

# 07

## Ereditarietà

Mirko Viroli  
`mirko.viroli@unibo.it`

C.D.L. Ingegneria e Scienze Informatiche  
ALMA MATER STUDIORUM—Università di Bologna, Cesena

a.a. 2022/2023

## Goal della lezione

- Illustrare il riuso via ereditarietà
- Introdurre i vari meccanismi collegati all'ereditarietà

## Argomenti

- Estensione di classi
- Livello d'accesso `protected`
- Overriding dei metodi
- Gestione dei costruttori e chiamate `super`
- Il modificatore `final` su classi e metodi

- 1 Riuso via ereditarietà
- 2 Uno scenario completo
- 3 Ulteriori dettagli

# Ereditarietà

È un meccanismo che consente di definire una nuova classe **specializzandone** una esistente, ossia “ereditando” i suoi campi e metodi (quelli privati non sono visibili), possibilmente/eventualmente modificando/aggiungendo campi/metodi, e quindi riusando codice già scritto e testato.

## L'ereditarietà è un concetto chiave dell'OO

- È connesso al meccanismo delle interfacce
- È uno degli elementi chiave insieme a incapsulamento e interfacce
- Non riguarda solo il riuso di codice, ma influenza anche il polimorfismo conseguente

## Solito approccio

- Illustreremo i meccanismi base attraverso semplici classi
- Successivamente recupereremo l'importanza nei casi reali
- Utilizzeremo l'idea di **contatore**

# Scenari di riuso ed estensione

- Data una classe, realizzarne un'altra con caratteristiche solo in parte diverse (o nuove)
- Come sopra, ma senza disporre dei sorgenti della classe originaria (p.e., la classe di partenza è di libreria)
- Data una classe, crearne una più specializzata (ad esempio, più robusta e sicura anche se meno prestante, o viceversa, più prestante)
- Creare gerarchie di classi ossia di comportamenti

## Esempio base: Counter

```
1 public class Counter {  
2  
3     private int value;  
4  
5     public Counter(final int initialValue) {  
6         this.value = initialValue;  
7     }  
8  
9     public void increment() {  
10        this.value++;  
11    }  
12  
13    public int getValue() {  
14        return this.value;  
15    }  
16 }
```

# Uso della classe Counter

```
1 public class UseCounter {  
2     public static void main(String[] s) {  
3         final Counter c = new Counter(0);  
4  
5         System.out.println(c.getValue()); // 0  
6         c.increment();  
7         c.increment();  
8         System.out.println(c.getValue()); // 2  
9     }  
10 }
```

# Una nuova classe senza riuso: MultiCounter

```
1 public class MultiCounter {
2
3     private int value;
4
5     public MultiCounter(final int initialValue) {
6         this.value = initialValue;
7     }
8
9     public void increment() {
10         this.value++;
11     }
12
13     public int getValue() {
14         return this.value;
15     }
16
17     /* Nuovo metodo */
18     public void multiIncrement(final int n) {
19         for (int i = 0; i < n; i++) {
20             this.increment();
21         }
22     }
23 }
```



# Uso della classe MultiCounter

```
1 public class UseMultiCounter {  
2     public static void main(String[] s) {  
3         final MultiCounter mc = new MultiCounter(10);  
4         System.out.println(mc.getValue()); // 10  
5         mc.increment();  
6         mc.increment();  
7         System.out.println(mc.getValue()); // 12  
8         mc.multiIncrement(10);  
9         System.out.println(mc.getValue()); // 22  
10    }  
11 }
```

## Versione con riuso via composizione: MultiCounter2

```
1 public class MultiCounter2 {
2
3     private Counter counter;
4
5     public MultiCounter2(final int initialValue) {
6         this.counter = new Counter(initialValue);
7     }
8
9     public void increment() {
10         this.counter.increment();
11     }
12
13     public int getValue() {
14         return this.counter.getValue();
15     }
16
17     /* Nuovo metodo */
18     public void multiIncrement(final int n) {
19         for (int i = 0; i < n; i++) {
20             this.counter.increment();
21         }
22     }
23 }
```

