

2010 - 2011

Programare Orientata spre Obiecte (*Object-Oriented Programming*)

a.k.a. Programare Obiect-Orientata

Titular curs: Eduard-Cristian Popovici

Suport curs: <http://electronica08.curs.ncit.pub.ro/course/view.php?id=113>

Suport curs vechi: <http://discipline.elcom.pub.ro/POO-Java/> si

<http://electronica07.curs.ncit.pub.ro/course/view.php?id=132>

Continut curs Programare Orientata spre Obiecte (in Java)

1. Introducere in abordarea orientata spre obiecte (OO)

- 1.1. Obiectul cursului si relatia cu alte cursuri
- 1.2. Evolutia catre abordarea OO
- 1.3. Caracteristicile si principiile abordarii OO
- 1.4. Scurta recapitulare a programarii procedurale/structurate (introducere in limbajul Java)

2. Orientarea spre obiecte in limbajul Java

- 2.1. Obiecte si clase. Metode (operatii) si campuri (attribute)
- 2.2. Particularitati Java. Clase de biblioteca Java (de uz general)**
- 2.3. Clase si relatii intre clase. Asociere, delegare, agregare, compunere
- 2.4. Generalizare, specializare si mostenire
- 2.5. Clase abstracte si interfete Java
- 2.6. Polimorfismul metodelor
- 2.7. Clase pentru interfete grafice (GUI) din biblioteca Java Swing

3. Programarea la nivel socket cu Java (pe platforma Java SE)

- 3.1. Clase pentru fluxuri de intrare-iesire (IO)
- 3.2. Introducere in Protocolul Internet (IP) si stiva de protocoale IP
- 3.3. Socketuri flux (TCP) Java.
- 3.4. Clase Java pentru programe multifilare. Servere TCP multifilare
- 3.5. Socketuri datagrama (UDP) Java

2. Orientarea spre obiecte in limbajul Java

2.2. Particularitati Java. Clase de biblioteca Java (de uz general)

Reluarea problemei pasarea argumentelor prin valoare

Cazul pasarii unei valori primitive de tip `int`

```
// Program care incrementeaza o valoare intreaga
public class C1 {

    // declaratie (semnatura) metoda inc()
    public static void inc(int i) {

        i++; // i este parametru formal (pe scurt, parametru)
    }

    public static void main(String[] args) {

        int x = 10;

        inc(x); // apel metoda inc()

        System.out.println("x=" + x); // x este parametru actual (argument)
    } // Rezultat final: x = 10
}
```

Reluarea problemei pasarea argumentelor prin valoare

Cazul pasarii unui tablou de tip `int[]`

```
// Program care incrementeaza un element al unui tablou
public class C2 {

    // primeste o copie a valorii referintei, asa incat refera acelasi tablou
    public static void inc(int[] i) {

        i[0]++;    // este incrementat primul element al tabloului
    }

    public static void main(String[] args) {

        int[] x = {10};    // tablou cu un element, referit de x

        inc(x);    // este pasata valoarea referintei

        System.out.println("x[0]=" + x[0]); // primul element al tabloului

        // Rezultat final: x[0] = 11
    }
}
```

Reluarea problemei pasarea argumentelor prin valoare

Cazul pasarii unui **obiect care contine un camp public** (accesibil oricarui cod exterior) – caz in care se poate vorbi de “**lucrul cu**” **obiecte!**

```
// Program care incrementeaza un camp (atribut) public al unui obiect
public class C3 {

    public static void inc(ClasaInt i) { // primeste o copie a referintei cu
        // aceeasi valoare, asa incat refera acelasi obiect
        i.camp++; // e incrementat campul continut in obiect
    }

    public static void main(String[] args) {
        ClasaInt x = new ClasaInt(); // obiect continand camp public tip int
        x.camp = 10; // initializat cu valoarea 10
        inc(x); // este pasata referinta (valoarea ei)
        System.out.println("x.camp = " + x.camp); // Rezultat: x.camp = 11
    }
}

class ClasaInt {
    public int camp;
}
```

