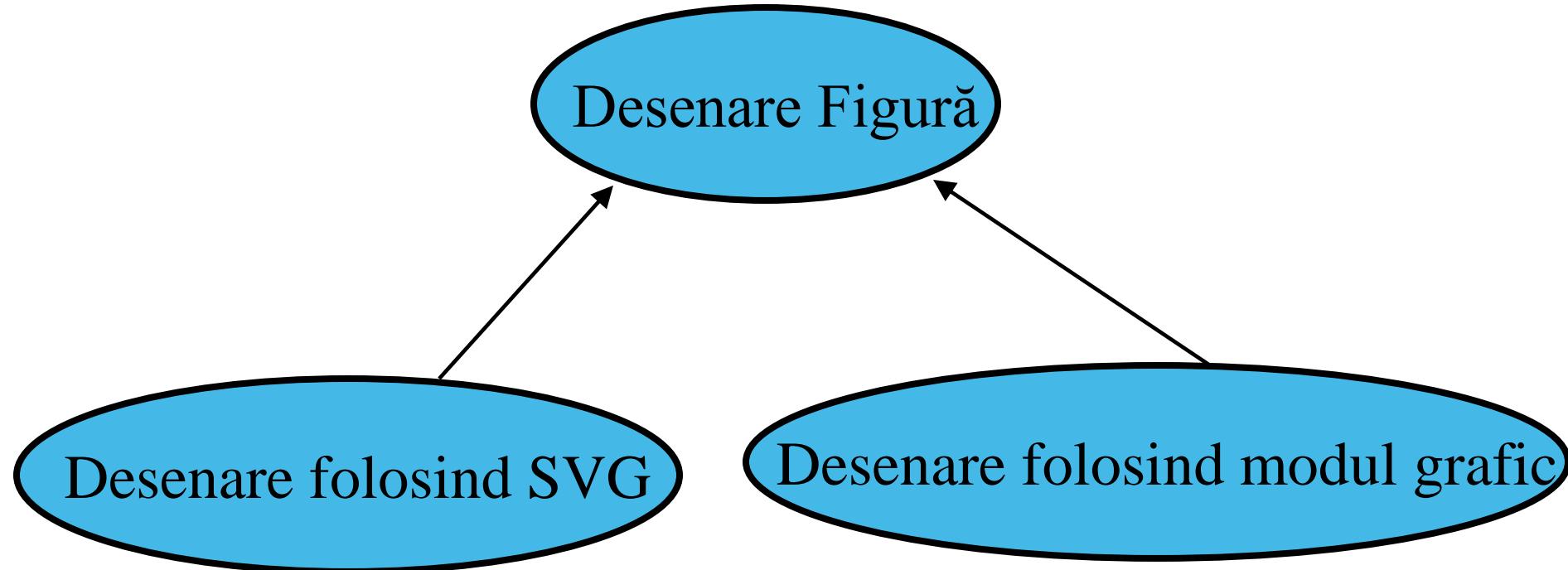
Relația de Generalizare

- ▶ Se realizează între elemente de același tip ⇒ ierarhii
- ▶ Modeleză situații în care un element este un caz particular al altui element
- ▶ Elementul particular moștenește relațiile în care este implicat elementul general
- ▶ Notație: ←

Exemplu 1



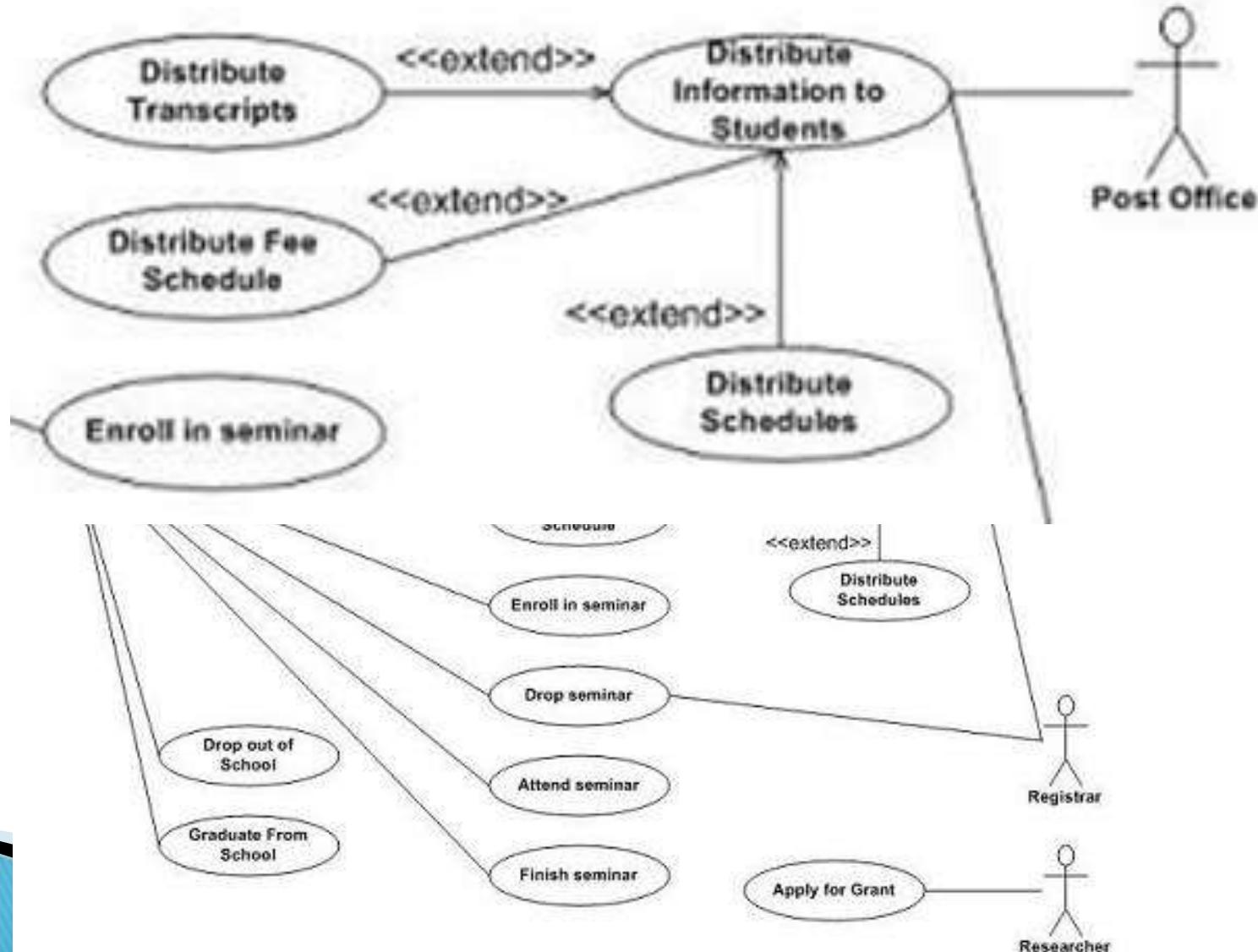
Relația de Generalizare



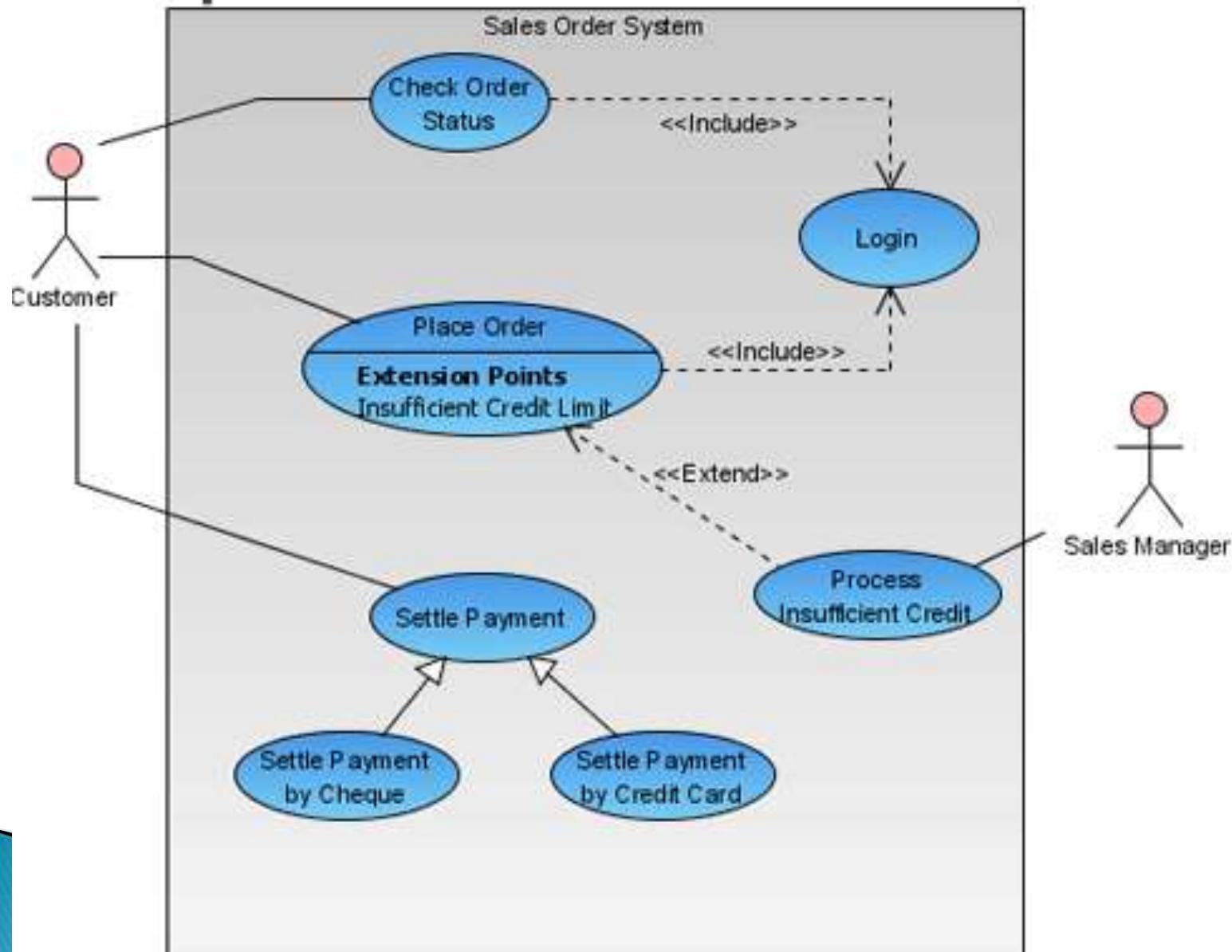
Relația de Dependență

- ▶ Apare între două UseCase-uri.
- ▶ Modelează situațiile în care
 - Un UseCase **folosește** comportamentul definit în alt UseCase (<<include>>)
 - Comportamentul unui UseCase **poate fi extins** de către un alt UseCase (<<extend>>)
- ▶ Notație 

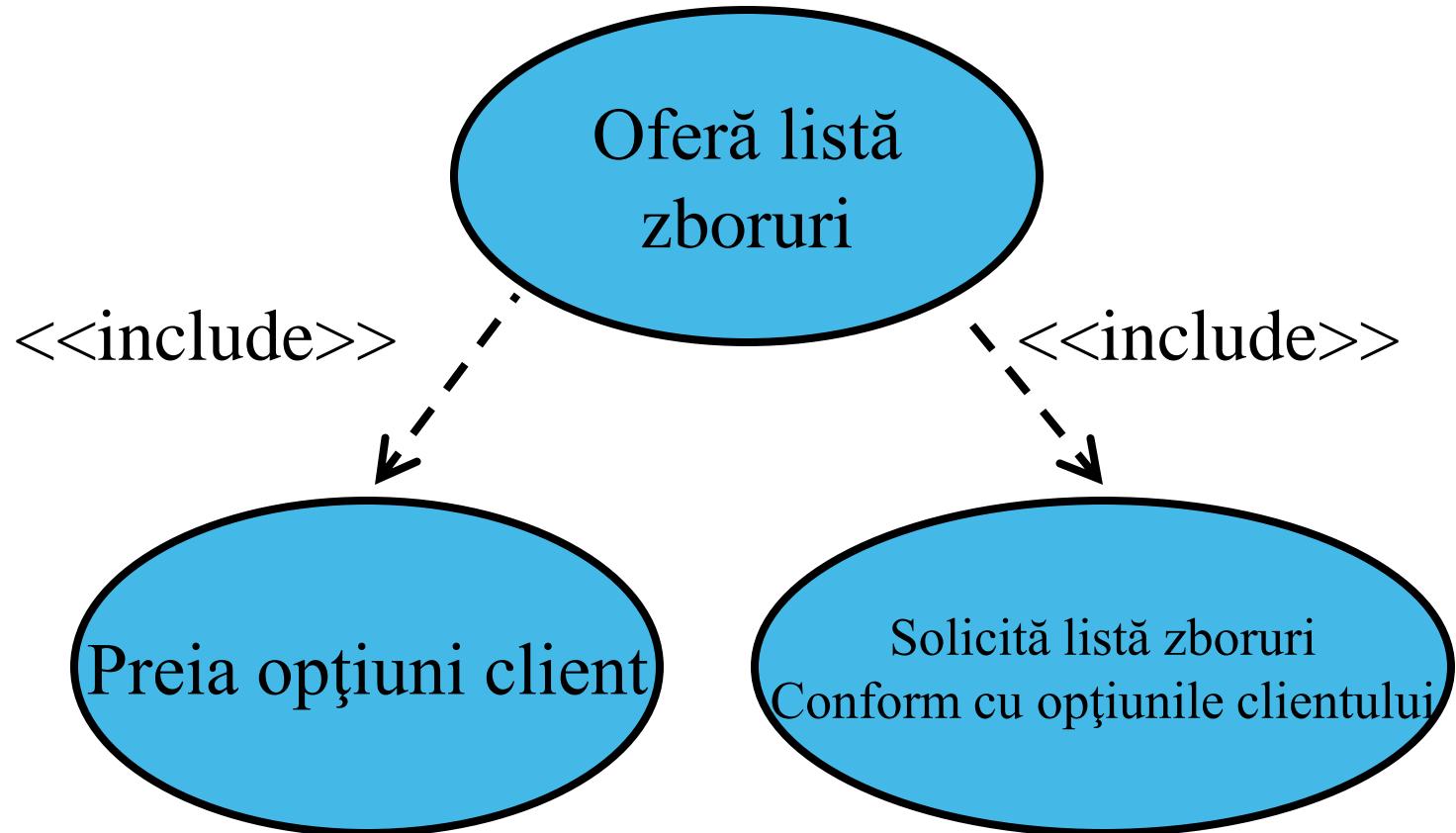
Exemplul 2



Exemplul 3



Exemplu 4



Concepție OO – Recapitulare

► Obiect, Clasă, Instanță

► **Obiect:** Entitate care are: identitate, stare, comportament

- Exemplu: Mingea mea galbenă de tenis, cu diametrul de 10 cm, care sare

► **Clasă:** Descriere a unei multimi de obiecte cu aceleași caracteristici structurale, aceleași caracteristici comportamentale

