POO – Laborator 3

Metode si constructori. Supraincarcarea numelor. Relatii intre clase: asocierea si utilizarea

3.1. Descrierea laboratorului

In aceasta lucrare de laborator vor fi acoperite urmatoarele probleme:

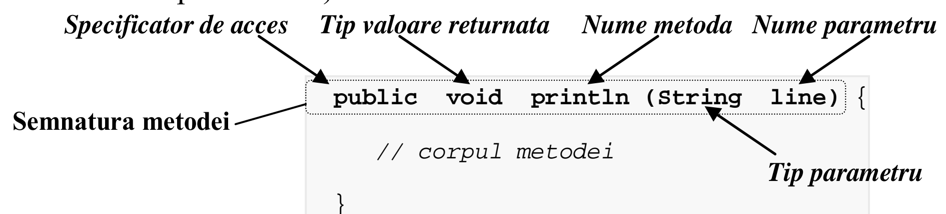
- **Elemente de programare orientata spre obiecte (in Java)**
  - **Specificarea comportamentului (metodele si constructorii):**
    - **Semnaturile metodelor si returnarea valorilor (in Java)**
    - **Constructorii** - functiile pentru initializarea obiectelor
    - **Supraincarcarea numelor metodelor si constructorilor** – polimorfismul static
  - **Relatii intre clase:**
    - **Asocierea si utilizarea**
- **Studiu de caz: clasele Mesaj si Pachet**
  - **Structura de baza: campuri, constructori, metode**
  - **Supraincarcarea numelor**
  - **Relatii intre clase**
- Mediul de dezvoltare JCreator
- **Teme de casa**

3.2. Metode si constructori. Supraincarcarea numelor

3.2.1. Semnatura metodei. Returnarea valorilor

Dupa **invocare** (apelare) **metodele** (functiile membru) **obiectelor efectueaza sarcini** (in general utilizand argumentele pasate in momentul apelului si valorile campurilor obiectului) **care se pot finaliza** (sau nu) inclusiv **prin returnarea unei valori.**

Definitia unei metode contine 2 parti: semnatura (antetul, declaratia) **si corpul** (blocul, segmentul, secventa de instructiuni a implementarii).



Semnatura specifica:
- **numele** metodei,
- **lista de parametri formali** (numarul, ordinea, tipul si numele lor),
- **tipul valorii returnate**,
- **specificatori ai unor proprietati explicite** (modificatori ai proprietatilor implicite).

Daca **metoda nu returneaza** nici o valoare, **tipul valorii returnate** este declarat **void**. Tipul valorii returnate poate fi unul dintre **cele 8 tipuri primitive** Java (byte, short, int, long, float, double, boolean si char), sau unul dintre **cele 3 tipuri referinta** (tablourile, clasele si interfetele Java).

Corpul metodei contine **secventa de instructiuni** care specifica **pasii necesari indeplinirii sarcinilor** (evaluari de expresii, atribuiri, decizii, iteratii, apeluri de metode, etc.). **Returnarea valorilor** este **specificata in codul metodelor** prin instructiunea **return** urmata de o **expresie care poate fi evaluata la o valoare** de **tipul declarat in semnatura**.

In laborator: Pentru exemplul de mai jos:

1. **Identificati numele metodelor**.
2. **Identificati numele** si **tipul parametrilor** in fiecare caz.
3. **Identificati tipul valorilor returnate** in fiecare caz.
4. **Identificati instructiunile return** si comparati tipul expresiilor cu tipul declarat.

```
import javax.swing.JoptionPane;            // clasa de biblioteca (package) Java, externa
                                           // dar accesibila codului care urmeaza
public class DialogUtilizator01 {          // clasa definita de utilizator (declaratia)
                                           // corpul clasei:
   public String nextLine(String text) {              // metode Java (operatii)
       return JOptionPane.showInputDialog(text);      // returneaza o valoare tip String
   }
   public int nextInt(String text) {                  // returneaza o valoare tip int
       return Integer.parseInt(JOptionPane.showInputDialog(text));
   }
   public void printLine(String text) {               // nu returneaza nici o valoare
       JOptionPane.showMessageDialog(null, text);
   }
}
```

In documentatia (API-ul) claselor Java pot fi gasite detalii privind **clasa `JoptionPane`**.

In laborator:

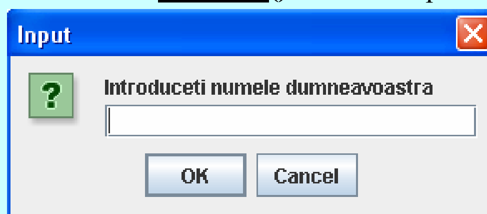
1. **Lansati** mediul **BlueJ**. **Inchideti proiectele anterioare** (cu **Ctrl+W** sau **Project** si **Close**).
2. **Creati un nou proiect** numit ***dialog*** (cu **Project**, apoi **New Project…**, selectati **D:/**, apoi **Software2006**, apoi **numarul grupei**, apoi scrieti **dialog**).

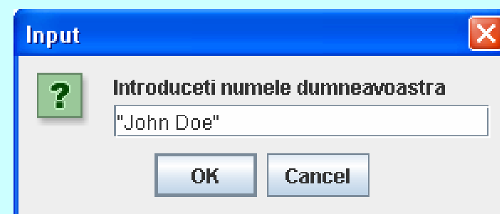
3. **Creati o noua clasa**, numita **DialogUtilizator01**, cu **New Class…**
4. ***Double-click* pe noua clasa** (deschideti editorul) si **inlocuiti codul cu cel de sus**.

5. **Compilati codul** apoi **creati un obiect din noua clasa**.

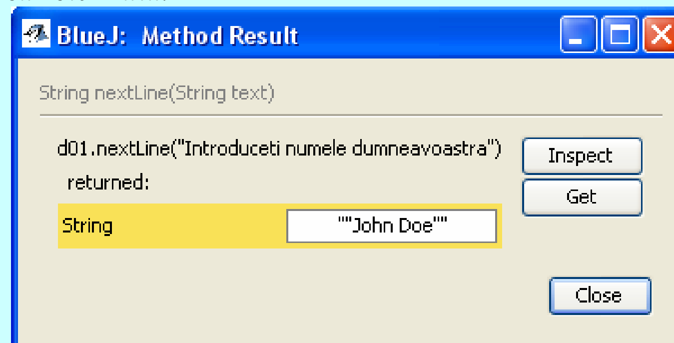
In laborator:

1. **Executati** metoda **nextLine()** dandu-i ca parametru **"Introduceti numele dumneavoastra"**.

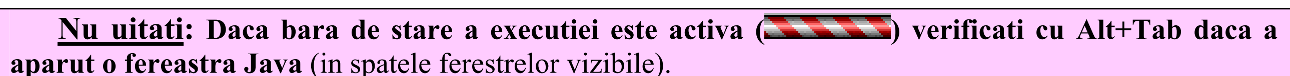


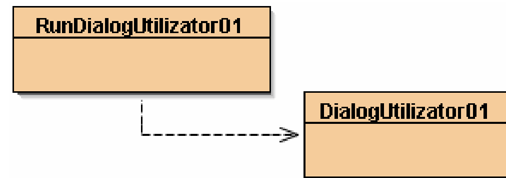


2. **Inspectati valoarea returnata**.



3. **Executati** si metodele **nextInt()** si **printLine()** si **urmariti efectul lor**.

Nu uitati: Daca bara de stare a executiei este activa (**) verificati cu Alt+Tab daca a aparut o fereastra Java** (in spatele ferestrelor vizibile).



Clasa Java pentru testarea clasei anterior definite:

```
public class RunDialogUtilizator01 {
   public static void main(String[] args) {
       DialogUtilizator01 d01 = new DialogUtilizator01();
       String linie = d01.nextLine("Introduceti numele dumneavoastra");
       d01.printLine("Buna ziua " + linie + ".  Bine ai venit in lumea Java!");
   }
}
```

In laborator:

1. Tot **in proiectul** ***dialog*****, creati o noua clasa** numita **RunDialogUtilizator01**
2. ***Double-click*** **pe noua clasa** (deschideti editorul) si **inlocuiti codul cu cel de sus**.
3. **Compilati codul** si **executati metoda main()** a noii clase (***right-click*** **pe clasa** si **selectare main()**).

3.2.2. Constructorii

Constructorul Java este un **tip special de functie**, care
- are acelasi nume cu numele clasei in care este declarat,
- este utilizat pentru a initializa orice nou obiect de acel tip (stabilind valorile campurilor/ atributelor obiectului, **in momentul crearii lui dinamice**),
- nu returnează nici o valoare,
- are aceleasi niveluri de accesibilitate, **reguli de implementare a corpului** si **reguli de supraincarcare a numelui** ca si metodele obisnuite.

Declaratia minimala a unui constructor Java (fara elemente optionale) este:

```
NumeClasa() {
   // Corp constructor
}
```

Daca elementele optionale nu sunt declarate compilatorul Java presupune implicit despre constructorul curent declarat **ca doar clasele din acelasi director cu clasa curenta au acces la el**.

Pentru a specifica in mod explicit un alte proprietati ale constructorilor pot fi folosite **elementele din tabelul urmator**.

Element al declaratiei constructorului	Semnificatie
`public`	Orice cod exterior clasei are acces la constructor
`protected`	Doar codul exterior din sublcase sau aflat in acelasi director are acces la constructor
`private`	Nici un cod exterior nu are acces la constructor
`( listaParametri )`	Lista de parametri primiti de constructor, despartiti prin virgule, cu formatul: `tipParametru numeParametru`

In Java nu este neaparat necesara scrierea unor constructori pentru clase, deoarece **un constructor implicit** este **generat automat de sistemul de executie (DOAR) pentru o clasa care nu declara explicit constructori**. Acest constructor nu face nimic (nici o initializare, **implementarea lui continand un bloc de cod vid**: `{ }`). De aceea, **orice initializare dorita explicit impune scrierea unor constructori**.

Un exemplu de clasa similara celei anterioare, dar care defineste explicit un constructor:

```
import java.util.Scanner;             // clasa de biblioteca (package) Java
public class DialogUtilizator02 {     // clasa definita de utilizator
   private Scanner sc;                        // camp Java (atribut)
   private String prompt;
   public DialogUtilizator02(String nume) {    // constructor (initializator)
       this.sc = new Scanner(System.in);
       this.prompt = nume + "> ";
   }
   public String nextLine(String text) {       // metode Java (operatii)
       System.out.print(text);
       return this.sc.nextLine();
   }
   public int nextInt(String text) {
       System.out.print(text);
       return Integer.parseInt(this.sc.nextLine());
   }
   public void printLine(String text) {   System.out.println(text);   }
}
```

In documentatia (API-ul) claselor Java pot fi gasite detalii privind **clasa `Scanner`**.

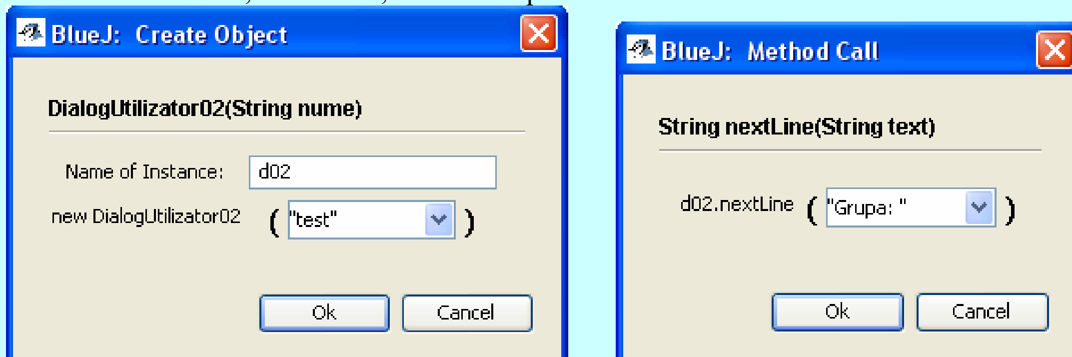
In laborator:

DialogUtilizator02

1. Tot **in proiectul *dialog***, **creati o noua clasa** numita **DialogUtilizator02**
2. **Intrati in codul clasei** (in editor), **inlocuiti-i codul cu cel dat**, apoi **compilati-l**.

   d02: DialogUtilizatoı

3. **Creati un obiect nou**, numit **d02**, pasandu-i constructorului valoarea **"test"**.



4. **Executati** metoda **nextLine()** dandu-i ca parametru **"Grupa : "**. Ce apare in **Terminal Window**?
5. **Inspectati valoarea returnata**.
6. **Executati** si metodele **nextInt()** si **printLine()** si **urmariti efectul lor**.

3.2.3. Supraincarcarea numelor metodelor si constructorilor

Java suporta **supraincarcarea numelor (*name overloading*)** pentru metode si constructori. Astfel, **o clasa poate avea orice numar de metode cu acelasi nume** cu conditia ca **listele lor de parametri sa fie diferite**.

In mod similar, **o clasa poate avea orice numar de constructori** (acestia avand toti acelasi nume - identic cu numele clasei) cu conditia ca **listele lor de parametri sa fie diferite**. De exemplu, codul clasei anterioare poate fi completat cu constructorul:

```
   public DialogUtilizator02() {          // constructor (initializator)
       this.sc = new Scanner(System.in);
       this.prompt = "IMPLICIT" + "> ";  // echivalent cu: this.prompt = this("IMPLICIT ");
   }
```

In laborator (*codul complet este parte din tema de casa!*):

1. **Intrati in codul clasei DialogUtilizator02** (in editor), **adaugati constructorul**, apoi **recompilati**.
2. **Creati un obiect** nou **folosind noul constructor**. Ce observati?
3. **Executati-i** metoda **printLine()** dandu-i ca parametru "`POO`". Ce apare in **Terminal Window**?
4. **Creati un obiect** nou **folosind primul constructor**, caruia ii pasati "`EXPLICIT`".
5. **Executati-i** metoda **printLine()** dandu-i ca parametru "`POO`". Ce apare in **Terminal Window**?

In laborator (*codul complet este parte din tema de casa!*):

1. **Concepeti si editati codul unei metode noi** a clasei **DialogUtilizator02**, cu semnatura:
   `public void printLine()`
   care nu primeste nici un parametru dar **afiseaza** in **Terminal Window**
   (folosind `System.out.println()`) textul: "`Nu am primit nici un parametru`".
2. **Recompilati clasa si creati un obiect** nou **folosind noul constructor**.
3. **Executati noua metoda printLine()**. Ce apare in **Terminal Window**?
4. **Executati** din nou **vechea metoda printLine()**, cu parametru "`x`". Ce apare in **Terminal Window**?

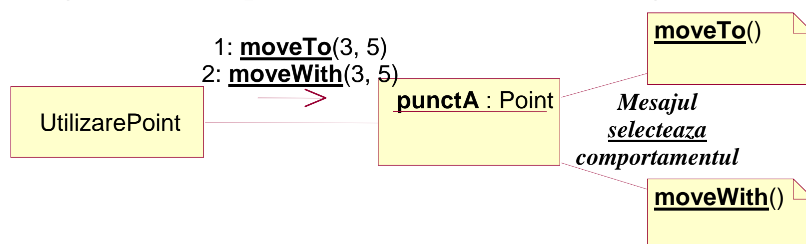
3.2.4. Relatii intre clase: asocierea si utilizarea

Legătura este o **cale între obiectele care se cunosc** (văd) unul pe altul (îşi pot transmite mesaje – apelurile de metode), **pentru aceasta avand referinte unul către celălalt**. Fie clasele Java:

```
 1  public class Point {
 2     private int x;
 3     private int y;
 4  
 5     public Point(int abscisa, int ordonata) {
 6        x = abscisa;
 7        y = ordonata;
 8     }
 9     public void moveTo(int abscisaNoua, int ordonataNoua) {
10        x = abscisaNoua;
11        y = ordonataNoua;
12     }
13     public void moveWith(int deplasareAbsc, int deplasareOrd) {
14        x = x + deplasareAbsc;
15        y = y + deplasareOrd;
16     }
17     public int getX() {   return x;   }
18     public int getY() {   return y;   }
19  }
```

```
1  public class UtilizarePoint {
2     private static Point punctA;     // referinta, legatura catre un obiect Point
3  
4     public static void main(String[] args) {
5        punctA = new Point(3, 4);     // alocare si initializare atribut punctA
6        punctA.moveTo(3, 5);          // trimitere mesaj moveTo() catre punctA
7        punctA.moveWith(3, 5);        // trimitere mesaj moveWith() catre punctA
8     }
9  }
```

Interactiunile intre obiecte pot fi **reprezentate prin intermediul unor diagrame** (numite **de colaborare intre obiecte** in limbajul **UML** – ***Unified Modeling Language***) in care **obiectele care interactioneaza sunt legate intre ele prin linii continue** denumite **legaturi**.



Prezenta unei legaturi (linii) semnifica faptul ca **un obiect cunoaste sau vede un alt obiect**, ca ii poate apela/invoca functiile membru/metodele, ca **poate comunica cu acesta prin intermediul mesajelor declansate** de apelurile/invocarile functiilor membru/metodelor.

Fiecărei familii de legături între obiectele aceleiasi clase **ii corespunde o relaţie între clasele acelor obiecte**.

Asocierea este **abstracţia legăturilor** care există între obiectele instanţe ale claselor asociate (implicit **bidirecţională**). **Implicit, asocierea exprimă un cuplaj (dependenta) redus între abstracţii**, clasele asociate rămânând relativ independente. **Asocierile bidirectionale** intre doua clase corespund situatiei in care **ambele clase au referinte una catre cealalta**.

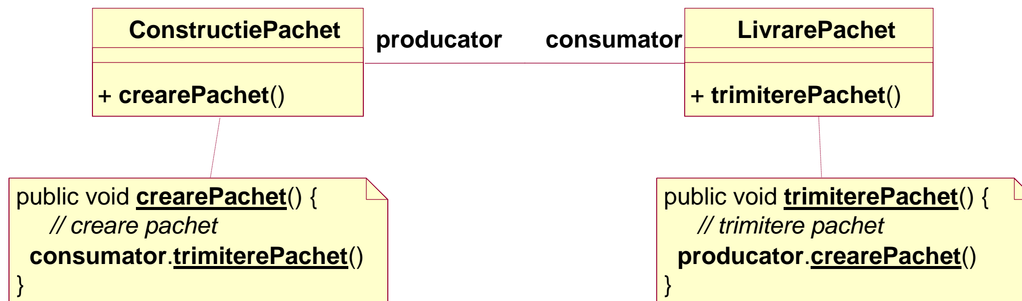
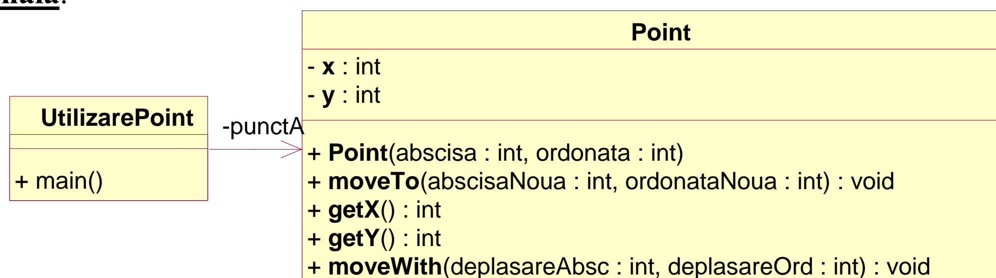


Diagrama UML de mai sus are drept corespondent **codul Java**:

```
1  public class ConstructiePachet {
2     LivrarePachet consumator; // referinta care permite apelul trimiterePachet()
3
4     public void crearePachet(  /* eventuali parametri */ ) {
5        // creare pachet
6        consumator.trimiterePachet()
7     }
8  }
```

```
1  public class LivrarePachet {
2     ConstructiePachet producator; // referinta care permite apelul crearePachet()
3
4     public void trimiterePachet( /* eventuali parametri */ ) {
5        // trimitere pachet
6        producator.crearePachet()
7     }
8  }
```

Clasele `Point` si `UtilizarePoint` sunt de exemplu intr-o **relatie de asociere (cu navigabilitate) unidirectionala**:

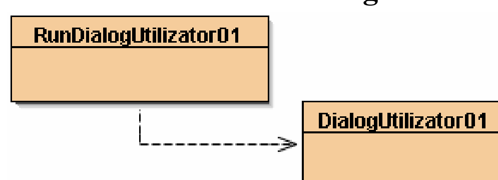


Clasa `UtilizarePoint` are un atribut `punctA` de tip `Point` care permite clasei `UtilizarePoint` sa trimita mesaje unui obiect (`pointA`) al clasei `Point`.

```
private Point punctA;                              // atribut de tip Point
```

Clasa `Point` in schimb **nu are nici o referinta** catre clasa `UtilizarePoint` care sa ii permita trimiterea de mesaje (invocari de metode). **Legatura intre obiectele** celor doua clase **fiind unidirectionala, asocierea** dintre ele **este tot unidirectionala**.

