1. Introducere

1.1. Obiectul cursului si relatia cu alte cursuri

Programul de calcul (sistemul *software*)
- o **secventa** de **instrucțiuni** (pasi) care conduc **acțiunea** unei **maşini programabile** (de calcul)
- pentru **programator**: o **solutie** la o **problema** pe care o are de rezolvat
- pentru **utilizator**: un **serviciu oferit**

Masina programabila (sistemul *hardware*)
- detalii la Circuite Digitale Integrate **(CID)** si Arhitectura Microprocesoarelor **(AMP)** in sem. II

Sarcinile realizate de programele de calcul le separa in categorii**:**
- programe de **aplicație** diverse **-** se adreseaza direct utilizatorilor (**ele sunt tinta noastra**),
- **-** programe **utilitare** (biblioteci, medii) – pentru programatorii care dezvolta programe de aplicatie,
- programe de **gestiune a resurselor** *hardware* si *software* (sistemul de operare).

Programarea
- activitatea de **concepere** si **scriere** a programelor de calcul.
- **etapa centrala a unui proces larg - dezvoltarea programelor** (analiza, proiectare, ..., testare)

Limbaj de programare	Format	Controlul executiei realizat prin	Folosit in	Cat de abstract e pentru masina	Cat de abstract este pentru om
cod masina	numeric	salturi conditionate, iteratii simple	masina de calcul	deloc	extrem de mult
asamblare	alfa-numeric	salturi conditionate, iteratii simple	programarea la nivel asamblare	destul de mult	foarte mult
procedural (nivel inalt)	alfa-numeric	decizii (simple+multiple), iteratii (mai multe tipuri)	programarea la nivel inalt	mult	mult
orientat spre obiecte (OO)	alfa-numeric	colaborarea obiectelor plus decizii si iteratii	programarea OO	foarte mult	destul de putin

Abstractizarea (are *mai multe definitii corecte*, e un *concept relativ*!!!)
- **neglijarea unor elemente (detalii) si punerea accentului pe altele (esenta)**
- inseamna in primul rand **aproximare** (*problema hartii si teritoriului*, relativitatea aproximarilor)
- **depinde de context** si de **interesul celui care abstractizeaza**!

Limbajele imperative sau procedurale (Pascal, C) au permis introducerea **programarii structurate**,
- detalii la Programarea Calculatoarelor **(PC)** si Structuri de Date si Algoritmi **(SDA)**,
- introduc **structuri de program noi** (inclusiv **definite de programator – functiile**)
- introduc **structuri de date (tipuri) noi** (inclusiv **definite de programator – struct, union**, etc.)
- cer totusi o **gandire in termenii structurii calculatorului**

Limbajele orientate spre obiecte (OO)
- folosesc o **gandire in termenii problemei** – prin **analogia cu entitati din domeniul problemei**
- **entitatile conceptuale din domeniul problemei** devin **obiecte software**
- folosesc **abstractizarea (modelarea) domeniului problemei - solutia e un model** construit din **detaliile esentiale** retinute din domeniul problemei
- introduc **modularizarea programelor** – sub forma **obiectelor**
- **programele devin *societati* de obiecte** care **colaboreaza/comunica - prin mesaje** (**apeluri de functii/metode**)
- programele pot fi **reprezentate prin *scheme bloc*** (diagrame) **– blocuri fiind obiectele/clasele**
- folosesc **tipuri de date abstracte** (ADT - liste, stive, cozi, arbori, grafuri) – **clasele de obiecte**
- introduc **controlul accesului la informatia modularizata** (public/privat/protected) – **incapsularea in obiecte/clase**
- introduc **generalizarea/specializarea claselor** de obiecte – prin **mecanismul mostenirii**
- folosesc **selectia dinamica a comportamentului functiilor** bazata pe pozitia in ierarhia de clase a obiectului curent – **polimorfismul**

1.2. Recapitularea programarii procedurale / structurate

Programul, in sensul clasic (procedural-structurat) se ocupa cu **prelucrari** asupra unor **date**.

```
1   int suma;                          // declaratia (tipului) variabilei - cod Java
2   suma = 0;                          // initializarea variabilei
3   for (int i=1; i<=10; i++) {
4      suma = suma + i;                // utilizarea variabilei (citire+scriere valoare)
5   }
```

Datele sunt reprezentate ca **variabile** (locatii de memorie cu nume). **O variabila are**:

- **numele ei**, care o identifica si este un *alias* pentru adresa numerica (de exemplu, **suma**)
- **valoarea** continuta (de exemplu, **suma** contine pe rand valorile: **0, 1, 3, 6, 10, 15, 21, 28, 36, 45, 55**)
- **locatia** in care e continuta valoarea (in cazul **suma**, locatia ocupa in Java 4B = 32b)
- **adresa** numerica (inaccesibila in anumite limbaje, cum este Java)
- **tipul de date** (de exemplu, **suma** este de tip **int**)

Tipurile de date specifica **structura variabilelor si domeniul de definitie al valorilor**. Mai exact, tipurile de date specifica :

- **spatiul de memorie alocat** pentru stocarea valorii,
- **gama valorilor** posibile,
- **formatul valorilor literale**/de tip imediat (de ex., sufixul **f** pentru valori de tip **float**),
- **conventiile privind conversiile** catre alte tipuri (**direct, implicit, prin extindere** sau **explicit, prin *cast***, **prin trunchiere**),
- **valorile implicite** (daca este cazul),
- **operatorii asociati** (**permisi**) – tin de partea de **prelucrare** asupra datelor.

Tipurile de date primitive Java:

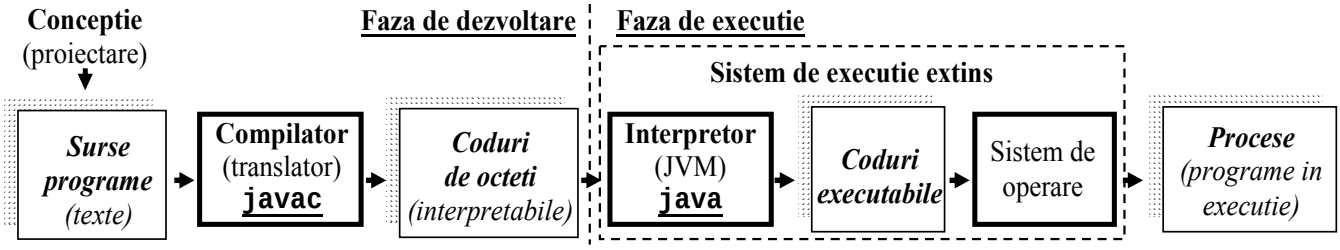
Categorie	Tip	Valoare implicita	Spatiu memorie	Gama valori	Conversii explicite (cast, trunchiere)	Conversii implicite (extindere)
Valori intregi cu semn	**byte**	**0**	**8 biti (1B)**	**-128 … 127**	**Nu sunt necesare**	**La short, int, long, float, double**
	short	**0**	**16 biti (2B)**	**-32768 … 32767**	**La byte**	**La int, long, float, double**
	int	**0**	**32 biti (4B)**	**-2147483648 … 2147483647**	**La byte, short**	**La long, float, double**
	long	**0l**	**64 biti (8B)**	**-9223372036854775808 …9223372036854775807**	**La byte, short, int**	**La float, double**
Valori in virgula mobile cu semn	**float**	**0.0f**	**32 biti (4B)**	**+/-1.4E-45 … +/-3.4028235E+38, +/-infinity, +/-0, NAN**	**La byte, short, int, long**	**La double**
	double	**0.0**	**64 biti (8B)**	**+/-4.9E-324 … +/-1.7976931348623157E+308, +/-infinity, +/-0, NaN**	**La byte, short, int, long, float**	**Nu exista**
Caractere codificate UNICODE	**char**	**\u0000 (null)**	**16 biti (2B)**	**\u0000 … \uFFFF**	**La byte, short**	**La int, long, float, double**
Valori logice	**boolean**	**`false`**	**1 bit folosit din 32 biti**	**true, false**	**Nu exista**	**Nu exista**

Tipurile de date **referinta Java**, ca si pointerii din C, C++, etc., permit definirea unor variabile care contin adrese ale unor locatii. Spre deosebire insa de cazul pointerilor, **in Java nu se poate accesa valoarea adresei continuta** in variabila de tip referinta.

Exemple de conversii intre tipurile primitive:

- intre valori intregi
- intre valori si valori char
- intre valori intregi si valori cu virgula

In **sistemul de programare Java** *codurile sursa* sunt ***compilate*** (translatate) de la limbajul de programare Java la coduri executabile de procesorul software Java (JVM = *Java Virtual Machine*), numite coduri de octeti (*bytecodes*), pentru ca apoi *codurile de octeti* sa fie ***interpretate*** (executate de interpretorul Java, care este parte din JVM, si care ofera coduri executabile sistemului de operare).



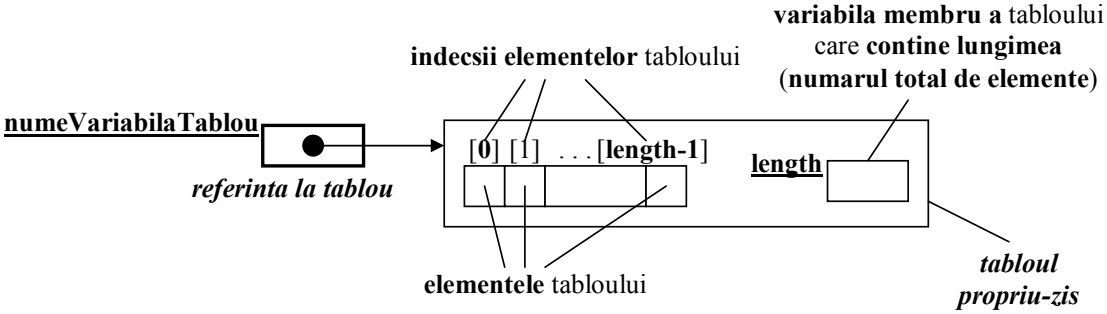
Programul **`SumaArgumenteIntregi.java`** calculeaza **suma valorilor pasate ca argumente la lansarea in executie** (in interpretor).

```
1    public class SumaArgumenteIntregi {
2      public static void main(String[] args)  {
3        System.out.println("Au fost primite " + args.length + " argumente");
4
5        if (args.length > 0) {
6          int suma = 0;
7          for (int index = 0; index < args.length; index++) {
8            suma = suma + Integer.parseInt(args[index]);
9          }
10         System.out.println("Suma valorilor primite este " + suma);
11       }
12       else {
13         System.out.println("Utilizare tipica:");
14         System.out.println("\t java SumaArgumenteIntregi 12 31 133 -10");
15       }
16     }
17   }
```

Pasii necesari sunt compilarea cu compilatorul **`javac`** si **lansarea in executie** in interpretorul **`java`**.

```
directorcurent> javac SumaArgumenteIntregi.java
directorcurent> java SumaArgumenteIntregi 12 31 133 -10
Au fost primite 4 argumente
Suma valorilor primite este 166
directorcurent>
```

Un tablou Java este **o structura** care **contine mai multe valori de acelasi tip**, numite **elemente**. **Lungimea** unui tablou este fixa, **stabilita in momentul crearii tabloului** (cu operatorul **`new`**).



