07
Ereditarietà

Mirko Viroli
mirko.viroli@unibo.it

C.D.L. Ingegneria e Scienze Informatiche
Alma Mater Studiorum—Università di Bologna, Cesena

a.a. 2025/2026

Outline

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

Alcuni scenari di riuso per le classi

Riuso con estensione e/o sostituibilità

- Data una classe, realizzarne un'altra con caratteristiche solo in parte diverse (o nuove)
  - avendo o non avendo a disposizione i sorgenti della classe originaria
  - per crearne una specializzazione (ad esempio, una versione più robusta, o più prestante, o con più funzionalità)
- . . . ovviamente, senza violare il principio DRY
- La nuova classe potrebbe dover essere sostituibile alla precedente
  - fino a formare vere e proprie gerarchie "is-a"
  - ad esempio, data la classe `Vehicle`, potrei poi creare le classi `Car`, `Truck`, `Boat`, che sono tra loro abbastanza simili

Outline

Esempio base: Counter

```java
public class Counter {

  private int value;

  public Counter(final int initialValue) {
    this.value = initialValue;
  }

  public void increment() {
    this.value++;
  }

  public int getValue() {
    return this.value;
  }
}
```

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

```java
public class MultiCounter {

    private int value;

    public MultiCounter(final int initialValue) {
        this.value = initialValue;
    }

    public void increment() {
        this.value++;
    }

    public int getValue() {
        return this.value;
    }

    /* Nuovo metodo */
    public void multiIncrement(final int n) {
        for (int i = 0; i < n; i++) {
            this.increment();
        }
    }
}
```

Uso della classe MultiCounter

```java
public class UseMultiCounter {
    public static void main(String[] s) {
        final MultiCounter mc = new MultiCounter(10);
        System.out.println(mc.getValue()); // 10
        mc.increment();
        mc.increment();
        System.out.println(mc.getValue()); // 12
        mc.multiIncrement(10);
        System.out.println(mc.getValue()); // 22
    }
}
```

Versione con riuso via composizione: MultiCounter2

```java
public class MultiCounter2 {

    private final Counter counter;

    public MultiCounter2(final int initialValue) {
        this.counter = new Counter(initialValue);
    }

    public void increment() {
        this.counter.increment();
    }

    public int getValue() {
        return this.counter.getValue();
    }

    /* Nuovo metodo */
    public void multiIncrement(final int n) {
        for (int i = 0; i < n; i++) {
            this.counter.increment();
        }
    }
}
```

Riuso via composizione

Uso di composizione/delegazione

- si incapsula un oggetto della classe da modificare/estendere
- si forniscono i metodi necessari, che posson essere più o meno
- alcuni di questi delegano il lavoro all'oggetto incapsulato

Analisi del meccanismo

- nella programmazione moderna è considerata una ottima soluzione
- possibile limite: esplicitare le delegazione vìola un poco DRY
- si può/deve usare in combinazione con le interfacce per gestire sostituibilità

Possibile design per ottenere anche sostituibilità

- interfaccia `CounterAbs`, implementazione `CounterImpl`, e poi `MultiCounterImpl` che implementa `CounterAbs` e si compone di un `CounterImpl` (nomi non ottimali)

Interfaccia Counter e classe UseMultiCounter

```java
public interface CounterAbs {

    void increment();

    int getValue();

}
```

```java
public class UseMultiCounter {
    public static void main(String[] s) {
        final MultiCounterImpl mc = new MultiCounterImpl(10);
        final CounterAbs c = mc;

        System.out.println(c.getValue()); // 10
        c.increment();
        c.increment();
        System.out.println(c.getValue()); // 12
        mc.multiIncrement(10);
        System.out.println(c.getValue()); // 22
    }
}
```

Classe CounterImpl

```java
public class CounterImpl implements CounterAbs {

  private int value;

  public CounterImpl(final int initialValue) {
    this.value = initialValue;
  }

  public void increment() {
    this.value++;
  }

  public int getValue() {
    return this.value;
  }
}
```

Classe MultiCounterImpl

```java
public class MultiCounterImpl implements CounterAbs {

    private final CounterAbs counter;

    public MultiCounterImpl(final int initialValue) {
        this.counter = new CounterImpl(initialValue);
    }

    public void increment() {
        this.counter.increment();
    }

    public int getValue() {
        return this.counter.getValue();
    }

    /* Nuovo metodo */
    public void multiIncrement(final int n) {
        for (int i = 0; i < n; i++) {
            this.increment();
        }
    }
}
```

Outline

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, che è simile a quello delle interfacce e che descriveremo in dettaglio

Solito approccio

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

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

```java
/* Si noti la clausola extends, per estendere una classe */
public class MultiCounter extends Counter {

    /*
     * I costruttori vanno ridefiniti. Devono tuttavia richiamare
     * con super(...) quelli ereditati dalla sopraclasse
     */
    public MultiCounter(int initialValue) {
      super(initialValue);
    }

    // increment e getValue automaticamente ereditati

    // si aggiunge multiIncrement
    public void multiIncrement(final int n) {
        for (int i = 0; i < n; i++) {
            this.increment();
        }
    }
}
```

Razionale

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

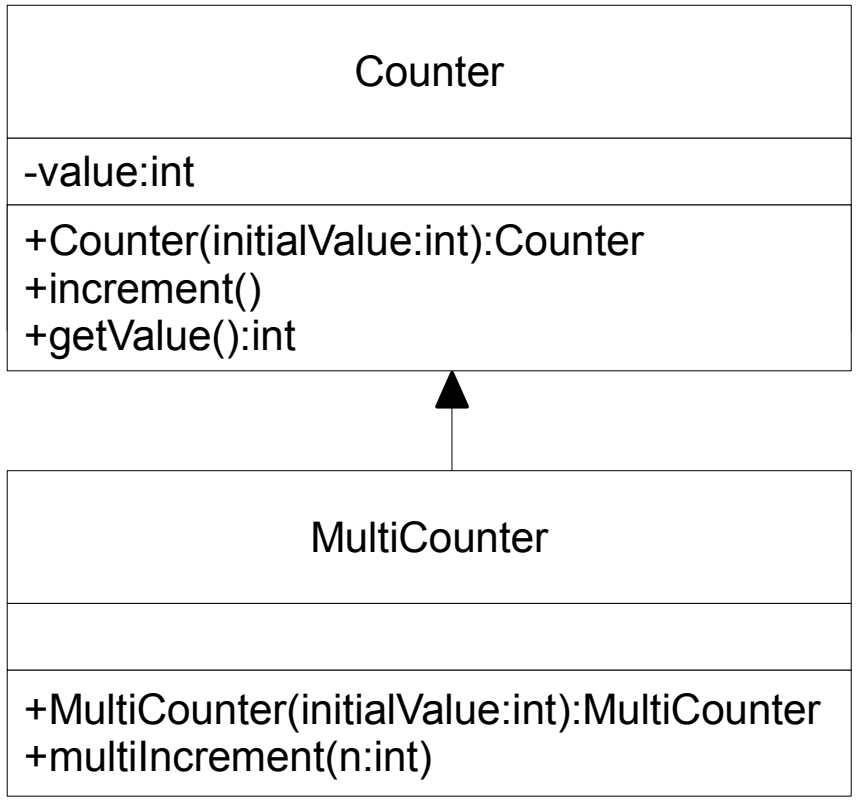
- Definiamo il nuovo metodo `multiIncrement()`
- Definiamo il costruttore necessario
  - Il costruttore di una sottoclasse può cominciare con l'istruzione `super`, che chiama un costruttore (non privato) della classe padre
  - Se non lo fa, si chiama il costruttore di default del padre
  - Senza costruttori,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

Il senso della definizione

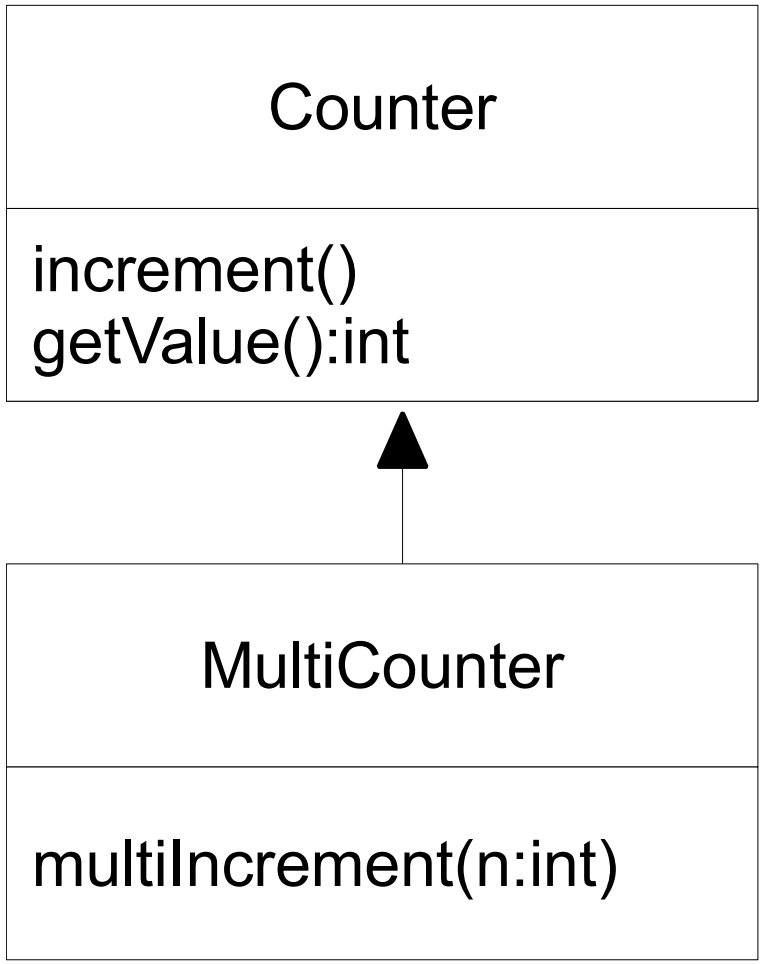
- Un oggetto di `MultiCounter` è simile ad un oggetto di `Counter`
  - Ha i metodi `increment()` e `getValue()`
  - Ha anche il campo `value` (che in effetti è incrementato), anche se essendo privato è inaccessibile dal codice della classe `MultiCounter`
- Due modifiche necessarie rispetto a `Counter`: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

Notazione UML per l'estensione

- Arco a linea continua (punta a triangolo vuoto) 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; serve infatti a realizzare framework estendibili "anche da fuori"

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`
- Preferibile avere campi privati e getter/setter protetti

Esempio classe `BiCounter` – contatore bidirezionale

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

Un contatore estendibile: ExtendibleCounter

```java
/* Il nome ExtendibleCounter è di comodo, più propriamente
 andrebbe chiamata semplicemente Counter */

public class ExtendibleCounter {

    /* campo value protetto */
    protected int value;

    public ExtendibleCounter(final int initialValue) {
        this.value = initialValue;
    }

    public void increment() {
        this.value++;
    }

    public int getValue() {
        return this.value;
    }
}
```

Classe MultiCounter

```java
public class MultiCounter extends ExtendibleCounter {

    public MultiCounter(final int initialValue) {
        super(initialValue);
    }

    public void multiIncrement(final int n) {
        // Ora realizzabile più efficientemente
        if (n > 0) {
            this.value = this.value + n;
        }
    }
}
```

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`
- Di norma, i metodi che fanno override hanno una annotazione `@Override` – ma al momento non ce ne occupiamo