Asocierile unidirectionale pot fi considerate **relatii de utilizare**. Ele se reprezinta prin **sageti indreptate pe directia catre care exista referinta** (catre care se pot trimite mesaje). Clasa **RunDialogUtilizator01** utilizeaza un obiect al clasei **DialogUtilizator01**:

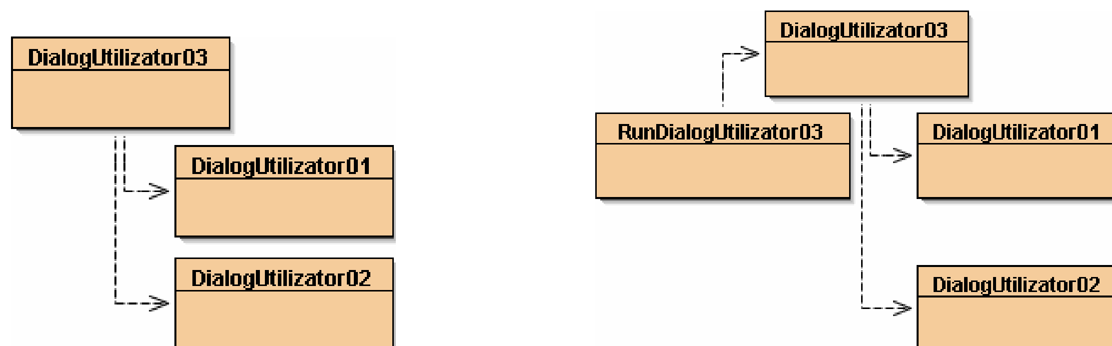


Pornind de la clasele **DialogUtilizator01** si **DialogUtilizator01 se poate construi o clasa DialogUtilizator03** care **sa utilizeze obiecte ale celor doua clase** de mai sus:

```
public class DialogUtilizator03 {       // clasa definita de utilizator
   private boolean grafic;                       // campuri Java (atribute)
   private DialogUtilizator01 d01;               // (ansamblul lor formeaza starea)
   private DialogUtilizator02 d02;
   public DialogUtilizator03(boolean grafic) {  // constructor (initializator)
       this.grafic = grafic;
       if (this.grafic)    this.d01 = new DialogUtilizator01();
       else                this.d02 = new DialogUtilizator02();
   }
   public String nextLine(String text) {        // metode Java (operatii)
       if (this.grafic)    return d01.nextLine(text);
       else                return d02.nextLine(text);
   }
   public int nextInt(String text) {
       if (this.grafic)    return d01.nextInt(text);
       else                return d02.nextInt(text);
   }
   public void printLine(String text) {
       if (this.grafic)    d01.printLine(text);
       else                d02.printLine(text);
   }
}
```

In laborator:

1. **In proiectul** ***dialog*** **creati** clasa **DialogUtilizator03** folosind **codul dat mai sus**.
2. **Compilati** codul si **creati 2 obiecte**, unul **cu parametru** **true** si unul **cu parametru** **false.**
3. **Inspectati cele doua obiecte**.



Codul de mai jos arata felul in care se poate utiliza chiar clasa **clasa DialogUtilizator03** pentru a obtine **acces simultan** si la **interactivitate grafica** si la **interactivitate prin consola**:

```
public class RunDialogUtilizator03 {
    public static void main(String[] args) {
        DialogUtilizator03 consola = new DialogUtilizator03(false);
        DialogUtilizator03 grafic = new DialogUtilizator03(true);
        consola.printLine("Test dialog consola");
        grafic.printLine("Test dialog grafic");
    }
}
```

In laborator:

1. **In proiectul** ***dialog*** **creati** clasa **RunDialogUtilizator03** folosind **codul dat mai sus**.
2. **Compilati codul si executati metoda main()**.
3. **Ce observati**?

3.3. Studiu de caz: clasele Mesaj si Pachet

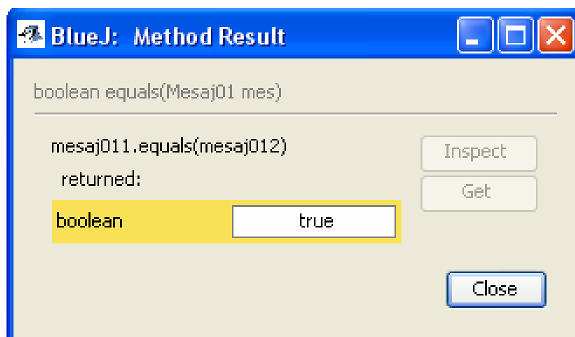
3.3.1. Structura de baza a clasei Mesaj: campuri, constructori, metode

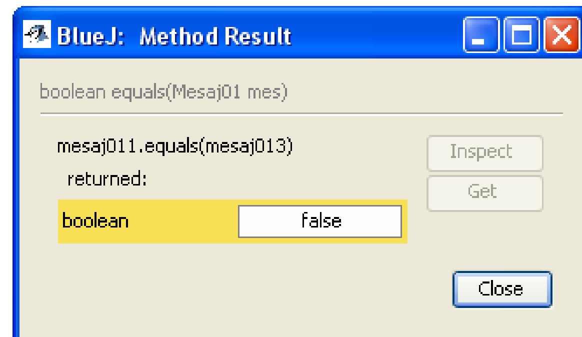
Clasa Mesaj01 incapsuleaza un obiect de tip String care reprezinta un mesaj de la utilizatorul curent (**regrupand textul mesajului cu metodele prin care este controlat accesul la acesta**):

```
public class Mesaj01 {
    private String text;
    public Mesaj01(String text) {          // constructor cu parametru
        this.text = text;
    }
    public String getText() {              // obtinerea valorii campului
        return this.text;
    }
    public String toString() {
        return ("Mesaj: " + this.text);
    }
    public void display() {
        System.out.println(this.toString());
    }
    public boolean equals(Mesaj01 mes) {
        return this.text.equals(mes.text);
    }
}
```

In laborator:

1. **Lansati** mediul **BlueJ**. **Inchideti toate proiectele** (**Ctrl+W**). **Creati un proiect** numit ***mesaj***.
2. **In proiectul *mesaj* creati** clasa **Mesaj01** folosind **codul dat mai sus**.
3. **Compilati codul** si **creati 3 obiecte** tip **Mesaj01 - doua dintre ele cu aceleasi valori ale** campului **text** si **al treilea** cu **alte valori ale** campului **text**.
4. **Inspectati obiectele**.
5. **Apelati** metodele **getText**(), **toString**() si **display**() pentru unul dintre obiecte.
6. **Apelati** metoda **equals**() **a primului obiect** folosind **ca parametri celelalte doua obiecte**.





In laborator (***codul complet este parte din tema de casa!***):

1. **In proiectul *mesaj* creati o noua clasa** numita **Mesaj02**, pornind de la codul **Mesaj01**:
   - **adaugati un camp** de tip **int** numit **tip**,
   - **adaptati constructorul** pentru a **initializa si campul** numit **tip**,
   - **adaugati o metoda** pentru **obtinerea valorii campului** numit **tip**,
   - **adaptati metoda toString() pentru a include si** campul **tip** in String-ul returnat (de exemplu, pentru un **<text>** si un **<tip>** dat, va returna: **Mesaj de tip <tip>: <text>**),
   - **adaptati metoda equals(Mesaj02 mes) pentru a include si** comparatia campurilor **tip**
2. **Compilati codul** si **creati 3 obiecte** tip **Mesaj02 - doua dintre ele cu aceleasi valori ale** campului **text** si **al treilea** cu **alte valori ale** campului **text**.
3. **Inspectati obiectele**.
4. **Apelati** metodele **getText**(), **toString**() si **display**() pentru unul dintre obiecte.
5. **Apelati** metoda **equals**() **a primului obiect** folosind **ca parametri celelalte doua obiecte**.

3.3.2. Supraincarcarea numelor in cazul clasei Mesaj

In cazul clasei Mesaj01, **supraincarcarea numelui constructorului** ar insemna **crearea unui constructor suplimentar**, de exemplu unul care nu primeste nici un parametru:

```
public Mesaj01() { this(""); }    // corpul este echivalent cu:  { this.text = ""; }
```

In laborator

1. **Editati codul** clasei **Mesaj01** si **adaugati constructorul dat mai sus**.
2. **Compilati codul** si **creati 2 obiecte** tip **Mesaj01, fiecare cu cate un constructor**.
3. **Apelati** metoda **equals**() **a primului obiect** folosind **ca parametru cel de-al doilea obiect**.

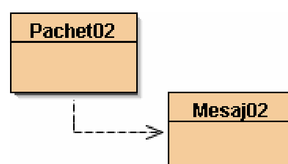
In laborator (***codul complet este parte din tema de casa!***):

1. **Editati codul** clasei **Mesaj02** si **adaugati un constructor fara parametri, care sa initializeze cele doua campuri cu niste valori implicite**.
2. **Compilati codul** si **creati 2 obiecte** tip **Mesaj02, fiecare cu cate un constructor**.
3. **Inspectati obiectele**.
4. **Apelati** metoda **equals**() **a primului obiect** folosind **ca parametru cel de-al doilea obiect**.

3.3.3. Relatii intre clase: cazul claselor Mesaj si Pachet

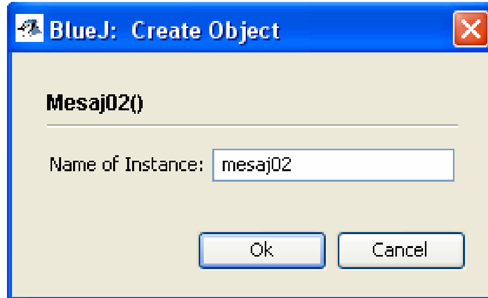
Pentru a exemplifica **relatia de utilizare intre clase** va fi creata **o clasa Pachet02** care **incapsuleaza un obiect Mesaj02 (regrupand mesajul si sursa lui cu metodele prin care este controlat accesul la acestea)**:

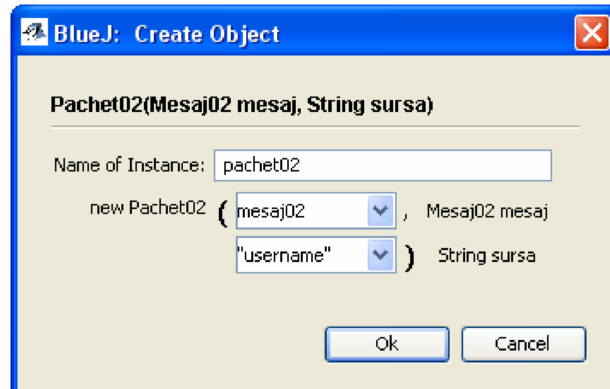
```
public class Pachet02 {
    private Mesaj02 mesaj;
    private String sursa;
    public Pachet02(Mesaj02 mesaj, String sursa) {
        this.mesaj = mesaj;
        this.sursa = sursa;
    }
    public Mesaj02 getMesaj() {
        return this.mesaj;
    }
    public String getSursa() {
        return this.sursa;
    }
    public String toString() {
        return ("Pachetul de la " + this.sursa + " contine: " + this.mesaj);
    }
    public boolean equals(Pachet02 pac) {
        return (this.mesaj.equals(pac.mesaj)) &&(this.sursa.equals(pac.sursa));
    }
}
```



