Programmazione Java

Interfacce: ereditarietà multipla, collisioni di nomi, raggruppare costanti, inizializzare i campi nelle interfacce

Davide Di Ruscio
Dipartimento di Informatica
Università degli Studi dell'Aquila

diruscio@di.univaq.it

Sommario

» Interfacce

- Ereditarietà multipla
- Collisioni di nomi
- Estendere un'interfaccia con l'ereditarietà
- Raggruppare costanti
- Inizializzare i campi nelle interfacce

Interfacce

- Permette di stabilire la *forma* per una classe
  - Nomi metodi, elenchi argomenti, e tipi restituiti
  - Non sono definiti i corpi dei metodi
- Può contenere campi ma sono soltanto `static` e `final` (le cui inizializzazioni sono obbligatorie)
- Può essere considerata una classe abstract pura, in quanto fornisce solo una forma, ma nessuna implementazione
- L'interfaccia è utilizzata per stabilire un "protocollo" fra classi

Interfacce

- Classe che implementa un'interfaccia si impegna ad implementare tutti i metodi definiti nell'interfaccia ovvero accetta di rispondere a determinati comportamenti
- Per dichiarare una interfaccia si utilizza la parola chiave `interface`
- Classe che si adatta ad una particolare interfaccia/e utilizza `implements`
- E' possibile dichiarare variabili di tipo interfaccia
- Non è possibile creare oggetti di tipo interfaccia

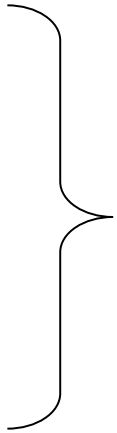
Interfacce


```
[ public ] interface <nome dell'interfaccia>
        [extends interfaccia1, interfaccia2, ..] {

[<dichiarazione di costanti>]

[<dichiarazione dei metodi>]

}
```



Corpo dell'interfaccia

Senza public l'interfaccia è visibile solo dalle classi definite nello stesso package.

Interfacce

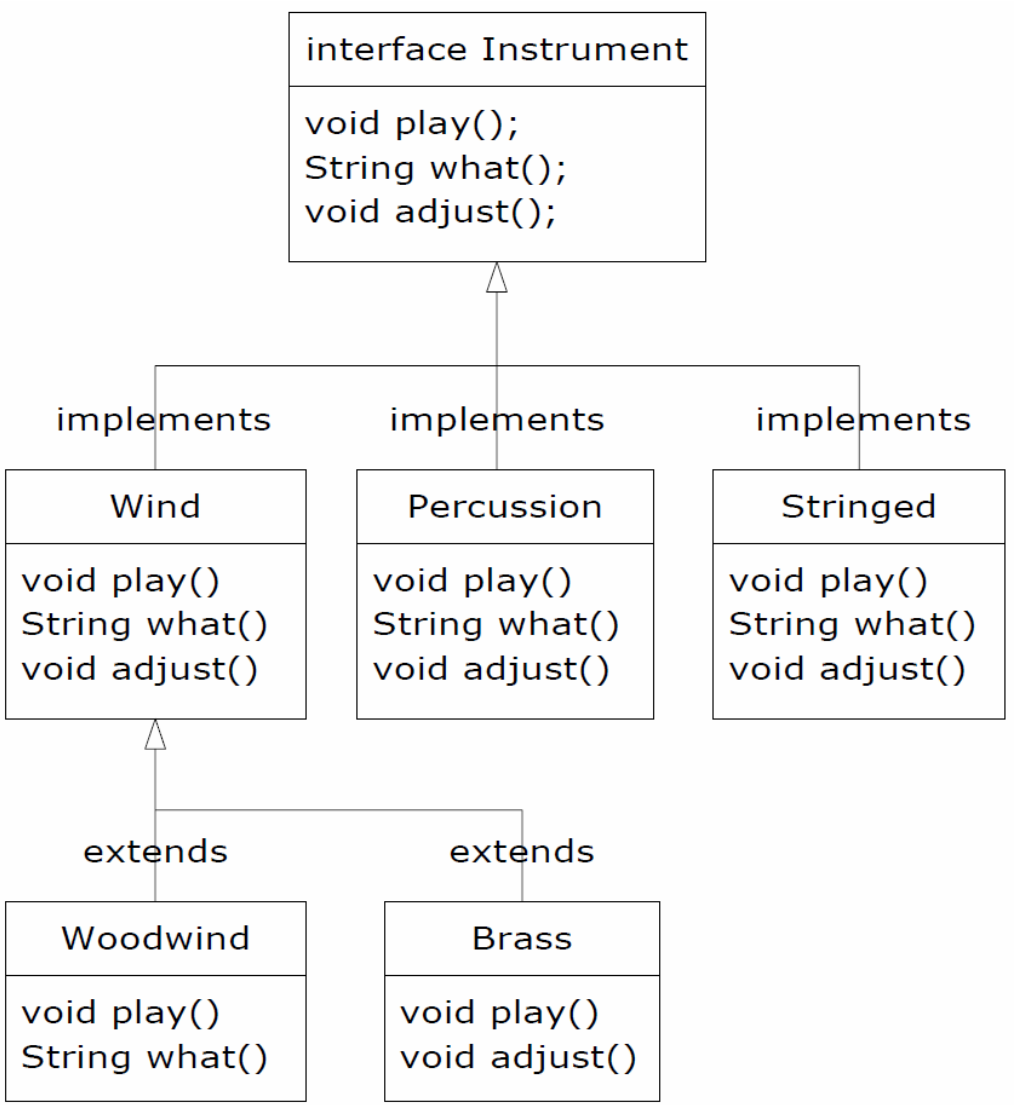
» La seguente dichiarazione

```
interface Cutable {
 void cut();
}
```

