Interfeţe

O interfaţa este o listă de metode care trebuie definite de către orice clasa care implementează interfaţa respectivă. O interfaţa poate defini şi constante (public static final).

Seamănă cu o clasa abstractă. O interfaţa este asemănătoare cu o clasă abstractă care nu are variabile instanţă şi statice (constantele static final sunt permise), şi fără corpuri pentru metode. Aceasta este în esenţa ceea ce face o clasă complet abstractă, dar clasele abstracte permit definiţii de metode statice, pe când interfeţele nu permit acest lucru.

Obligaţie contractuală. Atunci când o clasă specifica faptul că ea implementează o interfaţă, clasa trebuie să definească toate metodele respectivei interfeţe. O clasă poate implementa mai multe interfeţe deiferite. Daca o clasă nu defineşte toate metodele interfeţelor pe care a fost de acord să le definească (prin clauza *implement*), compilatorul dă un mesaj de eroare, care tipic spune ceva de genul "This class must be declared abstract". (O clasă abstractă este una care nu implementează toate metodele pe care le-a declarat.) Soluţia pentru eroare este aproape întotdeauna să se implementeze metodele lipsă din interfaţă. Un nume de metodă scris greşit sau o lista incorecta de parametri este cauza uzuală, nu faptul că respectiva clasă ar fi trebuit să fie abstractă!

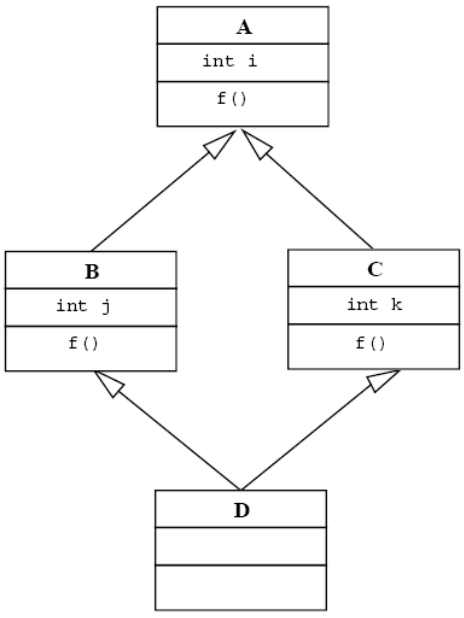
O utilizare foarte frecventă a interfeţelor este pentru ascultători (listeners). Un ascultător (listener) este un obiect dintr-o clasă care implementează metodele cerute pentru interfaţa respectivă. Puteţi crea ascultători interiori (inner) anonimi sau puteţi implementa interfaţa ceruta în oricare clasă.

Interfeţele sunt de asemenea folosite extensiv în pachetul de structuri de date (Java Collections).

2.1. Clasele în comparaţie cu interfeţele

Clasele sunt folosite pentru a reprezenta ceva care are **atribute** (variabile, câmpuri) **şi capabilităţi/ responsabilităţi** (metode, funcţii).

Interfeţele descriu doar capabilităţi. Spre exemplu, suntem oameni pentru că avem atributele unui om (clasă). Cineva este instalator pentru ca are abilitatea unui instalator (interfaţă). Cineva poate fi şi electrician (interfaţă). Puteţi implementa multe interfeţe, dar obiectele pot fi de o singură clasă.



Această analogie nu merge, totuşi, într-un anume sens. Capabilităţile (metodele) unei clase nu se schimbă (dacă o clasă implementează o interfaţă, ea este implementată pentru toate instanţele), pe când îndemânările omeneşti despre care vorbeam sunt dinamice şi pot fi învăţate sau uitate. Analogia are un defect, cum sunt toate analogiile de altfel, dar oferă o idee despre cum se poate face o distincţie între clase şi interfeţe.

Interfeţele înlocuiesc moştenirea multiplă

O clasa din C++ poate avea mai mult de o clasa părinte. Aceasta se numeşte moştenire multiplă. Gestionarea definiţiilor variabilelor instanţă în cazul moştenirii multiple poate deveni destul de încurcată şi duce la mai multe probleme – d.e., aşa numitul "Diamant ucigător al morţii" ("Deadly Diamond of Death") – decât soluţii. Figura din

