Linguaggi di Programmazione I – Lezione 12

Prof. Marcello Sette
mailto://marcello.sette@gmail.com
http://sette.dnsalias.org

20 maggio 2008

Modificatori

- I modificatori sono parole riservate che danno al compilatore informazioni sulla natura del codice, dei dati, delle classi.
- Un gruppo di modificatori, detti *di accesso*, specificano a quali classi è permesso usare quella caratteristica.
- Altri modificatori possono essere usati, in combinazione con i precedenti, per qualificare quella caratteristica.
- Per ora descriveremo come i modificatori si applicano alle classi top-level. Le classi `inner` saranno discusse in una successiva lezione.

Modificatori di accesso 3 / 39

Generalità

- Una *caratteristica* di una classe può essere:
  - La classe stessa.
  - Un attributo (variabile) della classe.
  - Un metodo o un costruttore della classe.
- Le sole variabili che possono essere controllate con i modificatori di accesso sono gli attributi (variabili di heap) e non le variabili dei metodi (variabili di stack): una variabile di un metodo può essere vista solo dal metodo in cui si trova.
- I modificatori di accesso sono:
  - `public`
  - `protected`
  - `private`
- Il solo modificatore di accesso permesso per una classe top-level è `public`: non esistono classi top-level `protected` o `private`.
- Una caratteristica può avere al più un modificatore di accesso.
- Se non è presente il modificatore, si intende "accesso consentito dallo stesso pacchetto in cui è situata qualla caratteristica".

`public`

- È il modificatore più generoso. Accesso consentito a tutte le altre classi.
- Attenzione: l'accesso ad un attributo o un metodo `public` è subordinato all'accesso alla classe che lo contiene.

`private` (1)

- È il modificatore meno generoso.
- Può essere usato solo per attributi o metodi, non per classi top-level.

```
public class Complesso {
  private double reale, immag;

  public Complesso(double r, double i) {
    reale=r; immag=i;
  }
  public Complesso add(Complesso c) {
    return new Complesso(reale + c.reale,
                         immag + c.immag);
  }
}

public class Cliente {
  void usali() {
    Complesso c1 = new Complesso(1, 2);
    Complesso c2 = new Complesso(3, 4);
    Complesso c3 = c1.add(c2);
    double d = c3.reale;                // Illegale
  }
}
```


`private` (2)

- Le variabili private possono essere nascoste anche allo stesso oggetto che le possiede.

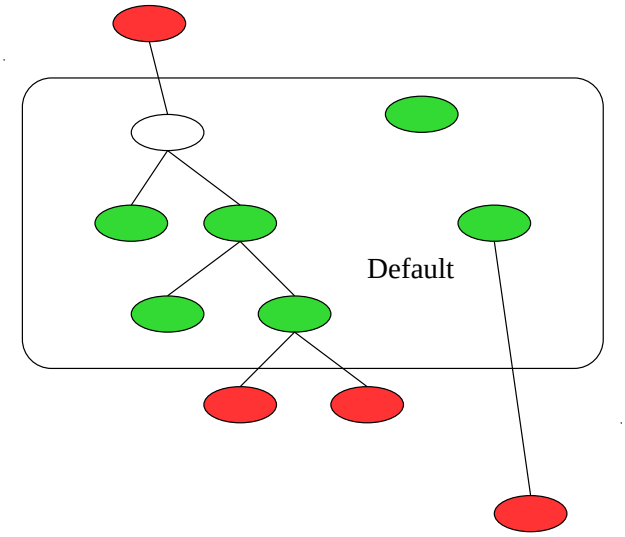
```
class Complesso {
  private double reale, immag;
}

class SubComplesso extends Complesso {
  SubComplesso(double r, double i) {
    reale = r;                          // Illegale
  }
}
```

- La classe `SubComplesso` eredita gli attributi della superclasse, MA quegli attributi possono essere usati solo dal codice della classe `Complesso`.
- Per quanto riguarda la classe `SubComplesso`, è come se quegli attributi non li avesse ereditati affatto.
- Tutto quanto detto è valido anche per metodi privati; i costruttori, invece, non sono mai ereditati.

