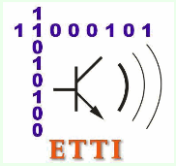
Catedra de Telecomunicatii



Programare Orientata spre Obiecte (POO)

11/11/2010 http://discipline.elcom.pub.ro/**POO-Java**

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:

- Specificarea comportamentului claselor (metode si constructori):
  - **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**

- Studiu de caz: Clasele Grupa, DatePersonale si SituatieCurs si clasa Student actualizata

- Teme de casa

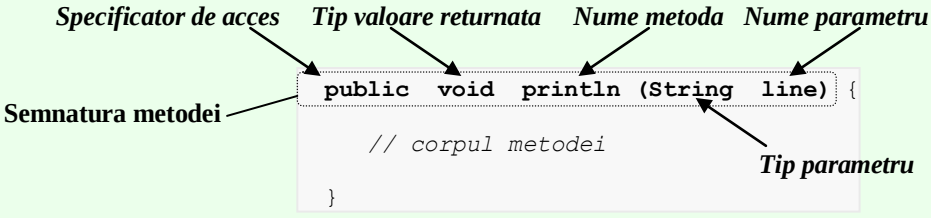
- Anexa. Mediile de dezvoltare **JCreator** si **NetBeans IDE BlueJ Edition**

3.2. Metode si constructori. Supraincarcarea numelor

3.2.1. Semnatura metodei. Returnarea valorilor

Dupa **invocare** (apelare) **metodele obiectelor efectueaza sarcini** (in general utilizand argumentele si valorile campurilor obiectului) **care se pot finaliza 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 expresii, atribuiri, decizii, iteratii, apeluri metode). **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**.

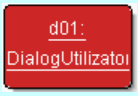
In laborator: Pentru exemplul de mai jos:
1. Identificati **numele metodelor**.
2. Incercati sa determinati **metodele definite de programator** si **metodele bibliotecilor Java**.
3. Identificati **metodele care sunt definite (cu semnatura si corp)** si **metodele care sunt apelate**.
4. **Identificati numele** si **tipul parametrilor** si **valorile argumentelor** in fiecare caz.
5. **Identificati tipul valorilor returnate** in fiecare caz.
6. **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 println(String text) {                 // nu returneaza nici o valoare
       JOptionPane.showMessageDialog(null, text);
   }
}
```

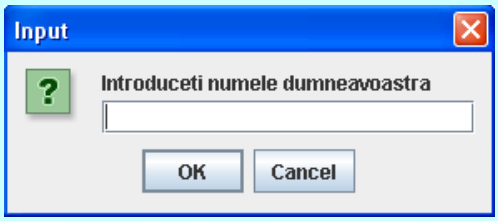
In documentatia (API-ul) claselor Java pot fi gasite detalii privind **clasa `JOptionPane`**.

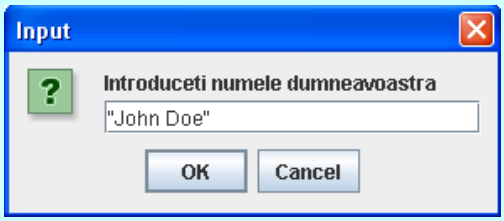
In laborator:
1. **Lansati** mediul **BlueJ**. **Inchideti proiectele anterioare** (cu **Ctrl+W** sau **Project** si **Close**).
2. **Creati un nou proiect** numit ***dialog*** (cu **Project**, apoi **New Project…**, selectati **D:/**, apoi **POO2007**, apoi **numarul grupei**, apoi scrieti **dialog**).
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**.

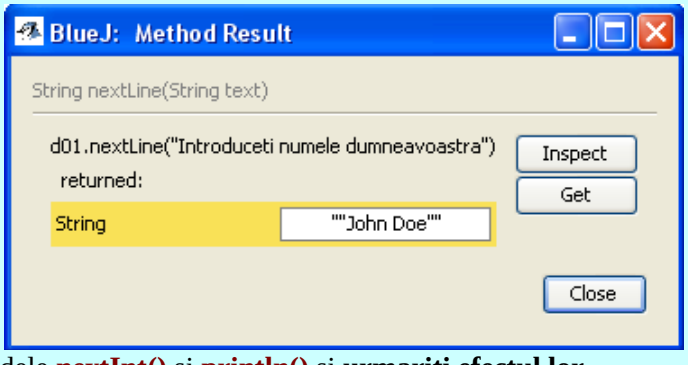


In laborator:
1. **Executati** metoda **nextLine()** dandu-i ca parametru **"Introduceti numele dumneavoastra"**.



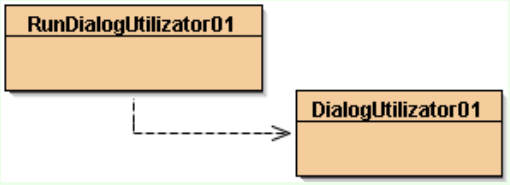


2. **Inspectati valoarea returnata**.



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

Atentie! 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();                // obiect testat
       String linie = d01.nextLine("Introduceti numele dumneavoastra");  // metoda testata
       d01.println("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**.

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**.

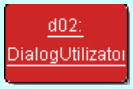
Un exemplu de clasa similara celei anterioare, dar care defineste explicit un constructor:

```
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(this.prompt + text);
       return this.sc.nextLine();
   }
   public int nextInt(String text) {
       System.out.print(this.prompt + text);
       return this.sc.nextInt();
   }
   public void println(String text) {   System.out.println(text);   }
}
```

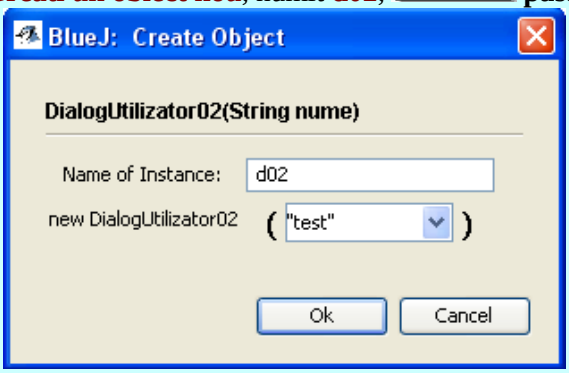
In documentatia (API-ul) claselor Java pot fi gasite detalii privind **clasa `Scanner`**.

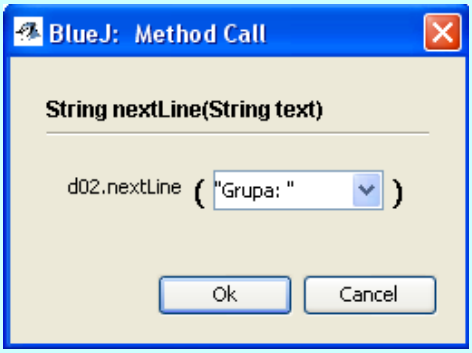
In laborator:

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**.
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 **println()** 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:

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 **println()** 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 **println()** dandu-i ca parametru "`POO`". Ce apare in **Terminal Window**?

In laborator:

1. **Concepeti si editati codul unei metode noi** a clasei **DialogUtilizator02**, **cu semnatura**:
   `public void println()`
   care nu primeste parametru si **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 println()**. Ce apare in **Terminal Window**?
4. **Executati** din nou **vechea metoda println()**, 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 }
```

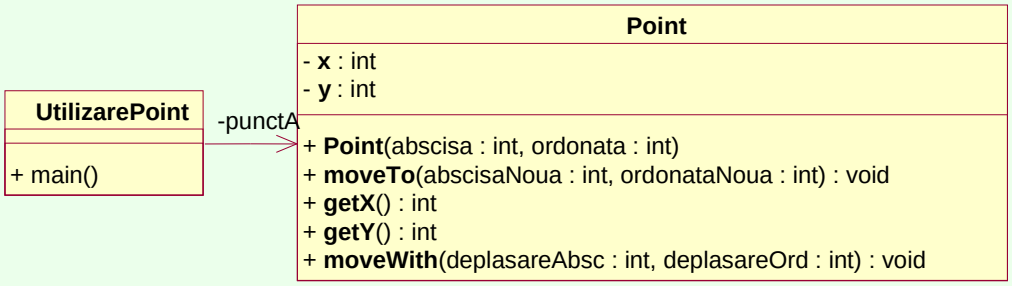
```
1  public class UtilizarePoint {
2     private static Point punctA;      // referinta, legatura catre un obiect Point
3
4     public static void main(String[] args) {
5        punctA = new Point(3, 4);      // alocare si initializare atribut punctA
6        punctA.moveTo(3, 5);           // trimitere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

Un obiect sau o clasa "vede" un alt obiect daca are o referinta catre el, si astfel **ii poate apela metodele**. Se spune ca exista o **legatura intre obiectul care are referinta catre obiectul referit**.

De exemplu, clasa **`UtilizarePoint`** are o referinta **`punctA`** catre un obiect al clasei **`Point`**.

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

Asocierea este o relatie care **exprimă un cuplaj (o dependenta) redus între clase** (clasele asociate rămânând relativ independente). **Clasele `Point`** si **`UtilizarePoint`** sunt de exemplu intr-o **relatie de asociere (cu navigabilitate) unidirectionala**:

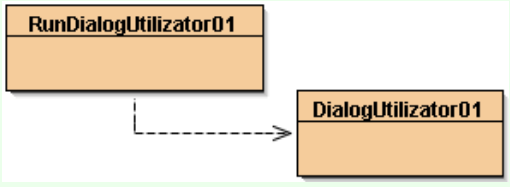


Clasa **`UtilizarePoint`** are un atribut **`punctA`** de tip **`Point`** care permite clasei **`UtilizarePoint`** sa trimita mesaje unui obiect (**`pointA`**) al clasei **`Point`**.

