Clase abstracte si interfete

De la POO

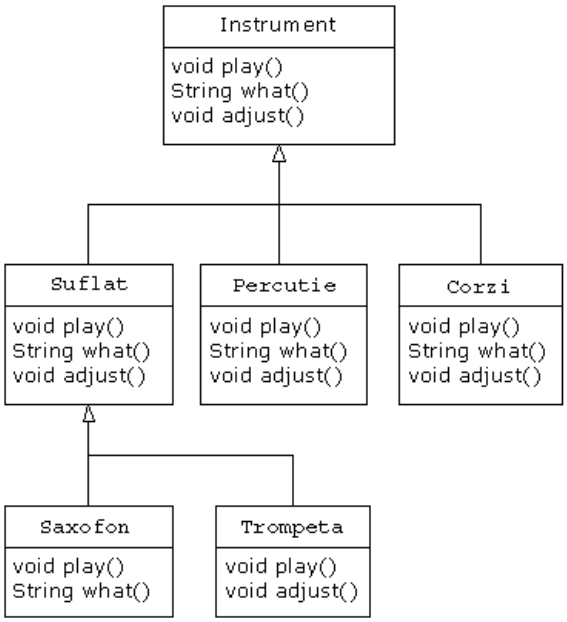
Salt la: Navigare, căutare

Cuprins

[ascunde] [ascunde]

Introducere

Fie urmatorul exemplu (Thinking in Java) care propune o ierarhie de clase pentru a descrie o suita de instrumente muzicale, cu scopul demonstrarii polimorfismului:



Clasa `Instrument` nu este instantiata niciodata pentru ca scopul sau este de a stabili o interfata comuna pentru toate clasele derivate. In acelasi sens, metodele clasei de baza nu vor fi apelate niciodata. Apelarea lor este ceva gresit din punct de vedere conceptual.

Clase abstracte

Dorim sa stabilim interfata comuna pentru a putea crea functionalitate diferita pentru fiecare subtip si pentru a sti ce anume au clasele derivate in comun.

O clasa cu caracteristicile enumerate mai sus se numeste **abstracta**. Cream o clasa abstracta atunci cand dorim sa:

- manipulam un set de clase printr-o **interfata comuna**
- **reutilizam** o serie metode si membrii din aceasta clasa in clasele derivate.

Metodele suprascrise in clasele derivate vor fi apelate folosind ***dynamic binding*** (*late binding*). Acesta este un mecanism prin care compilatorul, in momentul in care nu poate determina implementarea unei metode in avans, lasa la latitudinea JVM-ului (masinii virtuale) alegerea implementarii potrivite, in functie de tipul real al obiectului. Acesta legare a implementarii de numele metodei la **momentul executiei** sta la baza polimorfismului.

Nu exista instante ale unei clase abstracte, aceasta exprimand doar un punct de pornire pentru definirea unor instrumente reale. De aceea, crearea unui obiect al unei clase abstracte este o eroare, compilatorul Java semnaland eroare in acest caz.

Metode abstracte

Pentru a exprima faptul ca o metoda este abstracta (adica se declara doar interfata ei, nu si implementarea), Java foloseste cuvantul cheie `abstract`:

```
abstract void f();
```

O clasa care contine **metode abstracte** este numita **clasa abstracta**. Daca o clasa are una sau mai multe metode abstracte atunci ea trebuie sa contina in definite cuvantul `abstract`.

Exemplu:

```
abstract class Instrument {
    ...
}
```

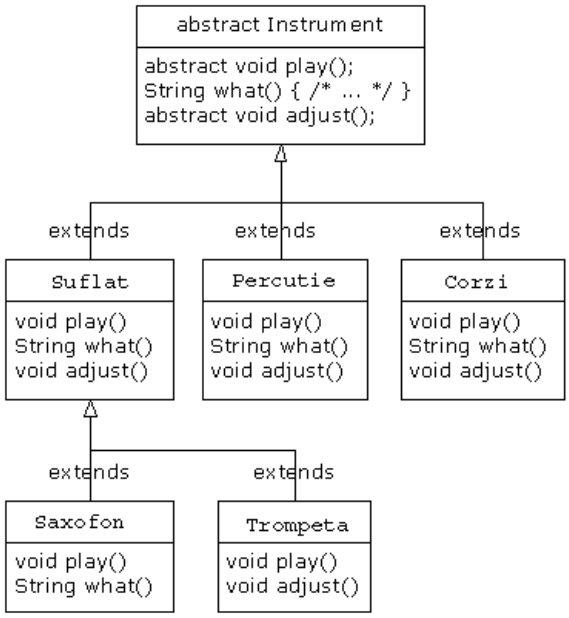
Deoarece o clasa abstracta este incompleta (exista metode care nu sunt definite), crearea unui obiect de tipul clasei este impiedicata de compilator.

Clase abstracte in contextul mostenirii

O clasa care mosteneste o clasa abstracta este ea insasi abstracta daca nu implementeaza **toate** metodele abstracte ale clasei de baza. Putem defini clase abstracte care mostenesc alte clase abstracte si tot asa. O clasa care poate fi instantiata (nu este abstracta) si care mosteneste o clasa abstracta trebuie sa implementeze (defineasca) toate metodele abstracte pe lantul mostenirii (ale tuturor claselor abstracte care ii sunt "parinti").

Este posibil sa declaram o **clasa abstracta fara** ca ea sa aiba **metode abstracte**. Acest lucru este folositor cand declaram o clasa pentru care nu dorim instante (nu este corect conceptual sa avem obiecte de tipul acelei clase, chiar daca definita ei este completa).

Iata cum putem sa modificam exemplul instrument cu metode abstracte:



Interfete

Interfetele duc conceptul **abstract** un pas mai departe. Se poate considera ca o interfata este o **clasa abstracta pura**: permite programatorului sa stabileasca o "forma" pentru o clasa: numele metodelor, lista de argumente, valori intoarse, dar fara **nicio implementare**. O interfata poate contine **campuri** dar acestea sunt in mod implicit **`static`** si **`final`**.

Interfata este folosita pentru a descrie un protocol intre clase: o clasa care implementeaza o interfata va implementa metodele definite in interfata. Astfel orice cod care foloseste o anumita interfata stie ce metode pot fi apelate pentru acea interfata.

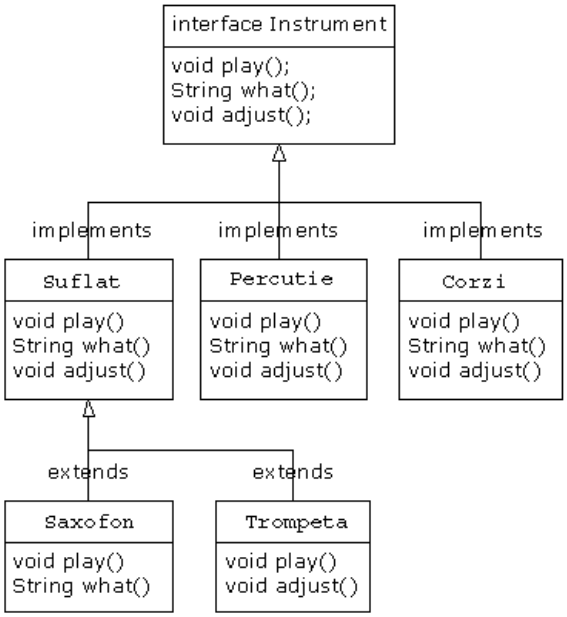
Pentru a crea o interfata folosim cuvantul cheie **`interface`** in loc de `class`. La fel ca in cazul claselor, interfata poate fi declarata `public` doar daca este definita intr-un fisier cu acelasi nume ca si interfata. Daca o interfata nu este declarata `public` atunci specifcatorul ei de acces este `package-private`.

Pentru a defini o **clasa** care este conforma cu o interfata anume folosim cuvantul cheie **`implements`**. Aceasta relatie este asemanatoare cu mostenirea, cu diferenta ca nu se mosteneste comportament, ci doar "interfata".

Pentru a defini o **interfata** care mosteneste alta interfata folosim cuvantul cheie **`extends`**.

Dupa ce o interfata a fost implementata, acea implementare devine o clasa obisnuita care poate fi extinsa prin mostenire.