Reluarea problemei pasarea argumentelor prin valoare

Cazul pasarii unui **obiect care contine un camp privat** (inaccesibil oricarui cod exterior) si metode de acces – caz in care vorbim de “**orientare spre**” **obiecte!**

```
// Program care incrementeaza un camp (atribut) private al unui obiect
public class C4 {

    public static void inc(ClasaInt i) { // primeste o copie a referintei cu
        // aceeasi valoare, asa incat refera acelasi obiect
        i.setCamp(i.getCamp()+1); // e incrementat campul incapsulat in obiect
    }

    public static void main(String[] args) {
        ClasaInt x = new ClasaInt(); // obiect continand camp privat tip int
        x.setCamp(10); // initializat cu valoarea 10
        inc(x); // este pasata referinta (valoarea ei)
        System.out.println("x.getCamp()= " + x.getCamp()); // Rez: x.getCamp()=11
    }
}

class ClasaInt {
    private int camp;
    public void setCamp(int c) { camp = c; }
    public int getCamp() { return camp; }
}
```

Structura unei clase Java

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


Structura unei clase Java

Declaratia unei **clase** ([] semnifica element optional)

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

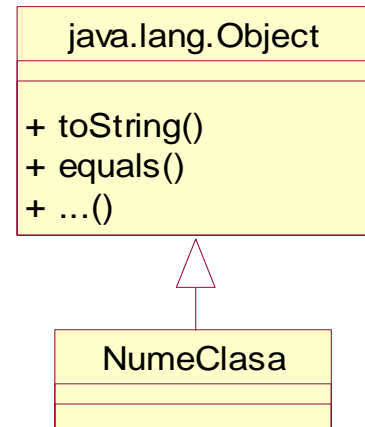
Element al declaratiei clasei	Semnificatie
public	Orice cod exterior are acces la membrii clasei
abstract	Clasa nu poate fi instantiata (din ea nu pot fi create direct obiecte, ci doar din subclasele ei non-abstracte)
final	Clasa nu poate avea subclase
class <i>NumeClasa</i>	Numele clasei este <i>NumeClasa</i>
extends <i>NumeSuperClasa</i>	Clasa extinde o superclasa <i>NumeSuperClasa</i> (este o subclasa a clasei <i>NumeSuperClasa</i>)
implements <i>NumeInterfata</i>	Clasa implementează o interfata <i>NumeInterfata</i>

Structura unei clase Java

Declaratia unei **clase**

– **dacă elementele opționale nu sunt declarate** compilatorul Java presupune **implicit** despre **clasa curent declarata** ca:

- doar **clasele din acelasi director (pachet)** cu clasa curenta **au acces la membrii clasei curente** (*prietenie de pachet*)
- este **instantiabila** (se pot crea obiecte avand ca tip clasa curenta)
- **poate avea subclase** (create extinzand clasa curenta)
- **extinde clasa **Object**** (radacina ierarhiei de clase Java) si nu implementeaza nici o interfata



```
class NumeClasa {
    // Corp clasa
}
```

Structura unei clase Java

Declaratia unui **atribut** ([] semnifica element optional)

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

Element al declaratiei	Semnificatie
public	Orice cod exterior clasei are acces la atribut
protected	Doar codul exterior din subclase sau aflat in acelasi director are acces la atribut
private	Nici un cod exterior nu are acces la atribut
static	Are caracter global, de clasa (e o variabila creata static, odata cu clasa, a carei locatie unica este partajata de toate obiectele clasei)
final	Valoarea atributului nu poate fi modificata dupa initializare (este o constanta)
transient	Semnificatia tine de serializarea obiectelor
volatile	Previne compilatorul de la efectua anumite optimizari asupra atributului
<i>tipAtribut numeAtribut</i>	Tipul este <i>tipAtribut</i> iar numele este <i>numeAtribut</i>
[=valInitiala];	Eventuala initializare

