Programmazione orientata agli oggetti
La classe Object, metodi e classi final, this

Leggere sez. 8.1.4, 8.3.2, 10.3.2, 10.3.3, 10.3.6, 10.3.7, 11.1.3 di Programmazione di base e avanzata con Java

La classe Object

- Negli esempi fatti nella prima parte del corso abbiamo definito alcune classi, per esempio Counter e Orologio, senza usare la parola chiave **extends**
- Questo non significa però che queste classi non abbiano una superclasse
- In Java tutte le classi discendono, direttamente o indirettamente, dalla classe **Object**
- Quando si definisce una classe senza specificare la clausola extends si sottointende **extends Object**
- Quindi tutte le classi ereditano i metodi della classe Object e, se lo ritengono opportuno, li possono ridefinire (overriding)

I metodi di Object

- Alcuni dei metodi di Object sono piuttosto interessanti e ci permettono di capire meglio alcuni meccanismi che abbiamo già usato:
- `public boolean equals(Object x)`
- Definisce il criterio di uguaglianza fra oggetti (uguaglianza dei riferimenti). Per avere un comportamento significativo dobbiamo ridefinirlo nelle classi derivate
- `public String toString()`
- Crea una rappresentazione dell'oggetto sotto forma di stringa. La definizione originale di questo metodo è poco significativa: scrive il nome della classe e un indirizzo: `Counter@712c1a3c`
- Anche in questo caso normalmente si ridefinisce il metodo nelle classi derivate

Deposito

- Scriviamo una semplice classe:

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

- Dal momento che non abbiamo specificato nessun extends la classe discende direttamente da Object
- In quanto tale eredita il metodo toString()
- Dal momento che non lo ridefinisce invocandolo viene eseguita la versione originale definita in Object

EsempioDeposito

- Scriviamo poi una semplice applicazione che la usa:

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

- A video otterremo:

```
Deposito@712c1a3c
```

Deposito 2

- Aggiungiamo a Deposito il metodo toString, ridefinendo così quello ereditato da Object (overriding)

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

EsempioDeposito 2

- Se usiamo la classe nell'applicazione precedente:

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

- A video otterremo:

```
Soldi: 312
```

System.out.println()

- Vediamo in dettaglio perché le cose funzionano in questo modo
- System.out è un attributo della classe System
- E' di tipo **PrintStrem**, una classe che serve per scrivere a video e che ha una definizione di questo tipo:

```
public class PrintStream extends FilterOutputStream
{
   public void println(Object x)
   {
     String s = x.toString();
     ...
   }
   ...
}
```

- In virtù del **subtyping** questo implica che possiamo passare come parametro qualunque oggetto, dal momento che tutte le classi discendono da Object
- In virtù del **polimorfismo** questo implica che se la classe dell'oggetto passato come parametro ridefinisce il metodo toString() verrà invocato il metodo ridefinito

Ora proviamo noi:

- Definiamo nella classe Counter il metodo toString() in modo da visualizzare la stringa:
  - “Sono un contatore di valore” + val

- Definiamo BiCounter come estensione di Counter, al quale aggiunge il metodo dec()
  - Se volete, ridefinite anche toString() per BiCounter
  - “Sono un contatore bidirezionale di valore” + val

- Definiamo il main in modo che crei c di tipo Counter e c1 di tipo BiCounter con valore iniziale 1, li incrementi 150 volte e li stampi (come oggetti)

Terza versione

Versione senza costruttore di default

```
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;}
public String toString()
  {return
     "Sono un contatore di
valore: " + val;
  }

}
```

```
public class BiCounter
  extends Counter
{
  public BiCounter()
  {  super(1);

System.out.println("Counter2:
       costruttore di default!");

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

public String toString()
  {return
     "Sono un contatore
bidirezionale di valore: " + val;
  }

}
```

main

```
public class Esempio
{
   public static void main(String[] args)
   {
    int n;
    Counter c = new Counter(1);
    BiCounter c1 = new BiCounter();
    for (int i=0;i<150;i++)
    { c.inc(); c1.inc(); }
    System.out.println(c);
    System.out.println(c1);
   }
 }
```

Metodi e classi final

- Abbiamo già visto la parola chiave **final** nella definizione delle costanti
- In Java **final** in generale serve per indicare qualcosa che non può cambiare
- Può essere utilizzato anche con i metodi e con le classi:
  - **Un metodo marcato come final** non può essere ridefinito (si inibisce l'overriding)
  - **Una classe marcata come final** non può essere estesa mediante ereditarietà. Non è cioè possibile creare sue sottoclassi

this

- Java definisce una parola chiave per rappresentare l'istanza corrente: **this**
- L'istanza corrente è quella su cui un metodo sta lavorando
- Questa parola chiave ha due utilizzi:
  - Eliminare i conflitti di nome quando un parametro o una variabile locale hanno lo stesso nome di un attributo dell'oggetto
  - Poter passare l'istanza corrente come parametro ad un metodo

Esempio 1: this per eliminare i conflitti di nome

- Scriviamo una variante di Counter in cui definiamo un costruttore che permette di stabilire il valore iniziale mediante un parametro:

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

- Avendo dato al parametro del costruttore lo stesso nome dell'attributo dobbiamo usare **this** per distinguere fra il parametro e l'attributo

Esempio 2: this come parametro

- Aggiungiamo alla classe Deposito il metodo print() che stampa a video l'**oggetto corrente**

```
public class Deposito
{
  float soldi;
  public Deposito() { soldi=0; }
  public Deposito(float s) { soldi=s; }
  public String toString()
  { return "Soldi: "+soldi }
  public void print()
  { System.out.println(this)
}
```

- System.out.println() richiede un oggetto come parametro
- Usiamo quindi **this** per indicare che vogliamo scrivere a video l'oggetto corrente