Modificatori di visibilità

- Attenzione nell'overriding dei metodi: la restrizione di visibilità può essere **allentata**, ma mai resa più restrittiva.
- In particolare non si può ridefinire come privato un metodo che era inizialmente pubblico.

```
public class classeBase
{ public int operazione() { ... }
}
public class classeEstesa extends ClasseBase
{ private int operazione() { ... } //ERRORE!
}
```

Modificatori di visibilità

- Il compilatore segnala errore perché dare visibilità ridotta al metodo **`operazione()`** di **`ClasseEstesa`** è illusorio.

```
this.operazione();    //PRIVATO, ma...
super.operazione();   //PUBBLICO
```

- Il metodo ridefinito non può essere invocato da altre classi, quello di partenza resta pubblico.
  - Si viola il principio per cui un'istanza della sottoclasse è anche istanza della superclasse (e quindi può fare più cose, non meno).

Modificatore `protected`

- Le sottoclassi ereditano dalle superclassi, ma sono classi distinte.
- Quindi le sottoclassi non possono interagire con i membri privati della loro superclasse.
- Tuttavia, oltre ai modificatori **`public`** e **`private`**, Java consente anche l'uso del modificatore **`protected`**.
- Il modificatore **`protected`** serve a consentire l'accesso alle sottoclassi.

Modificatore `protected`

- Il modificatore **`protected`** corrisponde a fornire visibilità package-friendly (come se non fosse messo alcun modificatore).
- Non si può applicare alla classe nel complesso, ma solo a un membro.
- All'interno del package, questo membro è visibile a tutte le classi del package.
- All'esterno, è visibile solo alle sottoclassi che estendono la classe di cui è membro.

Modificatore `protected`

- In altre parole, un membro dichiarato **`protected`** ha un livello di protezione intermedio tra pubblico e package-friendly
  - possiede sicuramente visibilità package-friendly
  - è visibile da istanze della stessa classe
  - però è visibile anche da classi di altri package se estendono quella in cui è definito
- Se il membro in questione è un metodo, se ne può fare l'override solo definendo il nuovo metodo come **`public`** o **`protected`**
  - non si può **restringere** la visibilità, solo ampliarla

Modificatore `protected`

- Se un **costruttore** è **`protected`**, può essere invocato come **`super()` solo** dai costruttori delle sottoclassi che estendono **direttamente** la sua classe.
- Se un **metodo** o un **campo** sono **`protected`**, possono essere acceduti da una sottoclasse (anche non diretta). Però l'oggetto di cui vengono invocati come membri dev'essere di un livello non superiore alla sottoclasse (ad esempio, la superclasse non va bene)

Modificatore `protected`

- Esempio

```
package mioPackage;

public class ClasseBase
{ protected int x;
  protected ClasseBase(int x) { this.x = x; }
  protected String mVisibile()
  { return "metodo visibile!"; }
}
```

Modificatore `protected`

- Uso corretto

```
import mioPackage.*;
class ClasseEstesa extends ClasseBase
{ public ClasseEstesa(int x) { super(x); }
  public void usaProtetto()
  { System.out.println("x vale: " + x;
    System.out.println( mVisibile() );
  }
  public static void main(String[] args)
  { ClasseEstesa e = new ClasseEstesa(3);
    e.usaProtetto();
  }
}
```

costrut tore

metodo

Modificatore `protected`

- Uso **non corretto**

```
import mioPackage.*;
class ClasseEstesa extends ClasseBase
{ public ClasseEstesa(int x) { super(x); }
  public void usaProtetto()
  { System.out.println("x vale: " + x;
    System.out.println( mVisibile() );
  }
  public static void main(String[] args)
  { ClasseBase b = new ClasseBase(3);
    e.usaProtetto();
  }
}
```

costruttore

non si può usare qua, ma solo nel costruttore come **`super()`**

Modificatore `protected`

- Uso **non corretto**

```
import mioPackage.*;
class ClasseEstesa extends ClasseBase
{ public ClasseEstesa(int x) { super(x); }
  public void usaProtetto(ClasseBase b)
  { System.out.println("x vale: " + b.x ;
    System.out.println( b.mVisibile() );
  }
  public static void main(String[] args)
  { ClasseEstesa e = new ClasseEstesa(3);
    e.usaProtetto(e);
  }
}
```

metodo

sono errati perché usano un oggetto della superclasse (prima invece si usava **this**)

Regole sulla visibilità

- Il problema di accedere un campo della superclasse si può risolvere ponendolo **`protected`**. Ad esempio potevamo scrivere:

```
class BankAccount
{ protected double balance;
  ...
}

class CheckingAccount extends BankAccount
{ private int transactions;
  ...
  public void deposit(double x)
  { transactions++; balance += x; }
}
```

Regole sulla visibilità

- In realtà all'atto pratico si verifica spesso che usare campi **`protected`** è poco efficace, e comporta problemi simili ad averli **`public`**.
- Tipicamente, il programmatore non ha il controllo su chi può scrivere le sottoclassi.
  - Parti di implementazione dichiarate **`protected`** risultano poi difficili da cambiare perché chi ha scritto una sottoclasse conta che esistano.
- Spesso la cosa migliore è specificare campi privati e gestirli con metodi set e get pubblici.