- Exemplu: mingi care au culoare, diametru, întrebuițare, sar

► **Instanță:** un obiect care aparține unei clase

- Exemplu: Popescu Viorel este un Student

Orientat Obiect 1

- ▶ Este orice abordare ce cuprinde
 - Încapsularea datelor
 - moștenire
 - polimorfism
- ▶ **Încapsularea datelor (exemplu clasa Punct)**
 - Înseamnă punerea la un loc a datelor (atributelor) și a codului (metodelor)
 - Datele pot modificate (doar) prin intermediul metodelor
 - Data hiding: nu ne interesează cum se oferă serviciile, ci doar ca se oferă
 - Dacă se schimbă structura, sau modul de realizare, interfața rămâne neschimbata

Orientat Obiect 2

▶ Moștenire:

- Anumite clase sunt specializări (particularizări) ale altor clase
- O subclasa are (moștenește) caracteristicile superclasei, pe care le poate extinde într-un anume fel
- O instanță a unei clase derivate este în mod automat și o instanță a clasei de bază
- Exemplu (Student – Persoana)

▶ Polimorfism

- Interpretarea semantică unui apel de metoda se face de către cel care primește apelul
- Exemplu: Eu spun unei forme: DESENEAZĂ-TE. Ea, dacă e pătrat trage 4 linii, dacă e cerc, face niște puncte de jur împrejurul centrului
- De altfel, nu mă interesează cine, cum face

Diagrama de clase – Class Diagram

▶ Scop:

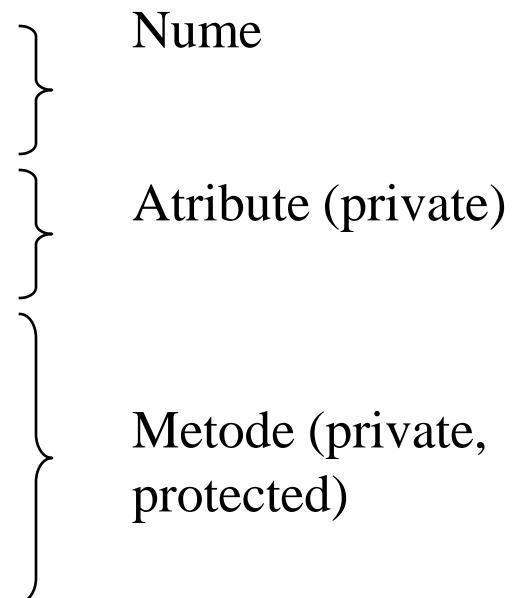
- Modeleză vocabularul sistemului ce trebuie dezvoltat
- Surprinde conexiunile semantice sau interacțiunile care se stabilesc între elementele componente
- Folosită pentru a modela structura unui program

▶ Conține

- Clase/Interfețe
- Obiecte
- Relații (Asociere, Agregare, Generalizare, Dependență)

Clase

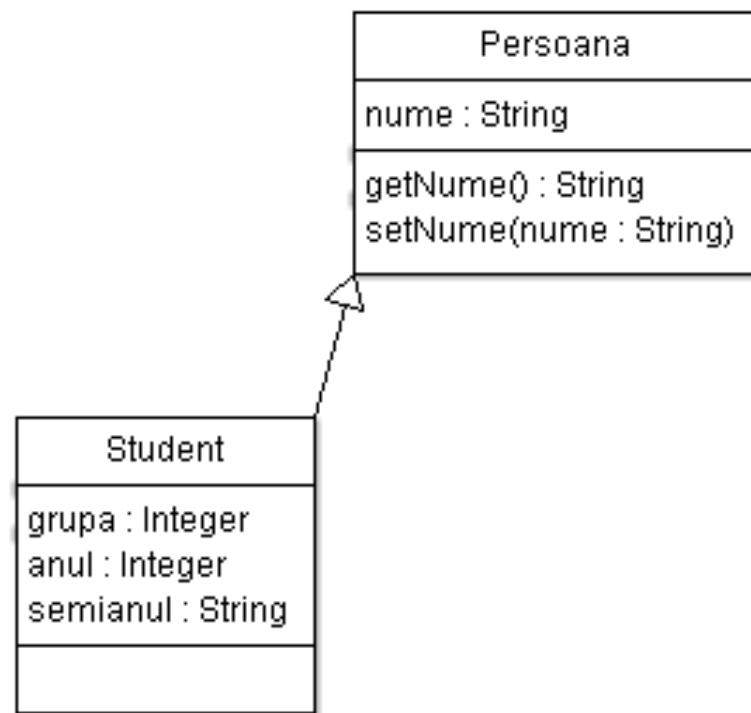
- ▶ Modeleză vocabularul = identifică conceptele pe care clientul sau programatorul le folosește pentru a descrie soluția problemei
- ▶ Elementele unei clase:
 - Nume: identifică o clasa
 - Atribute: proprietăți ale clasei
 - Metode: implementarea unui serviciu care poate fi cerut oricărui obiect din aceeași clasă



Relații – Generalizare – C#

- ▶ Modelează conceptul de moștenire între clase
- ▶ Mai poartă denumirea de relație de tip *is a* (este un/este o)

ArgoUML – Relația de generalizare



Relații – Asociere

- ▶ Exprima o conexiune semantica sau o interacțiune între obiecte aparținând diferitelor clase
- ▶ Pe măsura ce sistemul evoluează noi legaturi între obiecte pot fi create, sau legaturi existente pot distruze
- ▶ O asociere interacționează cu obiectele sale prin intermediul capetelor de asociere
- ▶ Elemente:
 - Nume: descrie relația
 - Capete de asociere
 - Nume = rolul jucat de obiect în relație
 - Multiplicitate = câte instanțe ale unei clase corespund unei singure instanțe ale celeilalte clase



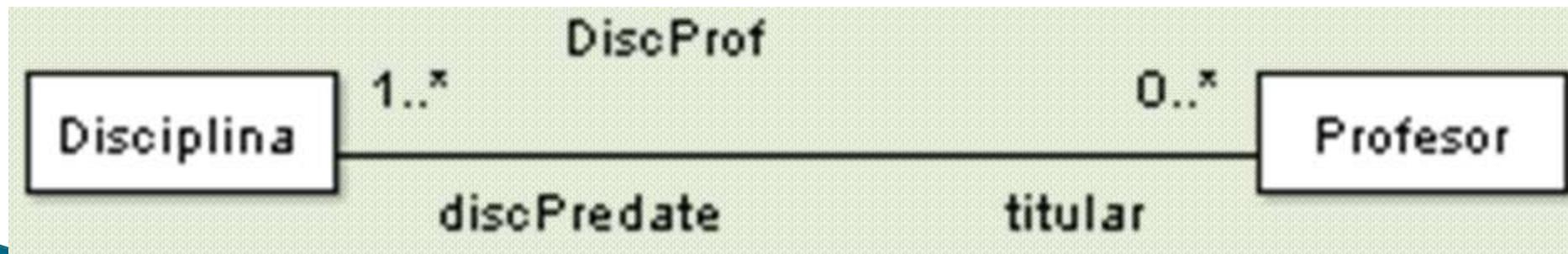
Relația de asociere 1

- ▶ Relația Student – Disciplină
 - **Student**: urmez 0 sau mai multe discipline, cunosc disciplinele pe care le urmez;
 - **Disciplină**: pot fi urmată de mai mulți studenți, nu cunosc studenții care mă urmează



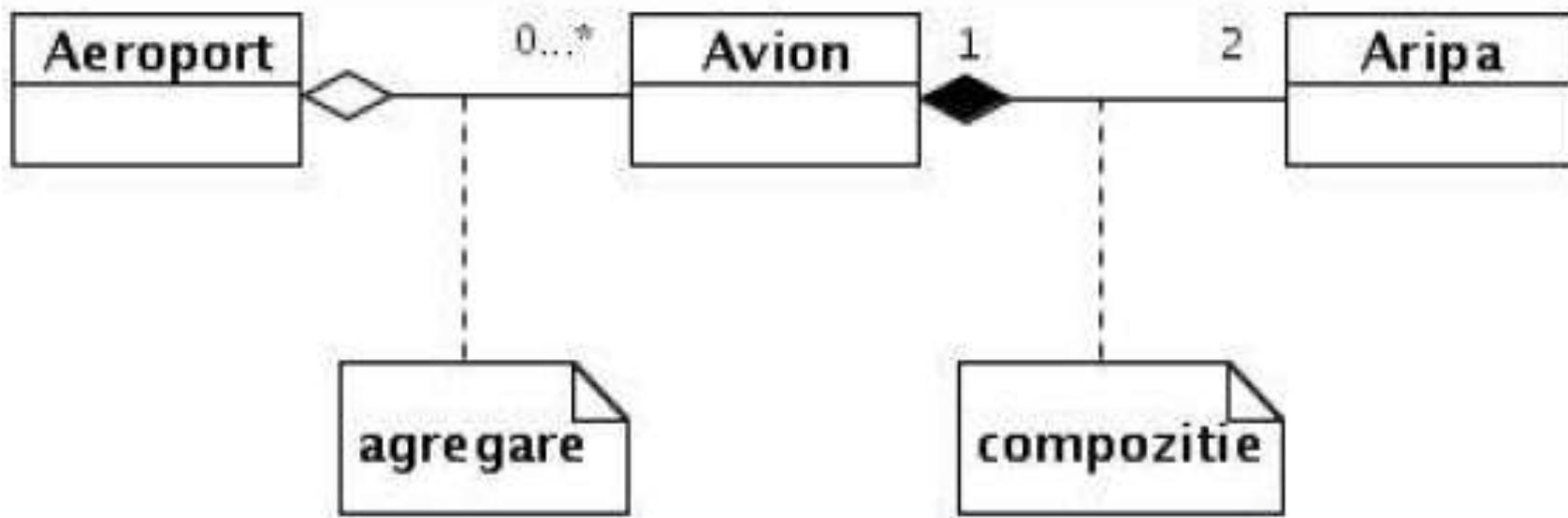
Relația de asociere 2

- ▶ Relația Disciplină – Profesor
 - **Disciplină**: sunt predată de un profesor, îmi cunosc titularul
 - **Profesor**: pot preda mai multe discipline, cunosc disciplinele pe care le predau

