Programmazione Java

Finalization di oggetti, Package, Modificatori di accesso

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

diruscio@di.univaq.it

Sommario

- Finalization di oggetti
- Package
- Modificatori di accesso
  - `private`
  - `protected`
  - `public`
  - `package`

Ancora sull'inizializzazione

- L'ordine di inizializzazione prevede:
  - prima i dati static, se non sono già stati inizializzati da una precedete creazione di oggetti o da un loro primo utilizzo,
  - seguiti poi dagli oggetti non statici

Ancora sull'inizializzazione

» Inizializzazione di dati static

– Quando i dati sono static, di tipo primitivo e non vengono inizializzati, allora assumo i valori iniziali primitivi standard. Se sono riferimenti ad oggetto il loro valore sara null

– C'è un unico segmento di memoria per un dato static, indipendentemente dal numero di oggetti creati

(vedere StaticInitialization.java)

» Inizializzazione statica esplicita

– Java permette di raggruppare altre inizializzazioni static in una classe all'interno di una speciale clausola *static { }*

(vedere ExplicitStatic.java)

Ancora sull'inizializzazione

» Inizializzazione di un'istanza non statica

– Java fornisce una sintassi simile per inizializzare le variabili non static di ciascun oggetto

(vedere Mugs.java)

Finalization di oggetti

» Java elimina automaticamente gli oggetti non più utilizzati mediante il garbage collector

» Un oggetto prima di essere eliminato ha l'opportunità di eliminare risorse non più utilizzate mediante la ridefinizione del metodo `finalize` della classe `Object`

» Tale processo viene detto *finalization*

» Tipicamente viene utilizzato quando ci sono risorse native che non sono sotto il controllo del garbage collector

» Java non specifica quando viene invocato tale metodo per cui non ci si può affidare all'invocazione di tale metodo per rimuovere risorse non più utilizzate

(vedere TerminationCondition.java)

Nascondere l'implementazione

- Java fornisce degli specificatori di accesso che permettono al progettista di una *libreria* di specificare ciò che è disponibile per il programmatore client e ciò che non lo è
- Il concetto di *libreria* non si esaurisce nell'elenco dei componenti e nella specifica di chi può accedere ad essi ma comprende anche il come i diversi componenti sono assemblati a formare la libreria
- Ciò in Java è realizzato mediante la parola chiave *package* e specificatori di accesso

Nascondere l'implementazione: package

- Un package è ciò che si rende disponibile utilizzando la parola chiave import per richiamare un'intera libreria
- Esempio:

  import java.util.*

  È possibile richiamare una singola classe di un package:

  import java.util.ArrayList

- Meccanismo per gestire lo "spazio dei nomi"

Nascondere l'implementazione: package

» Collezione di classi ed interfacce in relazione tra di loro che fornisce una protezione all'accesso e alla gestione del namespace

» Classi e interfacce che fanno parte della piattaforma Java sono raggruppati in package

» Pacchetti standard di java sono organizzati in modo gerarchico

» Livello più alto java e javax

» Esempi

> java.lang, java.io, java.net, java.awt, java.util

> javax.swing, javax.swing.border

Nascondere l'implementazione: package

» Package garantiscono l'univocità dei nomi delle classe e interfacce

» Nome completo di una classe ed interfaccia

> it.univaq.mwt.j2ee.Employee

> com.horstmann.corejava.Employee

» Sintassi

[package <nome_top_package>[.<nome_sub_package>]*;]

<dichiarazione classe e/o interfaccia>

Nascondere l’implementazione: package

» E’ necessario specificare la dichiarazione del package all’inizio del file sorgente

» Non è possibile avere più dichiarazioni in un file sorgente

» Se non è presente la dichiarazione del package allora la classe/interfaccia appartiene al package di default (o senza nome)

Nascondere l'implementazione: package

» Esempio

```
package graphics;
public class Circle extends Graphic implements Draggable {
. . .
}
```

Nome completo: graphics.Circle

```
package it.univaq.mwt.j2ee.graphics;
public class Circle extends Graphic implements Draggable {
. . .
}
```

Nome completo: it.univaq.mwt.j2ee.graphics.Circle

Nascondere l'implementazione: package

» Una classe può utilizzare tutte le classi contenute nello stesso pacchetto e tutte le classi/interfacce pubbliche di altri pacchetti

