SAPIENZA Università di Roma, Facoltà di Ingegneria

Corso di

# Progettazione del Software - Esercitazioni

Canale A-L

Anno Accademico 2008-2009

Corso di Laurea in Ingegneria Informatica

Fabio Patrizi

# Informazioni

- Docente: Fabio Patrizi
- sito web: `www.dis.uniroma1.it/~patrizi` (seguire link *Didattica*)
- e-mail: `patrizi@dis.uniroma1.it`
- Ricevimento studenti:
  - Martedì, ore 11 - 13
  - Via Ariosto, 25 (Stanza B 213)

# Programma

- Ereditarietà in Java (Ripasso)
- Java Swing (librerie per GUI)
- Java 2D (librerie Java per grafica 2D)
- Eventi in Java e programmazione ad eventi
- Cenni di programmazione concorrente e librerie Java per la concorrenza

# Ereditarietà Java

Argomenti che tratteremo in questa parte di corso:

1. Package
2. Livello di accesso
3. Derivazioni tra classi ed ereditarietà
4. Classi astratte
5. Interfacce
6. Uguaglianza superficiale e uguaglianza profonda
7. Copia superficiale e copia profonda
8. Oggetti mutabili e oggetti immutabili

# Struttura di un programma Java

- Una *classe* è un aggregato di *campi*, che possono essere **dati**, **funzioni**, **classi**.

- La definizione di una classe è contenuta in un *file*, e un file contiene una o più definizioni di classi, una sola delle quali può essere `public`.

- Un *package* è una collezione di classi.

- Un file (con tutte le classi in esso contenute) appartiene ad uno ed un solo package.

- Un *programma* è una collezione di una o più classi, appartenenti anche a diversi package. Una di queste classi deve contenere la funzione che è il punto di accesso per l'esecuzione del programma (`main()`).

# Package

- Esistono nella libreria standard Java molti package (ad esempio `java.io`)

- Un **nuovo** package `mio_pack` viene dichiarato scrivendo all'inizio di un file `F.java` la dichiarazione:

```
package mio_pack;
```

- La stessa dichiarazione in un altro file `H.java`, dichiara che anche quest'ultimo appartiene allo stesso package.

- Se un file non contiene una dichiarazione di package, allora alle classi di tale file viene associato automaticamente un package di default, il cosiddetto *package senza nome*.

# Uso dei package

Se, in un file `G.java`, vogliamo usare una classe `C` definita nel package `mio_pack`, possiamo usare due metodi:

1. riferirci ad essa mediante `mio_pack.C` (oppure semplicemente `C`, se essa è definita nel package senza nome);

2. scrivere all'inizio del file `G.java` una delle seguenti dichiarazioni:

```
import mio_pack.C; // semplifica il riferimento alla classe C
                   // del package mio_pack
import mio_pack.*; // semplifica il riferimento a tutte le classi
                   // del package mio_pack
```

   A questo punto, possiamo riferirci alla classe mediante `C` (senza specificare esplicitamente che appartiene a `mio_pack`).

# Struttura dei package e dei direttori

Tutti i file relativi al package `mio_pack` devono risiedere in un **direttorio** dal nome `mio_pack`.

È possibile definire altri package con un nome del tipo `mio_pack.mio_subpack`.

In tal caso, tutti i file relativi al package `mio_pack.mio_subpack` devono risiedere in un **sottodirettorio** di `mio_pack` dal nome `mio_subpack`.

La dichiarazione `import mio_pack.*;` **non significa** che stiamo importando anche da `mio_pack.mio_subpack`.

Se desideriamo fare ciò, dobbiamo dichiararlo **esplicitamente** mediante la dichiarazione `import mio_pack.mio_subpack.*;`

# Esempio uso package

```
// File unita1/mio_package/C.java
package mio_package;

public class C {
  public void F_C() {
    System.out.println("Sono F_C()");
  }
}

// File unita1/Esempio1.java
// uso package

import mio_package.*;   // importo mio_package.*

public class Esempio1 {
  public static void main(String[] args) {
```

```
    C c = new C();
    c.F_C();
  }
}

//Nota: posso usare C definita in mio_package, usando il "nome corto"
```

# Esempio uso package

```
// File unita1/mio_package/C.java
package mio_package;

public class C {
  public void F_C() {
    System.out.println("Sono F_C()");
  }
}

// File unita1/Esempio2.java
// uso package

//Nota: non importo mio_package.*

public class Esempio2 {
  public static void main(String[] args) {
```

```
    mio_package.C c = new mio_package.C();
    c.F_C();
  }
}

//Nota: posso usare C definita in mio_package, ma devo usare il "nome lungo"
```

# Esempio uso package

```
// File unita1/mio_package/C.java
package mio_package;

public class C {
  public void F_C() {
    System.out.println("Sono F_C()");
  }
}

// File unita1/mio_package/mio_subpackage/D.java
package mio_package.mio_subpackage;
public class D {
  public void F_D() {
    System.out.println("Sono F_D()");
  }
}
```

```java
// File unita1/Esempio3.java
// uso package

import mio_package.*;
// Nota: non importo mio_package.mio_subpackage.*

public class Esempio3 {
  public static void main(String[] args) {
    C c = new C();
    c.F_C();
    mio_package.mio_subpackage.D d = new mio_package.mio_subpackage.D();
    d.F_D();
  }
}
```

# Esempio uso package

```
// File unita1/mio_package/C.java
package mio_package;

public class C {
  public void F_C() {
    System.out.println("Sono F_C()");
  }
}

// File unita1/mio_package/mio_subpackage/D.java
package mio_package.mio_subpackage;
public class D {
  public void F_D() {
    System.out.println("Sono F_D()");
  }
}
```

```java
// File unita1/Esempio4.java
// uso package

import mio_package.mio_subpackage.*;
// Nota: non importo mio_package.*

public class Esempio4 {
  public static void main(String[] args) {
    mio_package.C c = new mio_package.C();
    c.F_C();
    D d = new D();
    d.F_D();
  }
}
```

# Esercizio 1: package

```
// File unita1/Esempio5.java
import java.io.*;

public class Esempio5 {
  public static void main(String[] args) throws IOException {
    // stampa su schermo il file passato tramite linea di comando
    FileInputStream istream = new FileInputStream(args[0]);
    BufferedReader in = new BufferedReader(new InputStreamReader(istream));
    String linea = in.readLine();
    while(linea != null) {
      System.out.println(linea);
      linea = in.readLine();
    }
    in.close();
  }
}
```

Riscrivere il programma eliminando la dichiarazione `import java.io.*;`.

# Livelli di accesso di una classe

Una classe può essere specificata con **due** livelli di accesso:

1. `public`

2. non qualificato (è il *default*)

Se una classe è dichiarata `public` allora è accessibile fuori dal package al quale appartiene, altrimenti è accessibile solo all'interno del package al quale appartiene.

# Livelli di accesso dei campi di una classe

Un campo di una classe (dato, funzione o classe) può essere specificato con uno fra **quattro** livelli di accesso:

1. `public`,

2. `protected`,

3. `private`,

4. non qualificato (è il *default*, intermedio fra protetto e privato).

# Regole di visibilità tra campi

```
                          ==============================
                          IL METODO B VEDE IL CAMPO A ?
                          ==============================
METODO B \          +--------+-----------+-----------+---------+
 IN       \CAMPO A | public | protected | non qual. | private |
-----------\-------+--------+-----------+-----------+---------+
                    +--------+-----------+-----------+---------+
STESSA CLASSE       |   SI   |    SI     |    SI     |    SI   |  1 |
                    |        |           |           |         |    |
                    +--------+-----------+-----------+---------+    |
CLASSE STESSO       |   SI   |    SI     |    SI     |    NO   |  2 |   NOTA:
PACKAGE             |        |           |           |         |    | Decrescono
                    +--------+-----------+-----------+---------+    | i diritti
CLASSE DERIVATA     |   SI   |    SI     |    NO     |    NO   |  3 |
PACKAGE DIVERSO     |        |           |           |         |    |
                    +--------+-----------+-----------+---------+    |
CL. NON DERIVATA    |   SI   |    NO     |    NO     |    NO   |  4 |
PACKAGE DIVERSO     |        |           |           |         |    V
                    +--------+-----------+-----------+---------+    V

                     ----------------------------------------->>
                             NOTA: Decrescono i diritti
```

# Commento sulle regole di visibilità

Le regole di visibilità vengono sfruttate per aumentare l'*information hiding*.

Ricordiamo che uno dei princìpi di buona modularizzazione è che l'information hiding deve essere **alto**.

In base a questo principio, i campi dati **non sono mai pubblici**, ma **privati** o **protetti**.

In tal modo diamo al cliente un **accesso controllato** ai campi dati, mediante le funzioni pubbliche.

# Visibilità: esempio

```
// File unita1/Esempio6.java

class C {
  private int x, y;
  public C(int a, int b) { x = a; y = b; }
  public void stampa() { System.out.println("x: " + x + ", y: " + y); }
}

class Esempio6 {
  public static void main(String[] args) {
    C c = new C(7,12);    // OK: il costruttore di C e' pubblico
    c.stampa();           // OK: la funzione stampa() di C e' pubblica
  //    int val = c.x;    // NO: il campo x e' privato in C
  //                ^
  //Variable x in class C not accessible from class Esempio6.
  }
}
```

# Esercizio 2: visibilità

Verificare, tramite opportuni frammenti di codice, la veridicità delle regole di visibilità della tabella vista in precedenza.

# Regole di visibilità (cont.)

Il seguente "albero delle decisioni" fa notare che essere nello stesso package **dà più diritti** di essere una classe derivata.

```
Stessa classe?                /\
                         SI  /  \ NO
                         1      \
Stesso package?                 /\
                            SI /  \ NO
                            2      \
Classe derivata?                    /\
                               SI /  \ NO
                               3      4
```

Va inoltre ricordato che ogni package è "aperto", ovvero possiamo sempre dichiarare di fare parte di un package qualunque.

# Derivazione fra classi

È possibile dichiarare una classe `D` come *derivata* da una classe `B`.

```
class B {                    // CLASSE BASE
  int x;
  void G() { x = x * 20; }
// ...
}

class D extends B {          // CLASSE DERIVATA
  void H() { x = x * 10; }
// ...
}
```

# Princìpi fondamentali della derivazione

I quattro **princìpi fondamentali** del rapporto tra classe derivata e classe base:

1. Tutte le proprietà definite per la classe base vengono **implicitamente definite** anche nella classe derivata, cioè vengono **ereditate** da quest'ultima.

   Ad esempio, implicitamente la classe derivata `D` ha:
   - un campo dati `int x;`
   - una funzione `void G()`

2. La classe derivata può avere **ulteriori proprietà** rispetto a quelle ereditate dalla classe base.

   Ad esempio, la classe `D` ha una funzione `void H()`, in più rispetto alla classe base `B`.

# Princìpi fondamentali della derivazione (cont.)

3. Ogni oggetto della classe derivata **è anche** un oggetto della classe base.

   Ciò implica che è possibile usare un oggetto della classe derivata **in ogni situazione o contesto** in cui si può usare un oggetto della classe base.

   Ad esempio, i seguenti usi di un oggetto di classe `D` sono leciti.

```
  static void stampa(B bb) {
    System.out.println(bb.x);
  }
//...
D d = new D();
d.G();     // OK: uso come ogg. di invocazione di funz. definita in B
stampa(d); // OK: uso come argomento in funz. definita per B
```

*La classe* `D` *è* compatibile *con la classe* `B`

# Princìpi fondamentali della derivazione (cont.)

4. **Non è vero che** un oggetto della classe base è anche un oggetto della classe derivata.

   Ciò implica che **non è possibile** usare un oggetto della classe base laddove si può usare un oggetto della classe derivata.

```
B b = new B();
// b.H();
//    ^
// Method H() not found in class B.

// d = b;
//   ^
// Incompatible type for =. Explicit cast needed to convert B to D.

b = d;     // OK: D al posto di B
```

# Esercizio 3: derivazione

Ignorando i costruttori e i livelli d'accesso ai campi, riscrivere la classe `Segmento`, equipaggiandola con una funzione (interna) `stampa()`.

Scrivere una classe `SegmentoOrientato` derivata dalla classe `Segmento`, che contiene anche l'informazione sull'orientazione del segmento (dal punto di inizio a quello di fine, o viceversa).

Verificare se:

- le funzioni esterne precedentemente definite con argomenti di classe `Segmento` (ad es. `lunghezza()`) possano essere usate anche con argomenti di classe `SegmentoOrientato`;

- sia possibile usare la funzione `stampa()` con oggetti di invocazione di classe `SegmentoOrientato`.

# Derivazione e regole di visibilità

Una classe `D` derivata da un'altra classe `B`, anche se in un package diverso, ha una relazione particolare con quest'ultima:

- **non è un cliente qualsiasi** di `B`, in quanto possiamo usare oggetti di `D` al posto di quelli di `B`;

- **non coincide** con la classe `B`.

Per questo motivo, è possibile che `B` voglia mettere a disposizione dei campi (ad esempio i campi dati) solo alla classe `D`, e non agli altri clienti. In tal caso, questi campi devono essere dichiarati **protetti** (e non privati).

Ciò garantisce al progettista di `D` di avere accesso a tali campi (vedi tabella delle regole di visibilità), **senza tuttavia garantire tale accesso ai clienti generici** di `B`.

# Costruttori di classi derivate

Al momento dell'invocazione di un costruttore della classe derivata, se il costruttore della classe derivata non contiene esplicite chiamate al costruttore della classe base (vedi dopo), viene chiamato **automaticamente** anche il costruttore senza argomenti della classe base. Ciò avviene:

- sia se la classe base ha il costruttore senza argomenti standard,
- sia se la classe base ha il costruttore senza argomenti definito esplicitamente,
- sia se la classe non ha il costruttore senza argomenti(!), in questo caso si ha un errore di compilazione.

```
class B { ... }
```

```
class D extends B { ... }

...
D d = new D(); // invoca il costruttore senza argomenti di B()
               // e quello di D()
```

# Costruttori di classi derivate (cont.)

Il costruttore senza argomenti della classe base viene invocato:

- **anche se non definiamo** alcun costruttore per la classe derivata (che ha quindi quello standard senza argomenti),

- **prima** del costruttore della classe derivata (sia quest'ultimo definito esplicitamente oppure no).

# Costruttori di classi derivate: esempio

```
// File unita1/Esempio9.java

class B1 { protected int x_b1; }
class D1 extends B1 { protected int x_d1; } // OK: B1 ha cost. senza arg.

class B2 {
  protected int x_b2;
  public B2() { x_b2 = 10; }
}
class D2 extends B2 { protected int x_d2; } // OK: B2 ha cost. senza arg.

class B3 {
  protected int x_b3;
  public B3(int a) { x_b3 = a; }
}
// class D3 extends B3 { protected int x_d3; } // NO: B3 NON ha c. senza arg.
//        ^
// No constructor matching B3() found in class B3.
```

# Costruttori di classi derivate: uso di super()

Se la classe base ha costruttori con argomenti, è probabile che si voglia **riusarli**, quando si realizzano le classi derivate.

È possibile invocare esplicitamente un costruttore qualsiasi della classe base **invocandolo**, nel corpo del costruttore della classe derivata.

Ciò viene fatto mediante il costrutto `super()`, che deve essere la *prima istruzione eseguibile* del corpo del costruttore della classe derivata.

# Uso di super() nei costruttori: esempio

```
// File unita1/Esempio10.java
class B {
  protected int x_b;
  public B(int a) {  // costruttore della classe base
    x_b = a;
  }
}
class D extends B {
  protected int x_d;
  public D(int b, int c) {
    super(b);       // RIUSO del costruttore della classe base
    x_d = c;
  }
}
class Esempio10 {
  public static void main(String[] args) {
    D d = new D(3,4);  } }
```

# Costruttori di classi derivate: riassunto

Comportamento di un costruttore di una classe `D` derivata da `B`:

1. **se** ha come prima istruzione `super()`, allora viene chiamato il costruttore di `B` esplicitamente invocato;

   **altrimenti** viene chiamato il costruttore senza argomenti di `B`;

2. viene eseguito il corpo del costruttore.

Questo vale **anche per il costruttore standard** di `D` senza argomenti (come al solito, disponibile se e solo se in `D` non vengono definiti esplicitamente costruttori).