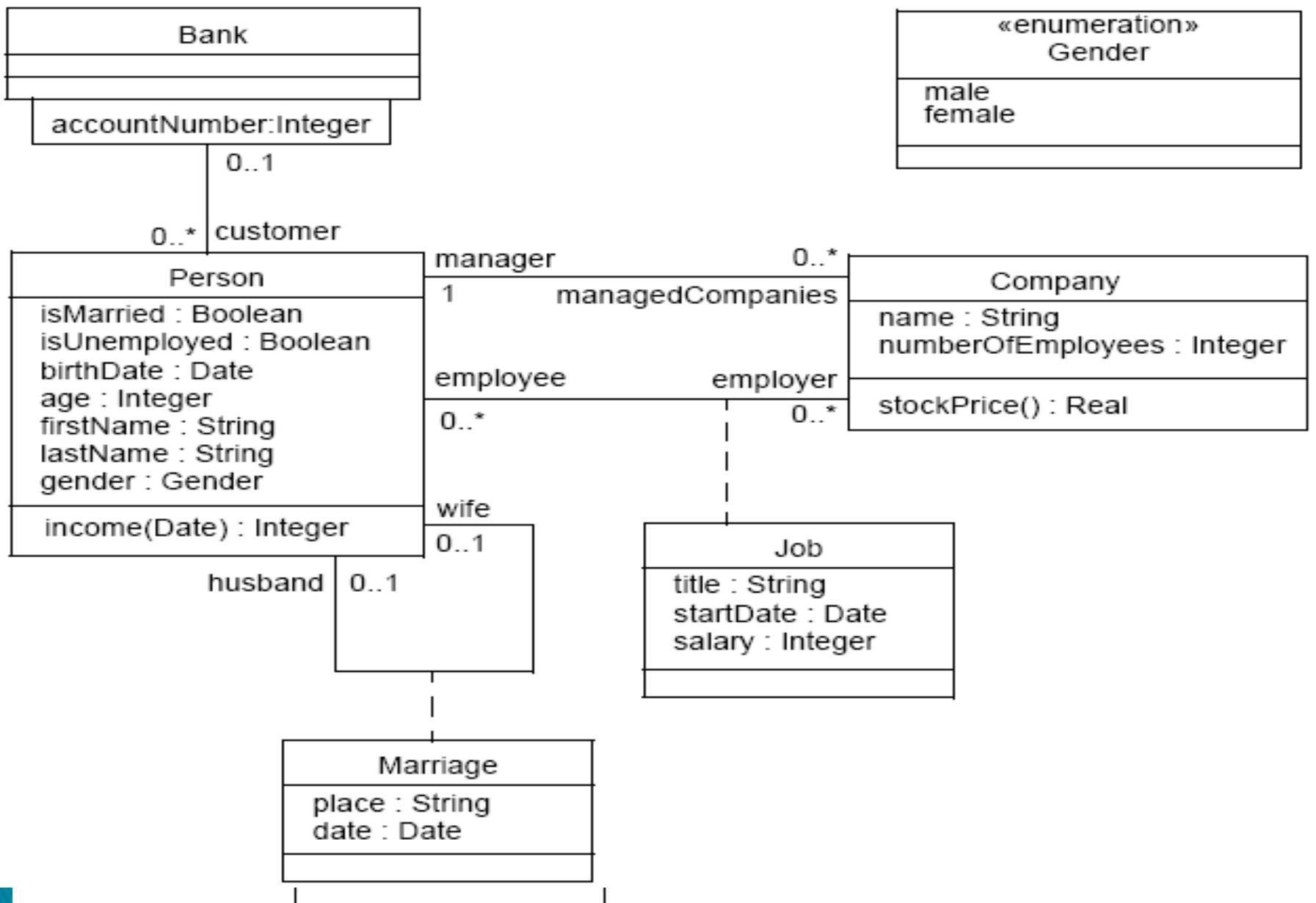


Relații – Agregare

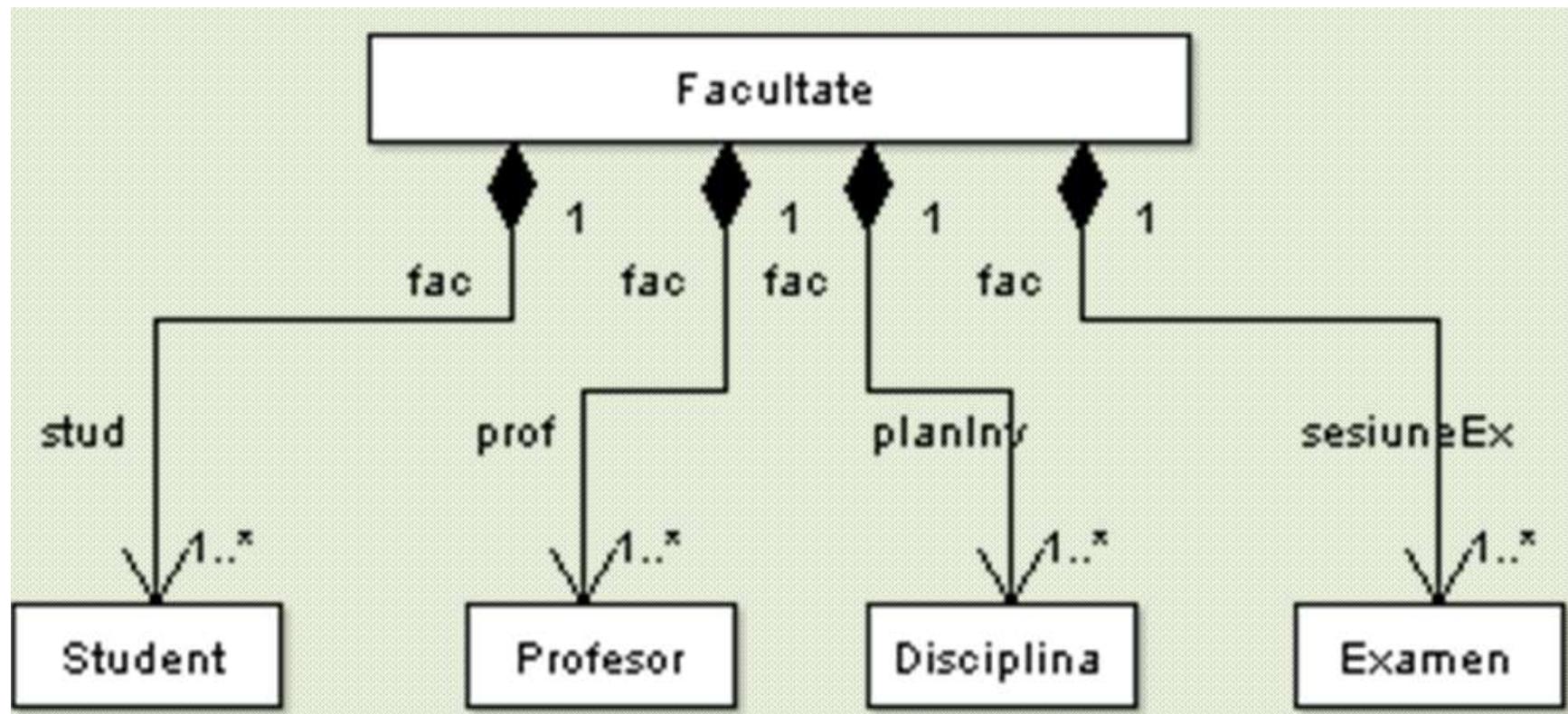
- ▶ Este un caz particular al relației de asociere
- ▶ Modeleză o relație de tip parte–întreg
- ▶ Poate avea toate elementele unei relații de asociere, însă în general se specifică numai multiplicitatea
- ▶ Se folosește pentru a modela situațiile între care un obiect este format din mai multe componente.



Exemplul



Relația de Compoziție (“hasA”)



Studiu de Caz

- ▶ Obținerea Studenților Bursieri

ArgoUML

- ▶ Link: <http://argouml-downloads.tigris.org/argouml-0.26.2/>
- ▶ Varianta “zip” trebuie doar dezarchivată
- ▶ Trebuie să aveți instalat Java
 - În Path sa aveti c:\Program Files\Java\jdk1.6.0_03\bin
 - Variabila JAVA_HOME=c:\Program Files\Java\jdk1.6.0_03\

Bibliografie

- ▶ **OMG Unified Modeling Language™ (OMG UML), Infrastructure, Version 2.2, May 2008,** <http://www.omg.org/docs/ptc/08-05-04.pdf>
- ▶ **ArgoUML User Manual, A tutorial and reference description,** <http://argouml-stats.tigris.org/documentation/printablehtml/manual/argomanual.html>
- ▶ **Ovidiu Gheorghies, Curs IP, Cursurile 3, 4**
- ▶ **Diagrame UML, Regie.ro**

Links

- ▶ OOSE: <http://cs-exhibitions.uni-klu.ac.at/index.php?id=448>
- ▶ ArgoUML: <http://argouml-stats.tigris.org/nonav/documentation/manual-0.22/>
- ▶ Wikipedia

Programarea orientată pe obiect 2

Clonarea obiectelor. Serializarea obiectelor.

Programarea în rețea.

Petic Mircea

petic.mircea@yahoo.com

www.facebook.com/mircea.petic

Clasa Object

- Clasa Object reprezintă implicit superclasa tuturor claselor în Java.
- Variabila de referință de tip Object poate să se refere la obiect al oricărei clase.
- Variabila de referință de tip Object poate să se refere la variabilele de tip tablou aşa cum acestea sunt organizate în formă de clasă.

Metodele clasei Object

Метод	Назначение
Object clone()	Создает новый объект, аналогичный клонируемому объекту
boolean equals(Object объект)	Определяет равнозначность объектов
void finalize()	Вызывается перед тем, как неиспользуемый объект будет удален сборщиком мусора (не рекомендуется в JDK 9)
Class<?> getClass()	Определяет класс объекта во время выполнения
int hashCode()	Возвращает хеш-код, связанный с вызывающим объектом
void notify()	Возобновляет работу потока, ожидающего уведомления от вызывающего объекта
void notifyAll()	Возобновляет работу всех потоков, ожидающих уведомления от вызывающего объекта
String toString()	Возвращает символьную строку, описывающую объект
void wait()	Ожидает выполнения другого потока
void wait(long миллисекунды)	
void wait(long миллисекунды, int наносекунды)	

Clonarea obiectelor

- Video material:

https://www.youtube.com/watch?v=tgfMAfI_1Z4

<https://www.youtube.com/watch?v=mH1EJuyHvrQ>

https://www.youtube.com/watch?v=TER05K_JG8

Serializare și Deserializare – definiții

- **Serializarea** este o metodă ce permite transformarea unui obiect într-o secvență de octeți sau caractere din care să poată fi refăcut ulterior obiectul original.
- Procesul invers, de citire a unui obiect serializat pentru a-i reface starea originală, se numește **deserializare**.
- Într-un cadru mai larg, prin serializare se înțelege procesul de scriere/citire a obiectelor.
- Tipurile primitive pot fi de asemenea serializate.

Utilitatea serializării

- Asigură un mecanism simplu de utilizat pentru salvarea și restaurarea a datelor.
- Permite **persistența obiectelor**, ceea ce înseamna că durata de viață a unui obiect nu este determinată de execuția unui program în care acesta este definit - obiectul poate exista și între apelurile programelor care îl folosesc.

Utilitatea serializării

- **Compensarea diferențelor între sisteme de operare** - transmiterea unor informații între platforme de lucru diferite se realizează unitar.
- **Transmiterea datelor în rețea** – Aplicațiile ce rulează în rețea pot comunica între ele folosind fluxuri pe care sunt trimise, respectiv recepționate obiecte serializate.

Utilitatea serializării

- **RMI (Remote Method Invocation)** - este o modalitate prin care metodele unor obiecte de pe o altă mașină pot fi apelate ca și cum acestea ar exista local pe mașina pe care rulează aplicația.
- **Java Beans** - sănătătoare ce pot fi utilizate în medii vizuale de dezvoltare a aplicațiilor.

Serializarea tipurilor primitive

- Fluxurile care realizează:
 - DataOutputStream și DataInputStream
 - ObjectOutputStream și ObjectInputStream
- Implementează interfețele DataInput și DataOutput.
- Metodă pentru scrierea datelor primitive și sirului de caractere: **writeTipPrimitiv();**
- Metodă pentru citirea datelor primitive și sirului de caractere: **readTipPrimitiv();**

Exemplu de serializare

```
FileOutputStream fos = new  
FileOutputStream("test.dat");  
DataOutputStream out = new  
DataOutputStream(fos);  
out.writeInt(12345);  
out.writeDouble(12.345);  
out.writeBoolean(true);  
out.writeUTF("Sir de caractere");  
out.flush();  
fos.close();
```

Exemplu de deserializare

```
FileInputStream fis = new  
    FileInputStream("test.dat");  
  
DataInputStream in = new  
    DataInputStream(fis);  
  
int i = in.readInt();  
double d = in.readDouble();  
boolean b = in.readBoolean();  
String s = in.readUTF();  
fis.close();
```

Serializarea obiectelor

- Serializarea obiectelor se realizează prin intermediul fluxurilor definite de clasele
 - **ObjectOutputStream** (pentru salvare)
 - **ObjectInputStream** (pentru restaurare).
- Metodele pentru serializarea obiectelor sînt:
 - **writeObject**, pentru scriere
 - **readObject**, pentru restaurare.

Clasa ObjectOutputStream

```
ObjectOutputStream out = new  
ObjectOutputStream(fluxPrimitiv);  
out.writeObject(referintaObject);  
out.flush();  
fluxPrimitiv.close();
```

Exemplu: serializarea obiectelor

```
FileOutputStream fos = new  
    FileOutputStream("test.ser");  
  
ObjectOutputStream out = new  
    ObjectOutputStream(fos);  
  
out.writeObject("Ora curenta:");  
out.writeObject(new Date());  
  
out.flush();  
fos.close();
```

Clasa ObjectOutputStream

```
ObjectInputStream in = new  
    ObjectInputStream(fluxPrimativ);  
Object obj = in.readObject();  
//sau  
TipReferinta ref =  
    (TipReferinta)in.readObject();  
fluxPrimativ.close();
```

Exemplu: deserializarea obiectelor

```
FileInputStream fis = new  
    FileInputStream("test.ser");  
  
ObjectInputStream in = new  
    ObjectInputStream(fis);  
  
String mesaj =  
    (String)in.readObject();  
  
Date data = (Date)in.readObject();  
  
fis.close();
```

Ora curentă: Mon Oct 27 23:01:49 EET 2014

Atenție

- Date date = in.readObject(); // gresit
- Date date = (Date)in.readObject(); // corect

Obiecte serializabile

- Un obiect este serializabil dacă și numai dacă clasa din care face parte implementează interfața **Serializable**.
- Interfața **Serializable** nu conține nici o declarație de metodă sau constantă, singurul ei scop fiind de a identifica clasele ale căror obiecte sănăt serializable.

Declararea claselor ale căror instanțe trebuie să fie serializate este

```
public class ClasaSerializabila  
    implements Serializable {  
    // Corpul clasei  
}
```

- Orice subclasă a unei clase serializable este la rîndul ei serializabilă, întrucît implementează indirect interfața **Serializable**.

Controlul serializării

- Există cazuri când dorim ca unele variabile membre ale unui obiect să nu fie salvate automat în procesul de serializare.
- Chiar declarate private în cadrul clasei aceste cîmpuri participă la serializare.
- Pentru ca un cîmp să nu fie salvat în procesul de serializare el trebuie declarat cu modificatorul **transient** și trebuie să fie ne-static.

Exemple

```
transient private double temp;
```

```
// Ignorata la serializare
```

Modificatorul **static** anulează efectul
modificatorului **transient**.

```
static transient int N;
```

```
// Participa la serializare
```

Exemplu

```
import java .io.*;
class A {
    int x=1;
}
class B implements Serializable {
    int y=2;
}
public class Test2 implements Serializable {
    A a = new A(); // Exceptie
    B b = new B(); // DA
    public String toString () {
        return a.x + ", " + b.y;
    }
}
```

```
import java .io .*;
class C {
    int x=0;
    // Obligatoriu constructor fara argumente
}
class D extends C implements Serializable {
    int y=0;
}
public class Test3 extends D {
    public Test3 () {
        x = 1; // NU
        y = 2; // DA
    }
    public String toString () {
        return x + ", " + y;
    }
}
```

Video materiale serializare

- Materiale video

https://www.youtube.com/watch?v=vz6nyRXCu_s0

<https://www.youtube.com/watch?v=dBcqizwOWLg>

<https://www.youtube.com/watch?v=Kta6v6AqAWk>

Programarea în rețea

Introducere

- Programarea în rețea implică trimiterea de mesaje și date între aplicații ce rulează pe calculatoare aflate într-o rețea locală sau conectate la Internet.
- Pachetul care oferă suport pentru scrierea aplicațiilor de rețea este **java.net**.
- Noțiuni fundamentale: *protocol, adresă IP, port, socket*.