stânga constituie o ilustrare a problemei menţionate. În figură sunt prezentate patru clase aranjate într-o structura care creează nevoia moştenirii virtuale. Atât clasa B cât şi clasa C moştenesc din clasa A. D moşteneşte multiplu atât din B cât şi din C. De aici apar doua probleme. Prima: ce implementare a metodei 'f' să moştenească D? Ar trebui sa moştenească f() din B sau f() din C? În C++ răspunsul se dovedeşte a fi nici una. Trebuie declarat f() în D şi trebuie implementat. Aceasta elimina ambiguitatea şi cu siguranţa aceasta regulă simplă ar fi putut fi adoptată în Java.
Totuşi, cea de a doua problemă este un pic mai complicată. Clasa A are o variabila membru numită i. Amândouă clasele B şi C moştenesc aceasta variabila membru. Cum D o moşteneşte de la amândouă, avem de-a face cu o ambiguitate. Pe de o parte am dori ca i din B şi i din C să fie variabile separate în D, creând astfel două copii ale lui A în D. Pe de alta parte am dori să existe o singură copie a lui A în D astfel ca numai un i din A să existe în D.
Din acest motiv proiectanţii limbajului Java au ales să permită doar o singura clasă părinte, dar multiple interfeţe. Aceasta furnizează majoritatea funcţionalităţii moştenirii multiple, dar fără dificultăţile pe care le incumbă.

2.2. Implementarea unei interfeţe

Se pot implementa oricâte interfeţe într-o clasă; cele pe care doriţi să le implementaţi le separaţi în clauza implements cu virgule. Spre exemplu,

```
// Nota:
// ActionListener are nevoie sa fie definita metoda actionPerformed(..)
// MouseMotionListener are nevoie de definirea lui mouseMoved(..) si mouseDragged(..).

public class MyPanel extends JPanel implements ActionListener, MouseMotionListener {
    public void actionPerformed(ActionEvent e) {
        /* Corpul metodei */
    }
    public void mouseDragged(MouseEvent me) {
        /* Corpul metodei */
    }
    public void mouseMoved(MouseEvent me) {
        /* Corpul metodei */
    }
    // Tot ce mai este in aceasta clasa.
}
```

Este obişnuit ca un panou (panel) care foloseşte grafica şi răspunde la mouse să-şi implementeze ascultători pentru mouse proprii (dar nu ascultători pentru acţiuni – action listeners) ca mai sus.

2.3. Declararea unei interfeţe

Pentru programe simple este mai probabil să folosiţi o interfaţă decât să o definiţi. Iată cam ce conţine definiţia interfeţei **java.awt.event.ActionListener**.

```
public interface ActionListener
{
      public void actionPerformed(ActionEvent e);
}
```

2.4. Interfeţe imbricate

Interfeţele pot fi imbricate n clase şi în alte interfeţe. Acest lucru relevă o serie de caracteristici foarte interesante:

```
// InterfeteImbricate.java

class A
{
  interface B
  {
    void f();
  }
  public class BImp implements B
  {
    public void f() {}
  }
  private class BImp2 implements B
  {
    public void f() {}
  }
  public interface C
  {
    void f();
  }
  class CImp implements C
  {
    public void f() {}
  }
  private class CImp2 implements C
  {
    public void f() {}
  }
  private interface D
  {
    void f();
  }
  private class DImp implements D
  {
    public void f() {}
  }
  public class DImp2 implements D
  {
    public void f() {}
  }
  public D getD() { return new DImp2(); }
  private D dRef;
  public void receiveD(D d)
  {
    dRef = d;
    dRef.f();
  }
}

interface E
{
  interface G
  {
    void f();
  }
  // Redundant "public":
  public interface H
  {
    void f();
  }
  void g();
```

```
  // Nu poate fi privata intr- interfata:
  //! private interface I {}
}

public class NestingInterfaces
{
  public class BImp implements A.B
  {
    public void f() {}
  }
  class CImp implements A.C
  {
    public void f() {}
  }
  // Nu poate implementa o interfata private cu exceptia
  // clasei care defineste interfata respectiva:
  //! class DImp implements A.D {
  //!  public void f() {}
  //! }
  class EImp implements E
  {
    public void g() {}
  }
  class EGImp implements E.G
  {
    public void f() {}
  }
  class EImp2 implements E
  {
    public void g() {}
    class EG implements E.G
    {
      public void f() {}
    }
  }
  public static void main(String[] args)
  {
    A a = new A();
    // Nu poate accesa A.D:
    //! A.D ad = a.getD();
    // Nu retruneaza nimic, doar A.D:
    //! A.DImp2 di2 = a.getD();
    // Nu poate accesa un mebru al interfetei:
    //! a.getD().f();
    // Doar un lat A poate face ceva cu getD():
    A a2 = new A();
    a2.receiveD(a.getD());
  }
}
```

Sintaxa folosită la imbricarea unei interfeţe într-o clasă este destul de evidentă. Ca şi pentru interfeţele neimbricate, vizibilitatea poate fi publică sau la nivelul pachetului. De asemenea se poate vedea că atât interfeţele imbricate publice cât şi cele cu acces la nivelul pachetului pot fi implementate sub formă de clase imbricate publice, cu acces în pachet sau privat.
Ca un lucru nou, interfeţele pot fi şi ele private, cum se vede în A.D (aceeaşi sintaxă cu calificare se foloseşte şi pentru interfeţe şi pentru clase imbricate). La ce foloseşte of interfaţă privată imbricată? Se poate ghici că ea poate fi implementată doar ca clasă internă privată în DImp, dar A.DImp2 arată că ea poate fi implementată şi ca o clasă publică.