# La necessità di estendere e modificare codice

## Una tipica situazione

- È tipico nei progetti software, accorgersi di dover creare anche versioni modificate delle classi esistenti
- Appoggiarsi al “copia e incolla” di codice è **sempre** sconsigliabile (principio DRY), perché tende a spargere errori in tutto il codice, e complica la manutenzione
- Ottenere riuso via composizione (ossia delegazione) è in generale una **ottima soluzione**... ma ve ne sono di alternative, che possono avere dei vantaggi

## Si usa il meccanismo di ereditarietà

- Definizione: `class C extends D {..}`
- La nuova classe C eredita campi/metodi/costruttori non privati di D
  - ▶ Eredita anche campi/metodi privati, ma non sono accessibili da C
  - ▶ I costruttori di D non sono direttamente richiamabili con la **new**, bisogna sempre definirne di nuovi
- Terminologia: D superclasse, o classe base, o classe padre
- Terminologia: C sottoclasse, o classe figlio, o specializzazione
- Nota: non serve disporre dei sorgenti di D, basta il codice binario, infatti in futuro estenderemo classi di libreria senza averne il sorgente

# Una nuova versione di MultiCounter

```
1  /* Si noti la clausola extends */
2  public class MultiCounter extends Counter {
3
4      /*
5       * I costruttori vanno ridefiniti. Devono tuttavia richiamare
6       * quelli ereditati dalla sopraclasse
7       */
8      public MultiCounter(int initialValue) {
9          super(initialValue);
10     }
11
12     // increment e getValue automaticamente ereditati
13
14     // si aggiunge multiIncrement
15     public void multiIncrement(final int n) {
16         for (int i = 0; i < n; i++) {
17             this.increment();
18         }
19     }
20 }
```

## Ridefiniamo la classe `MultiCounter` come estensione di `Counter`

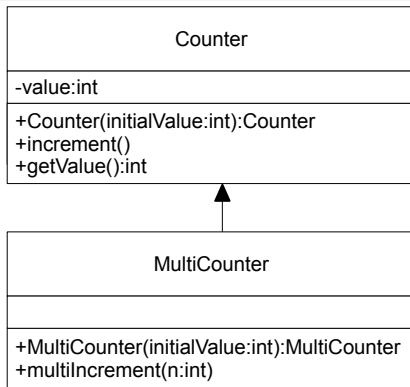
- Definiamo il nuovo metodo `multiIncrement()`
- Definiamo il costruttore necessario
  - ▶ Il costruttore di una sottoclasse può cominciare con l'istruzione `super`, che chiama un costruttore (non privato) della classe padre
  - ▶ Se non lo fa, si chiama il costruttore di default del padre
  - ▶ Senza costruttori, si ha al solito solo quello di default
- `UseMultiCounter` continua a funzionare!

## Il senso della definizione

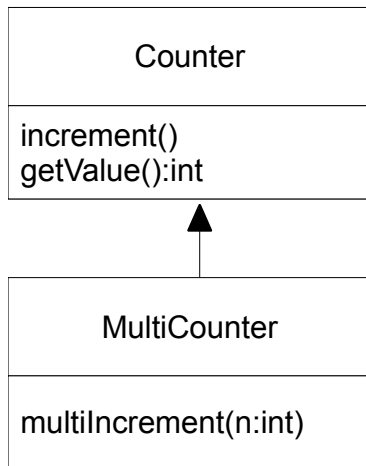
- Un oggetto di `MultiCounter` è simile ad un oggetto di `Counter`
  - ▶ Ha i metodi `increment()` e `getValue()`
  - ▶ Ha anche il campo `value` (che in effetti è incrementato), anche se essendo privato è inaccessibile dal codice della classe `MultiCounter`
- Due modifiche necessarie rispetto a `Counter`: metodo `multiIncrement()` e ridefinizione del costruttore

# Notazione UML per l'estensione

- Arco a linea continua (punta a triangolo pieno) per la relazione “**extends**”
- Archi raggruppati per migliorare la resa grafica



# Notazione UML – versione semplificata per il design



# Livello d'accesso `protected`

## Usabile per le proprietà d'una classe

- È un livello intermedio fra `public` e `private`
- Indica che la proprietà (campo, metodo, costruttore) è accessibile dalla classe corrente, da una sottoclasse, e dalle sottoclassi delle sottoclassi (ricorsivamente)
- Cavillo: `protected` in Java consente accesso anche da tutto il package, ovvero è “meno stringente” rispetto al livello di default package `protected`

## A cosa serve?

- Consente alle sottoclassi di accedere ad informazioni della sopraclasse che non si vogliono far vedere agli utilizzatori
- Molto spesso usato a posteriori rimpiazzando un `private`

## Esempio classe `BiCounter` – contatore bidirezionale

- Un contatore con anche il metodo `decrement`
- Irrealizzabile senza rendere accessibile il campo `counter`



# Un contatore estendibile: ExtendibleCounter

```
1  /* Il nome ExtendibleCounter è di comodo, più propriamente
2     andrebbe chiamata semplicemente Counter */
3
4  public class ExtendibleCounter {
5
6     /* campo value protetto */
7     protected int value;
8
9     public ExtendibleCounter(final int initialValue) {
10         this.value = initialValue;
11     }
12
13     public void increment() {
14         this.value++;
15     }
16
17     public int getValue() {
18         return this.value;
19     }
20 }
```

# Classe MultiCounter

```
1 public class MultiCounter extends ExtendibleCounter {  
2  
3     public MultiCounter(final int initialValue) {  
4         super(initialValue);  
5     }  
6  
7     public void multiIncrement(final int n) {  
8         // Ora realizzabile più efficientemente  
9         if (n > 0) {  
10             this.value = this.value + n;  
11         }  
12     }  
13 }
```

# Classe BiCounter

```
1 public class BiCounter extends ExtendibleCounter {  
2  
3     public BiCounter(final int initialValue) {  
4         super(initialValue);  
5     }  
6  
7     public void decrement() {  
8         /* Ora this.counter è accessibile */  
9         this.value--;  
10    }  
11 }
```

# Overriding di metodi

## Estensione e modifica

- Quando si crea una nuova classe per estensione, molto spesso non è sufficiente aggiungere nuove funzionalità
- A volte serve anche modificare alcune di quelle disponibili, eventualmente anche stravolgendone il funzionamento originario
- Questo è realizzabile riscrivendo nella sottoclasse uno (o più) dei metodi della superclasse (ossia, facendone l'**overriding**)
- Se necessario, il metodo riscritto può invocare la versione del padre usando il receiver speciale **super**

## Esempio

- Creare un contatore che, giunto ad un certo limite, non prosegue più
- È necessario fare overriding del metodo `increment()`
- Un ulteriore metodo `getter` ispeziona il raggiungimento del limite

# Classe LimitCounter

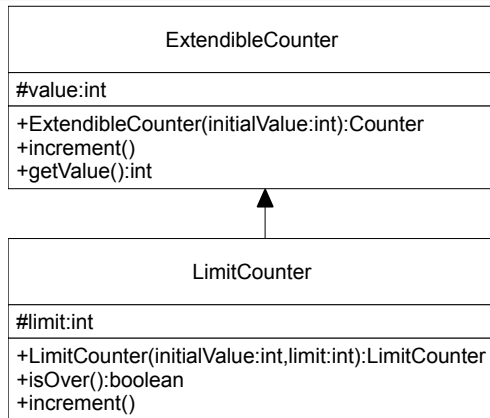
```
1 public class LimitCounter extends ExtendibleCounter {
2
3     /* Aggiungo un campo, che tiene il limite */
4     protected final int limit;
5
6     public LimitCounter(final int limit) {
7         super(0);
8         this.limit = limit;
9     }
10
11     public boolean isOver() {
12         return this.getValue() == this.limit;
13     }
14
15     /* Overriding del metodo increment() */
16     public void increment() {
17         if (!this.isOver()) {
18             super.increment();
19         }
20     }
21 }
```

# Uso della classe LimitCounter

```
1 public class UseLimitCounter {
2     public static void main(String[] s) {
3         final LimitCounter c = new LimitCounter(5);
4         System.out.println(c.getValue()); // 0
5         System.out.println(c.isOver()); // false
6         c.increment();
7         c.increment();
8         System.out.println(c.getValue()); // 2
9         System.out.println(c.isOver()); // false
10        c.increment();
11        c.increment();
12        c.increment();
13        c.increment();
14        c.increment();
15        c.increment();
16        c.increment();
17        System.out.println(c.getValue()); // 5
18        System.out.println(c.isOver()); // true
19    }
20 }
```

# Notazione UML

- I campi/metodi protetti si annotano con un “#”
- I metodi overridden si riportano anche nella sottoclasse



# Campi `protected`? Meglio di no...

## Incapsulamento e `protected`

- L'incapsulamento prevede di nascondere completamente l'implementazione (e quindi i campi) alle altre classi
- Un campo `protected` violerebbe l'incapsulamento nei confronti della sottoclasse
- La modifica dell'implementazione in una sovraclassa rischia di propagarsi sulle sottoclassi

## Best practice

- Campi sempre privati
- Se l'accesso in lettura o scrittura è necessario solo nelle sottoclassi, allora utilizzare getter o setter `protected`



# Estendibilità via getter/setter protected

```
1 public class ExtensibleCounter {
2
3     private int value;
4
5     public ExtensibleCounter(final int initialValue) {
6         this.value = initialValue;
7     }
8
9     public void increment() {
10         this.value++;
11     }
12
13     public int getValue() {
14         return this.value;
15     }
16
17     protected void setValue(int value) {
18         this.value = value;
19     }
20 }
```

```
1 public class MultiCounter extends ExtensibleCounter {
2
3     public MultiCounter(final int initialValue) {
4         super(initialValue);
5     }
6
7     public void multiIncrement(final int n) {
8         // Ora realizzabile più efficientemente
9         if (n > 0) {
10             this.setValue(this.getValue() + n);
11         }
12     }
13 }
```

- 1 Riuso via ereditarietà
- 2 Uno scenario completo
- 3 Ulteriori dettagli

# Una applicazione allo scenario domotica

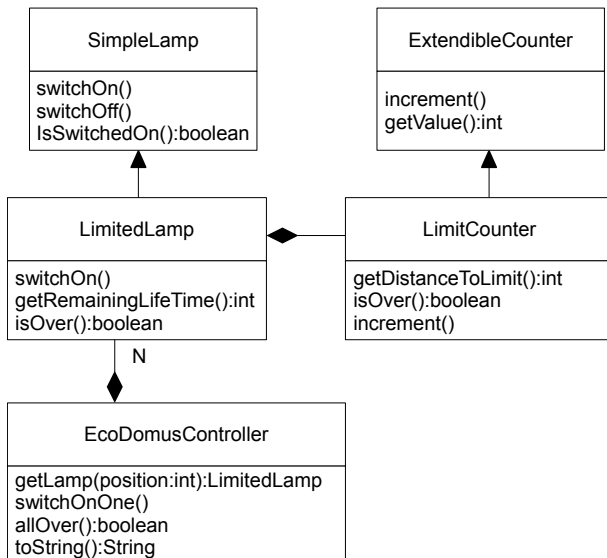
## Elementi

- Usiamo `LimitCounter`
- Definiamo una `LimitedLamp` (via estensione) che contiene un contatore, e che ha un tempo di vita basato sul numero di accensioni ammesse
- Un `EcoDomusController` si compone di  $n$  `LimitedLamp`, e ha la possibilità di verificare se tutte le lampadine sono esaurite, e di accendere la lampadina alla quale è rimasto più tempo di vita

## Note sulla soluzione

- Una alternativa era far sì che `EcoDomusController` componesse  $n$  `SimpleLamp` e  $n$  `LimitCounter`
- `LimitedLamp` realizza alcuni metodi per delegazione al suo contatore
- Per ora, non prevediamo l'aspetto di interazione con l'utente

# Diagramma UML complessivo



# LimitCounter

```
1 public class LimitCounter extends ExtendibleCounter {
2
3     private final int limit;
4
5     public LimitCounter(final int initialValue, final int limit) {
6         super(initialValue);
7         this.limit = limit;
8     }
9
10    public boolean isOver() {
11        return this.getDistanceToLimit() == 0;
12    }
13
14    public int getDistanceToLimit() {
15        return this.limit - this.value;
16    }
17
18    public void increment() {
19        if (!this.isOver()) {
20            super.increment();
21        }
22    }
23 }
```

