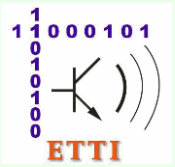
Catedra de Telecomunicatii



Programare Orientata spre Obiecte (POO)

21/11/2010 http://discipline.elcom.pub.ro/**POO-Java**

Laborator 4

Extinderea prin mostenire. Rescrierea codului (*overriding*). *Socket*-uri flux (TCP) Java

4.1. Descrierea laboratorului

In aceasta lucrare de laborator vor fi acoperite urmatoarele probleme:

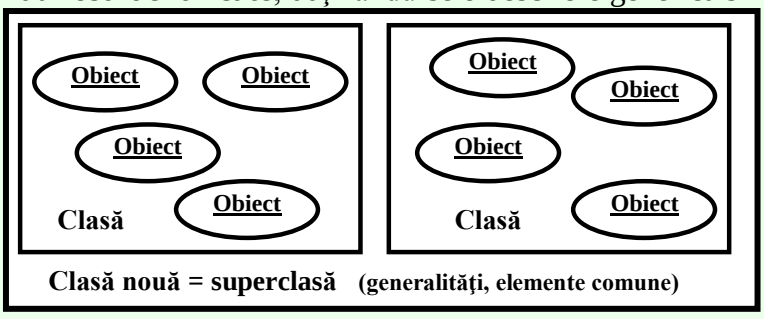
- **Programare OO:** - **Generalizarea si specializarea claselor**.
  - **Mostenirea. Rescrierea codului (*overriding*)**
- **Socket-uri:** - **Incapsularea adreselor IP in Java** (clasa `InetAddress`)
  - ***Socket-uri* pentru fluxuri TCP** (clasa `Socket`) Introducere in *socket*-uri -**link extern**
  - ***Socket-uri* pentru servere TCP** (clasa `ServerSocket`)
- **Studiu de caz: Aplicatii client-server bazate pe socket-uri TCP care folosesc mostenirea**
- **Clasele Persoana, Student, Profesor, DatePersonale, SituatieCurs si StudentMaster**
- **Teme de casa**

4.2. Extinderea prin mostenire

4.2.1. Generalizarea si specializarea claselor

Generalizarea reprezinta **extragerea elementelor comune** (atribute, operaţii şi constrângeri) ale unui ansamblu de clase **într-o nouă clasă mai generală** numită **superclasă** (care reprezinta o abstracţie a subclaselor ei).

Rezulta o **ierarhie** in care **arborii de clase** sunt **construiţi pornind de la frunze**, **atunci cand elementele modelului au fost identificate**, obţinandu-se o descriere generica si flexibila a soluţiilor.



Generalizarea semnifică relatia de tip "**este un**" atunci cand un obiect dintr-o clasa din ansamblul generalizat este **in acelasi timp si** obiect al superclasei, si de tip "**este un fel de**" atunci cand un obiect dintr-o clasa din ansamblul generalizat este **doar aproximativ si** obiect al superclasei (aproximatia venind din rescrierile codurilor operate in clasa generalizata).

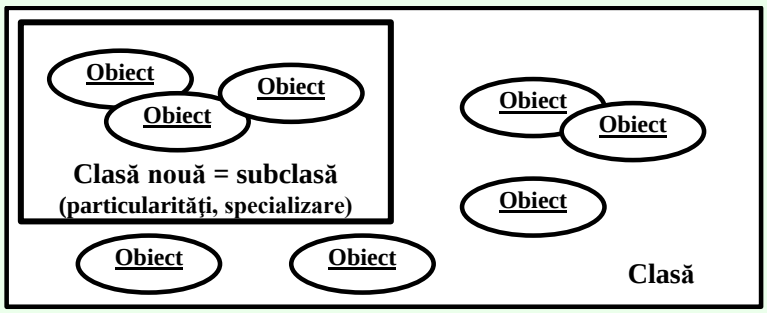
Generalizarea actioneaza in orientarea spre obiecte **la doua niveluri:**

- **clasele sunt generalizari ale ansamblurilor de obiecte** (un obiect este **de felul** definit de o clasa),
- **superclasele sunt generalizari ale unor clase** (obiectele **de felul** specificat intr-o clasa sunt in acelasi timp si **de felul** specificat in superclasa).

Orientarea spre obiecte (OO) presupune **ambele tipuri de generalizare**, iar **limbajele orientate spre obiecte** sunt acelea care **ofera ambele mecanisme de generalizare**. Limbajele care ofera doar constructii numite obiecte (si eventual clase) se pot numi **limbaje care lucreaza cu obiecte** (si eventual clase).

Specializarea claselor reprezinta **capturarea particularităţilor** unui ansamblu de obiecte nediscriminate ale unei clase existente, noile caracteristici fiind reprezentate **într-o nouă clasă mai specializată**, denumită **subclasă.**

Specializarea usureaza **extinderea coerentă a unui ansamblu de clase,** noile cerinţe fiind încapsulate în subclase care extind funcţiile existente.



În **elaborarea unei ierarhii de clase**, se cer diferite **aptitudini sau competenţe**:
- pentru **generalizare: capacitate de abstractizare**, independentă de cunoştinţele tehnice,
- pentru **specializare: experienţă si cunoştinţe aprofundate** într-un domeniu particular.