Iata exemplul dat mai sus, modificat pentru a folosi interfete:



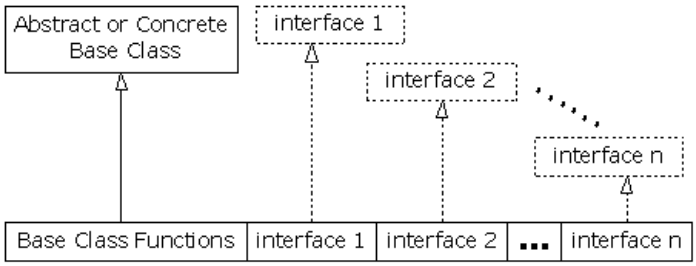
Codul arata astfel:

```
interface Instrument {

    // Compile-time constant:
    int i = 5; // static & final

    // Cannot have method definitions:
    void play(); // Automatically public
    String what();
    void adjust();
}

class Suflat implements Instrument {

    public void play() {
        System.out.println("Suflat.play()");
    }

    public String what() {
        return "Suflat";
    }

    public void adjust() {}
}

class Trompeta extends Suflat {

    public void play() {
        System.out.println("Trompeta.play()");
    }

    public void adjust() {
        System.out.println("Trompeta.adjust()");
    }
}
```

Implicit, specificatorul de acces pentru membrii unei interfete este **`public`**. Atunci cand implementam o interfata trebuie sa specificam ca functiile sunt `public` chiar daca in interfata ele nu au fost specificate explicit astfel. Acest lucru este necesar deoarece specificatorul de acces implicit in clase este `package-private`, care este **mai restrictiv** decat `public`.

Mostenire multipla in Java

Interfata nu este doar o forma "pura" a unei clase abstracte, ci are un scop mult mai inalt. Deoarece o interfata nu specifica nici un fel de implementare (nu exista nici un fel de spatiu de stocare pentru o interfata) este normal sa "combinam" mai multe interfete. Acest lucru este folositor atunci cand dorim sa afirmam ca "X este un A, un B si un C". Acest deziderat este mostenirea multipla si, datorita faptului ca o singura entitate (A, B sau C) are implementare, nu apar problemele mostenirii multiple din C++.




```
interface CanFight {
    void fight();
}

interface CanSwim {
    void swim();
}

interface CanFly {
    void fly();
}

class ActionCharacter {
    public void fight() {}
}

class Hero extends ActionCharacter implements CanFight, CanSwim, CanFly {
    public void swim() {}

    public void fly() {}
}

public class Adventure {
    static void t(CanFight x) {
        x.fight();
    }

    static void u(CanSwim x) {
        x.swim();
    }

    static void v(CanFly x) {
        x.fly();
    }

    static void w(ActionCharacter x) {
        x.fight();
    }

    public static void main(String[] args) {

        Hero h = new Hero();

        t(h); // Treat it as a CanFight

        u(h); // Treat it as a CanSwim

        v(h); // Treat it as a CanFly

        w(h); // Treat it as an ActionCharacter
    }
}
```

Se observa ca `Hero` combina clasa `ActionCharacter` cu interfetele `CanSwim` etc. Acest lucru se realizeaza specificand prima data clasa concreta (sau abstracta) (`extends`) si abia apoi `implements`.

Metodele clasei `Adventure` au drept parametri interfetele `CanSwin` etc. si clasa `ActionCharacter`. La fiecare apel de metoda din `Adventure` se face ***upcast*** de la obiectul `Hero` la clasa sau interfata dorita de metoda respectiva.

Coliziuni de nume la combinarea interfetelor

Combinarea unor interfete care contin o metoda cu acelasi nume este posibila doar daca metodele nu au tipuri intoarse diferite si aceeasi lista de argumente. Totusi este preferabil ca in interfete diferite care trebuie combinate sa nu existe metode cu acelasi nume deoarece acest lucru poate duce la confuzii evidente (sunt amestecate in acest mod 3 concepte: *overloading*, *overriding* si *implementation*).

Extinderea interfetelor

Se pot adauga cu usurinta metode noi unei interfete prin extinderea ei intr-o alta interfata:

Exemplu:

```
interface Monster {
    void menace();
}

interface DangerousMonster extends Monster {
    void destroy();
}
```

Deoarece campurile unei interfete sunt automat **static** si **final**, interfetele sunt un mod convenabil de a crea grupuri de constante, asemanatoare cu *enum*-urile din C , C++.

Initializarea campurilor in interfete

In interfete **nu** pot exista **blank finals** (campuri `final` neinitializate) insa pot fi initializate cu **valori neconstante**. Campurile fiind statice, ele vor fi initializate prima oara cand clasa este initializata.