Structura unei clase Java

Declaratia unui atribut

– **dacă elementele opționale nu sunt declarate** compilatorul Java presupune **implicit** ca:

- **doar clasele din acelasi director** cu clasa curenta **au acces la atributul curent**
- atributul are **caracter de obiect**
 - fiecare obiect din clasa curenta are un astfel de atribut nepartajat cu alte obiecte, creat dinamic in momentul crearii obiectului
- **valoarea** atributul **poate fi modificata oricand** (este o variabila)

```
tipAtribut numeAtribut;
```

Structura unei clase Java

Declaratia unui **constructor** ([] semnifica element optional)

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

Element al declaratiei	Semnificatie
public	Orice cod exterior clasei are acces la constructor
protected	Doar codul exterior din subclase sau aflat in acelasi director are acces la constructor
private	Nici un cod exterior nu are acces la constructor
NumeClasa	Numele constructorului este NumeClasa
(listaParametri)	Lista de parametri primiti de constructor, despartiti prin virgule, cu format <i>tipParametru numeParametru</i>

- **dacă elementele opționale nu sunt declarate** compilatorul presupune ca:
 - **doar clasele din acelasi director** cu clasa curenta **au acces la el**

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

Structura unei clase Java

Declaratia unei **metode** ([] semnifica element optional)

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

Dacă **elementele opționale nu sunt declarate** se presupune **implicit** ca:

- doar codurile claselor din acelasi director cu clasa curenta au acces la metoda curenta,
- are caracter de obiect (este creata dinamic in momentul crearii obiectului),
- este implementata (are corp),
- poate fi rescrisa (reimplementata) in subclase (create extinzand clasa curenta),
- este implementata in Java
- nu are protectie la accesul concurent la informatii partajate
- nu are parametri,
- nu “arunca” (declanseaza) exceptii.

Structura unei clase Java

Element al declaratiei metodei	Semnificatie
<code>public</code>	Orice cod exterior clasei are acces la metoda
<code>protected</code>	Doar codul exterior din subclase sau aflat in acelasi director are acces la metoda
<code>private</code>	Nici un cod exterior nu are acces la metoda
<code>static</code>	Are caracter global, de clasa (este creata static, odata cu clasa)
<code>abstract</code>	Nu are implementare (trebuie implementata in subclase) si impune declararea <code>abstract</code> a clasei din care face parte (prin urmare clasa din care face parte nu poate avea instante)
<code>final</code>	Nu poate fi rescrisa implementarea metodei
<code>native</code>	Metoda implementata in alt limbaj
<code>synchronized</code>	Are protectie la accesul concurent la informatii partajate
<code>tipReturnat numeMetoda</code>	Tipul returnat este <code>tipReturnat</code> iar numele <code>numeMetoda</code>
<code>(listaParametri)</code>	Lista de parametri primiti de metoda, despartiti prin virgule, cu formatul <code>tipParametru numeParametru</code>
<code>throws NumeExceptie</code>	Metoda arunca exceptia <code>NumeExceptie</code>

Scopul variabilelor

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 și distrusă**.

Exista **4 categorii** de **scop** al variabilelor Java:

1. **Camp Java** sau **atribut** sau **variabilă membru** (*member variable*)

- este **membru** al unei clase sau al unui obiect,
- poate fi **declarat oriunde în clasă**, dar **nu într-o metodă**,
- este disponibilă **în tot codul** clasei

2. **Variabilă locală** (*local variable*)

- poate fi declarată **oriunde într-o metodă** sau **într-un bloc de cod** al metodei
- este disponibilă **în codul metodei**, din **locul de declarare** și **până la sfârșitul blocului in care e declarata**