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

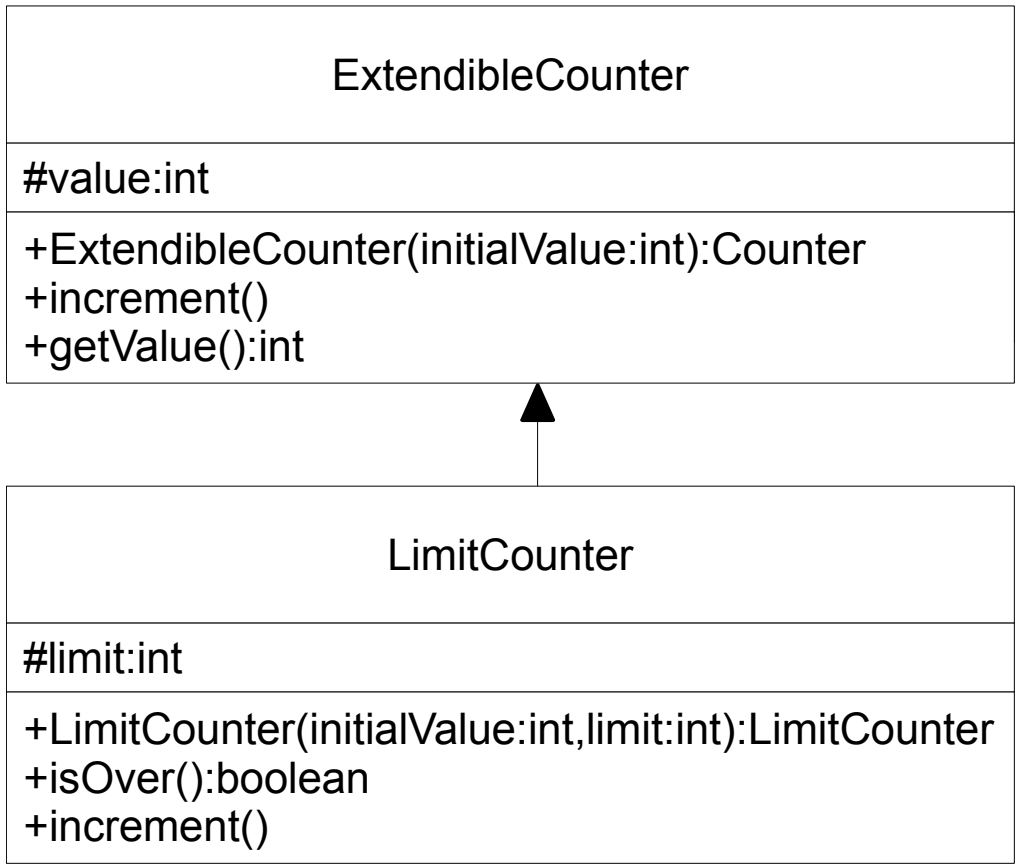
```java
public class LimitCounter extends ExtendibleCounter {

    /* Aggiungo un campo, che tiene il limite */
    protected final int limit;

    public LimitCounter(final int limit) {
        super(0);
        this.limit = limit;
    }

    public boolean isOver() {
        return this.getValue() == this.limit;
    }

    /* Overriding del metodo increment() */
    public void increment() {
        if (!this.isOver()) {
            super.increment();
        }
    }
}
```

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'incapsulamente 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 sopraclasse 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 proteceted

```java
public class ExtendibleCounter {

    private int value;

    public ExtendibleCounter(final int initialValue) {
        this.value = initialValue;
    }

    public void increment() {
        this.value++;
    }

    public int getValue() {
        return this.value;
    }

    protected void setValue(int value) {
      this.value = value;
    }
}
```

```java
public class MultiCounter extends ExtendibleCounter {

    public MultiCounter(final int initialValue) {
        super(initialValue);
    }

    public void multiIncrement(final int n) {
        // Ora realizzabile più efficientemente
        if (n > 0) {
            this.setValue(this.getValue() + n);
        }
    }
}
```

