2011 - 2012

Programare **O**rientata **spre O**biecte
(*Object-Oriented Programming*)

a.k.a. Programare Obiect-Orientata

Titular curs: Eduard-Cristian Popovici

Suport curs: *http://discipline.elcom.pub.ro/POO-Java/*

2. Orientarea spre obiecte in limbajul Java

2.5. Clase abstracte si interfete Java

Exemplu de clasa (de baza) abstracta

```
public abstract class Multime {          // clasa declarata abstract
  // atribute mostenite, reutilizate in subclase
  protected Object[] elemente;
  protected byte numarElem;
  // constructorul nu e reutilizabil
  public Multime(Object[] elemente) {    // parametru generic tip Object[]
    this.elemente = elemente;            // acces la obiectul curent cu this
    numarElem = (byte) elemente.length; // conversie explicita int la byte
  }
  // metoda declarata abstract, polimorfa (va fi rescrisa in subclase)
  // valoarea returnata e generica tip Multime
  public abstract Multime intersectieCu(Multime m);

  // metoda care va fi mostenita, reutilizata
  // valoarea returnata e generica tip Multime
  public Object[] obtinereElemente() {
    return elemente;
  }
  // metoda care va fi mostenita, reutilizabila
  // polimorfa (va fi rescrisa in subclase)
  public byte numarElemente() {
    return numarElem;
  }
}
```

Multime
elemente : Object[] # numarElem : byte
+ Multime(elemente : Object[]) + intersectieCu(m : Multime) : Multime + obtinereElemente() : Object[] + numarElemente() : byte

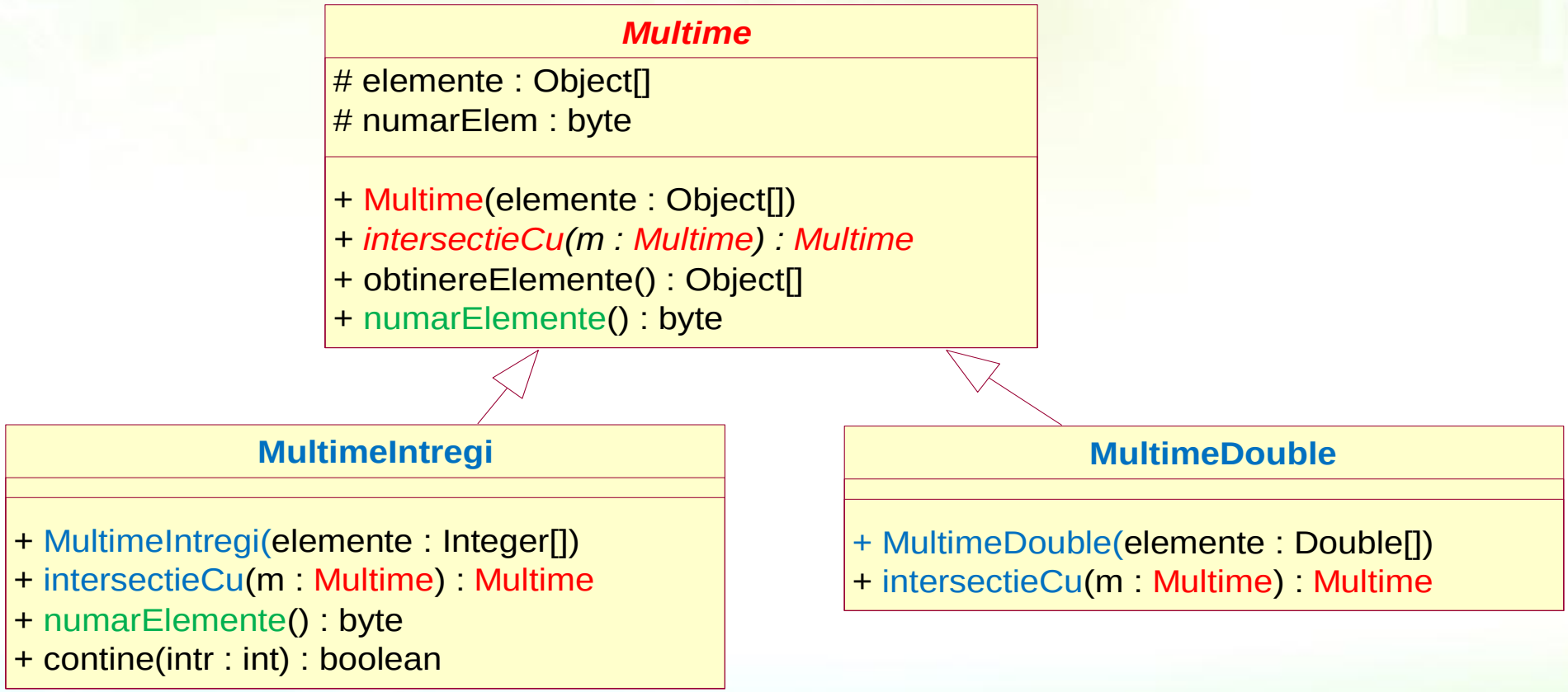
Exemplu de subclasa concreta care extinde clasa abstracta

```
public class MultimeIntregi extends Multime {

  // constructorul nu e reutilizabil
  // parametru concret tip Integer[]
  public MultimeIntregi(Integer[] elemente) {
    // apelul constructorului clasei de baza Multime
    super(elemente);
  }

  // reimplementare (rescriere cod), polimorfism (pseudo-extindere)
  public byte numarElemente() {
      return (byte) elemente.length;  // conversie de tip de la int la byte
  }

  // metoda noua (extindere 100%)
  public boolean contine(int intr) {
     for (int i=0; i< elemente.length; i++) {
        Integer inte = (Integer) elemente[i];
        if (inte.intValue() == intr) {
          return true;
        }
      }
      return false;
  }
```

Exemplu de subclasa concreta

```
// implementarea metodei care fusese declarata abstract in clasa de baza
// pseudo-extindere / polimorfism / rescriere
public final Multime intersectieCu(Multime m) {
    Multime mNoua;
    Integer[] elemIntersectie;
    int nrElemente = 0;
    for (int i=0; i< elemente.length; i++) {
      for (int j=0; j< m.elemente.length; j++) {
        if (elemente[i].equals(m.elemente[j]))    nrElemente++;
      }
    }
    int index = 0;
    elemIntersectie = new Integer[nrElemente];
    for (int i=0; i< elemente.length; i++) {
      for (int j=0; j< m.elemente.length; j++) {
        if (elemente[i].equals(m.elemente[j])) {
          elemIntersectie[index++] = new Integer(elemente[i].toString());
        }
      }
    }
    mNoua = new MultimeIntregi(elemIntersectie);
    return mNoua;
  }
}
```

Relatiile intre clasa abstracta de baza si subclasele concrete



Exemplu de subclasa care extinde prin mostenire

```
// noua clasa extinde prin mostenire clasa MultimeIntregi
public class MultimeIntregiExtinsaPrinMostenire extends MultimeIntregi {

   // constructorul nu e reutilizabil
   public MultimeIntregiExtinsaPrinMostenire(Integer[] elemente) {

      // apelul constructorului clasei de baza MultimeIntregi
      super(elemente);
   }

   // metoda noua (extindere 100%)
   public int sumaElemente() {
      int suma = 0;
      Integer[] ti = (Integer[]) elemente;  // utilizare atribut mostenit
      for (int i=0; i< ti.length; i++) {
         suma = suma + ti[i].intValue();
      }
      return suma;
   }
}
```