Scopul variabilelor

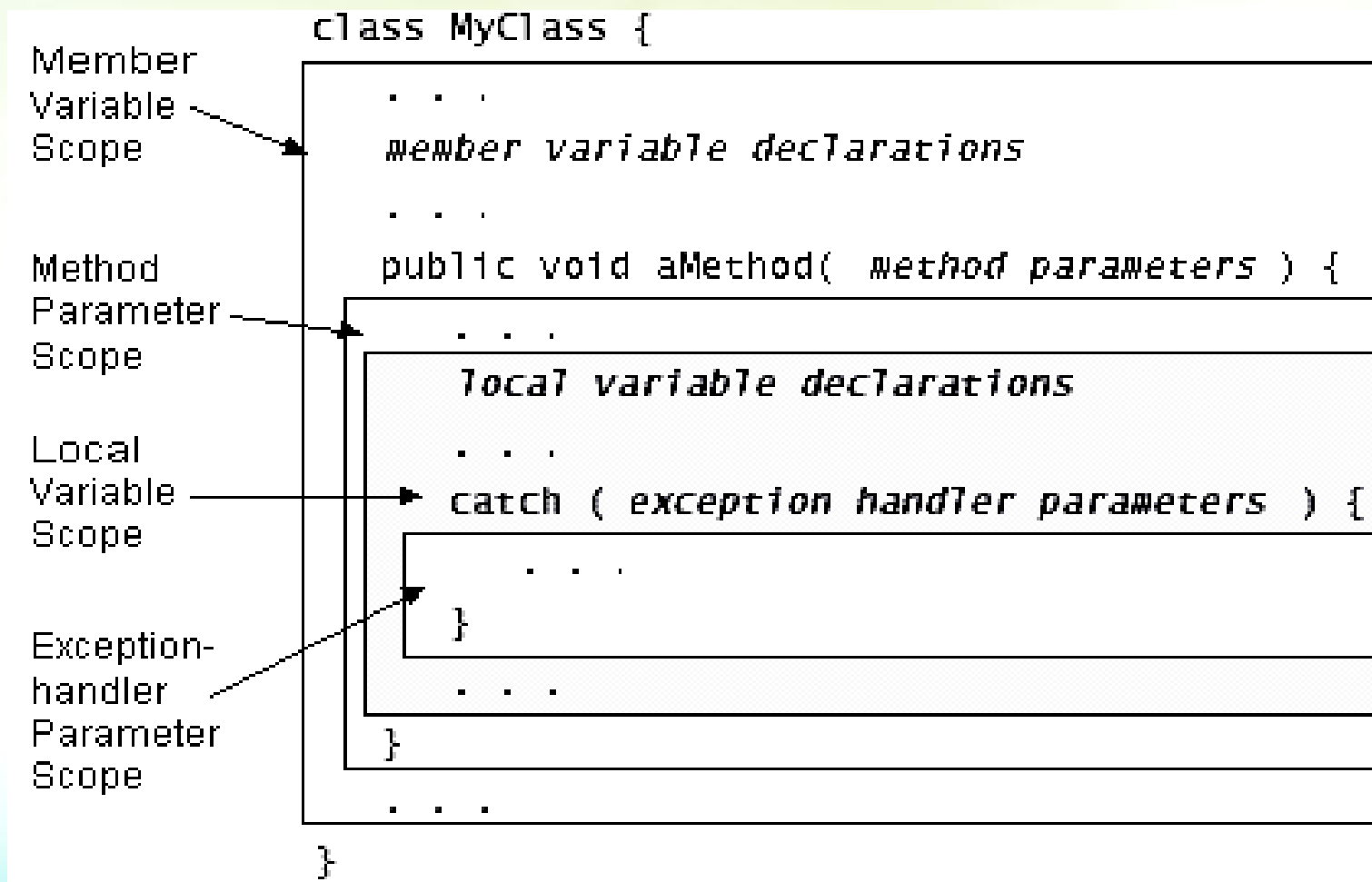
3. Parametrul unei metode

- este **argumentul formal** al metodei
- este utilizat **pentru a se pasa valori** metodei
- este disponibil **în întreg codul metodei**