4.2.2. Mostenirea. Rescrierea codului metodelor (*overriding*)

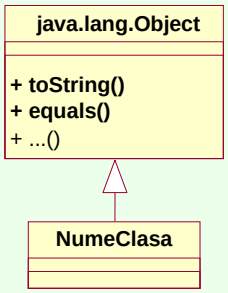
Moştenirea este o **tehnică de generalizare** oferită de limbajele de programare orientate spre obiecte pentru a construi o clasă pornind de la una sau mai multe alte clase, **partajând atributele si operaţiile** (campurile si metodele, in Java) **într-o ierarhie de clase**.

In limbajul Java, orice clasa care nu extinde in mod explicit (prin mostenire) o alta clasa Java, **extinde (prin mostenire)** in mod **implicit clasa** `Object` (radacina ierarhiei de clase Java), clasa care contine metodele necesare tuturor obiectelor create din ierarhia de clase Java. Urmatoarele doua **declaratii de clasa** sunt **echivalente**:

```
    class NumeClasa {                  // urmeaza corpul clasei ...
```

```
    class NumeClasa extends Object { // urmeaza corpul clasei ...
```

Notatia UML pentru extinderea prin mostenire este o **linie care uneste clasa extinsa** (de baza, superclasa) **de clasa care extinde** (subclasa), linie **terminata cu un triunghi in capatul dinspre clasa de baza**. **Diagrama UML** corespunzatoare **codului Java** anterior:



Metoda `toString()`, metoda care **are ca scop returnarea sub forma de** `String` **a informatiilor pe care le incapsuleaza obiectul caruia i se aplica aceasta metoda**.

1. In cazul claselor de biblioteca Java, metoda `toString()` returneaza ansamblul valorilor curente ale atributelor obiectului.

2. In cazul claselor scrise de programator, in mod implicit metoda `toString()` returneaza numele clasei careia ii apartine obiectul urmat de un cod alocat acelui obiect (*hashcode*).

Implementarea implicita a metodei `toString()` este urmatoarea:

```
1   // Implementarea implicita a metodei toString(),
2   // mostenita de la clasa Object
3
4   public String toString() {
5      // (nu returneaza continutul ci numele clasei si codul obiectului!)
6      return getClass().getName() + "@" + Integer.toHexString(hashCode());
7   }
```

3. In cazul in care programatorul doreste returnarea informatiilor incapsulate in obiect, trebuie specificat in mod explicit un nou cod (o noua implementare) pentru metoda **`toString()`**. Acest lucru se obtine adaugand clasei din care face parte acel obiect o metoda cu declaratia:

```
    public String toString() { // urmeaza corpul metodei ...
```

metoda care se spune ca **rescrie (*overrides*) codul metodei** cu acelasi nume din clasa extinsa (in acest caz clasa **`Object`**).

Dupa adaugarea acestei metode, apelul **`toString()` va conduce la executia noului cod**, pe cand apelul **`super.toString()` va duce la executia codului din clasa extinsa** (**superclasa**, **`Object`**).

Metoda `equals()` are ca scop compararea continutului obiectului primit ca parametru cu continutul obiectului caruia i se aplica aceasta metoda, returnand valoarea booleana **`true`** in cazul egalitatii si valoarea booleana **`false`** in cazul inegalitatii celor doua obiecte.

1. In cazul claselor de biblioteca Java, metoda **`equals()`** compara ansamblul valorilor curente ale atributelor obiectului (continutul sau starea obiectului). Iata, de exemplu, implementarea metodei **`equals()`** in cazul clasei **`String`**.

```
1   // Implementarea explicita a metodei equals() in clasa String
2
3   public boolean equals(Object obj) {
4     // se verifica existenta unui parametru (obiect) non-null
5     // si faptul ca parametrul e obiect al clasei String
6     if ((obj != null) && (obj instanceof String)) {
7       String otherString = (String)obj;          // conversie de tip
8       int n = this.count;
9       if (n == otherString.count) {   // se compara numarul de caractere
10        char v1[] = this.value;
11        char v2[] = otherString.value;
12        int i = this.offset;
13        int j = otherString.offset;
14        while (n-- != 0)
15          if (v1[i++] != v2[j++]) return false; // se compara caracterele
16        return true;
17      }
18    }
19    return false;
20  }
```

2. In cazul claselor scrise de programator, in mod implicit metoda **`equals()`** compara referinta obiectului caruia i se aplica aceasta metoda cu referinta obiectului pasat ca parametru. Implementarea implicita a metodei **`equals()`** este urmatoarea:

```
1   // Implementarea implicita a metodei equals(),
2   // mostenita de la clasa Object
3
4   public boolean equals(Object obj) {
5     return (this == obj);     // (nu compara continutul ci referintele!!!)
6   }
```

3. In cazul in care programatorul doreste compararea informatiilor incapsulate in obiect, (ansamblul valorilor curente ale atributelor obiectului) **trebuie specificat in mod explicit un nou cod (o noua implementare)** pentru metoda **`equals()`**. Acest lucru se obtine adaugand clasei din care face parte acel obiect o metoda cu declaratia:

```
    public boolean equals(Object obj) { // urmeaza corpul metodei ...
```

metoda care **rescrie (*overrides*) codul metodei** cu acelasi nume din clasa extinsa (**`Object`**).

Dupa adaugarea acestei metode, apelul **`equals()` va conduce la executia noului cod**, pe cand apelul **`super.equals()` duce la executia codului din clasa extinsa** (in acest caz codul din **`Object`**).

