# Ereditarietà e Polimorfismo

## Programmazione

Corso di Laurea in Informatica
Corso di Laurea in Informatica per la Comunicazione Digitale

Università di Camerino

Prof. Michele Loreti

Anno Accademico 2024/25

# Motivazione

## Problema

- realizzare una classe `RecipienteConTappo`, una versione raffinata di `Recipiente` dotata di un meccanismo di apertura e chiusura
- gran parte del codice è comune a quanto già scritto per la classe `Recipiente`
- posto che non tutti i recipienti sono dotati di tappo, e dunque che non è saggio **modificare** la classe `Recipiente`, c'è un modo per non dover duplicare tutto il codice della classe `Recipiente`?

## Soluzione

- ogni recipiente con tappo è anche un recipiente
- studiamo un meccanismo per realizzare la classe `RecipienteConTappo` come **estensione** della classe `Recipiente`, andando a specificare solo le differenze tra oggetti di una classe e dell'altra

# Estensione di una classe

```java
public class RecipienteConTappo extends Recipiente {
    private boolean aperto;

    ...
}
```

# Note

- `extends` è una parola chiave di Java, non può essere usata come identificatore
- la classe `RecipienteConTappo` **estende** la classe `Recipiente`, ne **eredita** tutti i campi e i metodi senza che ci sia bisogno di ridefinirli
- `Recipiente` è detta **classe base** o **super classe**
- `RecipienteConTappo` è detta **classe derivata** o **sotto classe**
- la classe `RecipienteConTappo` **aggiunge** il campo `aperto` a quelli ereditati

# Costruttore di una classe derivata

```
public class RecipienteConTappo extends Recipiente {
    private boolean aperto;

    public RecipienteConTappo(int volume) {
        super(volume);
        aperto = false;
    }
    ...
}
```

- ricordiamo che il costruttore deve preparare ogni istanza di una classe al primo utilizzo, in particolare deve **inizializzare** opportunamente tutti i campi
- `volume` e `contenuto` sono campi privati della classe `Recipiente`, dunque `RecipienteConTappo` **non può farsi carico** di inizializzarli
- `aperto` è un campo privato della classe `RecipienteConTappo`, tale classe **deve farsi carico** della sua inizializzazione
- `super(volume)` invoca il costruttore della classe base, è la prima cosa che il costruttore di una classe derivata deve fare
- `super` è una parola chiave di Java, non può essere usata come identificatore

# Metodi getter e setter per i nuovi campi

```
public class RecipienteConTappo extends Recipiente {
    private boolean aperto;

    ...

    public boolean get_aperto() {
        return aperto;
    }

    public void set_aperto(boolean aperto) {
        this.aperto = aperto;
    }
    ...
}
```

- definiamo nuovi metodi per gestire i campi aggiunti nella classe derivata
- scriviamo `this.aperto` per fare riferimento al campo (variabile di istanza), per non confonderlo con l'argomento omonimo
- `this` può essere usato solo dentro il costruttore e dentro un metodo non statico, è un riferimento all'**oggetto ricevente** l'invocazione

# Overriding di metodi

```
public class RecipienteConTappo extends Recipiente {
    private boolean aperto;

    ...

    public void aggiungi(int quantita) {
        if (aperto) super.aggiungi(quantita);
    }

    public void rimuovi(int quantita) {
        if (aperto) super.rimuovi(quantita);
    }
}
```

- è evidente che il comportamento dei metodi `aggiungi` e `rimuovi` deve essere ridefinito nella classe `RecipienteConTappo`
- il meccanismo di ridefinizione di un metodo in una classe derivata si chiama **overriding** del metodo
- in questo caso, la “vecchia” implementazione però è ancora utile: quando abbiamo bisogno di invocare il “vecchio” metodo del quale è stato fatto overriding, usiamo `super.nome_metodo`

# Polimorfismo

## Problema

In una classe `C != RecipienteConTappo` realizzare una metodo `riempi` che consenta di riempire completamente un recipiente con tappo (se è aperto)

In precedenza abbiamo già risolto il problema per `Recipiente`, siamo costretti a duplicare lo stesso codice anche per `RecipienteConTappo`?

```
public class C {
    ...
    public static void riempi(Recipiente a) {
        a.aggiungi(a.capacita());
    }
    ...
}
```

## Soluzione

- un “recipiente con tappo” è tutto sommato anche un “recipiente”, dunque possiamo usare un “recipiente con tappo” laddove è atteso un “recipiente”

# La classe `Object`

## Codice scritto

```
public class Recipiente {
    ...
}
```

## Come viene interpretato da Java

```
public class Recipiente extends Object {
    ...
}
```

La classe `Object` è definita nella libreria standard di Java ed è la classe “più semplice” di tutte. Tutte le altre classi estendono, direttamente o indirettamente, la classe `Object`

- `Recipiente` estende direttamente `Object` (anche se non l’abbiamo scritto)
- `RecipienteConTappo` estende `Recipiente` la quale estende `Object`, dunque `RecipienteConTappo` estende indirettamente `Object`

# La relazione di sottotipo

## Promemoria

Un **ordine parziale** $\preceq$ di un insieme $X$ è una relazione

- **riflessiva**: $x \preceq x$ per ogni $x \in X$
- **antisimmetrica**: se $x \preceq y$ e $y \preceq x$ allora $x = y$ per ogni $x, y \in X$
- **transitiva**: se $x \preceq y$ e $y \preceq z$ allora $x \preceq z$ per ogni $x, y, z \in X$

## Definizione

Indichiamo con `<:` il più piccolo ordine parziale tra classi tale che `C <: D` se `C` estende `D`.
Diciamo che `C` è **sottotipo** di `D` se `C <: D`

## Esempi

- `Recipiente <: Object`
- `RecipienteConTappo <: Recipiente`
- `RecipienteConTappo <: Object`

# Il principio di sostituzione

Per una classe, essere sottotipo non significa essere “più piccola” o “più semplice”. Quando `C` è **sottotipo** di `D`, la classe `C` in generale

- ha più campi di `D` (es. `aperto`)
- ha più metodi di `D` (es. `get_aperto` e `set_aperto`)
- ha codice più complesso (es. `aggiungi` e `rimuovi`)

# Principio di sostituzione di Liskov

Se `C <: D`, allora è possibile usare una istanza di `C` laddove è attesa una istanza di `D` poiché le istanze di `C` possono fare tutto quello che possono fare le istanze di `D`

# