06
Composizione, riuso e sostituibilità: le `interface`

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 concetti generali di composizione e riuso
- Introdurre il concetto di interfaccia
- Discutere il principio di sostituibilità
- Evidenziare il polimorfismo derivante dalle interfacce

Argomenti

- Tipi di composizione e loro realizzazione
- Notazione UML
- `interface` in Java e meccanismi collegati
- Polimorfismo con le interfacce

Outline

Intro

L'incapsulamento ci fornisce i meccanismi per ben progettare le classi, limitando il più possibile le dipendenze con chi le usa, e quindi in modo da ridurre l'impatto delle modifiche che si rendono via via necessarie.

$\Rightarrow$ ma le dipendenze fra classi non sono evitabili del tutto, anzi, sono un prerequisito per fare di un gruppo di classi un sistema! In più, le dipendenze sono anche manifestazione di un effettivo "riuso".

Forme di dipendenza e riuso fra classi nell'OO

- Associazione — Un oggetto ne usa un altro: "uses"
- Composizione — Un oggetto ne aggrega altri: "has-a"
- Specializzazione — Una classe ne specializza un'altra: "is-a"

Nella lezione corrente

Introdurremo la composizione (che è una versione più forte della associazione), mostrando la sua relazione con le interface di Java

Composizione – relazione "has-a"

Idea

- un oggetto della classe `A` è ottenuto componendo (o "si compone di") un insieme di altri oggetti, delle classi `B1`, `B2`, ..., `Bn`
- ossia, lo stato dell'oggetto di `A` include (il riferimento) a un oggetto di `B1`, uno di `B2`, ..., uno di `Bn`
- si noti che si parla propriamente di composizione quando `B1`, `B2`, ..., `Bn` non sono tipi primitivi, ma classi

Composizione vs aggregazione

- in fase di modellazione (definizione del "problem space"), quando gli oggetti composti hanno vita propria senza l'esistenza di `A` si parla anche di aggregazione
- nella programmazione, e in Java, molto spesso non si fa questa distinizione—la differenza sta perloppiù nel fatto che, con l'aggregazione, l'oggetto che aggrega rende disponibile all'esterno il riferimento all'oggetto aggregato

Qualche esempio di composizione

GUI

Un oggetto interfaccia grafica si compone di oggetti di tipo `Button`, `TextField`, `Label`, eccetera

Ateneo

Un oggetto ateneo si compone di oggetti di tipo `Faculty`, `Student`, `Teacher`, eccetera

Controllore Domotica

Un oggetto controllore domotica si compone di oggetti di tipo `Lamp`, `TV`, `Radio`, eccetera

Tipiche realizzazioni

Un oggetto `A` si compone esattamente di un oggetto di `B`

- La classe `A` avrà un campo (privato) di tipo `B`
- Tale campo (impostato dal costruttore di `A`) è sempre presente

Un oggetto `A` si compone opzionalmente di un oggetto di `B`

- La classe `A` avrà un campo (privato) di tipo `B`
- Il suo contenuto potrebbe essere `null` (oggetto di `B` assente)

Un oggetto `A` si compone di un numero noto *n* di oggetti di `B`

- La classe `A` avrà *n* campi (privati) di tipo `B` – se "n" piccolo

Un oggetto `A` si compone di una moltitudine non nota di oggetti di `B`

- La classe `A` avrà un campo (privato) di tipo `B[]` (o altro container)

Ricordiamo la classe Lamp – A

```java
public class Lamp {

    /* Costanti luminosità */
    private final static int LEVELS = 10;
    private final static 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;
    }
```

Ricordiamo la 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
```

Un esercizio: dispositivo `TwoLampsDevice`

Caratteristiche del sistema da modellare

- una base su cui poggiano due lampadine
- possibilità di accendere/spegnere entrambe
- possibilità di modalità "eco" (una sola accesa, a metà)

Idea realizzativa 1 :(

- una classe con 4 campi, ossia le due intensità e i due flag
- sarebbe un buon design?
- riuserei codice? starei aderendo al principio DRY? (Don't Repeat Yourself)

Idea realizzativa 2 :)

- riuso `Lamp` e sfrutto la composizione

Esempio: TwoLampsDevice pt 1

```java
public class TwoLampsDevice {

  /* Composizione immutabile di due Lamp! */
  private final Lamp l1;
  private final Lamp l2;

  /* Composizione inizializzata al momento della costruzione */
  public TwoLampsDevice() {
    this.l1 = new Lamp();
    this.l2 = new Lamp();
  }

  /* Metodi getter */
  public Lamp getFirst() {
    return this.l1;
  }

  public Lamp getSecond() {
    return this.l2;
  }
```

Esempio: TwoLampsDevice pt 2