» E' possibile accedere alle classi pubbliche in due modi

> Nome completo

> Nome semplice e con keyword import

» java.lang sono importati per default

» Esempio

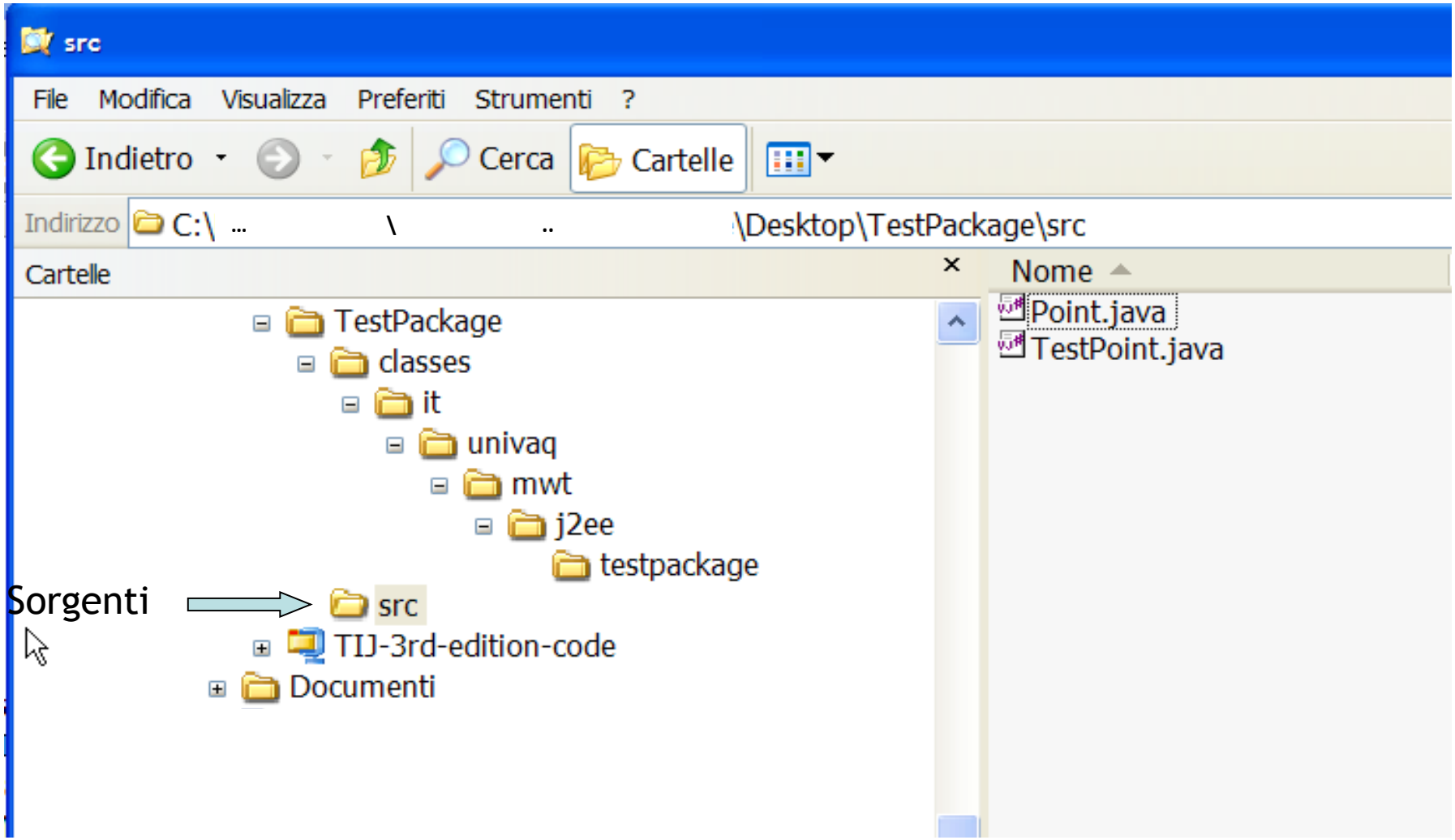
```
package graphics;
import java.awt.Component;
import java.util.*;
class Rectangle extends Component {
…
}
```

Layout sorgenti e destinazione (1)

» Se una classe è dichiarata all'interno di un package non è necessario che i sorgenti si trovino all'interno di una directory con lo stesso nome del package

» La struttura deve invece essere mantenuta per le classi compilate

Layout sorgenti e destinazione (2)



