Il metodo `toString()`

Qualsiasi oggetto di Java possiede un metodo `toString()`, che applicato ad un oggetto restituisce, **di default** una stringa così formata:

- La prima parte ha il nome completo della classe (in formato del package)
- La seconda parte ha l'indirizzo dov'è salvato in memoria l'oggetto stesso.
  Il metodo `toString()` è **riprogrammabile all'interno di una qualsiasi classe**:
- Tale metodo deve essere necessariamente `public` in modo che sia accessibile dall'utilizzatore
- Deve avere, come tipo di ritorno, una `String`, realizzata sulla base di cosa si preferisce stampare
  È sempre buona prassi riprogrammare in modo opportuno `toString()`.
  Quando utilizziamo `System.out.println` per stampare un oggetto, anche senza specificare il `toString()`, verrà comunque stampato l'output del `toString()`.
  Questo metodo risulta utile se si vogliono **stampare informazioni relative all'oggetto** in una determinata situazione; nella classe `Lamp` che abbiamo realizzato in classe è utile per restituire le informazioni relative allo stato attuale della lampadina.

La Composizione

L'incapsulamento fa in modo che ogni classe abbia quante meno dipendenze possibili da altre classi; un sistema senza dipendenze è però irrealizzabile, in quanto è alla base della soluzione di più problemi grazie all'approccio *divide et impera*.

Esistono 3 forme di dipendenza tra classi nella OOP:

- Associazione: un oggetto ne **usa** un altro (si dice *uses*); l'uso qui può essere sporadico
- Composizione: un oggetto ne **comprende** altri (si dice *has a*); in questo caso lo stato di un oggetto si compone di stati di altri oggetti da cui è quindi formato.
  - Si parla di aggregazione se un qualche oggetto nella composizione ha anche campi propri, che usa per altri scopi.
- Specializzazione: un oggetto ne specializza un altro; vedremo prossima settimana.

Parlando di **composizione**, si dice che un oggetto A si compone di un insieme di altri oggetti di altre classi B1, B2, ..., Bn.
Come realizziamo la composizione? Non dimenticando le regole di incapsulamento si hanno 4 modi possibili:

- Se A si compone solo di un oggetto di B, A avrà un campo `private` di tipo B.

```
class A {
    private final B b = new B(); // La classe A comprende un solo oggetto di
B

    ...
}

class B {
    ...
}
```

- Se A si compone opzionalmente di un oggetto di B, A avrà un campo `private` di tipo B che può essere anche `null`.

```
class A {
    private B b = new B(); // Rispetto a prima il valore non è final

    public void resetB(){
        this.b = null;
    }
    ...
}

class B {
    ...
}
```

- Se A si compone di un numero n di oggetti di B, A avrà n campi `private` di tipo B (oltre a 2 campi **non va usato**).
- Se A si compone di un numero non specifico di oggetti di B, A avrà un campo `private` di tipo `B[]`.

```
class A {
    private final B[] b = new B[] {new B(), new B(), new B()}; // La classe
A comprende un solo oggetto di B

    ...
}

class B {
    ...
}
```

Esempio: `TwoLampsDevice`

Vogliamo realizzare una base formata da due lampadine, in modo che posso accendere, spegnere o fare entrare in modalità "eco" entrambe le lampadine, andando anche a modificare lo stato delle lampadine singolarmente.
Come possiamo realizzare questo dispositivo, parlando di classi?

- Possiamo reimplementare da capo la classe, utilizzando un campo per ogni lampadina e per il suo stato attuale
- Oppure possiamo utilizzare la composizione, riutilizzando due oggetti della classe `Lamp` creata in precedenza.
  La seconda realizzazione è migliore rispetto alla prima, in quanto evita duplicazioni ed utilizza metodi e classi già realizzati (quindi **non ci ripetiamo**).

L'UML

Il linguaggio UML è uno standard che permette di realizzare diagrammi al fine di mostrare una determinata progettazione di un dato software. Vedremo il *class diagram*:

- Ogni classe è rappresentata da un box rettangolare, diviso in tre parti:
  - La prima contiene il nome della classe
  - La seconda contiene i campi
  - La terza contiene i metodi, inclusi i costruttori
- Per i campi e i metodi vi sono delle specifiche:
  - Si mette davanti - se privati, + se pubblici.
  - Si sottolineano se `static`
  - Dei metodi si riporta solo la dichiarazione, con relativi parametri e tipo di ritorno.
  UML si usa solitamente per la fase di design di un dato software, e quindi si utilizza solo una versione "ridotta" della rappresentazione di classi: in questo caso si inseriscono solamente i metodi e i campi che sono pubblici.

Le interfacce (`interface`)

Supponiamo che diverse classi abbiano tutte uno stesso metodo; al fine di usare tale metodo per ogni oggetto di tale classe è necessario invocarlo ogni volta per ogni oggetto; questa procedura, è sì funzionale, ma molto ripetitiva; potenzialmente, avendo un numero infinito di oggetti, si ha anche un numero infinito di righe di codice.
Al fine di risolvere questo problema è necessario separare in maniera esplicita (ossia tramite dichiarazioni diverse, spesso anche in sorgenti diversi) il "modello" della classe (ossia la sua **interfaccia**) dall'implementazione della classe stessa.