Outline

Una applicazione allo scenario domotica

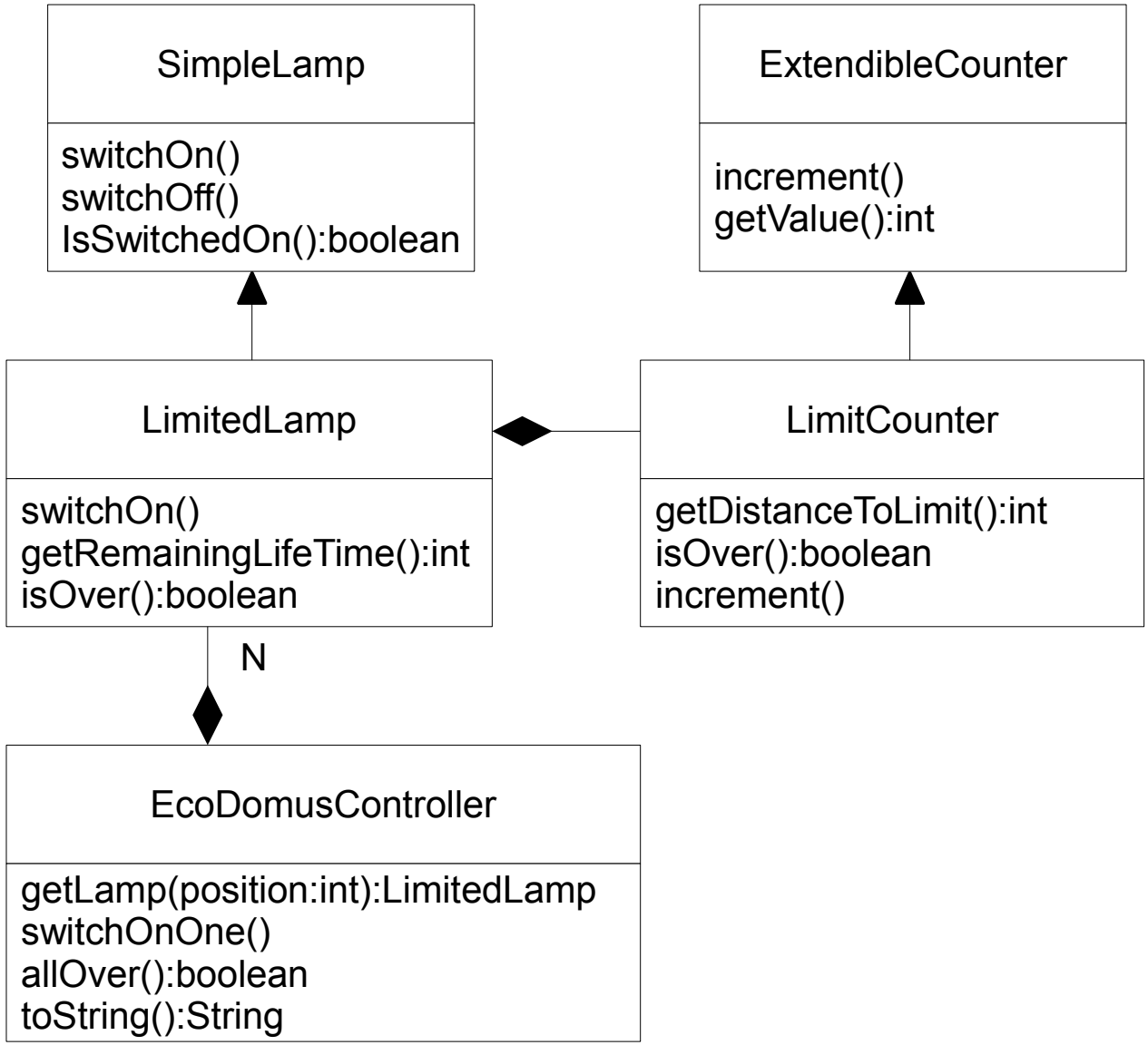
Elementi

- 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

```java
public class LimitCounter extends ExtendibleCounter {

    private final int limit;

    public LimitCounter(final int initialValue, final int limit) {
        super(initialValue);
        this.limit = limit;
    }

    public boolean isOver() {
        return this.getDistanceToLimit() == 0;
    }

    public int getDistanceToLimit() {
        return this.limit - this.value;
    }

    public void increment() {
        if (!this.isOver()) {
            super.increment();
        }
    }
}
```

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

```java
public class LimitedLamp extends SimpleLamp {

  private LimitCounter counter;

  public LimitedLamp(final int limit) {
    super(); // Questa istruzione è opzionale
    this.counter = new LimitCounter(0, limit);
  }

  public void switchOn() {
    if (!this.isSwitchedOn() && !this.counter.isOver()) {
      super.switchOn();
      this.counter.increment();
    }
  }

  public int getRemainingLifeTime() { // delegazione a counter
    return this.counter.getDistanceToLimit();
  }

  public boolean isOver() { // delegazione a counter
    return this.counter.isOver();
  }
}
```