4.3. Introducere in socket-uri Java

4.3.1. Utilizarea clasei java.net.InetAddress

Java ofera, in pachetul **`java.net`**, mai multe **clase pentru comunicatii in retele bazate pe IP** (*Internet Protocol*). Clasa **`InetAddress`** **incapsuleaza o adresa IP** intr-un obiect care **poate intoarce informatia utila**. Aceasta informatie utila se obtine invocand metodele unui obiect al acestei clase. De exemplu, **`equals()`** intoarce adevarat daca doua obiecte reprezinta aceeasi adresa IP.

Clasa `InetAddress` nu are constructor public. De aceea, **pentru a crea obiecte ale acestei clase trebuie invocata una dintre metodele** de clasa **`getByAddress()`** si **`getByName()`**.

O **adresa IP speciala** este **adresa IP *loopback*** (tot ce este trimis catre aceasta adresa IP se intoarce si devine intrare IP pentru gazda locala), **cu ajutorul careia pot fi testate local programe care utilizeaza *socket-uri***. **Pentru a identifica adresa IP *loopback*** sunt folosite numele **`"localhost"`** si valoarea numerica **`"127.0.0.1"`**.

Pentru a obtine **`InetAddress`** **care incapsuleaza adresa IP *loopback*** pot fi folosite apelurile:

```
InetAddress.getByName(null)
InetAddress.getByName("localhost")
InetAddress.getByName("127.0.0.1")
```

4.3.2. Socket-uri flux (TCP) Java

Java ofera, in pachetul **`java.net`**, mai multe clase pentru lucrul cu *socket-uri* flux (TCP). Urmatoarele **clase Java** sunt **implicate in realizarea conexiunilor TCP** obisnuite: `ServerSocket`, `Socket`.

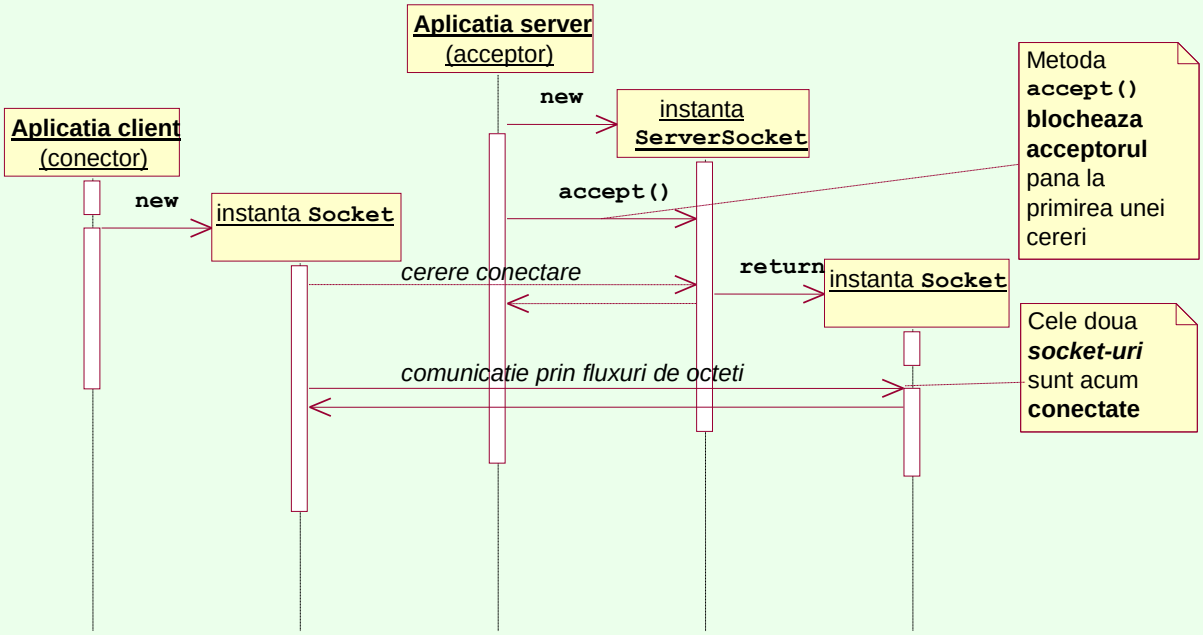
Clasa **`ServerSocket`** reprezinta ***socket-ul*** (aflat eventual pe un server bazat pe TCP) **care asteapta si accepta cereri de conexiune** (eventual de la un client bazat pe TCP).

Clasa **`Socket`** reprezinta **punctul terminal al unei conexiuni TCP** intre doua masini (eventual un client si un server).

Clientul (sau, mai general, **masina conector**) **creeaza un punct terminal `Socket`** in momentul in care cererea sa de conexiune este lansata si acceptata.

Serverul (sau, mai general, **masina acceptor**) **creeaza un `Socket`** in momentul in care primeste si accepta o cerere de conexiune, **si continua sa asculte si sa astepte alte cereri pe `ServerSocket`**.

Secventa tipica a mesajelor schimbate intre client si server este urmatoarea:



Odata conexiunea stabilita, metodele `getInputStream()` si `getOutputStream()` ale clasei `Socket` trebuie utilizate pentru a obtine fluxuri de octeti, de intrare respectiv iesire, pentru comunicatia intre aplicatii.