Exemplu de clasa care extinde prin agregare

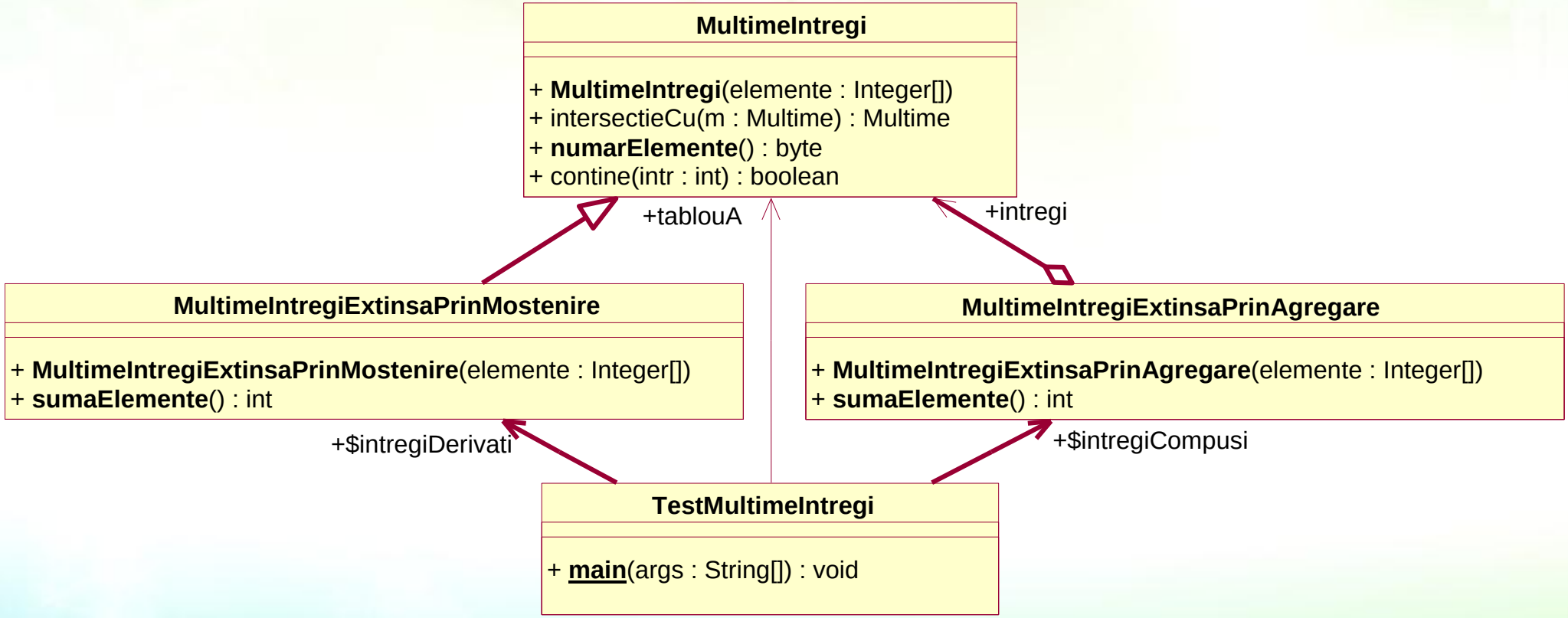
```
// noua clasa extinde clasa MultimeIntregi prin agregare
// folosind un obiect al ei drept atribut (camp)
public class MultimeIntregiExtinsaPrinAgregare {

   // atribut, obiect, componenta
   public MultimeIntregi intregi;

   // constructorul noii clase
   public MultimeIntregiExtinsaPrinAgregare(Integer[] elemente) {

      // apelul constructorului clasei MultimeIntregi a atributului
      intregi = new MultimeIntregi(elemente);
   }

   // metoda noua (extindere 100%)
   public int sumaElemente() {
      int suma = 0;
      Integer[] ti = (Integer[]) intregi.obtinereElemente();
      for (int i=0; i< ti.length; i++) {
         suma = suma + ti[i].intValue();
      }
      return suma;
   }
}
```

Exemplu de clasa care permite testarea claselor anterioare

```
public class TestMultimeIntregi {

  public MultimeIntregi intregi;
  public static MultimeIntregiExtinsaPrinMostenire intregiDerivati;
  public static MultimeIntregiExtinsaPrinAgregare intregiCompusi;

  public static void main(String[] args) {
    int i;
    Integer[] tablouA = { new Integer(1), new Integer(3), new Integer(5) };
    MultimeIntregi multimeA = new MultimeIntregi(tablouA);

    intregiCompusi = new MultimeIntregiExtinsaPrinAgregare(tablouA);
    int suma = intregiCompusi.sumaElemente();
    System.out.println("Suma elementelor " + suma);

    intregiCompusi.intregi.numarElemente(); // seamana cu System.out.println()

    intregiDerivati = new MultimeIntregiExtinsaPrinMostenire(tablouA);
    suma = intregiDerivati.sumaElemente();
    System.out.println("Suma elementelor " + suma);

    intregiDerivati.numarElemente();
  }
}
```

Relatiile intre clasa de baza si clase care extind

Extindere prin mostenire vs. extindere prin agregare



Efectele utilizarii mostenirii

Subclasele (clasele care **extind pe baza de mostenire**) pot

- sa mareasca gradul de detaliere al obiectelor

  (care este **extindere 100%**)

  - adaugand **noi atribute**, inexistente in clasa de baza

    (stari mai detaliate / complexe ale obiectelor in subclasa)

  - adaugand **noi metode**, inexistente in clasa de baza

    (comportament mai detaliat al obiectelor in subclasa)

- sa reduca gradul de abstractizare a obiectelor

  (prin **rescriere**, care este **pseudo-extindere**, si duce la **polimorfism**)

  - **implementand eventualele metode abstracte** din clasa de baza

    (comportament mai putin abstract)

Efectele utilizarii mostenirii

Subclasele (clasele care **extind pe baza de mostenire**) pot

- sa introduca **diferentieri** / **specializari** ale obiectelor

(prin **rescriere**, care este **pseudo-extindere**, si duce la **polimorfism**)

- **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***)

Efectele utilizarii mostenirii

Subclasele mostenesc (**reutilizeaza**)

- toate **atributele** din clasa de baza **care nu sunt ascunse**

  (nu sunt redeclarate)