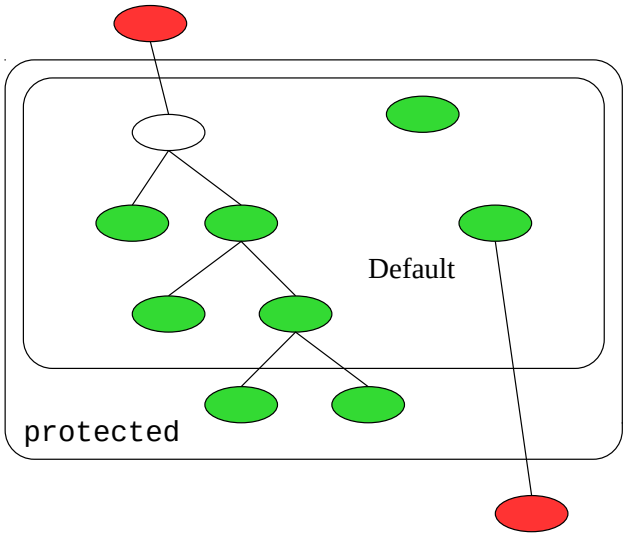
Default

- Non è una parola riservata: si intende accesso di default quando non è presente nessun modificatore di accesso.
- Accesso consentito a tutte le classi nello stesso pacchetto.



`protected` (1)

- Accesso consentito a tutte le classi nello stesso pacchetto e
- a tutte le sottoclassi in pacchetti diversi, ma solo per ereditarietà: una sottoclasse in un altro pacchetto può accedere ad un membro `protected` nella superclasse solo attraverso un riferimento ad un oggetto del proprio tipo (anche all'oggetto corrente, mediante `this`, o, esplicitamente, alla sua parte ereditata, mediante `super`) o di un sottotipo.



`protected` (2)

Esempio:

```
package p1;

public class A {
  protected int x = 7;
}
```

```
package p2;
import p1.A;

public class B extends A {
  public void testProt() {
    System.out.println(x); // Ok: x e' ereditato
    // uso implicito di this
    A a = new A();
    System.out.println(a.x); // Errore di comp.
    // a non e' di tipo B, ne' di sottotipo di B.
    B b = new B();
    System.out.println(b.x); // Ok: accesso
    // attraverso oggetto di tipo B.
  }
}
```


Overriding

Lo dico ancora:

- I metodi visibili di una superclasse non possono essere sovrapposti in una sottoclasse da altri meno visibili.
- **I metodi visibili di una superclasse non possono essere sovrapposti in una sottoclasse da altri meno visibili.**
- **I metodi visibili di una superclasse non possono essere sovrapposti in una sottoclasse da altri meno visibili.**

Altri modificatori

`final`

- Si applica a classi, metodi e variabili (non a costruttori).
- Il significato varia da contesto a contesto, ma l'essenza è: una caratteristica `final` non può essere modificata.
  - Una classe `final` non può essere estesa, cioè non può avere sottoclassi.
  - Una variabile `final` è praticamente una costante, cioè può solo essere inizializzata. Attenzione: un riferimento `final` ad un oggetto non può essere modificato (cioè riassegnato ad un altro oggetto), ma l'oggetto a cui esso fa riferimento si.
  - Un metodo `final` non può essere sovrapposto in una sottoclasse.
  - Non ha senso dire `final` ad un costruttore, perché esso non è mai ereditato dalle sottoclassi, quindi mai sovrapponibile.

`abstract`

- Si applica solo a classi e a metodi.
- Un metodo `abstract` non possiede corpo ("`;`" invece di "`{...}`"):

```
abstract void getValore();
```

- Una classe PUÒ essere marcata `abstract`. In questo caso il compilatore suppone che essa contenga (anche per ereditarietà) metodi `abstract`: essa non potrà essere istanziata, anche se non contiene affatto metodi `abstract`.
- Una classe DEVE essere marcata `abstract` se:
  - essa contiene almeno un metodo `abstract`, oppure
  - essa eredita almeno un metodo `abstract` per il quale non fornisce una realizzazione, oppure
  - essa dichiara di implementare una interfaccia [vedremo in seguito], ma non fornisce una realizzazione di tutti i metodi di quell'interfaccia.
- In un certo senso, `abstract` è opposto a `final`: una classe `final`, per esempio, non può essere specializzata; una classe `abstract` esiste solo per essere specializzata.

`static`

- Si applica ad attributi, metodi ed anche a blocchi di codice che non fanno parte di metodi.
- In generale, una caratteristica `static` appartiene alla classe, non alle singole istanze: essa è unica, indipendentemente dal numero (anche zero) di istanze di quella classe.

- **Attributi** `static`

- L'inizializzazione di un attributo `static` avviene nel momento in cui la classe viene caricata in memoria (anche se non esisterà mai nessuna istanza di quella classe).

```
class Ecstatic {
  static int x = 0;
  Ecstatic () { x++; }
}
```

- L'accesso ad un attributo `static` di una classe può avvenire (con la dot-notation):
  - o partendo da un riferimento ad una istanza di quella classe,
  - o partendo dal nome stesso della classe.

```
System.out.println(Ecstatic.x);
Ecstatic e = new Ecstatic();
e.x = 100;
Ecstatic.x = 100;
```


- **Metodi** `static`

- Esistono nel momento in cui la classe viene caricata in memoria (anche senza istanze).
- Non possono accedere a membri non `static` della stessa classe (perché potrebbero anche non esistere).
- Si veda il metodo `main`.

```
class Boo {
  static int i = 48;
  int j = 1;

  public static void main {String args[]) {
    i += 100;
    // j *= 5; Per fortuna e' commentata
  }
}
```

- Non possono essere sovrapposti da metodi non-`static`.
- Non possono sovrapporre metodi non-`static`.

- **Blocchi** `static`

- È lecito che una classe contenga blocchi di codice marcati `static`.
- Tali blocchi sono eseguiti una sola volta, nell'ordine in cui compaiono, quando la classe viene caricata in memoria ed, ovviamente, possono accedere (come i metodi `static`) solo a caratteristiche `static`.

```
public class EsempioStatic {
  static double d=1.23;

  static {
    System.out.println("Codice static: d=" + d++);
  }

  public static void main(String[] args) {
    System.out.println("main: d=" + d++);
  }

  static {
    System.out.println("Codice static: d=" + d++);
  }
}
```


`native`

- Si applica solo a metodi.
- Come per `abstract`, `native` indica che il corpo di un metodo deve essere trovato altrove, in questo caso all'esterno della JVM, in una libreria di codice dipendente dalla architettura fisica.

`transient`

- Si applica solo ad attributi.
- Indica che quell'attributo contiene informazioni sensibili e che, nel caso in cui lo stato interno dell'oggetto debba essere salvato (nel gergo "serializzato"), il valore di quell'attributo non dovrà essere considerato.

`synchronized`

- Si applica solo a metodi o a blocchi anonimi.
- Controlla l'accesso al codice in programmi multi-thread.

`volatile`

- Si applica solo ad attributi.
- Avverte il compilatore che quell'attributo può essere modificato in modo asincrono, in architetture multiprocessore.

Sommario

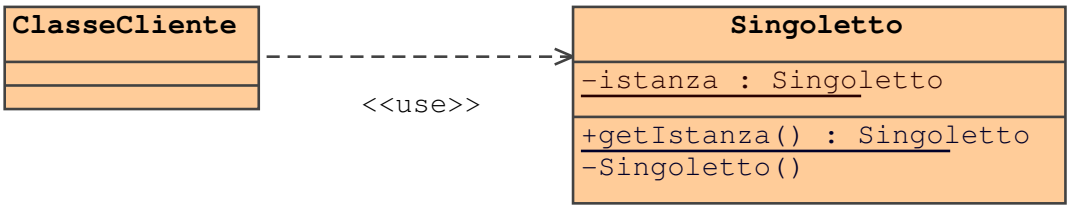
Modificatore	Classe	Attributo	Metodo	Costruttore	Blocco
`public`	√	√	√	√	
`protected`		√	√	√	
"default"	√	√	√	√	√
`private`		√	√	√	
`final`	√	√	√		
`abstract`	√		√		
`static`		√	√		√
`native`			√		
`transient`		√			
`volatile`		√			
`synchronized`			√		√

Forme ricorrenti

- *Forme progettuali ricorrenti* (Design Patterns) sono soluzioni a problemi ricorrenti nella progettazione OO.
- Visitare http://hillside.net/patterns per maggiori informazioni.

