Esempi al calcolatore su:

1)Costruttori ed ereditarietà

2)Subtyping e polimorfismo

Esempio 1:
Costruttori ed ereditarietà

Introduzione

- Java prevede due automatismi legati ai costruttori:
  - Se una classe non ha costruttori viene creato automaticamente il **costruttore di default** (quello senza parametri)
  - Se in un costruttore di una classe derivata non viene invocato il costruttore delle classe base, viene automaticamente inserita la **chiamata a super()**
- Questi due meccanismi sono molto comodi ma bisogna fare attenzione perché si possono creare situazioni di errore non facili da comprendere
- Sviluppiamo un esempio che ha lo scopo di chiarire questi aspetti

Esempio 1 – prima versione

- Counter
  - con attributo val intero protected e metodi pubblici reset(), inc(), getValue()
  - con costruttore di default definito che inizializza a 1 l'attribito val e costruttore con un parametro intero n che inizializza val a n (anche un po' di stampe …)
- BiCounter estende Counter
  - con un metodo pubblico dec()
  - senza alcun costruttore
- Main
  - con un metodo pubblico statico main()
  - Definisce Counter c e lo crea
  - Definisce BiCounter c1 e lo crea

Prima versione

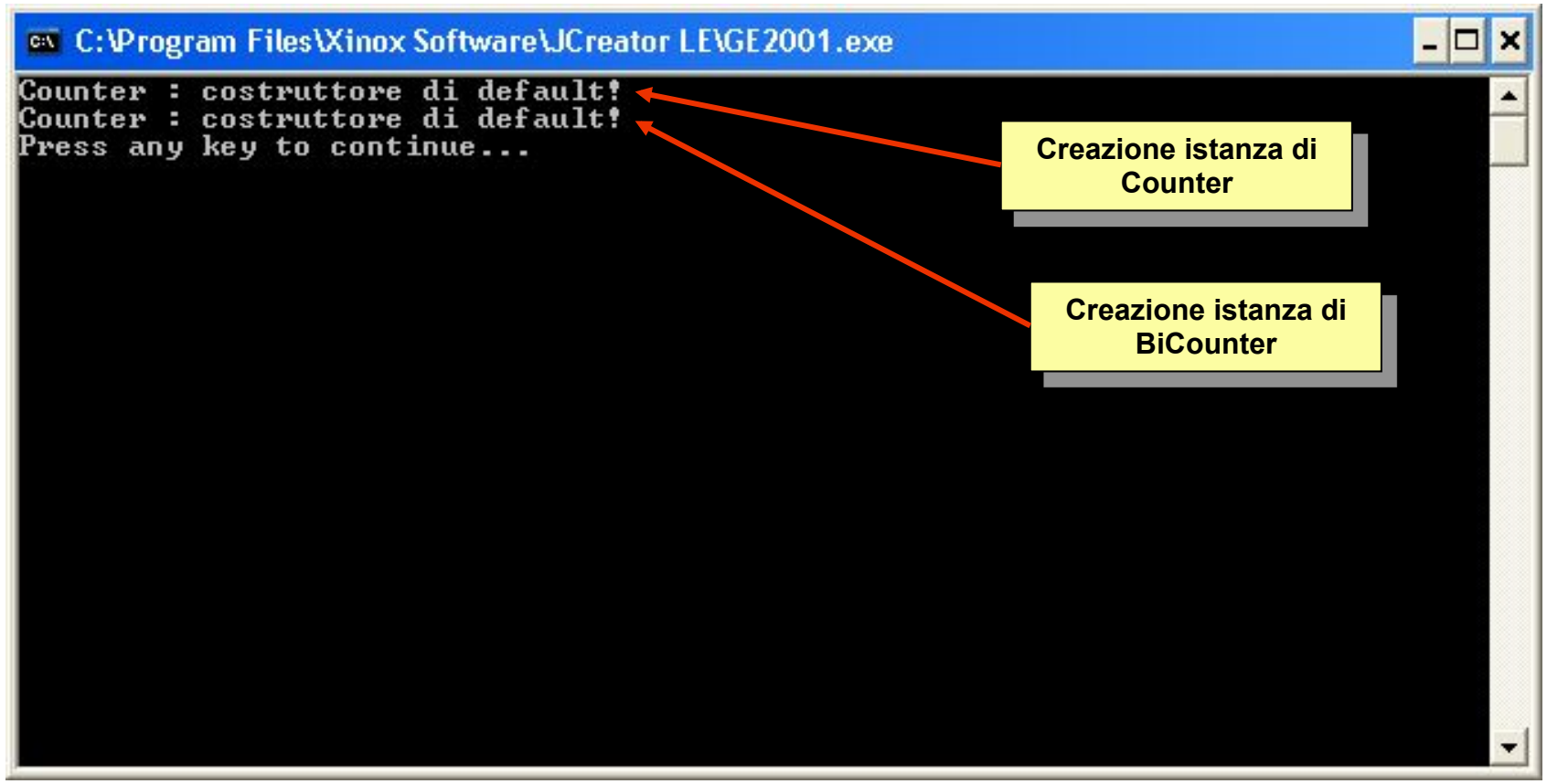
```
public class Counter
{
  protected int val;
  public Counter()
  {
    System.out.println("Counter:
     costruttore default!");
    val = 1;
  }
  public Counter(int v)
  {
    System.out.println(
     "Counter:costruttore");
    val = v;
  }
  public void reset()
  { val = 0; }
  public void inc()
  { val++; }
  public int getValue()
  { return val;}
}
```