Pentru **a obtine numarul de elemente ale unui tablou** se foloseste:

```
// Obtinerea dimensiunii tabloului de argumente pasate de utilizator
int numarArgumentePasateDeUtilizator = args.length;
```

Pentru **a converti o valoare de la `String` la `int`** se foloseste sintaxa:

```
// Conversia unei valori de la tip int la tip String
int numarStudenti = Integer.parseInt("25");
```

Pentru **a obtine de la utilizator o valoare de tip sir de caractere (`String`)** se foloseste sintaxa:

```
// Obtinerea numelui utilizatorului folosind fereasta de dialog
String nume = JOptionPane.showInputDialog("Introduceti-va numele:");
```

Pentru a se **crea un tablou** cu valorile `1, 2, 3` se foloseste **sintaxa simplificata**:

```
// Crearea unui tablou de 3 valori intregi, varianta simplificata
int[] tab = { 1, 2, 3 };
```

Acelasi efect se obtine folosind **sintaxa complexa pentru crearea unui tablou**:

```
// Crearea unui tablou de 3 valori intregi, varianta complexa
int[] tab = new int[3];    // declararea variabilei si alocarea memoriei
tab[0]= 1;                 // popularea tabloului
tab[1]= 2;                 // popularea tabloului
tab[2]= 3;                 // popularea tabloului
```

Determinarea anilor bisecti (divizibili cu 4)

```
1      // Obtinerea anului testat printr-o fereastra grafica
2      int anulTestat = Integer.parseInt(JOptionPane.showInputDialog("Anul testat"));
3
4      // Verificarea divizarii cu 4 - Varianta folosita in limbajul C
5      // if (anulTestat%4) System.out.println("Anul testat este bisect");
6      // Genereaza eroare:  (incompatible types
7      //                    found   : int
8      //                    required: boolean)
9      // Expresia din paranteza trebuie sa fie tip boolean
10
11     // Verificarea divizarii cu 4 - Varianta corecta in limbajul Java
12     if ((anulTestat%4)==0) System.out.println("Anul testat este bisect");
13
14     // Varianta incorecta in limbajul Java
15     // if ((anulTestat%4)=0) System.out.println("Anul testat este bisect");
16     // Genereaza eroare:  (incompatible types
17     //                    required: variable
18     //                    found   : value)
19     // Trebuie folosit "==" in loc de "="
```

Varianta cu structura de program `if...else`

```
1      // Verificarea divizarii cu 4 - Varianta corecta in limbajul Java
2      if ((anulTestat%4)==0)
3         System.out.println("Anul testat este bisect");
4      else
5         System.out.println("Anul testat nu este bisect");
```

Determinarea anilor bisecti (divizibili cu 4 dar nu cu 100, sau divizibili cu 400)

```
1    // Verificarea divizarii cu 4
2    if ((anulTestat%4)==0)                          // multiplu de 4
3       // Verificarea divizarii cu 100
4       if ((anulTestat%100)==0)                     // multiplu de 100
5          // Verificarea divizarii cu 400
6          if ((anulTestat%400)==0)                  // multiplu de 400
7              System.out.println("Anul testat este bisect");
8          else                                      // nu e multiplu de 400, ci e de 100
9              System.out.println("Anul testat nu este bisect");
10      else                                         // nu e multiplu de 100, ci e de 4
11         System.out.println("Anul testat este bisect");
12   else                                            // nu e multiplu de 4
13      System.out.println("Anul testat nu este bisect");
```

Cum se poate rescrie acest cod folosind mai putine structuri de program de tip `if...else`?

Pentru a calcula *X^i* (in variabila **`xLaI`**) se poate folosi functia matematica **`pow()`** din biblioteca matematica **`Math`** aflata in pachetul de clase **`java.lang`** (pachet care contine si **`String`** si **`System`** si nu necesita importul), metoda **`pow()`** **primeste doua argumente (primul este valoarea de ridicat la putere iar al doilea este puterea)** si care **returneaza o valoare de tip `double`**, de aceea este necesara **conversia explicita (cast) a valorii returnate la tipul `int`**.

```
int xLaI;
for (int i=0; i<=N; i++) {
   // - calculul valorii X^i, unde i=1,N
   xLaI = (int) Math.pow(X, i);
}
```

Lucrul cu tablouri - program histograma

```
1    public class Histograma {
2      public static void main(String[] args) {
3        int max = 3;                      // Valoarea maxima (valoarea minima este 0)
4        int N = 10;                       // Numarul valorilor de intrare (intrarilor)
5
6        int[] intrari = {1, 3, 1, 0, 3, 1, 2, 3, 2, 1};        // Tabloul intrarilor
7        System.out.print("Intrarile: ");                       // Afisarea tabloului
8        for (int i=0; i<N; i++)  System.out.print(intrari[i] + " ");
9
10       int[] histo = new int[max+1];                          // Tabloul histograma
11       for (int i=0; i<N; i++)  histo[intrari[i]]++;          // Populatea tabloului
12
13       System.out.println();                                  // Afisarea tabloului
14       for (int i=0; i<=max; i++) System.out.println(i + " apare de " + histo[i] + " ori");
15     }   // Rezultatul este:
16   }     //    Intrarile: 1 3 1 0 3 1 2 3 2 1
17         //    Valoarea 0 apare de 1 ori
18         //    Valoarea 1 apare de 4 ori
19         //    Valoarea 2 apare de 2 ori
20         //    Valoarea 3 apare de 3 ori
```

Functii - necesitatea existentei:

- tot codul intr-o metoda (se observa redundanta):

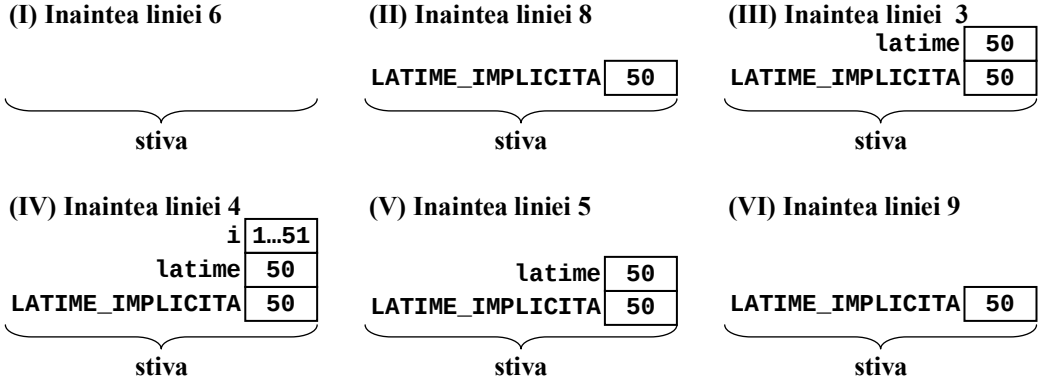
```
1  public class Raport01 {
2     public static void main(String[] args) {
3        final int LATIME = 50;                // variabila finala (constanta!!)
4
5        for (int i = 1; i <= LATIME; i++) System.out.print('-');
6        System.out.println();                 // „traseaza o linie" de 50 de caractere
7        System.out.println("Prima parte a raportului");
8        for (int i = 1; i <= LATIME; i++) System.out.print('-');
9        System.out.println();                 // „traseaza o linie" de 50 de caractere
10       System.out.println("A doua parte a raportului");
11       for (int i = 1; i <= LATIME; i++) System.out.print('-');
12       System.out.println();                 // „traseaza o linie" de 50 de caractere
13    }
14 }
```

- delegarea catre o metoda de tip static (pentru eliminarea redundantelor si modularizarea sarcinilor):

```
1  public class Raport02 {
2     private static void linie() {                    // definitia metodei
3        final int LATIME = 50;
4        for (int i = 1; i <= LATIME; i++) System.out.print('-');
5        System.out.println();                 // „traseaza o linie" de 50 de caractere
6     }
7     public static void main(String[] args) {
8        linie();                                      // apelul metodei
9        System.out.println("Prima parte a raportului");
10       linie();                                      // apelul metodei
11       System.out.println("A doua parte a raportului");
12       linie();                                      // apelul metodei
13    }
14 }
```

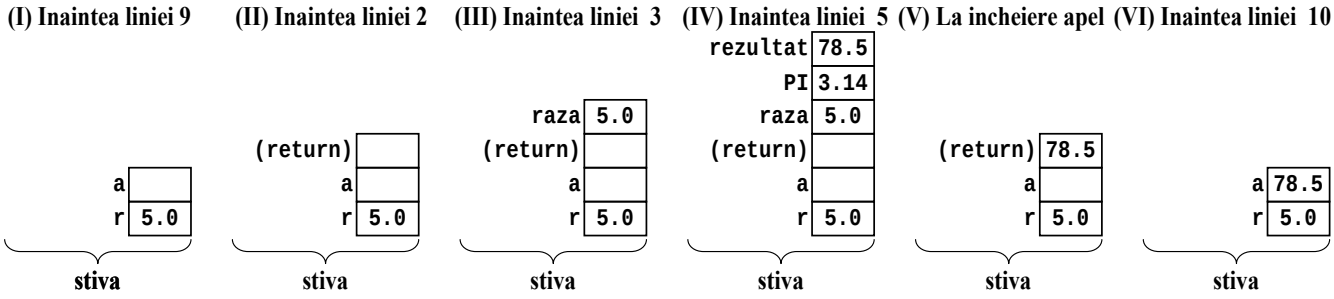
- utilizare parametri / primire argumente (pt. genericitatea codului si flexibilitatea utilizarii):

```
1  public class Raport03 {
2     private static void linie(int latime) {          // definitia metodei
3        for (int i = 1; i <= latime; i++) System.out.print('-');
4        System.out.println();    // „traseaza o linie" de numar variabil de caractere
5     }
6     public static void main(String[] args) {
7        final int LATIME_IMPLICITA = 50;
8        linie(LATIME_IMPLICITA);                      // apelul metodei
9
10       System.out.println("Prima parte a raportului");
11       linie(LATIME_IMPLICITA - 5);                  // apelul metodei
12
13       System.out.println("A doua parte a raportului");
14       linie(LATIME_IMPLICITA);                      // apelul metodei
15    }
16 }
```



Functii - returnarea unor valori:

```
 1  public class Cerc {
 2     private static double arie(double raza) {      // definitia metodei
 3        final double PI = 3.14159;                  // variabila finala (constanta!!)
 4        return 3.14159 * raza * raza;               // returnarea unei valori
 5     }
 6     public static void main(String[] args) {
 7        double r = 5.0;                             // variabila locala r
 8        double a;                                   // variabila locala a
 9        a = arie(r);                                // apelul metodei
10        System.out.println("Un cerc de raza " + r + " are aria " + a + ".");
11     }
12  }
```



Functii - pasarea argumentelor prin valoare (efectul utilizarii unei **copii a valorii primite**)

1. Cazul pasarii unei valori primitive

```
 1   public class C1 {
 2      public static void inc(int i) {  // declaratie (semnatura) metoda inc()
 3         i++;                          // i este parametru formal (pe scurt, parametru)
 4      }
 5      public static void main(String[] args) {
 6         int x = 10;
 7         inc(x);                                // apel metoda inc()
 8         System.out.println("x = " + x);       // x este parametru actual (sau argument)
 9      }                                         // Rezultat: x = 10
10   }
```

2. Cazul pasarii unui tablou

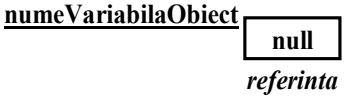
```
 1   public class C2 {
 2      public static void inc(int[] i) {  // primeste o copie a referintei cu aceeasi
 3                                         // valoare, asa incat refera acelasi tablou
 4         i[0]++;                         // este incrementat primul element al tabloului
 5      }
 6      public static void main(String[] args) {
 7         int[] x = {10};                          // tablou cu un element, referit de x
 8         inc(x);                                  // este pasata referinta (valoarea ei)
 9         System.out.println("x[0] = " + x[0]);  // Rezultat: x[0] = 11
10      }
11   }
```

2. Obiecte si clase

2.1. Definitii

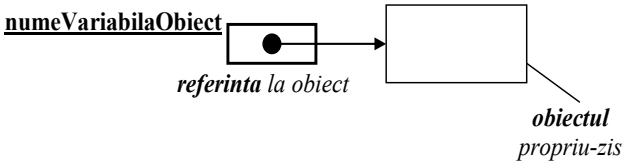
1. Clasa = **tip de date (domeniu de definitie)** al unor **variabile** numite **obiecte**.
= **structura complexa,** reuneste **elemente de date** (campuri, **atribute**) **si algoritmi** (metode**, operatii**)
= **tip referinta Java** (**obiectele** sunt accesate prin **referinta**, care **contine adresa obiectului propriu-zis**)