4.3.3. Utilizarea clasei Socket

Secventa tipica pentru crearea *socket-ului* unei aplicatii conector (client):

```
1       // Stabilirea adresei serverului
2       String adresaServer = "localhost";
3
4       // Stabilirea portului serverului
5       int portServer = 2000;
6
7       // Crearea socketului (implicit este realizata conexiunea cu serverul)
8       Socket socketTCPClient = new Socket(adresaServer, portServer);
```

Dupa utilizare, *socket-ul* este inchis. Secventa tipica pentru inchiderea *socket-ului*:

```
1       // Inchiderea socketului (implicit a fluxurilor TCP)
2       socketTCPClient.close();
```

4.3.4. Utilizarea clasei ServerSocket

Secventa tipica pentru crearea *socket-ului* server al unei aplicatii acceptor (server):

```
1       // Stabilirea portului serverului
2       int portServer = 2000;
3
4       // Crearea socketului server (care accepta conexiunile)
5       ServerSocket serverTCP = new ServerSocket(portServer);
```

Secventa tipica pentru crearea *socket-ului* pentru tratarea conexiunii TCP cu un client:

```
1       // Blocare in asteptarea cererii de conexiune - in momentul acceptarii
2       // cererii se creaza socketul care serveste conexiunea
3
4       Socket conexiuneTCP = serverTCP.accept();
```

Secventa tipica pentru crearea fluxurilor de octeti asociate *socket-ului* (*detalii fluxuri IO*):

```
1       // Obtinerea fluxului de intrare octeti TCP
2       InputStream inTCP = socketTCPClient.getInputStream();
3
4       // Obtinerea fluxului scanner de caractere dinspre retea
5       Scanner scannerTCP = new Scanner(inTCP);
6
7       // Obtinerea fluxului de iesire octeti TCP
8       OutputStream outTCP = socketTCPClient.getOutputStream();
9
10      // Obtinerea fluxului de iesire spre retea (similar consolei de iesire)
11      PrintStream outRetea  = new PrintStream(outTCP);
```

Secventa tipica pentru trimiterea de date:

```
1       // Crearea unui mesaj
2       String mesajDeTrimis = "Continut mesaj";
3
4       // Scrierea catre retea (trimiterea mesajului) si fortarea trimiterii
5       outRetea.println(mesajDeTrimis);
6       outRetea.flush();
```

Secventa tipica pentru primirea de date:

```
1       // Citirea dinspre retea (receptia unui mesaj)
2       String mesajPrimit = inRetea.readLine();
3
4       // Afisarea mesajului primit
5       System.out.println(mesajPrimit);
```

4.4. Programe de lucru cu socket-uri – clienti si servere

4.4.1. Clasele Client si Server (varianta fara mostenire)

Codul unei clase care incapsuleaza tratarea conexiunilor TCP:

```
 1  import java.net.*;
 2  import java.io.*;
 3  import java.util.Scanner;
 4
 5  public class ConexiuneRetea {
 6     private Socket conexiune;
 7     private Scanner scannerTCP;
 8     private PrintStream printerTCP;
 9     public ConexiuneRetea(Socket conexiune) throws IOException {
10         this.conexiune = conexiune;
11         this.scannerTCP = new Scanner(conexiune.getInputStream());
12         this.printerTCP = new PrintStream(conexiune.getOutputStream());
13     }
14     public String nextLine() {
15         return this.scannerTCP.nextLine();
16     }
17     public int nextInt() {
18         return this.scannerTCP.nextInt();
19     }
20     public void printLine(String text) {
21         this.printerTCP.println(text);
22         this.printerTCP.flush();
23     }
24  }
```

Codul unei clase Client (pentru server cu socket TCP) care utilizeaza `ConexiuneRetea`:

```
 1  import java.net.*;
 2  import java.io.*;
 3  import javax.swing.JOptionPane;
 4
 5  public class Client {
 6     private ConexiuneRetea conexiune;
 7     private Socket socketTCP;
 8     private int portTCP;
 9     private InetAddress adresaIP;
10
11     public Client() throws IOException {
12         portTCP = Integer.parseInt(JOptionPane.showInputDialog(
13                         "Client: introduceti numarul de port al serverului"));
14         adresaIP = InetAddress.getByName(JOptionPane.showInputDialog(
15                         "Client: introduceti adresa serverului"));
16         socketTCP = new Socket(adresaIP, portTCP); // Creare socket
17         conexiune = new ConexiuneRetea(socketTCP);
18     }
19
20     public static void main (String args[]) throws IOException {
21         Client client = new Client();
22         String mesaj;
23
24         while(true) {
25             mesaj = JOptionPane.showInputDialog(
26                          "Client: introduceti mesajul de trimis");
27             client.conexiune.printLine(mesaj);
28             if (mesaj.equals(".")) break;     // Testarea conditiei de oprire
29         }
30
31         client.socketTCP.close();    // Inchiderea socketului si a fluxurilor
32         JOptionPane.showMessageDialog(null, "Client: Bye!");
33     }
34  }
```

Codul unei clase Server (bazat pe socket TCP) care utilizeaza `ConexiuneRetea`:

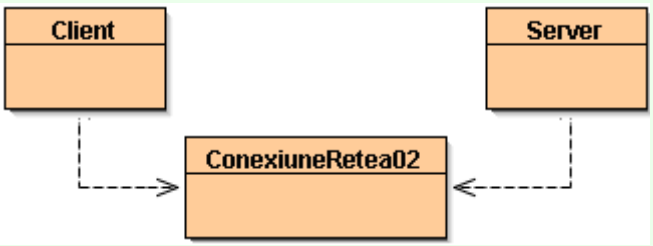
```
1  import java.net.*;
2  import java.io.*;
3  import javax.swing.JOptionPane;
4
5  public class Server {
6     private ConexiuneRetea conexiune;
7     private ServerSocket serverTCP;
8     private Socket socketTCP;
9     private int portTCP;
10
11    public Server() throws IOException {
12        portTCP = Integer.parseInt(JOptionPane.showInputDialog(
13                        "Server: introduceti numarul de port al serverului"));
14        serverTCP = new ServerSocket(portTCP); // Creare socket server
15        socketTCP = serverTCP.accept();        // Creare socket
16        conexiune = new ConexiuneRetea(socketTCP);
17    }
18
19    public static void main (String args[]) throws IOException {
20        Server server = new Server();
21        String mesaj;
22
23        while(true) {
24          mesaj = server.conexiune.nextLine();
25          JOptionPane.showMessageDialog(null, "Server: s-a primit mesajul "+mesaj);
26          if (mesaj.equals(".")) break;     // Testarea conditiei de oprire
27        }
28
29        server.socketTCP.close();    // Inchiderea socketului si a fluxurilor
30        JOptionPane.showMessageDialog(null, "Server: Bye!");
31    }
32 }
```

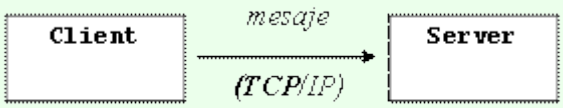
Nu uitati: Daca bara de stare a executiei este activa ( ) verificati cu Alt+Tab daca a aparut o fereastra Java (in spatele ferestrelor vizibile).

Observatie: Cat timp **bara de stare a executiei este activa ( ) codul nu poate fi recompilat, nu poate fi inchisa** fereastra **Terminal Window**, etc. **Pentru a opri executia, folositi *right click* pe si selectati Reset Machine (sau folositi direct Ctrl+Shift+Tab).**

In laborator:

1. **Lansati in executie BlueJ**. **Inchideti proiectele anterioare** (**Ctrl+W**). **Creati un proiect *socket*** (**Project->New Project…**, selectati **D:/**, **POO2007**, **numarul grupei**, si scrieti **socket**).
2. **In proiectul *socket* creati** clasele **ConexiuneRetea**, **Client** si **Server** folosind **codurile de mai sus**.
3. **Compilati codurile** si **creati obiecte** de tip **Server**. **Inspectati obiectele**.
4. **Inspectati campurile obiectelor** (pe cele de tip **ConexiuneRetea si ServerSocket**).





In laborator:

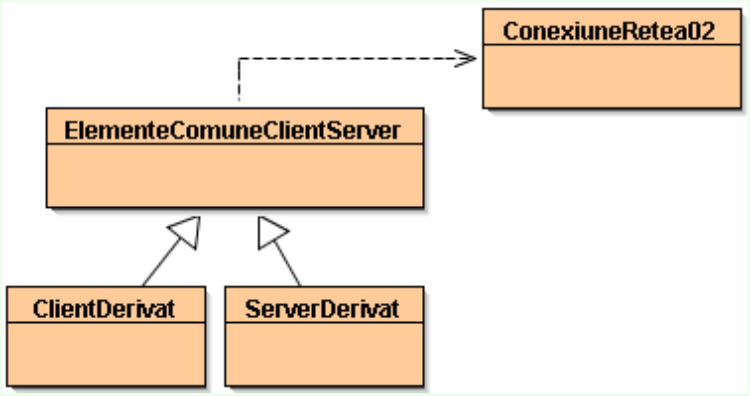
1. **La unul dintre calculatoare *right-click*** pe clasa **Server**.
2. **Selectati** si **executati main()**. **Folositi** numarul de port **3000**.

3. **La un alt calculator** (**daca nu aveti la dispozitie un alt calculator in retea, deschideti inca o sesiune BlueJ**) ***right-click*** pe clasa **Client**, selectati si **executati main()**.
4. **Folositi adresa primului calculator**, pe care se executa **Server** (**adresa `"localhost"` in cazul in care folositi doua sesiuni BlueJ pe acelasi calculator**), si numarul de port **3000**.
5. **Urmariti efectul**.

4.4.2. Clasele ClientDerivat si ServerDerivat (varianta cu mostenire)

Elementele comune claselor Client si Server pot forma o superclasa numita ElementeComuneClientServer:

```
1  import java.net.*;
2  import java.io.*;
3  import javax.swing.JOptionPane;
4
5  public class ElementeComuneClientServer {
6     protected ConexiuneRetea conexiune;
7     protected Socket socketTCP;
8     protected int portTCP;
9
10    public ElementeComuneClientServer(String tip) throws IOException {
11        portTCP = Integer.parseInt(JOptionPane.showInputDialog(tip +
12                          ": introduceti numarul de port al serverului"));
13    }
14 }
```



Codul unei clase ClientDerivat care mosteneste si extinde clasa ElementeComuneClientServer si ofera acelasi serviciu ca si clasa Client:

```
1  import java.net.*;
2  import java.io.*;
3  import javax.swing.JOptionPane;
4
5  public class ClientDerivat extends ElementeComuneClientServer {
6     private InetAddress adresaIP;
7
8     public ClientDerivat() throws IOException {
9         super("Client");
10        adresaIP = InetAddress.getByName(JOptionPane.showInputDialog(
11                       "Client: introduceti adresa serverului"));
12        socketTCP = new Socket(adresaIP, portTCP);           // Creare socket
13        conexiune = new ConexiuneRetea(socketTCP);
14    }
15
16    public static void main (String args[]) throws IOException {
17        ClientDerivat client = new ClientDerivat();
18        String mesaj;
19
20        while(true) {
21            mesaj = JOptionPane.showInputDialog(
22                        "Client: introduceti mesajul de trimis");
23            client.conexiune.printLine(mesaj);
24            if (mesaj.equals(".")) break;    // Testarea conditiei de oprire
25        }
26        client.socketTCP.close();        // Inchiderea socketului si a fluxurilor
27        JOptionPane.showMessageDialog(null, "Client: Bye!");
28    }
29 }
```

Codul unei clase ServerDerivat care mosteneste si extinde clasa ElementeComuneClientServer si ofera acelasi serviciu ca si clasa Server:

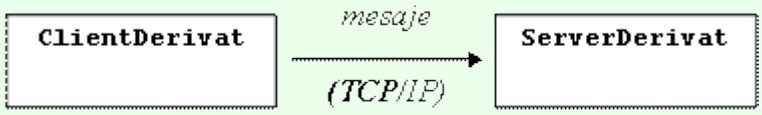
```
1  import java.net.*;
2  import java.io.*;
3  import javax.swing.JOptionPane;
4
5  public class ServerDerivat extends ElementeComuneClientServer {
6     private ServerSocket serverTCP;
7
8     public ServerDerivat() throws IOException {
9         super("Server");
10        serverTCP = new ServerSocket(portTCP);              // Creare socket server
11        socketTCP = serverTCP.accept();                     // Creare socket
12        conexiune = new ConexiuneRetea(socketTCP);
13    }
14
15    public static void main (String args[]) throws IOException {
16        ServerDerivat server = new ServerDerivat();
17        String mesaj;
18
19        while(true) {
20          mesaj = server.conexiune.nextLine();
21          JOptionPane.showMessageDialog(null, "Server: s-a primit mesajul "+mesaj);
22          if (mesaj.equals(".")) break;     // Testarea conditiei de oprire
23        }
24        server.socketTCP.close();    // Inchiderea socketului si a fluxurilor
25        JOptionPane.showMessageDialog(null, "Server: Bye!");
26    }
27 }
```

In laborator:

1. **In proiectul *socket* creati** clasele **ClientDerivat**, **ServerDerivat** si **ElementeComuneClientServer** folosind **codurile date mai sus**.
2. **Compilati codurile.**

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

Nu uitati: Pentru a opri executia, *right click* pe si Reset Machine (sau Ctrl+Shift+Tab).



In laborator:

1. **La unul dintre calculatoare *right-click*** pe clasa **ServerDerivat**.
2. **Selectati** si **executati main()**. **Folositi** numarul de port **4000**.

3. **La un alt calculator calculator (daca nu aveti la dispozitie un alt calculator in retea, deschideti inca o sesiune BlueJ) *right-click*** pe clasa **ClientDerivat**, selectati si **executati main()**.
4. **Folositi adresa primului calculator**, pe care se executa **ServerDerivat** (**adresa `"localhost"` in cazul in care folositi doua sesiuni BlueJ pe acelasi calculator**) si numarul de port **4000**.
5. **Urmariti efectul**.

4.5. Studiu de caz: clasele Persoana, Student, Profesor, DatePersonale, SituatieCurs, si StudentMaster

4.5.1. Clasele Persoana si Profesor si actualizarea clasei Student

Sa presupunem ca dorim sa **adaugam o clasa `Profesor`** care **abstractizeaza un profesor real,** prin intermediul unui camp **`date`** de tip **`DatePersonale`**, si a unui camp **`titlu`** de tip **`String`**.

```
 1  /**
 2    * Incapsuleaza informatiile despre un Profesor. Permite testarea locala.
 3    * @version 1.3
 4    */
 5   public class Profesor {
 6    // Campuri ascunse
 7    private DatePersonale date;
 8    private String titlu;
 9
10    // Constructori
11    public Profesor(String nume, String initiale, String prenume, int anNastere) {
12       date = new DatePersonale(nume, initiale, prenume, anNastere); // copiere „hard"
13    }
14
15    // Interfata publica si implementarea ascunsa (include punct intrare program)
16    public void setTitlu(String t) {
17       titlu = new String(t);      // copiere „hard" a obiectului primit ca parametru
18    }
19    public String toString() {                // forma „String" a campurilor
20       return ("Profesorul " + date + " are titlul " + titlu);
21    }
22    public static void main(String[] args) {
23       // Crearea unui nou Profesor, initializarea campurilor noului obiect
24       Profesor pr = new Profesor("Nulescu", "Ion", "A.", 1960);
25       pr.setTitlu("Lector Dr.");
26       // Utilizarea informatiilor privind Profesorul
27       System.out.println(pr.toString());    // afisarea formei „String" a campurilor
28    }
29   }
```

Se observa ca **avem un camp (`date`) comun cu** clasa **`Student`**.

Putem crea **o clasa `Persoana`** care **sa contina acest element comun**, **rescriind apoi** codurile claselor **`Profesor`** si **`Student`** pentru a **extinde** clasa **`Persoana`** (si a-i refolosi campul **`date`**).

Codul noii clase `Persoana` va fi:

```
 1  /**
 2    * Incapsuleaza informatiile despre o Persoana. Permite testarea locala.
 3    * @version 1.4
 4    */
 5  public class Persoana {
 6    // Campuri ascunse
 7    protected DatePersonale date;
 8
 9    // Constructori
10    public Persoana(String nume, String initiale, String prenume, int anNastere) {
11       date = new DatePersonale(nume, initiale, prenume, anNastere); // copiere „hard"
12    }
13    // Interfata publica si implementarea ascunsa (include punct intrare program)
14    public String toString() {                  // forma „String" a campurilor
15       return (super.toString());
16    }
17    public static void main(String[] args) {
18       // Crearea enei noi Persoane, initializarea campurilor noului obiect
19       Persoana p = new Persoana("Julescu", "Ion", "C.", 1965);
20       // Utilizarea informatiilor privind Persoana
21       System.out.println(p.toString());    // afisarea formei „String" a campurilor
22    }
23  }
```

Vom rescrie acum codul clasei **`Profesor`** pentru a **incorpora schimbarile anuntate**.

```
/**
   * Incapsuleaza informatiile despre un Profesor. Permite testarea locala.
   * @version 1.4
   */
 public class Profesor extends Persoana {
  // Campuri ascunse
  protected String titlu;

  // Constructori
  public Profesor(String nume, String initiale, String prenume, int anNastere) {
     super(nume, initiale, prenume, anNastere); // apel constructor supraclasa
  }                                             // (reutilizare cod)

  // Interfata publica si implementarea ascunsa
  public void setTitlu(String t) {
     titlu = new String(t);       // copiere „hard" a obiectului primit ca parametru
  }

  public String toString() {                // forma „String" a campurilor
     return ("Profesorul " + date + " are titlul " + titlu);
  }
  public static void main(String[] args) {
     // Crearea unui nou Profesor, initializarea campurilor noului obiect
     Profesor pr = new Profesor("Nulescu", "Ion", "A.", 1960);
     pr.setTitlu("Lector Dr.");
     // Utilizarea informatiilor privind Profesorul
     System.out.println(pr.toString());    // afisarea formei „String" a campurilor
  }
 }
```

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

```
/**
   * Incapsuleaza informatiile despre un Student. Permite testarea locala.
   * @version 1.4
   */
 public class Student extends Persoana {
  // Campuri ascunse
  protected SituatieCurs[] cursuri;
  protected int numarCursuri = 0;             // initializare implicita

  // Constructori
  public Student(String nume, String initiale, String prenume, int anNastere) {
     super(nume, initiale, prenume, anNastere); // apel constructor supraclasa
     cursuri = new SituatieCurs[10];        // se initializeaza doar date si cursuri
  }

  // Interfata publica si implementarea ascunsa (include punct intrare program)
  public void addCurs(String nume) {         // se adauga un nou curs
     cursuri[numarCursuri++] = new SituatieCurs(nume);
  }

  public void notare(int numarCurs, int nota) {
     cursuri[numarCurs].notare(nota);       // se adauga nota cursului specificat
  }

  public String toString() {                // forma „String" a campurilor
     String s = "Studentul " + date + " are urmatoarele rezultate:\n";
     for (int i=0; i<numarCursuri; i++)   s = s + cursuri[i].toString() + "\n";
     return (s);
  }

  public static void main(String[] args) {
     // Crearea unui nou Student, initializarea campurilor noului obiect
     Student st1 = new Student("Xulescu", "Ygrec", "Z.", 1987);
     st1.addCurs("CID");
     st1.addCurs("MN");
     st1.notare(0, 8);
     // Utilizarea informatiilor privind Studentul
     System.out.println(st1.toString());    // afisarea formei „String" a campurilor
  }
 }
```

Codul clasei `DatePersonale` ramane nemodificat:

```
 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  }
```

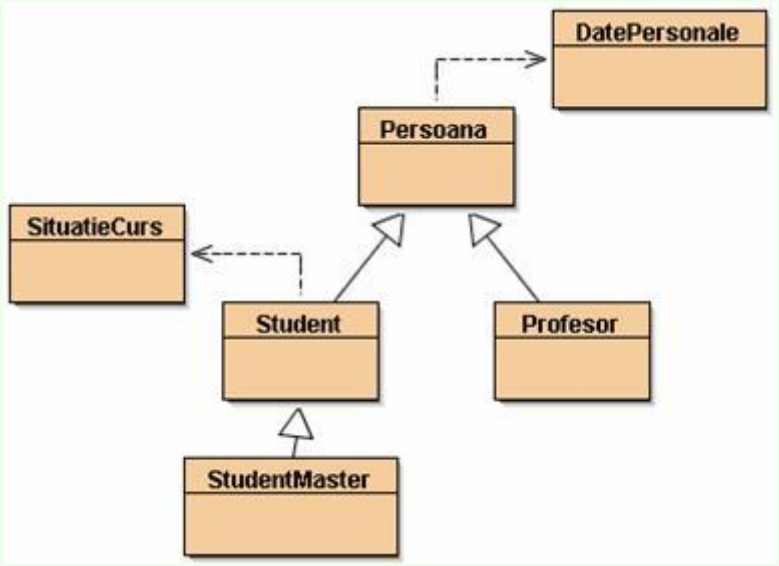
Codul clasei `SituatieCurs` ramane nemodificat:

```
 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  }
```

4.5.2. Clasa StudentMaster

Sa presupunem ca dorim sa **scriem codul unei clase noi** numita `StudentMaster` care **sa abstractizeze un anumit subtip** al tipului `Student`, prin adaugarea la codul clasei `Student` a unui camp numit `specializare` de tip `String`.

Relatiile de utilizare (linie punctata) si mostenire (sageti cu triunghi in capat) in UML:



Putem scrie codul acestei clase **care extinde clasa** `Student` astfel:

```
1  /**
2     * Incapsuleaza informatiile despre un Student. Permite testarea locala.
3     * @version 1.4
4     */
5    public class StudentMaster extends Student {
6     // Campuri ascunse
7     private String specializare;
8
9     // Constructori
10    public StudentMaster(String nume, String initiale, String prenume, int anNastere) {
11       super(nume, initiale, prenume, anNastere); // apel constructor supraclasa
12    }
13
14    // Interfata publica si implementarea ascunsa (include punct intrare program)
15    public void setSpecializare(String spec) {        // se stabileste specializarea
16       specializare = new String(spec);   // copiere „hard" a obiectului primit
17    }
18    public String toString() {                // forma „String" a campurilor
19       String s = "Studentul " + date + " cu specializarea " + specializare +
20                                                  " are urmatoarele rezultate:\n";
21       for (int i=0; i<numarCursuri; i++)   s = s + cursuri[i].toString() + "\n";
22       return (s);
23    }
24    public static void main(String[] args) {
25       // Crearea unui nou Student, initializarea campurilor noului obiect
26       StudentMaster sm = new StudentMaster("Rulescu", "Ygrec", "T.", 1983);
27       sm.addCurs("Securitate Retele");
28       sm.addCurs("Servicii Web");
29       sm.setSpecializare("Retele si Software de Telecomunicatii");
30       sm.notare(0, 9);
31       // Utilizarea informatiilor privind Studentul
32       System.out.println(sm.toString());    // afisarea formei „String" a campurilor
33    }
34   }
```

4.6. 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 4.5 (`Persoana`, `Profesor`, `Student`, si `StudentMaster`) cu urmatoarea specificatie generala:**

(1) Va fi **creata o clasa care are elemente comune cu clasa primita ca tema la lucrarea a 2-a (actualizata la lucrarea a 3-a),** cu numele alocat din tabelul care urmeaza, dupa modelul clasei `Profesor` (`@version 1.3`)

- noua clasa va contine **1-2 campuri protected**, un **constructor public**, **1-2 metode publice** pentru lucrul cu campurile **si o metoda** publica de tip `toString()`,

(2) Va fi **creata o clasa care contine elementele comune ale clasei primite ca tema la lucrarea a 2a (actualizata la lucrarea a 3-a) si a clasei nou create la punctul (1) – adica le generalizeaza,** cu numele alocat din tabelul care urmeaza, dupa modelul clasei `Persoana`:

- noua clasa va contine **campurile comune protected**, un **constructor public**, **si o metoda** publica de tip `toString()`,

(3) Va fi **modificata clasa primita ca tema la lucrarea a 2-a (actualizata la lucrarea a 3-a)** pentru a **extinde clasa nou creata la punctul (2)** si a-i reutiliza campurile si codul constructorului, **campurile ei fiind declarate protected**

(4) Va fi **modificata clasa de la punctul (1)** pentru a **extinde clasa nou creata la punctul (2)** si a-i reutiliza campurile si codul constructorului, **campurile ei fiind declarate protected**, dupa modelul clasei `Profesor` (`@version 1.4`)

(5) Va fi **creata o clasa care extinde** (**specializeaza**) **clasa primita ca tema la lucrarea a 2-a (actualizata la lucrarea a 3-a),** cu numele alocat din tabelul care urmeaza, dupa modelul clasei `StudentMaster`,

- noua clasa va contine **1-2 campuri noi private**, un **constructor public**, **1-2 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 extinsa (ca `Persoana`)	Clasa initiala (ca `Student`) si o pereche a ei (ca `Profesor`)	Clasa care extinde (ca `StudentMaster`)
1	`AparatVideo`	`Monitor + Televizor`	`MonitorTouchscreen + TelevizorPlat`
2	`InstrumentEducational`	`Proiector + TablaInteractiva`	`ProiectorFix + TablaInteractivaMobila`
3	`FotoVideo`	`AparatFoto + CameraVideo`	`AparatFotoMecanic + CameraVideoFixa`
4	`Angajat`	`Sofer + Pilot`	`SoferProfesionist + PilotElicopter`
5	`TerminalPersonal`	`Telefon + PDA`	`TelefonDualSIM + PDATouchscreen`
6	`SursaLumina`	`PointerLaser + Lanterna`	`PixCuPointerLaser + LanternaCuDinam`
7	`MediuStocare`	`StickUSB + HardDiskExtern`	`StickUSBCuCardSD + HardDiskExternAlimentatPrinUSB`
8	`FirmaVanzari`	`Magazin + MagazinOnline`	`MagazinInMall + MagazinOnlineCuShowroom`
9	`Autovehicul`	`Autocar + Microbuz`	`AutocarTurTuristic + MicrobuzHypermarket`
10	`VehiculCuDouaRoti`	`Scuter + Motocicleta`	`ScuterDeInchiriat + MotocicletaCuAtas`
11	`ProdusMedia`	`Blog + Ziar`	`BlogPersonal + ZiarGratuit`
12	`MediuStocare`	`CDROM + CDAudio`	`CDROMDemoProdus + CDAudio`
13	`SistemMesagerie`	`Messenger + Twitter`	`YahooMessenger + TwitterForDisabledUsers`
14	`DezvoltatorSoftware`	`ProjectManager + Proiectant`	`ProjectManagerDebutant + ProiectantFrameworkuri`

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.

III. Adaugarea unei **comunicatii client-server** prin **socket-uri** TCP (a se vedea si **lucrarea 5** **si** **lucrarea 6**) in care **clasele create se afla pe server**, iar **codul care le acceseaza la client.**