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

Non definiamo un costruttore per BiCounter

```
public class Main
{
 public static void
   main(String Args[])
 {
   Counter c =
     new Counter();
   BiCounter c1 =
     new BiCounter();
  }
}
```

Risultato



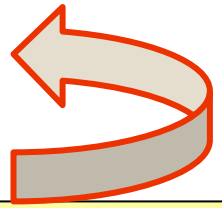
- Il compilatore Java ha aggiunto un costruttore di defaul in BiCounter e ha inserito in esso una chiamata a super()

Prima versione – come se fosse:

```
public class Counter
{
  protected int val;
  public Counter()
  {
    System.out.println("Counter:
     costruttore default!");
    val = 1;
  }
  public Counter(int v)
  {
    System.out.println(
     "Counter:costruttore");
    val = v;
  }
  public void reset()
  { val = 0; }
  public void inc()
  { val++; }
  public int getValue()
  { return val;}
}
```

```
public class BiCounter
  extends Counter
{
   public BiCounter()
  { super();  }

  public void dec()
  {
    val--;
  }
}
```



Non definiamo un costruttore per BiCounter

```
public class Main
{
 public static void
   main(String Args[])
 {
   Counter c =
     new Counter();
   BiCounter c1 =
     new BiCounter();
  }
}
```

Esempio 1 – seconda versione

- Counter
  - *invariato*
- BiCounter estende Counter
  - con un metodo pubblico dec() *(come prima)*
  - definisce un costruttore che inizializza val a 1 (e fa un po' di stampe …)
- Main
  - *invariato*

