Java

interfacce e classi annidate

G. Prencipe
prencipe@di.unipi.it

Interfacce

- La parola chiave **interface** serve a creare classi **abstract** pure
- Permette di stabilire lo scheletro di una classe
  - Nomi di metodi, liste di argomenti, e tipi di ritorno, ma nessuno corpo per i metodi
  - Può contenere anche membri variabile, ma essi sono implicitamente **static** e **final**
- In altre parole, serve a fornire una forma, ma non un'implementaizone

Interfacce

- Una interfaccia dice
  - Ecco come devono essere le classi che *implementeranno* questa particolare interfaccia
- Per crearne una è sufficiente utilizzare la parola chiave **interface** invece di **class**
  - Tutto il resto rimane uguale alle definizioni di classe
    - Es: nome file uguale al nome interfaccia se **public**
- Per implementare una interfaccia si utilizza la parola chiave **implements**

Interfacce

- I metodi all'interno di una interfaccia sono resi automaticamente **public** (quindi la parola chiave **public** può essere anche omessa)
  - Una interfaccia definisce cose sempre visibili
  - Le classi che la implementano devono dichiarare **public** questi metodi
- Vediamo un esempio: **Music5.java**

Interfacce

- Non importa se si fa *upcast* a una classe "regolare" chiamata **Instrument**, a una classe **abstract Instrument**, o a una **interface** chiamata **Instrument**
  - Il comportamento è sempre lo stesso
  - Come si vede dal metodo **tune()**, da cui non si riesce a capire che tipo di classe stiamo trattando
  - Vengono forniti al programmatore *diversi controlli* con i quali creare e usare oggetti

Ereditarietà multipla in Java

- L'interfaccia è qualcosa in più di una semplice classe astratta "pura"
- Infatti, è possibile combinare più interfacce
  - Ci sono situazioni in cui vogliamo dire che un oggetto **x** è sia di tipo **a** che **b** che **c**
  - In C++ questo si chiama *ereditarietà multipla*
    - Una classe sottoclasse di classi diverse
  - In Java però se si eredità da una non-interfaccia, si può ereditare *solo da una*
    - Ma è possibile estendere più interfacce

Esempio

- Quindi non esiste la ereditarietà multipla del C++, ma possiamo fare qualcosa estendendo più interfacce
- Vediamo un esempio: **Adventure.java**

Collisioni di nomi con interfacce

- Nell'esempio visto, si ha collisione di nome sul metodo **fight()**, ma tutto va bene perché i due metodi (in **ActionCharacter** e **CanFight**) sono lo stesso
- Cosa accade se così non è?
- Vediamo….

Collisioni di nomi con interfacce

```
interface I1 { void f(); }
interface I2 { int f(int i); }
interface I3 { int f(); }
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
}
class C4 extends C implements I3 {
  // Identico, nessun problema
  public int f() { return 1; }
}
```

Collisioni di nomi con interfacce

```
interface I1 { void f(); }
interface I2 { int f(int i); }
interface I3 { int f(); }
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
}
class C4 extends C implements I3 {
  // Identico, nessun problema
  public int f() { return 1; }
}
```

Domanda: cosa accade se aggiungo il seguente codice:

```
class C5 extends C implements I1 {}
interface I4 extends I1, I3 {}
```

Collisioni di nomi con interfacce

- Ottengo errori!!
- class C5 extends C implements I1 {}
  - Tipi di ritorno incompatibili per **f()**
- interface I4 extends I1, I3 {}
  - I1 e I3 definiscono entrambe **f()**, ma con tipi di ritorno diversi

Estendere interfacce

- Nell'esempio di prima abbiamo visto che è anche possibile *estendere le interfacce* utilizzando ereditarietà
- Vediamo un esempio: **HorrorShow.java**
- Nota: è possibile *estendere più di una interfaccia alla volta*, mentre questo non è possibile con le classi (vedi **Vampire**)

Interfacce e costanti

- Dato che tutti i membri variabile di una interfaccia sono **final** e **static**, interface è uno strumento utile per creare gruppi di valori costanti
  - Tipo **enum** in C o C++

