Java

nascondere l'implementazione

G. Prencipe
prencipe@di.unipi.it

Introduzione

- Un fattore importante nella OOP è la separazione tra le cose che possono cambiare e quelle che non devono cambiare
- Questo è particolarmente importante per le *librerie*
  - L'utente deve fare affidamento sulle librerie che usa ed essere sicuro che non dovrà riscrivere nulla se una nuova versione della libreria viene scritta
  - Chi fornisce le librerie deve avere la libertà di poter modificare le librerie senza però creare inconsistenze nei programmi degli utenti

Java

- In Java questa seprazione è ottenuta tramite *specificatori degli accessi*
  - Permettono a chi definisce le librerie di stabilire cosa è accessibile agli utenti e cosa no
  - In questo modo chi fornisce le librerie sa esattamente cosa può modificare e cosa no, senza correre il rischio di rendere inutilizzabile il codice dei clienti

Controllo degli accessi

- Il controllo degli accessi avviene tramite le parole chiave **public**, **protected**, e **private**
  - In generale, il progettista di librerie vorrà "tenere le cose" quanto più **private** possibile
- Infine, i componenti di una libreria in Java sono raccolte in un **package**

Package

- Un **package** serve a rendere disponibile le classi accedute con **import**
- L'**import** serve per fornire un meccanismo per gestire lo spazio dei nomi
  - I nomi di tutti i membri di una classe sono separati tra lori
    - Un metodo **f()** in una classe **A** è diverso da un metodo **f()** in **B**
  - Purtroppo però possono esserci conflitti tra i nomi di classi
  - Negli esempi visti finora tutte le classi sono in un unico file, e quindi i nomi delle classi sono necessariamente diversi

Package

- Problemi possono sorgere se abbiamo più file, e nomi di classi uguali in diversi file
  - Abbiamo già visto che il meccanismo di *naming* di Java permette di distinguere comunque fra classi in file diversi, ma comunque può crearsi confusione….porre attenzione a questo fattore

Unità di compilazione

- Quando scriviamo un file Java, esso viene tipicamente chiamato *unità di compilazione*
- Ogni unità deve avere il nome che termina con **.java** e può contenere una classe **public** che deve avere lo stesso nome del file
  - Ci può essere *una* sola classe **public** in ogni unità, altrimenti abbiamo errore in compilazione
  - Vedere esempio **DuePublic.java**
  - Altre classi (non **public**) non sono accessibili all'esterno

Compilazione

- Il risultato della compilazione di un **.java** è un file **.class** per ogni classe presente nel file **.java**
  - I file **.class** contengono *bytecode* che sarà poi interpretato dalla JVM
- Un programma funzionante sarà dunque un insieme di file **.class** che possono essere "impacchettati" e compressi in un file JAR

Librerie

- Una libreria è un gruppo di questi file **.class**
- Per specificare che tutti questi file sono insieme, si utilizza la parola chiave **package** all'inizio (primo comando in assoluto) di ogni file

  package miopacchetto;
  - La classe **public** presente in quel file farà parte del pacchetto **miopacchetto**

Riassumendo

- Ogni file **.java** contiene al più una classe **public**, che deve avere lo stesso nome del file
  - Tutte le classi non **public** non sono visibili all'esterno
- La compilazione produce un file .**class** per ogni classe
- I **.class** corrispondenti alle classi **public** faranno parte di un pacchetto se viene usata la parola chiave **package** in cima al file che contiene quella classe

Nomi di pacchetti

- Tipicamente tutti i **.class** contenuti in un pacchetto sono contenuti in una singola directory
  - Il path che porta alla directory che contiene i **.class** per un certo pacchetto è codificato nel nome del pacchetto stesso (**java.lang.util**)
- L'interprete Java, per localizzare i **.class** utilizza la variabile d'ambiente (del SO) CLASSPATH
  - Contiene il path a una o più directory utilizzate come *radice* nella ricerca di file **.class**

Nomi di pacchetti

- L'interprete considera il nome del pacchetto e, partendo dalla *radice*, sostituisce ogni **.** nel nome del **package** con uno **/** generando così il path che conduce ai **.class**
- Ad esempio, supponendo che il CLASSPATH sia **C:/Doc/JavaT**, un pacchetto chiamato **miopacchetto.primo** si trova in **C:/Doc/JavaT/miopacchetto/primo**

