Java

riutilizzo delle classi

G. Prencipe
prencipe@di.unipi.it

Introduzione

- Una delle caratteristiche fondamentali in Java è il riutilizzo del codice
- Ci sono due modi per ottenerlo
  - Creare oggetti di classi esistenti nelle nuove classi che definiamo
    - *Composizione*: la nuova classe è *composta* da oggetti di classi esistenti
    - Si riutilizza la *funzionalità* delle classi esistenti, non la loro *struttura*
  - Creare una nuova classe come *tipo* di una classe esistente
    - Si aggiunge codice alla *struttura* delle classi esistenti, senza modificare la classe esistente
    - *Ereditarietà*

Composizione

- È qualcosa che abbiamo già visto varie volte
  - Si ottiene inserendo riferimenti a oggetti all'interno della nuova classe
- Vediamo un esempio: **SprinklerSystem.java**

Esempio -- composizione

- In questo esempio notiamo la presenza del metodo **toString()**
- Ogni oggetto non primitivo ne ha uno, ed è invocato quando il compilatore vuole una **String**a ma ottiene un oggetto
- Se abbiamo

  String s = ""+ source

  Il compilatore trasforma l'oggetto source in **String**a invocando il metodo **toString()**

Inizializzazione oggetti

- Come già detto, le variabili di classe che sono primitive vengono automaticamente inizializzate con dei valori di default
- Nel caso di variabili di classe che sono oggetti, il valore di default è **null**
  - Se si cerca di utilizzare un metodo di un oggetto non inizializzato con **new,** si ottiene una *eccezione*

Inizializzazione oggetti

- Se vogliamo inizializzare un riferimento ad un oggetto (variabile di classe), abbiamo tre opzioni
  1. Nel punto in cui l'oggetto è definito
     1. Viene inizializzato prima che il costruttore della classe venga invocato
  2. Nel costruttore della classe
  3. Subito prima di utilizzare l'oggetto
     1. *Lazy initializaztion*
     2. Riduce l'overhead in situazione dove la creazione dell'oggetto è costosa e l'oggetto non deve essere creato sempre e comunque

Esempio -- inizializ. oggetti

- Vediamo l'esempio **Bath.java** che mostra queste tre opzioni

Ereditarietà

- È un aspetto fondamentale della OOP
- In Java, quando si crea una classe, si ha sempre ereditarietà: infatti, tutte le classi sono sottoclassi di **Object**
- La parola chiave che indica ereditarietà è **extends** seguita dal nome della superclasse
- Con l'ereditarietà, la sottoclasse eredita automaticamente tutti i campi e i metodi della superclasse

Ereditarietà

- La sintassi è:

  class NomeSottoclasse extends Nomesuperclasse {….}
- Vediamo l'esempio **Detergent.java**

Esempio -- ereditarietà

- Nel metodo **append()** di **Cleanser**, le stringhe sono concatenate con **+=**
- Sia **Cleanser** che **Detergent** contengono il main
  - Verrà invocato il **main** della classe invocata da linea di comando
  - Eclipse lo chiede al momento del **Run**
  - Si inseriscono più **main** per facilitare il testing delle varie classi

Esempio -- ereditarietà

- Tutti i metodi di **Cleanser** (**append()**, **dilute()**, **apply()**, **scrub()**, and **toString()**) esistono anche in **Detergent**
  - Esempio di riutilizzo del codice
- Il metodo **scrub()** è ridefinito in **Detergent**
  - Per invocare il metodo **scrub()** della superclasse si utilizza **super** (**super.scrub()**)
    - Altrimenti faremmo una invocazione *ricorsiva*

Esempio -- ereditarietà

- È possibile anche aggiungere nuovi metodi (estendere l'interfaccia) della sottoclasse
  - Come **foam()**
  - Dal **main** in **Detergent** si nota come per un oggetto **Detergent** è possibile invocare tutti i metodi definiti sia in **Cleanser** che in **Detergent**

Inizializzazioni

- Quando viene creato un oggetto della sottoclasse, è come esso contenesse un *sotto-oggetto* della superclasse
- Questo sotto-oggetto è lo stesso che si otterrebbe creando un oggetto della superclasse stessa
- È fondamentale che questo sotto-oggetto sia inizializzato correttamente
- Le inizializzazioni vengono fatte nei costruttori
  - Java inserisce automaticamente chiamate al costruttore della superclasse nei costruttori della sottoclasse
  - Vediamo un esempio: **Cartoon.java**

Esempio -- inizializzazioni

- Dall'esempio si nota come le inizializzazioni avvengono prima nella superclasse e poi nelle sottoclassi
- Anche se non ci fosse un costruttore **per Cartoon()**, il compilatore creerebbe un costruttore di default che chiamerebbe i costruttori delle superclassi
  - Provare per credere!!