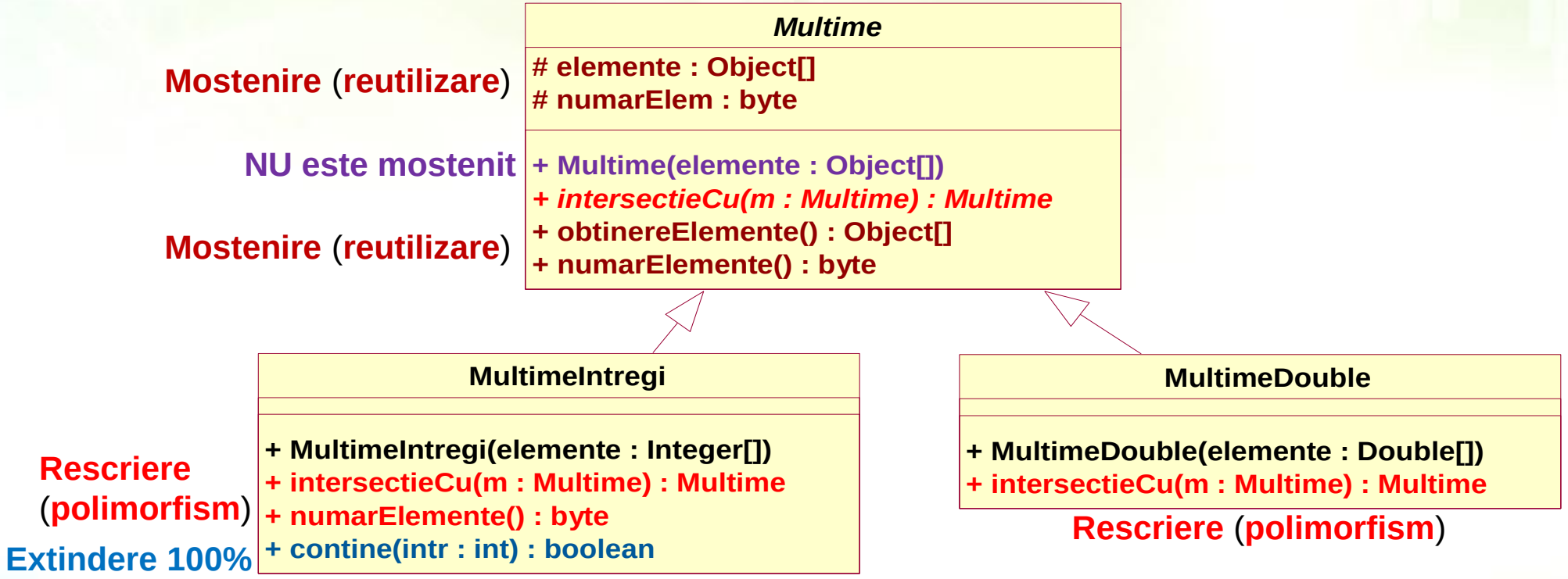
- toate **metodele** din clasa de baza **care nu sunt rescrise**

  (nu sunt 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 (declarate **static**)
  - deoarece **tin strict de clasa in care au fost declarati**

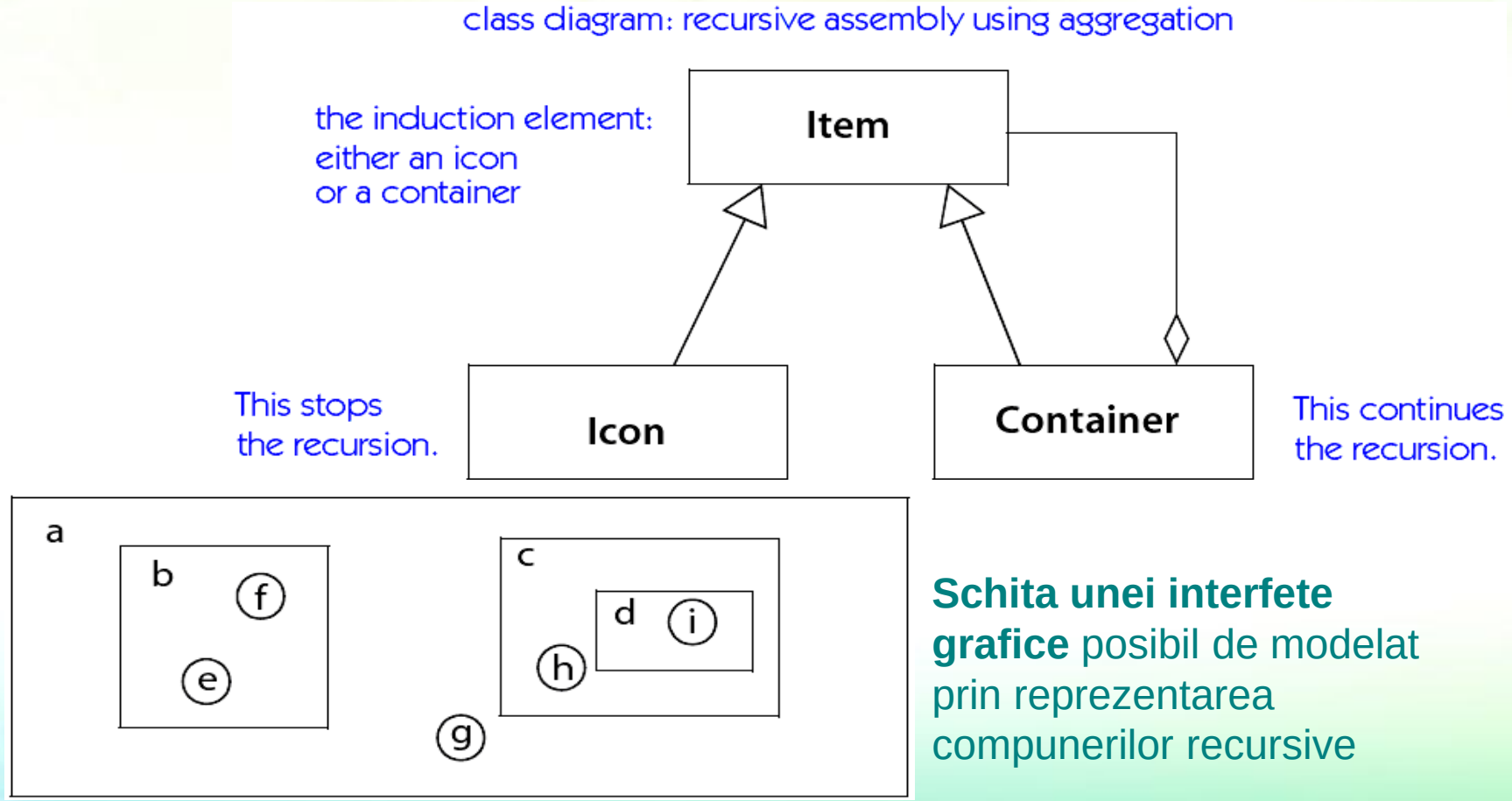
Efectele utilizarii mostenirii



Utilizarea generalizarii si a agregarii

Reprezentarea agregarilor recursive (*pattern-ul Composite*)

– exemplul **containerelor de elemente ale interfetelor grafice**



Schita unei interfete grafice posibil de modelat prin reprezentarea compunerilor recursive

Utilizarea generalizarii si a agregarii

Codul Java echivalent

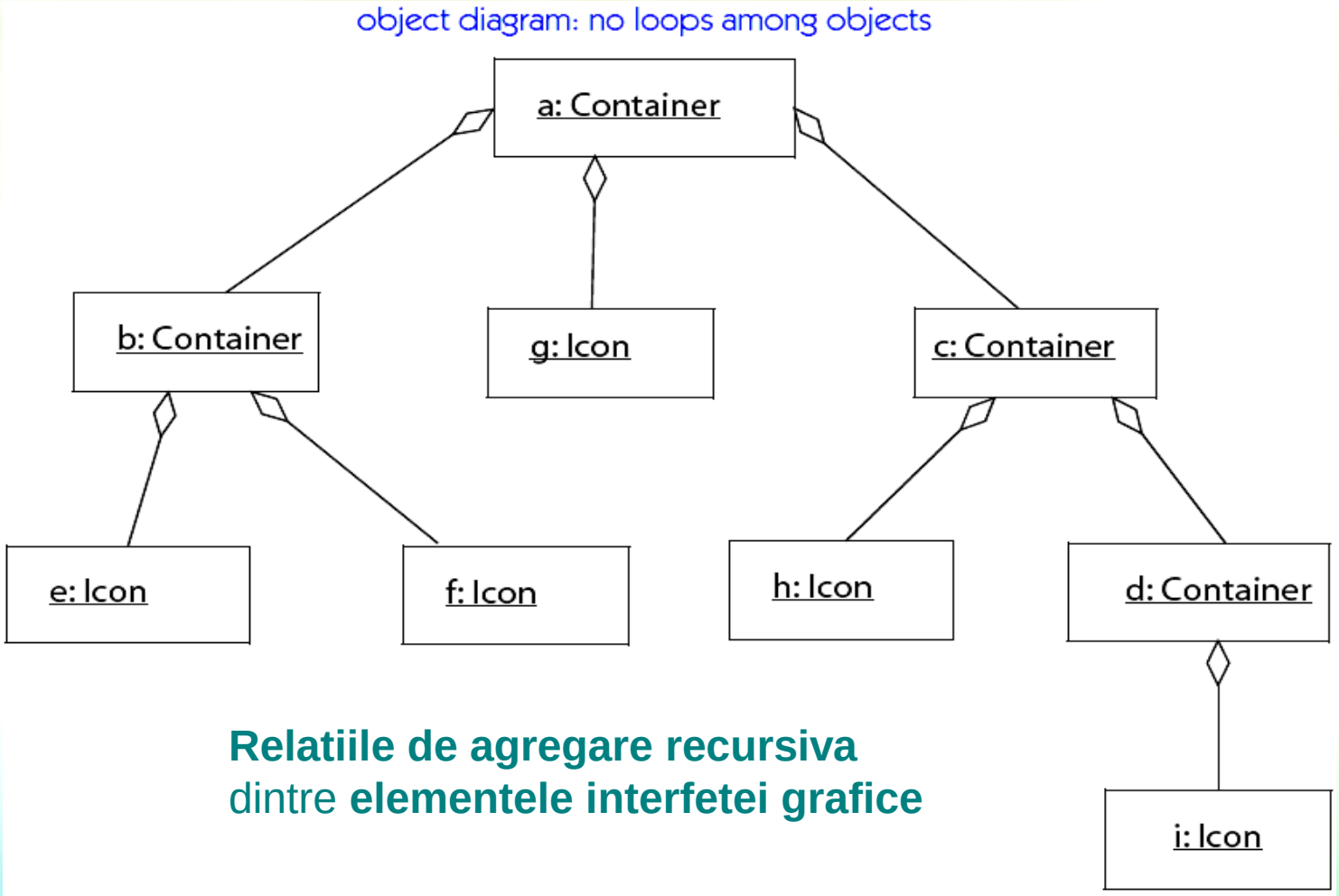
```
1    public abstract class Item {
2       // corpul clasei Item
3    }
```

```
1    public class Container extends Item {
2
3       ArrayList<Item> items;          // parti agregate
4
5       public void add(Item item) {    // legarea partilor
6          items.add(item);
7       }
8    }
```

```
1    public class Icon extends Item {
2       // corpul clasei Icon
3    }
```

