Interfacce

Una interfaccia è un'astrazione per un insieme di funzioni pubbliche delle quali si definisce solo la segnatura, e non le istruzioni. Un'interfaccia viene poi implementata da una o più classi (anche astratte). Una classe che implementa un'interfaccia deve definire o dichiarare tutte le funzioni della interfaccia.

Dal punto di vista sintattico, un'interfaccia è costituita da un insieme di dichiarazioni di funzioni pubbliche (**no campi dati**, a meno che non sia `final`), la cui definizione **è necessariamente lasciata alle classi che la implementano**. Possiamo quindi pensare ad una interfaccia come ad una dichiarazione di un tipo di dato (inteso come un insieme di operatori) di cui non vogliamo specificare l'implementazione, ma che comunque può essere utilizzato da moduli software, indipendentemente appunto dall'implementazione.

Esempio: interfaccia `I` con una sola funzione `g()`

```
public interface  I {
  void g(); // implicitamente public;  e' una DICHIARAZIONE: notare ';'
}
```

Cosa si fa con un'interfaccia

Se `I` è un'interfaccia, allora **possiamo**:

- definire una o più classi che **implementano** `I`, cioè che definiscono tutte le funzioni dichiarate in `I`

- definire variabili e campi di tipo `I` (durante l'esecuzione, conterranno indirizzi di oggetti di classi che implementano `I`),

- usare i riferimenti di tipo `I`, sapendo che in esecuzione essi conterranno indirizzi di oggetti (quindi possiamo definire funzioni che prendono come argomento un riferimento di tipo `I`, restituire riferimenti di tipo `I`, ecc.);

mentre **non possiamo**:

- creare oggetti di tipo `I`, cioè non possiamo eseguire `new I()`, perchè non esistono oggetti di tipo `I`, ma esistono solo riferimenti di tipo `I`.

Utilità delle interfacce

Le funzioni di un'interfaccia costituiscono un modulo software `S` che:

- può essere utilizzato da un modulo esterno T (ad esempio una funzione `t()` che si aspetta come parametro un riferimento di tipo `S`), **indipendentemente** da come le funzioni di `S` sono implementate; in altre parole, non è necessario avere deciso l'implementazione delle funzioni di `S` per progettare e scrivere altri moduli che usano `S`;

- può essere implementato in modi alternativi e diversi tra loro (nel senso che più classi possono implementare le funzioni di `S`, anche in modo molto diverso tra loro);

- ovviamente, però, al momento di attivare un modulo `t()` che ha un argomento tipo `S`, occorre passare a `t()`, in corrispondenza di `S`, un oggetto di una classe che implementa `S`.

Tutto ciò aumenta la possibilità di **riuso**.

Esempio di interfaccia e di funzione cliente

Vogliamo definire una interfaccia `Confrontabile` che offra una operazione che verifica se un oggetto è **maggiore** di un altro, ed una operazione che verifica se un oggetto è **paritetico** ad un altro. Si noti che **nulla si dice** rispetto al criterio che stabilisce se un oggetto è maggiore di o paritetico ad un altro.

Si vuole scrivere poi una funzione che, dati tre riferimenti a `Confrontabile`, restituisca il maggiore tra i tre (o più precisamente un *massimale*, ovvero uno qualunque che non abbia tra gli altri due uno maggiore di esso).

Notiamo che, denotando con gli operatori binari infissi '$>$' e '$=$' le relazioni "maggiore" e "paritetico" (rispettivamente), x_1 è massimale in $\{x_1, x_2, x_3\}$ se e solo se:

$$(x_1 > x_2 \ \vee \ x_1 = x_2) \quad \wedge \quad (x_1 > x_3 \ \vee \ x_1 = x_3)$$

Esempio di interfaccia e di f. cliente (cont.)

```
// File unita1/Esempio16.java
interface  Confrontabile {
  boolean Maggiore(Confrontabile x);
  boolean Paritetico(Confrontabile x);
}

class Utilita {
  static public Confrontabile MaggioreTraTre(Confrontabile x1,
                                             Confrontabile x2,
                                             Confrontabile x3) {
    if ((x1.Maggiore(x2) || x1.Paritetico(x2)) &&
        (x1.Maggiore(x3) || x1.Paritetico(x3)))
      return x1;
    else if ((x2.Maggiore(x1) || x2.Paritetico(x1)) &&
             (x2.Maggiore(x3) || x1.Paritetico(x3)))
      return x2;
    else return x3;  }  }
```

Implementazione di un'interfaccia

Definiamo due classi che implementano l'interfaccia `Confrontabile`:

1. Una di queste è la classe `Edificio` (per la quale il confronto concerne l'altezza).

2. L'altra è una classe astratta `Persona` (per la quale il confronto concerne l'aliquota).

Implementazione di un'interfaccia (cont.)

```
// File unita1/Esempio17.java

class Edificio implements Confrontabile {
  protected int altezza;
  public boolean Maggiore(Confrontabile e) {
    if (e != null && getClass().equals(e.getClass()))
      return altezza > ((Edificio)e).altezza;
    else return false;
  }
  public boolean Paritetico(Confrontabile e) {
    if (e != null && getClass().equals(e.getClass()))
      return altezza == ((Edificio)e).altezza;
    else return false;
  }
}
```

```
abstract class Persona implements Confrontabile {
  protected int eta;
  abstract public int Aliquota();
  public int Eta() { return eta; }
  public boolean Maggiore(Confrontabile p) {
    if (p != null && Persona.class.isInstance(p))
      return Aliquota() > ((Persona)p).Aliquota();
    else return false;
  }
  public boolean Paritetico(Confrontabile p) {
    if (p != null && Persona.class.isInstance(p))
      return Aliquota() == ((Persona)p).Aliquota();
    else return false;
  }
}
```

Commenti sull'implementazione

Notiamo che nelle classi `Edificio` e `Persona` abbiamo usato due criteri differenti per stabilire se possiamo effettuare i confronti fra due oggetti tramite le funzioni `Maggiore()` e `Paritetico()`:

- per la classe (non astratta) `Edificio`, verifichiamo se i due oggetti siano della stessa classe (`Edificio` o derivata da essa);

- per la classe (astratta) `Persona`, verifichiamo se i due oggetti siano entrambi derivati dalla classe `Persona`.

  Ciò permette di effettuare il confronto anche fra oggetti di classi differenti, purché entrambe derivate da `Persona`.

Esempio di uso di interfaccia

A questo punto possiamo chiamare la funzione `MaggioreTraTre()`:

- sia su Persone (cioè passandole tre oggetti della classe `Persona`),
- sia su Edifici (cioè passandole tre oggetti della classe `Edificio`).

Esempio di uso di interfaccia (cont.)

```
// File unita1/Esempio18.java
class Studente extends Persona {
  public int Aliquota() { return 25; } }
class Professore extends Persona {
  public int Aliquota() { return 50; } }
class Esempio31 {
  public static void main(String[] args) {
    Studente s = new Studente();
    Professore p = new Professore();
    Professore q = new Professore();
    Edificio e1 = new Edificio();
    Edificio e2 = new Edificio();
    Edificio e3 = new Edificio();
    Persona pp = (Persona)Utilita.MaggioreTraTre(s,p,q);
    Edificio ee = (Edificio)Utilita.MaggioreTraTre(e1,e2,e3);
  }
}
```

Esercizio 10

Arricchire la classe `Utilita` con:

- una funzione `Massimale()` che, ricevuto come argomento un vettore di riferimenti a `Conparabile`, restituisca un elemento massimale fra quelli del vettore;

- una funzione `QuantiMassimali()` che, ricevuto come argomento un vettore di riferimenti a `Confrontabile`, restituisca un intero che corrisponde al numero di elementi massimali fra quelli del vettore.

Interfacce e classi che le implementano

Una classe può implementare anche più di una interfaccia (implementazione multipla), come mostrato da questo esempio:

```
public interface  I {
  void g();
}

public interface J {
  void h();
}

class C implements I,J {
  void g() { ... }
  void h() { ... }
}
```

Esempio di implementazione multipla

```
// File unita1/Esempio19.java
interface  I  { void g(); }
interface  I2 { void h(); }
class B {
  void f() { System.out.println("bye!"); }
}
class C extends B implements I, I2 {
  public void g() { System.out.println("ciao!"); }
  public void h() { System.out.println("hello!"); }
}
public class Esempio19 {
  public static void main(String[] args) {
    C c = new C();
    c.g();
    c.h();
    c.f();
  }
}
```

Interfacce ed ereditarietà

L'ereditarietà si può stabilire anche tra interfacce, nel senso che una interfaccia si può definire derivata da un'altra. Se una interfaccia `J` è derivata da una interfaccia `I`, allora tutte le funzioni dichiarate in `I` sono implicitamente dichiarate anche in `J`.

Ne segue che una classe che implementa `J` deve anche definire tutte le funzioni di `I`.

```
public interface  I {
  void g();
}
public interface J extends I {
  void h();
}
class C implements J {
  void g() { ... }
  void h() { ... }
}
```

Interfacce ed ereditarietà multipla

Limitatamente alle interfacce, Java supporta l'**ereditarietà multipla**: una interfaccia può essere derivata da un qualunque numero di interfacce.

```
public interface  I {
  void g();
}
public interface J {
  void h();
}
public interface M extends I, J {
  void k();
}
class C implements M {
  void g() { ... }
  void h() { ... }
  void k() { ... }
}
```

Differenza tra interfacce e classi astratte

Interfacce e classi astratte hanno qualche similarità. Ad esempio: entrambe hanno funzioni dichiarate e non definite; non esistono istanze di interfacce, e non esistono istanze dirette di classi astratte.

Si tenga però presente che:

- Una classe astratta è comunque una classe, ed è quindi un'astrazione di un insieme di oggetti (le sue istanze). Ad esempio, la classe `Persona` è un'astrazione per l'unione delle istanze di `Studente` e `Professore`.

- Una interfaccia è un'astrazione di un insieme di funzioni. Ad esempio, è difficile pensare concettualmente ad una classe `Confrontabile` che sia superclasse di `Persona` ed `Edificio`, e che quindi metta insieme le istanze di `Persona` ed `Edificio`, solo perché ha senso confrontare tra loro (con le funzioni `Maggiore()` e `Paritetico()`) sia le persone sia gli edifici.

Riassunto classi, classi astratte, interfacce

	`class`	`abstract class`	`interface`
Riferimenti	SI	SI	SI
Oggetti	SI	SI (indirettamente)	NO
Campi dati	SI	SI	NO*
Funzioni solo dichiarate	NO	SI	SI
Funzioni definite	SI	SI	NO
`extends (abstract) class`	0 o 1	0 o 1	0
`implements interface`	≥ 0	≥ 0	0
`extends interface`	0	0	≥ 0

*Eccetto che per campi dati `final` (cioè costanti).