Programmazione orientata agli oggetti
Ereditarietà

Programmazione basata su oggetti

- Il modello visto finora costituisce un sottoinsieme del modello orientato agli oggetti
- Questo sottoinsieme viene chiamato normalmente programmazione basata su oggetti (object-based)
- La programmazione basata su oggetti poggia su due concetti fondamentali:
  - Astrazione: separazione fra interfaccia e implementazione
  - Incapsulamento: insieme di meccanismi che consentono di proteggere lo stato di un oggetto e in generale di nascondere gli aspetti che non si vogliono rendere pubblici
- E' un modello evolutivo: è nato dall'evoluzione delle pratiche di programmazione

Vantaggi della programmazione object-based

- E' comunque un passo avanti notevole rispetto alla programmazione procedurale
  - Consente di realizzare programmi con struttura modulare in cui ogni modulo è indipendente dai dettagli implementativi degli altri
  - Consente di anche realizzare componenti riusabili (le classi)
- Anche nella programmazione procedurale abbiamo forme di riuso: le librerie sono collezioni di elementi riusabili (funzioni)
- Le librerie però sono una forma di riuso di tipo "prendere o lasciare": una funzione ci va bene così com'è oppure dobbiamo scrivercene una da soli

Riuso fai da te: ahi, ahi, ahi!

- Per quello che abbiamo visto finora anche le classi si comportano nello stesso modo
- Nel definire la classe Orologio abbiamo riusato la classe Counter perché ci andava bene così com'era
- Se non ci fosse andata del tutto bene avremmo certo potuto modificarla ma questo approccio pone due problemi:
  - E' attuabile solo se riusiamo classi scritte da noi, di cui possediamo il sorgente (file .java)
  - Richiede di andare a rimettere mano a parti di codice già collaudate e potenzialmente utilizzate in molti punti.
- Questo secondo aspetto è molto rischioso perché è facile introdurre errori che si propagano in tutto il sistema!

Una forma migliore di riuso

- Per fare un vero salto in avanti in questo campo sarebbe utile poter fare una cosa di questo tipo:
  - La classe Counter ci va quasi bene ma non completamente
  - Non per questo dobbiamo rinunciare ad utilizzarla o a modificarla: ne creiamo una variante con le modifiche che ci servono
- In questo modo abbiamo una forma di riuso molto più flessibile:
  - Non siamo costretti a rifare da zero qualcosa che in gran parte è già pronto
  - Non corriamo rischio di introdurre errori in parti già stabili del sistema modificandole

Ereditarietà

- Il modello orientato agli oggetti (object-oriented e non object-based) ci mette a disposizione uno strumento per fare quallo che abbiamo appena descritto
- Questo strumento si chiama ereditarietà (inheritance)
- Grazie all'ereditarietà possiamo creare una nuova classe che estende un classe già esistente
- Su questa classe possiamo:
  - Introdurre nuovi comportamenti
  - Modificare i comportamenti esistenti

- Attenzione: la classe originale non viene assolutamente modificata. Le modifiche vengono fatte sulla classe derivata
- E' un concetto del tutto nuovo: non esiste nulla di simile nella programmazione procedurale

Esempio di ereditarietà

- **Partiamo dalla classe Counter:**

```
public class Counter
{
  private int val;
  public void reset()
  { val = 0; }
  public void inc()
  { val++; }
  public int getValue()
  { return val;}
}
```

- **Counter implementa un contatore monodirezionale: può andare solo avanti**

BiCounter

- **Immaginiamo di aver bisogno di un contatore bidirezionale, che può andare avanti e indietro**
- **Ci troviamo nella situazione appena descritta: abbiamo una classe che va quasi bene ma non del tutto**
- **Vorremmo poter aggiungere un metodo, dec(), che permetta di decrementare il valore del contatore**
- **Con la programmazione object-based abbiamo due sole alternative:**
  - **Scrivere una nuova classe, che potremmo chiamare BiCounter: fattibile ma è un peccato rifare tutto da zero**
  - **Modificare Counter, ma se facciamo un errore potremmo mettere in crisi tutti i programmi che usano già Counter**

BiCounter con l'ereditarietà

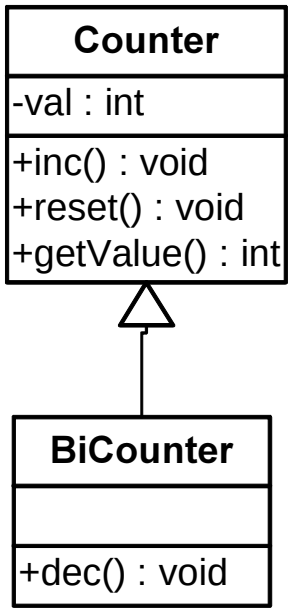
- **L'ereditarietà ci consente di trovare la situazione ottimale:**

```
public class BiCounter extends Counter
{
 public void dec()
 { val--; }
}
```

- **Questa nuova classe:**
  - **eredita da Counter il campo val (un int)**
  - **eredita da Counter tutti i metodi**
  - **aggiunge a Counter il metodo dec()**
- **Il segreto è nella parola chive extends che ci dice che BiCounter non è una classe creata da zero ma estende Counter riusandola in modo flessibile**

Rappresentazione dell'ereditarietà in UML

- **UML mette a disposizione una notazione grafica particolare per rappresentare l'ereditarietà**
- **Si usa una linea con un triangolo per collegare la classe che eredita da quella originale**
- **Il triangolo ha la parte larga (la base) rivolta verso la classe BiCounter per rappresentare l'idea di estensione**



Esempio di uso di BiCounter

- **Vediamo un esempio di utilizzo:**

```
public class EsempioBiCounter
{
   public static void main(String[] args)
   {
    int n;
    BiCounter b1;
    b1 = new BiCounter();
    b1.inc(); // metodo ereditato
    b1.dec(); // metodo nuovo
    n = b1.getValue();
    System.out.println(n);
   }
 }
```

- **Come si può notare possiamo invocare sull'istanza di BiCounter sia il metodo dec() definito in BiCounter che il metodo inc() che BiCounter eredita da Counter**

C'è qualcosa che non va…

- **Sembra tutto a posto, ma se si prova a compilare questo esempio si ottiene un errore nel metodo dec()**

```
public void dec()
{ val--; } // questa riga dà un errore di compilazione!
```

- **Il problema è che l'attributo val è stato definito in Counter come private e tutto quello che è private può essere visto e usato solo all'interno della classe che lo ha definito.**
- **Il metodo dec() appartiene a BiCounter, che è una classe diversa da Counter, e quindi gli attributi e gli eventuali i metodi privati di Counter non sono accessibili**

La visibilità protected

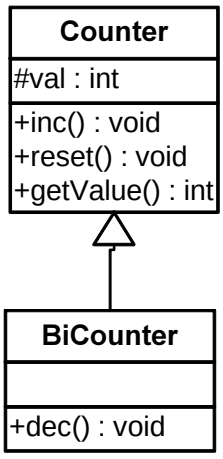
- Potremmo definire val come public, ma è una soluzione un po' eccessiva: val sarebbe visibile a tutti.
- E' evidente che serve un nuovo livello di visibilità.
- Java, come buona parte dei linguaggi ad oggetti, definisce a questo scopo il livello protected.
- Un attributo o metodo dichiarato protected è visibile nella classe che lo definisce e in tutte le classi che ereditano, direttamente o indirettamente, da essa.
- Attenzione: indirettamente vuol dire che se io definisco val come protected in Counter, val è accessibile in BiCounter, che eredita da Counter, ma anche in una eventuale classe SuperCounter che erediti da BiCounter

La soluzione corretta

- Vediamo la dichiarazione corretta di Counter e BiCounter:

```
public class Counter
{
  protected int val;
  public void reset()
  { val = 0; }
  public void inc()
  { val++; }
  public int getValue()
  { return val;}
}

public class BiCounter extends Counter
{
 public void dec()
 { val--; }
}
```