Utilizarea generalizarii si a agregarii

Diagrama de obiecte care prezinta agregarile recursive



Relatiile de agregare recursiva
dintre **elementele interfetei grafice**

Utilizarea generalizarii si a agregarii

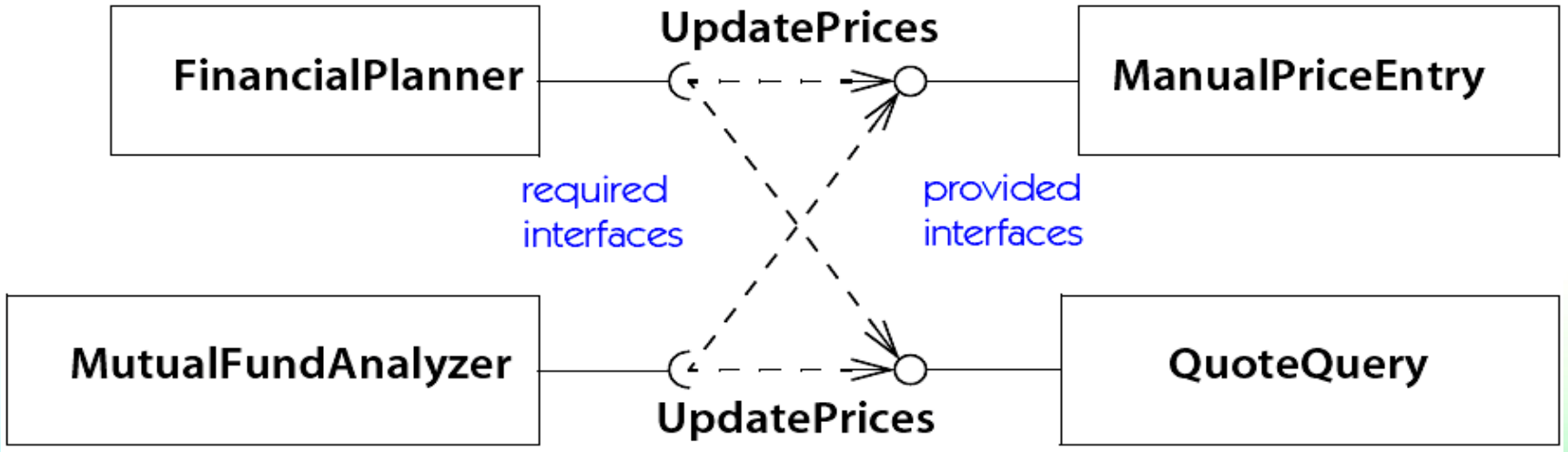
Codul Java necesar pentru a obtine aceste legaturi intre obiecte

```
Container a = new Container(); // containerul radacina
Container b = new Container(); // componenta a cont. radacina
Container c = new Container(); // componenta a cont. radacina
Container d = new Container();
Icon e = new Icon();
Icon f = new Icon();
Icon g = new Icon(); // componenta a containerului radacina
Icon h = new Icon();
Icon i = new Icon();
a.add(b); // adaugarea componentelor in containerul radacina
a.add(c); // adaugarea componentelor in containerul radacina
a.add(g); // adaugarea componentelor in containerul radacina
b.add(e);
b.add(f);
c.add(d);
c.add(h);
d.add(i);
```

Interfetele Java (un fel de clase complet abstracta)

Interfata

- **tip de date** folosit pentru a **descrie comportamentul vizibil** al unei clase, componente sau al unui pachet
- **cea oferita** e reprezentata in UML ca un mic **cerc legat printr-un segment de elementul care ofera serviciile** din spatele interfetei
- **cea necesara** e reprezentata in UML ca un mic **semicerc legat printr-un segment de elementul care cere serviciile** din spatele interfetei



Interfetele Java (un fel de clase complet abstracta)

Codul Java echivalent

```
1    public interface UpdatePrices {
2       public void update();  // metoda neimplementata
3    }                         // (semnatura, CONTRACT)
```

```
1    public class ManualPriceEntry implements UpdatePrices {
2       public void update() {
3          // implementare metoda = concretizare CONTRACT
4       }
5    }
```

```
1    public class QuoteQuery implements UpdatePrices {
2       public void update() {
3          // implementare metoda = concretizare CONTRACT
4       }
5    }
```

Interfetele Java (un fel de clase complet abstracta)

Codul Java echivalent

```
1   public class FinancialPlanner {
2
3       private UpdatePrices[] updaters = new UpdatePrices[2];
4
5       public void someMethodOfFinancialPlanner() {
6
7           // stocare uniforma
8           updaters[0] = new QuoteQuery();
9           updaters[1] = new ManualPriceEntry();
10
11          // apelare polimorfa uniforma
12          for (int i=0; i<updaters.length; i++) {
13              updaters[i].update();
14          }
15      }
16  }
```

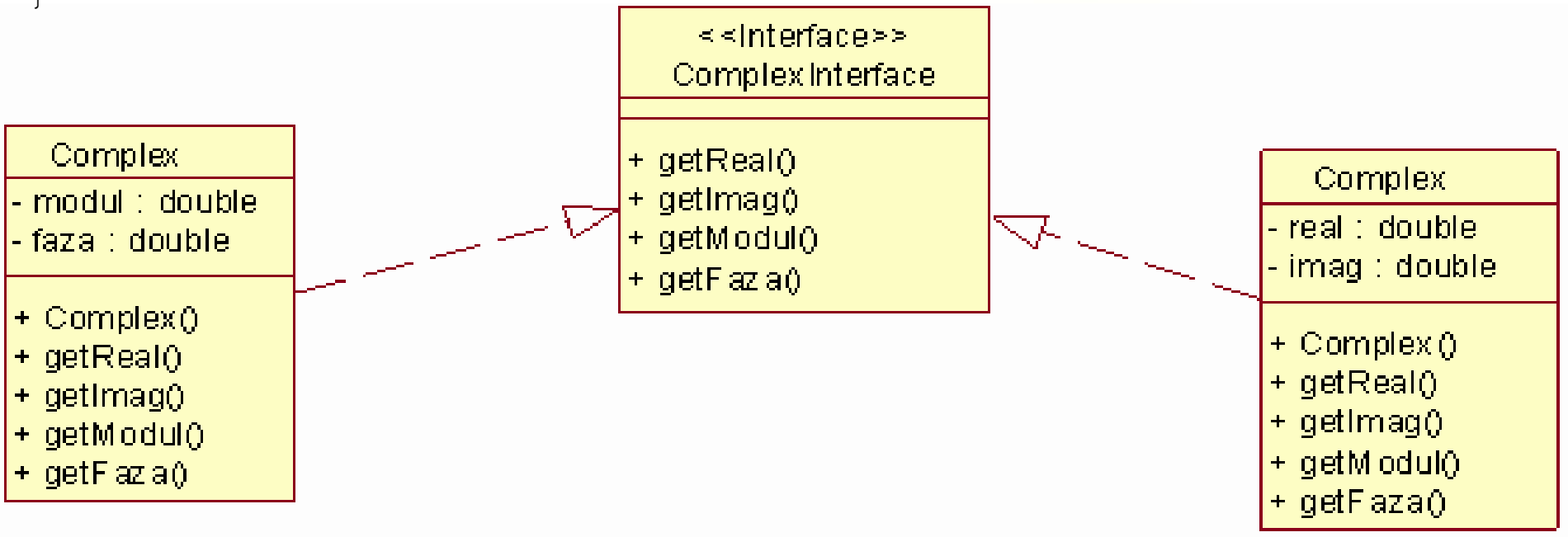
Interfetele Java (un fel de clase complet abstracta)

