Esempi al calcolatore su:

1) Costruttori ed ereditarietà

2) Subtyping e Polimorfismo

3) Overriding

4) Object

Esempio 1:
Costruttori ed ereditarietà

Esempio 1 – versione zero

- Counter
  - con attributo val intero protected e metodi pubblici reset(), inc(), getValue()
  - senza alcun costruttore
- 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

versione zero

```
public class Counter
{
  protected int val;

  public void reset()
  { val = 0; }

  public void inc()
  { val++; }

  public int getValue()
  { return val;}
}
```

Non definiamo un costruttore per Counter

```
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();
  }
}
```

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;}
}
```

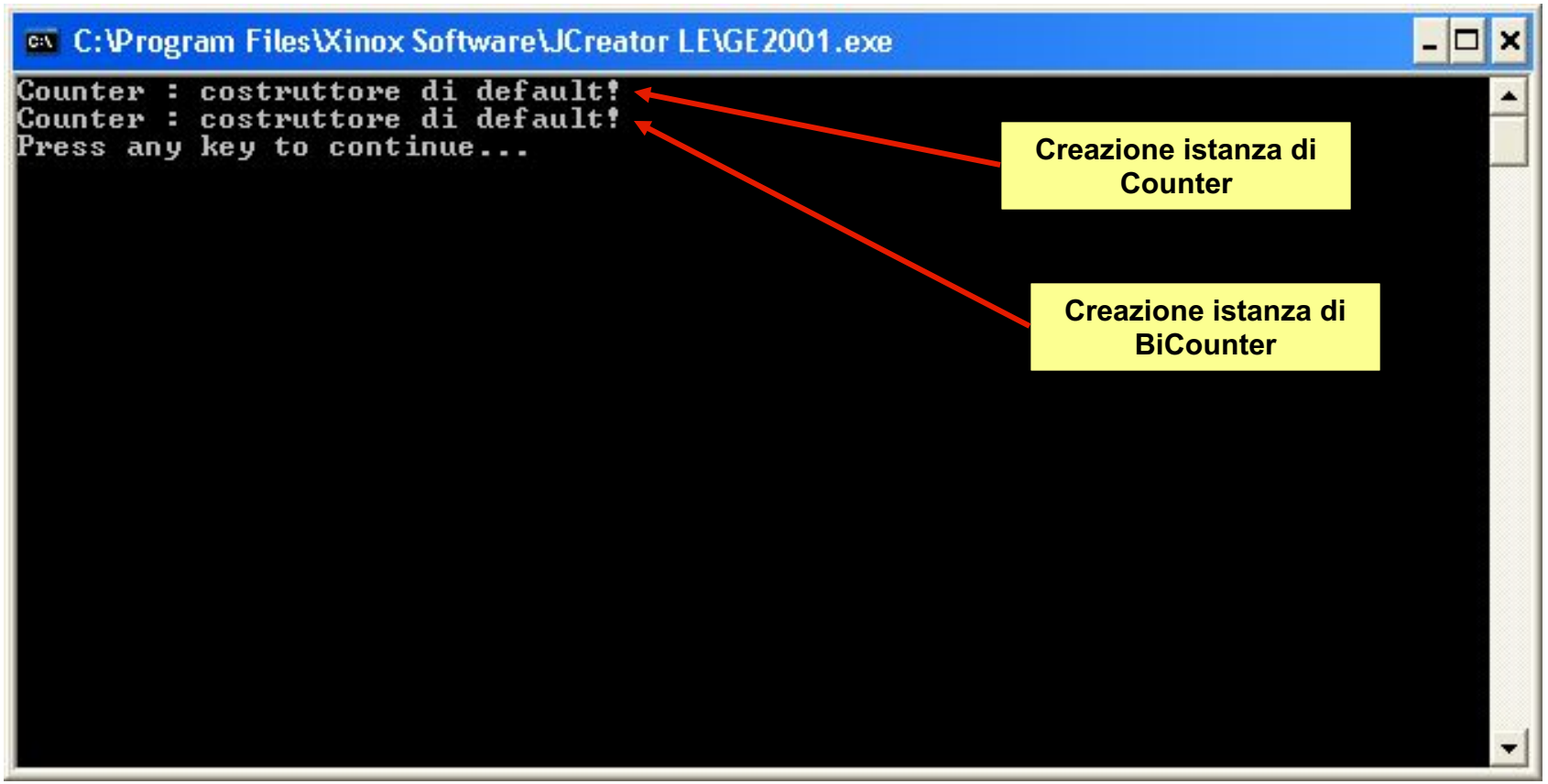
```
public class BiCounter extends Counter