- In UML gli elementi protetti vengono indicati con il simbolo # (vedi a lato)

Un po' di terminologia

- La relazione espressa in Java dalla parola chiave extends prende il nome di ereditarietà (inheritance)
- In una relazione di ereditarietà:
  - La classe di partenza (nel nostro esempio Counter) prende il nome di classe base
  - La classe di arrivo (nel nostro caso BiCounter) prende il nome di classe derivata
- Si dice anche che:
  - BiCounter è una sottoclasse di Counter
  - Counter è una superclasse di BiCounter
- Il fatto che usando BiCounter possiamo utilizzare metodi definiti Counter prende il nome di riuso

Ricapitolando

- L'ereditarietà è uno strumento, tipico della programmazione orientata agli oggetti (OOP)
- Ci consente di creare una nuova classe che riusa metodi e attributi di una classe già esistente
- Nella classe derivata (sottoclasse) possiamo fare tre cose:
  - Aggiungere metodi
  - Aggiungere attributi
  - Ridefinire metodi

Attenzione: non è possibile togliere né metodi né attributi

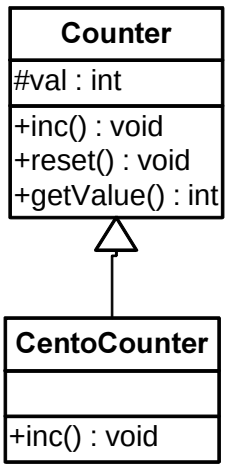
Ridefinizione di metodi

- Prendiamo in considerazione un altro esempio: ci serve un contatore monodirezionale che possa contare fino a 100 e non oltre
- Anche in questo caso Counter ci va quasi bene, ma non del tutto
- Però è un caso diverso un po' diverso dal precedente perché non dobbiamo aggiungere un comportamento (metodo) ma cambiare il funzionamento di un metodo esistente (inc())
- L'ereditarietà consente di fare anche questo: se in una classe derivata ridefiniamo un metodo già presente nella classe base questo sostituisce il metodo preesistente.
- Questo meccanismo prende il nome di overriding (sovrascrittura)

Esempio: la classe CentoCounter

- Definiamo quindi la classe CentoCounter:

```
public class CentoCounter extends Counter
{
  public void inc()
  {
    if (val<100)
      val++;
  }
}
```



- Come possiamo vedere anche in questo caso usiamo la parola chiave extends,
- Però non aggiungiamo un metodo ma ne ridefiniamo uno già esistente (overriding)
- A lato vediamo la rappresentazione UML di questa situazione
- Vedremo in seguito che la scelta che abbiamo fatto pone dei problemi

Overriding e overloading

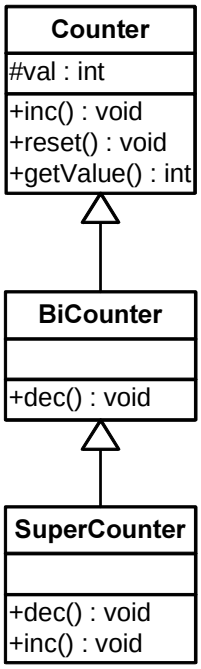
- Attenzione: non bisogna assolutamente confondere l'overloading con l'overriding!
- L'overloading ci permette di creare un nuovo metodo con lo stesso nome ma parametri diversi
- Il nuovo metodo si affianca a quello già esistente, che continua a poter essere utilizzato
- L'overriding ci permette di ridefinire un metodo esistente: il metodo ridefinito deve avere lo stesso nome e gli stessi parametri
- Il nuovo metodo sostituisce quello preesistente che non è più accessibile nella classe derivata
- Attenzione: se per caso ci sbagliamo e nel fare un overriding cambiamo il tipo di un parametro, Java lo interpreta coma un overloading!