Exercitii

1. (**2p**) Implementati interfata `Task` in cele 3 moduri de mai jos. La crearea claselor in Eclipse, utilizati interfata vizuala pentru precizarea interfetelor ce se doresc implementate (metodele corespunzatoare vor fi adaugate automat).
   - Un task care sa afiseze un mesaj la output. Mesajul este specificat in constructor.
   - Un task care genereaza un numar aleator si afiseaza un mesaj cu numarul generat la output. Generarea se face in constructor
   - Un task care incrementeaza un contor global si afiseaza valoarea contorului dupa fiecare incremenetare.

     Nota: Acesta este un exemplu simplu pentru ***Command Pattern***.

2. (**3p**) Interfata `Container` specifica interfata comuna pentru colectii de obiecte `Task`, in care se pot adauga si din care se pot elimina elemente. Creati doua tipuri de containere care implementeaza aceasta clasa:
   1. (**1.5p**) `Stack` - care implementeaza o strategie de tip ***LIFO***
   2. (**1.5p**) `Queue` - care implementeaza o strategie de tip ***FIFO***

   Evitati codul similar in locuri diferite!
   Hint: Puteti retine intern colectia de obiecte, utilizand clasa `ArrayList` din biblioteca Java, ce va fi folosita asemanator clasei `MyArrayList` din Laboratorul 2. Iata un exemplu de initializare pentru siruri:
   ```
   ArrayList<String> list = new ArrayList<String>();
   ```

3. (**1p**) Implementati interfata `IFactory` care contine o metoda ce primeste ca parametru o strategie - `Strategy` si care intoarce un container asociat acelei strategii. Din acest punct inainte, in programul vostru veti crea containere folosind doar aceasta clasa (nu puteti crea direct obiecte de tip `Stack` sau `Queue`). **Evitati** instantierea clasei `Factory` implementate de voi la fiecare creare a unui container!

   Nota: Acest mod de a crea obiecte de tip `Container` elimina problema care apare in momentul in care decidem sa folosim o implementare diferita pentru un anumit tip de strategie si nu vrem sa facem modificari si in restul programului. De asemenea o astfel de abordare este utila cand avem implementari care urmaresc scopuri diferite (putem avea un `Factory` care sa creeze containere optimizate pentru viteza sau un Factory cu containere ce folosesc minimum de memorie). Sablonul acesta de programare poarta denumirea de ***Factory Method Pattern***.

4. (**3p**) Creati clasa `AbstractTaskRunner` care executa Task-uri, ce va contine:
   - Un constructor ce primeste ca parametru o strategie prin care se specifica in ce ordine se vor executa task-urile (*LIFO* sau *FIFO*)
   - O metoda `addTask`, care adauga un task in colectia de task-uri de executat
   - O metoda `executeAll`, care executa toate task-urile din colectia de task-uri
   - Dupa executia fiecarui task va avea loc o actiune - actiune abstracta, specifica claselor care mostenesc `AbstractTaskRunner` (vezi ex. urmator). Metoda abstracta care specifica aceasta actiune trebuie sa fie **vizibila** doar pentru clasele din lantul de mostenire ce porneste din `AbstractTaskRunner`.
5. (**4p**) Extindeti clasa `AbstractTaskRunner` in 3 moduri:
   - (**1p**) `PrintTaskRunner` - care afiseaza un mesaj dupa executia unui task in care se specifica ora la care s-a executat task-ul (vezi clasa `Date`)
   - (**1p**) `CountTaskRunner` - incrementeaza un contor local care tine minte cate task-uri s-au executat.
   - (**1p**) `RedoTaskRunner` - salveaza fiecare task executat intr-un container in ordinea inversa a executiei si are o metoda prin care se permite reexecutarea task-urilor.
   - (**1p**) Extindeti `PrintTaskRunner` astfel incat sa se introduca o intarziere intre executiile task-urilor. Intarzierea este specificata in constructor. Pentru introducerea unei intarzieri in executia programului puteti utiliza secventa de mai jos:
     ```
     try {
         Thread.sleep(1000); // milisecunde
     } catch (InterruptedException e) {
         e.printStackTrace();
     ```

```
}
```

- **Solutii**

Adus de la "http://cursuri.cs.pub.ro/~poo/wiki/index.php/Clase_abstracte_si_interfete"