Interfacce e costanti

- 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;
} ///:~
```

- Stile Java: tutto maiuscolo per **static final** (si usa _ per separare identificatori composti da più nomi)

Interfacce e costanti

- I campi in una interfaccia sono automaticamente **public**
- Inserendo l'interfaccia in un pacchetto, è possibile importare i valori con espressioni tipo

  Months.JANUARY

Interfacce annidate

- Le interfacce possono essere annidate all'interno di classi e all'interno di altre interfacce
  - La sintassi è la stessa vista finora
  - Possono essere **public** o avere visibilità *package access* (non si specifica **public**)
  - Inoltre, le interfacce annidate in classi possono essere **private**
    - Interfacce **private** possono essere implementate solo all'interno della classe in cui sono definite
    - Non è possibile avere interfacce **private** annidate in altre interfacce (tutto in una interfaccia deve essere **public**)

Interfacce annidate

- Vediamo un esempio: **NestingInterfaces.java**

Classi annidate

- Dall'esempio precedente si può notare come in Java è possibile avere classi annidate
- Vengono create esattamente come visto prima
- Vediamo un esempio: **Parcel1.java**

Classi annidate

- *Uno scopo* delle classi annidate è quello di *nascondere l'implementazione*
  - In una classe viene definito un nuovo tipo (classe)
  - Tipicamente, come è fatto questo tipo non viene reso disponibile all'esterno

Classi e interfacce annidate

- Vediamo meglio cosa accade nel caso di classi e interfacce annidate
- Prima, definiamo due interfacce in due file distinti: **Destination.java** e **Contents.java**
- Poi in **TestParcel.java** implementiamo queste interfacce
- Notare la classe annidata **private Pcontents**
  - Normalmente le classi non possono essere **private** (solo **public** o con *package access*)

Link alla classe esterna

- Un secondo scopo delle classi interne è quello di avere *chiusura*
  - Le classi interne hanno accesso ai metodi e ai campi della classe esterna (anche se essi sono **private**)
- Infatti, quando si crea una classe interna, un oggetto di quella classe ha un *link* all'oggetto della classe esterna che lo ha creato
  - Cioè, la classe interna mantiene un riferimento al particolare oggetto della classe esterna che è responsabile della sua creazione
  - Quindi, un oggetto della classe interna può essere creato solo in associazione con un oggetto della classe esterna
  - Vediamo un esempio: **Sequence.java**

Link alla classe esterna

- Se si ha bisogno di generare un riferimento esplicito all'oggetto della classe esterna, si utilizza **this**
  - NomeClasseEsterna.this
  - Es.: nell'esempio **Sequence.java**, un riferimento alla classe **Sequence** si ottiene con Sequence.this
  - Il riferimento che si ottiene è del tipo corretto (questo è noto già a tempo di compilazione, quindi non produce *overhead* a *run-time*)

Creare un oggetto "interno"

- Da quanto detto finora, segue che per creare un oggetto di una classe interna è necessario avere un oggetto della classe esterna
  - Vedi considerazioni sul *link* fatte prima
- La sintassi è

  NomeClasseEsterna.NomeClasseInterna pippo = oggettoClasseEsterna.new NomeClasseInterna()
- Vediamo un esempio: **Parcel11.java**
- Non è necessario creare un oggetto della classe esterna per crearne uno interno, se la classe interna è dichiarata **static**
  - Questo caso viene detto di *nested classes*

Annidamento multiplo

- Non importa quanto sia alto il grado di annidamento, una classe interna può sempre accedere a tutti i membri delle classi esterne in cui è annidata
- Vediamo un esempio: **MultiNestingAccess.java**

Ereditare da una classe interna

- È possibile ereditare da una classe interna
- In questo caso però, dato che un oggetto di una classe interna per esistere ha bisogno di un oggetto della classe esterna, bisogna fare attenzione in fase di inizializzazione
- Infatti, bisogna sempre creare prima un oggetto della classe esterna

Ereditare da una classe interna