Esempio su overriding e overloading

- Vogliamo derivare da BiCounter la classe SuperCounter che permette di fare incrementi e decrementi di valore specificato

```
public class SuperCounter
  extends BiCounter
{
  public void inc(int n)
  { val = val + n; }
  public void dec(int n)
  { val = val - n; }
}
```

- In questo caso abbiamo overloading e non overriding: i metodi inc() e dec() di BiCounter rimangono accessibili e vengono affiancati da inc(int n) e dec(int n)



Ereditarietà e costruttori - 1

- Abbiamo visto che quando usiamo l'ereditarietà la classe derivata (sottoclasse) eredita dalla classe base (superclasse):
  - Tutti gli attributi, anche quelli privati, a cui comunque la classe derivata non potrà accedere direttamente.
  - Tutti i metodi, anche quelli privati che la classe derivata non potrà usare direttamente
- Ma non eredita i costruttori: i costruttori sono specifici di una particolare classe: il costruttore di una classe non a caso ha il nome uguale alla classe
- Questo significa che quando creo un'istanza di BiCounter non viene invocato il costruttore di Counter ma il costruttore di default di BiCounter
- Dato che non ne abbiamo definito uno esplicitamente è quello creato automaticamente dal sistema

Ereditarietà e costruttori - 2

- Il fatto che vengano ereditati gli attributi implica che in ogni istanza della classe derivata abbiamo anche tutti gli attributi (lo stato) di un'istanza della classe base
- Alcuni di questi attributi non sono accessibili perché sono stati dichiarati come privati e quindi non abbiamo alcun modo per inizializzarli direttamente nel costruttore della classe derivata
- Ogni costruttore della classe derivata deve quindi invocare un costruttore della classe base affinché esso inizializzi gli attributi ereditati dalla classe base
- Ognuno deve costruire quello che gli compete

Ereditarietà e costruttori - 3

- Perché ogni costruttore della classe derivata deve invocare un costruttore della classe base?
- Per almeno tre motivi:
  - Solo il costruttore della classe base può sapere come inizializzare i dati ereditati in modo corretto
  - Solo il costruttore della classe base può garantire l'inizializzazione dei dati privati, a cui la classe derivata non potrebbe accedere direttamente
  - E' inutile duplicare nella sottoclasse tutto il codice necessario per inizializzare i dati ereditati, che è già stato scritto

Ereditarietà e costruttori - Super

- Ma come può un costruttore della classe derivata invocare un costruttore della classe base?
- Abbiamo visto che i costruttori non si possono mai chiamare direttamente!
- Occorre un modo consentire al costruttore della classe derivata di invocare un opportuno costruttore della classe base: la parola chiave super
- La definizione completa di BiCounter sarà quindi

```
public class BiCounter extends Counter
{
  public void BiCounter()
  { super() }
  public void dec()
  { val--; }
}
```

Ereditarietà e costruttori - automatismi

- E se non indichiamo alcuna chiamata a super(...)?
- Il sistema inserisce automaticamente una chiamata al costruttore di default della classe base aggiungendo la chiamata a super().
- In questo caso il costruttore dei default della classe base deve esistere, altrimenti si ha errore.
- Attenzione: il sistema genera automaticamente il costruttore di default solo se noi non definiamo alcun costruttore!
- Se c'è anche solo una definizione di costruttore data da noi, il sistema assume che noi sappiamo il fatto nostro, e non genera più il costruttore di default automatico!

Ancora su super

- La parola chiave super non è limitata solo ai costruttori.
- Nella forma super(...) invoca un costruttore della classe base ma può essere usata ovunque ci sia il bisogno di invocare un metodo della classe base
- Quando noi ridefiniamo un metodo (overriding) rendiamo invisibile il metodo della classe base
- Se all'interno del metodo ridefinito vogliamo invocare quello originale possiamo usare super
- Nella classe CentoCounter avremmo potuto fare così:

```
public class CentoCounter extends Counter
{
  public void inc()
  {
    if (val<100)
      super.inc();
  }
}
```

- E' una forma ancora più flessibile di riuso: in questo modo riusiamo il metodo originale aggiungendo solo quello che ci serve