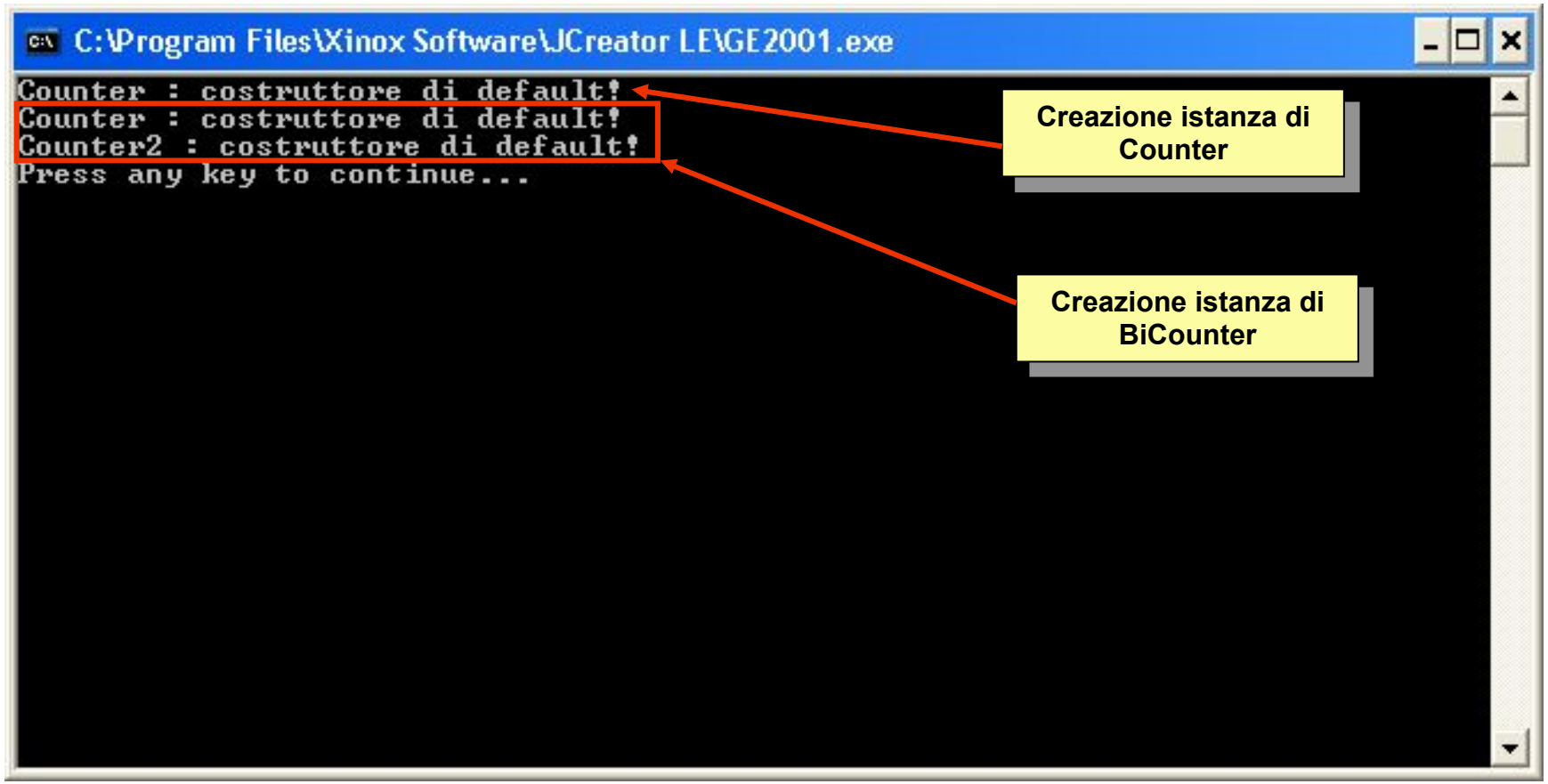
Seconda versione

```
public class Counter
{
  protected int val;
  public Counter()
  {
    System.out.println("Counter:
     costruttore default!");
    val = 1;
  }
  public Counter(int v)
  {
    System.out.println(
     "Counter:costruttore");
    val = v;
  }
  public void reset()
  { val = 0; }
  public void inc()
  { val++; }
  public int getValue()
  { return val;}
}
```

```
public class BiCounter
  extends Counter
{
  public BiCounter()
  {  System.out.println("Counter2:
       costruttore di default!");
     val = 1;
  }
  public void dec()
  { val--; }
}
```

```
public class Main
{
 public static void
   main(String Args[])
 {
   Counter c =
     new Counter();
   BiCounter c1 =
     new BiCounter();
  }
}
```