Protocol

- Un protocol reprezintă o convenție de reprezentare a datelor folosită în comunicarea între două calculatoare.
- Două dintre cele mai utilizate protocoale sînt:
 - **TCP** (*Transport Control Protocol*) - este un protocol ce furnizează un flux sigur de date între două calculatoare aflate în rețea.
 - **UDP** (*User Datagram Protocol*) - este un protocol bazat pe pachete independente de date, numite datagrame, trimise de la un calculator către altul fără a se garanta în vreun fel ajungerea acestora la destinație sau ordinea în care acestea ajung.

Adresă IP

- Orice calculator conectat la Internet este identificat în mod unic de adresa sa **IP** (**IP** este acronimul de la *Internet Protocol*).
- Un număr reprezentat pe 32 de biți (4 octeți) - 193.231.30.131 - *adresa IP numerică*.
- o adresa *IP simbolică* - thor.infoiasi.ro
- Clasa Java care reprezintă noțiunea de adresă IP este InetAddress.

Ce este un port?

- Un calculator are în general o singură legătură fizică la rețea.
- Pe un calculator pot exista concurent mai multe procese care au stabilite conexiuni în rețea, așteptînd diverse informații.
- Identificarea proceselor se realizează prin intermediul **porturilor**.
- Un port este un număr pe 16 biți (între 0 și 65535) care identifică în mod unic procesele care rulează pe o anumită mașină.
- numerele cuprinse între 0 și 1023 fiind însă rezervate unor servicii sistem, de aceea nu se folosesc.

Clasele din `java.net` permit comunicarea între procese folosind protocolele *TCP* și *UDP* și sănătate prezentate în tabelul de mai jos:

TCP	UDP
URL	DatagramPacket
URLConnection	DatagramSocket
Socket	MulticastSocket
ServerSocket	

URL

- Termenul **URL** este acronimul pentru *Uniform Resource Locator* și reprezintă o referință (adresă) la o resursă aflată pe Internet.
- Un URL are două componente principale:
 - *Identifierul protocolului folosit* (http, ftp, etc);
 - *Numele resursei referite*.
- Clasa care permite lucrul cu **URL**-uri este *java.net.URL*.

Un obiect de tip **URL** poate fi folosit pentru:

- Aflarea informațiilor despre resursa referită (numele calculatorului gazdă, numele fișierului, protocolul folosit. etc).
- Citirea printr-un flux a continutului fișierului respectiv.
- Conectarea la acel URL pentru citirea și scrierea de informații.

Crearea obiectului de tip URL

```
try {  
    URL adresa = new URL("http://xyz.abc");  
}  
} catch (MalformedURLException e) {  
    System.err.println("URL invalid!\n"+e);  
}
```

Citirea conținutului unui URL

```
String adresa = "http://meteo.md";
BufferedReader br = null ;
try {
    URL url = new URL(adresa);
    InputStream in = url.openStream();
    br = new BufferedReader( new InputStreamReader(in));
    String linie;
    while (( linie = br.readLine()) != null ) {
        // Afisam linia citita
        System.out.println (linie);
    }
}
catch ( MalformedURLException e) {
    System.err.println("URL invalid !\n" + e);
}
finally {
    br.close();
}
```

Socket-uri

- Un socket (soclu) este o abstracțiu software folosită pentru a reprezenta fiecare din cele două ”capete” ale unei conexiuni între două procese ce rulează într-o rețea.
- Fiecare socket este atașat unui port astfel încât să poată identifica unic programul căruia îi sînt destinate datele.
- Socket-urile sunt de două tipuri:
 - **TCP**, implementate de clasele *Socket* și *ServerSocket*;
 - **UDP**, implementate de clasa *DatagramSocket*.

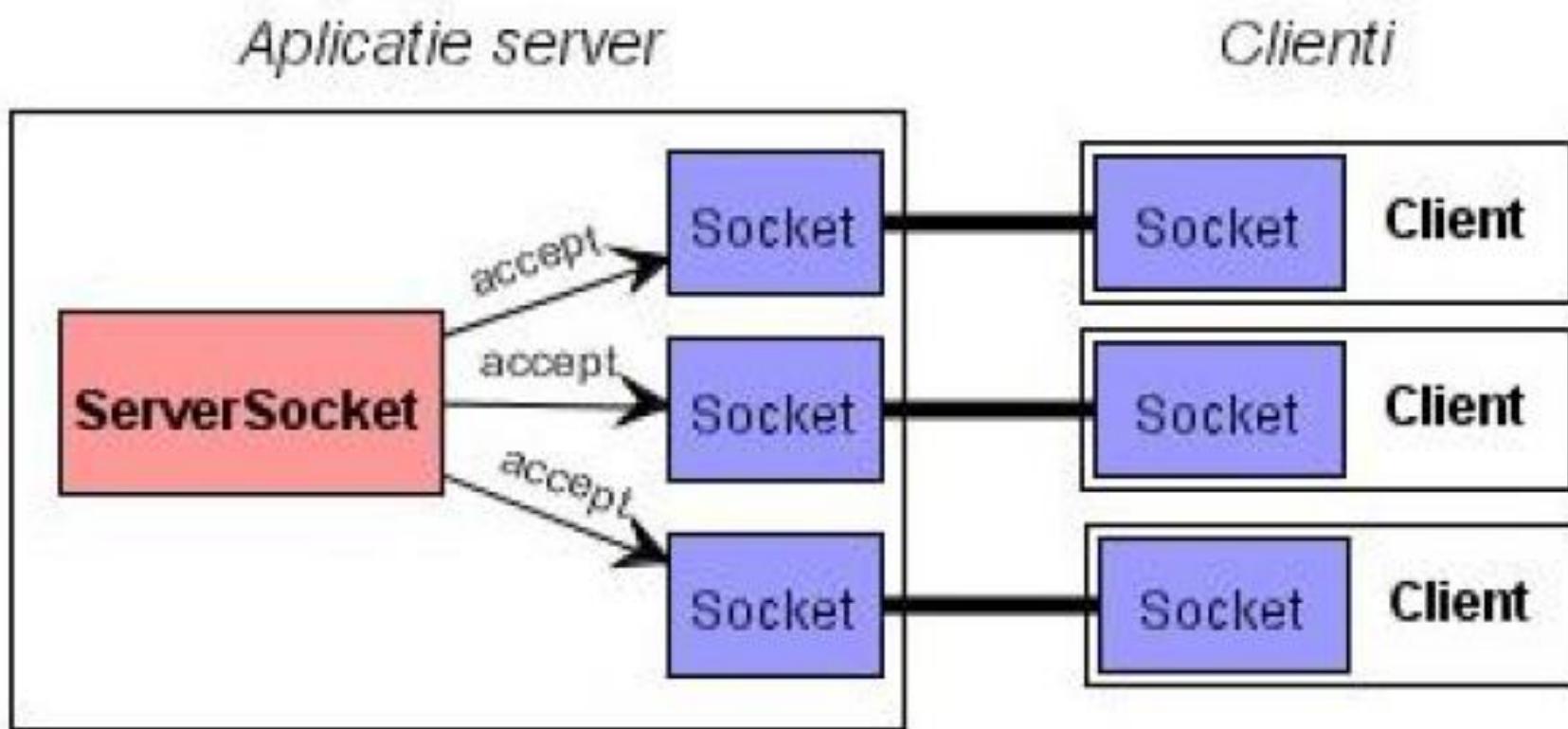
Aplicații client-server

- În acest model aplicația este formată din două categorii distincte de programe numite servere, respectiv clienți.
- **Programele de tip server** sănt cele care oferă diverse servicii eventualilor clienți, fiind în stare de așteptare atîta vreme cît nici un client nu le solicită serviciile.
- **Programele de tip client** sănt cele care inițiază conversația cu un server, solicitînd un anumit serviciu.
- *Un server trebuie să fie capabil să trateze mai mulți clienți simultan și, din acest motiv, fiecare cerere adresată serverului va fi tratată într-un fir de execuție separat.*

Comunicarea prin conexiuni (TCP/IP)

- Legătura între un client și un server se realizează prin intermediul a două obiecte de tip Socket, cîte unul pentru fiecare capăt al "canalului" de comunicație dintre cei doi.
- La nivelul clientului crearea socketului se realizează specificînd adresa IP a serverului și portul la care rulează acesta, constructorul uzual folosit fiind:
Socket(IetAddress address, int port)
- La nivelul serverului, acesta trebuie să creeze întîi un obiect de tip **ServerSocket (int port)**.
- Metoda clasei **ServerSocket** care așteaptă "ascultă" rețeaua este **accept**.

Comunicarea prin conexiuni (TCP/IP)



Comunicarea prin conexiuni (TCP/IP)

- Pentru fiecare din cele două socketuri deschise pot fi create apoi două fluxuri pe octeți pentru citirea (**getInputStream**), respectiv scrierea datelor (**getOutputStream**).
- În funcție de specificul aplicației acestea pot fi perechile:
 - **BufferedReader**, **BufferedWriter** și **PrintWriter** - pentru comunicare prin intermediul sirurilor de caractere;
 - **DataInputStream**, **DataOutputStream** - pentru comunicare prin date primitive;
 - **ObjectInputStream**, **ObjectOutputStream** - pentru comunicare prin intermediul obiectelor;

Structura generală a unui server bazat pe conexiuni este:

1. Creează un obiect de tip **ServerSocket** la un anumit port

```
while (true) {
```

2. Așteaptă realizarea unei conexiuni cu un client, folosind metoda **accept**;
(va fi creat un obiect nou de tip **Socket**)

3. Tratează cererea venită de la client:

- 3.1 Deschide un flux de intrare și primește cererea

- 3.2 Deschide un flux de ieșire și trimite răspunsul

- 3.3 Închide fluxurile și socketul nou creat

```
}
```

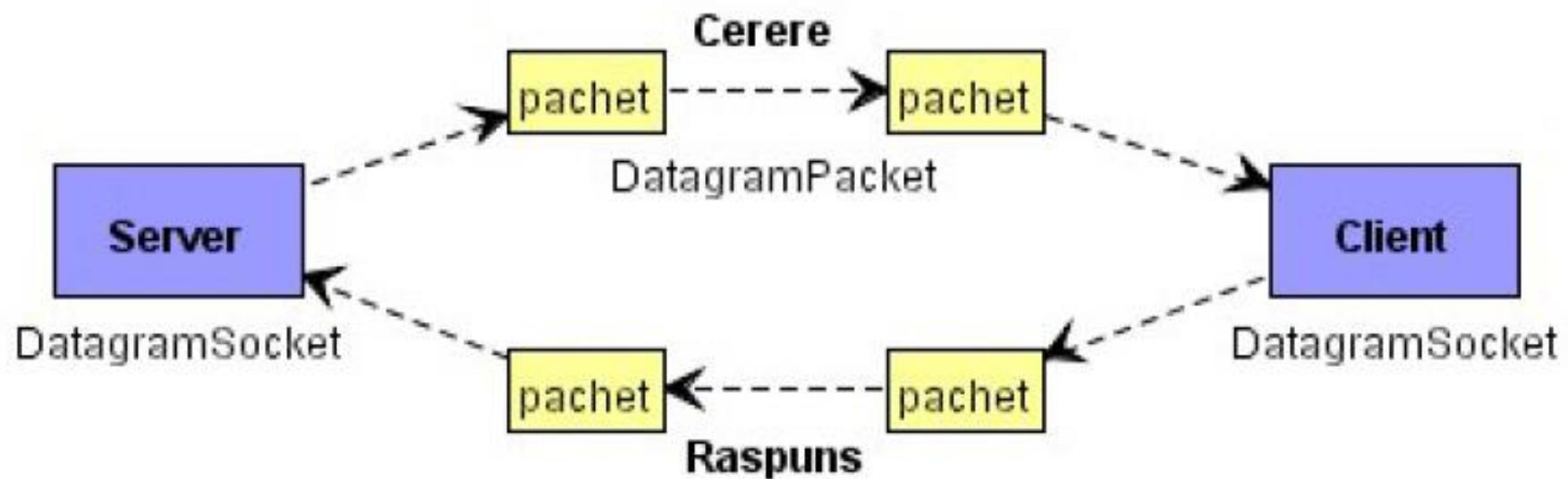
Structura generală a unui client bazat pe conexiuni este:

1. Citește sau declară adresa IP a serverului și portul la care acesta rulează;
2. Creează un obiect de tip **Socket** cu adresa și portul specificate;
3. Comunica cu serverul:
 - 3.1 Deschide un flux de ieșire și trimite cererea;
 - 3.2 Deschide un flux de intrare și primește răspunsul;
 - 3.3 Închide fluxurile și socketul creat;

Comunicarea prin datagrame

- În acest model nu există o conexiune permanentă între client și server prin intermediul căreia să se realizeze comunicarea.
- Clientul trimite cererea către server prin intermediul unuia sau mai multor pachete de date independente, serverul le recepționează, extrage informațiile conținute și returnează răspunsul tot prin intermediul pachetelor.
- Un astfel de pachet se numește datagramă și este reprezentat printr-un obiect din clasa **DatagramPacket**.

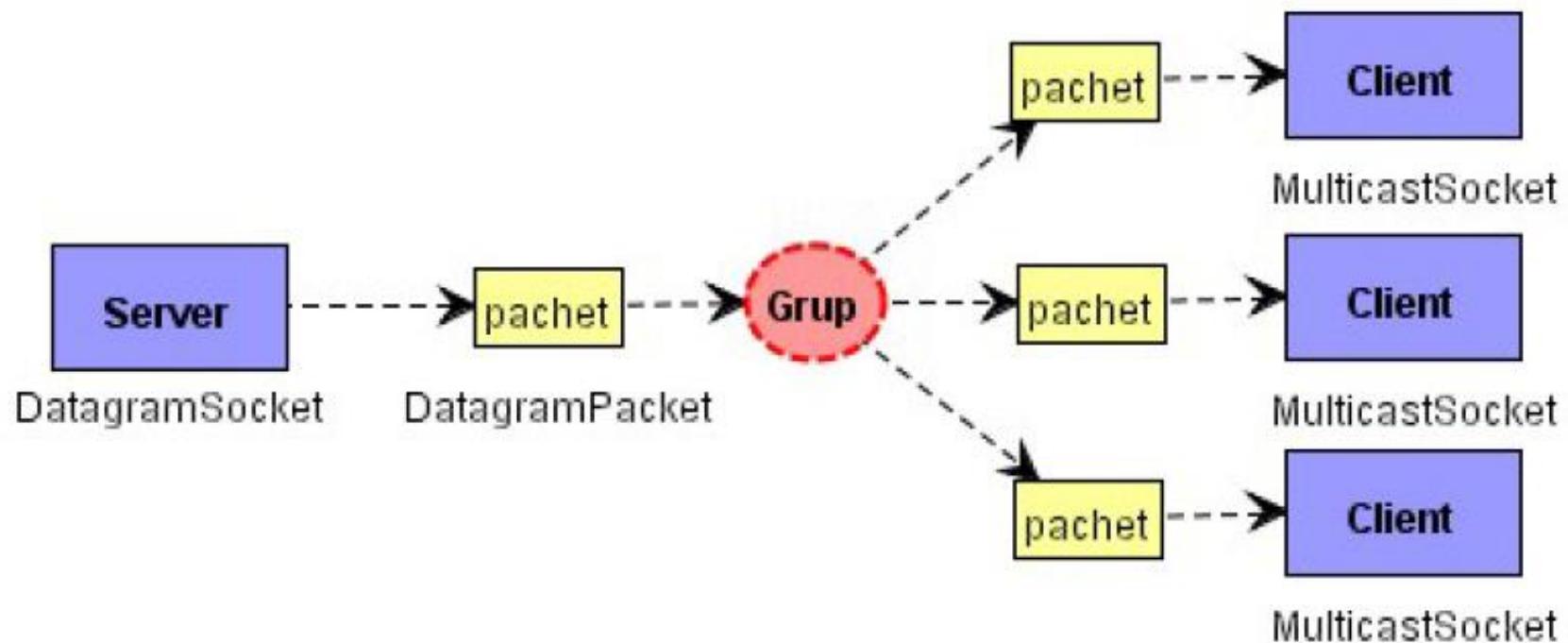
Comunicarea prin datagrame



Comunicarea prin datagrame

- După crearea unui pachet procesul de trimitere și primire a acestuia implică apelul metodelor **send** și **receive** ale clasei **DatagramSocket**.
- În cazul în care refolosim pachete, putem schimba conținutul acestora cu metoda **setData**, precum și adresa la care le trimitem prin **setAddress**, **setPort** și **setSocketAddress**.
- Extragerea informațiilor conținute de un pachet se realizează prin metoda **getData** din clasa **DatagramPacket**.

Trimiterea de mesaje către mai mulți clienți



Programarea orientată pe obiect 2

Patternul MVC. Componente si arhitectură.

Petic Mircea

petic.mircea@yahoo.com

www.facebook.com/mircea.petic

Definiția patternului de programare

- Pattern-ul reprezintă o metodă eficientă de rezolvare a unor probleme de proiectare tipice, în particular, proiectarea programelor pentru calculatoare.
- Pattern-ul nu reprezintă varianta finală a proiectului, care poate fi direct trecut în cod, ci, mai degrabă, un eşantion sau o descriere a metodei de rezolvare a problemei, care ulterior poate fi utilizat în mai multe situații.

Patternul MVC

- **Model – View – Controller**
- **Model** – codul de program care face tot ceea de ce aplicația a fost elaborată. – **logica de business**;
- **View** - conține cod care controlează lucru cu interfața grafică de exemplu Windows, pagini, mesaje etc. – **interfața cu utilizatorul**;
- **Controller** – Orice acțiuni ale utilizatorilor vizând schimbarea modelului ar trebui procesate aici.

Acesta este principiul construirii arhitecturii unei aplicații mari în care este împărțită în trei părți.

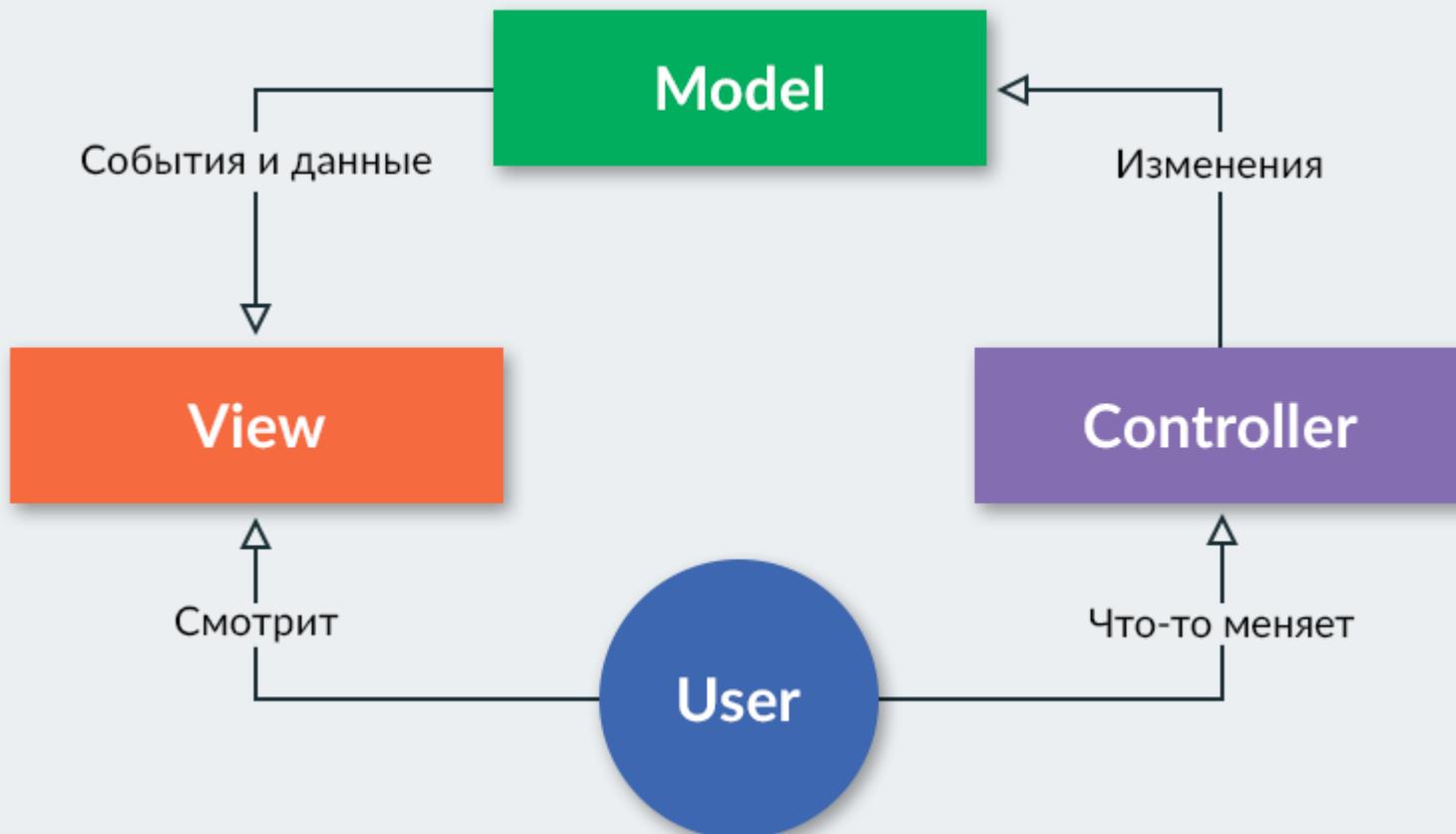
Trei caracteristici ale MVC

- **logica programului (Model),**
- **mecanism pentru afișarea tuturor datelor programului către utilizator (View),**
- **procesarea intrărilor / acțiunilor utilizatorului (Controller).**

MVC conține trei grupuri de clase în care:

- fiecare parte are propria destinație;
- legăturile dintre clasele unui grup sunt destul de puternice;
- conexiunile dintre grupuri sunt destul de slabe;
- metodele de interacțiune dintre grupuri sunt destul de mult reglementate.

Schema de lucru pentru MVC



Caracteristicile grupului Model

- **Model** este cea mai independentă parte a sistemului.
- **Model** nu depinde de **View & Controller**.
- Modelul nu poate utiliza clase din secțiuni **View & Controller**

Caracteristicile grupului **View**

- **View** nu poate schimba modelul.
- Clasele **View** pot accesa modelul pentru date sau evenimente

Caracterisiticile grupului **Controller**

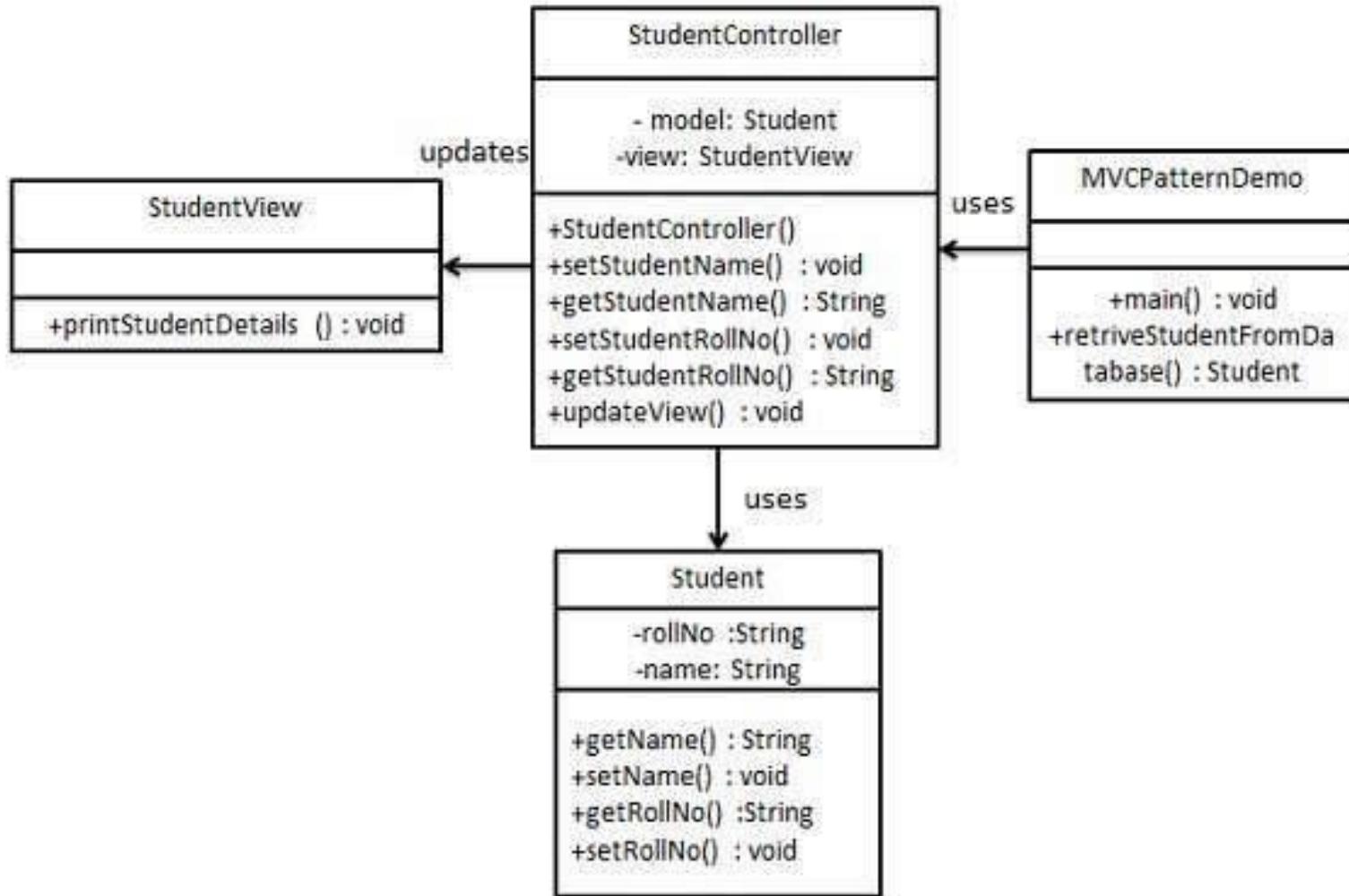
- **Controller** nu se ocupă cu afişarea datelor.
- **Controller-ul** procesează acţiunile utilizatorului și schimbă modelul în conformitate cu acestea.

Exemplu simplu de program cu modelul MVC

Exemplu

- Vom crea un obiect *Student* care să acționeze ca **Model**.
- *StudentView* va fi o clasă **View** care poate tipări detaliile studentului pe consolă,
- *StudentController* este clasa **Controller** responsabilă pentru stocarea datelor în obiectul *Student* și actualizarea vizualizării *StudentView* în consecință.

