APPROCCI PER IL RIUTILIZZO:

- *ricopiare manualmente il codice* della classe esistente e *cambiare quel che va cambiato*
- *creare un oggetto composto*
  - che incapsuli il componente esistente...
  - … gli "inoltri" le operazioni già previste…
  - … e crei, *sopra di esso*, le nuove operazioni richieste (eventualmente definendo nuovi dati)
  - ***sempre che ciò sia possibile!***
- *specializzare (per ereditarietà) la classe Counter*

Delega: Oggetti composti

```
public class CounterDec {
 private Counter c;
 public CounterDec() { c = new Counter(); }
 public CounterDec(int v){c=new Counter(v); }
 public void reset() { c.reset(); }
 public void inc()   { c.inc(); }
 public int getValue() { return c.getValue();
  }
 public void dec() {int v = c.getValue();
      c.reset();
      for (int i=0; i<v-1; i++) c.inc(); }
}
```

… e il loro uso:

```
public class EsempioNuovo {
 public static void main(String v[]) {
    CounterDec c = new CounterDec();
    c.reset();
    c.inc(); c.inc();
    System.out.println(c.getValue());
    c.dec();
    System.out.println(c.getValue());
 }
}
```

MODELLO OBJECT-BASED

- Questo modello 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' comunque un passo avanti rispetto alla programmazione procedurale

RIUTILIZZO

- Per potere effettivamente riusare codice (senza modificarlo) occorre poter:
  - Usare la classe Counter che ci va quasi bene ma non completamente
  - Utilizzarla per quel che mette a disposizione, ma crearne 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 quello 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!

EREDITARIETA'

- Ci consente di definire una nuova classe *a partire da una già esistente*
- Bisogna dire:
  - *quali dati* la nuova classe *ha in più* rispetto alla precedente
  - *quali metodi* la nuova classe *ha in più* rispetto alla precedente
  - *quali metodi* la nuova classe *modifica* rispetto alla precedente (overriding)

ESEMPIO

Dal contatore (solo in avanti) …

```
public class Counter {
  private int val;
  public Counter() { val = 1; }
  public Counter(int val) {
              this.val=val;
             }
  public void reset() { val = 0; }
  public void inc()   { val++; }
  public int getValue() { return val;}
}
```

Attenzione alla protezione!

ESEMPIO

… al contatore avanti/indietro *(con decremento)*

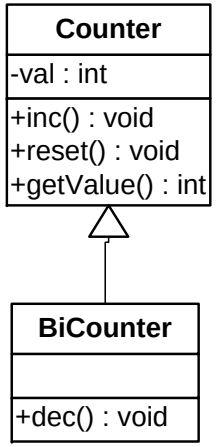
```
public class BiCounter extends Counter {
 public void dec() { val--; }
}
```

Questa nuova classe:

- **eredita da `Counter` il campo `val` (un `int`)**
- **eredita da `Counter` *tutti i metodi***
- **aggiunge a `Counter` il metodo `dec()`**

ESEMPIO UML

- **UML mette a disposizione una notazione grafica particolare per rappresentare classi e l'ereditarietà tra classi**
- **In UML gli elementi di una classe public / private vengono indicati con il simbolo + / - (vedi a lato)**
- **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**



… e il loro uso:

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

Non funziona (per via della protezione)

ESEMPIO EREDITARIETA'

Il contatore "riadattato"...

```
public class Counter {
  protected int val;
  public Counter() { val = 1; }
  public Counter(int v) { val = v; }
  public void reset() { val = 0; }
  public void inc()   { val++; }
  public int getValue() { return val;}
}
```

Nuovo tipo di protezione!

LA QUALIFICA `protected`

Un dato o un metodo `protected`

- **è come `private` (il default) per chiunque non sia una classe derivata**
- **ma *consente libero accesso* a una classe derivata, indipendentemente dal package in cui essa è definita.**

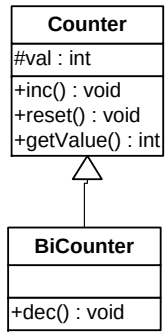
Occorre dunque *cambiare la protezione del campo `val` nella classe `Counter`*

ESEMPIO - UML

… e il contatore con decremento:

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

Ora funziona !



- **In UML gli elementi protected vengono indicati con il simbolo # (vedi sopra)**

EREDITARIETÀ

<u>Cosa si eredita</u>?

- **tutti i dati della classe base**
  - **anche quelli privati, a cui comunque la classe derivata non potrà accedere direttamente**
- **tutti i metodi...**
  - **anche quelli che la classe derivata non potrà usare direttamente**
- ***... tranne i costruttori*, perché sono specifici di quella particolare classe.**

ESEMPIO

Il contatore con decremento:

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

L'espressione `super(…)` invoca il costruttore della classe base <u>che corrisponde come numero e tipo di parametri</u> alla lista data.

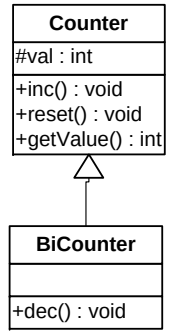
`super`: RIASSUNTO

La parola chiave ***`super`***

- nella forma `super(...)`, invoca un costruttore della classe base
- nella forma `super.val`, consente di accedere al campo `val` della classe base *(sempre che esso non sia private)*
- nella forma `super.metodo()`, consente di invocare il metodo `metodo()` della classe base *(sempre che esso non sia private)*

EREDITARIETÀ: CONSEGUENZE

- Se una classe eredita da un'altra, ***la classe derivata mantiene l'interfaccia di accesso della classe base***
  - anche se, naturalmente, può *specializzarla*, aggiungendo nuovi metodi



- Quindi, un `BiCounter` può essere usato *al posto di un* `Counter` se necessario
- *Ogni `BiCounter` è anche un `Counter` ! (NON E' VERO IL VICEVERSA)*

EREDITARIETÀ

- 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

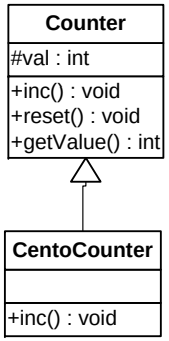
RIDEFINIZIONE DI METODI

- 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 CentoCounter

… il contatore limitato a 100 come valore massimo:

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



- Overriding del metodo inc()
- Scriviamo extends ma non stiamo aggiungendo, stiamo ridefinendo un metodo

OVERRIDING E OVERLOADING

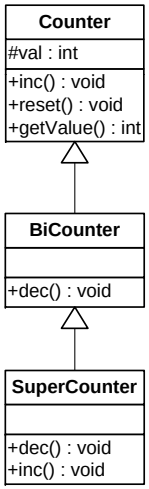
- Attenzione: non bisogna assolutamente confondere l'overloading con l'overriding!
- L' overloading ci permette di creare un nuovo metodo con lo stesso nome, ma parametri diversi
- Il nuovo metodo si affianca a quello già esistente, che continua a poter essere utilizzato
- L'overriding ci permette di 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

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)



OVERRIDING E SUPER

- 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:

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

- E' una forma ancora più flessibile di riuso

EREDITARIETÀ: CONSEGUENZE

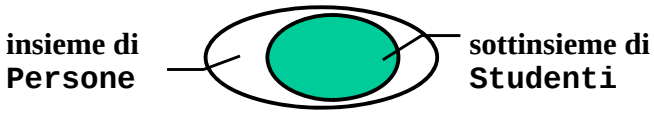
- Se ogni `BiCountr` è anche un `Counter`, è possibile usare un `BiCounter` al posto di un `Counter` *senza che il sistema se ne accorga!*

```
public class Esempio6 {
 public static void main(String args[]) {
  Counter  c1 = new Counter(10);
  BiCounter c2 = new BiCounter(20);
  c2.dec();     // OK: c2 è un BiCounter
  // c1.dec(); // NO: c1 è solo un Counter
  c1=c2;        // OK: c2 è anche un Counter
  // c2=c1;     // NO: c1 è solo un Counter
 }
}
```

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 (o subtyping)
- In modo simile si parla di ereditarietà di implementazione (o subclassing)
- Infatti una classe derivata comprende l'implementazione della classe base

UN ESEMPIO COMPLETO



- Una classe `Persona`
- e una sottoclasse `Studente`
  - *è aderente alla realtà*, perché è vero nel mondo reale che tutti gli studenti sono persone
  - compatibilità di tipo: potremo usare uno studente (che è *anche* una persona) ovunque sia richiesta una generica persona
    ma non viceversa: se serve uno studente, non si può accontentarsi di una generica persona!