Costruttori con argomenti

- Abbiamo visto che automaticamente invoca costruttori delle superclassi
- Se essi sono *tutti* definiti per prendere argomenti (cioè *non ci sono* costruttori di dafault), il compilatore non sa cosa fare
  - Ricordiamo che il costruttore di default viene inserito automaticamente solo se *non sono definiti affatto* costruttori
- In questo caso bisogna invocare esplicitamente i giusti costruttori della superclasse con **super**
- Vediamo un esempio: **Chess.java**

Esempio -- costruttori

- Se non si invocasse esplicitamente il costruttore **super(i)**, il compilatore si lamenterebbe
- Inoltre, la chiamata al costruttore della superclasse *deve* essere la *prima* cosa che si fa nel costruttore della sottoclasse
  - Altrimenti, errore a compilazione

Composizione e ereditarietà

- È molto comune combinare composizione e ereditarietà
- Vediamo un esempio: **PlaceSetting.java**

Overloading

- Se in una classe abbiamo utilizzato *overloading* per un certo metodo, ridefinire lo stesso metodo in una sottoclasse non *nasconde* nessuna delle versioni definite nella superclasse
- Quindi *l'overloading* funziona indipendentemente da dove il metodo è stato definito
- Vediamo un esempio: **Hide.java**

Esempio -- overloading

- Quindi vengono invocati i *metodi giusti (*nella superclasse o nella sottoclasse) a seconda del tipo del parametro passato come argomento

Composizione vs ereditarietà

- La composizione è tipicamente utilizzata quando si vogliono caratteristiche di una classe già esistente all'interno della nuova classe, ma non la sua interfaccia
  - Cioè, si include un oggetto nella nuova classe per utilizzare le sue funzionalità, ma l'utente della *nuova classe* avrà a disposizione l'interfaccia definita per *essa* e non quella dell'oggetto incluso
  - A questo scopo si inseriscono tipicamente gli oggetti come **private** nella nuova classe
  - A volte comunque ha anche senso permettere all'utente di accedere direttamente a questo oggetto, rendendolo **public**

Esempio

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

Composizione vs ereditarietà

- Con l'ereditarietà si prende una classe esistente e si crea una sua *speciale versione*
- Tipicamente, questo vuol dire che si prende una classe *generale* e la si *specializza* per delle necessità specifiche
- In generale, la relazione è-un si esprime tramite ereditarietà,
  - *Auto **è-un** veicolo*
- mentre la relazione ha-un tramite composizione
  - *Auto **ha-un** finestrino*

Protected

- Serve per estendere l'accessibilità di variabili e metodi anche alle sottoclassi
  - È **private** esteso alle sottoclassi
- Vediamo un esempio: **Orc.java**

Sviluppo incrementale

- Uno dei vantaggi dell'ereditarietà è che permette lo *sviluppo incrementale*
- Infatti, ereditando da classi esistenti e aggiungendo nuovi metodi e variabili, si lascia intatto il codice esistente, senza il pericolo di introdurre in esso errori
- Inoltre, non è nemmeno necessario avere il codice delle classi esistenti per riutilizzare il codice (al più si importa un pacchetto)

Esempio

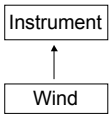
- L'aspetto più importante dell'ereditarietà è l'introduzione di una relazione tra la superclasse e la sottoclasse
  - *"La nuova classe è del tipo di una classe esistente"*
- Vediamo un ulteriore esempio che descrive strumenti musicali: **Wind.java**