Collisioni

- Cosa succede se due librerie sono importate con ***** e includono classi con nomi uguali?
  - Ad esempio ci sono due **package** che contengono due classi con lo stesso nome, e entrambi sono importati
- Fino a quando non si utilizza la classe in conflitto, tutto OK
- Altrimenti il compilatore chiederà di specificare esplicitamente la classe a cui ci si riferisce (specificando tutto il path)

Creare una libreria

- A questo punto è chiaro che in genere è conveniente scriversi una propria libreria dei tool che si utilizzeranno durante lo sviluppo, per ridurre la duplicazione di codice
- Proviamo a fare un esempio "stupido": creare una libreria tools che contiene i metodi **rint()** e **rintln()** per stampare a Console

Esercizio

- Creiamo un pacchetto **mieiPacchetti.tools** (in un progetto **P**) in cui definiamo i nuovi metodi di stampa
- Scriviamo una classe **ToolTest.java** in cui importiamo il nuovo pacchetto e invochiamo i due nuovi metodi
  - Nota: in Eclipse si specifica l'inclusione del nuovo pacchetto nel path dalle *proprietà* di **ToolTest** (*Java Build Path*)

Esempio

- Notare nel test la trasformazione di un numero in **String**a, utilizzando la **String**a vuota ("")
  - **System.out.println(100)** funziona lo stesso
  - Se vogliamo che anche la nostra **P.rintln(100)** funzioni, dobbiamo fare un po' di *overloading*!!
  - Al momento infatti funziona solo se chiamata su **String**he

Package

- Un utilizzo interessante dei **package** è quello di poterli usare in fase di debugging
- Infatti, durante lo sviluppo è possibile utilizzare un **package** con funzioni di *debugging*
- Quando il prodotto è pronto, è sufficiente importare un **package** simile a quello utilizzato normalmente, ma con le funzioni di *debugging* disabilitate

Controllo degli accessi

- Abbiamo già detto che **public**, **private** e **protected** sono le parole chiave che consentono in Java di controllare gli accessi alle funzionalità delle classi
- Analizziamo queste tre modalità

Package access

- Cosa succede se non si utilizza nessuna di queste tre parole chiave nella definizione di variabili, metodi e classi?
- Questo tipo di accesso viene detto **package access**
  - Tutte le altre classi nel pacchetto corrente hanno accesso al membro in questione, ma a tutte le classi al di fuori del pacchetto corrente esso risulta **private**
    - Controllare in Eclipse: le classi create finora sono sempre tutte sotto un unico pacchetto
  - Quindi, chi utilizza questo pacchetto non vede questi membri (questo è il comportamento di *default*, visto che non prevede la specifica di alcuna parola chiave)

Esercizio -- package access

- Provare, in un progetto qualunque tra quelli creati finora, a creare una classe senza *specificatore di accesso*
- Successivamente, provare a creare una nuova classe nello stesso progetto che utilizza questa classe
- ….tutto dovrebbe funzionare

Public

- La dichiarazione che segue **public** è *visibile a tutti*

Public

- La dichiarazione che segue **public** è *visibile a tutti*
- Esempio

```
package c05.dessert;
public class Cookie {
  public Cookie() {
   System.out.println("Cookie constructor");
  }
  void bite() { System.out.println("bite"); }
} ///:~
```

Viene creato un pacchetto nella directory c05/dessert

Package access a **bite()**

Public

- Se in un'altra directory abbiamo

```
import c05.dessert.*;
public class Dinner {
public Dinner() {
   System.out.println("Dinner constructor");
  }
  public static void main(String[] args) {
    Cookie x = new Cookie();
    //! x.bite();
    }
} ///:~
```

Public

- Se in un'altra directory abbiamo

```
import c05.dessert.*;
public class Dinner {
public Dinner() {
   System.out.println("Dinner constructor");
  }
  public static void main(String[] args) {
    Cookie x = new Cookie();
    //! x.bite(); // Questo accesso è negato,
                  // a causa del package access a bite()!!
  }
} ///:~
```