# SimpleLamp

```
1 public class SimpleLamp {  
2  
3     private boolean switchedOn;  
4  
5     public SimpleLamp() {  
6         this.switchedOn = false;  
7     }  
8  
9     public void switchOn() {  
10         this.switchedOn = true;  
11     }  
12  
13     public void switchOff() {  
14         this.switchedOn = false;  
15     }  
16  
17     public boolean isSwitchedOn() {  
18         return this.switchedOn;  
19     }  
20 }
```

# LimitedLamp

```
1 public class LimitedLamp extends SimpleLamp {
2
3     private LimitCounter counter;
4
5     public LimitedLamp(final int limit) {
6         super(); // Questa istruzione è opzionale
7         this.counter = new LimitCounter(0, limit);
8     }
9
10    public void switchOn() {
11        if (!this.isSwitchedOn() && !this.counter.isOver()) {
12            super.switchOn();
13            this.counter.increment();
14        }
15    }
16
17    public int getRemainingLifeTime() { // delegazione a counter
18        return this.counter.getDistanceToLimit();
19    }
20
21    public boolean isOver() { // delegazione a counter
22        return this.counter.isOver();
23    }
24 }
```

# EcoDomusController pt 1

```
1 public class EcoDomusController {
2
3     /* Compongo n LimitedLamp */
4     final private LimitedLamp[] lamps;
5
6     public EcoDomusController(final int size, final int lampsLimit) {
7         this.lamps = new LimitedLamp[size];
8         for (int i = 0; i < size; i++) {
9             this.lamps[i] = new LimitedLamp(lampsLimit);
10        }
11    }
12
13    public LimitedLamp getLamp(final int position) {
14        return this.lamps[position];
15    }
16
17    private LimitedLamp toBeUsedNext() {
18        LimitedLamp best = null;
19        for (final LimitedLamp lamp : this.lamps) {
20            if (!lamp.isSwitchedOn() &&
21                (best == null ||
22                 lamp.getRemainingLifeTime() > best.getRemainingLifeTime())) {
23                best = lamp;
24            }
25        }
26        return best;
27    }
28 }
```



## EcoDomusController pt 2

```
1  /* Accendo una lampadina spenta, scegliendola in modo economico */
2  public void switchOnOne() {
3      final LimitedLamp lamp = this.toBeUsedNext();
4      if (lamp != null) {
5          lamp.switchOn();
6      }
7  }
8
9
10 /* Verifico se sono tutti accesi */
11 public boolean allOver() {
12     for (final LimitedLamp lamp : this.lamps) {
13         if (!lamp.isOver()) {
14             return false;
15         }
16     }
17     return true;
18 }
19
20 public String toString() {
21     String s = "";
22     for (final LimitedLamp lamp : this.lamps) {
23         s += (lamp.isSwitchedOn() ? "on" : "off");
24         s += "(" + lamp.getRemainingLifeTime() + ")" + " | ";
25     }
26     return s;
```

# UseEcoDomusController

```
1 public class UseEcoDomusController extends Object{
2     public static void main(String[] s) {
3         // Simulazione sessione di lavoro
4         final EcoDomusController controller;
5         controller = new EcoDomusController(5, 10);
6         System.out.println(controller);
7         // off(10) | off(10) | off(10) | off(10) | off(10) |
8         final LimitedLamp l = controller.getLamp(0);
9         l.switchOn();
10        l.switchOff();
11        l.switchOn();
12        System.out.println(controller);
13        // on(8) | off(10) | off(10) | off(10) | off(10) |
14        controller.switchOnOne();
15        controller.switchOnOne();
16        controller.switchOnOne();
17        controller.switchOnOne();
18        System.out.println(controller);
19        // on(8) | on(9) | on(9) | on(9) | on(9) |
20    }
```

- 1 Riuso via ereditarietà
- 2 Uno scenario completo
- 3 Ulteriori dettagli

# Ereditarietà e costruttori

## Scenario standard

- Assumiamo si stia costruendo una catena di sottoclassi
- Ogni classe introduce alcuni campi, che si aggiungono a quelli della superclasse a formare la struttura di un oggetto in memoria

## Linee guida per la singola classe

- Dovrà definire tutti i costruttori necessari, seguendo l'approccio visto
- Ogni costruttore dovrà preoccuparsi di:
  1. Chiamare l'opportuno costruttore padre come prima istruzione (**super**), altrimenti il costruttore di default verrà chiamato, se c'è
  2. Inizializzare propriamente i campi localmente definiti