Declararea variabilelor obiect creaza o simpla referinta la obiect (implicit `null`)




```
NumeClasa  numeVariabilaObiect;
```

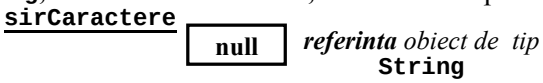
Crearea dinamica a structurii obiectului se face cu operatorul **new**:




```
numeVariabilaObiect =new NumeClasa(listaParametri);
```

Cazul clasei `String` care incapsuleaza siruri de caractere, din pachetul de clase implicite (**`java.lang`**)

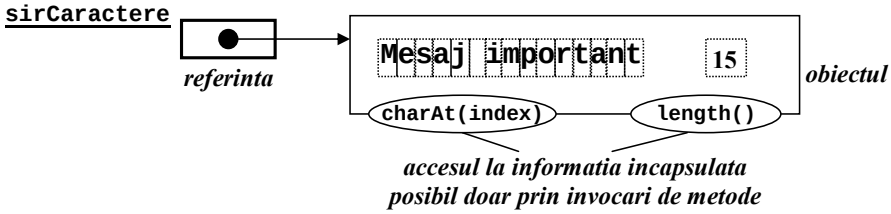
- **crearea unei referinte** la obiect de tip **`String`,** numita **sirCaractere,** initializata implicit cu **null**:




```
String  sirCaractere;
```

- **crearea dinamica a unui obiect tip String (obiectul incapsuleaza sirul de caractere "Mesaj important"):**

```
sirCaractere = new String("Mesaj important"); // alocare si initializare
```



- **accesul la caracterul de index 0** (primul):

```
sirDeCaractere.charAt(0)      // prin metoda charAt()
```

- **accesul la informatia privind numarul de caractere al sirului incapsulat (lungimea sirului)**:

```
sirDeCaractere.length()       // prin metoda length()
```

Pentru comparatie, **cazul unui tablou de caractere** (in Java este diferit de un sir de caractere):

```
char[] tablouCaractere = {'M','e','s','a','j',' ','i','m','p','o','r','t','a','n','t',};
```

- **accesul la caracterul de index 0** (primul):

```
tablouCaractere[0]          // prin index si operator de indexare
```

- **accesul la informatia privind numarul de caractere (lungimea tabloului)**:

```
tablouCaractere.length      // prin camp length
```

2. Obiectul = **reprezentare abstractă** a unor **entități** reale sau virtuale, caracterizată de:
- ***identitate***, prin care e deosebit de alte obiecte, implementata ca **variabila referinta la obiect**,
- ***comportament*** **(vizibil)**, accesibil altor obiecte, implementata ca **set de functii membru** = **operatii, metode**,
- ***stare internă*** **(ascunsă)**, proprie obiectului, implementata ca **set de variabile membru** = **atribute, campuri**.

3. Definitia clasei in Java:

```
1   class Nume {            // declaratie tip de date / structura de date
2      tip atribut;         // declaratie variabila membru, camp Java
3      tipReturnat operatie(tipParametru parametruFormal) {          // semnatura metoda Java
4          // corpul functiei membru (metodei) – returneaza valoare de tipul tipReturnat
5      }
6   }
```

4. Declararea variabilei referinta la un obiect Java si crearea dinamica a obiectului:

```
NumeClasa  numeObiect;                          // declararea variabilei referinta la obiect
numeObiect = new NumeClasa(tipParametru parametruActual);  // crearea dinamica a obiectului
```

- **UML**: **clasa**

Nume
atribut : tip
operatie(parametruFormal : tipParametru) : tipReturnat

obiectul:

numeObiect : NumeClasa

```
1   public class Point {
2      // atribute (variabile membru)
3      private int x;
4      private int y;
5      // operatie care initializeaza atributele = constructor Java
6      public Point(int abscisa, int ordonata) {
7         x = abscisa;
8         y = ordonata;
9      }
10     // operatii care modifica atributele = metode (functii membru)
11     public void moveTo(int abscisaNoua, int ordonataNoua) {
12        x = abscisaNoua;
13        y = ordonataNoua;
14     }
15     public void moveWith(int deplasareAbsc, int deplasareOrd) {
16        x = x + deplasareAbsc;
17        y = y + deplasareOrd;
18     }
19     // operatii prin care se obtin valorile atributelor = metode Java
20     public int getX() {  return x;  }
21     public int getY() {  return y;  }
22  }
```

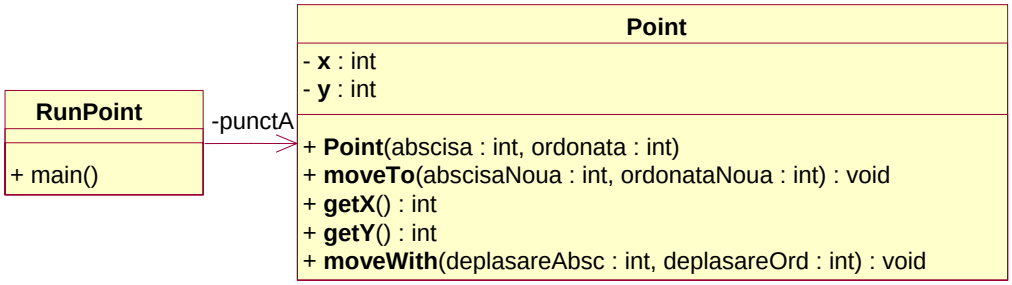
Declaratii (specificare) variabile

Semnaturi (declaratii, specificari) operatii

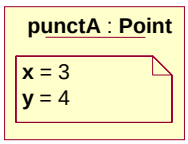
+

Implementari (corpuri) operatii

```
1   public class RunPoint {        // clasa de test pentru clasa Point
2      private static Point punctA;                    // atribut de tip Point
3
4      public static void main(String[] args) {        // declaratie metoda
5                                                      // corp metoda
6        punctA = new Point(3, 4);  // alocare si initializare atribut punctA
7        punctA.moveTo(3, 5);       // trimitere mesaj moveTo() catre punctA
8        punctA.moveWith(3, 5);     // trimitere mesaj moveWith() catre punctA
9      }
10  }
```



5. Crearea unui obiect `punctA` de tip `Point` ale carui atribute au valorile `x=3` si respectiv `y=4` :

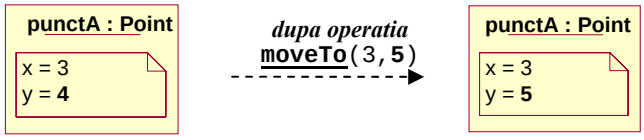



```
Point punctA = new Point(3, 4);
```

Obiectul `punctA` de tip `Point` incapsuleaza informatiile unui punct in plan de coordonate {3, 4}.
Starea obiectului `punctA` este perechea de coordonate **{3, 4}**.

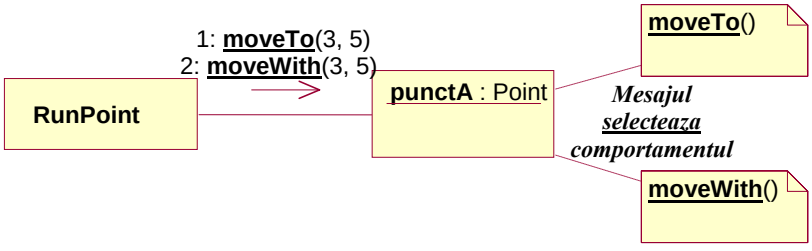
6. Schimbarea starii obiectului `punctA` in {3, 5}, prin deplasarea ordonatei (departarea cu 1 de abscisa).

```
Point punctA = new Point(3, 4);
```

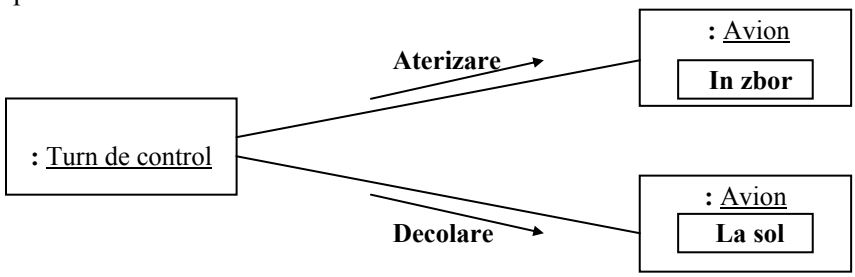



```
punctA.moveTo(3, 5);
```

7. Mesajul selecteaza operatia si declanseaza comportamentul (activeaza operatia, prin invocarea / apelul metodei / functiei membru):

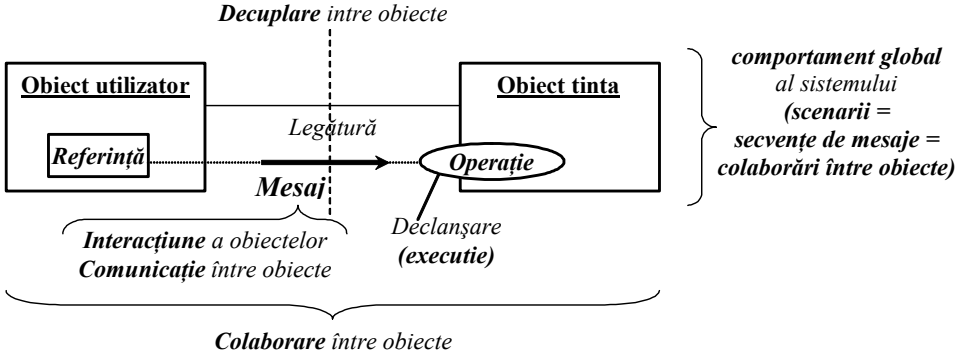


8. Starea si comportamentul sunt **dependente**. Comportamentul la un moment dat depinde de starea curenta. Starea poate fi modificata prin comportament.



In starea **"In zbor" doar comportamentul Aterizare e posibil**. El **duce la schimbarea starii** (in "La sol"). **Dupa aterizare, starea fiind "La sol", operatia Aterizare nu mai are sens**.