è equivalente alla seguente:

```
abstract class Cutable {
 abstract public void cut();
}
```

Interfacce > Esempio



Interfacce > Esempio


```
interface Instrument {
  //Costante
  int I = 5; // static & final
  void play(Note n); //public automatico
  String what();
  void adjust();
}

public class Note {
  private String noteName;
  private Note(String noteName) {
    this.noteName = noteName;
  }
  public String toString() { return noteName; }
  public static final Note
    MIDDLE_C = new Note("Middle C"),
    C_SHARP  = new Note("C Sharp"),
    B_FLAT   = new Note("B Flat");
    // Etc.
}
```

(Vedere Music5.java)

Interfacce > Esempio


```
class Wind implements Instrument {
  public void play(Note n) {
    System.out.println("Wind.play() " + n);
  }
  public String what() { return "Wind"; }
  public void adjust() {}
}

class Percussion implements Instrument {
  public void play(Note n) {
    System.out.println("Percussion.play() " + n);
  }
  public String what() { return "Percussion"; }
  public void adjust() {}
}
```

Interfacce > Esempio


```
class Stringed implements Instrument {
  public void play(Note n) {
    System.out.println("Stringed.play() " + n);
  }
  public String what() { return "Stringed"; }
  public void adjust() {}
}

class Brass extends Wind {
  public void play(Note n) {
    System.out.println("Brass.play() " + n);
  }
  public void adjust() {
    System.out.println("Brass.adjust()");
  }
}
class Woodwind extends Wind {
  public void play(Note n) {
    System.out.println("Woodwind.play() " + n);
  }
  public String what() { return "Woodwind"; }
}
```

Interfacce > Esempio


```
public class Music5 {
  static void tune(Instrument i) {
    i.play(Note.MIDDLE_C);
  }

  static void tuneAll(Instrument[] e) {
    for(int i = 0; i < e.length; i++)
      tune(e[i]);
  }
  public static void main(String[] args) {
    Instrument[] orchestra = {
      new Wind(),
      new Percussion(),
      new Stringed(),
      new Brass(),
      new Woodwind()
    };
    tuneAll(orchestra);
  }
}
```