Layout sorgenti e destinazione (3)


```
package it.univaq.mwt.j2ee.testpackage;

public class Point {

     private int x,y;

     public Point(int x, int y) {

          this.x = x;

          this.y = y;

     }

     public String toString() {

               return "x= " + x + ", y= " + y;

  }

}
```

Layout sorgenti e destinazione (4)


```
import it.univaq.mwt.j2ee.testpackage.point;


public class TestPoint {


      public static void main(String[] args) {
            Point p = new Point(10, 20);
            System.out.println("Point " + p );
      }
}
```

Package e CLASSPATH

- Ad ogni package viene assegnata una directory sul filesystem
- Per eseguire un programma il classloader carica i .class di cui ha bisogno
- Il classloader procede come segue:
  - Localizza la variabile CLASSPATH che contiene il percorso di una o più directory che vengono utilizzate come radici per la ricerca di file .class
  - A partire dalla radice, l'interprete prende il nome del package e sostituisce ciascun "." con "\" per generare un nome di percorso dalla radice CLASSPATH
    - Es. Package foo.bar.baz diventa foo\bar\baz (o simile dipendentemente dal sistema operativo)

Modificatori di accesso

- **public, protected, private**
- Vengono posizionati all'inizio di ciascuna definizione di un membro, sia questo un campo o un metodo
- Al contrario del C++, ogni specificatore di accesso controlla l'accesso solo per quella particolare definizione

Modificatori di accesso: package (1)

- Se non si utilizza nessun specificatore di accesso, si ha un accesso di default (accesso a livello di package)
- Tutte le classi nel package corrente hanno accesso a quell'elemento, ma a tutte le classi fuori dal quel package l'elemento appare come private

Modificatori di accesso: package (2)

- Definito quando non si specifica nessun modificatore di accesso
- Livello di accesso è ammesso alle classi dello stesso package
- E' possibile *utilizzarlo* anche nella definizione di una classe → classe è visibile all'interno del package

(vedi A.java, A1.java, B.java, C.java)

Modificatori di accesso: package (3)


```
package Greek;

class Alpha {
    int iampackage;
    void packageMethod() {
        System.out.println("packageMethod");
    }
}

package Greek;

class Beta {
    void accessMethod() {
        Alpha a = new Alpha();
        a.iampackage = 10;      // legal
        a.packageMethod();      // legal
    }
}
```

Modificatori di accesso

» L'unico modo per garantire l'accesso a un elemento e quello di:

- Rendere l'elemento **public**, in questo caso ognuno, dovunque, può accedervi
- Rendere l'elemento accessibile a livello di package, tralasciando ogni specificatore di accesso, e mettendo le altre classi nello stesso package
- Una classe ereditata può accedere sia a un elemento **protected** sia a un elemento **public** ma non a elementi **private**
- Fornire metodi "accessor/mutator" ("get/set") che leggono e cambiano il valore dell'elemento

Modificatori di accesso: public (1)

- E’ il più semplice dei modificatori
- Chiunque può accedere alle variabili, metodi e costruttori dichiarati pubblici
- E’ possibile utilizzare `public` anche nella definizione di una classe → Indica che tale classe è visibile al di fuori del package

(vedere Dinner.java e Cookie.java)

Modificatori di accesso: public (2)


```
package Greek;

public class Alpha {
    public int iampublic;
    public void publicMethod() {
        System.out.println("publicMethod");
    }
}

package Roman;

import Greek.*;

class Beta {
    void accessMethod() {
        Alpha a = new Alpha();
        a.iampublic = 10;       // legal
        a.publicMethod();       // legal
    }
}
```

(vedere Alpha.java e Beta.java)

Modificatori di accesso: public (3)

» Tuttavia ci sono ulteriori limitazioni:

- Ci può essere una sola classe **public** per ogni unità di compilazione (file), altrimenti il compilatore restituirà un messaggio di errore
- Il nome della classe public deve corrispondere esattamente al nome del file contenente l'unità di compilazione, comprese le lettere maiuscole
- E' possibile, anche se non abituale, avere un'unità di compilazione senza alcuna classe **public**. In questo caso, si può dare al file il nome che si vuole

Modificatori di accesso: il package di default


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

}
```

Cake.java

Pie è disponibile in Cake.java perchè risiedono nella stessa directory e non hanno un package esplicito. In tale caso fanno parte implicitamente del “package di default”

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

}
```