4. Parametrul unei rutine de tratare a exceptiilor (*handler* de excepție)

- este **argumentul formal** al *handler-ului* de excepție
- este utilizat **pentru a se pasa valori *handler-ului*** de excepție
- este disponibil **în întreg codul *handler-ului*** de excepție

Scopul variabilelor



Scopul variabilelor

```
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) { // real = parametru metoda
7         this.real = real; // real = atribut, real = parametru
8     }
9     public void setImag(double imag) { // imag = parametru metoda
10        this.imag = imag; // imag = atributul, imag = parametru
11    }
12
13    public static void main(String[] args) { // args = parametru metoda
14        double real = Double.parseDouble(args[0]); // real = variabila locala
15        double imag = Double.parseDouble(args[1]); // imag = variabila locala
16
17        Complex c = new Complex(); // c = variabila locala
18        c.setReal(real); // echiv cu c.real = real // c, real = var. locale
19        c.setImag(imag); // c, imag = var. locale
20
21        System.out.println("{ " + c.real + // c.real = atributul lui c
22            ", " + c.imag + " }"); // c.imag = atributul lui c
23    }
24 } // sfarsitul corpului clasei
```

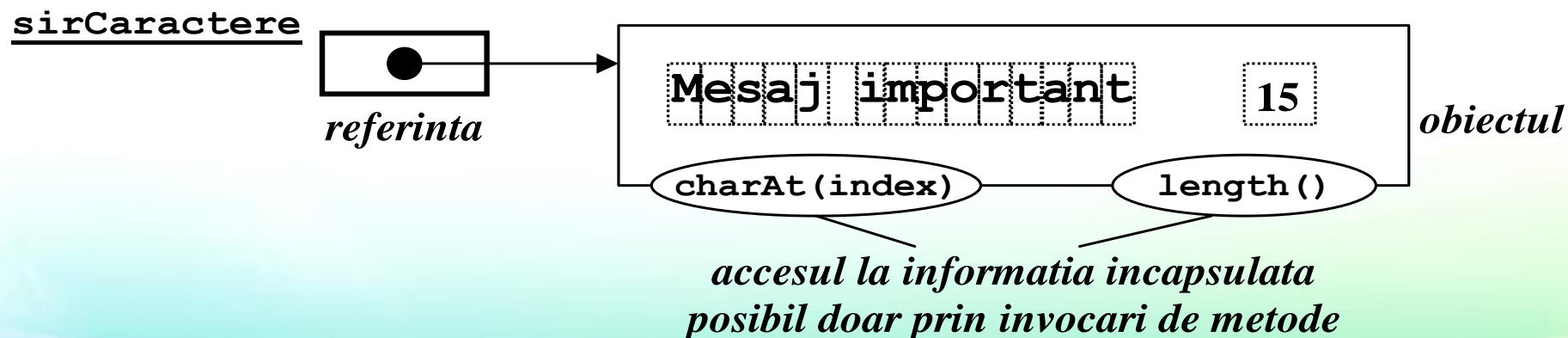
Clasa String

- incapsuleaza **siruri de caractere** – in **obiecte nemodificabile** (*immutable*)
- face parte din **pachetul de clase implicite** (*java.lang*)
- **crearea unei referinte la obiect** de tip **String**, numita **sirCaractere**, initializata implicit cu **null**:

```
String sirCaractere; sirCaractere null referinta obiect de tip String
```

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

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



Clasa String

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

```
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 caracter):

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

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

```
tablouCaractere.length // prin campul length
```

Clasa String

Lucrul cu obiecte de tip String

```
1 // variabile referinta
2 String a; // referinta la String neinitializata
3 String b = null; // referinta la String initializata explicit cu null
4
5 // constructie siruri de caractere utilizand constructori String()
6 String sirVid = new String(); // sirVid.length() = 0, sirVid = ""
7
8 byte[] tabByte = {65, 110, 110, 97}; // coduri ASCII
9 String sirTablouByte = new String(tabByte); // sirTablouByte = "Anna"
10
11 char[] tabChar = {'T', 'e', 's', 't'};
12 String sirTabChar = new String(tabChar); // sirTabChar = "Test"
13
14 String s = "Sir de caractere";
15 String sir = new String(s); // sir = "Sir de caractere"
```

Clasa String

Lucrul cu obiecte de tip String

```
1 // constructie siruri de caractere utilizand metode de clasa
2 boolean adevarat = true;
3 String sirBoolean = String.valueOf(adevarat); // sirBoolean = "true"
4
5 char caracter = 'x';
6 String sirChar = String.valueOf(caracter); // sirChar = "x"
7
8 char[] tab2Char = {'A', 'l', 't', ' ', 't', 'e', 's', 't'};
9 String sirTab2Char = String.valueOf(tab2Char); // sirTabChar2="Alt test"
10
11 int numar = 10000;
12 String sirInt = String.valueOf(numar); // sirInt = "10000"
13
14 double altNumar = 2.3;
15 String sirDouble = String.valueOf(altNumar); // sirDouble = "2.3"
```

Clasa String

Lucrul cu obiecte de tip String

```
1 // echivalente functionale - 1
2 char[] caractere = {'t', 'e', 's', 't'};
3 String sir = new String(caractere);
4 // echivalent cu String sir = String.valueOf(caractere);
5
6 // echivalente functionale - 2
7 char[] caractere = {'t', 'e', 's', 't', 'a', 'r', 'e'};
8 String sir = new String(caractere, 2, 5);
9 // echivalent cu String sir = String.valueOf(caractere, 2, 5);
10
11 // echivalente functionale - 3
12 String original = "sir";
13 String copie = new String(original);
14 // echivalent cu String copie = original.toString();
15 // echivalent cu String copie = String.valueOf(original);
16 // echivalent cu String copie = original.substring(0);
17
18 // complementaritati functionale
19 String sir = "test";
20 byte[] octeti = sir.getBytes();
21 String copieSir = new String(octeti);
```


Clasa String – exemplu de analiza lexicala (parsing)

```
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.";
6         String[] cuvinteCheie = { "string" , "token" };
7         // Pentru toate cuvintele cheie cautate
8         for (int i=0; i<cuvinteCheie.length; i++) {
9             String text = textAnalizat;
10            int pozitie=0;
11
12            // Daca un anumit cuvant cheie este gasit intr-un anumit text
13            // Varianta cu String.indexOf()
14            while ( text.indexOf(cuvinteCheie[i]) > -1 ) {
15                pozitie = pozitie + text.indexOf(cuvinteCheie[i])+1;
16
17                // Informeaza utilizatorul (indicand si pozitia)
18                System.out.println("Cuvantul cheie \" + cuvinteCheie[i] +
19                    "\" a fost gasit in text pe pozitia \" + pozitie + \"\n");
20                text = text.substring(text.indexOf(cuvinteCheie[i])+1);
21            }
22        }
23    }
24 }
```

Clasa StringBuffer – alternativa a carei obiecte sunt modificabile

```
1 class ReverseString {
2     public static String reverseIt(String source) {
3         int i, len = source.length();
4         StringBuffer dest = new StringBuffer(len);
5
6         for (i = (len - 1); i >= 0; i--)
7             dest.append(source.charAt(i));
8         return dest.toString();
9     }
10 }
11 public class StringsDemo {
12     public static void main(String[] args) {
13         String palindrome = "ele fac cafele";
14         String reversed = ReverseString.reverseIt(palindrome);
15         System.out.println(reversed);
16     }
17 } // se va afisa elefac caf ele
```

Codul:

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

este compilat ca:

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

Clasa Integer

Lucrul cu obiecte de tip Integer

- **incapsuleaza** intregi **int** – in **obiecte nemodificabile** (*immutable*)