Codul Java echivalent

```
1   public class MutualFundAnalyser {
2
3       private UpdatePrices[] updaters = new UpdatePrices[2];
4
5       public void someMethodOfMutualFundAnalyser() {
6
7           // stocare uniforma
8           updaters[0] = new QuoteQuery();
9           updaters[1] = new ManualPriceEntry();
10
11          // apelare polimorfa uniforma
12          for (int i=0; i<updaters.length; i++) {
13              updaters[i].update();
14          }
15      }
16  }
```

Exemplu de interfata Java (un fel de clasa complet abstracta)

```
// Interfata (colectie de metode neimplementate)
// care reprezinta un contract privind interfata publica
public interface ComplexInterface {
  // Metode abstracte, neimplementate
  public abstract double getReal();
  public abstract double getImag();
  public abstract double getModul();
  public abstract double getFaza();
}
```



Exemplu de clasa care implementeaza o interfata Java

```
// Clasa concreta, din care pot fi create direct obiecte, care
// implementeaza (concretizeaza) interfata ComplexInterface
public class Complex implements ComplexInterface {
  // Atribute private (ascunse, inaccesibile din exteriorul clasei)
  private double real;
  private double imag;
  // Constructori (cu nume supraincarcat) si operatii (metode)
  public void Complex(float real, float imag) {
      this.real = real;
      this.imag = imag;
  }
  public void Complex(double modul, double faza) {
      this.real = modul * Math.cos(faza);
      this.imag = modul * Math.sin(faza);
  }
  public double getReal() {        return this.real;     }
  public double getImag() {        return this.imag;    }
  public double getModul() {
        return Math.sqrt(this.real*this.real + this.imag*this.imag);
  }
  public double getFaza() {
        return Math.atan2(this.real, this.imag);
  }
}
```

Exemplu de clasa care implementeaza o interfata Java

```
// Clasa concreta, din care pot fi create direct obiecte, care
// implementeaza (concretizeaza) interfata ComplexInterface
public class Complex implements ComplexInterface {
  // Atribute private (ascunse, inaccesibile din exteriorul clasei)
  private double modul;
  private double faza;
  // Constructori (cu nume supraincarcat) si operatii (metode)
  public void Complex(float real, float imag) {
      this.modul = Math.sqrt(real*real + imag*imag);
      this.faza = Math.atan2(real, imag);
  }
  public void Complex(double modul, double faza) {
      this.modul = modul;
      this.faza = modul;
  }
  public double getReal() {
        return this.modul*Math.cos(this.faza);
  }
  public double getImag() {
        return this.modul*Math.sin(this.faza);
  }
  public double getModul() {       return this.modul;     }
  public double getFaza() {        return this.faza;     }
}
```

Exemplu de interfata Java si de clasa care o implementeaza

```
public interface StackInterface {
  boolean empty();
  void push( Object x);
  Object pop() throws EmptyStackException;
  Object peek() throws EmptyStackException;
}


public class Stack implements StackInterface {
  private Vector v = new Vector(); // utilizeaza clasa java.util.Vector

  public void push(Object item) { v.addElement(item); }

  public Object pop() {
    Object obj = peek();
    v.removeElementAt(v.size() - 1);
    return obj;
  }

  public Object peek() throws EmptyStackException {
    if (v.size() == 0) throw new EmptyStackException();
    return v.elementAt(v.size() - 1);
  }

  public boolean empty() { return v.size() == 0; }
}
```

Clasa care **extinde prin agregare**

Exemplu de interfata Java si de clasa care o implementeaza

```
public interface StackInterface {
  boolean empty();
  void push( Object x);
  Object pop() throws EmptyStackException;
  Object peek() throws EmptyStackException;
}

public class Stack extends Vector implements StackInterface {
  public Object push(Object item) { addElement(item); return item; }
  public Object pop() {
    Object obj;
    int len = size();
    obj = peek();
    removeElementAt( len - 1);
    return obj;
  }
  public Object peek() {
    int len = size();
    if (len == 0) throw new EmptyStackException();
    return elementAt( len - 1);
  }
  public boolean empty() { return size() == 0;}
}
```

Subclasa care **extinde prin mostenire**

Exemplu de clasa care implementeaza o interfata Java

In **cazul subclasei care extinde prin mostenire**

- toate **metodele publice** ale clasei **Vector**
  - printre care si metode de **inserare**
  - **pot fi invocate pentru** obiectele clasei **stiva**
- astfel, **pe langa comportamentul tipic stivei**
  - **utilizatorii pot declansa comportamente atipice**
    - invocand **metode** de **inserare**, etc.
    - care **incalca principiul de functionare**

De exemplu:

```
Vector v = new Stack();   // cod legal - referinta la clasa de baza
                          // poate fi initializata cu obiect din subclasa

v.insertElementAt(x, 2); // cod legal - dar inserarea unui obiect in stiva
                         // incalca principiul de functionare al stivei
```