Upcasting

- Un aspetto interessante dell'esempio è il metodo **tune()**, che accetta un riferimento a un **Instrument**
- Nel main però, il metodo è invocato passando un riferimento a Wind
- In fatti, un Wind è anche un Instrument
- In **tune()**, il codice funziona per **Instrument** e per qualsiasi cosa derivata da **Instrument**, e l'atto di trasformare un riferimento **Wind** in un **Instrument** è detto *upcasting*

Upcasting

- Il termine upcasting deriva dal modo con cui tradizionalmente si rappresenta graficamente l'ereditarietà
  - Con la radice delle classi in cima, il diagramma si sviluppa verso il basso
  - Il *casting* da un tipo derivato a uno base si muove verso *l'alto* nel diagramma



Upcasting

- L'*upcasting* è sempre sicuro, perché si va da un tipo specifico a uno più generale
  - Cioè, la classe derivata è un *sovrainsieme* della classe base
    - La classe derivata può contenere più metodi della classe base, ma deve contenere *almeno* quelli della classe base
  - L'unica cosa che può succedere alla interfaccia della classe derivata durante l'*upcasting* è che può perdere metodi, non guadagnarli
  - Questo è il motivo per cui il compilatore permette l'*upcasting* senza alcun cast esplicito o notazione speciale

La parola chiave **final**

- In Java **final** vuol dire "questo non può essere modificato"
- Si possono voler evitare cambiamenti per due ragioni: progettazione e efficienza
- Analizziamo le tre situazioni in cui la parola chiave **final** può essere utilizzata
  1. Dati
  2. Metodi
  3. Classi

Dati **final**

- Serve ad indicare dati costanti
- Una costante è utile per due motivi
  - Può essere una costante a *tempo di compilazione* che non cambia mai
  - Può essere un valore inizializzato a *tempo d'esecuzione* che non vogliamo cambi

Dati **final**

- Nel caso di costanti a *tempo di compilazione*, il compilatore può utilizzare il valore nei calcoli in cui è utilizzato
  - Cioè, i calcoli possono essere svolti a tempo di compilazione, risparmiando tempo in esecuzione
- Queste costanti devono essere tipi primitivi
  - Un valore deve chiaramente essere fornito al momento della definizione della costante

Dati **final**

- Un campo che è sia **static** che **final** occupa un unico pezzo di memoria che non può essere modificato
- Se utilizzato con riferimenti ad oggetti, allora il riferimento è *costante*
- Una volta che il riferimento è *inizializzato* con un oggetto, non può più essere modificato per riferire un altro oggetto
  - L'oggetto stesso può comunque essere modificato
  - In Java *non c'è modo* per rendere un oggetto costante

Dati **final**

- Queste restrizioni coinvolgono anche gli *array*, che sono oggetti
- Vediamo un esempio: **FinalData.java**
- Tipi primitivi **final static** con valori costanti iniziali sono indicati per convenzione con tutte le lettere maiuscole, con le parole separate da _ (come in C)

Esempio -- dati **final**

- Essere **final** non implica che il valore sia noto sempre a tempo di compilazione, come per **i4** e **i5**, che non possono dunque essere costanti a tempo di compilazione
  - Notare la differenza tra **final** e **final static**
    - Nel risultato **i4** può cambiare tra **fd1** e **fd2**, mentre per **i5** c'è un solo possibile valore (che non cambia creando un secondo oggetto **FinalData**)

Blank **final**

- Java permette la creazione di campi dichiarati **final** ma che non hanno dichiarazione (*blank **final***)
- Devono comunque essere inizializzati prima di essere utilizzati
  - In ogni costruttore

Esempio -- blank **final**

```
class Poppet {
  private int i;
  Poppet(int ii) { i = ii; }
}
public class BlankFinal {
  private final int i = 0; // final inizializzato
  private final int j; // Blank final
  private final Poppet p; // riferimento blank final
  // Blank finals DEVONO essere inizializzati nel costruttore
/* continua */
```

Esempio -- blank **final**

```
public BlankFinal() {
    j = 1; // Inizializza blank final
    p = new Poppet(1); // Inizializza riferimento blank final
  }
  public BlankFinal(int x) {
    j = x; // Inizializza blank final
    p = new Poppet(x); // Inizializza riferimento blank final
  }
  public static void main(String[] args) {
    new BlankFinal();
    new BlankFinal(47);
  }
} ///:~
```