```
   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).

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**:



3.3. Studiu de caz: clasele Mesaj si Pachet

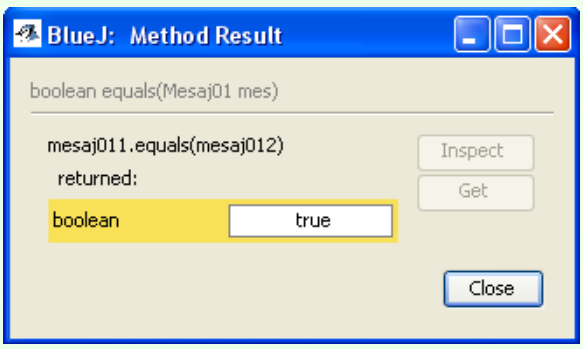
3.3.1. Structura de baza a clasei Mesaj: campuri, constructori, metode

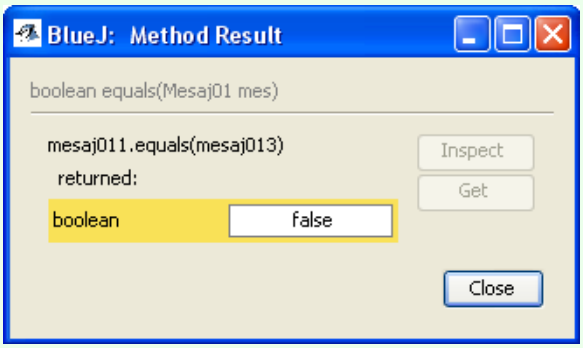
Clasa Mesaj01 incapsuleaza un obiect de tip String care reprezinta un mesaj de la utilizatorul curent (**regrupand textul mesajului cu metodele prin care este controlat accesul la acesta**):

```
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(Object obj) {
       return this.text.equals(((Mesaj01)obj).text);
   }
}
```

In laborator:

1. **Lansati** mediul **BlueJ**. **Inchideti toate proiectele** (**Ctrl+W**). **Creati un proiect** numit ***mesaj***.
2. **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:

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(Object obj) 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** metoda **toString**() 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 adaugand 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**.

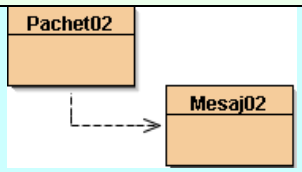
In laborator

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

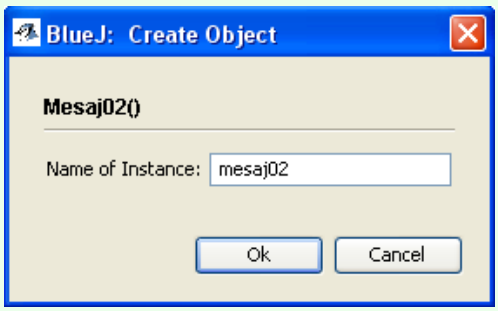
Pentru a exemplifica **relatia de utilizare intre clase** va fi creata **o clasa Pachet02** care **incapsuleaza un obiect Mesaj02** (**regrupand mesajul si sursa lui cu metodele prin care este controlat accesul la acestea**):