{
  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()

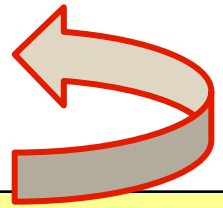
Prima versione

```
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*

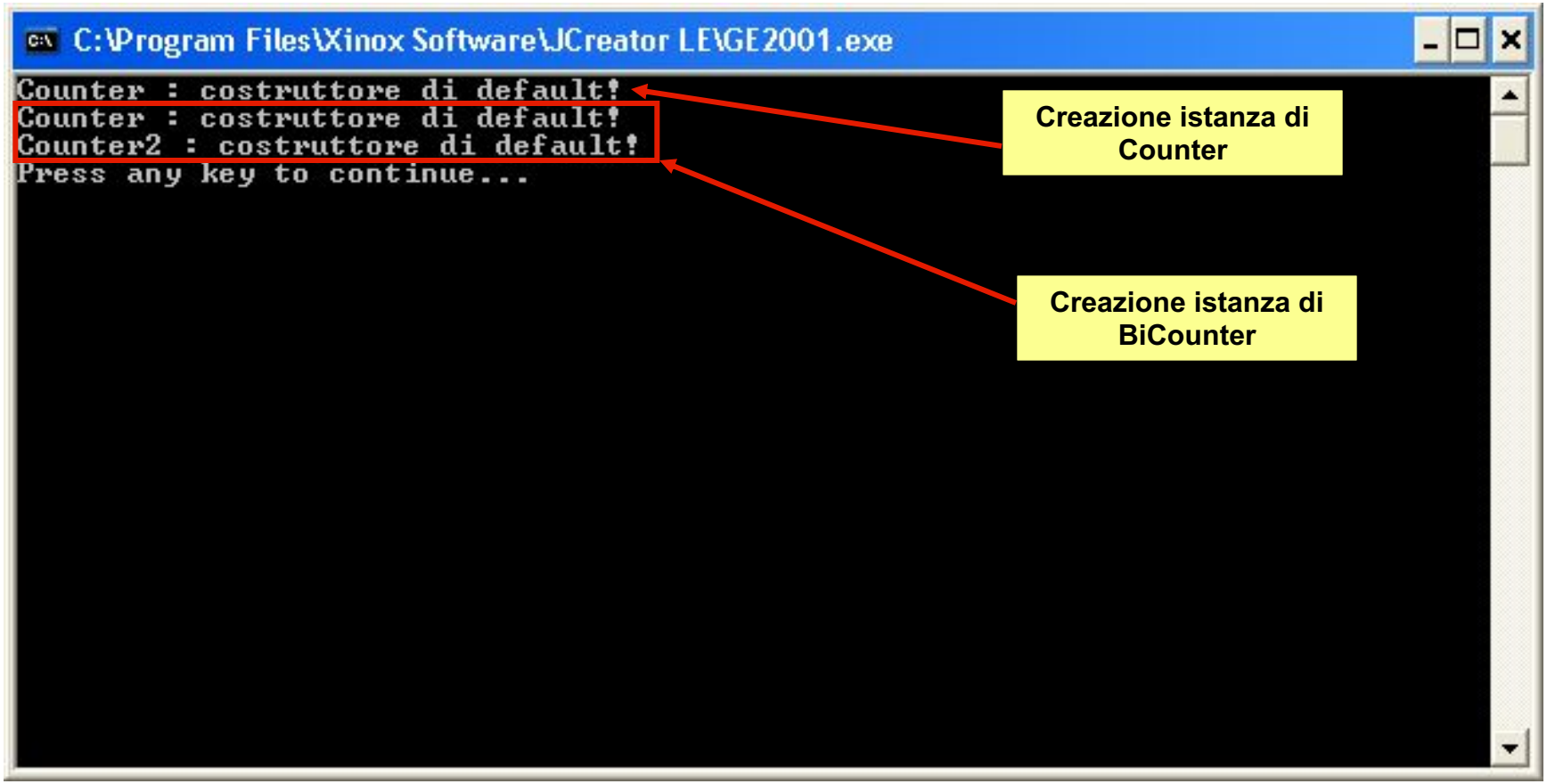
Seconda versione