EcoDomusController pt 1

```java
public class EcoDomusController {

  /* Compongo n LimitedLamp */
  final private LimitedLamp[] lamps;

  public EcoDomusController(final int size, final int lampsLimit) {
    this.lamps = new LimitedLamp[size];
    for (int i = 0; i < size; i++) {
      this.lamps[i] = new LimitedLamp(lampsLimit);
    }
  }

  public LimitedLamp getLamp(final int position) {
    return this.lamps[position];
  }

  private LimitedLamp toBeUsedNext() {
    LimitedLamp best = null;
    for (final LimitedLamp lamp : this.lamps) {
      if (!lamp.isSwitchedOn() &&
        ( best == null ||
          lamp.getRemainingLifeTime() > best.getRemainingLifeTime())) {
        best = lamp;
      }
    }
    return best;
  }
```

EcoDomusController pt 2

```
1
2   /* Accendo una lampadina spenta, scegliendola in modo economico */
3   public void switchOnOne() {
4     final LimitedLamp lamp = this.toBeUsedNext();
5     if (lamp != null) {
6       lamp.switchOn();
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

```java
public class UseEcoDomusController extends Object{
  public static void main(String[] s) {
    // Simulazione sessione di lavoro
    final EcoDomusController controller;
    controller = new EcoDomusController(5, 10);
    System.out.println(controller);
    // off(10) | off(10) | off(10) | off(10) | off(10) |
    final LimitedLamp l = controller.getLamp(0);
    l.switchOn();
    l.switchOff();
    l.switchOn();
    System.out.println(controller);
    // on(8) | off(10) | off(10) | off(10) | off(10) |
    controller.switchOnOne();
    controller.switchOnOne();
    controller.switchOnOne();
    controller.switchOnOne();
    System.out.println(controller);
    // on(8) | on(9) | on(9) | on(9) | on(9) |
  }
}
```

Outline

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?

```java
class A {
  protected int i;

  public A(int i) {
    System.out.println("A().. prima " + this.i);
    this.i = i;
    System.out.println("A().. dopo " + this.i);
  }
}
```

```java
class B extends A {
  protected String s;

  public B(String s, int i) {
    super(i);
    System.out.println("B().. prima " + this.s + " " + this.i);
    this.s = s;
    System.out.println("B().. dopo " + this.s + " " + this.i);
  }
  public static void main(String[] s) {
    B b = new B("prova", 5); // Cosa succede?
  }
}
```

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?

```java
class C {
  protected int i;

  void m() {
    System.out.println("C.m.. prima " + i);
    this.i++;
    System.out.println("C.m.. dopo " + i);
  }
}
```

```java
class D extends C {
  D(int i) {
    this.i = i;
  }
  void m() {
    super.m();
    System.out.println("D.m.. dopo " + this.i);
  }
  public static void main(String[] s) {
    new D(5).m(); // Cosa succede?
  }
}
```

Altra analisi: cosa succede?

```java
class E {
  protected int i;

  void m() {
    this.i++;
    this.n();
  }
  void n() {
    this.i = this.i + 10;
  }
}
```

```java
class F extends E {
  void n() {
    this.i = this.i + 100;
  }
  public static void main(String[] s) {
    F f = new F();
    f.i = 10;
    f.m();
    System.out.println("" + f.i);
  }
}
```

Un esempio: riprendiamo LimitCounter

```java
public class LimitCounter extends ExtendibleCounter {

    private final int limit;

    public LimitCounter(final int initialValue, final int limit) {
        super(initialValue);
        this.limit = limit;
    }

    public boolean isOver() {
        return this.getDistanceToLimit() == 0;
    }

    public int getDistanceToLimit() {
        return this.limit - this.value;
    }

    public void increment() {
        if (!this.isOver()) {
            super.increment();
        }
    }
}
```

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

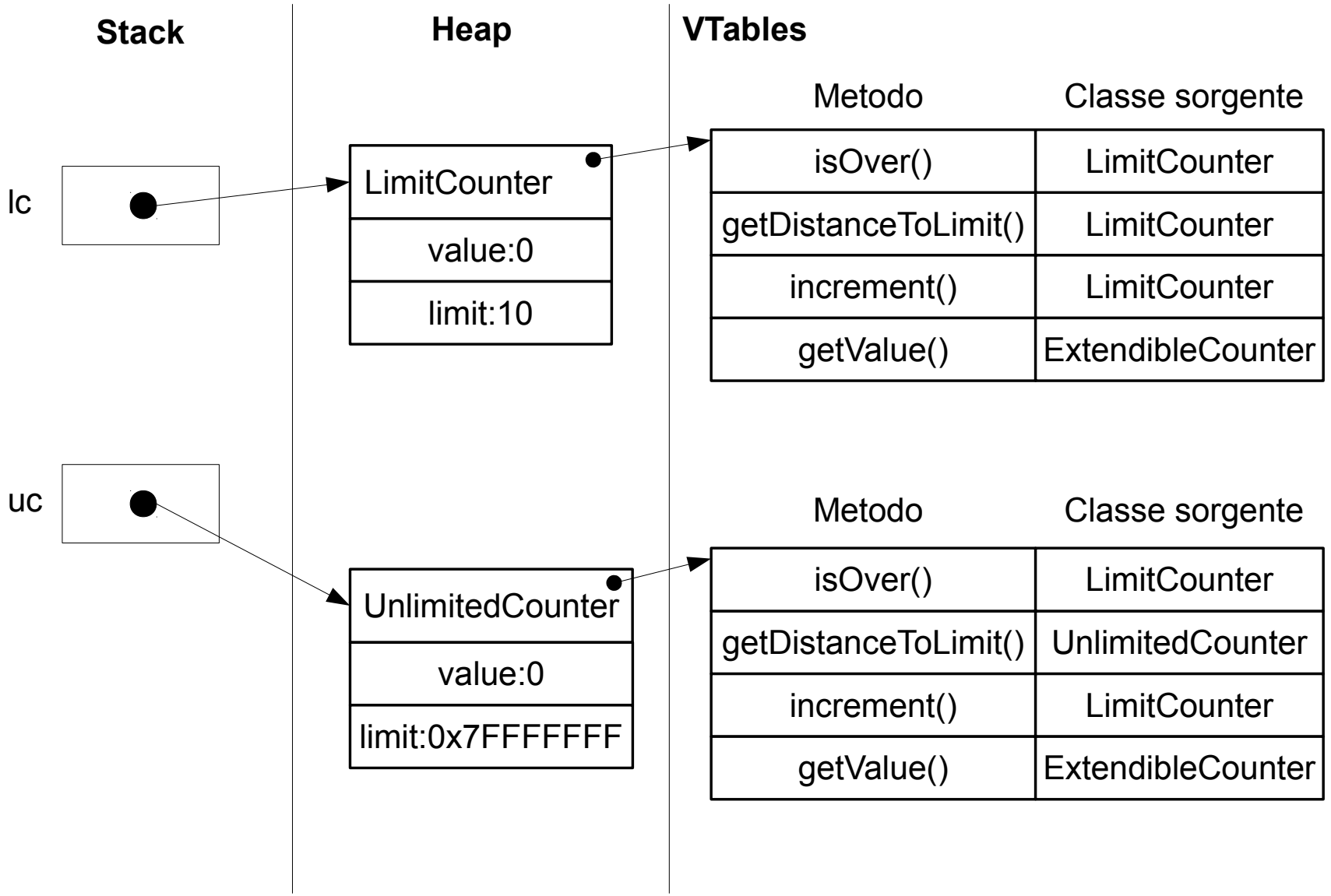
Anche detta: vtable, call table, dispatch table

- 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 dichiare `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`

Overriding e controllo d'accesso

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 possile limitare la visibilità di un metodo (p.e. da `public` a `protected`, o da `public` a `private`)
- È possibile indicare il metodo `final`

$\Rightarrow$ 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
- ...