```
class WithInner {
  class Inner {}
}
public class InheritInner extends WithInner.Inner {
  //! InheritInner() {} // Non Compila!! Bisogna inizializzare
  //! prima il riferimento all'oggetto esterno!!
  InheritInner(WithInner wi) {
    wi.super(); // Sintassi per invocare il costruttore dell'oggetto esterno
  }
  public static void main(String[] args) {
    WithInner wi = new WithInner();
    InheritInner ii = new InheritInner(wi);
  }
} ///:~
```

Altri utilizzi delle classi interne

- Un utilizzo tipico delle classi annidate consiste nell'avere un metodo nella classe esterna che restuituisce un riferimento alla classe interna
- Vediamo un esempio: **Parcel2.java**
- Notare come (nel **main**), per creare un oggetto della classe interna si deve specificare il tipo della classe interna come **NomeClasseEsterna.NomeClasseInterna**

Overriding di classi annidate

- Se estendiamo una classe che ha una classe annidata, e poi facciamo *overriding* della classe annidata, non succede nulla
  - L'overriding è ignorato perché le due classi sono entità separate
- Vediamo un esempio: **BigEgg.java**

Overriding di classi annidate

- È comunque possibile ereditare esplicitamente da una classe interna (e quindi ottenere *overriding* dei suoi metodi), come mostrato in **BigEgg2.java**

Classi interne a metodi

- È possibile definire anche classi all'interno di metodi
- Vediamo un esempio

Esempio -- classe in metodo

```
class D {}
abstract class E{}
// Si estende D
class Z extends D{
   // Si definisce un metodo per implementare E
   E makeE(){
   // Definizione di classe all'interno di un metodo
        class MyE extends E {}
        return new MyE(); // restituisco un oggetto E
   }
}
```

Classi interne anonime

- Il codice di prima poteva essere scritto anche così

```
class D {}
abstract class E{}  // Si estende D
class Z extends D{
   E makeE(){ return new MyE(){};}
}
```

- È detta *classe interna anonima* e consente di definire il corpo della classe al momento della creazione di un oggetto (implementa una classe astratta)
  - Si evita di introdurre un nome fittizio per la classe interna

Identificatori .class

- Sappiamo che per ogni classe definita, dopo la compilazione viene generato un file .class
- Quando si creano classi interne accade la stessa cosa
  - Viene generato un file chiamato **NomeClasseEsterna$NomeClasseInterna.class**

A cosa servono??

- Come mai esistono le classi annidate e a cosa servono??
- Riassumendo, gli scopi principali e tipici delle classi annidate sono due
  - Nascondere l'implementazione
    - Si dà l'opportunità a una classe di definire dei tipi interni non visibili all'esterno
    - Quindi non è molto utile avere una classe interna public (sarebbe equivalente a definirla in una classe esterna separata)
  - Chiusura
    - Accesso totale ai membri della classe esterna

Esercizi

1. Scrivere un programma che dimostri che i membri variabile di una interfaccia sono implicitamente **static** e **final**
2. Creare una interfaccia in un pacchetto **MiaInterfaccia** che contiene tre metodi. Implementare l'interfaccia in un pacchetto diverso

Esercizi

3. In **EsTreInter.java** creare 3 interfacce, ognuna con due metodi
   1. Creare una nuova interfaccia che erediti dalle 3 precedenti, aggiungendo un nuovo metodo
   2. Creare una classe **C** implementando la nuova interfaccia e estendendo da una classe concreta
   3. Scrivere 4 metodi, ognuno dei quali prende una della 4 interfacce come argomento
   4. Nel **main**, creare un oggetto della classe **C** e passarlo come argomento ad ognuno dei 4 metodi

Esercizi

4. Creare due interfacce e una classe concreta
   1. Creare una classe con due classi interne: la prima implementa l'interfaccia e la seconda estende la classe concreta
5. Implementare un array associativo con la classe **Vector**
   1. Si crei una classe interna **Entry** definita come coppia di interi, che serve come chiave per inserire ed estrarre gli elementi dal **Vector**
   2. Si implementino i metodi per estrarre e inserire elementi
   3. Nota: utilizzo delle classi interne per nascondere l'implementazione della entry

Java

interfacce e classi annidate

fine