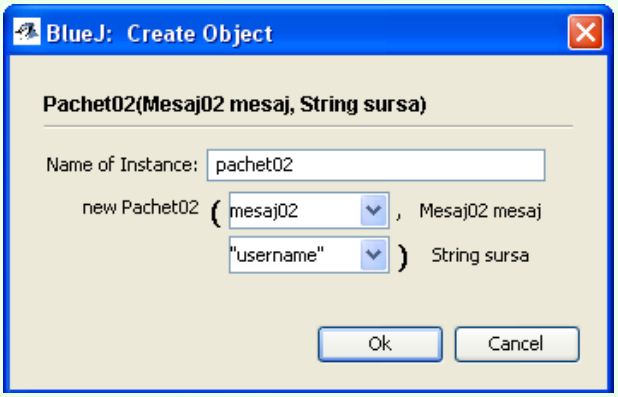
```
public class Pachet02 {
   private Mesaj02 mesaj;
   private String sursa;
   public Pachet02(Mesaj02 mesaj, String sursa) {
       this.mesaj = mesaj;
       this.sursa = sursa;
   }
   public Mesaj02 getMesaj() { return this.mesaj; }
   public String getSursa()  { return this.sursa; }
   public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



Optional, in laborator:

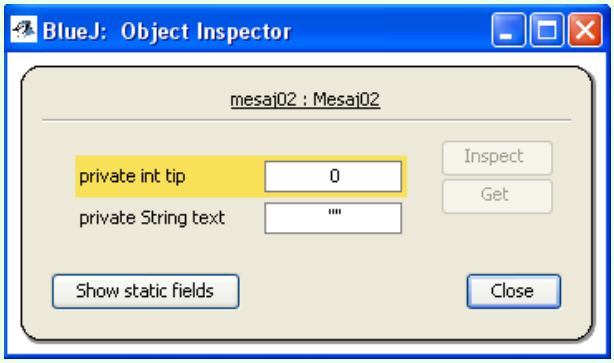
1. **In proiectul *mesaj* creati o noua clasa** numita **Pachet02**, folosind codul de mai sus.
2. **Compilati codul** si **creati mai intai un obiect** tip **Mesaj02** si **apoi un obiect** tip **Pachet02** (**pasandu-i** constructorului **Pachet02(Mesaj02 mesaj, String sursa) obiectul** tip **Mesaj02**).

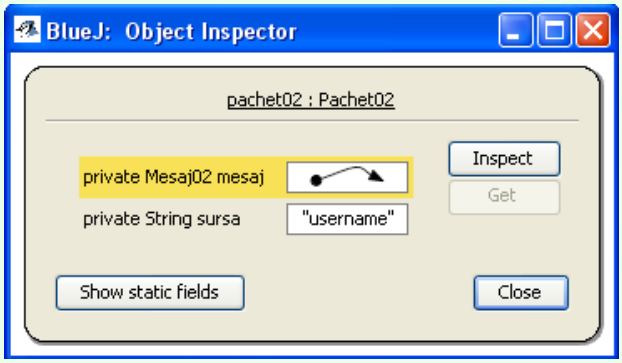




Optional, in laborator:

1. **Inspectati ambele obiecte**.





Optional, 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");
```

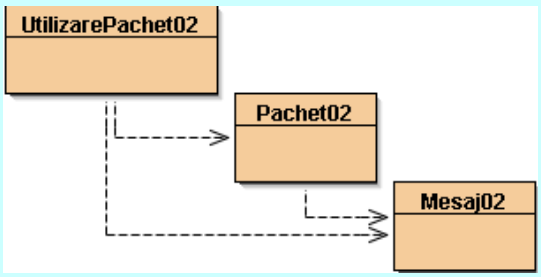
In documentatia (API-ul) claselor Java pot fi gasite si **alte proprietati care pot fi obtinute cu apelul** `System.getProperties().getProperty()` (`java.home`, `java.class.path`, `os.name`, `user.home`).

Optional, in laborator

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());
   }
}
```



Optional, 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. Studiu de caz: clasele Grupa, Student, DatePersonale si SituatieCurs

3.4.1. Clasele DatePersonale si SituatieCurs si actualizarea clasei Student

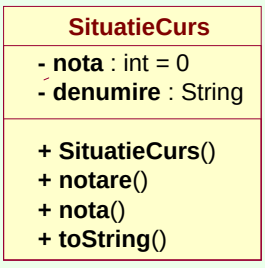
Sa presupunem ca dorim sa **modificam codul clasei** `Student` care **abstractiza un student real, introducand detalii suplimentare** referitoare la `Student` ca persoana, pe langa campul `nume`.

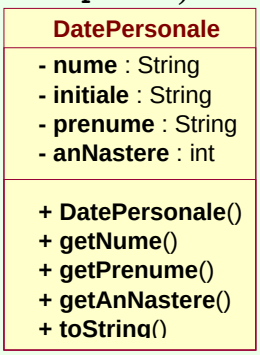
De exemplu, putem inlocui campul `nume` de tip `String` cu un camp `date` de un tip nou, `DatePersonale`, care va fi **o noua clasa** ce va contine pe langa un camp `nume` de tip `String` si campurile `initiale` si `prenume` de tip `String` si `anNastere` de tip `int`. **Codul noii clasei** va fi:

```
 1  public class DatePersonale {
 2     // Campuri ascunse
 3     private String nume;
 4     private String initiale;
 5     private String prenume;
 6     private int anNastere;
 7
 8     // Constructori
 9     public DatePersonale(String n, String i, String p, int an) {
10        nume = new String(n);       // copiere „hard" a obiectelor primite ca parametri,
11        initiale = new String(i);   // adica se copiaza obiectul camp cu camp,
12        prenume = new String(p);    // nu doar referintele ca pana acum
13        anNastere = an;
14     }
15     // Interfata publica si implementarea ascunsa
16     public String getNume() {       return (nume);    }
17     public String getPrenume() {    return (prenume);    }
18     public int getAnNastere() {     return (anNastere);    }
19     public String toString() {     // forma „String" a campurilor obiectului
20        return (nume + " " + initiale + " " + prenume + " (" + anNastere + ")");
21     }
22  }
```

De asemenea, presupunem ca dorim sa **modificam codul clasei** `Student` **regrupand elementele pereche ale campurilor** `cursuri` si `rezultate` **(care sunt tablouri) in obiecte ale unei clase noi,** `SituatieCurs`. Vom inlocui in clasa `Student` tablourile `cursuri` cu elemente de tip `String` si `rezultate` cu elemente de tip `int`, cu un singur tablou, `cursuri`, cu elemente de tip `SituatieCurs`.

Cele doua clase noi pot fi **reprezentate in UML** (**-** = acces `private`, **+** = acces `public`) astfel:





Codul Java al noii clasei va fi:

```
 1  public class SituatieCurs {
 2     // Campuri ascunse
 3     private int nota = 0;                          // initializare implicita
 4     private String denumire;
 5
 6     // Constructor
 7     public SituatieCurs(String d) { denumire = new String(d); }     // copiere „hard"
 8                                                   // se initializeaza doar denumire
 9     // Interfata publica si implementarea ascunsa
10     public void notare(int n) {     nota = n; }     // se adauga nota
11     public int nota() {             return(nota); } // se returneaza nota
12     public String toString() {                      // forma „String" a campurilor
13        if (nota==0) return ("Disciplina " + denumire + " nu a fost notata");
14        else return("Rezultat la disciplina " + denumire + ": " + nota);
15     }
16  }
```

In cele doua clase, se observa ca **se foloseste copierea "hard" a obiectelor primite ca parametri**, adica crearea unor **copii ale obiectelor argument**, camp cu camp, **nu doar copierea referintelor**.

In clasa **`SituatieCurs`** nu am utilizat metode de tip **`getCamp()`** si **`setCamp()`**. Metoda **`nota()`**ar fi putut fi denumita **`getNota()`** iar metoda **`notare()`** ar fi putut fi denumita **`setNota()`**.

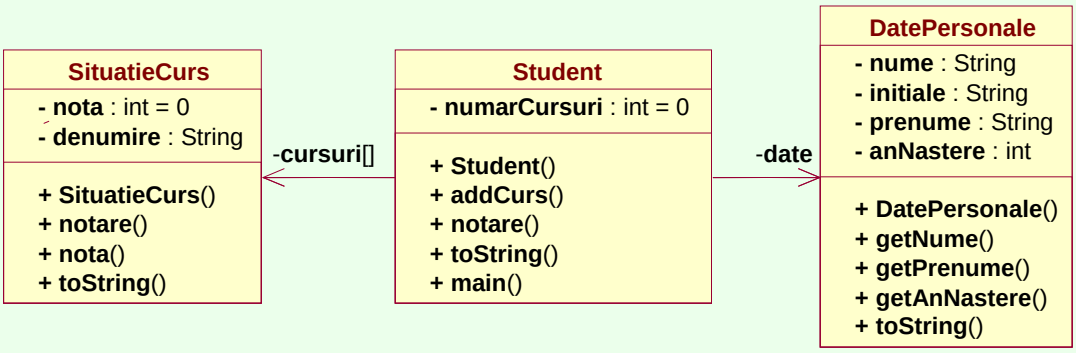
Printre metodele declarate regasim si **`toString()`**, **cu scopul de a returna sub forma de `String` informatiile pe care le incapsuleaza obiectul caruia i se aplica**. Nota initiala **0** inseamna notei (notarea poate incepe doar cu **1**). De aceea **`toString()` returneaza diferit** pentru valori nule/nenule.

Vom rescrie acum codul Java al clasei **`Student`** pentru a **incorpora schimbarile anuntate**.

```
1  /**
2     * Incapsuleaza informatiile despre un Student. Permite testarea locala.
3     * @version 1.3
4     */
5    public class Student {
6     // Campuri ascunse
7     private DatePersonale date;
8     private SituatieCurs[] cursuri;
9     private int numarCursuri = 0;              // initializare implicita
10
11    // Constructori
12    public Student(String nume, String initiale, String prenume, int anNastere) {
13       date = new DatePersonale(nume, initiale, prenume, anNastere); // copiere „hard"
14       cursuri = new SituatieCurs[10];         // se initializeaza doar date si cursuri
15    }
16
17    // Interfata publica si implementarea ascunsa (include punct intrare program)
18    public void addCurs(String nume) {         // se adauga un nou curs
19       cursuri[numarCursuri++] = new SituatieCurs(nume);
20    }
21    public void notare(int numarCurs, int nota) {
22       cursuri[numarCurs].notare(nota);        // se adauga nota cursului specificat
23    }
24    public String toString() {                 // forma „String" a campurilor
25       String s = "Studentul " + date + " are urmatoarele rezultate:\n";
26       for (int i=0; i<numarCursuri; i++)   s = s + cursuri[i].toString() + "\n";
27       return (s);
28    }
29    public static void main(String[] args) {
30       // Crearea unui nou Student, initializarea campurilor noului obiect
31       Student st1 = new Student("Xulescu", "Ygrec", "Z.", 1987);
32       st1.addCurs("CID");
33       st1.addCurs("MN");
34       st1.notare(0, 8);
35       // Utilizarea informatiilor privind Studentul
36       System.out.println(st1.toString());     // afisarea formei „String" a campurilor
37    }
```

Regasim **copierea "hard" a obiectelor** si metoda **`toString()`**. Tabloul cursurilor este initial gol, rand pe rand fiind adaugate cursuri noi (si e incrementat numarul lor).

In UML avem asocierile (clasa **`Student`** utilizeaza **`DatePersonale`** si **`SituatieCurs`**):



Clasa are o metoda **`main()`** cu rolul de a testa lucrul cu obiectele **`Student`**. Executia ei conduce la:

```
Studentul Xulescu A. Ygrec (1987) are urmatoarele rezultate:
Rezultat la disciplina CID: 8
Disciplina MN nu a fost notata
```

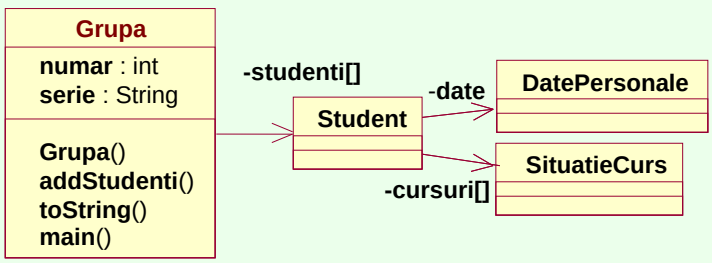
3.4.2. Clasa Grupa

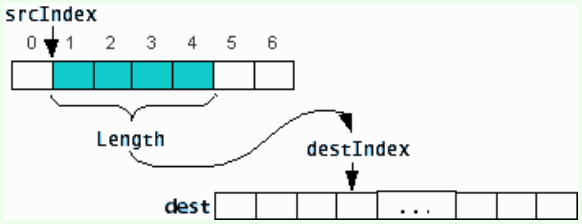
Sa presupunem ca dorim sa **scriem codul unei clase noi** numita **`Grupa`** care **sa abstractizeze o grupa de studenti** in cadrul programului care gestioneaza informatii intr-o universitate, facultate, etc.

Clasa va avea **campuri separate** pentru **`serie`**, de care tine, si **`numar`**, care o identifica (cele doua ar putea forma o alta clasa, de exemplu **`InfoGrupa`**) si un **tablou pentru referinte** spre **`studenti`**.

```
 1  public class Grupa {
 2     // Campuri ascunse
 3     private int numar;
 4     private String serie;
 5     private Student[] studenti;
 6
 7     // Constructori
 8     public Grupa(int nr, String sr) {          // se initializeaza doar serie si numar
 9        numar = nr; serie = new String(sr);     // copiere „hard” pentru serie
10     }
11
12     // Interfata publica si implementarea ascunsa (include punct intrare program)
13     public void addStudenti(Student[] st) {    // se adauga tabloul de studenti
14        studenti = new Student[st.length];
15        System.arraycopy (st, 0, studenti, 0, st.length);  // copiere „hard” a tabloului
16     }
17     public String toString() {
18        String g = "\nRezultatele grupei " + numar + serie + ":\n";
19        for (int i=0; i<studenti.length; i++)
20          g = g + studenti[i].toString() + "\n";
21        return (g);
22     }
23     public static void main(String[] args) {
24        // Crearea unui nou Student, initializarea campurilor noului obiect
25        Student st1 = new Student("Xulescu", "A.", "Ygrec", 1987);
26        st1.addCurs("CID");
27        st1.addCurs("MN");
28        st1.notare(0, 8);
29        // Crearea unui nou Student, initializarea campurilor noului obiect
30        Student st2 = new Student("Zulescu", "B.", "Ics", 1988);
31        st2.addCurs("CID");
32        st2.addCurs("MN");
33        st2.notare(1, 9);
34        // Crearea unei noi Grupe, initializarea campurilor noului obiect
35        Grupa g1 = new Grupa(424, "A");
36        Student[] st = {st1, st2};
37        g1.addStudenti(st);
38        // Utilizarea informatiilor privind Grupa
39        System.out.println(g1.toString());
40     }
41  }
```

Asocierile in UML (clasa **`Grupa`** utilizeaza **`Student`**):





Pentru copiere „hard” a tablourilor (element cu element), clasa **`System`** ofera metoda **`arraycopy()`**, care **copiaza un subtablou din tabloul sursa `src`**, de lungime **`length`**, incepand de la index **`srcPos`**, **in tabloul destinatie `dest`**, la index **`destPos`**.

Semnatura **`arraycopy()`** este:

```
static void arraycopy(Object src, int srcPos, Object dest, int destPos, int length)
```

Metoda **`main()`** testeaza lucrul cu obiectele **`Grupa`** si **`Student`**. Executia ei are ca efect:

```
Rezultatele grupei 424A:
Studentul Xulescu A. Ygrec (1987) are urmatoarele rezultate:
Rezultat la disciplina CID: 8
Disciplina MN nu a fost notata
Studentul Zulescu B. Ics (1988) are urmatoarele rezultate:
Disciplina CID nu a fost notata
Rezultat la disciplina MN: 9
```

3.5. Teme pentru acasa

Temele vor fi **predate la lucrarea urmatoare,** cate **un exemplar** pentru fiecare **grup de 2 studenti, sub forma de listing, continand** atat **codurile sursa** cat si **screenshot-uri ale ecranului BlueJ** in care **sa se poata vedea mesajele generate de program**, si avand **numele** celor doi studenti scrise pe prima pagina sus.

Tema obligatorie: **Codurile sursa ale unor clase create dupa modelul din sect 3.4 (`DatePersonale`, pag 9, `SituatieCurs`, pag 9, si `Grupa`, pag 11) cu urmatoarea specificatie:**

- va fi **creata o clasa care este utilizata de clasa primita ca tema la lucrarea a 2-a,** cu numele alocat din tabelul care urmeaza, dupa modelul claselor **`DatePersonale` si `SituatieCurs`**, **fie prin regruparea fie prin detalierea unor campuri** ale clasei primite ca tema la lucrarea a 2-a,
  - noua clasa va contine **2-3 campuri private**, un **constructor public**, **2-3 metode publice** pentru lucrul cu campurile **si o metoda** publica de tip **`toString()`**,
- va fi **modificata clasa primita ca tema la lucrarea a 2-a** pentru a putea **utiliza obiecte ale clasei nou create**, **si i se va adauga o metoda** publica **`toString()`**,
  - metoda ei **`main()` va afisa** ceea ce returneaza metoda **`toString()`**,
- va fi **creata o clasa care utilizeaza clasa primita ca tema la lucrarea a 2-a,** cu numele alocat din tabelul care urmeaza, dupa modelul clasei **`Grupa`**,
  - noua clasa va contine **2-3 campuri private**, un **constructor public**, **2-3 metode publice** pentru lucrul cu campurile **si o metoda** publica de tip **`toString()`**,
  - metoda ei **`main()` va afisa** ceea ce returneaza metoda **`toString()`**,

Numele claselor propuse corespunzatoare **numerelor de ordine alocate la lucrarea a 2-a** sunt:

	Clasa care utilizeaza (ca `Grupa`)	Clasa initiala (ca `Student`)	Clasa utilizata (ca `DatePersonale`)