Risultato



- Il compilatore ha aggiunto una chiamata a super() nel costruttore di BiCounter

Seconda versione – come se fosse:

```
public class Counter
{
  protected int val;
  public Counter()
  {
    System.out.println("Counter:
     costruttore default!");
    val = 1;
  }
  public Counter(int v)
  {
    System.out.println(
     "Counter:costruttore");
    val = v;
  }
  public void reset()
  { val = 0; }
  public void inc()
  { val++; }
  public int getValue()
  { return val;}
}
```

```
public class BiCounter
  extends Counter
{
  public BiCounter()
  {  super();
     System.out.println("Counter2:
       costruttore di default!");
     val = 1;
  }
  public void dec()
  { val--; }
}
```

```
public class Main
{
 public static void
   main(String Args[])
 {
   Counter c =
     new Counter();
   BiCounter c1 =
     new BiCounter();
  }
}
```

Esempio 1 – terza versione

- Counter
  - con attributo val intero protected e metodi pubblici reset(), inc(), getValue()
  - **senza costruttore di default definito**, ma solo costruttore con un parametro intero n che inizializza val a n (anche un po' di stampe …)
- BiCounter estende Counter
  - con un metodo pubblico dec()
  - costruttore di default definito che inizializza val a 1
- Main
  - *invariato*

Terza versione

```
public class Counter
{
  protected int val;

  public Counter(int v)
  {
    System.out.println(
     "Counter:costruttore");
    val = v;
  }
  public void reset()
  { val = 0; }
  public void inc()
  { val++; }
  public int getValue()
  { return val;}
}
```

Eliminiamo il costruttore di default

```
public class BiCounter
  extends Counter
{
  public BiCounter()
  {  System.out.println("Counter2:
       costruttore di default!");
     val = 1;
  }
  public void dec()
  { val--; }
}
```

```
public class Main
{
 public static void
   main(String Args[])
 {
   Counter c =
     new Counter();
   BiCounter c1 =
     new BiCounter();
  }
}
```

Risultato

```
C:\work\costruttori\Main.java:6: cannot resolve symbol
symbol  : constructor Counter  ()
location: class Counter
    Counter c = new Counter();
                ^
C:\work\costruttori\Counter.java:1: cannot resolve symbol
symbol  : constructor Counter  ()
location: class Counter
public class BiCounter extends Counter {
       ^
2 errors
Process completed.
```

- Nella classe Counter, il costruttore di default non è stato inserito automaticamente, perché Counter ha almeno un costruttore (quello con parametro)
- Abbiamo quindi **2 errori di compilazione**

Una prima correzione

- Il primo errore è dovuto al fatto che in main() creiamo un'istanza di Counter invocando il costruttore di default che non esiste più
- Modifichiamo quindi main() in modo che l'istanza venga creata invocando il costruttore non di default

```
public class Main
{
 public static void main(String Args[])
 {
    Counter c = new Counter(1);
    Counter c1 = new BiCounter();
   }
}
```

Risultato

```
C:\work\costruttori\Counter.java:1: cannot resolve
symbol
symbol  : constructor Counter  ()
location: class Counter
public class BiCounter extends Counter {
       ^
1 error
```

- Ci resta ancora un errore dovuto al fatto che nel costruttore di BiCounter **il compilatore ha inserito una chiamata a super()** (costruttore di default della classe base) e questo non esiste in Counter

Seconda correzione

- Inseriamo quindi nel costruttore di BiCounter una chiamata esplicita al costruttore effettivamente esistente di Counter (quello con il parametro intero)

```
public class BiCounter
  extends Counter
{
  public BiCounter()
  {  System.out.println("Counter2:
       costruttore di default!");
     super(0);
  }
  public void dec()
  { val--; }
}
```