Totuşi, A.DImp2 poate fi folosită doar ca sine însăşi. Nu este permis să se menţioneze faptul că ea implementează interfaţa privată, aşa că **implementarea unei interfeţe private este o modalitate de forţare a definiţiei metodelor din interfaţă fără a adăuga nici o informaţie de tip** (adică fără a permite nici o conversie de tip în sus pe ierarhie – "upcasting").
Metoda getD( ) produce o nouă ciudăţenie în ceea ce priveşte interfaţa privată: O metodă publică care returnează o referinţă spre o interfaţă privată. Ce se poate face cu valoarea returnată de această metodă? În main( ), se pot vedea câteva încercări de folosire a valorii returnate, şi toate eşuează. Singurul lucru care funcţionează este dacă valoarea returnată este pasată unui obiect care are permisiunea de a o folosi— în acest caz un alt A, via metoda receiveD( ).
Interfaţa E arată că interfeţele se pot imbrica una în alta. Totuşi, regulile referitoare la interfeţe — în particular, regula că **toate elementele interfeţei trebuie să fie publice** — sunt impuse cu stricteţe, aşa că o interfaţă imbricată într-o alta este în mod automat publică şi nu poate fi făcută privată. Interfeţele imbricate arată diferitele căi în care pot fi implementate interfeţele imbricate. În particular, observaţi că atunci când implementaţi o interfaţă, nu se cere a se implementa nici o interfaţă care este imbricată în ea. De asemenea, interfeţele private nu pot fi implementate în afara claselor care le definesc.

2.5. Câteva interfeţe Java predefinite

2.5.1. Interfaţa Clonable

Interfaţa **Cloneable** este un exemplu de interfaţa Java predefinită:

- Nu conţine antete de metodă sau constante
- Se foloseşte pentru a indica în ce mod trebuie folosită şi redefinită metoda **clone** (moştenită din clasa **Object**)
- Metoda **Object.clone()** face o copie bit-cu-bit a datelor obiectului
- Dacă toate datele sunt tipuri primitive sau clase care nu se schimbă (cum este **String**), atunci acest lucru este adecvat.
- Dacă datele din obiectul de clonat include variabile instanţă al căror tip este mutabil, atunci simpla implementare a lui clone poate cauza o *scurgere de informaţii private* (*privacy leak*)
- La implementarea interfeţei **Cloneable** pentru o clasă procedaţi astfel:
  - Invocaţi mai întâi metoda **clone** a obiectului din clasa de bază **Object** (sau ce altă clasă va fi fiind)
  - Apoi resetaţi valorile pentru orice alte variabile instanţă noi ale căror valori sunt tipuri de clase mutabile. Aceasta se face prin copierea variabilelor instanţa apelând metodele clone ale lor *proprii*
  - Observaţi că aceasta va funcţiona corect doar dacă interfaţa **Cloneable** este implementată corespunzător pentru clasele de care aparţin variabilele instanţă respective şi pentru clasele de care aparţin oricare dintre variabilele instanţa din clasele menţionate şi aşa mai departe

2.5.2. Interfaţa Comparable

- Interfaţa **Comparable** este definita in pachetul **java.lang**, fiind automat disponibilă fiecărui program. Nu are decât următoarea metoda care trebuie implementată:
  public int compareTo(Object other);
- Este responsabilitatea programatorului să urmeze semantica interfeţei **Comparable** atunci când o implementează. Metoda **compareTo** trebuie să returneze
  - Un număr negativ atunci când un obiect "este înainte de" parametrul **other**
  - Zero atunci când obiectul apelant "este egal cu" parametrul **other**
  - Un număr pozitiv dacă obiectul apelant "urmează după" parametrul **other**
- Dacă parametrul **other** nu este de acelaşi tip cu clasa în curs de definire, atunci trebuie aruncată o excepţie de tipul **ClassCastException**
- Aproape orice noţiune rezonabilă de "este înainte de" este acceptabilă

  - In particular, toate relaţiile standard mai-mic-decât peste numere şi ordonările lexicografice peste şirurile de caractere sunt potrivite
- Relaţie "urmează după" este doar inversul lui "este înainte de"
- Pot fi luate în considerare alte ordonări, atâta vreme cât sunt relaţii de *ordine totală.* O ordine totală trebuie să satisfacă următoarele condiţii:
  - (*Antireflexivitate*) Pentru nici un obiect **o, o** nu este înaintea lui **o**
  - (*Trihotomi-antisimetrie*) Pentru oricare două obiecte **o1** şi **o2**, una şi numai una dintre următoarele este adevărată: **o1** este înainte de **o2**, **o1** urmează după **o2**, sau **o1** este egal cu **o2**
  - (*Tranzitivitate*) Dacă **o1** este înainte de **o2** şi **o2** este înainte de **o3**, atunci **o1** este înainte de **o3**
- Semantica lui "equals" din metoda **compareTo** ar trebui să coincidă cu cea a metodei **equals** dacă se poate, dar aceasta nu este absolut necesar
- Observaţie: atât clasa **Double** cât şi clasa **String** implementeză interfaţa **Comparable**
  - Interfeţele se aplică doar claselor
  - Un tip primitiv (d.e., **double**) nu poate implementa o interfaţă

2.5.3. Interfaţa Enumeration

Un obiect care implementează interfaţa **Enumeration** generează o serie de elemente, câte unul o dată, adică reprezintă un *iterator*

- Apelurile succesive la metoda **nextElement()**returnează elementele succesive ale seriei
- **hasMoreElements()** testează dacă enumerarea respectivă mai conţine elemente.
- Spre exemplu, pentru a tipări toate elementele unui **vector** *v*:

```
for (Enumeration e=v.elements(); e.hasMoreElements() ;)
{
    System.out.println(e.nextElement());
}
```

2.5.4. Un exemplu pentru Coneable şi Comparable

Cf. exemplul Employee prezentat la curs.

```
abstract class Employee implements Comparable
 {
   // instance variables
  private String name;
  public Employee(String name)
  {
     this.name = name;
  }
  public String getName()
  {
     return name;
  }
  public int compareTo(Object other)
  {
     Employee e = (Employee) other;
    return name.compareTo(e.name);
  }

  public abstract double calculatePay();
}
```

```
abstract class Employee implements Cloneable, Comparable
 {
   // instance variables
   private String name;
   public Employee(String name)
  {
     this.name = name;
  }
  public String getName()
  {
     return name;
  }
  public int compareTo(Object other)
  {
     Employee e = (Employee) other;
     return name.compareTo(e.name);
  }
  public Object clone()
  {
     Cloneable theClone = new Employee(this.name);
     // Initializeaza theClone. Aici nu e cazul, singura variabila instanta fiind imutabila
     return theClone;
   }
  public abstract double calculatePay();
}
```

3. Mersul lucrării

3.1. Studiaţi materialul şi exemplele furnizate.

3.2. Creaţi o interfaţă care să conţină trei metode într-un pachet propriu. Implementaţi interfaţa într-un alt pachet.

3.3. Pentru codul următor, Sandwich.java, creaţi o interfaţă numită FastFood (cu metodele corespunzătoare) şi modificaţi Sandwich ca să implementeze FastFood.

```
// Sandwich.java
class Meal
{
  Meal() { System.out.println("Meal()"); }
}
class Bread
{
  Bread() { System.out.println("Bread()"); }
}
class Cheese
{
  Cheese() { System.out.println("Cheese()"); }
}
class Lettuce
{
  Lettuce() { System.out.println("Lettuce()"); }
}
```

```
class Lunch extends Meal
{
  Lunch() { System.out.println("Lunch()"); }
}
class PortableLunch extends Lunch
{
  PortableLunch() { System.out.println("PortableLunch()");}
}
public class Sandwich extends PortableLunch
{
  private Bread b = new Bread();
  private Cheese c = new Cheese();
  private Lettuce l = new Lettuce();
  public Sandwich()
  {
    System.out.println("Sandwich()");
  }
  public static void main(String[] args)
  {
    new Sandwich();
  }
}
```

3.4. Creaţi trei interfeţe, fiecare cu două metode. Moşteniţi o interfaţă noua din cele trei adăugând o nouă metodă. Creaţi o clasă care să implementeze noua interfaţă şi să şi moştenească din-tr-o clasă concretă. Scrieţi apoi patru metode, fiecare dintre ele să primească una dintre cele patru interfeţe ca argument. În **main( ),** creaţi un obiect din clasa Dvs. Şi transmiteţi-l fiecăreia dintre cele trei metode.