```
1   /* Altri metodi */
2   public void switchOnBoth() {
3     this.l1.switchOn();
4     this.l2.switchOn();
5   }
6
7   public void switchOffBoth() {
8     this.l1.switchOff();
9     this.l2.switchOff();
10  }
11
12  public void ecoMode() {
13    this.l1.switchOff();
14    this.l2.switchOn();
15    this.l2.setIntensity(0.5);
16  }
17
18  public String toString() {
19    return "TwoLampsDevice [l1=" + l1 + ", l2=" + l2 + "]";
20  }
21 }
```

La necessità di una notazione grafica – UML

UML – Unified Modelling Language

- È un linguaggio grafico e OO-based per modellare software
- Facilita lo scambio di documentazione, e il ragionamento su sistemi articolati e complessi
- È uno standard dell'OMG dal 1996
- È molto utile anche a fini didattici
- Noi ne useremo solo la parte chiamata Class Diagram
- Nel corso di Ingegneria del Software lo approfondirete

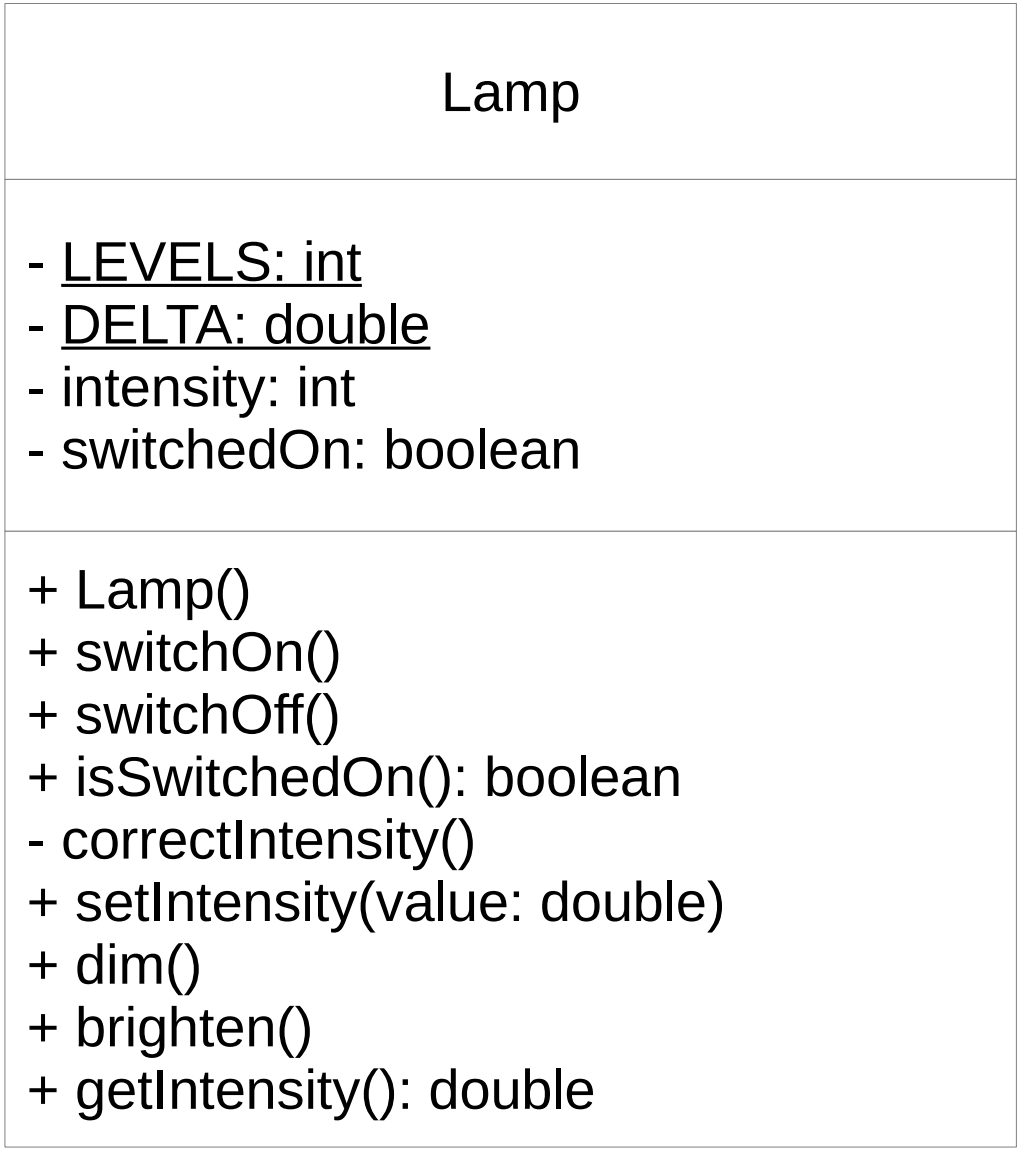
Class Diagram

...diagramma delle classi, una prima descrizione

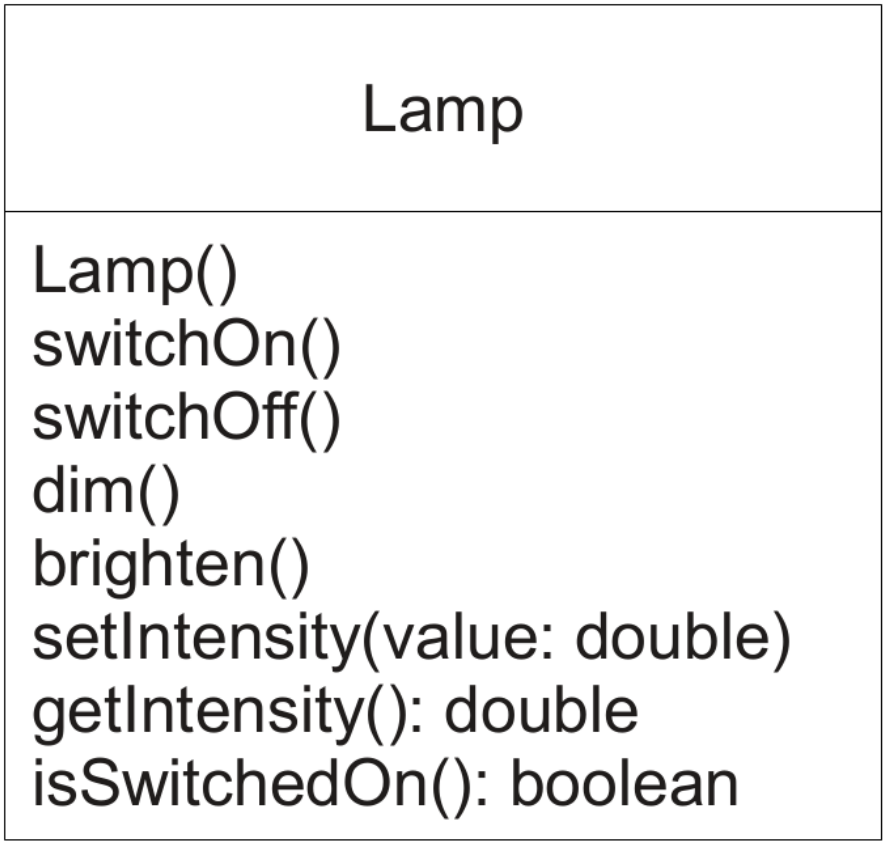
- Un box rettangolare per classe, diviso in tre aree:
  - 1. nome della classe, 2. campi, 3. metodi (e costruttori)
- Su campi e metodi
  - si antepone il simbolo - se privati, + se pubblici
  - si sottolineano se `static`
  - dei metodi si riporta solo la signature, con sintassi:
    `nome(arg1: tipo1, arg2: tipo2, ..): tipo_ritorno`
- archi fra classi indicano relazioni speciali:
  - con rombo (composizione), con freccia (semplice associazione)
  - con triangolo (generalizzazione)
  - l'arco può essere etichettato con la molteplicità (1, 2, 0..1, 0..n, 1..n)

A seconda dello scopo per cui si usa il diagramma, non è necessario riportare tutte le informazioni, ad esempio spesso si omettono le proprietà, le signature complete, ed alcune relazioni

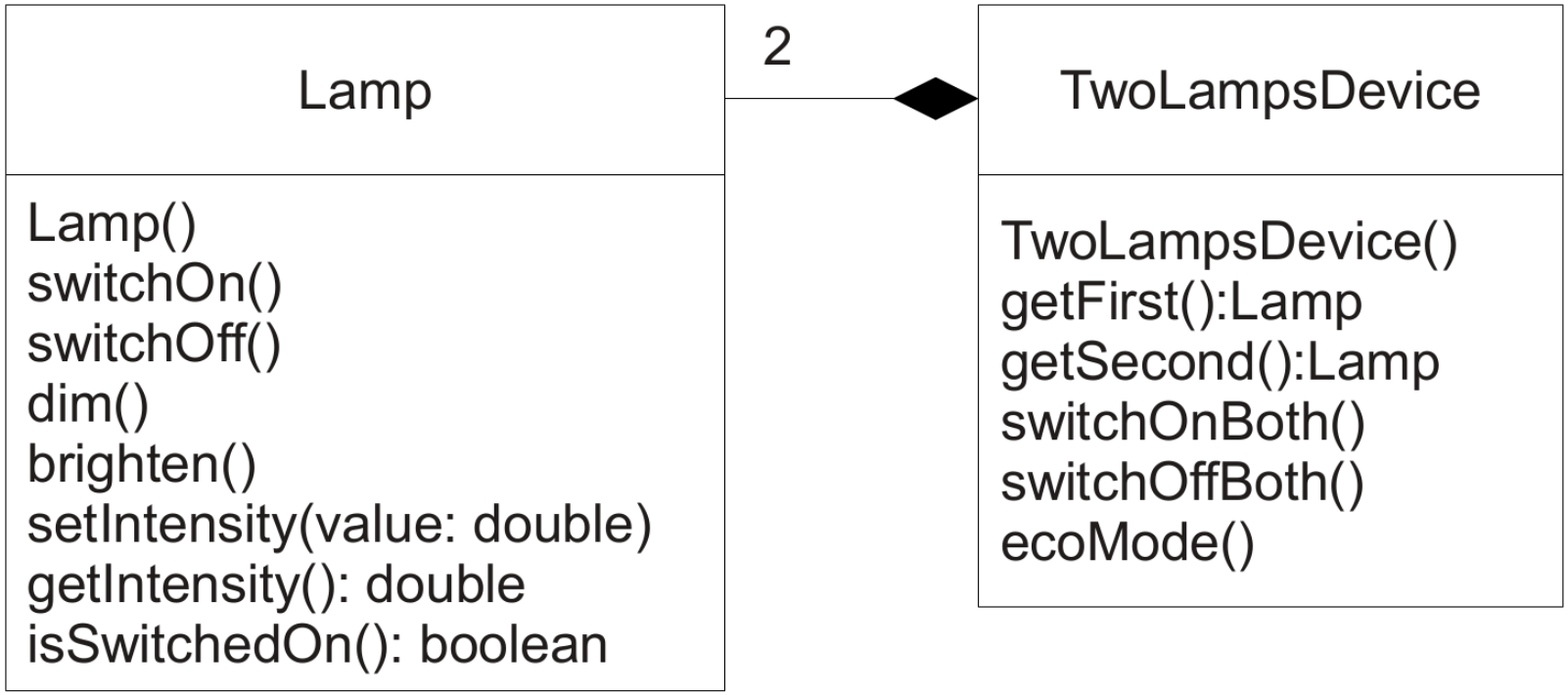
Notazione UML completa per la classe `Lamp`: tipicamente usata in fase di implementazione



Notazione parziale: solo parte pubblica: tipicamente usata in fase di design



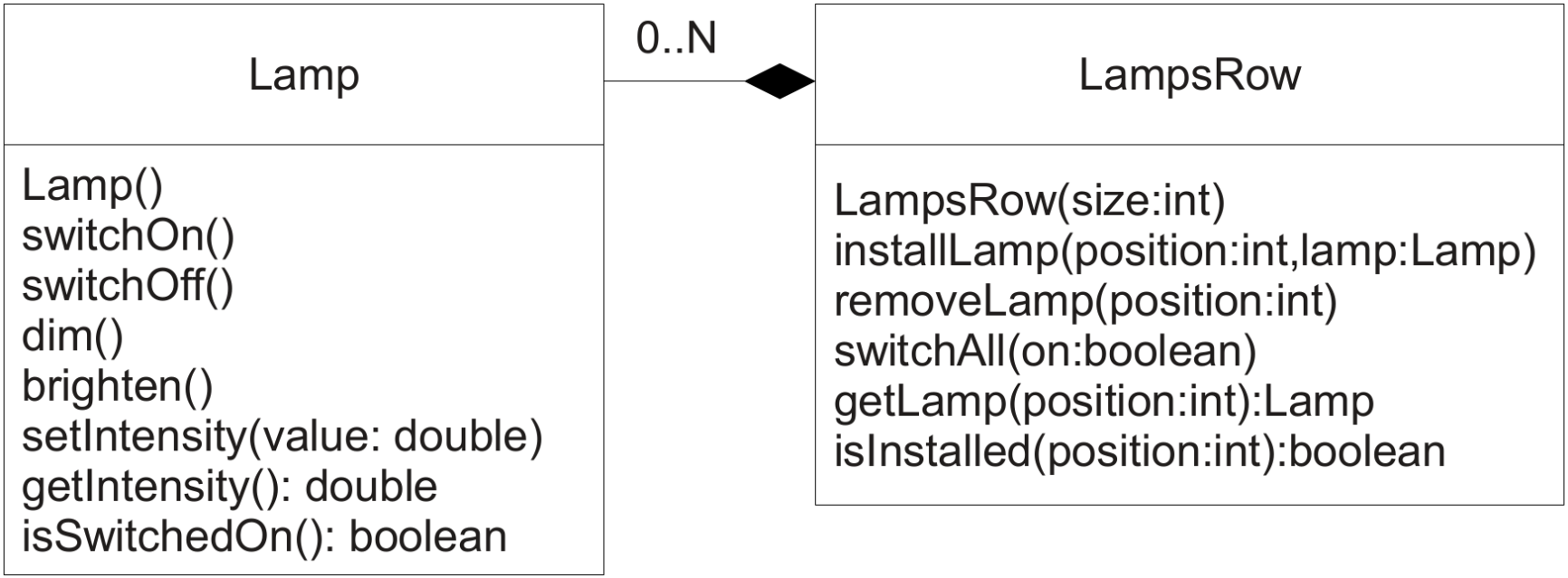
UML: Lamp e TwoLampsDevice



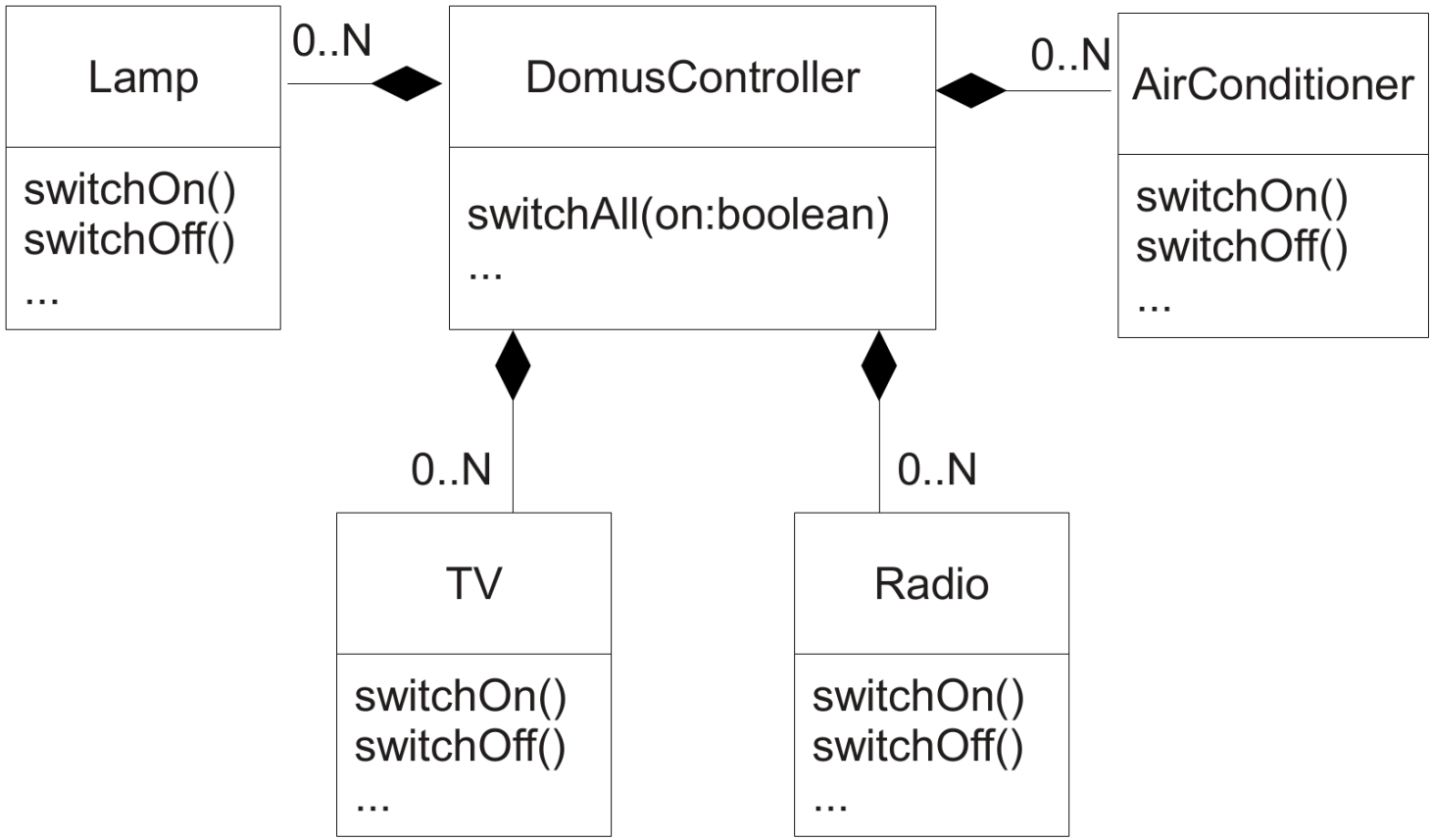
Altro caso di composizione: LampsRow

```
1  public class LampsRow{
2
3      private final Lamp[] row; // Campo
4
5      public LampsRow(final int size){
6        this.row = new Lamp[size]; // Tutti i valori a null
7      }
8      public void installLamp(final int position, final Lamp lamp){
9        this.row[position] = lamp;
10     }
11     public void removeLamp(final int position){
12       this.row[position] = null;
13     }
14     public void switchAll(final boolean on){
15       for (Lamp lamp: this.row){
16           if (lamp != null){ // Previene il NullPointerException
17             if (on){
18                 lamp.switchOn();
19             } else {
20                 lamp.switchOff();
21             }
22           }
23       }
24     }
25     public Lamp getLamp(final int position){ // Selettore
26       return this.row[position];
27     }
28     public boolean isInstalled(final int position){ // Selettore
29       return this.row[position] != null;
30    }
31 }
```

UML: Lamp e LampsRow



Scenario DomusController



Come scrivereste il metodo `switchAll` in modo riusabile, e possibilmente rimandendo aperti all'introduzione di nuovi tipi di dispositivi? Come garantire che i metodi delle varie classi conformino ad una stessa intestazione?

Realizzazione senza riuso: schema

```
1  public class DomusController{
2
3      private Lamp[] lamps;
4      private TV[] tvs;
5      private AirConditioner[] airs;
6      private Radio[] radios;
7
8      ...
9      public void switchAll(boolean on){
10       for (Lamp lamp: this.lamps){
11           if (lamp != null){
12             if (on){
13                 lamp.switchOn();
14             } else {
15                 lamp.switchOff();
16             }
17           }
18       }
19       for (TV tv: this.tvs){
20           if (tv != null){
21             if (on){
22                 tv.switchOn();
23             } else {
24                 tv.switchOff();
25             }
26           }
27       }
28       ... // e così via per tutti i dispositivi
29     }
30
31 }
```

Outline

Motivazioni

Specifica

- Serve un meccanismo per separare esplicitamente, ossia in dichiarazioni diverse, l'interfaccia della classe e la sua realizzazione
- Questo consente di tenere separate fisicamente la parte di "contratto" (tipicamente fissa) con quella di "implementazione" (modificabile frequentemente)
- Consente di separare bene "abstraction" e "concretion"

Polimorfismo

- Serve un meccanismo per poter fornire diverse possibili realizzazioni di un contratto (o sue versioni implementative successive)
- Tutte devono poter essere utilizzabili in modo omogeneo
- Nel caso di `DomusController`:
  - Avere un unico contratto per i "dispositivi", e..
  - ..diverse classi che lo rispettano
  - `DomusController` gestirà un unico array di "dispositivi"

Java `interfaces`

Cos'è una `interface`

- È un nuovo tipo di dato dichiarabile (quindi come le classi)
- Ha un nome, e include "solo" un insieme di intestazioni di metodi
- Viene compilato da `javac` come una classe, e produce un `.class`

Una `interface I` può essere "implementata" da una classe

- Attraverso una classe `C` che lo dichiara esplicitamente (`class C implements I{..}`)
- `C` dovrà definire (il corpo di) tutti i metodi dichiarati in `I`
- Un oggetto istanza di `C`, avrà come tipo `C` al solito, ma anche `I`!!

Esempio: dispositivi DomusController

`Lamp`, `TV`, `Radio`, `AirConditioner` hanno una caratteristica comune, sono dispositivi e come tali possono come minimo essere accesi o spenti. È possibile definire una interfaccia `Device` che tutti e 4 implementano.

Interface Device

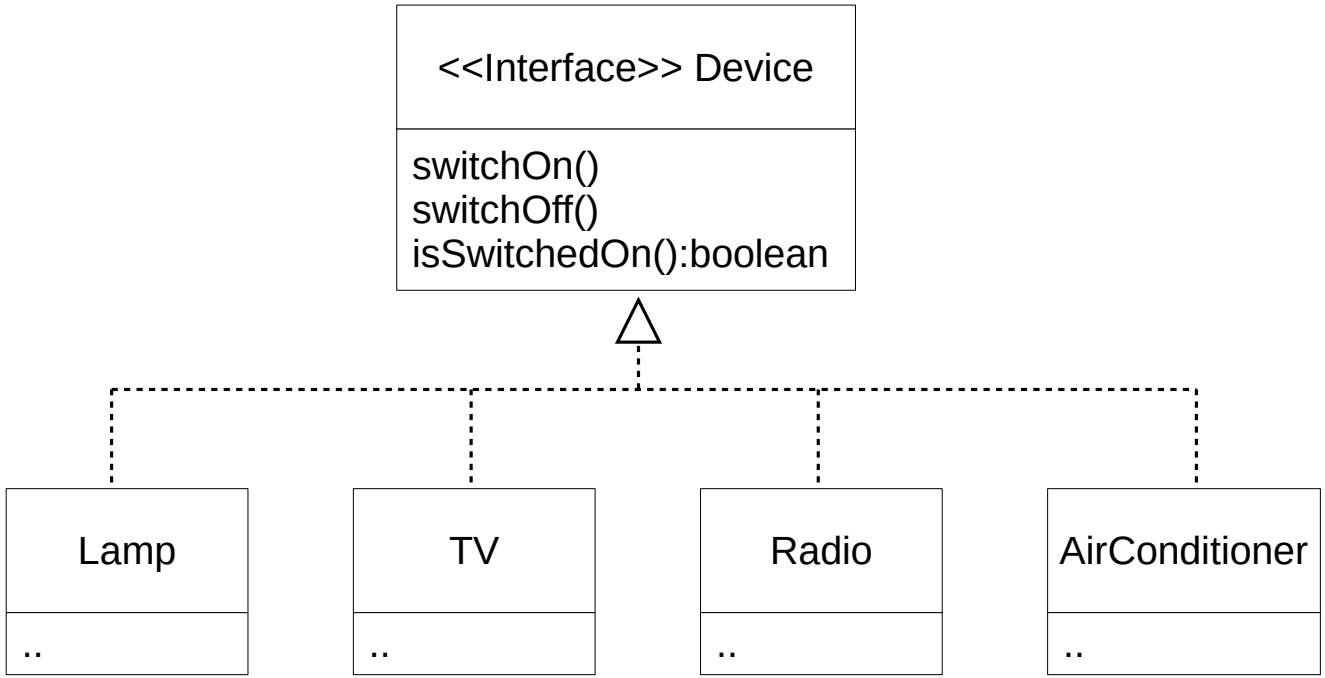
```
1  /* Interfaccia per dispositivi
2  Definisce un contratto:
3  - si può accendere
4  - si può spegnere
5  - si può verificare se acceso/spento
6
7  Nota: nessuna indicazione
8        public/private nei metodi!
9  */
10
11 public interface Device {
12
13   void switchOn();
14
15   void switchOff();
16
17   boolean isSwitchedOn();
18 }
```

Implementazioni di Device

```java
public class Lamp implements Device{
    ...  // Nessun cambiamento necessario rispetto a prima!
    private boolean switchedOn;

    public void switchOn(){
      this.switchedOn = true;
    }
    public void switchOff(){
      this.switchedOn = false;
    }
    public boolean isSwitchedOn(){
      return this.switchedOn;
    }
    ...
}

public class TV implements Device{
    ... // Nessun cambiamento necessario rispetto a prima!
}

public class Radio implements Device{
    ... // Nessun cambiamento necessario rispetto a prima!
}
```

Notazione UML per le interfacce

- interfaccia come box con titolo "<< `interface` >> Nome"
- arco tratteggiato (punta a triangolo) per la relaz. "`implements`"
- archi raggruppati per migliorare la resa grafica



Interfacce come tipi di dato

Data l'interfaccia `I`, in che senso `I` è un tipo?

- `I` è un tipo come gli altri (`int`, `float`, `String`, `Lamp`, `Lamp[]`)
- è usabile per dichiarare variabili, come tipo di input/output di una funzione, come tipo di un campo

Quali operazioni sono consentite?

esattamente (e solo) le chiamate dei metodi definiti dall'interfaccia

Quali sono i valori (/oggetti) di quel tipo?

gli oggetti delle classi che dichiarano implementare quell'interfaccia

Interfacce e assegnamenti

```
1  /* Su Lamp effettuo le usuali operazioni */
2  Lamp lamp = new Lamp();
3  lamp.switchOn();
4  boolean b = lamp.isSwitchedOn();
5
6  /* Una variabile di tipo Device può contenere
7     un Lamp, e su essa posso eseguire le
8     operazioni definite da Device */
9  Device dev;   // creo la variabile
10 dev = new Lamp(); // assegnamento ok
11 dev.switchOn();   // operazione di Device
12 boolean b2 = dev.isSwitchedOn();  // operazioni di Device
13
14 Device dev2 = new Lamp();  // Altro assegnamento
15
16 /* Attenzione, le interfacce non sono istanziabili */
17 // Device dev3 = new Device(); NO!!!!
```

Ridurre la molteplicità / aumentare riuso – prima

```
1  class DeviceUtilities{
2
3      /* Senza interfacce, non resta altro che... */
4
5      public static void switchOnIfCurrentlyOff(Lamp lamp){
6        if (!lamp.isSwitchedOn()){
7            lamp.switchOn();
8        }
9      }
10
11     public static void switchOnIfCurrentlyOff(TV tv){
12       if (!tv.isSwitchedOn()){
13           tv.switchOn();
14       }
15     }
16
17     public static void switchOnIfCurrentlyOff(Radio radio){
18       if (!radio.isSwitchedOn()){
19           radio.switchOn();
20       }
21     }
22     ..
23 }
```

Ridurre la molteplicità / aumentare riuso – dopo

```
1  class DeviceUtilities{
2
3      /* Grazie alle interfacce, fattorizzo in un solo metodo */
4
5      public static void switchOnIfCurrentlyOff(Device device){
6        if (!device.isSwitchedOn()){
7            device.switchOn();
8        }
9      }
10 }
11
12 /* Codice cliente */
13 Lamp lamp = new Lamp();
14 TV tv = new TV();
15 Radio radio = new Radio();
16
17 switchOnIfCurrentlyOff(lamp); // OK, un Lamp è un Device
18 switchOnIfCurrentlyOff(tv); // OK, una TV è un Device
19 switchOnIfCurrentlyOff(radio); // OK, una Radio è un Device
```

Razionale delle interfacce

Quando costruire una interfaccia?

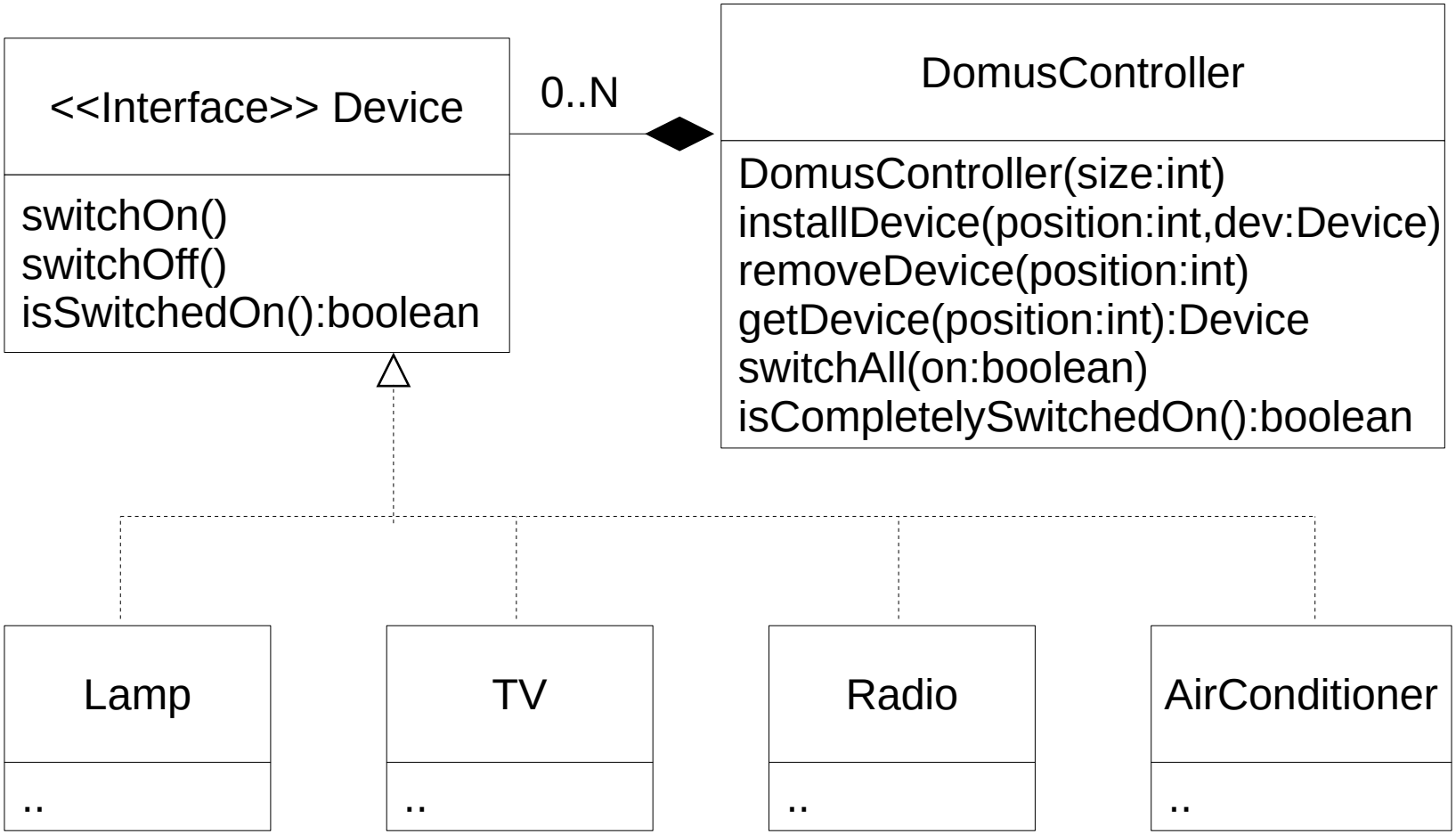
- quando si ritiene utile separare contratto da implementazione (sempre vero per i concetti cardine in applicazioni complesse)
- quando si prevede la possibilità che varie classi possano voler implementare un medesimo contratto
- quando si prevede che una classe possa essere evoluta in una nuova versione, come ulteriore implementazione
- quando si vogliono costruire funzionalità che possano lavorare con qualunque oggetto che implementi il contratto
  - caso particolare: Quando si vuole comporre ("has-a") un qualunque oggetto che implementi il contratto

$\Rightarrow$ l'esperienza mostra che classi riusabili di norma hanno sempre una loro `interface`

Quindi:

- laddove ci si aspetta un oggetto che implementi il contratto si usa il tipo dell'interfaccia
- questo consente il riuso della funzionalità a tutte le classi che implementano il contratto

Scenario DomusController rivisitato



Codice DomusController pt 1

```java
public class DomusController {

  /* Compongo n oggetti che implementano Device */
  private final Device[] devices;

  public DomusController(final int size) {
    this.devices = new Device[size];
  }

  /* Aggiungo un device */
  public void installDevice(final int position, final Device dev) {
    this.devices[position] = dev;
  }

  /* Rimuovo un device */
  public void removeDevice(final int position) {
    this.devices[position] = null;
  }

  public Device getDevice(final int position) {
    return this.devices[position];
  }
```

Codice DomusController pt 2

```
1   /* Spengo/accendo tutti i device */
2   public void switchAll(final boolean on) {
3     for (Device dev : this.devices) {
4       if (dev != null) { // Prevengo il NullPointerException
5         if (on) {
6           dev.switchOn();
7         } else {
8           dev.switchOff();
9         }
10      }
11    }
12  }
13
14  /* Verifico se sono tutti accesi */
15  public boolean isCompletelySwitchedOn() {
16    for (Device dev : this.devices) {
17      if (dev != null && !dev.isSwitchedOn()) {
18        return false;
19      }
20    }
21    return true;
22  }
23 }
```

Codice TV

```java
public class TV implements Device {

  /* Campi della classe */
  private boolean switchedOn;

  /* Costruttore */
  public TV() {
    this.switchedOn = false;
  }

  /* Metodi per accendere e spegnere */
  public void switchOn() {
    this.switchedOn = true;
  }

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

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

  public String toString() {
    return "I'm a TV..";
  }
}
```

Uso di DomusController

```
1  public class UseDomusController{
2
3      public static void main(String[] s){
4        // Creo un DomusController
5        final DomusController dc = new DomusController(10);
6
7        // Installo dei dispositivi
8        dc.installDevice(0,new Lamp());
9        dc.installDevice(1,new Lamp());
10       dc.installDevice(2,new Lamp());
11       dc.installDevice(3,new TV());
12       dc.installDevice(4,new TV());
13       dc.installDevice(5,new Radio());
14
15       // Li accendo tutti
16       dc.switchAll(true);
17
18       // Verifico l'accensione
19       final boolean b = dc.isCompletelySwitchedOn(); // true
20     }
21 }
```

Outline

`implements` come relazione di "sottotipo"

Un tipo è considerabile come un set di valori/oggetti

- $T_{\texttt{boolean}} = \{\texttt{true}, \texttt{false}\}$
- $T_{\texttt{int}} = \{-2147483648, \ldots, -1, 0, 1, 2, \ldots, 2147483647\}$
- $T_{\texttt{Lamp}} = \{X | X \text{ is an object of class } \texttt{Lamp}\}$
- $T_{\texttt{Device}} = \{X | X \text{ is an object of a class implementing } \texttt{Device}\}$
- $T_{\texttt{String}} = \{X | X \text{ is an object of class } \texttt{String}\}$

`Lamp` è un sottotipo di `Device`!

- Un oggetto della classe `Lamp` è anche del tipo `Device`
- Quindi, da $X \in T_{\texttt{Lamp}}$ segue $X \in T_{\texttt{Device}}$
- Ossia, $T_{\texttt{Lamp}} \subseteq T_{\texttt{Device}}$, scritto anche: `Lamp <: Device`

Ogni classe è sottotipo delle interfacce che implementa!

Sottotipi e principio di sostituibilità

Principio di sostituibilità di Liskov (1993)

Se `A` è un sottotipo di `B` allora ogni oggetto (o valore) di `A` deve essere utilizzabile dove un programma si attende un oggetto (o valore) di `B`

Nel caso delle interfacce

Se la classe `C` implementa l'interfaccia `I`, allora ogni istanza di `C` può essere passata dove il programma si attende un elemento del tipo `I`.

Si rischiano errori?

Non per via di chiamate a metodi non definiti! Il programma può manipolare gli elementi del tipo `I` solo mandando loro i messaggi dichiarati in `I`, che sono sicuramente "accettati" dagli oggetti di `C`. Il viceversa non è vero.

Nota: `I` è più generale di `C`, ma fornisce meno funzionalità!

Polimorfismo

Polimorfismo = molte forme (molti tipi)

Ve ne sono di diversi tipi nei linguaggi OO

- Polimorfismo inclusivo: subtyping
- Polimorfismo parametrico: genericità

Polimorfismo inclusivo

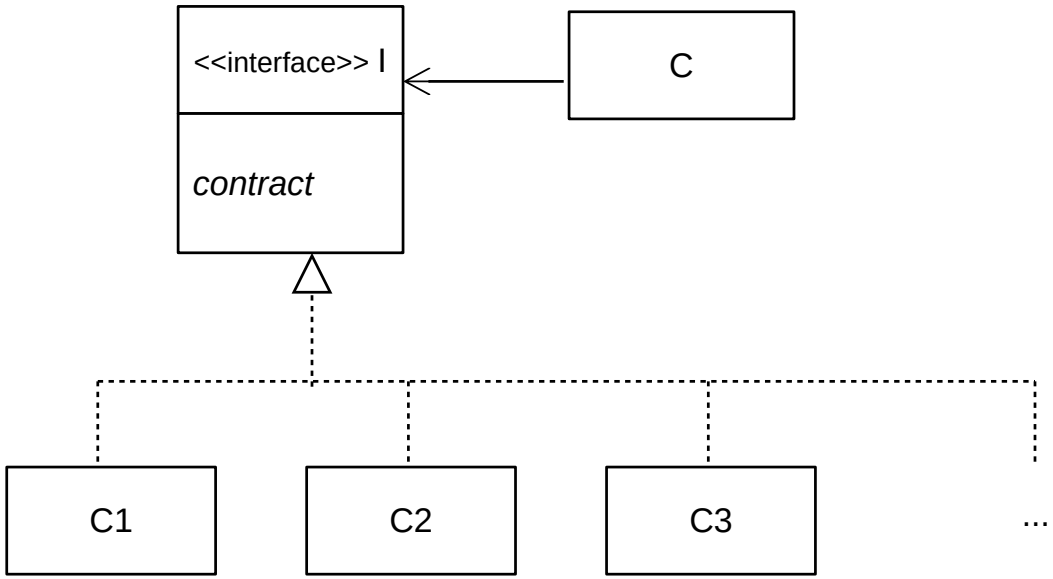
È precisamente l'applicazione del principio di sostituibilità

- Se il tipo `A` è una specializzazione di `B` (lo implementa)..
- ..si può usare un oggetto `A` dove se ne attende uno di `B`

Polimorfismo e interfacce

Una delle pietre miliari dell'OO

- `C` deve usare uno o più oggetti di un tipo non predeterminato
- l'interfaccia `I` cattura il contratto comune di tali oggetti
- varie classi `C1`,`C2`,`C3` (e altre in futuro) implementano `I`
- `C` non ha dipendenze rispetto `C1`,`C2`,`C3`
- (l'uso potrebbe essere una composizione, come nel caso precedente)



Late binding (o dynamic binding)

```
1 public static void switchOnIfCurrentlyOff(Device device){
2     // Collegamento dinamico con i metodi da invocare..
3     if (!device.isSwitchedOn()){
4         device.switchOn();
5     }
6 }
7 /* Codice cliente */
8 Lamp lamp = new Lamp();
9 switchOnIfCurrentlyOff(lamp); // OK, un Lamp è un Device
```

Collegamento ritardato

Accade con le chiamate a metodi non-statici

- Dentro a `switchOnIfCurrentlyOff()` mandiamo a `device` due messaggi (`isSwitchedOn` e `switchOn`), ma il codice da eseguire viene scelto dinamicamente (ossia "late"), dipende dalla classe dell'oggetto `device` (`Lamp`,`TV`,..)
- Terminologia: `device` ha tipo `Device` (tipo statico), ma a tempo di esecuzione è un `Lamp` (tipo run-time)

Early (static) vs late (dynamic) binding

```
1  interface I {
2      void m();
3  }
4
5  class C1 implememts I {
6      void m(){ System.out.println("I'm istance of C1");}
7  }
8
9  class C2 implememts I {
10     void m(){ System.out.println("I'm istance of C2");}
11     static void m2(){ System.out.println("I'm a static method of C2");}
12 }
13
14 // Codice cliente
15
16 I i = Math.random() > 0.5 ? new C1() : new C2();
17 i.m(); // collegamento al body da eseguire è late, ossia dinamico
18
19 C2.m2(); // collegamento al body da eseguire è early, ossia statico
```

Differenze

- Early: con metodi statici o finali
- Late: negli altri casi

Outline

Altri aspetti sulle interfacce

Cosa non può contenere una `interface`?

- Non può contenere campi istanza
- Non può contenere il corpo di un metodo istanza
- Non può contenere un costruttore

Cosa potrebbe contenere una `interface`?

- Java consentirebbe anche di includere dei campi statici e metodi statici (con tanto di corpo), ma è sconsigliato utilizzare questa funzionalità per il momento
- In Java 8 i metodi possono avere una implementazione di default, che vedremo

Le `interface` includano solo intestazioni di metodi!

Implementazione multipla

Implementazione multipla

Dichiarazione possibile: `class C implements I1,I2,I3{..}`

- Una classe `C` implementa sia `I1` che `I2` che `I3`
- La classe `C` deve fornire un corpo per tutti i metodi di `I1`, tutti quelli di `I2`, tutti quelli di `I3`
  - se `I1,I2,I3` avessero metodi in comune non ci sarebbe problema, ognuno andrebbe implementato una volta sola
- Le istanze di `C` hanno tipo `C`, ma anche i tipi `I1`, `I2` e `I3`

Esempio Device e Luminous

```
1  /* Interfaccia per dispositivi */
2  public interface Device{  // Contratto:
3      void switchOn();     // - si può accendere
4      void switchOff();    // - si può spegnere
5      boolean isSwitchedOn(); // - si può verificare se acceso/spento
6  }
7
8  /* Interfaccia per entità luminose */
9  public interface Luminous{       // Contratto:
10     void dim();      // - si può ridurre la luminosità
11     void bright();     // - si può aumentare la luminosità
12 }
13
14 public class Lamp implements Device, Luminous{
15     ...
16     public void switchOn(){ ... }
17     public void switchOff(){ ... }
18     public boolean isSwitchedOn(){ ... }
19     public void dim(){ ... }
20     public void bright(){ ... }
21 }
```

Estensione interfacce

Estensione

Dichiarazione possibile: `interface I extends I1,I2,I3{..}`

- Una interfaccia `I` definisce certi metodi, oltre a quelli di `I1`, `I2`, `I3`
- Una classe `C` che deve implementare `I` deve fornire un corpo per tutti i metodi indicati in `I`, più tutti quelli di `I1`, tutti quelli di `I2`, e tutti quelli di `I3`
- Le istanze di `C` hanno tipo `C`, ma anche i tipi `I`, `I1`, `I2` e `I3`

Esempio LuminousDevice

```
1  public interface Device{
2      void switchOn();
3      void switchOff();
4      boolean isSwitchedOn();
5  }
6  public interface Luminous{
7      void dim();
8      void bright();
9  }
10
11 /* Interfaccia per dispositivi luminosi */
12 public interface LuminousDevice extends Device, Luminous{
13     // non si aggiungono ulteriori metodi
14 }
15
16 public class Lamp implements LuminousDevice{
17     ...
18     public void switchOn(){ ... }
19     public void switchOff(){ ... }
20     public boolean isSwitchedOn(){ ... }
21     public void dim(){ ... }
22     public void bright(){ ... }
23 }
```

Qualche esempio dalle librerie Java

Interfacce base

- `java.lang.Appendable`
- `java.io.DataInput`
- `java.io.Serializable`: interfaccia "tag"
- `javax.swing.Icon`

Implementazioni multiple

- `class ImageIcon implements Icon, Serializable, Accessible{...}`

Estensioni di interfacce

- `interface ObjectInput extends DataInput{...}`

Preview del prossimo laboratorio

Obbiettivi

Familiarizzare con:

- Costruzione di semplici classi con incapsulamento
- Relativa costruzione di test
- Costruzione di classi con relazione d'uso verso una interfaccia