Argomenti **final**

- Java permette anche di rendere gli argomenti **final**
- Questo significa che all'interno del metodo non è possibile modificare quello a cui punta il riferimento relativo all'argomento **final**

Esempio -- argomenti **final**

```
class Gizmo {
  public void spin() {}
}
public class FinalArguments {
  void with(final Gizmo g) {
    //! g = new Gizmo(); // Illegale -- g è final
  }
  void without(Gizmo g) {
    g = new Gizmo(); // OK -- g non final
    g.spin();
  }
  // void f(final int i) { i++; } // Non modificabile
  // Si può solo leggere da una primitiva final

/* continua */
```

Esempio -- argomenti **final**

```
int g(final int i) { return i + 1; }
  public static void main(String[] args) {
    FinalArguments bf = new FinalArguments();
    bf.without(null);
    bf.with(null);
  }
} ///:~
```

Metodi **final**

- Ci sono due motivi per avere un metodo **final**
  1. Evitare che sottoclassi possano modificarne il comportamento tramite *overriding*
  2. Per ragioni legate all'efficienza
     - Quando il compilatore vede un metodo final può (a sua discrezione) evitare il normale meccanismo di invocazione metodi
       - *Argomenti sullo stack, saltare al codice ed esgeuirlo, eliminare gli argomenti dallo stack*
     - e rimpiazzare la chiamata con una copia del codice presente nel corpo del metodo
     - Se il metodo ha tanto codice, non è detto che ci si guadagni (il compilatore analizza la situazione e decide cosa fare con il metodo **final**)

final e **private**

- Tutti i metodi **private** in una classe sono implicitamente **final**
  - Dato che non è possibile accedere un metodo **private**, non è possibile modificarlo (con *overriding*)
  - È possibile aggiungere **final** a un metodo **private**, ma non si aggiunge alcun significato
- Apparentemente però il compilatore non sempre si lamenta se si utilizza *overriding* su un metodo **private**
  - Vediamo un esempio: **FinalOverridingIllusion.java**

Esempio -- **final** e **private**

- *Overriding* è possibile solo se un metodo è parte dell'interfaccia della superclasse
- Metodi **private** non ne fanno parte!!
- Quindi nell'esempio non abbiamo fatto *overriding*, ma semplicemente *creato nuovi metodi*!!
  - I metodi **private** non sono raggiungibili e servono solo per organizzare il codice all'interno della classe

Classi **final**

- Quando una classe è final si dichiara che non si vuole permettere ereditarietà su quella classe
  - Si dice che per quella classe non sono necessari cambi e quindi nessuna sottoclasse è necessaria

Esempio -- classi **final**

```
class SmallBrain {}
final class Dinosaur {
  int i = 7;
  int j = 1;
  SmallBrain x = new SmallBrain();
  void f() {}
}
//! class Further extends Dinosaur {}
// errore: non si può estendere la classe 'Dinosaur'

/* continua */
```

Esempio -- classi **final**

```
public class Jurassic {
  public static void main(String[] args) {
    Dinosaur n = new Dinosaur();
    n.f();
    n.i = 40;
    n.j++;
  }
} ///:~
```

Classi **final**

- Dato che una classe **final** impedisce la creazione di sue sottoclassi, tutti i suoi metodi sono implicitamente **final**,
  - Dato che non è possibile fare il loro *overriding*

Esercizi

1. Creare due classi **A** e **B**, con costruttori di default, che annunciano se stessi
2. Inserire una nuova classe **C** sottoclasse di **A**, e creare un membro della classe **B** in **C**. Non creare un costruttore per **C**
3. Creare un oggetto della classe **C** e osservare il risultato
4. Modificare **A** e **B** in modo che abbiano costruttori con argomenti
5. Scrivere un costruttore per **C** e effettuare tutte le inizializzazioni nel costruttore di **C**

Esercizi

6. Creare una classe con un metodo su cui si applica *overloading* 3 volte
7. Inserire una sua sottoclasse, in cui si applica ancora una volta *overloading* sul metodo del punto precedente
8. Mostrare che tutti e 4 i metodi sono disponibili nella sottoclasse

Java

riutilizzo delle classi

fine