## Ordine operazioni a seguito di una **new**

- Prima si crea l'oggetto con tutti i campi non inizializzati
- Il codice dei costruttori sarà eseguito, dalle superclassi in giù

# Analisi: cosa succede?

```
1 class A {  
2     protected int i;  
3  
4     public A(int i) {  
5         System.out.println("A().. prima " + this.i);  
6         this.i = i;  
7         System.out.println("A().. dopo " + this.i);  
8     }  
9 }
```

```
1 class B extends A {  
2     protected String s;  
3  
4     public B(String s, int i) {  
5         super(i);  
6         System.out.println("B().. prima " + this.s + " " + this.i);  
7         this.s = s;  
8         System.out.println("B().. dopo " + this.s + " " + this.i);  
9     }  
10    public static void main(String[] s) {  
11        B b = new B("prova", 5); // Cosa succede?  
12    }  
13 }
```

# Chiamate di metodo alla superclasse (**super**)

## Chiamate **super**

- Una sottoclasse *C* può includere una invocazione del tipo **super**.*m*(..args..)
- Non solo in caso di overriding
- Cosa ci aspettiamo succeda?

## Semantica

- Accade quello che accadrebbe se la classe corrente non avesse il metodo *m*, ossia viene eseguito il metodo *m* della superclasse
  - ▶ O, se anche lì assente, quello nella sopraclasse più specifica che lo definisce
- Se tale metodo al suo interno chiama un altro metodo *n* (su **this**), allora si ritorna a considerare la versione più specifica a partire dalla classe di partenza *C*

# Analisi: cosa succede?

```
1 class C {  
2     protected int i;  
3  
4     void m() {  
5         System.out.println("C.m.. prima " + i);  
6         this.i++;  
7         System.out.println("C.m.. dopo " + i);  
8     }  
9 }
```

```
1 class D extends C {  
2     D(int i) {  
3         this.i = i;  
4     }  
5     void m() {  
6         super.m();  
7         System.out.println("D.m.. dopo " + this.i);  
8     }  
9     public static void main(String[] s) {  
10         new D(5).m(); // Cosa succede?  
11     }  
12 }
```

# Altra analisi: cosa succede?

```
1 class E {  
2     protected int i;  
3  
4     void m() {  
5         this.i++;  
6         this.n();  
7     }  
8     void n() {  
9         this.i = this.i + 10;  
10    }  
11 }
```

```
1 class F extends E {  
2     void n() {  
3         this.i = this.i + 100;  
4     }  
5     public static void main(String[] s) {  
6         F f = new F();  
7         f.i = 10;  
8         f.m();  
9         System.out.println("" + f.i);  
10    }  
11 }
```



## Un esempio: riprendiamo LimitCounter

```
1 public class LimitCounter extends ExtendibleCounter {
2
3     private final int limit;
4
5     public LimitCounter(final int initialValue, final int limit) {
6         super(initialValue);
7         this.limit = limit;
8     }
9
10    public boolean isOver() {
11        return this.getDistanceToLimit() == 0;
12    }
13
14    public int getDistanceToLimit() {
15        return this.limit - this.value;
16    }
17
18    public void increment() {
19        if (!this.isOver()) {
20            super.increment();
21        }
22    }
23 }
```

# Un esempio: nuova specializzazione

Cosa succede chiamando `increment()` su un `UnlimitedCounter`?

- Non avendo fatto overriding, si chiama la versione di `LimitCounter`
- In `LimitCounter` si chiama `this.isOver()` che chiama `this.getDistanceToLimit()`
- La versione di `this.getDistanceToLimit()` eseguita è quella di `UnlimitedCounter`

```
1 public class UnlimitedCounter extends LimitCounter {  
2  
3     public UnlimitedCounter() {  
4         super(0, Integer.MAX_VALUE);  
5     }  
6  
7     public int getDistanceToLimit() {  
8         // Quindi il contatore non scade mai  
9         return Integer.MAX_VALUE;  
10    }  
11 }
```