Diagrama UML a aplicației



Student.java

```
public class Student {  
    private String rollNo;  
    private String name;  
  
    public String getRollNo() {  
        return rollNo;  
    }  
  
    public void setRollNo(String rollNo) {  
        this.rollNo = rollNo;  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    public void setName(String name) {  
        this.name = name;  
    }  
}
```

StudentView.java

```
public class StudentView {  
    public void printStudentDetails(String studentName,  
String studentRollNo){  
        System.out.println("Student: ");  
        System.out.println("Name: " + studentName);  
        System.out.println("Roll No: " + studentRollNo);  
    }  
}
```

StudentController.java

```
public class StudentController {  
    private Student model;  
    private StudentView view;  
  
    public StudentController(Student model, StudentView view){  
        this.model = model;  
        this.view = view;  
    }  
  
    public void setStudentName(String name){  
        model.setName(name);  
    }  
  
    public String getStudentName(){  
        return model.getName();  
    }  
  
    public void setStudentRollNo(String rollNo){  
        model.setRollNo(rollNo);  
    }  
  
    public String getStudentRollNo(){  
        return model.getRollNo();  
    }  
  
    public void updateView(){  
        view.printStudentDetails(model.getName(), model.getRollNo());  
    }  
}
```

MVCPatternDemo.java

```
public class MVCPatternDemo {  
    public static void main(String[] args) {  
  
        //fetch student record based on his roll no from the database  
        Student model = retriveStudentFromDatabase();  
  
        //Create a view : to write student details on console  
        StudentView view = new StudentView();  
  
        StudentController controller = new StudentController(model, view);  
  
        controller.updateView();  
  
        //update model data  
        controller.setStudentName("John");  
  
        controller.updateView();  
    }  
  
    private static Student retriveStudentFromDatabase(){  
        Student student = new Student();  
        student.setName("Robert");  
        student.setRollNo("10");  
        return student;  
    }  
}
```

Rezultatul execuției

Student:

Name: Robert

Roll No: 10

Student:

Name: John

Roll No: 10

Linkuri utile:

- [https://www.tutorialspoint.com/design pattern/mvc pattern.htm](https://www.tutorialspoint.com/design_pattern/mvc_pattern.htm)
- <https://javarush.ru/quests/lectures/questcollections.level06.lecture01>
- https://www.youtube.com/watch?v=fBM78NJB_S_A

Programarea orientată pe obiect II

Prelegerea nr. 12

Elaborarea aplicațiilor JavaFX.

Petic Mircea

mircea.petic@yahoo.md

www.facebook.com/mircea.petic

Momente istorice

- Primele instrumente de interfață grafică – **Graphic User Interface (GUI)** – a fost implementat în Java cu ajutorul bibliotecii AWT (**Abstract Window Toolkit**).
- Componentele AWT sunt definite nu de instrumentele Java dar de resursele platformei.
- În 1997 a apărut biblioteca de componente Swing (în componenta JFC – Java Foundation Classes). Inițial ca bibliotecă separată de Java, apoi în versiunea JDK1.2 intră în componența Java.

Characteristicile Swing

- Componentele sunt scrise pentru Java și nu arată în dependență de platformă.
- Elementele Swing arată în toate SO la fel.
- Logica și interfața sunt separate în Swing – MVC (model-view-controler).
 - **Model** – informația despre starea componentei;
 - **Prezentarea** – cum arată componenta;
 - **Controlul** – reacția componentei la acțiunea utilizatorului.

Structura swing

- Tehnologia Swing face parte dintr-un proiect mai amplu numit JFC (Java Foundation Classes) care pune la dispoziție o serie întreagă de facilități pentru scrierea de aplicații cu o interfață grafică mult îmbogățită funcțional și estetic față de vechiul model AWT. În JFC sunt incluse următoarele:
- **Componente Swing** - sunt componente ce înlocuiesc și în același timp extind vechiul set oferit de modelul AWT.
- **Look-and-Feel** (uite și simte) – permite schimbarea înfățișării și a modului de interacțiune cu aplicația în funcție de preferințele fiecărui. Același program poate utiliza diverse moduri Look-and-Feel, cum ar fi cele standard Windows, Mac, Java, sau altele oferite de diversi dezvoltatori, acestea putând fi interschimbată de către utilizator și la momentul execuției.

Componente și container

- Toate container-e sunt concomitent și componente;
- Componentă este un element vizual independent, dar containerul poate conține alte componente;
- Containerul este o componentă specială pentru a conține alte componente.
- Pentru a vizualiza o componentă ea trebuie inclusă într-un container.
- Pentru a plasa o componentă trebuie să fie plasat cel puțin un container.
- Un container poate conține alt container.

Exemple de компонент:

JApplet (не рекомендуется в JDK 9)	JButton	JCheckBox	JCheckBoxMenuItem
JColorChooser	JComboBox	JComponent	JDesktopPane
JDialog	JEditorPane	JFileChooser	JFormattedTextField
JFrame	JInternalFrame	JLabel	JLayer
JLayeredPane	JList	JMenu	JMenuBar
JMenuItem	JOptionPane	JPanel	JPasswordField
JPopupMenu	JProgressBar	JRadioButton	JRadioButtonMenuItem
JRootPane	JScrollBar	JScrollPane	JSeparator
JSlider	JSpinner	JSplitPane	JTabbedPane
JTable	JTextArea	JTextField	JTextPane
JToggleButton	JToolBar	JToolTip	JTree
JViewport	JWindow		

Prima fereastră

- O fereastră este un obiect al clasei **JFrame**.
- Clasa **JFrame** este reținută în pachetul **javax.swing**.

Exemplu 1.

```
import javax.swing.*;  
public class pv {  
    public static void main(String args[]){  
        JFrame fer=new JFrame("Prima fereastra!");  
        fer.setSize(200,300);  
        fer.setLocation(300,400);  
        fer.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
        fer.setVisible(true);  
    }  
}
```

Metodele utilizate

- **JFrame()** – constructor.
- **void setSize(int width, int height)** – stabilește lungimea și lățimea ferestrei.
- **void setLocation(int x, int y)** – stabilește poziția unde va fi afișat colțul din stînga sus al ferestrei, în raport cu colțul din stînga sus al ecranului.
- **void setResizable(boolean ac)** – dacă parametrul este false nu se pot modifica dimensiunile ferestrei.

Metodele utilizate

- **void setDefaultCloseOperation(int a)**
– stabilește ce se întimplă atunci când se închide fereastra.
- **setVisible(boolean x)** – stabilește dacă fereastra este vizibilă (apare pe ecran) sau nu (deși există, nu este afișată).

Exemplu 2. Clasa utilizator Fereastra

```
import javax.swing.*;
class Fereastra extends JFrame {
    Fereastra (String Nume, int lat, int inalt, int dreapta, int stinga)
    {
        super(Nume);
        setSize(lat, inalt);
        setLocation(dreapta, stinga);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setVisible(true);
    }
}
public class pv{
    public static void main(String args[]){
        Fereastra f=new Fereastra("Prima", 100,200,25,100);
    }
}
```



Atașarea componentelor ferestrei

- Structura care reține referințele către obiectele care se află pe fereastă este un obiect al clasei **Container**.
- Un obiect de tip **JFrame** sau dintr-o subclasă a lui **JFrame** conține datele membru și metodele clasei **Container**.
- Accesul la containerul unei ferestre se face utilizând o metodă a clasei **JFrame**:
Container getContentPane();

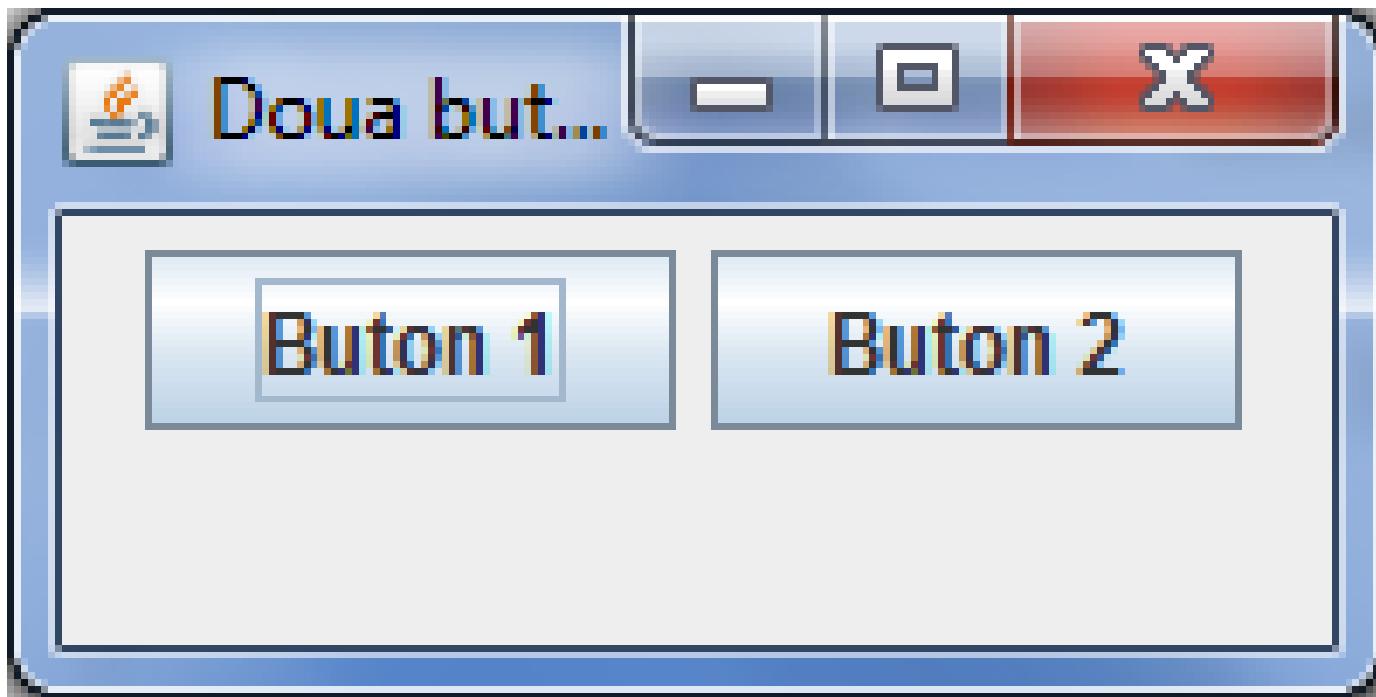
Pornind de la referinta getContentPane se pot realiza două lucruri:

- Pot fi adăugate ferestrei componentele dorite.
 - Pentru aceasta se folosește metoda add() a clasei Container:
 - Component add(Component comp) – adaugă o componentă a ferestrei.
 - Toate componentele sunt derivate din clasa Component.
- Pot fi aranjate în fereastră componentele adăugate.
 - Mecanismul din Java presupune existența gestionarilor de poziționare.
 - ✓ Pentru a ataşa unui container un gestionar de poziționare, se utilizează metoda clasei **Container** numită **setLayout()**:
 - ✓ **void setLayout(LayoutManager gest)** – ataşează unui container un gestionar de poziționare.

Exemplu: O fereastră cu două butoane.

Apăsarea butoanelor nu are nici un efect.

```
import java.awt.*;
import javax.swing.*;
class Fer extend JFrame {
    public Fer (String titlu) {
        super(titlu);
        setSize(200,100);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        Container x=getContentPane();
        x.setLayout(new FlowLayout());
        JButton A=new JButton("Buton 1"); x.add(A);
        JButton B=new JButton("Button 2"); x.add(B);
        setVisible(true);
    }
}
public class pv{
    public static void main(String args[]){
        Fer fp=new Fer("Doua butoane");
    }
}
```



Interfața ActionListener

- Interfața **ActionListener** – ascultătorul de evenimente.
- Un eveniment de tip **ActionListener** este “apăsarea” unui buton.
- Interfața conține antetul unei singure metode:
 - **actionPerformed (ActionEvent e)**
- Pentru ca o componentă să poată răspunde la un eveniment de tipul **ActionEvent**, trebuie să implementeze clasa **ActionListener**.

Implementarea clasei **ActionListener**

- Clasa care include componenta (fereastra) să conțină clauza
implements ActionListener;
- Să fie implementată metoda
actionPerformed()
- Această metodă se va executa automat atunci când este apăsat butonul. Prin urmare, implementarea ei va scrie codul necesar acțiunii dorite.

Clasa ActionEvent

- **ActionEvent** este clasa care conține metoda:
 - **String getActionCommand()** – returnează sirul de caractere asociat componentei care a transmis evenimentul. Metoda poate fi utilizată pentru a depista componenta care a transmis evenimentul.