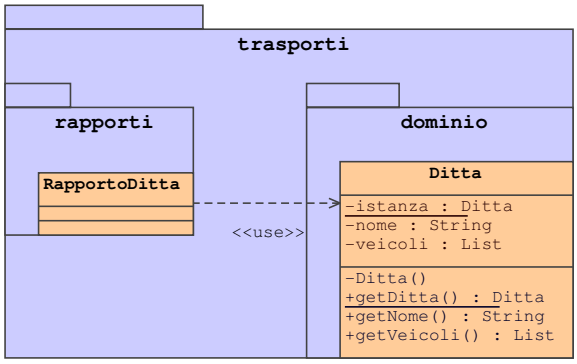
Singoletto (1)

- Uno dei problemi ricorrenti è il seguente: si vuole che una classe abbia una sola istanza.
- Se si scrivesse la classe nel modo solito, gli utilizzatori potrebbero creare istanze a volontà.
- Lo scopo della forma progettuale *Singleton* è quello di assicurare il progettista della classe che esisterà sempre una ed una sola istanza di quella classe.
- La forma è la seguente:



Nota: In UML le componenti `static` in una classe sono sottolineate. L'associazione tratteggiata <<use>> descrive non un attributo della `ClasseCliente` di tipo `Singoletto`, ma un uso diretto della classe `Singoletto`.

Singoletto (2)




```
package trasporti.rapporti;

import trasporti.dominio.*;

public class rapportoDitta {
  public void generaTesto
      (PrintStream output) {
    Ditta d = Ditta.getDitta();
    // usa l'oggetto Ditta per
    // reperire e stampare le
    // info della Ditta.
  }
}
```

```
package trasporti.dominio;

public class Ditta {
  private static Ditta istanza =
      new Ditta();
  private String nome;
  private Veicolo[] flotta;

  public static Ditta getDitta() {
    return istanza;
  }

  private Ditta() {...}

  // ulteriore codice
}
```


Esercizi

Esercizi

1. Modificare la classe `Banca` in modo che realizzi un singoletto.
2. È possibile modificare la visibilità `protected` di `saldo` nella classe `Conto` in modo che sia nuovamente `private` e ripristinare così l'incapsulazione perfetta?
   La risposta è si. Ma questo ha conseguenze nel codice della classe `ContoCorrente`.
   Esibire le necessarie (ed eleganti) modifiche nella classe `ContoCorrente` che rendano nuovamente il programma funzionante.

D 1

Quale dei seguenti frammenti viene correttamente compilato e stampa "Uguale" in esecuzione?

A.
```
Integer x = new Integer(100);
Integer y = new Integer(100);
if (x == y) {
  System.out.println("Uguale");
}
```

B.
```
int x=100;
Integer y = new Integer(100);
if (x == y) {
  System.out.println("Uguale");
}
```

C.
```
int x=100; float y=100.0F;
if (x == y) {
  System.out.println("Uguale");
}
```

D.
```
String x = new String("100");
String y = new String("100");
if (x == y) {
  System.out.println("Uguale");
}
```

E.
```
String x = "100";
String y = "100";
if (x == y) {
  System.out.println("Uguale");
}
```


D 2

Quali delle seguenti dichiarazioni sono illegali?

A. `default String s;`
B. `transient int i = 41;`
C. `public final static native int w();`
D. `abstract double d;`
E. `abstract final double cosenoIperbolico();`

D 3

Quale delle seguenti affermazioni è vera?

A. Una classe `abstract` non può avere metodi `final`.
B. Una classe `final` non può avere metodi `abstract`.

D 4

Qual è la minima modifica che rende il seguente codice compilabile?

```
 1. final class Aaa {
 2.   int xxx;
 3.   void yyy() { xxx=1; }
 4. }
 5.
 6. class Bbb extends Aaa {
 7.   final Aaa fref = new Aaa();
 8.   final void yyy() {
 9.     System.out.println(
10.       "In yyy()");
11.     fref.xxx = 12345;
12.   }
13. }
```

A. Alla linea 1, rimuovere `final`.
B. Alla linea 7, rimuovere `final`.
C. Rimuovere la linea 11.
D. Alle linee 1 e 7, rimuovere `final`.
E. Nessuna modifica è necessaria.

D 5

Riguardo al codice seguente, quale affermazione è vera?

```
 1. class Roba {
 2.   static int x = 10;
 3.   static { x += 5; }
 4.
 5.   public static void main(String[] args) {
 6.     System.out.println("x=" + x);
 7.   }
 8.
 9.   static { x /= 5; }
10. }
```

A. Le linee 3 e 9 non sono compilate, poiché mancano i nomi di metodi e i tipi ri ritorno.
B. La linea 9 non è compilata, poiché si può avere solo un blocco top-level `static`.
C. Il codice viene compilato e l'esecuzione produce `x=10`.
D. Il codice viene compilato e l'esecuzione produce `x=15`.
E. Il codice viene compilato e l'esecuzione produce `x=3`.