```
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();
   System.out.println(c.getValue());
   BiCounter c1 =
     new BiCounter();
  System.out.println(c1.getValue());
   }
}
```

Risultato



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

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()
  {  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();
   System.out.println(c.getValue());
   BiCounter c1 =
     new BiCounter();
  System.out.println(c1.getValue());
   }
}
```

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);
     BiCounter 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:

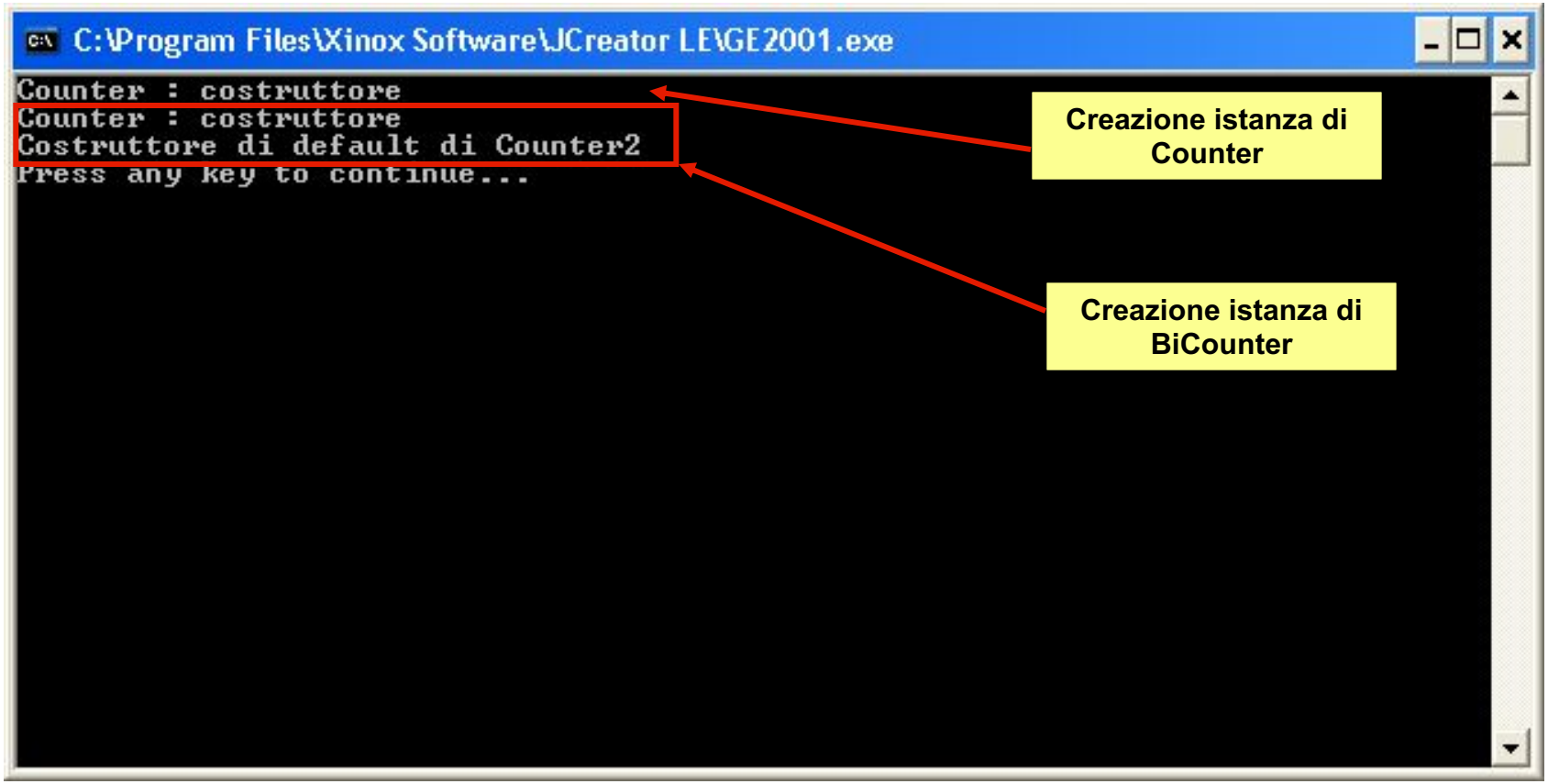
```
public class BiCounter
  extends Counter
{
  public BiCounter()
  {
    super(0);
    System.out.println(
      "Costruttore di default
       di Counter2");
  }
  public void dec()
  {
    val--;
  }
}
```

```
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 della classe base (super() con o senza argomenti) 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) se questo c'è

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.intValue(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 ….