Modificatore `final`

- Il modificatore **`final`** si può usare per indicare che qualcosa non può essere più cambiato.
  - Si può applicare a **variabili**, **metodi**, **classi**.
- Un campo **`final`** è una costante, dato che non cambia più il proprio valore
  - Le costanti sono spesso poste anche **`public`**.
- Un metodo **`final`** non può essere ridefinito in alcuna sottoclasse.
- Una classe **`final`** non può essere estesa; implicitamente i suoi metodi sono tutti **`final`**.

Modificatore `final`

- Nel caso di variabile riferimento, quello che è immutabile è il riferimento all'oggetto.

```
final String str = "Hello";
str = str + "!";
```

```
final Punto a = new Punto();
a.setX(5.0);
```

- Ricordiamo che quando si usa il modificatore **`final`** a una variabile si può comunque anche attendere prima di dare il valore (*blank final*).
  - **`final`** dice che si può fare un solo assegnamento. Tuttavia, ne occorre uno prima dell'uso!

Modificatore `final`

- Perciò, se un campo statico (variabile di classe) viene denotato come **`final`** bisogna dirne subito il valore (*blank final* non ammesso).
  - Altrimenti l'assegnamento di default (**`0`** per gli **`int`**, **`0.0`** per i **`double`**, **`null`** per i tipi riferimento) conterebbe come primo e unico assegnamento
- Per un campo non statico (variabile d'istanza) invece è ammesso di porlo *blank final* ma prima di usare l'oggetto questo deve avere valore.
  - Quindi dev'essere esplicitamente inizializzato all'interno di ogni costruttore della classe.

Modificatore `final`

- Il campo **`RGL`** è posto esplicitamente a **`0`** per ogni oggetto della classe.
- Il campo **`BLK`** invece deve essere inizializzato da ogni costruttore, altrimenti il compilatore segnalerà un errore.

```
class Prova
{ final int RGL = 2; final int BLK;
  public Prova() { BLK = 1; }
  public Prova(int x) { BLK = x; }
  ...
  }
}
```

Modificatore `final`

- Il modificatore **`final`** si può anche usare in altri contesti (non identici, ma simili).
- Se abbinato a un parametro di un metodo, il valore di questo non può essere modificato.

```
public void unMetodo(final int X)
{ ...
  X = 0; //NON AMMESSO (non è un bello
         //          stile in ogni caso!)
}
```

Modificatore `final`

- Porre un metodo **`final`** significa che non se ne può fare overriding nelle sottoclassi.
- Questo serve a:
  - impedire alle sottoclassi di cambiarne il significato
  - ottimizzare la compilazione (il compilatore tratta questi metodi costanti in modo diverso, più efficiente)
- Supponiamo di avere una classe **`Test`** con un metodo **`verifica`** per valutare certe condizioni
- Una classe estesa **`Imbroglio`** vorrebbe sovrascrivere **`verifica`** per fargli ritornare sempre **`true`**.

Modificatore `final`

```
class Test
{ public final boolean verifica(parametri)
  { //valuta parametri : responso
    return responso;
  }
}
```

non ammesso! **`verifica()`** deve essere quello originale di **`Test`**

```
class Imbroglio extends Test
{ public boolean verifica(parametri)
  { return true; }
}
if (p.verifica(parametri)) {...}
...
```

per forza questo

Modificatore `final`

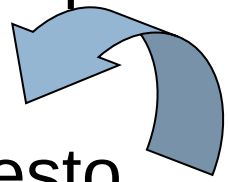
- Per le classi, il modificatore **`final`** specifica semplicemente che non possono essere estese.
- Il tentativo di estendere una classe indicata come **`final`** dà un errore in compilazione.
- Come per i metodi, ci sono aspetti di:
  - **efficienza**: il compilatore tratta diversamente una classe del genere
  - **correttezza**: si può impedire che qualcuno violi il contratto estendendo la classe in modo errato
- La classe **`String`** è **`final`**: non si può estendere

Modificatore `final`

- Usare il modificatore **`final`** su una classe, assomiglia ad usarlo per tutti i suoi metodi.
- Tuttavia, se usiamo **`final`** per la classe non possiamo più estenderla, mentre se usiamo **`final`** per tutti i metodi non ci pregiudichiamo la possibilità di estendere la classe in futuro.
- I campi su cui si basano dei metodi finali dovrebbero essere dichiarati privati. Questo perché porre il metodo finale significa che non si considera utente come particolarmente esperto.