Risultato

```
C:\work\costruttori\Counter2.java:3: cannot resolve symbol
symbol  : constructor Counter  ()
location: class Counter
 {
 ^
C:\work\costruttori\Counter2.java:5:
call to super must be first statement in constructor
  super(0);
        ^
2 errors
```

- Otteniamo ancora degli errori di compilazione
- Infatti **la chiamata a super(0) deve essere la prima istruzione del costruttore!**

Terza correzione

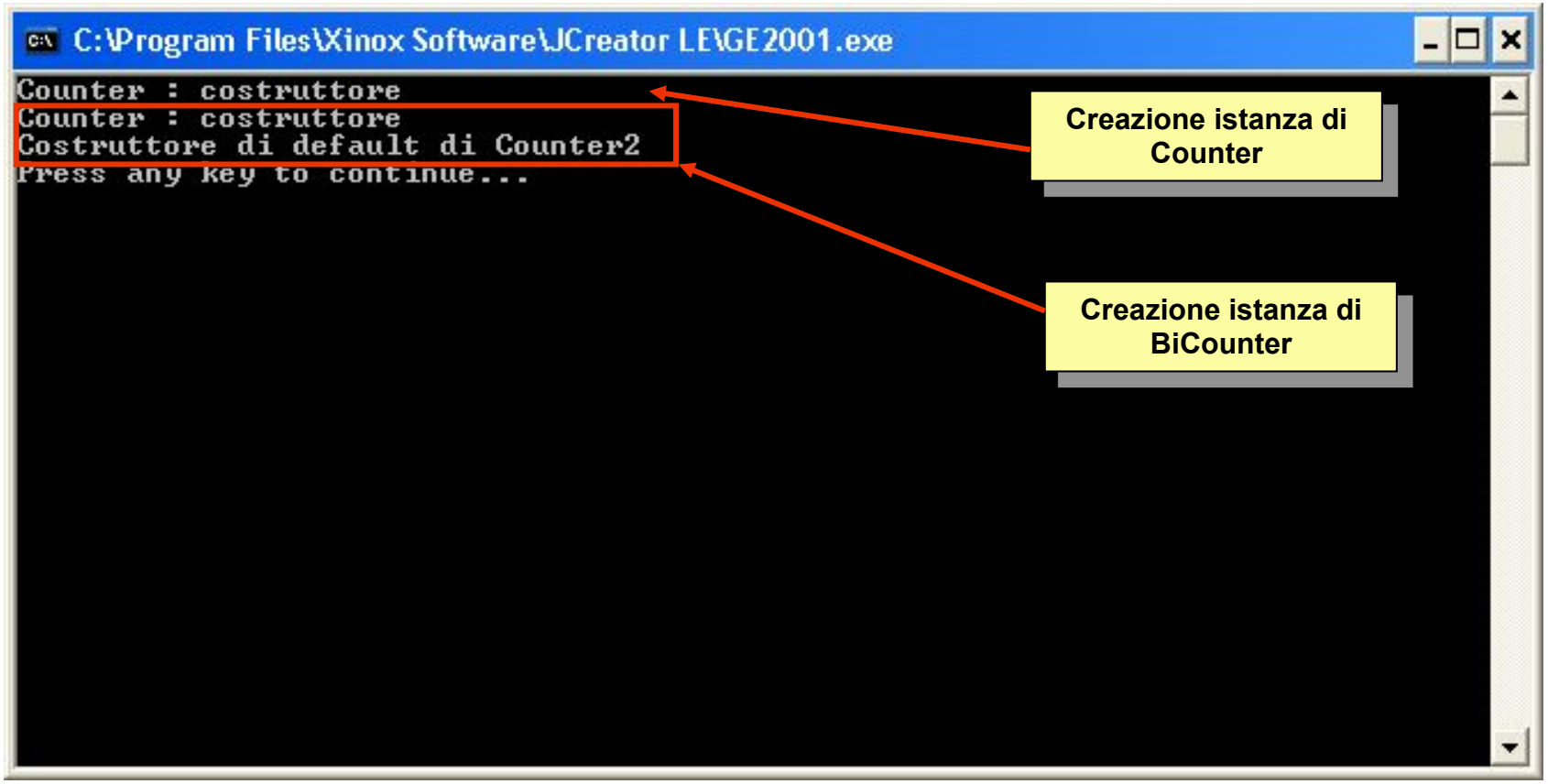
- Mettiamo la chiamata a super in prima posizione:

```
public class BiCounter
  extends Counter
{
  public BiCounter()
  {
    super(0);
    System.out.println(
      "Costruttore di default
       di Counter2");
  }
  public void dec()
  {
    val--;
  }
}
```

```
public class Main
{
 public static void
  main(String Args[])
 {
  Counter c = new Counter(1);
  BiCounter c1 =  new BiCounter();
   }
}
```