```
1 // declarare variabile de tip intreg
2 // int - primitiv
3 int i, j, k; // intregi ca variabile de tip primitiv
4 // Integer - obiect care incapsuleaza un int
5 Integer m, n, o; // intregi incapsulati in obiecte Integer
6
7 // si variabile de tip String
8 String s, r, t; // siruri de caractere (incapsulate in obiecte)
9
10 // constructia intregilor incapsulati utilizand constructori ai clasei
11 i = 1000;
12 m = new Integer(i); // echivalent cu m = new Integer(1000);
13
14 r = new String("30");
15 n = new Integer(r); // echivalent cu n = new Integer("30");
16
17 // constructia intregilor incapsulati utilizand metode de clasa
18 t = "40";
19 o = Integer.valueOf(t); // echivalent cu o = new Integer("40");
```

Clasa Integer

Lucrul cu obiecte de tip Integer

```
1 // conversia intregilor incapsulati la valori numerice primitive
2 // obiectul m incapsuleaza valoarea 1000
3 byte iByte = m.byteValue(); // diferit de 1000! (trunchiat)
4
5 int iInt = m.intValue(); // = 1000
6
7 float iFloat = m.floatValue(); // = 1000.0F
8
9 double iDouble = m.doubleValue(); // = 1000.0
10
11 // conversia valorilor intregi primitive la siruri de caractere
12 String douaSute = Integer.toString(200); // metoda de clasa (statica)
13
14 String oMieBinary = Integer.toBinaryString(1000); // metoda de clasa
15
16 String oMieHex = Integer.toHexString(1000); // metoda de clasa
17
18 // conversia sirurilor de caractere la valori intregi primitive
19 int oSuta = Integer.parseInt("100"); // metoda de clasa (statica)
```

Tratarea exceptiilor – blocurile `try {} catch (ex) {}`

In programul urmator

- in cazul in care argumentul nu are format intreg
- apelul metodei `parseInt()`
 - genereaza o exceptie de tip `NumberFormatException` (definita in `java.lang`)
- exceptia trebuie tratata cu un bloc `try {} catch (NumberFormatException ex) {}`

```
1 public class VerificareArgumenteIntregi {
2     public static void main(String[] args) {
3         int i;
4
5         for ( i=0; i < args.length; i++ ) {
6             try {
7                 System.out.println(Integer.parseInt(args[i]));
8             }
9             catch (NumberFormatException ex) {
10                System.out.println("Argumentul " + args[i] +
11                    " nu are format numeric intreg");
12            }
13        }
14    }
15 }
```

Tratarea exceptiilor – blocurile `try {} catch (ex) {}`

Formatul blocului de tratare a unei exceptii de tip `NumberFormatException`:

```
1      try {  
2  
3          // aici este plasata secventa de cod  
4          // care poate genera exceptia  
5  
6      }  
7      catch (NumberFormatException ex) {  
8  
9          // aici este plasata secventa de cod  
10         // care trateaza exceptia  
11  
12     }
```

Tratarea exceptiilor – blocurile try {} catch (ex) {}

```
1 public class ClasificareArgumenteConsola {
2
3     // stabilirea la lansare a valorilor, ca argumente ale programelor
4     public static void main(String[] args) {
5         int i;
6         for ( i=0; i < args.length; i++ ) {
7
8             try {
9                 int intreg = Integer.parseInt(args[i]);
10                System.out.println("Argumentul " + intreg + "are format intreg");
11            }
12            catch (NumberFormatException ex1) {
13
14                try {
15                    double real = Double.parseDouble(args[i]);
16                    System.out.println("Argumentul " + real + " are format real");
17                }
18                catch (NumberFormatException ex2) {
19                    System.out.println("Argumentul " + args[i] + " nu este numar");
20                }
21            }
22        }
23    }
24 }
```