Ok, perché il costruttore di **Cookie** è **public**

Il default package

- Come già visto, quando si crea un file **.java**, esso viene inserito automaticamente nel **default package** (a meno di uso della parola chiave **package**)
- Per questo motivo, il seguente esempio è del tutto corretto e funzionante

Il default package

- In un file definiamo

```
class Cake {
public static void main(String[] args) {
    Pie x = new Pie();
    x.f();
  }
} ///:~
```

Accesso a **f()**

Il default package

- In un altro file, ma nella stessa directory del precedente, definiamo

```
class Pie {
  void f() { System.out.println("Pie.f()"); }
} //
```

Package access: ok per le classi nello stesso pacchetto (in questo caso, **default package**)

Private

- Nessuno può accedere il membro dichiarato in questo modo, tranne la classe che contiene il membro
- Altre classi nello stesso pacchetto *non* possono accedere membri **private**
- Vediamo un esempio

Private

```
Class Gelato {
   private Gelato() {};
   static Gelato faiGelato() {
        return new Gelato();
   }
}
public class Cono {
   public static void main (String[] args) {
        //! Gelato x = new Gelato();
        Gelato x = Gelato.faiGelato();
   }
}
```

Private

```
Class Gelato {
   private Gelato() {};
   static Gelato faiGelato() {
        return new Gelato();
   }
}
public class Cono {
   public static void main (String[] args) {
        //! Gelato x = new Gelato();
        Gelato x = Gelato.faiGelato();
   }
}
```

Il costruttore non è accessibile direttamente, ma solo tramite il metodo **faiGelato()**

Private

Serve nei casi in cui si voglia controllare *come* viene creato un oggetto

```
Class Gelato {
   private Gelato() {};
   static Gelato faiGelato() {
        return new Gelato();
   }
}
public class Cono {
   public static void main (String[] args) {
        //! Gelato x = new Gelato();
        Gelato x = Gelato.faiGelato();
   }
}
```

Il costruttore non è accessibile direttamente, ma solo tramite il metodo **faiGelato()**

Protected

- Questa parola chiave ha a che fare con il concetto di ereditarietà
- Quando si crea una classe in un certo pacchetto **A** come sottoclasse di una classe in un altro pacchetto **B**, **B** ha normalmente accesso solo a cosa è **public** in **A**
- Però è possibile voler estendere l'accessibilità di alcune classi di **A** anche alle loro sottoclassi
  - In questo caso queste classi si dichiarano **protected**
    - Non è visibile a tutto il mondo, ma alle sottoclassi
    - **protected** fornisce anche *package access*, cioè altre classi nello stesso pacchetto possono accedere elementi **protected**

Interfaccia e implementazione

- Le principali motivazioni legate al controllo degli accessi sono
  - Stabilire chiaramente cosa l'utente di una libreria può e non può fare
  - *Separare* interfaccia e implementazione
    - Se l'unica cosa che può fare l'utente è mandare messaggi a interfacce **public**, allora è possibile modificare tutto quello che non è **public** senza timore di intaccare il codice utente

Accesso alle classi

- Gli specificatori di accesso possono essere usati anche per le classi
  - Specificano *quali* classi *all'interno* di una libreria sono disponibili agli utenti di quella libreria
- Gli specificatori vengono scritti prima della parola chiave **class**
- Ci sono però delle limitazioni

Accesso alle classi

- Ci può essere *una* sola classe **public** per ogni unità di compilazione (file)
  - Più di una classe **public** nello stesso file produce un errore in compilazione
  - L'idea è che ogni unità ha una singola interfaccia pubblica rappresentata da quella classe **public**
- Ci possono essere quante classi di supporto si vogliono (senza specificatori -- *package access*)

Accesso alle classi

- Il nome della classe **public** deve essere *esattamente* lo stesso del nome del file che la contiene
  - In caso contrario, si ottiene un errore in compilazione
- È possibile (anche se atipico) avere un file senza classi **public**
  - In questo caso, la classe (o il file) può avere qualsiasi nome

Accesso alle classi

- Tutte le classi che vengono utilizzate come appoggio da altre classi all'interno di un pacchetto, e che non devono essere usate direttamente dagli utenti, vengono definite senza specifcatore di accesso
  - Si ottiene *package access*