LA CLASSE `Persona`

```
public class Persona {
 protected String nome;
 protected int anni;
 public Persona() {
      nome = "sconosciuto"; anni = 0; }
 public Persona(String n) {
      nome = n; anni = 0; }
 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");
 }
}
```

LA CLASSE Studente

```
public class Studente extends Persona {
 protected int matr;
 public Studente() {
      super(); matr = 9999; }
 public Studente(String n) {
      super(n); matr = 8888; }
 public Studente(String n, int a) {
      super(n,a); matr=7777; }
 public Studente(String n, int a, int m) {
      super(n,a); matr=m; }
 public void print() {
  super.print();
  System.out.println("Matricola = " + matr);
 }
}
```

LA CLASSE Studente

```
public class Studente extends Persona {
 protected int matr;
 pub
 pu
 pu
 pub
      super(n,a); matr=m; }
 public void print() {
  super.print();
  System.out.println("Matricola = " + matr);
 }
}
```

Ridefinisce il metodo `void print()`

- sovrascrive quello ereditato da `Persona`
- è una versione specializzata per `Studente` che però *riusa quello di `Persona`* (`super`), estendendolo per stampare la matricola.

LA CLASSE EsempioDiCittà

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

L'assegnamento `p=s` *non comporta perdita di informazione,* perché si assegnano *riferimenti* (gli oggetti puntati rimangono inalterati)

LA CLASSE EsempioDiCittà

```
public class EsempioDiCitta {
  s.print();  // stampa nome, età, matricola
  p=s;
  p.print();   // COSA STAMPA ???
 }
}
```

- Se prevale la natura del riferimento, stamperà solo nome ed età
- Se prevale la natura dell'oggetto puntato, stamperà nome, età *e matricola*

È un problema di POLIMORFISMO

PROBLEMA: cosa stampa?

- `p` è un riferimento a `Persona`
- *ma gli è stato assegnato un oggetto `Studente`*

POLIMORFISMO

In pratica, *dipende cosa prevale:*

- se prevale il tipo del riferimento, non ci sarà mai polimorfismo
  - in tal caso, `p.print()` stamperà solo nome ed età, perché verrà invocato il metodo `print()` della classe `Persona`
- se invece prevale il tipo dell'oggetto, allora c'è polimorfismo
  - in tal caso, `p.print()` stamperà nome, età e matricola, perché verrà invocato il metodo `print()` della classe `Studente`

SUBTYPING E POLIMORFISMO

- Il tipo del riferimento determina quello che si può fare: possiamo invocare solo i metodi definiti nella classe a cui il riferimento appartiene (subtyping)

```
Persona  p = new Persona("John");
Studente s = new Studente("Tom");
```

- Il tipo dell'istanza determina cosa viene effettivamente fatto: viene invocato il metodo definito nella classe a cui l'istanza appartiene (polimorfismo)

```
p=s;
p.print();
```

POLIMORFISMO

- 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
- Ereditarietà e polimorfismo sono i due principi che differenziano la programmazione object-oriented dalla programmazione object-based

POLIMORFISMO

In pratica, *dipende cosa prevale:*

- se prevale il tipo del riferimento, non ci sarà mai polimorfismo
  - in tal caso, `p.print()` stamperà solo nome ed età, perché verrà invocato il metodo `print()` della classe `Persona`
- se invece prevale il tipo dell'oggetto, allora c'è polimorfismo
  - in tal caso, `p.print()` stamperà nome, età e matricola, perché verrà invocato il metodo `print()` della classe `Studente`

Java supporta il Polimorfismo
→ prevale il tipo dell'oggetto

LATE BINDING: le chiamate ai metodi sono collegate *alla versione opportuna* del metodo *al momento della chiamata*, in base all'ogget-to effettivamente referenziato (a "run-time")

LA CLASSE `EsempioDiCittà`

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

`void print()` è un metodo polimorfo
Poiché `p` referenzia uno `Studente`,
stampa nome, età ***e matricola***

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! (ereditarietà di interfaccia - subtyping)

Downcasting

- Se scriviamo:

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

- Possiamo utilizzare i metodi definiti in Counter ma non quelli definiti in BiCounter
- Quindi non è ammessa un'istruzione come:

```
c.dec();
```

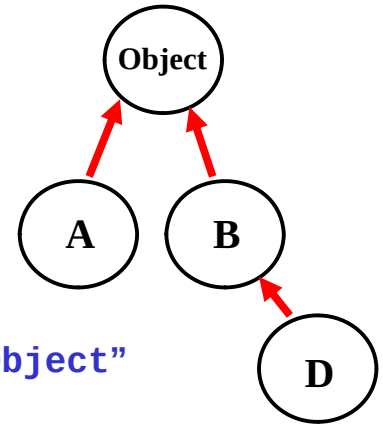
- Se vogliamo chiamare dec come possiamo fare?
- Dobbiamo ricorrere ad una conversione esplicita (typecasting)

```
BiCounter b = (BiCounter)c;
b.dec();
```

- Questa conversione viene chiamata downcasting

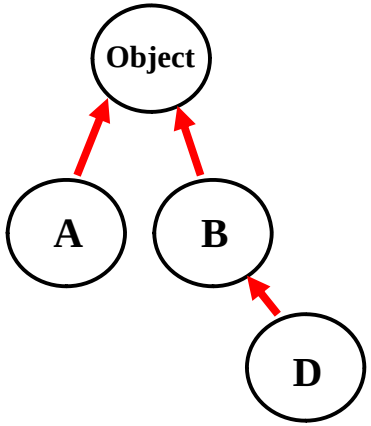
GERARCHIE DI EREDITARIETÀ

- La relazione di ereditarietà determina la nascita di *gerarchie* o *tassonomie* di ereditarietà
- In Java, ***ogni classe deriva implicitamente dalla classe-base `Object`***, che è la radice della gerarchia
- La frase "`class A`" sottintende "`extends Object`"



`Object`, LA RADICE DI TUTTO

- La classe base ***`Object`*** definisce alcuni metodi, *ereditati da tutte le altre classi:*
  - `clone()`
  - `equals()`
  - `toString()`
  - `...`



`Object`, LA RADICE DI TUTTO

Alcuni metodi interessanti:

- `protected Object clone()`
  - duplica l'oggetto su cui è invocato
  - è un metodo protetto: per rendere disponibile questa funzionalità all'esterno di una classe, occorre *ridefinirlo come pubblico*
- `public boolean equals(Object x)`
  - definisce il criterio di uguaglianza fra oggetti
  - per default, è l'uguaglianza fra riferimenti
- `public String toString()`
  - crea una rappresentazione dell'oggetto sotto forma di stringa

ESEMPIO

Una piccola classe (*eredita implicitamente* il metodo `toString()`da `Object`):

```
public class Deposito {
  float soldi;
  public Deposito() { soldi=0; }
  public Deposito(float s) { soldi=s; }
}
```

ESEMPIO

... e una classe che la usa:

```
public class Esempio7 {
 public static void main(String args[]){
  Deposito d1 = new Deposito(312);
  System.out.println(d1);
 }
}
```

Per stampare d1, viene invocato automaticamente il metodo `toString()`

È una forma compatta per `System.out.println(d1.toString());`

ESEMPIO

Se il `toString()` predefinito da `Object` non soddisfa, si può ridefinirlo:

```
public class Deposito {
  float soldi;
  public Deposito() { soldi=0; }
  public Deposito(float s) { soldi=s; }
  public String toString() {
   return "Deposito di valore " + soldi;
  }
}
```

Viene creato un nuovo oggetto `String` concatenando la frase "`Deposito di valore `" con il risultato di `Float.toString(soldi)`

ESEMPIO

L'output nel primo caso...

```
Deposito@712c1a3c
```

Identificativo univoco generato da Java: nome della classe + indirizzo dell'oggetto

... e nel secondo caso:

```
Deposito di valore 312.0
```

ESEMPIO `equals`

```
public class Esempio8 {
 public static void main(String args[])
 {Deposito d1 = new Deposito(312);
  Deposito d2 = new Deposito(104*3);
  if (d1.equals(d2))
       System.out.println("uguali!");
 }
}
```

ESEMPIO `equals`

Se l' `equals(Object x)` predefinito da `Object` non soddisfa, si può ridefinirlo:

```
public class Deposito {
  float soldi;
  public Deposito() { soldi=0; }
  public Deposito(float s) { soldi=s; }
  public boolean equals(Deposito x) {
    return (soldi==x.soldi); }
}
```

Consideriamo uguali due Deposito se e solo se hanno *identico valore*

ESEMPIO equals

Se l' equals(Object x) predefinito da Object non soddisfa, si può ridefinirlo:

```
public class Deposito {
  float soldi;
  public Deposito() { soldi=0; }
  public Deposito(float s) { soldi=s; }
  public boolean equals(Deposito x) {
    return (this.soldi==x.soldi); }
}
```

ESEMPIO equals

```
public class Esempio8 {
 public static void main(String args[])
 {Deposito d1 = new Deposito(312);
  Deposito d2 = new Deposito(104*3);
  if (d1.equals(d2))
       System.out.println("uguali!");
 }
}
```

- this è d1

ESEMPIO equals

```
public class Esempio8 {
 public static void main(String args[])
 {Deposito d1 = new Deposito(312);
  Deposito d2 = new Deposito(104*3);
  if (d2.equals(d1))
       System.out.println("uguali!");
 }
}
```

- this è d2

CLASSI FINALI

- Una classe finale (final) è una classe di cui si vuole *impedire a priori* che possano essere definite, un domani, delle sottoclassi
- Esempio:

```
public final class TheLastCounter
   extends CentoCounter {
   ...
}
```