In laborator:

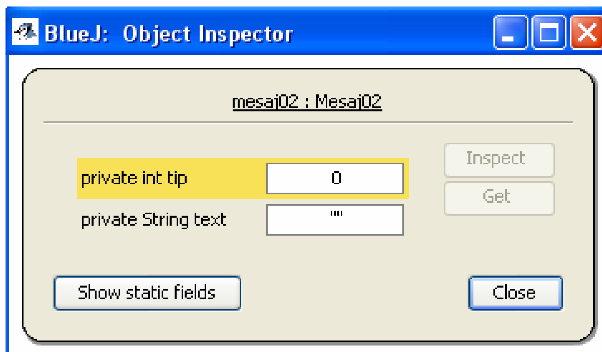
1. **In proiectul *mesaj* creati o noua clasa** numita **Pachet02**, folosind codul de mai sus.
2. **Compilati codul** si **creati mai intai un obiect** tip **Mesaj02** si **apoi un obiect** tip **Pachet02** (**pasandu-i** constructorului **Pachet02(Mesaj02 mesaj, String sursa) obiectul** tip **Mesaj02**).

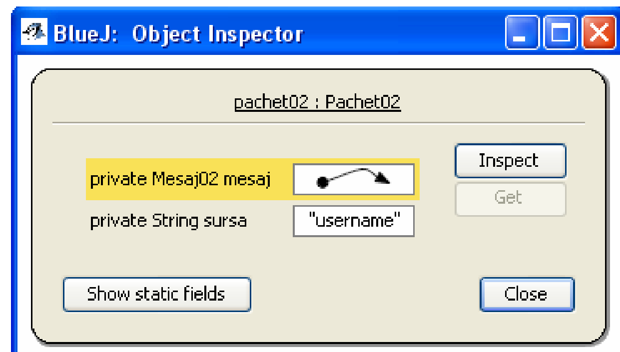


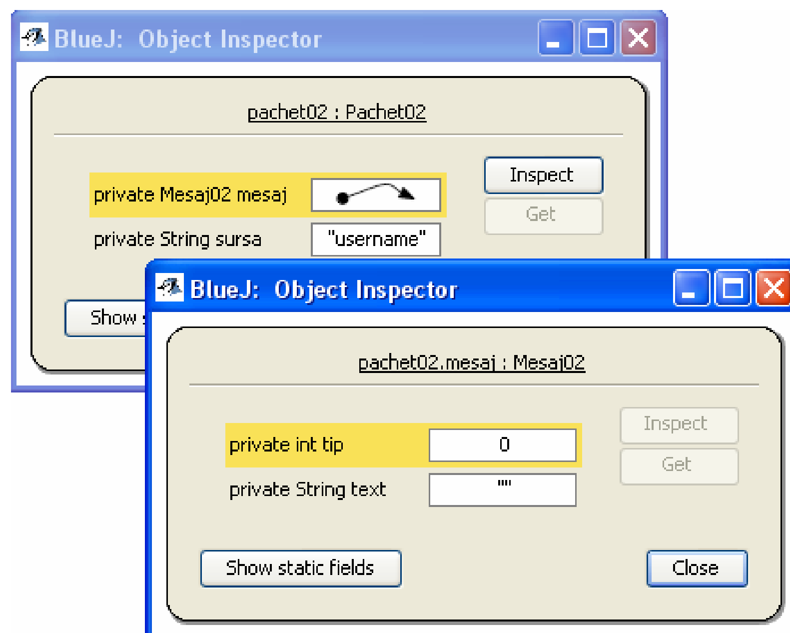


In laborator:

1. **Inspectati ambele obiecte**.







In laborator:

1. **Apelati** metodele **getMesaj**(), **getSursa**() si **toString**() pentru obiectul tip **Pachet02**.
2. **Creati un nou obiect** tip **Pachet02**.
3. **Apelati** metoda **equals**() **a primului obiect** folosind **ca parametru al doilea obiect** tip **Pachet02**.

In cazul clasei **Pachet02**, **supraincarcarea numelui constructorului** ar insemna **crearea unui constructor suplimentar**, de exemplu unul cu semnatura:

```
public Pachet02(Mesaj02 mesaj)
```

Pentru a obtine numele de cont in care se lucreaza (`"user.name"`) **se poate utiliza apelul :**

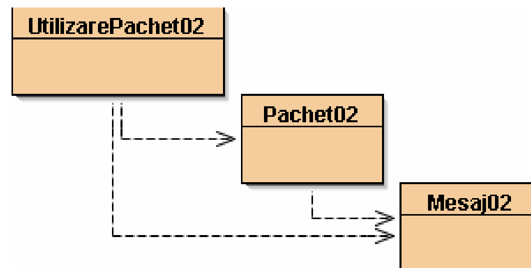
```
String numeUtilizator = System.getProperties().getProperty("user.name");
```

In documentatia (API-ul) claselor Java pot fi gasite si alte proprietati care pot fi obtinute cu apelul `System.getProperties().getProperty()` (`java.home`, `java.class.path`, `java.library.path`, `os.name`, `user.home`, `user.dir`).