Pie.java

Modificatori di accesso: private (1)

- Più restrittivo è `private`
- Può accedere ad un membro privato (variabile o metodo) soltanto la classe dove è definito
- Generalmente si dichiarano variabili o metodi privati che renderebbero l'oggetto in uno stato inconsistente
- E' possibile utilizzare tale modificatore anche in un costruttore

Modificatori di accesso: private (2)


```
class Alpha {
    private int iamprivate;
    private void privateMethod() {
        System.out.println("privateMethod");
    }
}

class Beta {
    void accessMethod() {
        Alpha a = new Alpha();
        a.iamprivate = 10;      // illegal
        a.privateMethod();      // illegal
    }
}
```

(vedere Lunch.java)
(vedere IceCream.java)

Modificatori di accesso: private (3)

- Qualsiasi metodo “sussidiario” per una classe può essere reso private, per essere sicuri di non utilizzarlo involontariamente altrove nel package
- Lo stesso vale per un campo privato di una classe

Modificatori di accesso: protected (1)

- Ha a che fare con il concetto di ereditarietà
- Se si crea un nuovo package e si eredità da una classe di un'altro package, i soli elementi ai quali si ha accesso sono gli elementi public del package originale
- Se si eredita all'interno dello stesso package, si possono manipolare tutti gli elementi accessibili a livello di package
- Talvolta il programmatore della classe base vorrebbe che un particolare elemento sia accessibile da tutte le classi derivate, ma non dal mondo in generale
- La keyword *protected* permette di fare questo

Modificatori di accesso: protected (2)

- Può accedere ad un membro protetto (variabile, metodo o costruttore) soltanto
  - Classe dove è definito
  - Sottoclassi
  - Classi che appartengono allo stesso package
- Generalmente è utilizzato quando si ha bisogno di far accedere ai membri alle sottoclassi e non a classi che non sono in relazione

(vedi ChocolateChip.java)

Modificatori di accesso: protected (3)


```
public class Point {
  protected int x,y;
}

public class Point3D extends Point {
  protected int z;

    public void move(int x, int y, int z) {
      this.x = x;                    //OK
      this.y = y;                    //OK
      this.z = z;                    //OK
    }
}
```

Modificatori di accesso: protected (4)


```
package Greek;

public class Alpha {
    protected int iamprotected;
    protected void protectedMethod() {
        System.out.println("protectedMethod");
    }
}

package Greek;

class Gamma {
    void accessMethod() {
        Alpha a = new Alpha();
        a.iamprotected = 10;    // legal
        a.protectedMethod();    // legal
    }
}
```

Modificatori di accesso: protected (5)


```
package Latin;

import Greek.*;

class Delta extends Alpha {
    void accessMethod(Alpha a, Delta d) {
        a.iamprotected = 10;    // illegal
        d.iamprotected = 10;    // legal
        a.protectedMethod();    // illegal
        d.protectedMethod();    // legal
    }
}
```

Esercizi

» Scrivere una classe Matrix che rappresenta le matrici di numeri interi. Tale classe ha:

- un costruttore che prende due interi e crea una matrice con la dimensione specificata;
- un metodo *getHeight* che restituisce il numero delle righe;
- un metodo *getWidth* che restituisce il numero delle colonne;
- un metodo *getVal* che prende due interi i e j e restituisce l'elemento $a_{i, j}$;
- un metodo *setVal* che prende tre interi i, j e v e assegna il valore v all'elemento $a_{i, j}$;
- un metodo *plus* che fa la somma di due matrici;
- un metodo *mult* che fa il prodotto di due matrici.

Esercizi

- Create una classe con campi e metodi public, private, protected e accessibili a livello di package. Create un oggetto di questa classe e verificate quali messaggi ricevete dal compilatore quando cercate di accedere a tutti i membri della classe
- Create una classe con un campo protected. Create una seconda classe nello stesso file con un metodo che manipoli il campo protected nella prima classe

Esercizi

» In base alla struttura dell'esempio Lunch.java, create una classe chiamata *ConnectionManager* che gestisca un array fisso di oggetti *Connection*. Il programmatore client non deve essere in grado di creare esplicitamente oggetti *Connection*, ma può ottenerli solo attraverso un metodo static in ConnectionManager. Quando il *ConnectionManager* rimane senza oggetti, restituisce un riferimento *null*. Collaudate le classi in main()