D 6

Rispetto al codice seguente, quale affermazione è vera?

```
 1. class A {
 2.   private static int x=100;
 3.
 4.   public static void main(
 5.       String[] args); {
 6.     A hs1 = new A();
 7.     hs1.x++;
 8.     A hs2 = new A();
 9.     hs2.x++;
10.     hs1 = new A();
11.     hs1.x++;
12.     A.x++;
13.     System.out.println(
14.       "x = " + x);
15.   }
16. }
```

A. La linea 7 non compila, poiché è un riferimento `static` ad una variabile `private`.

B. La linea 12 non compila, poiché è un riferimento `static` ad una variabile `private`.

C. Il programma viene compilato e stampa `x = 102`.

D. Il programma viene compilato e stampa `x = 103`.

E. Il programma viene compilato e stampa `x = 104`.

D 7

Dato il codice seguente:

```
1. class SuperC {
2.   void unMetodo() { }
3. }
4.
5. class SubC extends SuperC {
6.   void unMetodo() { }
7. }
```

1. Quali modificatori di accesso possono essere legalmente dati ad `unMetodo` alla linea 2, lasciando il resto del codice inalterato?
2. Quali modificatori di accesso possono essere legalmente dati ad `unMetodo` alla linea 6, lasciando il resto del codice inalterato?

D 8

```
 1. package abcd;
 2.
 3. public class SupA {
 4.   protected static int count=0;
 5.   public SupA() { count++; }
 6.   protected void f() {}
 7.   static int getCount() {
 8.     return count;
 9.   }
10. }
```

```
 1. package abcd;
 2.
 3. class A extends abcd.SupA {
 4.   public void f() {}
 5.   public int getCount() {
 6.     return count;
 7.   }
 8. }
```

Riguardo ai codici a sinistra, quale affermazione è vera?

A. La compilazione di `A.java` fallisce alla linea 4, poiché il metodo `f()` è `protected` nella superclasse e `A` è nello stesso pacchetto di `SupA`.

B. La compilazione di `A.java` fallisce alla linea 4, poiché il metodo `f()` è `protected` nella superclasse e `public` nella sottoclasse.

C. La compilazione di `A.java` fallisce alla linea 5, poiché il metodo `getCount()` è `static` nella superclasse e non può essere sovrapposto da un metodo non-`static`.

D. I codici sono compilati, ma viene lanciata una eccezione quando viene invocato il metodo `f()` su una istanza di `A`.

E. I codici sono compilati, ma viene lanciata una eccezione quando viene invocato il metodo `getCount` su una istanza di `SupA`.

D 9

Riguardo ai codici seguenti, quale affermazione è vera?

```
 1. package abcd;
 2.
 3. public class SupA {
 4.   protected static int count=0;
 5.   public SupA() { count++; }
 6.   protected void f() {}
 7.   static int getCount() {
 8.     return count;
 9.   }
10. }
```

```
 1. package ab;
 2.
 3. class A extends abcd.SupA {
 4.   A() { count++; }
 5.
 6.   public static void main(
 7.       String[] args) {
 8.     System.out.print(
 9.       "Prima:" + count);
10.     A a = new A();
11.     System.out.println(
12.       " Dopo:" + count);
13.     a.f();
14.   }
15. }
```

A. Il programma viene compilato e stampa: `Prima:0 Dopo:2`

B. Il programma viene compilato e stampa: `Prima:0 Dopo:1`

C. La compilazione di `A` fallisce alla linea 4.

D. La compilazione di `A` fallisce alla linea 13.

E. Il programma viene compilato, ma viene lanciata una eccezione alla linea 13.

D 10

Si consideri la classe seguente:

```
1. public class Test1 {
2.   public float unMetodo(float a, float b) {
3.   }
4.
5. }
```

Quali dei seguenti metodi può lecitamente essere inserito alla linea 4?

A. `public int unMetodo(int a, int b) { }`

B. `public float unMetodo(float a, float b) { }`

C. `public float unMetodo(float a, float b, int c) { }`

D. `public float unMetodo(float c, float d) { }`

E. `private float unMetodo(int a, int b, int c) { }`