Si fa l'upcasting come se `Instrument` fosse una classe "regolare"

Interfacce > Ereditarietà multipla

» E' possibile che una classe implementi diverse interfacce (simile *ereditarietà multipla*)

» Esempio

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

interface CanSwim {
  void swim();
}

interface CanFly {
  void fly();
}

class ActionCharacter {
  public void fight() {}
}
```

(Vedere Adventure.java)

Interfacce > Ereditarietà multipla


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

public class Adventure {
  public static void t(CanFight x) { x.fight(); }
  public static void u(CanSwim x) { x.swim(); }
  public static void v(CanFly x) { x.fly(); }
  public 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
  }
}
```

Ciascuna interfaccia diventa un tipo indipendente verso il quale poter operare l'upcast

Interfacce > Collisioni di nomi

- Se si implementano diverse interfacce è possibile avere problemi con i metodi dichiarati

- Esempio

```
interface I1 {
     void f();
}
interface I2 {
     int f(int i);
}
interface I3 {
     int f();
}
```

(Vedere InterfaceCollision.java)

Interfacce > Collisioni di nomi


```
class C {
    public int f() {
        return 1;
    }
}

class C2 implements I1, I2 {
  public void f() {}
  public int f(int i) { return 1; } // overloaded
}

class C3 extends C implements I2 {
  public int f(int i) { return 1; } // overloaded
}
```

Interfacce > Collisioni di nomi


```
class C4 extends C implements I3 {
  // Identical, no problem:
  public int f() { return 1; }
}
```

Il metodo `f()` in `I3` implementato in `C` viene ridefinito in `C4`. La cosa non crea problemi perchè il metodo `f()` in `C4` è identico a quello in `C` ed `I3`

```
class C5 extends C implements I1 {
      public void f() {}
}
interface I4 extends I1, I3 {
}
```

Qui ci sono collisioni :

- `C5` ridefinisce `f()` che ha un tipo di ritorno diverso rispetto a quello in `I1`
- `I1` e `I3` definisco `f()` con tipo di ritorno diverso

Estendere un'interfaccia con l'ereditarietà

- E' possibile dichiarare un'interfaccia estendendola da un'altra/e
- Esempio

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

interface DangerousMonster extends Monster {
  void destroy();
}

interface Lethal {
  void kill();
}
```

(Vedere HorrorShow.java)

Estendere un'interfaccia con l'ereditarietà


```
class DragonZilla implements DangerousMonster {
  public void menace() {}
  public void destroy() {}
}

interface Vampire extends DangerousMonster, Lethal {
  void drinkBlood();
}

class VeryBadVampire implements Vampire {
  public void menace() {}
  public void destroy() {}
  public void kill() {}
  public void drinkBlood() {}
}
```

E' possibile usare `extends` in questo modo solo quando si ereditano interfacce

Estendere un'interfaccia con l'ereditarietà


```
public class HorrorShow {
  static void u(Monster b) { b.menace(); }
  static void v(DangerousMonster d) {
    d.menace();
    d.destroy();
  }
  static void w(Lethal l) { l.kill(); }
  public static void main(String[] args) {
    DangerousMonster barney = new DragonZilla();
    u(barney);
    v(barney);
    Vampire vlad = new VeryBadVampire();
    u(vlad);
    v(vlad);
    w(vlad);
  }
}
```

Raggruppare costanti

- Siccome qualsiasi campo in un'interfaccia è automaticamente `static` e `final`, l'interfaccia è uno strumento utile per creare gruppi di valori costanti (come con enum in C o in C++)
- Esempio

```
public interface Months {
  int
    JANUARY = 1, FEBRUARY = 2, MARCH = 3,
    APRIL = 4, MAY = 5, JUNE = 6, JULY = 7,
    AUGUST = 8, SEPTEMBER = 9, OCTOBER = 10,
    NOVEMBER = 11, DECEMBER = 12;