In Java, la keyword `interface` è un nuovo **tipo di dato** dichiarabile in modo simile alle classi:

```
interface I {
    ...
}
```

La differenza sostanziale con una `class` è che all'interno di una `interface` sono contenute **solo delle intestazioni di metodi** (ossia gli "scheletri" di metodi).
Una classe può **implementare** una data interfaccia, specificandolo tramite `implements` - in questo caso la classe che ha implementato tale interfaccia deve **implementare i metodi presenti all'interno dell'interfaccia**:

```
interface I { // Interfaccia, con metodo di test
    void testMethod();
}

class C implements I { // La classe C implementa l'interfaccia I
    public void testMethod(){ // Pertanto il metodo in I va implementato.
        System.out.println("Hello!");
    }
}
```

In questo esempio, se implemento o meno l'interfaccia `I` nella classe `C` e provo a chiamare il metodo `testMethod()` esso funzionerà lo stesso. Ma allora perché usare `interface`?

- Ogni classe che implementa una data `interface` **deve** contenere i metodi presenti al suo interno, altrimenti verrà restituito errore di compilazione. In particolare le `interface` stabiliscono un **contratto** dove le classi che implementano tale `interface` sono costrette ad implementare i metodi presenti al suo interno.
- Le `interface` sono un tipo di dato vero e proprio, pertanto possiedono tutte le proprietà che un tipo di dato può avere; in particolare:
  - Un nome, scelto adeguatamente;
  - Un insieme di oggetti che possono contenere, che in questo caso **sono oggetti delle classi che implementano tale `interface`**
  - Un insieme di operazioni che possono eseguire, in questo caso **sono i metodi presenti all'interno della dichiarazione della `interface`**. Non è possibile utilizzare altri metodi relativi alla classe a cui la `interface` è assegnata.

Possiamo utilizzare come esempio il seguente codice:

```java
public interface ShowHello{
    void hello();
}

class Person implements ShowHello {
    public void hello(){
        System.out.println("Hello!");
    }
}

class Bro implements ShowHello {
    public void hello(){
        System.out.println("Yo!");
    }
}

class Greetings {
    public static void greet(ShowHello someone){
        someone.hello();
    }
}

...

Person normalPerson = new Person();
Bro myBro = new Bro();

greet(normalPerson); // OK, normalPerson è un ShowHello -> "Hello!"
greet(myBro); // OK, myBro è un ShowHello -> "Yo!"
```

In questo esempio, il tipo di classe `ShowHello` raccoglie gli oggetti delle classi `Person` e `Bro` - se in futuro volessi includere altre classi che vogliono utilizzare `greet()`, basterà che esse implementino l'interfaccia `ShowHello`; questo rende il codice notevolmente riusabile e implementabile in più maniere.

In UML, le interfacce sono rappresentate da box rettangolari, divisi in due:

- Nella parte superiore si troverà `<<interface>> <nome_interfaccia>`
- Nella parte inferiore si troveranno i metodi presenti all'interno dell'interfaccia.
  Le classi che implementano tale interfaccia sono collegate a quest'ultima tramite un arco tratteggiato, che si conclude con un triangolo vuoto diretto verso l'interfaccia. Se più classi implementano una stessa interfaccia gli archi vengono collegati.

Tipi, sottotipi e polimorfismo

In termini di insiemistica, se più classi implementano una data `interface`, allora le classi sono **sottoinsiemi** dell'insieme formato dalla `interface`. In OOP i sottoinsiemi prendono il nome di **subtypes** (sottotipi) della `interface`.
I sottotipi aprono un discorso di **sostituibilità** all'interno del codice di un qualsiasi software; in particolare ci riferiamo al **Principio di Sostituibilità di Liskov**:
*"Se A è un sottotipo di B, allora ogni oggetto di A **deve essere utilizzabile** dove un programma si aspetta un oggetto di B"*.
Nel caso di Java, **se abbiamo una `interface I` e una `class C`, se `C` implementa `I`, allora possiamo usare istanze di `C` quando ci vengono richiesti elementi di `I`**. Questa cosa in Java è sempre possibile senza errori perché per manipolare oggetti di `I` si usano solamente metodi dichiarati all'interno di `I`, che sono sicuramente compresi anche in `C`.

Possiamo quindi utilizzare un oggetto `A` di una classe `C` (la quale implementa una `interface I`), sia tramite l'uso diretto di `A`, con i suoi relativi metodi e campi, oppure utilizzarlo nella sua generalizzazione tramite `interface`. Questo è un caso specifico di **polimorfismo**, in particolare si parla di **polimorfismo inclusivo** poiché un tipo di dato ne include un altro.
Questa tipologia di polimorfismo è alla base della OOP; supponiamo che una classe `C` deve usare uno o più oggetti di un tipo non predeterminato - un interfaccia `I` raggruppa gli elementi comuni a tali oggetti. Supponiamo che un insieme di altre classi `C1, C2, C3` implementino tutte `I` - in questo caso la nostra classe `C` iniziale **non subisce ulteriori variazioni** poiché non ha dipendenze dalle altre classi.

Una classe può implementare più interfacce, separandole da virgole dopo la keyword `implements`. In tal caso, gli oggetti della classe hanno sia il tipo della classe stessa, ma anche il tipo delle `interface` che essa ha implementato.

Una `interface` può estendere altre `interface`: ciò significa di fatto andare ad aggiungere altri metodi oltre a quelli presenti nella `interface` che si vuole estendere. Per estendere una `interface` si usa la keyword `extends` in modo simile a come si usa `implements`.