In laborator (*codul complet este parte din tema de casa!*):
1. **Editati codul** clasei **Pachet02** si **adaugati un constructor cu semnatura de mai sus, care:**
   - sa initializeze campul numit **mesaj cu parametrul primit,**
   - sa initializeze campul numit **sursa cu numele de cont in care se lucreaza** (vezi mai sus).
2. **Compilati codul** si **creati un obiect** tip **Mesaj02.**
3. **Creati 2 obiecte** tip **Pachet02, fiecare cu cate un constructor. Inspectati obiectele.**
4. **Care este valoarea campului sursa** in cazul obiectului creat cu **constructorul cu un parametru**?

O clasa Java pentru testarea claselor Pachet02 si Mesaj02:

```
public class UtilizarePachet02 {
   public static void main(String[] args) {
       Pachet02 p = new Pachet02();
       System.out.println(p.toString());
   }
}
```



In laborator:
1. **In proiectul** ***mesaj*** **creati o noua clasa** numita **UtilizarePachet02, folosind codul de mai sus.**
2. **Compilati codul.** Ce observati?
3. **Cum puteti rescrie codul clasei UtilizarePachet02 pentru a elimina eroarea la compilare**?
4. **Dupa corectura, deschideti** succesiv **in editor** clasele **UtilizarePachet02**, **Pachet02** si **Mesaj02** si **modificati optiunea** Implementation (aflata in dreapta-sus) **in** Interface.
5. **Studiati continutul paginilor respective.**

3.4. Mediul de dezvoltare JCreator

O alternativa utila la **BlueJ** (***http://www.bluej.org/***) este mediul **JCreator** (prezentare, *how to* 1, *how to* 2, caracteristici, ***http://www.jcreator.com/***):

- Organizeaza **proiectele** cu usurinta folosind o interfata care se aseamana cu Microsoft Visual Studio.

- Permite definirea propriilor **scheme** color in XML, oferind variante nelimitate de organizare a codului.

- **Impacheteaza** proiectele existente si permite folosirea de profile JDK diferite.

- **Browser**-ul sau faciliteaza vizualizarea proiectelor.

- **Depanarea** se face simplu, cu o interfata intuitiva, fara a fi nevoie de prompt-uri DOS.

- Economiseste timpul consumat pentru configurarea **Classpath** si face aceasta configurare in locul utilizatorului.

- Permite modificarea **interfetei** utilizatorului dupa dorinta acestuia.

- Permite setarea mediului de rulare pentru rularea aplicatiilor ca **applet-uri**, intr-un mediu JUnit sau fereastra DOS.

- Necesita putine **resurse** din partea sistemului si totusi ofera o **viteza** foarte buna.

3.5. Teme pentru acasa

I. Codurile finale ale claselor `DialogUtilizator02`, `Mesaj02` si `Pachet02`.

II. Codurile celor 2 clase date ca tema data trecuta, fiecare continand:

- **cel putin 3 campuri/atribute,**
- **cel putin 2 constructori,**
- **cel putin 3 metode/operatii.**

Codurile metodelor si constructorilor vor fi complete.

Clasele propuse (inca de data trecuta) sunt:

Nume clasa	Exemplu de declaratie de camp (atribut)	Exemplu de declaratie (semnatura) de metoda (operatie)
Scrisoare	String **destinatar**	void **setDestinatar**(String **nume**)
Mail	String **subject**	void **setSubject**(String **text**)
Masina	String **proprietar**	void **setProprietar** (String **nume**)
Bicicleta	double **vitezaCurenta**	void **setVitezaCurenta**(double **viteza**)
Avion	int **numarMotoareActive**	void **defectiuneMotor**()
Caiet	int **numarFoi**	void **rupereFoaie**()
Clipboard	int **numarFoi**	void **adaugareFoi**(int **foiNoi**)
SalaCurs	int **locuriOcupate**	void **asezareStudent**(String **nume**)
Laborator	int **numarPlatforme**	void **adaugarePlatforma**()

Fiecare student a primit deja un numar de ordine, caruia ii corespunde un set de 2 clase:

Nr. ord.	Setul de clase (tema de casa)	Nr. ord.	Setul de clase (tema de casa)	Nr. ord.	Setul de clase (tema de casa)
1	Scrisoare + Masina	**11**	Mail + Caiet	**21**	Bicicleta + SalaCurs
2	Scrisoare + Bicicleta	**12**	Mail + Clipboard	**22**	Bicicleta + Laborator
3	Scrisoare + Avion	**13**	Mail + SalaCurs	**23**	Avion + Caiet
4	Scrisoare + Caiet	**14**	Mail + Laborator	**24**	Avion + Clipboard
5	Scrisoare + Clipboard	**15**	Masina + Caiet	**25**	Avion + SalaCurs
6	Scrisoare + SalaCurs	**16**	Masina + Clipboard	**26**	Avion + Laborator
7	Scrisoare + Laborator	**17**	Masina + SalaCurs	**27**	Caiet + SalaCurs
8	Mail + Masina	**18**	Masina + Laborator	**28**	Caiet + Laborator
9	Mail + Bicicleta	**19**	Bicicleta + Caiet	**29**	Clipboard + SalaCurs
10	Mail + Avion	**20**	Bicicleta + Clipboard	**30**	Clipboard + Laborator