1	`Magazin`	`Monitor + Televizor`	`Ecran`
2	`SalaCurs`	`Proiector + TablaInteractiva`	`Slideuri`
3	`Husa`	`AparatFoto + CameraVideo`	`Obiectiv`
4	`Firma`	`Sofer + Pilot`	`Licenta`
5	`Service`	`Telefon + PDA`	`CardSD`
6	`Geanta`	`PointerLaser + Lanterna`	`Baterie`
7	`Laptop`	`StickUSB + HardDiskExtern`	`Producator`

	Clasa care utilizeaza (ca `Grupa`)	Clasa initiala (ca `Student`)	Clasa utilizata (ca `DatePersonale`)
8	`Firma`	`Magazin + MagazinOnline`	`Produs`
9	`Parcare`	`Autocar + Microbuz`	`Locuri`
10	`Garaj`	`Scuter + Motocicleta`	`Motor`
11	`Patron`	`Blog + Ziar`	`Articol`
12	`PC`	`CDROM + CDAudio`	`Producator`
13	`Utilizator`	`Messenger + Twitter`	`Mesaj`
14	`Departament`	`ProjectManager + Proiectant`	`Proiect`

Membrii fiecarui grup vor crea fie **coduri separate care sunt utilizate de / utilizeaza clasa aleasa data trecuta**, fie **un cod comun care este utilizat de / utilizeaza ambele clase ale grupului**.

Teme suplimentare, pentru bonus:

I. Codurile temei obligatorii rescrise pentru a lucra cu obiecte de tip ArrayList (ArrayList<E>)

II. Adaugarea altor metode claselor create, dandu-le astfel un caracter mai aplicativ.

Anexe

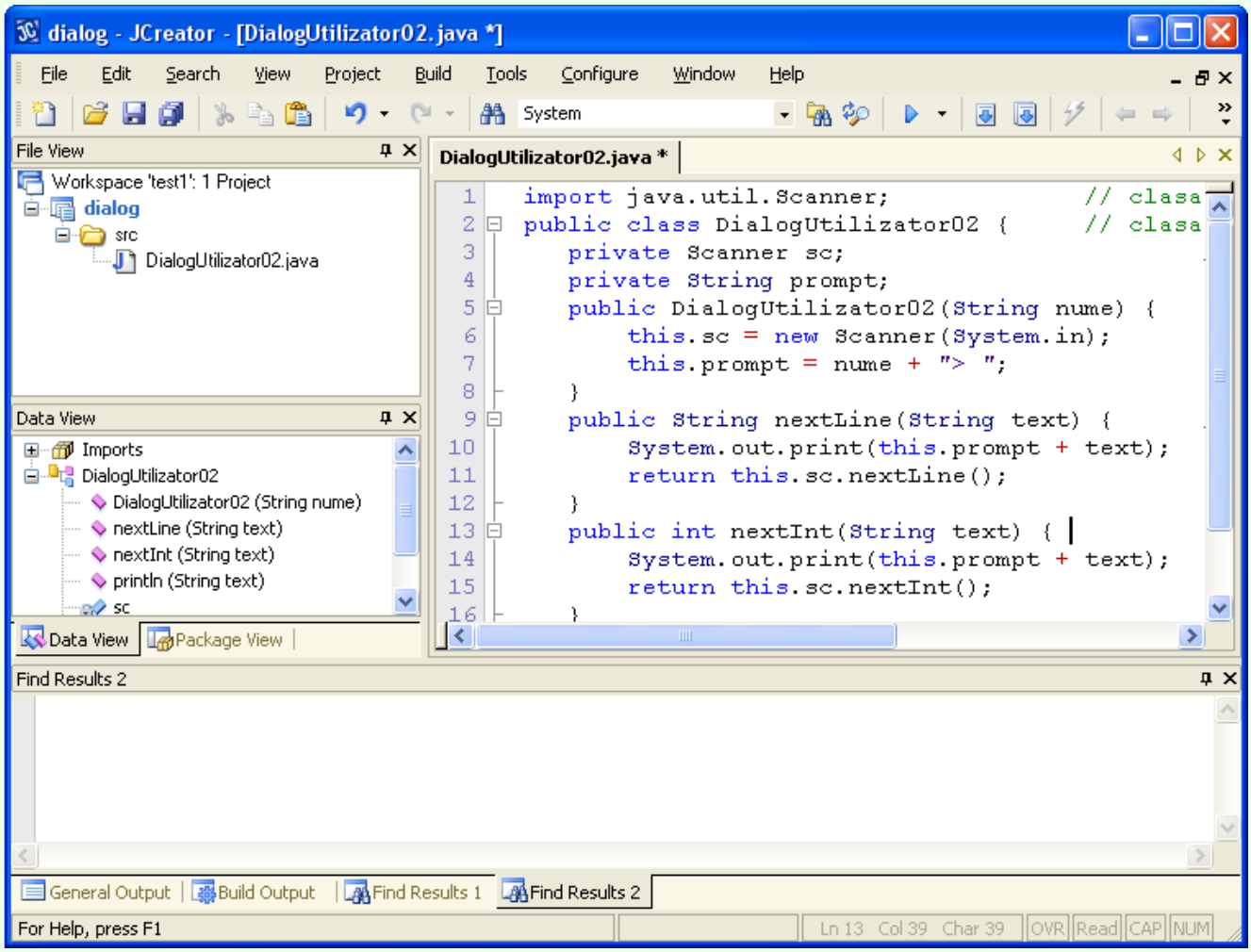
1. Mediul de dezvoltare JCreator

O alternativa utila la **BlueJ** (*http://www.bluej.org/*) este mediul **JCreator** (*http://www.jcreator.com/*).

Principalele caracteristici ale mediului JCreator:

- 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.

Interfata grafica a mediului **Jcreator 3.50 LE**:



2. Mediul de dezvoltare NetBeans IDE BlueJ Edition

Mediul de dezvoltare **NetBeans IDE BlueJ Edition** este un **hibrid** intre **NetBeans** **si** **BlueJ**, un **IDE modular scris in limbajul de programare Java**.

Cateva **tutoriale si alte documentatii** despre **NetBeans IDE BlueJ Edition**:

- pagina de **Tutoriale**
- pagina de blog **A note on how to enable code completion** a lui ***Greg Sporar*** (04.10.2006)
- un **video** despre NetBeans/BlueJ creat de ***Sun Developer Network***
- un set de **note de laborator** (**Lab Notes to help transition from BlueJ**)

Kitul de instalare NetBeans IDE 5.5.1 BlueJ Edition poate fi gasit la adresele:

- versiune Windows: *NetBeansBlueJ-5.5.1-**win**-ml.exe*
- versiune Linux: *NetBeansBlueJ-5.5.1-**linux**-ml.bin*

Interfata grafica a mediului **NetBeans IDE 5.5.1 BlueJ Edition**: