05
Incapsulamento

Mirko Viroli
mirko.viroli@unibo.it

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

a.a. 2022/2023

Outline

Goal della lezione

- Illustrare concetti generali di incapsulamento e information hiding
- Mostrare tecniche di programmazione standard
- Fornire primi esempi di classi ben progettate

Argomenti

- Convenzioni su formattazione
- Incapsulamento in Java
- Una metodologia di progettazione
- Ulteriori convenzioni

Ambiente integrato Visual Studio Code

Funzionalità

- Supporto multi-linguaggio
- Vasta quantità di plug-in
- Supporta editing "intelligente", compilazione on-the-fly, debug, esecuzione
- Concetto di progetto

Metodologia

- creazione progetto, package, classi
- esecuzione dalla classe col main
- possibilità di importare/esportare file ZIP (p.e., codice aula)

Outline

Dai meccanismi alla buona progettazione/programmazione

La nostra analisi dell'OO in Java finora, ci ha insegnato:

- Parte imperativa/procedurale di Java (tipi primitivi, operatori, cicli)
- Classi, oggetti, costruttori, campi, metodi
- Codice statico, controllo d'accesso

Detto ciò, come realizziamo un buon sistema?

Come programmiamo il sistema

1. per giungere al risultato voluto, e
2. così che sia facilmente manutenibile (estendibile, flessibile, leggibile)?

$\Rightarrow$ un percorso articolato: muoviamo i primi passi..

Passi

Il nostro approccio

1. Ricapitoleremo le principali convenzioni sul codice Java
2. Anticiperemo alcune tecniche di programmazione efficace basate sulle tecniche di incapsulamento, e conseguenti al fondamentale principio di decomposizione ($\Leftarrow$ aspetto cruciale della OO)
3. Daremo qualche linea guida utile in futuro per costruire sistemi di più grosse dimensioni

Nota: tutte queste tecniche e linee guida sono necessarie per gestire il livello di articolazione del linguaggio Java, ossia per rendere il suo uso più semplice possibile

Outline

Convenzioni sulla formattazione, pt 1

Formattazione generale

- Indentazione di 2-4 caratteri (comunque non 1, non 10..)
- Linee lunghe non più di 90 caratteri – spezzare in modo coerente

Commenti nel codice

- "`// ...`" su una linea
- "`/* ..*/`" su più linee per commentare sezioni
- "`/**..*/`" su più linee per commenti che generano documentazione

Istruzioni

- Definizione di variabile: una per linea, solo quando/se serve
- Meglio inizializzare sempre le variabili!
- Una sola istruzione per riga

Convenzioni sulla formattazione, pt 2

Costrutti vari

- Apertura graffa: a fine linea della dichiarazione
- Chiusura graffa: in linea a sè, dove inizia la linea che la apre
- Metodi/costruttori separati da una linea vuota (poche separazioni altrimenti)
- Usare graffe anche con blocchi ad uno statement
- Non usare assegnamenti dentro a espressioni
- Disambiguare priorità non banali fra operatori, con parentesi

Nomi

- Classi (e interface): iniziano con maiuscola
- Metodi, campi, variabili: iniziano con minuscola
- Se nome strutturato usare "camelCasing"
- Campi costanti: tutte maiuscole con eventuale separatore "_"

Convenzioni sulla formattazione, pt 3

Ordine delle proprietà della classe

- Campi statici (pubblici, poi privati)
- Campi istanza (pubblici, poi privati)
- Costruttori (pubblici, poi privati)
- Metodi (raggruppati per ruolo)

Nota finale

- L'uso delle corrette convenzioni rende il codice molto più leggibile, ma anche a volte meno conciso
- Nelle slide è impossibile mostrarlo sempre in questo modo

Esempio: Point3D pt 1

```java
/**
 * Point3D is an example showcasing some functionality of OO in Java, focussing
 * on formatting. All properties have default (package) access
 */

public class Point3D {

  /*
   * A Point3D object is made of three doubles, and has some manipulation functions
   */

  // A constant ZERO point
  static final Point3D ZERO = new Point3D(0, 0, 0);

  double x; // x coordinate
  double y; // y coordinate
  double z; // z coordinate

  /* A standard constructor initializing fields */
  Point3D(double x, double y, double z) {
    this.x = x;
    this.y = y;
    this.z = z;
  }

  /* A simple function that extracts useful info from a point */

  double getSquareModule() {
    return this.x * this.x + this.y * this.y + this.z * this.z;
  }
```

Esempio: Point3D pt 2

```
  /* The following three are called selector methods */

  double getX() {
    return this.x;
  }

  double getY() {
    return this.y;
  }

  double getZ() {
    return this.z;
  }

  /* An example of a method changing the object state */
  void translate(double x, double y, double z) {
    this.x += x;
    this.y += y;
    this.z += z;
  }

  /* We also provide an example static method */
  static Point3D max(Point3D[] ps) {
    Point3D max = Point3D.ZERO; // Inizializing max
    for (Point3D elem : ps) { // Iterating over all input points
      if (elem.getSquareModule() > max.getSquareModule()) {
        max = elem; // Update max if needed..
      }
    }
    return max; // Return max
  }
}
```

Outline

divide et impera

Dividi e conquista: approccio top-down

- La soluzione di un problema complesso avviene dividendolo in problemi più semplici, tra loro indipendenti
- La suddivisione è spesso multi-livello

Esempi

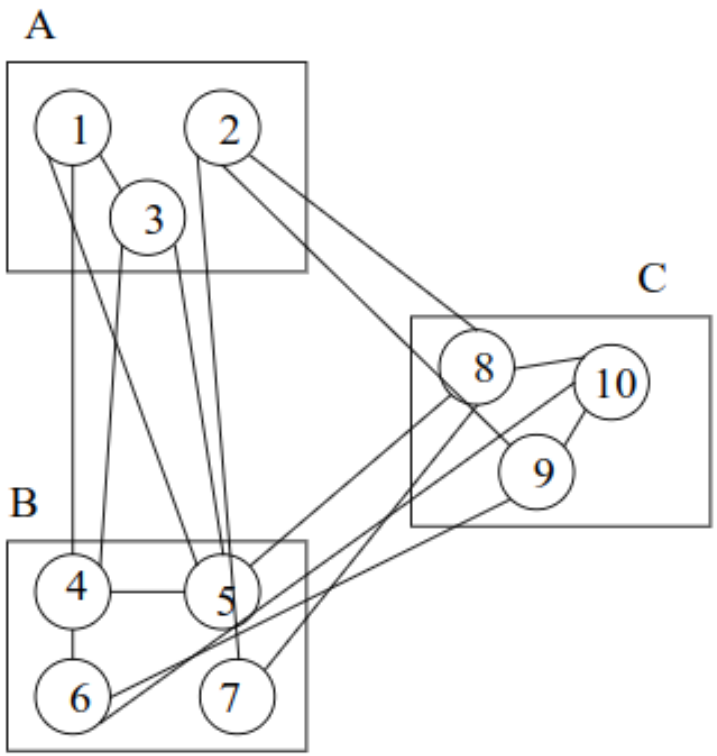
- SW calcolatrice con GUI: GUI, gestione eventi, calcoli matematici
- Disegno Mandelbrot: Complex, Mandelbrot, MandelbrotApp

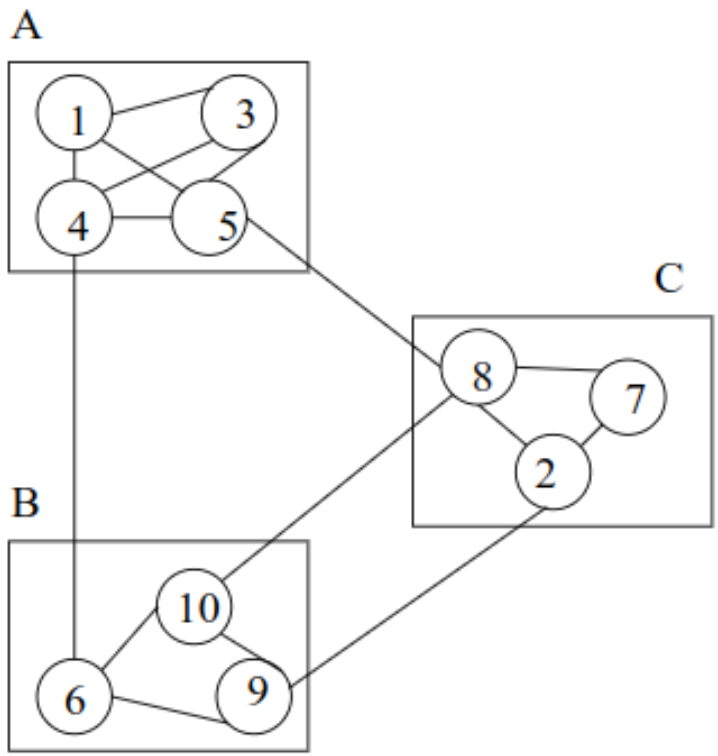
Decomposizione, modularità e dipendenze

Un punto cruciale della decomposizione è la "modularità"

- La suddivisione va fatta in modo tale che sia effettivamente conveniente
- Bisogna isolare i "sottoproblemi" più semplici
- Ciò è possibile se riduciamo al massimo le "dipendenze" fra i sottoproblemi, il che consente:
  - più autonomia decisionale
  - meno interazione con altri
  - meno influenze negative nel caso di modifiche

Modularità: quale situazione la preferibile?





Decomposizione e programmazione OO

Nella programmazione OO, almeno 3 livelli di decomposizione

1. Suddivisione in package (dell'intero programma)
2. Suddivisione in classi (di un package o programma)
3. Suddivisione in metodi (di una classe)

La OOP affronta principalmente la suddivisone in classi

- È necessario suddividire il codice in classi nel modo opportuno
- Creando il miglior link col "problem space"
- Diminuendo il più possibile le dipendenze fra classi

Tecnica

- Esistono consolidate pratiche di programmazione efficace che risolvono il problema, e che cominceremo ad analizzare in questa lezione – farlo bene richiede significativa esperienza!

Dipendenze e OO

Dipendenza

Si dice che una classe `A` dipende da una classe `B` se all'interno del codice di `A` si menziona la classe `B` (ad esempio come input di un metodo) o qualche suo mmebro. La dipendenza è tanto più profonda quanto più in `A` si usano anche costruttori e/o campi e/o metodi definiti in `B`.

Implicazione

Ogni dipendenza vincola fortemente la possibilità di fare modifiche, perché ne comporta altre da fare in cascata. Se `A` dipende da `B` e modifico `B`, dovrò probabilmente modificare anche `A`.

La sindrome dell'"intoccabilità"—SW rigido

Costruendo software complesso con troppe dipendenze, si giunge al punto che ogni singola modifica ne richiederebbe molte altre, e rischierebbe quindi di essere troppo costosa – risultato: non si cambia più il software!

Incapsulamento

Due ingredienti cruciali della programmazione OO

1. Impacchettamento dati + funzioni per manipolarli
2. Information hiding via controllo d'accesso

Filosofia

- Ogni classe dichiari `public` solo quei (pochi) metodi/costruttori necessari a interagire con (o creare) le sue istanze
- Il resto (che quindi include meri aspetti realizzativi) sia `private`
  - metodi/costruttori a solo uso interno
  - **tutti** i campi (ossia lo stato interno)

Incapsulamento e dipendenze

Così facendo il "cliente" è debolmente influenzato da possibili modifiche future riguardanti aspetti realizzativi e non di "design".

Esempio base: Classe Counter

```
1  class Counter {
2
3    /* Il campo è reso inaccessibile */
4    private int countValue;
5
6    /* E' il costruttore che inizializza i campi! */
7    public Counter() {
8      this.countValue = 0;
9    }
10
11   /* Unico modo per osservare lo stato */
12   public int getValue() {
13     return this.countValue;
14   }
15
16   /* Unico modo per modificare lo stato */
17   public void increment() {
18     this.countValue++;
19   }
20 }
```

Semplice uso

```
1  public class UseCounter {
2
3    public static void main(String[] args) {
4      final Counter c = new Counter();
5      System.out.println(c.getValue()); // 0
6      c.increment();
7      c.increment();
8      c.increment();
9      c.increment();
10     System.out.println(c.getValue()); // 4
11   }
12 }
```

Uso contatore

La classe contatore

- Incapsula una semplice funzionalità di conteggio
- Dà un approccio più astratto rispetto all'uso diretto di un `int`
- Consende di agire sul conteggio solo con `getValue()` e `increment()`

$\Rightarrow$ è impossibile modificare il conteggio a piacimento (o per errore), ad esempio decrementando invece che incrementando, o azzerando

$\Rightarrow$ nota, il codice qui sotto è più astratto, ossia più di alto livello: non sono presenti operazioni matematiche esplicite!

```
1  /* Conto su un array, con Contatore creato internamente */
2  static int countElements(int[] array, int elem) {
3      final Counter c = new Counter();
4      for (final int i : array) { // schema "for-each"
5          if (i == elem) {
6              c.increment();
7          }
8      }
9      return c.getValue();
10 }
```

Uso contatore, pt 2

Altra possibilità

- Passo il contatore alle funzioni che hanno bisogno di conteggi
- Ciò consente un più ampio grado di riuso
- In generale, sisono aperte nuove possibilità di riuso

```
1   /* Conto su un array, con Contatore passato in input */
2   static void countInArray(int[] array, int elem, Counter c) {
3       for (final int i : array) {
4           if (i == elem) {
5               c.increment();
6           }
7       }
8   }
9
10  /* Conto su una matrice, riusando appieno la countInArray */
11  static void countInMatrix(int[][] matrix, int elem, Counter c) {
12      for (final int[] array : matrix) {
13          countInArray(array, elem, c);
14      }
15  }
```

Riflessione: incapsulazione e contratto

Contratto

- Il contratto di un oggetto corrisponde ai suoi scenari di utilizzo
- E quindi alle aspettative che un cliente ha quando usa l'oggetto
- Grazie all'incapsulamento, è possibile vincolare fortemente questi contratti, controllando meglio il comportamento degli oggetti

Il caso del Contatore

- Il valore del conteggio all'atto della costruzione è `0`
- Il valore del conteggio in ogni altro istante è pari al numero di chiamate di `increment()`

Osservazione

È grazie a questa idea che è più facile comporre oggetti in sistemi più complicati (vedi funzione `countInMatrix`)

Oggetti immutabili

Cosa sono

- Oggetti per i quali è garantito che il valore iniziale dei campi non si modificherà mai
- Portano un ulteriore livello di indipendenza (e semplicità/eleganza) nel codice
- In alcuni casi potrebbero portare a soluzioni poco performanti

Come si costruiscono

- I campi della classe sono dichiarati `final` (oltre che `private`), e..
- ..contengono a loro volta valori primitivi o oggetti immutabili
- Quindi nessun metodo può modificarli (solo i costruttori possono, la prima volta)
- Invece che cambiare i campi si possono solo creare oggetti con nuovo stato

Osservazione

Per ora è sufficiente saperli riconoscere e costruire

Favorire sempre immutabilità ove possibile – un principio avanzato

Indicare anche `final` variabili e argomenti che non verranno modificati

ImmutablePoint3D

```java
public class ImmutablePoint3D {

    private final double x; // x coordinate
    private final double y; // y coordinate
    private final double z; // z coordinate

    public ImmutablePoint3D(final double x, final double y, final double z) {
        this.x = x;
        this.y = y;
        this.z = z;
    }

    public double getSquareModule() {
        return this.x * this.x + this.y * this.y + this.z * this.z;
    }

    public double getX() {
        return this.x;
    }

    public double getY() {
        return this.y;
    }

    public double getZ() {
        return this.z;
    }

    /* A method that changes state is mimicked by creating a new object */
    public ImmutablePoint3D translate(final double x, final double y, final double z) {
        return new ImmutablePoint3D(this.x + x, this.y + y, this.z + z);
    }
}
```

UseImmutablePoint3D

```java
public class UseImmutablePoint3D {

    public static void main(String[] args) {
        final ImmutablePoint3D p = new ImmutablePoint3D(10, 20, 30);
        // l'oggetto puntato da p non potrà essere modificato

        ImmutablePoint3D q = p.translate(1, 1, 1);
        // q punta ad un nuovo oggetto

        System.out.println(p.getX() + " " + p.getY() + " " + p.getZ());
        // 10,20,30
        System.out.println(q.getX() + " " + q.getY() + " " + q.getZ());
        // 11,21,31

        q = q.translate(1, 1, 1);
        // la variabile q punta ad un nuovo oggetto
        // il vecchio verrà eleminato dal GC

        System.out.println(q.getX() + " " + q.getY() + " " + q.getZ());
        // 12,22,32
    }
}
```

Outline

Altro esempio: classe `Lamp`

Analisi del problema

In un sistema domotico, dovremo gestire un certo numero di lampadine (da accendere/spegnere e pilotare tramite un apposito controllore centralizzato, oltre che tramite i comandi a muro). Tali comandi sono a pulsante, dotato anche di controllo di intensità "dimmer" (10 livelli). Il controllore deve poter accedere alla situazione di ogni lampadina (accesa/spenta, livello di intensità) e modificarla a piacimento. Al primo avvio, le lampadine sono spente e il controllo di intensità è a zero (in un intervallo $[0, 1]$).

Come procediamo alla costruzione della classe `Lamp`?

Progettazione e implementazione: fasi

Fasi nella costruzione di una classe

1. Progettazione della parte pubblica della classe
2. Costruzione dello stato
3. Completamento implementazione
4. Miglioramento codice finale
5. Test del risultato

(negli approcci moderni si testa anche tra la fase 3 e 4)

Fase 1: Progettazione della parte pubblica della classe

Ovvero, del nome della classe e delle signature di operazioni pubbliche (metodi e costruttori)

Linee guida

- Considerare tutti i vari casi d'uso di un oggetto della classe
- Inserire costruttori e metodi pubblici solo per le operazioni necessarie
- Evitare ove possibile di inserire un numero elevato di tali operazioni

Il caso `Lamp`

- Un costruttore unico senza argomenti
- Metodi per accendere/spegnere
- Metodi per aumentare/ridurre/impostare intensità
- Metodi per accedere allo stato della lampadina

Parte pubblica della classe Lamp

```
1  /* Classe d'esempio che modella il concetto di Lampadina
2     in un sistema Domotico */
3  public class Lamp{
4
5      /* Inizializzazione */
6      public Lamp(){ .. }
7
8      /* Accendo/Spengo */
9      public void switchOn(){ .. }
10     public void switchOff(){ .. }
11
12     /* Meno intenso/Più intenso/Quanto intenso */
13     public void dim(){ .. }
14     public void brighten(){ .. }
15     public void setIntensity(double value){ .. }
16
17     /* Selezione */
18     public double getIntensity(){ .. }
19     public boolean isSwitchedOn(){ .. }
20 }
```

Fase 2: Costruzione dello stato

Ovvero, dei campi privati della classe

Linee guida

- Considerare che esistono varie scelte possibili (è un aspetto implementativo, ritrattabile successivamente)
- L'insieme dei campi deve essere più piccolo possibile, per esigenze di performance (spazio in memoria) e di non duplicazione
- L'insieme dei campi deve essere sufficiente a tenere traccia di tutti i modi in cui il comportamento dell'oggetto può cambiare a fronte dei messaggi ricevuti

Il caso `Lamp`

- Dovremo sapere se è accesa o spenta (`boolean switchedOn`)
- Dovremo sapere il livello attuale di intensità (`double intensity`)
- Non sembrano servire altre informazioni

Stato e metodi della classe Lamp

```
1  /* Classe d'esempio che modella il concetto di Lampadina
2     in un sistema Domotico */
3  public class Lamp{
4
5      /* Campi */
6      private boolean switchedOn;
7      private double intensity;
8
9      /* Costruttore */
10     public Lamp(){ .. }
11
12     /* Metodi */
13     public void switchOn(){ .. }
14     public void switchOff(){ .. }
15     public void dim(){ .. }
16     public void brighten(){ .. }
17     public void setIntensity(double value){ .. }
18     public double getIntensity(){ .. }
19     public boolean isSwitchedOn(){ .. }
20 }
```

Fase 3: Completamento implementazione

Ovvero, del corpo di costruttori e metodi

Linee guida

- Realizzare il corpo di ogni costruttore e metodo in modo compatibile col contratto previsto per la classe
- Accettare il fatto che la prima versione prodotta non necessariamente sarà quella finale

Il caso `Lamp`

- `switchOn()`, `switchOff()` sono semplici setter del campo `switchedOn`
- `isSwitchedOn()`, `getIntensity()` semplici getter dei due campi
- `dim()` e `brigthen()` modificano il campo `intensity` (se nel range!)

Prima versione Classe Lamp

```java
public class Lamp{
    private double intensity;
    private boolean switchedOn;

    public Lamp(){
      this.switchedOn = false;
      this.intensity = 0;
    }
    public void switchOn(){
      this.switchedOn = true;
    }
    public void switchOff(){
      this.switchedOn = false;
    }
    public void dim(){
      this.intensity = (this.intensity < 0.1 ? 0 : this.intensity-0.1);
    }
    public void brighten(){
      this.intensity = (this.intensity > 0.9  ? 1 : this.intensity+0.1);
    }
    public void setIntensity(double value){
      this.intensity = value;
      if (value < 0) { this.intensity = 0; } // Mal formattato!
      if (value > 1) { this.intensity = 1; } // Mal formattato!
    }
    public double getIntensity(){
      return this.intensity;
    }
    public boolean isSwitchedOn(){
      return this.switchedOn;
    }
}
```

Fase 4: Miglioramento codice finale

Linee guida

- Inserire commenti nel codice
- Verificare la necessità di costanti per evitare numeri "magici"
- Eventualmente fattorizzare sotto-funzioni in metodi/costruttori pubblici/privati, per evitare duplicazioni

In concreto in questo caso

- Vi sono numeri magici, usare costanti!
- Gestire meglio il limite 0..1
- Evitare livelli intermedi (0.145) di luminosità
- Ritrattare la scelta del tipo del campo `intensity` – meglio un `int` fra 0 e 10!!
- Grazie all'incapsulamento possiamo cambiare i campi senza modificare i client!

Versione finale Classe Lamp – A

```java
public class Lamp {

    /* Costanti luminosità */
    private static final int LEVELS = 10;
    private static final double DELTA = 0.1;

    /* Campi della classe */
    private int intensity;
    private boolean switchedOn;

    /* Costruttore */
    public Lamp() {
        this.switchedOn = false;
        this.intensity = 0;
    }

    /* Gestione switching */
    public void switchOn() {
        this.switchedOn = true;
    }

    public void switchOff() {
        this.switchedOn = false;
    }

    public boolean isSwitchedOn() {
        return this.switchedOn;
    }
```

Versione finale Classe Lamp – B

```
1    /* Gestione intensita' */
2    private void correctIntensity() { // A solo uso interno
3        if (this.intensity < 0) {
4            this.intensity = 0;
5        } else if (this.intensity > LEVELS) {
6            this.intensity = LEVELS;
7        }
8    }
9
10   public void setIntensity(final double value) {
11       this.intensity = Math.round((float) (value / DELTA));
12       this.correctIntensity();
13   }
14
15   public void dim() {
16       this.intensity--;
17       this.correctIntensity();
18   }
19
20   public void brighten() {
21       this.intensity++;
22       this.correctIntensity();
23   }
24
25   public double getIntensity() {
26       return this.intensity * DELTA;
27   }
28
```

Fase 5: Test del risultato

Linee guida

- Definire un insieme di scenari d'uso di un oggetto
- Per ognuno costruire una procedura che crea l'oggetto, lo usa, e stampa i risultati necessari

Il caso `Lamp`

Un possibile caso (non costituisce da solo un test esaustivo):

- Costruisco l'oggetto lampadina
- La accendo
- Imposto la luminosità, poi la vario un poco
- Leggo e stampo lo stato del sistema

Classe UseLamp

```
1  public class UseLamp {
2      private static void test1() {
3          final Lamp l = new Lamp();
4          System.out.println(l);
5          l.switchOn();
6          l.setIntensity(0.5);
7          l.dim();
8          l.dim();
9          System.out.println(l);
10         l.brighten();
11         System.out.println(l);
12         // Acceso: true Intensità: 0.4
13     }
14
15     public static void main(final String[] s) {
16         UseLamp.test1();
17         // altri test...
18     }
19 }
```

Il metodo toString()

Una convenzione Java

- Ogni classe dovrebbe definire un metodo `toString()`
- Questo deve restituire una rappresentazione in stringa dell'oggetto
- Così si incapsula anche la funzionalità di presentazione (su console)
- Tale metodo è quello che viene automaticamente chiamato quando si usa l'operatore + per concatenare stringhe a oggetti

UseLamp: uso di toString

```
1  public class Lamp{
2      ...
3      public String toString(){
4        return "Acceso: "+this.isSwitchedOn()+
5            ", Intensità: "+this.getIntensity();
6      }
7  }
8  public class UseLampString{
9      public static void test1(){
10       Lamp l=new Lamp();
11       l.switchOn();
12       l.setIntensity(0.5);
13       l.dim();
14       l.dim();
15       l.brighten();
16       System.out.println(l.toString());  // toString esplicito
17       System.out.println("Oppure : "+l); // toString implicito
18     }
19     public static void main(String[] s){
20       UseLamp.test1();
21       // altri test...
22     }
23 }
```

Il collaudo di Lamp è completato?

Quanti scenari di test vanno preparati?

- Non c'è un numero giusto
- Non è possibile in generale controllare in modo completo
- Bisogna trovare il giusto rapporto tempo/risultato
- Certamente, un unico scenario è insufficiente
- La metodologia "test-driven development" consiglia di costruire test esaustivi prima dell'effettivo sviluppo di ogni funzionalità
  - Inizialmente percepito come un po' noioso
  - Sembra far perdere tempo
  - Spesso ripaga in termini di temi complessivi e qualità del software profotto
  - Vedremo framework dedicati ai test (`JUnit`)

Outline

Convenzione Java sui nomi di metodi getter/setter

Metodi getter/setter

- Un metodo getter è un metodo che senza input restituisce un valore, una proprietà dell'oggetto
- Un metodo setter è un metodo che restituisce `void` e accetta un valore che modifica una proprietà dell'oggetto
- (Tali proprietà sono spesso campi, ma ciò non è necessario)
- In `Lamp`, `getIntensity` e `isSwitchedOn` sono getter, `setIntensity` è un setter

Convenzione sul nome del metodo: sia data la proprietà `XYZ` di tipo `T`

- Getter non booleano: `T getXYZ(){...}`
- Getter booleano: `boolean isXYZ(){...}`
- Setter: `void setXYZ(T xYZ){...}`

Getter/setter in Lamp

```
1  public class Lamp{
2      ...
3
4      // Setter per proprieta' Intensity di tipo double
5      public void setIntensity(double value){
6        ...
7      }
8
9      // Getter per proprieta' Intensity di tipo double
10     public double getIntensity(){
11       ...
12     }
13
14     // Getter per proprieta' SwitchedOn di tipo boolean
15     public boolean isSwitchedOn(){
16       ...
17     }
18 }
```