Exemplu: Când se apasă un buton, în fereastra va apărea sirul reținut de butonul apăsat.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
class Fereastra extends JFrame implements ActionListener{
    Fereastra (String Nume)
    {
        super(Nume); setSize(200,100); setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        Container x=getContentPane();
        x.setLayout(new FlowLayout());
        JButton A=new JButton("Buton 1"); x.add(A);
        JButton B=new JButton("Buton 2"); x.add(B);
        A.addActionListener(this); B.addActionListener(this);
        setVisible(true);
    }

    @Override
    public void actionPerformed(ActionEvent ae) {
        if (ae.getActionCommand().compareTo("Button 1")==0) System.out.println("Ai apasat Buton 1");
                                                else     System.out.println("Ai apasat Buton 2");
    }
}
public class testfer{
    public static void main(String[] args) {
        Fereastra fp=new Fereastra("Doua butoane");
    }
}
```

Clasa **JComponent**

- Prin componentă se înțelege un obiect care are o reprezentare grafică.
- Exemple de componente: butoane, liste, edit-uri, etichete...
- Fiecare tip de componentă rezultă în urma instanțierea unei clase specifice ei.
- De exemplu, un buton rezultă în urma instanțierii clasei **JButton**.
- Clasele tuturor componentelor enumerate sunt subclase ale clasei **JComponent**.
- Putem folosi metodele clasei **JComponent** pentru orice altă componentă concretă.

JavaFX – pachete

- Biblioteca ce ține de aplicațiile Java FX conține pachetele care se încep cu javafx.
- Exemplu de pachete JavaFX:
 - javafx.application
 - javafx.scene
 - javafx.stage
 - etc.
- Numărul pachetelor este mai mult de 30...
- Metafora de bază se referă la platforme teatrale!
- O *platformă teatrală* poate conține *scene teatrale* care la rândul său să conțină anumite *elemente*.

Reprezentarea schematică

Platformă (Stage)

Scena 1

Scena 2

elem1

elem2

elem3

Aplicația Java FX

- Pentru a elabora o aplicație JavaFX este necesar ca aplicația să conțină cel puțin o platformă (stage) și o scenă (scene).
- Corespunzător pentru fiecare există clasele: **Stage** și **Scene**.
- Astfel este necesar de a include în obiectul clasei **Stage** un obiect al clasei **Scene**.

Clasa Stage

- Clasa Stage este un container de nivel superior.
- Toate aplicațiile JavaFX obțin acces la containerul de nivel superior care se numește platformă de bază (primary stage).
- Platforma de bază este oferită de către mediul de execuție la lansarea aplicației.

Clasa Scene

- Clasa Scene este un container pentru elemente care constituie scena.
- Aceste elemente pot fi butoane, etichete de text, elemente grafice... etc.
- Pentru a crea scena este necesar de a adăuga aceste elemente la exemplarul scenei.

Nodurile – elementele Scene-lor

- Butonul este considerat un nod.
- Nodurile pot conține alte grupuri de noduri.
- Nodurile pot avea o structură ramificată.
- Reuniunea tuturor nodurilor poartă numele de *graful scenei* și formează un arbore (tree).
- Un rol important în ierarhia de noduri îl are nodul rădăcină (root) – nodul nivelului cel mai superior.

Clasa Node

- Clasa Node este clasa de bază pentru toate tipurile de noduri.
- Există alte clase care sunt descendenți direcți sau indirecti ale clasei Node.
- De exemplu: Parent, Region, Group, Control.
- Există mai multe modalități de a amplasa elementele în scenă: FlowPane, GridPane, BorderPane – toate clasele din pachetul javafx.scene.layout.

Clasa Application

- Aplicațiile JavaFX trebuie să fie subclasele clasei *Application*.
- Clasa *Application* se află în pachetul `favafx.application`.
- Clasa *Application* conține 3 metode importante **init()**, **start()** și **stop()** care pot fi rescrise.

Metodele init(), start() și stop()

- Sintaxa:
 - void **init()** – *inițializarea tuturor variabilelor, nu se folosește la crearea platformei sau elementelor scenei;*
 - abstract void **start(Stage platforma_de_bază)** – *începutul aplicației; crearea și construirea scenei; obligator este suprascris în program.*
 - void **stop()** – *eliberarea resurselor folosite în aplicație.*

Lansarea aplicației JavaFX

- Se apelează metoda **launch**:
- Sintaxa metodei:
 - `public static void launch(String ... argumentii)`
- *argumentii* – lista de siruri de caractere.
- Apelul metodei **launch()** generează apelul metodelor: **init()** și **start()**.
- La finalizarea lucrului aplicației se revine în metoda **launch()**.

Structura codului de program JavaFX

```
import ...
public class NumeClasa extends Application {
    public static void main(String args[]) {
        ...
        launch(args);
    }
    public void init() {
        ...
    }
    public void start(Stage myStage) {
        ...
    }
    public void stop() {
        ...
    }
}
```

Exemplu 1 – Partea I

```
import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;

public class JavaFXSkel extends Application {

    public static void main(String[] args) {
        System.out.println("Запуск приложения JavaFX");

        // Запустить приложение JavaFX, вызвав метод launch()
        launch(args);
    }

    // Переопределить метод init()
    public void init() {
        System.out.println("В теле метода init()");
    }
}
```

Exemplu 1 – Partea II

```
// Переопределить метод start()
public void start(Stage myStage) {
    System.out.println("В теле метода start()");

    // Задать заголовок окна приложения
    myStage.setTitle("Каркас приложения JavaFX");

    // Создать корневой узел. В данном случае
    // используется плавающая компоновка, но возможны
    // и другие варианты.
    FlowPane rootNode = new FlowPane(); ← Создание корневого узла

    // Создать сцену
    Scene myScene = new Scene(rootNode, 300, 200); ← Создание сцены

    // Установить сцену на платформе
    myStage.setScene(myScene); ← Установка сцены на платформе

    // Отобразить платформу вместе с ее сценой
    myStage.show(); ← Отображение сцены
}

// Переопределить метод stop()
public void stop() {
    System.out.println("В теле метода stop()");
}
```

Запуск приложения JavaFX
В теле метода init()
В теле метода start()
При закрытии окна на консоли отображается следующее сообщение:
В теле метода stop()

Compilarea și rularea programelor JavaFX

- Unul și același program JavaFX poate rula în diferite medii de rulare (ca aplicație desktop, aplicație Web sau aplicație WebStart).
- Compilarea aplicațiilor JavaFX are loc ca și compilarea altor aplicații Java.

Exemplu de aplicație cu **Label** – etichetă de text

- Clasa **Label** intră în componentă pachetului *javafx.scene.control*.
- Clasa **Label** moștenește metodele din clasele *Labeled* (toate elemente cu text) și *Control* (elemente de control).
- Constructorul clasei **Label**: *Label (String sdc)*
- **sdc** - sirul care va fi afișat.

Adăugarea elementului Label

- Eticheta creată trebuie adăugată în conținutul scenei, adică în graful scenei.
- Trebuie de apelat metoda `getChildren()` pentru nodul rădăcină al scenei – rezultatul este o listă.
- În final se adaugă cu metoda `add()` de la lista nodului rădăcină obținut.

```
// Добавить метку в граф сцены  
rootNode.getChildren().add(myLabel);
```

Добавление метки
в граф сцены

Exemplu 2 Label

```
// Демонстрация использования меток JavaFX

import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;
import javafx.scene.control.*;

public class JavaFXLabelDemo extends Application {
    public static void main(String[] args) {

        // Запустить приложение JavaFX, вызвав метод launch()
        launch(args);
    }

    // Переопределить метод start()
    public void start(Stage myStage) {

        // Задать заголовок окна приложения
        myStage.setTitle("Использование метки JavaFX");

        // Использовать компоновку FlowPane для корневого узла
        FlowPane rootNode = new FlowPane();

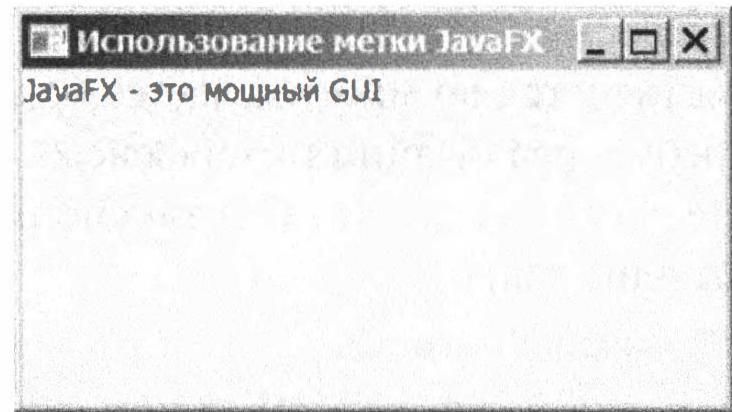
        // Создать сцену
        Scene myScene = new Scene(rootNode, 300, 200);

        // Установить сцену на платформе
        myStage.setScene(myScene);

        // Создать метку
        Label myLabel = new Label("JavaFX - это мощный GUI");
        Создание метки

        // Добавить метку в граф сцены
        rootNode.getChildren().add(myLabel); Добавление метки  
в граф сцены

        // Отобразить платформу вместе с ее сценой
        myStage.show();
    }
}
```



Prelucrarea evenimentelor

- Clasa de bază pentru prelucrarea evenimentelor este clasa **Event** din pachetul javafx.event.
- Clasa **Event** moștenește clasa java.util.eventObject.
- Clasa **Event** are mai multe subclase, din care ca exemplu va fi **ActionEvent** – clasa care încapsulează evenimentele generate de buton.
- Prelucrarea evenimentelor seamănă mult cu abordarea Swing.

Interfața EventHandler

- Se află în pachetul javafx.event.
- Conține o metodă **void handler (T object_event)** care primește ca parametru evenimentul generat – de la oricare element din scenă.

Componenta Button

- Clasa **Button** din pachetul *javafx.scene.control*
- Subclasele clasei Button: *ButtonBase, Labeled, Region, Control, Parent și Node*.
- Butoanele pot conține text, grafică sau ambele la un loc.
- La apăsarea butonului este generat evenimentul *ActionEvent* (pachetul *javafx.event*).
- Metoda care înregistrează apăsarea butonului este:

```
final void setOnAction(EventHandler<ActionEvent> обработчик)
```

Exemplu 3 – Button + Event – Partea I

```
// Демонстрация обработки событий JavaFX для кнопок

import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;
import javafx.scene.control.*;
import javafx.event.*;
import javafx.geometry.*;

public class JavaFXEventDemo extends Application {

    Label response;

    public static void main(String[] args) {
        // Запустить приложение JavaFX, вызвав метод launch()
        launch(args);
    }

    // Переопределить метод start()
    public void start(Stage myStage) {
        // Задать заголовок окна приложения
        myStage.setTitle("Использование кнопок и событий JavaFX");

        // Использовать компоновку FlowPane для корневого узла.
        // В данном случае величина вертикального и горизонтального
        // зазоров составляет 10.
        FlowPane rootNode = new FlowPane(10, 10);

        // Центрировать компоненты на сцене
        rootNode.setAlignment(Pos.CENTER);

        // Создать сцену
        Scene myScene = new Scene(rootNode, 300, 100);
    }
}
```

Exemplu 3 – Button + Event – Partea II

```
// Установить сцену на платформе
myStage.setScene(myScene);

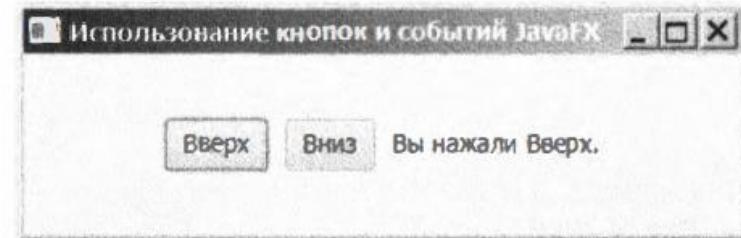
// Создать метку
response = new Label("Нажмите кнопку");

// Создать две кнопки
Button btnUp = new Button("Вверх");
Button btnDown = new Button("Вниз"); // Создание двух кнопок

// Обработать события действий для кнопки "Вверх"
btnUp.setOnAction(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent ae) {
        response.setText("Вы нажали Вверх.");
    }
}); // Создание обработчиков событий для кнопок

// Обработать события действий для кнопки "Вниз"
btnDown.setOnAction(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent ae) {
        response.setText("Вы нажали Вниз.");
    }
}); // Добавить метку и кнопки в граф сцены
rootNode.getChildren().addAll(btnUp, btnDown, response);

// Отобразить платформу вместе с ее сценой
myStage.show();
})
```



Componenta CheckBox

- Reprezentat de clasa CheckBox, subclasa clasei ButtonBase, astfel se poate spune că este un tip de buton.
- Elementul CheckBox în JavaFX are 3 stări: *marcat, nemarcat și nedeterminat*.
- Constructorul clasei CheckBox:
`CheckBox (String sdc)`

Exemplul 4 – CheckBox – Partea I

```
// Демонстрация использования флагков

import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;
import javafx.scene.control.*;
import javafx.event.*;
import javafx.geometry.*;

public class CheckboxDemo extends Application {

    CheckBox cbSmartphone;
    CheckBox cbTablet;
    CheckBox cbNotebook;
    CheckBox cbDesktop;

    Label response;
    Label selected;

    String computers;

    public static void main(String[] args) {

        // Запустить приложение JavaFX, вызвав метод launch()
        launch(args);
    }

    // Переопределить метод start()
    public void start(Stage myStage) {

        // Задать заголовок окна приложения
        myStage.setTitle("Демонстрация флагков");

        // Использовать компоновку FlowPane для корневого узла.
        // В данном случае величина вертикального и горизонтального
        // зазоров составляет 10.
        FlowPane rootNode = new FlowPane(Orientation.VERTICAL, 10, 10);

        // Центрировать компоненты на сцене
        rootNode.setAlignment(Pos.CENTER);
```

Exemplul 4 – CheckBox – Partea II

```
// Создать сцену
Scene myScene = new Scene(rootNode, 230, 200);

// Установить сцену на платформе
myStage.setScene(myScene);

Label heading = new Label("Какие у вас есть устройства?");

// Создать метку, извещающую об изменении состояния флагка
response = new Label("");

// Создать метку, извещающую о выборе любого флагка
selected = new Label("");

// Создать флагки
cbSmartphone = new CheckBox("Смартфон");
cbTablet = new CheckBox("Планшет");
cbNotebook = new CheckBox("Ноутбук");
cbDesktop = new CheckBox("ПК");
```

Создание флагков

```
// Обработка событий действий для флагков
cbSmartphone.setOnAction(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent ae) {
        if(cbSmartphone.isSelected())
            response.setText("Был выбран смартфон.");
        else
            response.setText("Выбор смартфона отменен.");
        showAll();
    }
});
```

Обработка событий флагков

```
cbTablet.setOnAction(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent ae) {
        if(cbTablet.isSelected())
            response.setText("Был выбран планшет.");
        else
            response.setText("Выбор планшета отменен.");
        showAll();
    }
});
```

```
cbNotebook.setOnAction(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent ae) {
        if(cbNotebook.isSelected())
            response.setText("Был выбран ноутбук.");
        else
            response.setText("Выбор ноутбука отменен.");
        showAll();
    }
});
```

```
cbDesktop.setOnAction(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent ae) {
```

Exemplul 4 – CheckBox – Partea III