Risultato finale

- Adesso funziona tutto correttamente



Ricapitolando…

- La chiamata al costruttore del parent (super()) deve essere la prima istruzione dei costruttori delle classi derivate
- Se non è specificato super() nei costruttori delle classi derivate, è invocato automaticamente il costruttore di default della classe base
- Il costruttore di default può essere generato automaticamente solo se nella classe base non è definito alcun costruttore
- Se non è presente un costruttore di default (definito dal programmatore o generato in automatico) di una classe, le classi derivate da questa devono esplicitamente usare la super(…) ed invocare il corretto costruttore (non di default)

Esempio 2:
Subtyping e polimorfismo

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

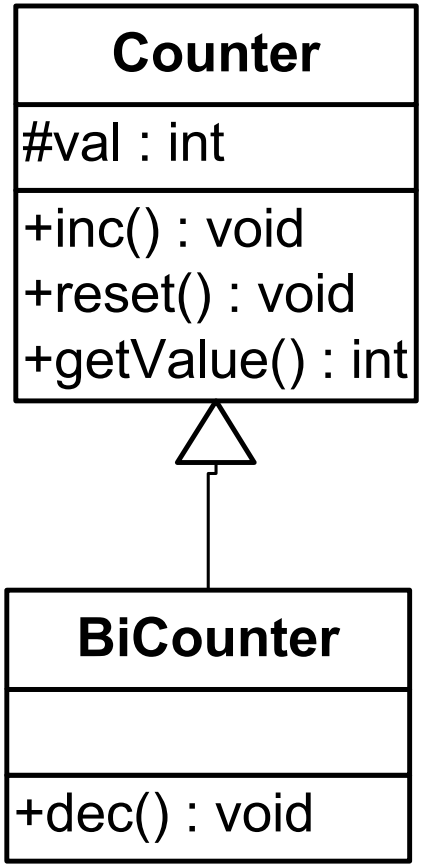
- Modifichiamo l'esempio usando un'istanza di BiCounter anziché una di Counter

```
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

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



Sostituibilità: Counter e CentoCounter - 1

- Definiamo ora la classe CentoCounter che, come BiCounter , è una sottoclasse di Counter
- In questo caso però in CentoCounter, anziché aggiungere un metodo, ne **ridefiniamo** uno:

```
public class CentoCounter extends Counter
   {
    public CentoCounter()    {
       super();
       System.out.println("CentoCounter: costruttore");}
     public void inc()
     {
       if (val<100)
        val++;
     }
   }
```

Sostituibilità: Counter e BiCounter - 3

- Modifichiamo l'esempio usando un'istanza di CentoCounter anziché una di Counter
- Definiamo il reference come Counter, ma creiamo l'istanza come CentoCounter nel main
- Poi stampiamone il valore
- Cosa stampa?

Sostituibilità: Counter e CentoCounter - 2

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

- Cosa stampa?

Sostituibilità: Counter e CentoCounter - 2

- Sostituendo 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()
    c1.reset();
    for (int 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!**

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**

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 …

- … tuttavia:

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