

---

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

# 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 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`
- `public boolean equals(Object x)`
- Definisce il criterio di uguaglianza fra oggetti (come uguaglianza dei riferimenti). Per avere un comportamento significativo dobbiamo ridefinirlo nelle classi derivate
- Anche in questo caso normalmente si ridefinisce il metodo nelle classi derivate

# Counter

---

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

- Dal momento che non abbiamo specificato nessun extends la classe discende direttamente da Object
- In quanto tale eredita il metodo toString()

# Counter

---

```
public class Counter extends Object
{
    protected int val;

    public void reset()
        { val = 0; }

    public void inc()
        { val++; }

    public int getValue()
        { return val; }

}
```

- Dal momento che non abbiamo specificato nessun extends la classe discende direttamente da Object
- In quanto tale eredita il metodo toString()

# EsempioCounter

---

- Scriviamo poi una semplice applicazione che la usa:

```
public class EsempioCounter
{
    public static void main(String args[])
    {
        Counter c = new Counter();
        c.reset();
        System.out.println(c);
    }
}
```

- A video otterremo:

Counter@712c1a3c

## Counter v.2

---

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

```
public class Counter
{
    protected int val;

    public void reset()
        { val = 0; }

    public void inc()
        { val++; }

    public int getValue()
        { return val; }

    public String toString()
        { return "Valore: "+val; }

}
```

## EsempioCounter v.2

---

- Scriviamo poi una semplice applicazione che la usa:

```
public class EsempioCounter
{
    public static void main(String args[])
    {
        Counter c = new Counter();
        c.reset();
        System.out.println(c);
    }
}
```

- A video otterremo:

Valore: 0

# System.out.println()

---

- Vediamo in dettaglio perché le cose funzionano in questo modo
- System.out è un attributo della classe System
- E' di tipo **PrintWriter** (lo vedremo nelle lezioni sull'I/O), una classe che serve per scrivere a video e che ha una definizione di questo tipo:

```
public class PrintWriter extends Writer
{
    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()
  - 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, li incrementi 150 volte e li stampi (come oggetti)

# Terza versione

Versione 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--; }
}
```

...

```
Counter c= new Counter();
c.reset();
Counter c1=new BiCounter();
c1.reset();
for(int i=1;i<=150;i++)
    {c.inc(); c1.inc(); }
```

```
System.out.println( c.toString() );
System.out.println( c1.toString() );
```

# Terza versione

Versione senza costruttori

```
public class Counter
{
    protected int val;

    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 void dec()
    { val--; }

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

# 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(0);

        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)
    {
        Counter c = new Counter();
        Counter c1 = new BiCounter();
        c.reset();
        c1.reset();
        for (int i=0;i<150;i++)
        { c.inc(); c1.inc(); }
        System.out.println(c);
        System.out.println(c1);
    }
}
```

# main

---

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

## Altri metodi di Object

---

- La stessa cosa (**overriding**) può essere fatta per altri metodi ereditati da Object:
- `public boolean equals(Object x)`
- Definisce il criterio di uguaglianza fra oggetti (come uguaglianza dei riferimenti)
- Verifica se: `this==x`
- Per avere un comportamento più significativo dobbiamo ridefinirlo nelle classi derivate

# Terza versione

Versione 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--; }
}
```

Non stampa uguali

```
...
Counter c= new Counter();
c.reset();
Counter c1=new BiCounter();
c1.reset();
if (c.equals(c1)) //c==c1
    System.out.println( "uguali");
System.out.println( c );
System.out.println( c1 );
```

## Ora proviamo noi: equals

---

- Definiamo nella classe Counter il metodo equals(Object x) in modo da verificare se i due oggetti (this e x) hanno lo stesso valore per l'attributo val

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

    public boolean equals(Object x)
    { return (this.val==(Counterx).val); }
}
```

- Downcasting



# Terza versione

Versione senza costruttori

```
public class Counter
{
    protected int val;

    public void reset()
    { val = 0; }
    public void inc()
    { val++; }
    public int getValue()
    { return val;}
}
```

```
public boolean equals(Object
x)
{ return
(this.val==((Counter)x).val);
}
}
```

Stampa uguali (this.val e x.val valgono 0)

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

```
...
Counter c= new Counter();
c.reset();
Counter c1=new BiCounter();
c1.reset();
if (c.equals(c1)) //c==c1
    System.out.println( "uguali");
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 Counter il metodo print() che stampa a video l'**oggetto corrente**

```
public class Counter
{
    protected int val;
    public void reset() { val = 0; }
    public void inc() { val++; }
    public int getValue() { return val; }
    public String toString(){ return "Valore: "+val; }
    public void print()
    { System.out.println(this.toString()); }
}
```

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

## Esempio 2: this come parametro

---

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

```
public class Counter
{
    protected int val;
    public void reset() { val = 0; }
    public void inc() { val++; }
    public int getValue() { return val; }
    public String toString(){ return "Valore: "+val; }
    public void print()
    { System.out.println(this) ; }
}
```

- La chiamata di **toString()** può anche essere omessa

## The last example ...

---

```
public class Main {  
    public static void main(String[] args) {  
        int n;  
        Counter c;  
        BiCounter c1;  
        c = new Counter();  
        c.reset();  
        c1 = new BiCounter();  
        c1.reset();  
        for (int i=0; i<150; i++) c.inc();  
        for (int i=0; i<150; i++) c1.inc();  
        c.print();  
        c1.print();  
    }  
}
```