# Uso di UnlimitedCounter

```
1 public class UseUnlimitedCounter {
2     public static void main(String[] s) {
3         final UnlimitedCounter uc = new UnlimitedCounter();
4         System.out.println("isOver: " + uc.isOver()); // false
5         System.out.println("LifeTime: " + uc.getDistanceToLimit());
6         uc.increment();
7         uc.increment();
8         uc.increment();
9         System.out.println("isOver: " + uc.isOver()); // false
10        System.out.println("LifeTime: " + uc.getDistanceToLimit());
11    }
12 }
```

# La tabella dei metodi virtuali

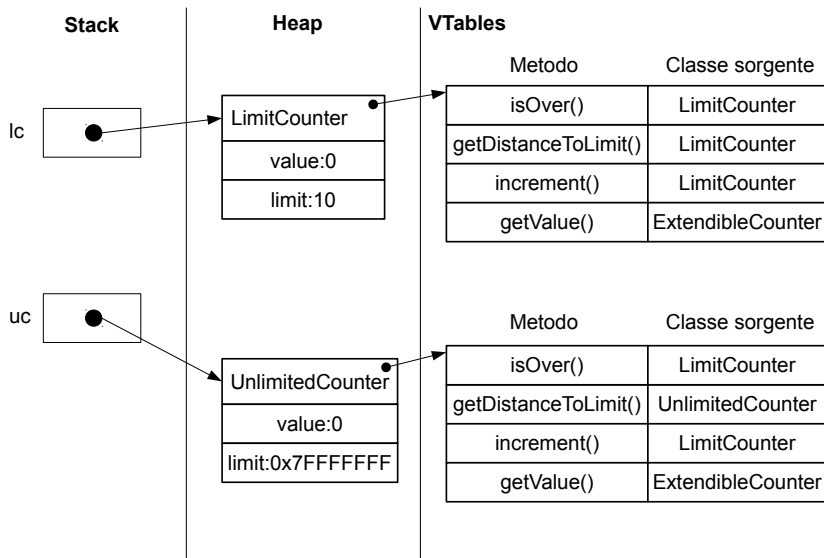
## Anche detta: vtable, call table, dispatch table

- ogni classe *C* ne ha una, ed è accessibile ai suoi oggetti
- ad ogni metodo definito (o ereditato) in *C*, associa il codice corrispondente da eseguire, ossia la classe che riporta il body
- le chiamate da risolvere con tale tabella sono quelle con **late binding**
- è una struttura che rende efficiente il polimorfismo fra classi (che vedremo)
- è utile conoscerla anche se non è detto che la JVM usi esattamente tale struttura
- fa comprendere il funzionamento di `this.` e `super.`

## Esempio

Come sono fatte le tabelle relative alle classi `LimitedCounter` e `UnlimitedCounter` nell'esempio precedente?

# Esempio gestione memoria: stack/heap/vtables



# Il modificatore `final`

## Problema

- Tramite l'overriding e le chiamate `super` è possibile prendere classi esistenti e modificarle con grande flessibilità
- Questo introduce problemi di sicurezza, specialmente connessi al polimorfismo che vedremo nella prossima lezione

## Soluzione: `final`

- Oltre che per i campi (e argomenti di funzione o variabili, come già visto), è possibile dichiarare `final` anche metodi e intere classi
- Un metodo `final` è un metodo che NON può essere ri-definito per overriding
- Una classe `final` non può essere estesa

## Nelle librerie Java

- Moltissime classi sono `final`, ad esempio `String`

## Regole per fare l'overriding di un metodo $M$

- La nuova versione deve avere esattamente la stessa signature
  - È possibile estendere la visibilità di un metodo (da `protected` a `public`)
  - Non è possibile limitare la visibilità di un metodo (p.e. da `public` a `protected`, o da `public` a `private`)
  - È possibile indicare il metodo `final`
- ⇒ sono tutte conseguenze del principio di sostituibilità

# La classe Object

## Estensione di default

- Una classe deve per forza estendere da qualcosa
- Se non lo fa, si assume che estenda `java.lang.Object`
- Quindi ogni classe eredita (indirettamente) da `Object`
- `Object` è la radice della gerarchia di ereditarietà di Java

## Classe Object

Fornisce alcuni metodi di utilità generale

- `toString()`, che stampa informazioni sulla classe e la posizione in memoria dell'oggetto
- `clone()`, per clonare un oggetto
- `equals()` e `hashCode()`, usati nelle collection
- `notify()` e `wait()`, usati nella gestione dei thread
- ...