9. Sistemele software OO sunt **societăți de obiecte care colaborează** pentru a realiza funcțiile aplicației.



2.2. Particularitati Java

Reluarea problemei pasarii argumentelor prin valoare: **1. Problema pasarii unei valori primitive**

2. Solutia pasarii unui tablou

3. Solutia pasarii unui obiect care contine un camp public (accesibil de catre orice cod exterior) – caz in care se poate vorbi de **"lucrul cu" obiecte!**

```
 1   public class C3 {
 2      public static void inc(ClasaInt i) {  // primeste o copie a referintei cu aceeasi
 3                                            // valoare, asa incat refera acelasi obiect
 4         i.camp++;                          // e incrementat campul continut in obiect
 5      }
 6      public static void main(String[] args) {
 7         ClasaInt x = new ClasaInt();   // obiect referit de x, continand camp tip int
 8         x.camp = 10;                   // initializat cu valoarea 10
 9         inc(x);                        // este pasata referinta (valoarea ei)
10         System.out.println("x.camp = " + x.camp);  // Rezultat: x.camp = 11
11      }
12   }
13   class ClasaInt {
14      public int camp;
15   }
```

4. Solutia pasarii unui obiect care contine un camp privat (inaccesibil oricarui cod exterior) **si metode de acces** – caz in care se poate vorbi de **"orientare spre" obiecte!**

```
 1   public class C4 {
 2      public static void inc(ClasaInt i) {  // primeste o copie a referintei cu aceeasi
 3                                            // valoare, asa incat refera acelasi obiect
 4         i.setCamp(i.getCamp()+1);          // e incrementat campul incapsulat in obiect
 5      }
 6      public static void main(String[] args) {
 7         ClasaInt x = new ClasaInt();   // obiect referit de x, continand camp tip int
 8         x.setCamp(10);                 // initializat cu valoarea 10
 9         inc(x);                        // este pasata referinta (valoarea ei)
10         System.out.println("x.getCamp() = " + x.getCamp());  // Rez.: x.getCamp() = 11
11      }
12   }
13   class ClasaInt {
14      private int camp;
15      public void setCamp(int c) { camp=c; }
16      public int getCamp() { return camp; }
17   }
```

Structura unei clase Java:

```
1   import java.util.Vector;                  // clase importate
2   import java.util.EmptyStackException;
3
4   public class Stack              // declaratia clasei
5   {                               // inceputul corpului clasei
6
7     private Vector elemente;                // atribut (variabila membru)
8
9     public Stack() {                        // constructor
10      elemente = new Vector(10);            // (functie de initializare)
11    }
12
13    public Object push(Object element) {    // metoda
14      elemente.addElement(element);         // (functie membru)
15      return element;
16    }
17
18    public synchronized Object pop(){       // metoda
19      int lungime = elemente.size();        // (functie membru)
20      Object element = null;
21      if (lungime == 0)
22        throw new EmptyStackException();
23      element = elemente.elementAt(lungime - 1);
24      elemente.removeElementAt(lungime - 1);
25      return element;
26    }
27
28    public boolean isEmpty(){               // metoda
29      if (elemente.size() == 0)             // (functie membru)
30        return true;
31      else
32        return false;
33    }
34  }                               // sfarsitul corpului clasei
35
```

Declaratia de clasa

```
[public] [abstract] [final] class NumeClasa [extends NumeSuperclasa]
                                        [implements  NumeInterfata [, NumeInterfata]] {
  // Corp clasa
}
```

Declaratia de camp (atribut)

```
[nivelAcces] [static] [final] tipAtribut numeAtribut;
```

Declaratia de constructor (functie initializare obiect)

```
[nivelAcces] NumeClasa( listaParametri ) {
  // Corp constructor
}
```

Declaratia de metoda (operatie)

```
[nivelAcces] [static] [abstract] [final] [synchronized] tipReturnat numeMetoda (
                          [listaDeParametri] ) [throws NumeExceptie [,NumeExceptie] ] {
  // Corp metoda
}
```

Tipurile referinta Java sunt tipul **tablou**, tipul **clasa** si tipul **interfata**.

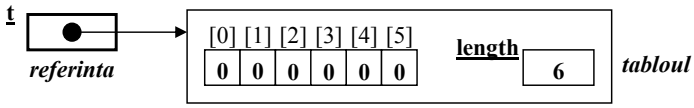
Variabilele de tip referinta sunt variabile **tablou**, al caror **tip** este **un tablou**, si variabile **obiect**, al caror **tip** este **o clasa sau o interfata**.

Variabilele de tip referinta **contin valoarea referintei** catre tablou/obiect (**creata in momentul declararii**) si sunt plasate in **stack**, pe cand **tabloul/obiectul** propriu-zis este **creat in mod dinamic** (**cu new**) in **heap**.

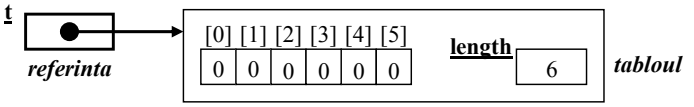
```
1    int[] t;                  // declarare simpla
2    t = new int[6];           // alocare si initializare
3    int[] v;                  // declarare simpla
4    v = t;                    // copiere referinte
5    int[] u = { 1, 2, 3, 4 }; // declarare, alocare si initializare
6    t[1] = u[0];              // atribuire intre elemente
7    v = u;                    // copiere referinte
```

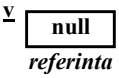
```
8    t = v;
9    int x = t[1];
10   u[1] = 5;
11   int y = t[1];
12   y = x;
13   int[] w;
14   w[0] = x;
```

- dupa linia 2:

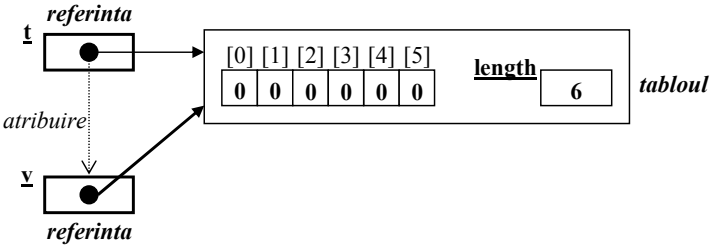


- dupa linia 3:

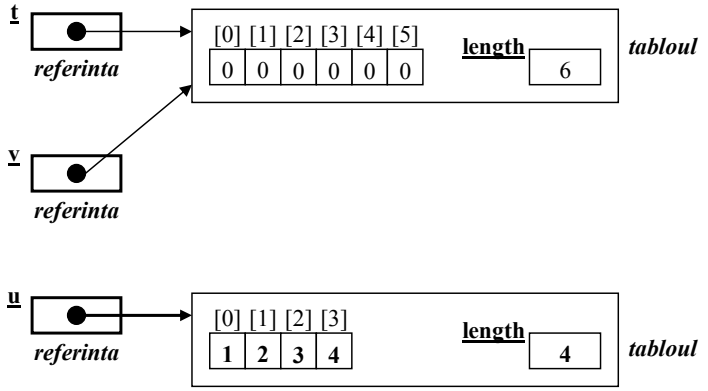




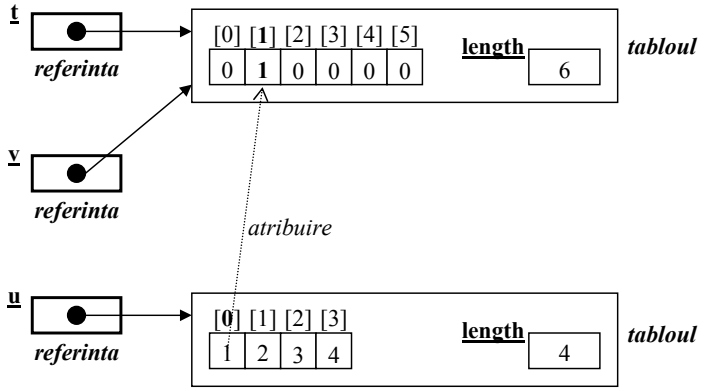
- dupa linia 4:



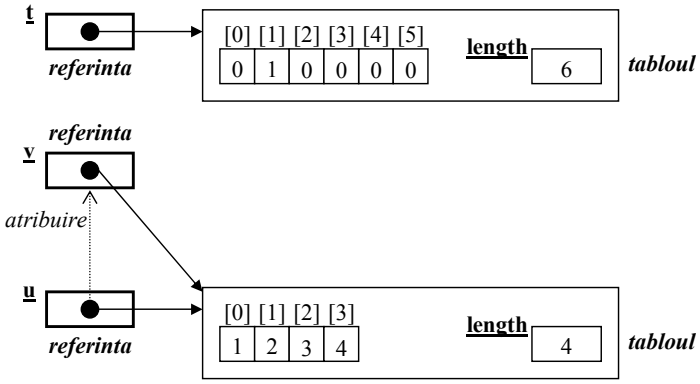
- dupa linia 5:



- dupa linia 6:



- dupa linia 7:



Clasa String - program de **cautare a unor cuvinte cheie bazata pe parsing** (analiza lexicala)

```
 1  public class CautareCuvinteCheie1 {
 2     public static void main(String[] args) {
 3
 4        String textAnalizat = "The string tokenizer class allows application " +
 5          "to break a string into tokens. The tokenization method is much simpler " +
 6          "than the one used by the StreamTokenizer class.";
 7
 8        String[] cuvinteCheie = { "string" , "token" };
 9
10        // Pentru toate cuvintele cheie cautate
11        for (int i=0; i<cuvinteCheie.length; i++) {
12           String text = textAnalizat;
13           int pozitie=0;
14
15           // Daca un anumit cuvant cheie este gasit intr-un anumit text
16           // Varianta cu String.indexOf()
17           while ( text.indexOf(cuvinteCheie[i]) > -1 ) {
18              pozitie = pozitie + text.indexOf(cuvinteCheie[i])+1;
19
20              // Informeaza utilizatorul (indicand si pozitia)
21              System.out.println("Cuvantul cheie \"" + cuvinteCheie[i] +
22                 "\" a fost gasit in text pe pozitia " + pozitie + "\n");
23              text = text.substring(text.indexOf(cuvinteCheie[i])+1);
24           }
25        }
26     }
27  }
```

Clasa String - program de **analiza lexicala**

```
 1   public class StatisticiText {
 2     public static void main(String[] args) {
 3       String textAnalizat = "The string tokenizer class allows application " +
 4         "to break a string into tokens. The tokenization method is much simpler " +
 5         "than the one used by the StreamTokenizer class.";
 6
 7       String[] cuvinte = textAnalizat.split(" ");  // separatorul este un spatiu (" ")
 8
 9       // Numarul de cuvinte
10       System.out.println("  Textul contine " + cuvinte.length + " cuvinte:");
11
12       // Cuvintele
13       for (int i=0; i<cuvinte.length; i++)    System.out.println(cuvinte[i]);
14     }
15   }
```

Echivalente functionale:

```
1   char[] caractere = {'t', 'e', 's', 't'};
2   String sir = new String(caractere);
3   // echivalent cu    String sir = String.valueOf(caractere);
```

```
1   char[] caractere = {'t', 'e', 's', 't', 'a', 'r', 'e'};
2   String sir = new String(caractere, 2, 5);
3   // echivalent cu    String sir = String.valueOf(caractere, 2, 5);
```

```
1   String original = "sir";
2   String copie = new String(original);
3   // echivalent cu    String copie = original.tostring();
4   // echivalent cu    String copie = String.valueOf(original);
```

Complementaritati functionale (conversii):

```
String sir = "test";
byte[] octeti = sir.getBytes();            // obtinere tablou octeti din String
String copieSir = new String(octeti);      // obtinere String din tablou octeti
```

Clasa **`StringBuffer`** (**sir de caractere modificabil**) **- codul**:

```
x = "a" + 4 + "c";
```

este compilat ca:

```
x = new StringBuffer().append("a").append(4).append("c").toString();
```

Utilizarea metodei `insert()`:

```
StringBuffer sb = new StringBuffer("Drink Java!");
sb.insert(6, "Hot ");
System.out.println(sb.toString());
```

Rezultatul executiei programului:

```
Drink Hot Java!
```

Exemple de lucru cu obiecte de tip **`Integer`**

```
1    int     i, j, k;      // intregi ca variabile de tip primitiv
2    Integer m, n, o;      // intregi incapsulati in obiecte Integer
3    String  s, r, t;      // siruri de caractere (incapsulate in obiecte)
4
5    // constructia intregilor incapsulati utilizand constructori ai clasei
6    i = 1000;
7    m = new Integer(i);        // echivalent cu   m = new Integer(1000);
8    r = new String("30");
9    n = new Integer(r);        // echivalent cu   n = new Integer("30");
10
11   // constructia intregilor incapsulati utilizand metode de clasa ale
12   t = "40";
13   o = Integer.valueOf(t);    // echivalent cu   o = new Integer("40");
14
15   // conversia intregilor incapsulati la valori numerice primitive
16   byte  iByte = m.byteValue();         // diferit de 1000! (trunchiat)
17   int   iInt = m.intValue();           // = 1000
18   float iFloat = m.floatValue();       // = 1000.0F
19   double iDouble = m.doubleValue();    // = 1000.0
20
21   // conversia valorilor intregi primitive la siruri de caractere
22   String douaSute = Integer.toString(200);  // metoda de clasa (statica)
23   String oMieBinary = Integer.toBinaryString(1000); // metoda de clasa
24   String oMieHex = Integer.toHexString(1000);       // metoda de clasa
25
26   // conversia sirurilor de caractere la valori intregi primitive
27   int oSuta = Integer.parseInt("100");       // metoda de clasa (statica)
```

Tratarea exceptiilor **In cazul in care argumentul nu are format intreg** apelul metodei **`parseInt()`** **genereaza o exceptie de tip** **`NumberFormatException`** (definita in pachetul **`java.lang`**), care **trebuie tratata exceptia** cu un bloc:

```
try {
   // aici este plasata secventa de cod care poate genera exceptia
}
catch (NumberFormatException ex) {
   // aici este plasata secventa de cod care trateaza exceptia
}
```

```
1    public class VerificareArgumenteIntregi {
2      public static void main(String[] args) {
3        int i;
4        for ( i=0; i < args.length; i++ ) {
5          try {
6            System.out.println(Integer.parseInt(args[i]));
7          }
8          catch (NumberFormatException ex) {
9            System.out.println("Argumentul " +args[i]+ " nu are format numeric intreg");
10         }
11       }
12     }
13   }
```

```
1    public class ClasificareArgumenteConsola {
2      // stabilirea la lansare a valorilor, ca argumente ale programelor
3      public static void main(String[] args) {
4        int i;
5        for ( i=0; i < args.length; i++ ) {
6          try {
7            int intreg = Integer.parseInt(args[i]);
8            System.out.println("Argumentul " +intreg+ " are format numeric intreg");
9          }
10         catch (NumberFormatException ex1) {
11           try {
12             double real = Double.parseDouble(args[i]);
13             System.out.println("Argumentul " +real+ " are format numeric real");
14           }
15           catch (NumberFormatException ex2) {
16             System.out.println("Argumentul " +args[i]+ " nu are format numeric");
17           }
18         }
19       }
20     }
21   }
```

Scopul variabilelor (vizibilitatea lor in interiorul clasei):
- reprezintă **portiunea de cod al clasei în care variabila este accesibilă** si
- determină **momentul în care variabila este creată şi distrusă**.

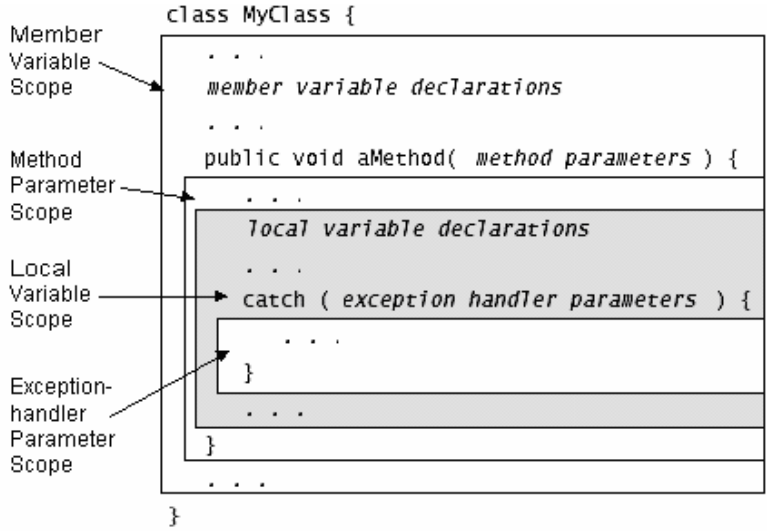
Exista 4 categorii de scop al variabilelor Java:

- camp sau **atribut** sau **variabilă membru** (*member variable*),
- este membrul unei clase sau al unui obiect,
- poate fi declarată oriunde în clasă, dar nu într-o metodă,
- e **disponibilă în tot codul clasei**.

- variabilă locală (*local variable*),
- poate fi declarată oriunde într-o metodă sau într-un bloc de cod al unei metode,
- e **disponibilă în codul metodei, din locul de declarare şi până la sfârşitul blocului de in care e declarata**

- parametru al unei metode (*method parameter*),
- este argumentul formal al metodei,
- este utilizat pentru a se pasa valori metodei,
- e **disponibil în întreg codul metodei**.

- parametru al unei proceduri de control al exceptiilor (e*xception-handler parameter*),
- este argumentul formal al *handler-ului* de excepție,
- este utilizat pentru a se pasa valori *handler-ului* de excepție,
- e **disponibil în întreg codul *handler-ului* de excepție**.

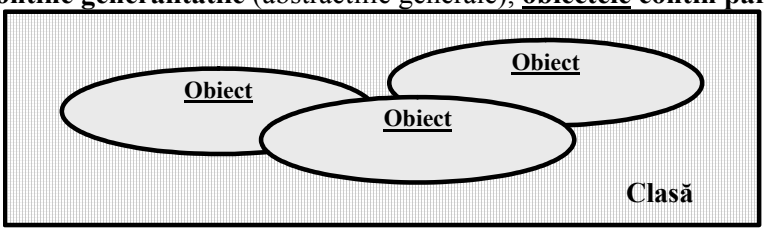


Exemplu

```
 1  public class Complex        // declaratia clasei
 2  {                           // inceputul corpului clasei
 3    private double real;            // real = atribut (camp)
 4    private double imag;            // imag = atribut (camp)
 5
 6    public void setReal(double real) {    // metoda, real = parametru
 7      this.real = real;                   // real = atributul, real = parametrul
 8    }
 9
10    public void.setImag(double imag) {    // metoda, imag = parametru
11      this.imag = imag;                   // imag = atributul, imag = parametrul
12    }
13
14    public static void main(String[] args) {      // metoda, args = parametru
15      double real = Double.parseDouble(args[0]);  // real = variabila locala
16      double imag = Double.parseDouble(args[1]);  // imag = variabila locala
17
18      Complex c = new Complex();                  // c = variabila locala
19
20      c.setReal(real);                            // c, real = variabilele locale
21      c.setImag(imag);                            // c, imag = variabilele locale
22
23      System.out.println("{" + c.real +           // c.real = atributul lui c
24                         ", " + c.imag + "}");    // c.imag = atributul lui c
25    }
26  }                           // sfarsitul corpului clasei
```

2.3. Clase si relatii intre clase

Clasa contine generalitatile (abstractiile generale), **obiectele contin particularitatile** (detaliile particulare).



O parte din codul clasei `DatagramPacket` (continuturile anumitor metode au fost simplificate):

```
 1    package java.net; // Importul pachetului de clase Java pentru comunicatii IP
 2    public final class DatagramPacket {  // incapsuleaza pachete UDP (datagrame)
 3      // Atribute, accesibile tuturor claselor din pachetul (directorul) java.net
 4      byte[] buf;           // tabloul de octeti care contine datele pachetului
 5      InetAddress address;  // adresa IP a masinii sursa/destinatie a pachetului
 6      int port;             // portul UDP al masinii sursa/destinatie a pachetului
 7
 8      // Constructori – initializeaza obiectele de tip DatagramPacket
 9      // Initializeaza un DatagramPacket pentru pachete de receptionat,
10      public DatagramPacket(byte buf[], int length) {
11         this.buf = buf;
12         this.address = null;
13         this.port = -1;
14      }
15      // Initializeaza un DatagramPacket pentru pachete de trimis
16      public DatagramPacket(byte buf[], int length, InetAddress address, int port) {
17         this.buf = buf;
18         this.address = address;
19         this.port = port;
20      }                          // Alti constructori, alte metode...
21      // Returneaza adresa IP a masinii sursa/destinatie a acestui pachet
22      public synchronized InetAddress getAddress() {  return this.address;  }
23
24      // Stabileste adresa IP a masinii sursa/destinatie a acestui pachet
25      public void setAddress(InetAddress iaddr) {  this.address = iaddr;   }
26   }
```

Cod care utilizeaza clasa `DatagramPacket`:

```
byte[] buffer = new byte[1024];                  // buffer date (de trimis/primit)

DatagramPacket packetDeTrimis = new DatagramPacket(buffer, buffer.length,
                          InetAddress.getByName("nume.elcom.pub.ro"), 2000);

DatagramPacket packetDeReceptionat = new DatagramPacket(buffer, buffer.length);
```

Specificatia generala a unei clase `Complex` (interfata publica) = **API-ul** (*Application Programming Interface*):

```
1    public class Complex {
2      // Atribute private (ascunse, inaccesibile din exteriorul clasei)
3      // Constructor – initializeaza obiectele de tip Complex
4      public Complex(double real, double imag) {  // Implementare  }
5      public double getReal() {  // Implementare  }    // Returneaza partea reala
6      public double getImag() {  // Implementare  }    // Returneaza partea imaginara
7      public double getModul() {  // Implementare  }    // Returneaza modulul
8      public double getFaza() {  // Implementare  }    // Returneaza faza
9    }
```

Implementarea carteziana a clasei `Complex` (atributele ascunse sunt coordonatele carteziene):

```
 1    public class Complex {
 2      private double real;           // partea reala (abscisa)
 3      private double imag;           // partea imaginara (ordonata)
 4      public Complex(double real, double imag) {
 5         this.real = real;
 6         this.imag = imag;
 7      }
 8      public double getReal() {       return this.real;    }
 9      public double getImag() {       return this.imag;    }
10      public double getModul() {
11         return Math.sqrt(this.real*this.real + this.imag*this.imag);
12      }
13      public double getFaza() {
14         return Math.atan2(this.real, this.imag);
15      }
16    }
```

Implementarea polara a clasei `Complex` (**atributele ascunse sunt coordonatele polare**):

```
1   public class Complex {
2     private double modul;              // modulul (raza)
3     private double faza;               // faza (unghiul)
4     public Complex(double real, double imag) {
5        this.modul = Math.sqrt(real*real + imag*imag);
6        this.faza = Math.atan2(real, imag);
7     }
8     public double getReal() {    return this.modul*Math.cos(this.faza);    }
9     public double getImag() {    return this.modul*Math.sin(this.faza);    }
10    public double getModul() {       return this.modul;    }
11    public double getFaza() {        return this.faza;    }
12  }
```

Urmatorul cod Java va conduce la acelasi rezultat, indiferent de implementarea clasei `Complex`:

```
Complex c1 = new Complex(2, -2);
System.out.println("Coordonatele carteziene: {" + c1.getReal() + ", " + c1.getImag() + "}");
System.out.println("Coordonatele polare: {" + c1.getModul() + ", " + c1.getFaza() + "}");
```

In cazul de mai sus:
- **specificația nu este afectată de schimbarea reprezentării interne** (polară sau carteziană),
- **obiectele utilizator cunosc doar specificația și nu sunt nici ele afectate de schimbarea reprezentării interne**.

Clasa **`Student`** **abstractizeaza un student real** (**incapsuland informatiile si comportamentul lui**)

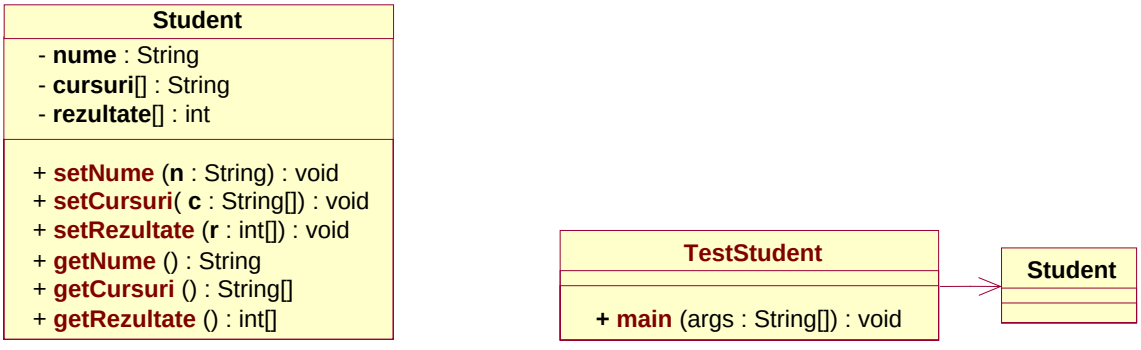
```
1   /**
2    * Incapsuleaza informatiile si comportamentul unui Student.
3    * @version 1.1
4    */
5   public class Student {
6     // Campuri (atribute) private (inaccesibile codurilor exterioare)
7     private String nume;
8     private String[] cursuri;
9     private int[] rezultate;
10
11    // Metode (operatii) publice (accesibile tuturor codurilor exterioare)
12    // Metoda stabilire nume
13    public void setNume(String n)        {  nume = n;  }
14
15    // Metoda stabilire cursuri
16    public void setCursuri(String[] c)   {  cursuri = c;  }
17
18    // Metoda stabilire rezultate
19    public void setRezultate(int[] r)    {  rezultate = r;  }
20
21    // Metoda obtinere nume
22    public String getNume()              {  return (nume);  }
23
24    // Metoda obtinere cursuri
25    public String[] getCursuri()         {  return (cursuri);  }
26
27    // Metoda obtinere rezultate
28    public int[] getRezultate()          {  return (rezultate);  }
29  }
```

Informatii ascunse = stare ascunsa (campuri private)

Interfata publica = servicii oferite (semnaturi metode)

Implementare ascunsa = comportament ascuns (coduri interne ale metodelor)

Reprezentarea UML a clasei, atributelor (nivel acces **private**, notat "-") si **metodelor** (nivel acces **public**, notat "+"):



Incapsulare = **gruparea mai multor elemente asociate** de date (atribute) si/sau de comportament (operatii)

Incapsularea OO inseamna in plus **ascunderea detaliilor interne** de tip **informatii** / **stare** (setul de **campuri** / **atribute**), si **implementare** / **comportament** (setul de **coduri interne**), **in spatele** unei **interfete publice** (setul de **declaratii / semnaturi ale metodelor / operatiilor**).

```
/**
 * Testeaza clasa Student.
 */
public class TestStudent {
  // Metoda de test. Punct de intrare in program.
  public static void main(String[] args) {
     // Crearea unui nou Student, fara informatii
     Student st1 = new Student();
     // Initializarea campurilor noului obiect
     st1.setNume("Xulescu Ygrec");
     String[] crs = {"CID", "AMP"};
     st1.setCursuri(crs);
     int[] rez = {8, 9};
     st1.setRezultate(rez);
     // Utilizarea informatiilor privind Studentul
     System.out.println("Studentul " + st1.getNume() + ":");
     for (int i=0; i<rez.length; i++)
         System.out.println("- are nota " + st1.getRezultate()[i]
                     + " la disciplina " + st1.getCursuri()[i]);
  }
}   // Rezultatul:  Studentul Xulescu Ygrec:
    //              - are nota 8 la disciplina CID
    //              - are nota 9 la disciplina AMP
```

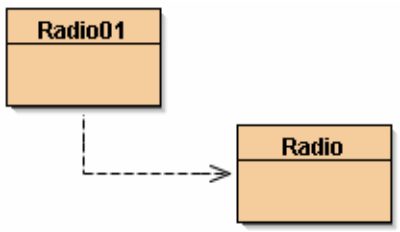
Clasa numita Radio (simuleaza planurile pentru crearea unui obiect radio)

```
public class Radio {
   protected double[] stationNumber = new double[5];      // camp (atribut,
                                                          // variabila membru)
   public void setStationNumber(int index, double freq){  // metoda (operatie,
      stationNumber[index] = freq;                        // functie membru)
   }
   public void playStation(int index){
      System.out.println("Playing the station at " + stationNumber[index] + " Mhz");
   }
}
```

Clasa numita Radio01 (creeaza si foloseste un obiect al clasei Radio – testeaza clasa Radio)

```
public class Radio01{
   public static void main(String[] args){
      Radio myObjRef = new Radio();

      myObjRef.setStationNumber(3, 93.5);
      myObjRef.playStation(3);
   }
} // Rezultat: Playing the station at 93.5 MHz
```



Variante fara orientare spre obiecte: - in clasa **Radio001** tot codul este scris in metoda principala, **main()**:

```
public class Radio001 {                      // varianta strict functionala (fara obiecte)
  public static void main(String[] args) {  // tot codul in metoda principala
     double[] stationNumber = new double[5];         // variabile locale
     int index = 3;
     double freq = 93.5;
     stationNumber[index] = freq;
     System.out.println("Playing the station at " + stationNumber[index] + " Mhz");
  }
}
```

- in clasa **Radio002** metoda principala **main()** deleaga 2 sarcini catre 2 metode (**delegare/modularizare functionala**):

```
public class Radio002 {                      // varianta strict functionala (fara obiecte)
   public static void main(String[] args) {  // delegare / modularizare functionala
      double[] stationNumber = new double[5];        // variabile locale
      int index = 3;
      double freq = 93.5;                            // „orientare spre functii”
      setStationNumber(stationNumber, index, freq);  // delegare sarcina
      playStation(stationNumber, index);             // delegare sarcina
   }
   public static void setStationNumber(double[]stationNumber, int index, double freq){
      stationNumber[index] = freq;                                  // efectuare sarcina
   }
   public static void playStation(double[] stationNumber, int index){ // efectuare sarcina
      System.out.println("Playing the station at " + stationNumber[index] + " Mhz");
   }
}
```

- in clasa **Radio003** apare un atribut declarat static, global la nivelul clasei (**modularizare la nivel de clasa**):

```
1   public class Radio003 {                        // varianta cu camp (atribut) global (static)
2     private static double[] stationNumber =new double[5]; // camp static (la nivel de clasa)
3                                                            //
4     private static void setStationNumber(int index, double freq){
5        stationNumber[index] = freq;                        // utilizare camp global (static)
6     }
7     private static void playStation(int index){          // utilizare camp global (static)
8        System.out.println("Playing the station at " + stationNumber[index] + " Mhz");
9     }
10    public static void main(String[] args) {
11       int index = 3;
12       double freq = 93.5;
13       setStationNumber(index, freq);                    // delegare sarcina
14       playStation(index);                               // delegare sarcina
15    }
   }
```

In clasa Polinom2 de la laborator, **`gradPolinom` si `coeficienti` sunt variabile locale**. Un polinom este insa **definit structural** tocmai **de `gradPolinom` si `coeficienti`.**

Este recomandabila declararea lor ca atribute (campuri) alaturi de metodele pentru obtinere date, afisare, calcul.

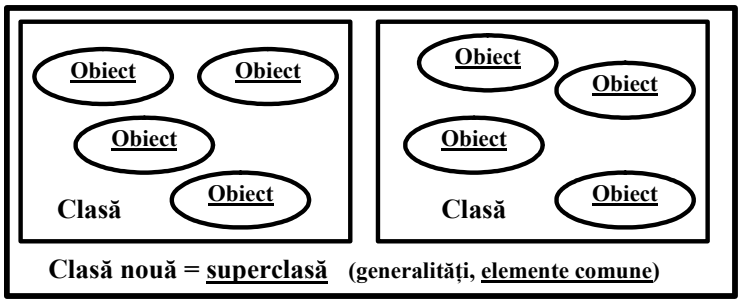
```
1   import javax.swing.JOptionPane;
2   public class Polinom4 {
3
4     // Campuri (atribute, variabile membru)
5     int gradPolinom;
6     int[] coeficienti;
7
8     // Metoda care obtine de la utilizator gradul polinomului
9     public void obtineGrad() {      // Obtinerea de la utilizator a gradului polinomului
10      gradPolinom = Integer.parseInt(JOptionPane.showInputDialog("Introduceti gradul"));
11    }
12    // Metoda care obtine de la utilizator coeficientii polinomului
13    public void stabilesteCoeficienti() {
14      coeficienti = new int[gradPolinom+1];
15      // Obtinerea de la utilizator a coeficientilor Ci, unde i=0,N
16      for (int i=0; i<=gradPolinom; i++)
17         coeficienti[i] = Integer.parseInt(JOptionPane.showInputDialog("Coeficientul " + i));
18    }
19    // Metoda care afiseaza polinomul P(X)
20    public void afisarePolinom() {
21      System.out.print("P(X) = " + coeficienti[0]);
22      for (int i=1; i<=gradPolinom; i++)
23         System.out.print(" + " + coeficienti[i]  + "*X^" + i);
24      System.out.println();
25    }
26    // Metoda care obtine de la utilizator valoarea necunoscutei
27    public int obtineNecunoscuta() {  // Obtinerea de la utilizator a valorii necunoscutei
28      int necunoscuta = Integer.parseInt(JOptionPane.showInputDialog("Necunoscuta"));
29      return (necunoscuta);
30    }
31    // Metoda care calculeaza valoarea polinomului pt o valoare a necunoscutei
32    public int valoarePolinom(int necunoscuta) {
33      int polinom = 0;
34      int X_i = 1;
35      for (int i=0; i<=gradPolinom; i++) {
36         polinom = polinom + coeficienti[i]*X_i;    X_i = X_i * necunoscuta;
37      }
38      return (polinom);
39    }
40  }
```

```
1   public class RunPolinom4 {
2     public static void main(String[] args) {
3       Polinom4 poli4 = new Polinom4();                // se creaza un obiect Polinom4
4       poli4.obtineGrad();                  // se obtine gradul (nu se precizeaza cum!)
5       poli4.stabilesteCoeficienti();       // se obtin coeficientii (nu se precizeaza cum!)
6       poli4.afisarePolinom();                         // se afiseaza polinomul
7       int necunoscuta = poli4.obtineNecunoscuta();    // se obtine o valoare a necunoscutei
8       int polinom = poli4.valoarePolinom(necunoscuta); // se calculeaza pentru necunoscuta
9       System.out.println("P(" + necunoscuta + ") = " + polinom);  // se afiseaza valoarea
10    }
11  }
```

2.4. Generalizare, specializare si mostenire

Generalizarea = **extragerea elementelor comune** (atribute, operaţii şi constrângeri) ale unui ansamblu de clase **într-o nouă clasă mai generală**, denumită **superclasă,**

- **superclasa** este **o abstracţie a subclaselor ei,**
- **arborii de clase** sunt **construiţi pornind de la frunze**
- **utilizată cand elementele modelului au fost identificate**, pentru a obţine o descriere detaşată a soluţiilor
- semnifică "**este un (fel de)**" sau "**este ca**", şi priveşte doar clasele, adică **nu este instanţiabilă**

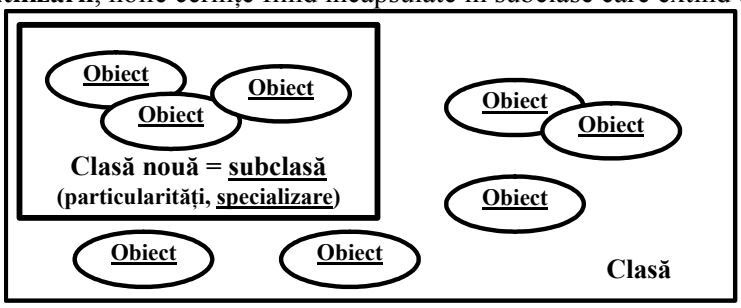


Generalizarea **actioneaza in OO la doua niveluri**:
- **clasele sunt generalizari ale ansamblurilor de obiecte** (un obiect este **de felul** specificat de o clasa),
- **superclasele sunt generalizari de clase** (obiectele **de felul** specificat in clasa sunt si **de felul** specificat in superclasa).

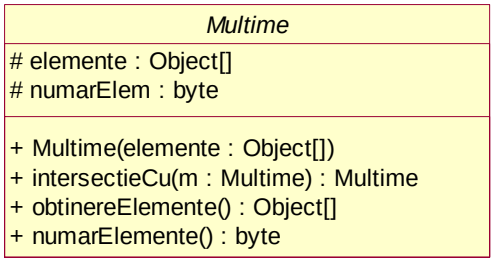
Limbajele OO ofera ambele mecanisme de generalizare. Limbajele care **ofera doar constructii numite obiecte** (si eventual clase) se pot numi **limbaje care lucreaza cu obiecte** (si eventual clase).

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

- utilă pentru **extinderea coerentă a unui ansamblu de clase**
- **bază a extinderii şi reutilizării**, noile cerinţe fiind încapsulate în subclase care extind coerent funcţiile existente

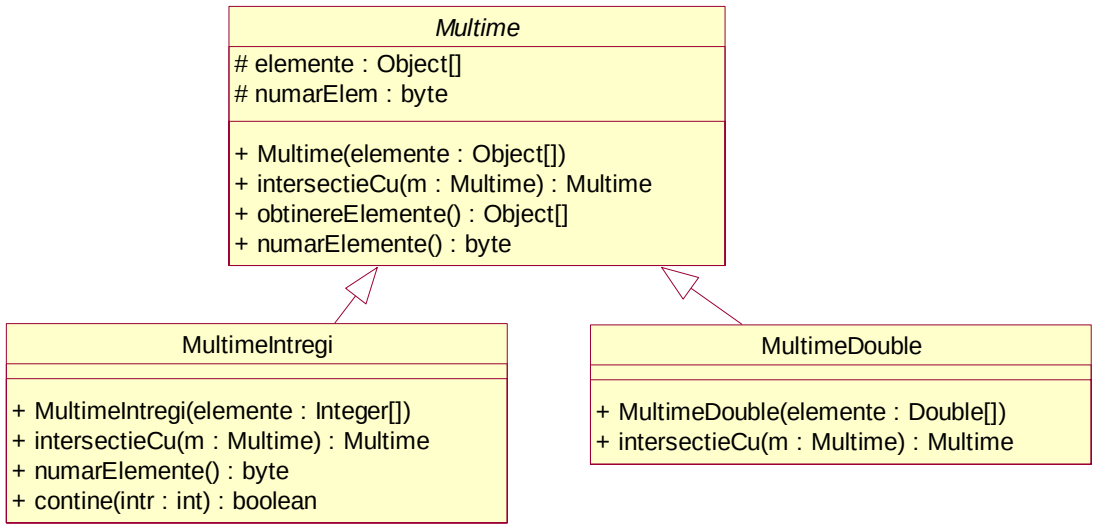


Moştenirea = **tehnică de generalizare** oferită de limbajele OO pentru a construi o clasă pornind de la una sau mai multe alte clase, **partajând atributele, operaţiile şi uneori constrângerile**, într-o ierarhie de clase.



Diagramei UML: **ii corespunde codul Java:**

```
 1  public abstract class Multime {                        // clasa declarata abstract
 2    protected Object[] elemente;
 3    protected byte numarElem;
 4
 5    public Multime(Object[] elemente) {       // parametru generic tip Object[]
 6        this.elemente = elemente;             // acces la obiectul curent cu this
 7        numarElem = (byte) elemente.length;   // conversie de tip de la int la byte
 8    }
 9    public abstract Multime intersectieCu(Multime m);  // metoda declarata abstract
10                                          // valoare returnata generica tip Multime
11    public Object[] obtinereElemente() { // valoare returnata generica tip Object[]
12        return elemente;
13    }
14    public byte numarElemente() {        // implementare de baza
15        return numarElem;
16    }
17  }
```

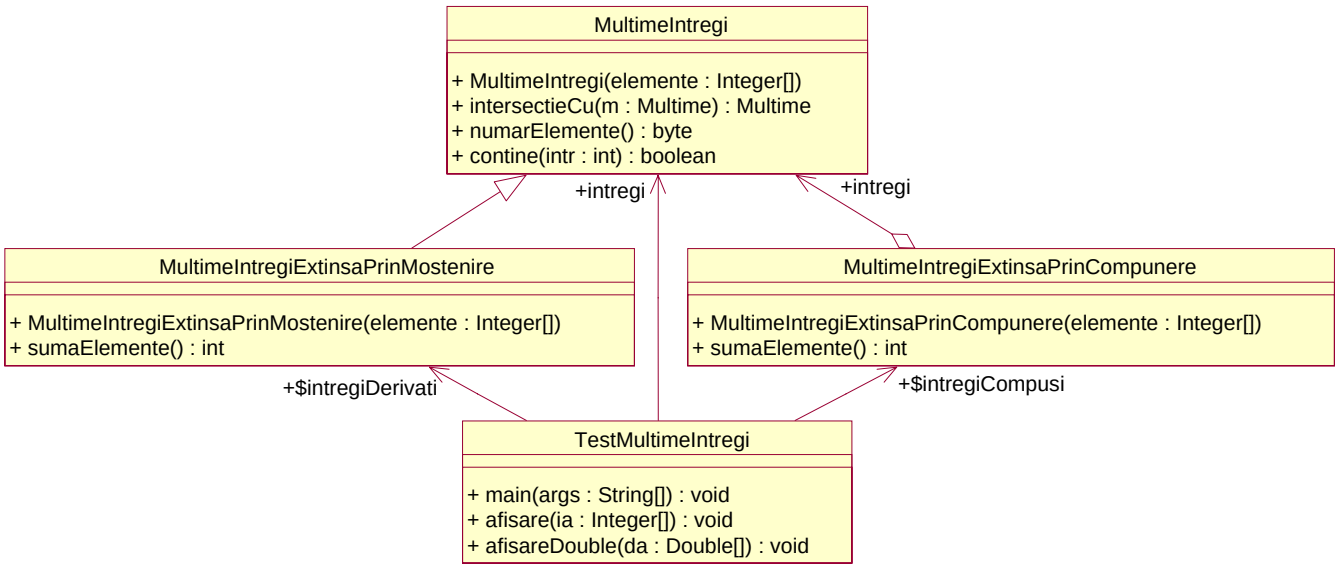


Diagramei UML:

ii corespunde codul Java:

```
 1  public class MultimeIntregi extends Multime {
 2
 3    public MultimeIntregi(Integer[] elemente) { // parametru concret tip Integer[]
 4        super(elemente);  // apelul constructorului clasei de baza Multime cu super
 5    }
 6
 7    public final Multime intersectieCu(Multime m) { // implementarea metodei
 8                                    // declarata abstract in clasa de baza
 9        Multime mNoua;
10        Integer[] elementeIntersectie;
11        int nrElemente = 0;
12
13        for (int i=0; i< elemente.length; i++) {
14          for (int j=0; j< m.elemente.length; j++) {
15            if (elemente[i].equals(m.elemente[j])) {
16              nrElemente++;
17            }
18          }
19        }
20        int index = 0;
21        elementeIntersectie = new Integer[nrElemente];
22        for (int i=0; i< elemente.length; i++) {
23          for (int j=0; j< m.elemente.length; j++) {
24            if (elemente[i].equals(m.elemente[j])) {
25              elementeIntersectie[index++] = new Integer(elemente[i].toString());
26            }
27          }
28        }
29        mNoua = new MultimeIntregi(elementeIntersectie);
30
31        return mNoua;
32    }
33
34    public byte numarElemente() {          // reimplementare (rescriere cod)
35        return (byte) elemente.length;     // conversie de tip de la int la byte
36    }
37
38    public boolean contine(int intr) {     // metoda noua
39        for (int i=0; i< elemente.length; i++) {
40          Integer inte = (Integer) elemente[i];
41          if (inte.intValue() == intr) {
42            return true;
43          }
44        }
45        return false;
46    }
47  }
```

Diagramei UML:



ii corespund codurile Java:

- ale unei clase care extinde prin mostenire (extindere):

```
 1  public class MultimeIntregiExtinsaPrinMostenire extends MultimeIntregi {
 2
 3    public MultimeIntregiExtinsaPrinMostenire(Integer[] elemente) {
 4        super(elemente);
 5    }
 6    public int sumaElemente() {     // metoda noua
 7        int suma = 0;
 8        Integer[] ti = (Integer[]) elemente;   // utilizare atribut mostenit elemente
 9        for (int i=0; i< ti.length; i++)     suma = suma + ti[i].intValue();
10        return suma;
11    }
12  }
```

- ale unei clase care extinde prin delegare (compunere):

```
 1  public class MultimeIntregiExtinsaPrinCompunere {
 2    public MultimeIntregi intregi;                  // obiect componenta
 3
 4    public MultimeIntregiExtinsaPrinCompunere(Integer[] elemente) {
 5        intregi = new MultimeIntregi(elemente);
 6    }
 7    public int sumaElemente() {     // metoda noua
 8        int suma = 0;
 9        Integer[] ti = (Integer[]) intregi.obtinereElemente();
10        for (int i=0; i< ti.length; i++)    suma = suma + ti[i].intValue();
11        return suma;
12    }
13  }
```

- ale unei clase care permite testarea comportamentului (si compararea modului de utilizare) in cele doua cazuri:

```
 1  public class TestMultimeIntregi {
 2    public MultimeIntregi intregi;
 3    public static MultimeIntregiExtinsaPrinMostenire intregiDerivati;
 4    public static MultimeIntregiExtinsaPrinCompunere intregiCompusi;
 5
 6    public static void main(String[] args) {
 7      int i;
 8
 9      Integer[] tablouA = { new Integer(1), new Integer(3), new Integer(5) };
10      MultimeIntregi multimeA = new MultimeIntregi(tablouA);
11
12      intregiCompusi = new MultimeIntregiExtinsaPrinCompunere(tablouA);
13      int suma = intregiCompusi.sumaElemente();
14      System.out.println("Suma elementelor " + suma);
15
16      intregiDerivati = new MultimeIntregiExtinsaPrinMostenire(tablouA);
17      suma = intregiDerivati.sumaElemente();
18      System.out.println("Suma elementelor " + suma);
19    }
20  }
```

Subclasele (clasele care extind prin mostenire) pot sa:

- mareasca gradul de **detaliere al obiectelor**:
  - **adaugand noi atribute**, inexistente in clasa de baza (**stari mai detaliate** ale obiectelor in subclasa)
  - **adaugand noi metode**, inexistente in clasa de baza (**comportament mai detaliat** al obiectelor in subclasa)
- reduca gradul de **abstractizare a obiectelor**:
  - **implementand eventualele metode abstracte** din clasa de baza (**comportament mai putin abstract**)
- introduca **diferentieri / specializari ale obiectelor**:
  - **redeclarand unele atribute** din clasa de baza (**ascunderea** atributelor cu acelasi nume – ***hiding***),
  - **reimplementand unele metode existente** in clasa de baza (**rescrierea** metodelor cu acelasi nume – ***overriding***)

Subclasele mostenesc (reutilizeaza) toate:

- **atributele** din clasa de baza **care nu sunt ascunse** (redeclarate),
- **metodele** din clasa de baza **care nu sunt rescrise** (reimplementate).

Subclasele nu mostenesc

- **constructorii** clasei de baza (**au constructori proprii**), dar **pot face apel la constructorii clasei de baza** (cu apelul `super()`, care trebuie sa fie prima declaratie din corpul constructorului subclasei),
- **atributele** si **metodele cu caracter global** (static), deoarece **tin strict de clasa in care au fost declarati**.

```
 1  public class ClasaDeBaza {
 2    protected int atributMostenit;    // atribut partajat cu subclasa (reutilizat)
 3    protected byte atributAscuns;     // atribut corespunzator clasei de baza
 4
 5    public void metodaMostenitaSupraincarcata(int argument) {
 6      // implementare corespunzatoare parametrului de tip int
 7    }
 8    public void metodaMostenitaSupraincarcata() {
 9      // implementare cu acelasi nume corespunzatoare lipsei parametrilor
10    }
11    public void metodaRescrisa() {
12      // implementare de baza
13    }
14  }
```

```
 1  public class ClasaCareExtinde extends ClasaDeBaza {
 2    protected byte atributAscuns;     // atribut corespunzator subclasei
 3    protected long atributNou;        // atribut nou, nepartajat cu clasa de baza
 4
 5    public void metodaRescrisa() {
 6      // reimplementare (rescriere a codului)
 7    }
 8    public void metodaNoua() {
 9      // metoda noua, nepartajata cu clasa de baza, specializare
10    }
11  }
```

```
 1  public class UtilizareClase {
 2    public static void main(String[] args) {
 3      ClasaDeBaza obiectDeBaza = new ClasaDeBaza();           // obiect din clasa de baza
 4
 5      obiectDeBaza.atributMostenit;   // utilizarea atributului partajat cu subclasa
 6      obiectDeBaza.atributAscuns;     // utilizarea atributului din clasa de baza
 7
 8      obiectDeBaza.metodaMostenitaSupraincarcata();
 9      obiectDeBaza.metodaMostenitaSupraincarcata(1000); // supraincarcare nume
10      obiectDeBaza.metodaRescrisa()                     // implementarea initiala
11
12      ClasaCareExtinde obiectExtins = new ClasaCareExtinde(); // obiect din subclasa
13
14      obiectExtins.atributNou;         // utilizarea atributului nou
15      obiectExtins.atributMostenit;    // utilizarea atributului partajat (reutilizare)
16      obiectExtins.atributAscuns;      // utilizarea atributului din subclasa (direct)
17      super.atributAscuns;             // utilizarea atributului din clasa de baza
18
19      obiectExtins.metodaMostenitaSupraincarcata(); // apelurile metodelor comune
20      obiectExtins.metodaMostenitaSupraincarcata(1000);
21      obiectExtins.metodaNoua();                  // apelul metodei noi din subclasa
22      obiectExtins.metodaRescrisa();              // apelul metodei din subclasa (rescrisa)
23      super.metodaRescrisa();                     // apelul metodei din clasa de baza
24    }
25  }
```

Orice clasa Java care nu extinde prin mostenire in mod explicit o alta clasa Java, extinde (prin mostenire) **in mod implicit clasa `Object`** (radacina ierarhiei de clase Java), clasa care contine metodele necesare tuturor obiectelor Java.

Urmatoarea **declaratie de clasa**:

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

este **echivalenta cu**:

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

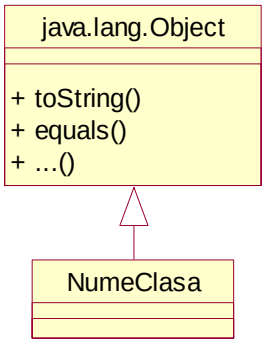


Diagrama UML corespunzatoare **codului Java** anterior:

Printre metodele declarate in clasa **`Object`** este si **`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:

```
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()`** (sau **`this.toString()`**) va conduce la executia noului cod, pe cand
- apelul **`super.toString()`** va conduce la executia codului din clasa extinsa (in acest caz codul implicit din **`Object`**).

Clasa Java care foloseste metoda **`toString()`** **mostenita de la** clasa **`Object`**:

```
1    public class Punct {
2      protected int x;                              // atribute
3      protected int y;
4      protected String numePunct;
5
6      public Punct(int a, int o, String id) {       // constructor
7        x = a;
8        y = o;
9        numePunct = id;
10     }
11     public void afisarePunct() {                  // metoda non-statica
12       System.out.println("Punctul " + this.toString()); // cod mostenit
13     }
14     public static void main(String[] args) {      // metoda principala
15       Punct p = new Punct(2, 1, "P");
16       p.afisarePunct();
17     }
18   }
```

Apelul **this.toString()** va returna un sir de caractere de forma "**Punct@1add2dd**", format prin concatenarea numelui clasei, "**Punct**", cu caracterul "**@**", si cu codul "**1add2dd**" (*hash code*).

Daca se modifica clasa Punct, **adaugandu-i o noua implementare metodei toString()**, astfel incat metoda **afisarePunct()** sa afiseze informatiile incapsulate in obiectul de tip **Punct** in formatul:

```
Punctul P (2, 1)
```

(se presupun **x**, **y**, si **numePunct** cu valorile **2**, **1** si respectiv "**P**"), **rezulta codul Java:**

```
1    public class Punct {
2      protected int x;
3      protected int y;
4      protected String numePunct;
5
6      public Punct(int a, int o, String id) {
7        x = a;
8        y = o;
9        numePunct = id;
10     }
11
12     public Punct(int xy, String numePunctDiagonala) { // supraincarcare nume
13       this.x = xy;
14       this.y = xy;
15       this.numePunct = numePunctDiagonala;
16     }                    // codul putea fi this(xy, xy, numePunctDiagonala);
17
18     public void afisarePunct() {
19       System.out.println("Punctul " + this.toString());  // cod nou
20     }
21
22     public String toString() {
23       return numePunct + " (" + this.x + ", " + this.y + ")";
24     }
25
26     public static void main(String[] args) {
27       Punct p = new Punct(2, 1, "P");
28       p.afisarePunct();
29     }
30   }
```

Apelul **this.toString()** va returna acum un sir de caractere de forma "**P (2, 1)**".

Metoda **equals()** **compara obiectul caruia i se aplica aceasta metoda cu un obiect pasat ca parametru**, returnand valoarea booleana **true** in cazul egalitatii si valoarea booleana **false** in cazul inegalitatii celor doua obiecte.

```
public boolean equals(Object obj) {   // semnatura metodei equals() mostenite de la Object
```

O noua clasa PunctExtins extinde prin mostenire clasa Punct, si **declara in mod explicit o metoda equals()** care rescrie codul metodei cu aceeasi semnatura a clasei **Object**:

```
1  class PunctExtins extends Punct {
2
3    public PunctExtins(int a, int o, String id) {
4      super(a, o, id);                    // reutilizare cod constructor Point()
5    }
6
7    public boolean equals(Object obj) {     // cod nou (rescriere - override)
8      if ((obj != null) && (obj instanceof PunctExtins)) {
9        PunctExtins celalaltPunct = (PunctExtins)obj;
10       return ((this.x == celalaltPunct.x) &&
11               (this.y == celalaltPunct.y) &&
12               (this.numePunct.equals(celalaltPunct.numePunct)));
13     }
14     return false;
15    }
16 }
```

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      if ((obj != null) && (obj instanceof String)) {
5        String otherString = (String) obj;        // conversie sigura
6        int n = this.count;
7        if (n == otherString.count) {
8          char v1[] = this.value;
9          char v2[] = otherString.value;;
10         int i = this.offset;
11         int j = otherString.offset;
12         while (n-- != 0)
13           if (v1[i++] != v2[j++]) return false;
14         return true;
15       }
16     }
17     return false;
18   }
```

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:

```
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 (in acest caz clasa **`Object`**).

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

Apelul constructorului din clasa extinsa se realizeaza utilizand apelul **`super()`,** apel care **trebuie sa fie prima instructiune din codul constructorului clasei care extinde**.

In cazul nostru, constructorul clasei care extinde (**`PunctExtins`**) trebuie sa apeleze mai intai constructorul din clasa extinsa (**`Punct`**) utilizand **`super()`**, orice alt cod al constructorului **`PunctExtins()`** urmand dupa acest apel:

```
1    public PunctExtins(int a, int o, String id) {
2      super(a, o, id);
3      // orice alt cod de initializare ...
4    }
```

Care dintre urmatoarele linii de cod va produce eroare?

```
1  public class UtilizarePunct {
2    public static void main(String[] args) {
3      Punct x = new Punct(3, 4, "X");
4
5      PunctExtins y = new Punct(5, 4, "Y");
6
7      Punct z = new PunctExtins(3, 2, "Z");
8
9      PunctExtins w = new PunctExtins(1, 4, "W");
10
11     Punct n = (Punct) new PunctExtins(3, -1, "N");
12
13     PunctExtins m = (PunctExtins) new Punct(5, -2, "M");
14   }
15 }
```

Care este tipul (clasa) fiecaruia dintre obiectele anterioare (`x, y, z, w, n, m`)?

Ce iesire va produce pe ecran urmatorul de cod?

```
1  public class Test1Equals {
2    public static void main(String[] args) {
3      Punct p1 = new Punct(2, 1, "P");  // obiect al clasei Punct
4      Punct p2 = p1;                    // noua referinta spre acelasi obiect
5      Punct p3 = new Punct(2, 1, "P");  // nou obiect cu acelasi continut
6      Punct p4 = new Punct(3, 2, "S");  // nou obiect cu alt continut
7
8      if (p1 == p2) System.out.println("p1 == p2");
9      else          System.out.println("p1 != p2");
10     if (p1 == p3) System.out.println("p1 == p3");
11     else          System.out.println("p1 != p3");
12     if (p1 == p4) System.out.println("p1 == p4");
13     else          System.out.println("p1 != p4");
14
15     if (p1.equals(p2)) System.out.println("p1.equals(p2)");
16     else               System.out.println("!p1.equals(p2)");
17     if (p1.equals(p3)) System.out.println("p1.equals(p3)");
18     else               System.out.println("!p1.equals(p3)");
19     if (p1.equals(p3)) System.out.println("p1.equals(p3)");
20     else               System.out.println("!p1.equals(p3)");)");
21   }
22 }
```

Ce iesire va produce pe ecran urmatorul de cod?

```
1  public class Test2Equals {
2    public static void main(String[] args) {
3      PunctExtins pex1 = new PunctExtins(2, 1, "P"); // obiect PunctExtins
4      PunctExtins pex2 = pex1;                       // noua referinta, acelasi obiect
5      PunctExtins pex3 = new PunctExtins(2, 1, "P"); // nou obiect, acelasi continut
6      PunctExtins pex4 = new PunctExtins(3, 2, "S");   // nou obiect, alt continut
7
8      if (pex1 == pex2) System.out.println("pex1 == pex2");
9      else              System.out.println("pex1 != pex2");
10     if (pex1 == pex3) System.out.println("pex1 == pex3");
11     else              System.out.println("pex1 != pex3");
12     if (pex1 == pex4) System.out.println("pex1 == pex4");
13     else              System.out.println("pex1 != pex4");
14
15     if (pex1.equals(pex2)) System.out.println("pex1.equals(pex2)");
16     else                   System.out.println("!pex1.equals(pex2)");
17     if (pex1.equals(pex3)) System.out.println("pex1.equals(pex3)");
18     else                   System.out.println("!pex1.equals(pex3)");
19     if (pex1.equals(pex4)) System.out.println("pex1.equals(pex4)");
20     else                   System.out.println("!pex1.equals(pex4)");
21   }
22 }
```

Ce iesire va produce pe ecran urmatorul de cod?

```
1  public class Test3Equals {
2    public static void main(String[] args) {
3      String s1 = new String("P");     // obiect al clasei String
4      String s2 = s1;                  // noua referinta spre acelasi obiect
5      String s3 = new String("P");     // nou obiect cu acelasi continut
6      String s4 = new String("S");     // nou obiect cu alt continut
7
8      if (s1 == s2) System.out.println("s1 == s2");
9      else              System.out.println("s1 != s2");
10     if (s1 == s3) System.out.println("s1 == s3");
11     else              System.out.println("s1 != s3");
12     if (s1 == s4) System.out.println("s1 == s4");
13     else              System.out.println("s1 != s4");
14
15     if (s1.equals(s2)) System.out.println("s1.equals(s2)");
16     else                   System.out.println("!s1.equals(s2)");
17     if (s1.equals(s3)) System.out.println("s1.equals(s3)");
18     else                   System.out.println("!s1.equals(s3)");
19     if (s1.equals(s4)) System.out.println("s1.equals(s4)");
20     else                   System.out.println("!s1.equals(s4)");
21   }
22 }
```