

---

# Programmazione orientata agli oggetti

## Subtyping

## 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.parseInt(s);**
- Si dice quindi che Java è un **linguaggio tipizzato (*typed*)** 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 (**a meno che non ridefinisca un metodo**)
- Proviamo alcuni esempi ....

- 
- Esempi al calcolatore su:
- 1) Costruttori ed ereditarietà
  - 2) Subtyping e Polimorfismo
  - 3) Overriding
  - 4) Object

## Subtyping (*recap*)

---

- La gerarchia delle classi "induce" una gerarchia di tipi e sottotipi
- 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)**

## Sostituibilità: Counter e BiCounter - 1

---

- Riprendiamo le due classi Counter e BiCounter (con o senza 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
- Definiamo nel main un oggetto Counter , creiamolo (new) e incrementiamolo 150 volte ...
  - Ciclo for per i=1..150 usando il metodo inc()
- Poi stampiamone il valore (acquisendolo con il metodo getValue())

## Sostituibilità: Counter e BiCounter - 2

---

```
public class Esempio
{
    public static void main(String[] args)
    { int n;
      Counter c1;
      c1 = new Counter();
      c1.reset();
      for (int 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
- Definiamo il reference come Counter, ma creiamo l'istanza come BiCounter nel main
- Poi stampiamone il valore
- Cosa stampa?

## 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()
        c1.reset();
        for (i=0;i<150;i++)
        {
            c1.inc();
            n = c1.getValue();
            System.out.println(n);
        }
    }
}
```

- Cosa stampa?

## Sostituibilità: Counter e BiCounter - 3

---

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

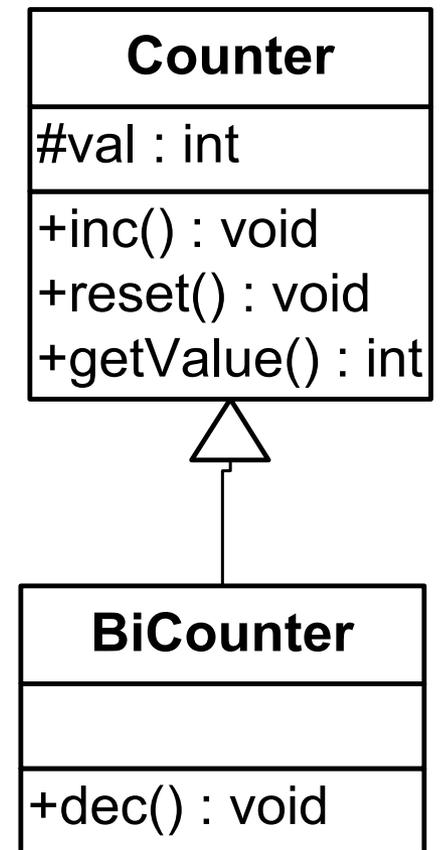
```
public class Esempio
{
    public static void main(String[] args)
    {
        int n;
        Counter c1;
        c1 = new BiCounter(); // Era c1 = new Counter()
        c1.reset();
        for (i=0;i<150;i++)
        {
            c1.inc();
            n = c1.getValue();
            System.out.println(n);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

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

## Subtyping (ereditarietà di interfaccia)

---

- Ogni oggetto istanza di BiCounter è un Counter
- Risponde agli stessi metodi di Counter (**ereditarietà di interfaccia**)
- Link **is-a**, **BiCounter is-a Counter**



## 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 dichiarata 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!**

## 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 ();
        c1.reset ();
        for (i=0;i<150;i++)
        {
            c1.inc ();
            n = c1.getValue ();
            // c1.dec ();
            System.out.println(n);
        }
    }
}
```

## Counter e BiCounter - 4

---

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

- Casting esplicito, e assegnamento a un reference di tipo `BiCounter` per poter invocare anche i metodi di `BiCounter`

## 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