Accesso alle classi

- Una classe non può essere definita **private** o **protected**
  - Definirla **private** vorrebbe dire permettere l'accesso a quella classe alla sola classe stessa
- Per fare in modo che nessuno possa accedere alla classe, si possono rendere tutti i suoi costruttori private
  - Un oggetto viene creato tramite la definizione di un metodo statico all'interno della classe che restituisce un riferimento all'oggetto

Esempio -- accesso alle classi

```
class Soup {
  private Soup() {}
  // (1) Permette la creazione tramite un metodo statico
  public static Soup makeSoup() {
    return new Soup();
  }
  // (2) Crea un oggetto statico e restituisce
  // un riferimento quando richiesto
  private static Soup ps1 = new Soup();
  public static Soup access() {
    return ps1;
  }
  public void f() {}
}
```

Nome file: **Lunch.java**

Esempio -- accesso alle classi

```
class Soup {
  private Soup() {}
  // (1) Permette la creazione tramite un metodo statico
  public static Soup makeSoup() {
    return new Soup();
  }
  // (2) Crea un oggetto statico e restituisce
  // un riferimento quando richiesto
  private static Soup ps1 = new Soup();
  public static Soup access() {
    return ps1;
  }
  public void f() {}
}
```

Restituisce un riferimento a un oggetto **Soup**

Nome file: **Lunch.java**

Esempio -- accesso alle classi

```
class Soup {
  private Soup() {}
  // (1) Permette la creazione tramite un metodo statico
  public static Soup makeSoup() {
    return new Soup();
  }
  // (2) Crea un oggetto statico e restituisce
  // un riferimento quando richiesto
  private static Soup ps1 = new Soup();
  public static Soup access() {
    return ps1;
  }
  public void f() {}
}
```

Rendendo tutti i costruttori **private**, è possibile impedire la creazione diretta di oggetti **Soup**

Esempio -- accesso alle classi

```
class Soup {
  private Soup() {}
  // (1) Permette la creazione tramite un metodo statico
  public static Soup makeSoup() {
    return new Soup();
  }
  // (2) Crea un oggetto statico e restituisce
  // un riferimento quando richiesto
  private static Soup ps1 = new Soup();
  public static Soup access() {
    return ps1;
  }
  public void f() {}
}
```

La definizione di un costruttore senza argomenti è necessaria per evitare la definizione di quello di *default* (che non sarebbe **private**)

Esempio -- accesso alle classi

```
class Soup {
  private Soup() {}
  // (1) Permette la creazione tramite un metodo statico
  public static Soup makeSoup() {
    return new Soup();
  }
  // (2) Crea un oggetto statico e restituisce
  // un riferimento quando richiesto
  private static Soup ps1 = new Soup();
  public static Soup access() {
    return ps1;
  }
  public void f() {}
}
```

Per utilizzare la classe:
1. Metodo **static** che restituisce un riferimento a **Soup**

Esempio -- accesso alle classi

```
class Soup {
  private Soup() {}
  // (1) Permette la creazione tramite un metodo statico
  public static Soup makeSoup() {
    return new Soup();
  }
  // (2) Crea un oggetto statico e restituisce
  // un riferimento quando richiesto
  private static Soup ps1 = new Soup();
  public static Soup access() {
    return ps1;
  }
  public void f() {}
}
```

Per utilizzare la classe:
2. Creazione di un unico singolo (**static**) oggetto, accessibile tramite il metodo **public access()**

Questa opzione è detta *design pattern* (in questo caso il pattern è detto *singleton*)

Esempio -- accesso alle classi

```
class Sandwich { // Utilizza Lunch
  void f() { new Lunch(); }
}
// Solo una classe public nel file
public class Lunch {
  void test() {
    // Non possibile!! Costruttore private!!
    //! Soup priv1 = new Soup();
    Soup priv2 = Soup.makeSoup();
    Sandwich f1 = new Sandwich();
    Soup.access().f();
  }
} ///:~
```

Esercizi

- Definire una libreria che definisce le classi Quadrato e Rettangolo che hanno metodi per calcolare l'area di un quadrato e un rettangolo passati come argomento, rispettivamente
  - Ogni classe ha delle variabili che specificano la dimensione del lato. Esse non devono essere accessibili all'esterno direttamente.
- Creare un nuovo progetto in cui si utilizzano questi metodi

Java

nascondere l'implementazione

fine