```
if(cbDesktop.isSelected())
    response.setText("Был выбран ПК.");
else
    response.setText("Выбор ПК отменен.");

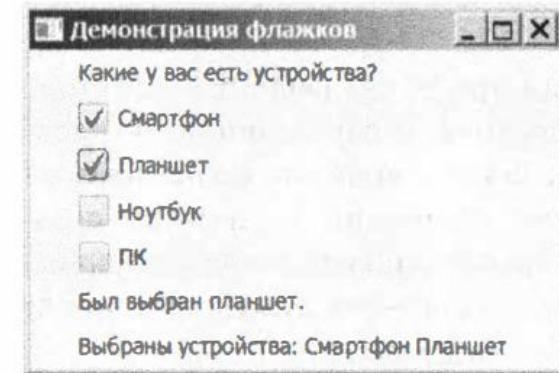
    showAll();
}
});

// Добавить компоненты в граф сцены
rootNode.getChildren().addAll(heading, cbSmartphone, cbTablet,
                            cbNotebook, cbDesktop, response,
                            selected);

// Отобразить платформу вместе с ее сценой
myStage.show();

showAll();
}

// Обновить и отобразить варианты выбора
void showAll() {
    computers = "";
    if(cbSmartphone.isSelected()) computers += "Смартфон ";
    if(cbTablet.isSelected()) computers += "Планшет ";
    if(cbNotebook.isSelected()) computers += "Ноутбук ";
    if(cbDesktop.isSelected()) computers += "ПК";
    selected.setText("Выбраны устройства: " + computers);
}
```



Использование
метода
isSelected()
для определения
состояния
флажков

Componenta ListView

- Clasa ListView permite vizualizarea unei liste de elemente (toate de un tip) cu posibilitatea selectării unui sau mai multor elemente.
- Dacă numărul de elemente din listă este mai mare decât zona vizibilă atunci apare ScrollBar.
- Clasa ListView este o clasă generică și poate înscrie în listă elemente de diferite tipuri.
- Implicit poate fi selectat doar un singur element, dar e posibilă reprogramarea pentru mai multe elemente.

Exemplul 5 – ListView – Partea I

```
// Демонстрация использования списка

import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;
import javafx.scene.control.*;
import javafx.geometry.*;
import javafx.beans.value.*;
import javafx.collections.*;

public class ListViewDemo extends Application {

    Label response;

    public static void main(String[] args) {

        // Запустить приложение JavaFX, вызвав метод launch()
        launch(args);
    }

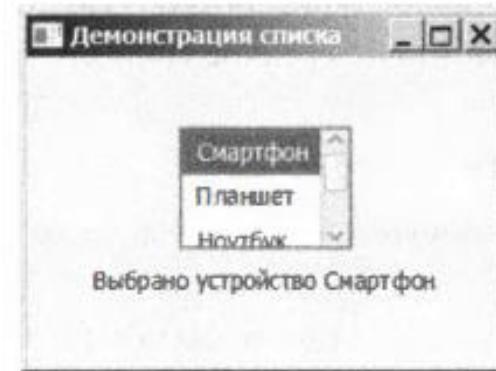
    // Переопределить метод start()
    public void start(Stage myStage) {

        // Задать заголовок окна приложения
        myStage.setTitle("Демонстрация списка");

        // Использовать компоновку FlowPane для корневого узла.
        // В данном случае величина вертикального и горизонтального
```

Exemplul 5 – ListView – Partea II

```
// зазоров составляет 10.  
FlowPane rootNode = new FlowPane(10, 10);  
  
// Центрировать компоненты на сцене  
rootNode.setAlignment(Pos.CENTER);  
  
// Создать сцену  
Scene myScene = new Scene(rootNode, 200, 120);  
  
// Установить сцену на платформе  
myStage.setScene(myScene);  
  
// Создать метку  
response = new Label("Выбор типа устройства");  
  
// Создать объект типа ObservableList для списка  
ObservableList<String> computerTypes =  
    FXCollections.observableArrayList("Смартфон", "Планшет",  
        "Ноутбук", "ПК");  
  
// Создать список  
ListView<String> lvComputers =  
    new ListView<String>(computerTypes);  
    ↑  
    └─ Создание списка, который отображает элементы  
        из объекта computerTypes  
  
// Задать предпочтительные значения высоты и ширины  
lvComputers.setPrefSize(100, 70);  
  
// Получить модель выбора для списка  
MultipleSelectionModel<String> lvSelModel =  
    lvComputers.getSelectionModel();  
  
// Использовать слушатель для реагирования на изменения  
// выделения внутри списка  
lvSelModel.selectedItemProperty().addListener(  
    new ChangeListener<String>() {  
        ↑  
        └─ Обработка событий  
            изменения  
        public void changed(ObservableValue<? extends String>  
            changed, String oldVal, String newVal) {  
  
            // Отобразить выбор  
            response.setText("Выбрано устройство " + newVal);  
        }  
    });  
  
// Добавить метку и список в граф сцены  
rootNode.getChildren().addAll(lvComputers, response);  
  
// Отобразить платформу вместе с ее сценой  
myStage.show();  
}  
}
```



Componenta TextField

- Componenta **TextField** este destinată introducerii a unui rând de text.
- Clasa **TextField** are 2 constructori: unul care setează lungimea componentei, al doilea setează textul implicit.
- Metodele de bază din clasa **TextField**:
 - *setPrefColumnCount* – setarea dimensiunii componentei;
 - *setText/getText* – înscrierea și preluarea textului din componentă;
 - *setPromptText* – setarea textului de sugestii pentru componentă;
- Evenimentul care poate fi prelucrat – apăsarea butonului Enter de pe tastatură la finalizarea introducerii textului (vezi exemplul următor).

Exemplul 6 – TextField – Partea I

```
// Демонстрация использования текстового поля

import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;
import javafx.scene.control.*;
import javafx.event.*;
import javafx.geometry.*;

public class TextFieldDemo extends Application {

    TextField tf;
    Label response;

    public static void main(String[] args) {

        // Запустить приложение JavaFX, вызвав метод launch()
        launch(args);
    }

    // Переопределить метод start()
    public void start(Stage myStage) {

        // Задать заголовок окна приложения
        myStage.setTitle("Демонстрация текстового поля");

        // Использовать компоновку FlowPane для корневого узла.
        // В данном случае величина вертикального и горизонтального
        // зазоров составляет 10.
        FlowPane rootNode = new FlowPane(10, 10);

        // Центрировать компоненты на сцене
        rootNode.setAlignment(Pos.CENTER);

        // Создать сцену
        Scene myScene = new Scene(rootNode, 230, 140);
```

Exemplul 6 – TextField – Partea II

```
// Установить сцену на платформе
myStage.setScene(myScene);

// Создать метку
response = new Label("Получить имя: ");

// Создать кнопку, управляющую получением текста
Button btnGetText = new Button("Получить имя");

// Создать текстовое поле
tf = new TextField(); ← Создание текстового поля

// Задать подсказку
tf.setPromptText("Введите имя."); ← Настройка подсказки для текстового поля

// Задать предпочтительное количество столбцов
tf.setPrefColumnCount(15); ← Настройка ширины текстового поля

// Использовать лямбда-выражение, обрабатывающее
// события действий для текстового поля. События
// действий генерируются при нажатии клавиши <ENTER>
// в то время, когда фокус ввода находится в текстовом
// поле. В данном случае обработка события заключается
// в получении и отображении текста.
tf.setOnAction( (ae) -> response.setText("Введено. Имя: " +
                                             tf.getText()));
                                             ↑
                                             Установка событий действий для текстового поля

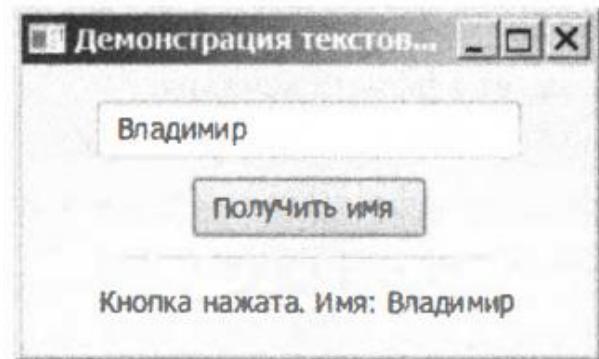
// Использовать лямбда-выражение для получения текста из
// текстового поля при нажатии кнопки
btnGetText.setOnAction((ae) ->
    response.setText("Кнопка нажата. Имя: " + tf.getText()));

// Использовать разделитель для лучшей организации вывода
Separator separator = new Separator();
separator.setPrefWidth(180);

// Добавить компоненты в граф сцены
rootNode.getChildren().addAll(tf, btnGetText, separator,
                             response);

// Отобразить платформу вместе с ее сценой
myStage.show();
}

}
```



Efectele

- Efectele sunt produse cu ajutorul clasei abstracte **Effect** din pachetul *javafx.scene.effect*.
- Pentru setarea unui efect se folosește metoda *setEffect*.
- Metode utile: *setWidth*, *Height*, *setIterations*, *setTopOpacity*, *setBottomOpacity*, *setTopOffset*, *setFraction*.

Bloom	Увеличивает яркость более светлых частей узла
BoxBlur	Размывает изображение узла
DropShadow	Отображает тени, отбрасываемые узлами
Glow	Создает эффект свечения
InnerShadow	Отображает тень внутри узла
Lighting	Создает эффекты теней при наличии источника света
Reflection	Создает эффекты отражения

Transformări

- Transformările sunt posibile cu folosirea clasei abstracte **Transform** din pachetul *javafx.scene.transform*.
- Clasa **Transform** conține mai multe subclase: **Rotate**, **Scale**, **Shear** și **Translate**.
- Pentru a aplicat transformarea la o componentă se aplică metoda *add*, corespunzător pentru a șterge transformarea – metoda *clear/remove*.
- Metode care schimbă proprietățile componentelor: *setRotate()*, *setScaleX()*, *setScaleY()*, *setTranslateX()*, *setTranslateY()*.

Exemplu 6 – Efecte și Transformări - I

```
import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;
import javafx.scene.control.*;
import javafx.event.*;
import javafx.geometry.*;
import javafx.scene.transform.*;
import javafx.scene.effect.*;
import javafx.scene.paint.*;

public class EffectsAndTransformsDemo extends Application {

    double angle = 0.0;
    double scaleFactor = 0.4;
    double blurVal = 1.0;

    // Создать начальные объекты преобразований и эффектов
    Reflection reflection = new Reflection();
    BoxBlur blur = new BoxBlur(1.0, 1.0, 1);
    Rotate rotate = new Rotate();
    Scale scale = new Scale(scaleFactor, scaleFactor);

    // Создать кнопки
    Button btnRotate = new Button("Повернуть");
    Button btnBlur = new Button("Размыть");
    Button btnScale = new Button("Масштабировать");

    Label reflect = new Label("Отражение добавляет визуальный блеск");

    public static void main(String[] args) {

        // Запустить приложение JavaFX, вызвав метод launch()
        launch(args);
    }

    // Переопределить метод start()
    public void start(Stage myStage) {

        // Задать заголовок окна приложения
        myStage.setTitle("Демонстрация эффектов и преобразований");

        // Использовать компоновку FlowPane для корневого узла.
        // В данном случае величина вертикального и горизонтального
        // зазоров составляет 20.
        FlowPane rootNode = new FlowPane(20, 20);

        // Центрировать компоненты на сцене
        rootNode.setAlignment(Pos.CENTER);
    }
}
```

Создание эффектов
и преобразований

Exemplu 6 – Efecte și Transformări - II

```
// Создать сцену
Scene myScene = new Scene(rootNode, 300, 120);

// Установить сцену на платформе
myStage.setScene(myScene);

// Добавить поворот в список преобразований для
// кнопки "Повернуть"
btnRotate.getTransforms().add(rotate); ← Добавление вращения
                                         к кнопке btnRotate

// Добавить масштабирование в список преобразований
// для кнопки "Масштабировать"
btnScale.getTransforms().add(scale); ← Добавление отражения
                                         к кнопке btnScale

// Задать эффект отражения для метки
reflection.setTopOpacity(0.7);
reflection.setBottomOpacity(0.3);
reflect.setEffect(reflection); ← Настройка отражения
                                         для метки reflect

// Обработать события действий для кнопки "Повернуть"
btnRotate.setOnAction(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent ae) {
        // При каждом нажатии кнопки она поворачивается
        // на 15 градусов вокруг центра
        angle += 15.0;

        rotate.setAngle(angle);
        rotate.setPivotX(btnRotate.getWidth()/2);
        rotate.setPivotY(btnRotate.getHeight()/2);
    }
});

// Обработать события действий для кнопки "Масштабировать"
btnScale.setOnAction(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent ae) {
        // При каждом нажатии кнопки изменяются ее размеры
        scaleFactor += 0.1;
        if(scaleFactor > 2.0) scaleFactor = 0.4;

        scale.setX(scaleFactor);
        scale.setY(scaleFactor);

    }
});

// Обработать события действий для кнопки "Размыть"
btnBlur.setOnAction(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent ae) {
        // При каждом нажатии кнопки изменяется
        // степень размытия ее изображения
        if(blurVal == 10.0) {
```

Exemplu 6 – Efecte și Transformări - III

```
        blurVal = 1.0;
        btnBlur.setEffect(null); ← Удалить размытие с кнопки btnBlur
        btnBlur.setText("Отменить размытие");
    } else {
        blurVal++;
        btnBlur.setEffect(blur); ← Добавить размытие к кнопке btnBlur
        btnBlur.setText("Добавить размытие");
    }
    blur.setWidth(blurVal);
    blur.setHeight(blurVal);
}
});

// Добавить метку и кнопки в граф сцены
rootNode.getChildren();
addAll(btnRotate, btnScale, btnBlur, reflect);

// Отобразить платформу вместе с ее сценой
myStage.show();
}
}
```

Результат выполнения программы представлен на приведенном ниже рисунке.