Modificatore `final`

- Se invochiamo un metodo non-**`final`**, la JVM seguirà questa strada:
  - determina il tipo dell'oggetto che esegue il metodo
  - ricerca e associa al metodo l'implementazione corretta, verificando corrispondenze di tipi e firme
  - invoca l'implementazione del metodo



- Nel caso di un metodo **finale** basta questo, mentre i primi passaggi, che non sono necessari, possono essere saltati (sono già fatti dall'early binding).

Classi interne

- In Java una classe può essere dichiarata in (quasi) qualsiasi punto. Se compare dentro un'altra classe si chiama **classe interna**.
- Un oggetto istanza della classe interna si crea come di consueto dentro la classe che lo contiene. Al di fuori invece la sintassi è:

```
Esterna.Interna i = new Esterna().new Interna();
```

oppure

```
Esterna e = new Esterna();
Esterna.Interna i = new e.Interna();
```

Classi interne

- La classe interna accede i membri della classe esterna in modo naturale.

```
public class Esterna
{ private String messaggio = "nella classe ";
  private void stampa()
  { System.out.println(messaggio+"Esterna");
  }

  public class Interna
  public void metodo()
    {System.out.println(messaggio+"Interna");
    }
}
```

Classi annidate statiche

- Può risultare utile definire una classe interna per raggruppare certe funzionalità comuni.
- Si chiama classe annidata una classe interna che è un **membro statico** di quella esterna.
  - Meglio ancora “classe annidata” per il caso base, chiamando questa “classe annidata statica”, e “classe interna” l’altro caso
  - Una classe **`B`** annidata in una classe **`A`** è sempre riferita come **`A.B`** (se nel package **`P`**: **`P.A.B`**).
- Esempio di sintassi:

```
Esterna.Annidata i = new Esterna.Annidata();
```

Classi annidate statiche

- Una classe annidata statica raggruppa funzionalità usate solo localmente.

```
public class Lista
{ ...
  private static class Nodo
  { Object info;
    Nodo next;
    Nodo(Object info, Nodo next)
    { this.info = info; this.next = next; }
  }

  ...
}
```

Classi annidate: proprietà

- L'annidamento di classi semplifica l'organizzazione logica del codice.
  - Incapsulamento più forte
  - Migliore leggibilità
- La classe annidata ha accesso privilegiato alla classe ospite: non viene ereditato al di fuori.
- Dato che sono membri di altre classi, possono essere anche **`private`** (mentre una classe potrebbe essere solo public o -niente-).

Classi annidate: proprietà

- Le classi interne non statiche non possono avere membri statici (poiché dipendono da un'istanza della classe esterna).
- Le classi annidate e interne hanno una doppia versione di **`this`** e **`super`** (e anche di **`new`**).
  - normale: quello della classe stessa
  - dopo "**`Esterna.`**" : quello della classe esterna
- Ad esempio, **`Esterna.super.x`** accede al campo **`x`** della superclasse di **`Esterna`**

Classi annidate

```
public class Lista2
{ private int count = 0;
  private Nodo2 testa = null;
  private static class Nodo2
  { Object info;
    Nodo2 next;
    Nodo2(Object info, Nodo2 next)
    { this.info = info;
      this.next = next;
      count++;
    }
  ...
}
```

vale come
`Lista2.this.count++`

Classi interne locali

- Se una classe interna è definita nel corpo di un metodo, è detta **classe interna locale**.
- Analogamente alle variabili locali, questa classe viene usata solo internamente al blocco dove è definita.
- Se ne possono usare i metodi, quante volte si vuole.

Esempio classe locale

```
public class Esterna
{ private int[] vett = new int[20];
  public Esterna()
  { for (int i=0; i<20; i++) {vett[i] = i;} }
  public void indiciPari()
  { private class Pari
    { private int n = -2;
      public boolean hasNext() {return (n < 18);}
      public int getNext() { return vett[n=n+2]; }
    }
    Pari aaa = new Pari();
    while (aaa.hasNext())
    { System.out.print(aaa.getNext() + " ");  }
  }
  public static void main (String[] args)
  { Esterna z = new Esterna(); z.indiciPari(); }
}
```

classe locale

Classi anonime

- Una **classe anonima** è analoga a una classe interna locale, ma non ha nome.
- Quindi serve per essere usata una volta sola.

```
public class Gatto
{ private String color;
  public Gatto(String x) {this.color = x};
  public String getColor() {return color;}
  public void setColor(String x) {color = x;}
  public String toString()
  {return "Gatto " + color;}
}
```

Esempio classe anonima

```
public class ListaDeiGatti
{ public static void main(String[] args)
  { Gatto[] prova = new Gatto[3];
    prova[0] = new Gatto("Bianco");
    prova[1] = new Gatto("Nero");
    prova[2] = new Gatto("Tigrato") {
      public String toString() {
        return("gattino..." + getColor());
      } ;
    }
    for (Gatto g : prova)
    { System.out.println
      ("C'è un " + g.toString() );
    }
}
```

classe anonima