}
```

Raggruppare costanti


```
public interface Months {
  Month JANUARY = new Month(1, "JANUARY");
  Month FEBRUARY = new Month(2, "FEBRUARY");
  Month MARCH = new Month(3, "MARCH");
  Month APRIL = new Month(4, "APRIL");
  Month MAY = new Month(5, "MAY");
  Month JUNE = new Month(6, "JUNE");
  Month JULY = new Month(7, "JULY");
  Month AUGUST = new Month(8, "AUGUST");
  Month SEPTEMBER = new Month(9, "SEPTEMBER");
  Month OCTOBER = new Month(10, "OCTOBER");
  Month NOVEMBER = new Month(11, "NOVEMBER");
  Month DECEMBER = new Month(12, "DECEMBER");
}
```

(Vedere Months.java, Month.java)

Raggruppare costanti


```
public class Month {
    private int month;
    private String name;

    public Month(int month, String name ) {
       this.month = month;
       this.name = name;
    }

    public int getMonth() {
       return month;
    }

    public String getName() {
       return name;
    }
}
```

Inizializzare i campi nelle interfacce

- I campi definiti nelle interfacce sono automaticamente static e final e devono essere obbligatoriamente inizializzati
- La loro inizializzazione può essere fatta anche con espressioni non costanti, esempio

```
public interface RandVals {
  Random rand = new Random();
  int randomInt = rand.nextInt(10);
  long randomLong = rand.nextLong() * 10;
  float randomFloat = rand.nextLong() * 10;
  double randomDouble = rand.nextDouble() * 10;
}
```

I campi vengono inizializzati la prima volta che la classe viene caricata (ovvero quando si accede a qualsiasi campo per la prima volta)

(Vedere RandVals.java, TestRandVals.java)

Callbacks

» Una applicazione importante delle interfacce sono i cosiddetti *callbacks* che permettono di implementare elegantemente i pointers

Callbacks

» Per condividere un'agenda, bisogna che le persone che condividono l'agenda possano essere avvertite delle modifiche

```
class Appointment {
  Date date;
  String info;
}

class Agenda {
  void addAppointment(Appointment a) {

  }
  void cancelAppointment(Appointment a) {
    //...
  }
  void moveAppointment(Appointment a) {
    //...
  }
}
```

Callbacks

» Si definisce una interfaccia con un metodo per avvertire di una modifica

```
interface AgendaUser {
  void update (Appointment a);
}
```

» Si usa il tipo *AgendaUser* per manipolare dentro *Agenda* gli utenti da avvertire

```
class Agenda {
  AgendaUser[] users;
  void addUser(AgendaUser u) {
    //...
  }
  void addAppointment(Appointment a) {
    //...
    for(int i=0; i< users.length; i++) {
      users[i].update(a);
    }
  }
  //...
}
```

Callbacks

» Quando si definisce una class che sia un utente dell'agenda, è sufficiente dire che questa class implementa *AgendaUser* e implementare il metodo update:

```
class A implements AgendaUser {
    A(Agenda a) {
    a.addUser(this);
}
 public void update (Appointment a) {
System.out.println("update");
}
//...
}
```

Esercizi

- Mostrare che i campi di un'interfaccia sono implicitamente static e final
- Mostrare che tutti i metodi di un'interfaccia sono automaticamente public
- Cambiare l'esempio Adventure.java aggiungendo un'interfaccia CanClimb, che segua la forma delle altre interfacce
- Create tre interfacce, ciascuna con due metodi. Ereditate una nuova interfaccia dalle tre, aggiungendo un nuovo metodo. Create una classe implementando la nuova interfaccia ed ereditando anche da una classe concreta. Ora scrivere quattro metodi, ciascuno dei quali prende una delle quattro interfacce come argomento. In main() create un oggetto della vostra classe e passatelo a ciascuno dei metodi