Programmazione orientata agli oggetti
Subtyping e polimorfismo

Subclassing e subtyping

- Fino ad ora abbiamo trattato l'ereditarietà come strumento che consente il riuso flessibile di classi già esistenti mediante l'aggiunta o la ridefinizione di metodi
- In realtà l'ereditarietà ha una doppia natura, comprende cioè due diversi aspetti:
  - **Subclassing** o **ereditarietà di implementazione**: è un meccanismo per il riuso che ci consente di estendere classi esistenti riusando il codice già scritto
  - **Subtyping** o **ereditarietà di interfaccia**: è un meccanismo flessibile di compatibilità fra tipi

Variabili e tipi

- Nei linguaggi tradizionali, soprattutto in quelli legati al modello procedurale, esiste un sistema rigido di corrispondenza fra variabili e tipi
- Ogni variabile viene dichiarata come appartenente ad un tipo e, tranne poche eccezioni, non è possibile assegnargli valori di tipi diversi da quello di appartenenza
- Questo vincolo è molto importante perché consente al compilatore di effettuare tutta una serie di controlli che evitano i più comuni errori di programmazione
- Questi controlli vengono chiamati **statici** perché vengono effettuati una volta sola al momento della compilazione e non devono essere ripetuti continuamente durante l'esecuzione del programma

Tipizzazione in Java

- In linea di principio questo vale anche per Java
- Infatti se scriviamo istruzioni come queste:

```
int n;
String s = "18";
n = s;
```

- Otteniamo un errore di compilazione alla terza riga perché interi e stringhe sono cose completamente diverse
- Se vogliamo passare da un tipo all'altro dobbiamo farlo esplicitamente

```
n = Integer.intValue(s);
```

- Si dice quindi che Java è un **linguaggio tipizzato** perché il suo compilatore verifica staticamente che non ci siano violazioni al sistema dei tipi

Conversioni implicite

- In realtà anche in un linguaggio tipizzato vengono fatte conversioni implicite.
- Un esempio molto comune sono le conversioni che avvengono in un'espressione matematica:

```
int n = 5;
double d;
d = n * 2.5;
```

- Oppure nella concatenazione di stringhe:

```
int n = 5;
String s;
s = "Numero "+n;
```

- Si tratta però di eccezioni, in generale in un linguaggio tipizzato il cambio di tipo deve essere esplicitato

Typecast

- Le conversioni implicite vengono fatte solo quando si ha la certezza che non si introducono errori o perdite di informazioni.
- Su quest'ultimo aspetto in particolare Java è più restrittivo del C
- Per esempio se dichiariamo due variabili in questo modo

```
int n = 7;
long l = 14;
double d = 7.5
```

- Si può scrivere `l = n; d = n; d = l;` perché la conversione non comporta perdita di precisione
- Ma non è possibile scrivere: `n = l; n = d; l = d;` perché in tutti questi casi abbiamo potenzialmente perdita di informazione
- Dobbiamo esplicitare la conversione usando usando il **typecast** con la stessa sintassi del C

```
n = (int)l;
n = (int)d;
l = (long)d;
```

- In questo modo il compilatore è sicuro che non si tratta di un errore, ma di una cosa voluta

Sottoclassi come sottotipi

- Un sistema di tipi come quello appena descritto rappresenta una sicurezza ma può anche risultare eccessivamente rigido
- La programmazione orientata agli oggetti mette a disposizione un meccanismo più flessibile, ma altrettanto sicuro, basato sull'ereditarietà
- In una sottoclasse noi possiamo solo **aggiungere** o **ridefinire** metodi, ma **non eliminarne**!
- Quindi un'istanza di una sottoclasse è capace di fare tutto quello che sa fare la sua superclasse
- **Ne consegue che possiamo utilizzare un'istanza di una sottoclasse al posto di un istanza di una superclasse**
- Si dice quindi che una **sottoclasse è un sottotipo (subtyping)**

Subtyping - 1

- In pratica nei linguaggi orientati agli oggetti **possiamo assegnare ad una variabile che ha come tipo una superclasse un'istanza di una qualsiasi delle sue sottoclassi**
- Per esempio possiamo scrivere:

```
Counter c;
c = new BiCounter();
```

- In queste due istruzioni è racchiuso il concetto di **subtyping**
- E' una forma estesa di conversione implicita:
  - L'insieme di metodi di BiCounter è un **sovrainsieme** di quello di Counter: BiCounter sa fare tutto quello che fa Counter
  - Il compilatore ha quindi la certezza che non possiamo chiedere all'istanza di BiCounter di fare qualcosa che non è in grado di fare

Ereditarietà di interfaccia e di implementazione

- L'insieme dei metodi di una classe viene anche chiamato **interfaccia della classe**
- Possiamo quindi dire che **l'interfaccia di una sottoclasse comprende l'interfaccia della sua superclasse (la eredita)**
- E' questo il senso del termine **ereditarietà di interfaccia** con cui spesso il **subtyping** viene designato
- In modo simile si parla di **ereditarietà di implementazione** per indicare il **subclassing**
- Infatti una classe derivata comprende l'implementazione della classe base

- Proviamo alcuni esempi ….

Sostituibilità: Counter e BiCounter - 1

- Riprendiamo le due classi Counter e BiCounter (senza i costruttori …)

```
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--; }
}
```

Sostituibilità: Counter e BiCounter - 2

- Proviamo a scrivere un'applicazione di esempio che usa Counter

```
public class Esempio
{
   public static void main(String[] args)
   {
    int n;
    Counter c1;
    c1 = new Counter(0);
    for (i=0; i<150; i++)
      c1.inc();
    n = c1.getValue();
    System.out.println("Valore: "+n);
   }
 }
```

- L'applicazione scriverà a video: **Valore: 150**

Sostituibilità: Counter e BiCounter - 3

- Modifichiamo l'esempio usando un'istanza di BiCounter anziché una di Counter

```
public class Esempio
{
   public static void main(String[] args)
   {
    int n;
    Counter c1;
    c1 = new BiCounter(); // Era c1 = new Counter()
    for (i=0;i<150;i++)
      c1.inc();
    n = c1.getValue();
    System.out.println(n);
   }
 }
```

- Il programma funziona esattamente come prima scrivendo a video: **Valore: 150**

Ancora sul subtyping

- Riprendiamo l'esempio di subtyping fra Counter e BiCounter

```
Counter c;
c = new BiCounter();
```

- **Attenzione:** anche se la variabile c fa riferimento ad un'istanza di BiCounter è di tipo Counter e quindi possiamo fare con c solo quello che sa fare Counter
- Possiamo scrivere: `c.inc();`
- Ma **non**: `c.dec();`
- **E' il tipo della variabile, e non il tipo dell'istanza, a determinare quello che possiamo fare!**

Overriding

- Se l'ereditarietà consentisse solo l'aggiunta di metodi non ci sarebbe molto altro da dire
- Sappiamo invece che è anche possibile la ridefinizione di un metodo esistente (**overriding**)
- Abbiamo visto che questo meccanismo aggiunge una notevole flessibilità ai meccanismi di riuso
- Ma oltre a ciò la combinazione tra subtyping e overriding apre nuove promettenti prospettive…
- … ma anche qualche possibile fonte di guai!

Sostituibilità: Counter e CentoCounter - 1

- Riprendiamo ora in esame la classe CentoCounter che, come BiCounter è una sottoclasse di Counter
- In questo caso però in CentoCounter anziché aggiungere un metodo ne avevamo ridefinito uno:

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

Sostituibilità: Counter e CentoCounter - 2

- Proviamo ora a sostituire un'istanza di Counter con una di CentoCounter nel nostro esempio:

```
public class Esempio
{
   public static void main(String[] args)
   {
    int n;
    Counter c1;
    c1 = new CentoCounter();// Era c1=new Counter()
    for (i=0;i<150;i++)
      c1.inc();
    n = c1.getValue();
    System.out.println(n);
   }
 }
```

- Il programma scrive a video: **Valore: 100**
- **CentoCounter non è sostituibile con Counter!**

Riassumendo

- **L'ereditarietà va sempre usata per estendere**
- Se si usa l'ereditarietà per restringere si viola la sostituibilità tra superclasse e sottoclasse
- Questo è il motivo per cui quando si eredità non è consentito eliminare metodi
- Aggiungendo metodi non ci corrono rischi

💣 **Attenzione:** quando si ridefinisce un metodo c'è un potenziale rischio: bisogna sempre operare in modo da non restringere il comportamento del metodo originale

Nota: Overriding e Overloading

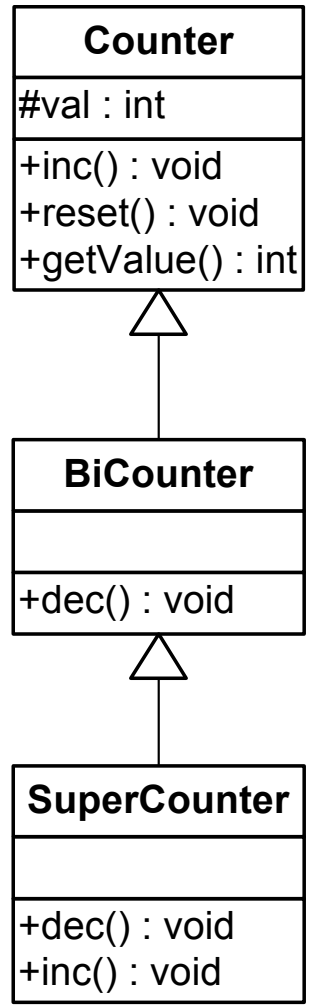
Sono due cose diverse!!!!

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)



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 scrivere 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

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 scrivere anche così:

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

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

Polimorfismo

Polimorfismo - Esempio

- Definiamo la classe Persona che gestisce i dati anagrafici di una persona (nome ed età per semplicità)
- La classe definisce un costruttore e il metodo print che stampa a video nome ed età:

```
public class Persona
{
  protected String nome;
  protected int anni;
  public Persona(String n, int a)
  { nome=n;
    anni=a;
  }
  public void print()
  {
    System.out.print(”Mi chiamo " + nome);
    System.out.println(" e ho ” + anni + "anni");
  }
}
```

Esempio: … e studenti

- Definiamo la classe Studente, sottoclasse di Persona che ridefinisce il metodo print():

```
public class Studente extends Persona
{
  protected int matr;
  public Studente(String n, int a, int m)
  {
    super(n,a);
    matr=m;
   }
  public void print()
  {
    super.print(); // stampa nome ed età
    System.out.println("Matr = " + matr);
  }
}
```

- In questo modo print() stampa nome, età e matricola

EsempioDiCitta

- Definiamo infine la classe EsempioDiCitta che implementa il metodo statico main ed è quindi la classe principale della nostra applicazione:

```
public class EsempioDiCitta
{
  public static void main(String args[])
  {
    Persona  p = new Persona("John",45);
    Studente s = new Studente("Tom",20,156453);
    p.print();  // stampa nome ed età
    s.print();  // stampa nome, età, matricola
    p=s;        // Ok, per il subtyping
    p.print();  // COSA STAMPA ???
  }
}
```

- Cosa stampa l'ultima istruzione?

Polimorfismo

- La risposta è: stampa **nome, età e matricola!**
- p è un riferimento di tipo Persona,
- Però p punta ad un'istanza di classe Studente
- Se scriviamo p.print() viene eseguito il metodo print() ridefinito dalla classe Studente e non quello originale definito nella classe Persona
- Quindi: anche se usiamo un riferimento che ha per tipo una superclasse il fatto che l'istanza a cui il riferimento punta appartenga alla sottoclasse fa sì che il metodo invocato sia quello della sottoclasse
- Questa proprietà prende il nome di **polimorfismo (verticale)**
- Ereditarietà e polimorfismo sono i due principi che differenziano la programmazione **object-oriented** dalla programmazione **object-based**

Subtyping e polimorfismo - 1

- Subtyping e polimorfismo sono strettamente correlati
- Grazie al subtyping possiamo scrivere:
- ```
  Persona p;
  p = new Studente("Pietro",22,456327);
  ```
- Abbiamo cioè assegnato un'istanza di tipo Studente a un riferimento di tipo Persona
- Di conseguenza abbiamo potuto scrivere:

  ```
  p.print();
  ```
- In virtù del **subtyping** questa espressione è valida
- In virtù del **polimorfismo** il metodo print() che viene eseguito è quello di Studente
  ```

Subtyping e polimorfismo - 2

- Riassumendo:
  - **Il tipo del riferimento determina quello che si può fare:** possiamo invocare solo i metodi definiti nella classe a cui il riferimento appartiene (**subtyping**)
  - **Il tipo dell'istanza determina cosa viene effettivamente fatto:** viene invocato il metodo definito nella classe a cui l'istanza appartiene (**polimorfismo**)

- Per risolvere le chiamate ai metodi Java utilizza **late binding**
- **NOTA BENE: il codice delle chiamate è individuato dinamicamente sulla base della natura dell'istanza**

Downcasting

- Abbiamo visto che se scriviamo:
- ```
  Counter c;
  c = new BiCounter();
  ```
- Possiamo utilizzare i metodi definiti in Counter ma non quelli definiti in BiCounter
- Quindi non è ammesso un'istruzione come: `c.dec();`
- Se vogliamo chiamare `dec()` come possiamo fare?
- Dobbiamo ricorrere ad una conversione esplicita (typecasting)
- ```
  BiCounter b = (BiCounter)c;
  b.dec();
  ```
- Questa conversione viene chiamata **downcasting**

Sostituibilità

- Il **subtyping** in qualche modo implica che le classi derivate possono essere sostituite alle classi base in modo “sicuro”
- Da un punto di vista sintattico questo è senz’altro vero: il tipo del riferimento decide cosa possiamo fare e quindi **possiamo invocare sulla classe derivata solo i metodi che erano presenti anche nella classe base**
- Non corriamo il rischio di chiamare metodi inesistenti
- Se nella classe derivata abbiamo soltanto aggiunto metodi questo è vero in assoluto: i metodi della classe base sono quelli originali e si comportano come nella classe base
- Se invece abbiamo **ridefinito qualche metodo**, in virtù del **polimorfismo** il metodo eseguito dipende dall’istanza e non dal riferimento
- **Potremmo trovarci di fronte a qualche sorpresa!**

Violazioni di sostituibilità

- **Cosa è successo?**
- Ridefinendo il metodo inc() in CentoCounter abbiamo fatto saltare la sostituibilità fra Counter e CentoCounter
- **La classe derivata non è più sostituibile con la classe base**
- Il pasticcio è nato dalla combinazione di due fattori:
  - Abbiamo ridefinito un metodo (overriding)
  - Nel ridefinirlo abbiamo **ristretto** il comportamento della classe derivata
- **L'errore è stato quello di usare l'ereditarietà per restringere e non per estendere**
- Non a caso la parola chiave che Java usa per indicare i legami di ereditarietà è **extends**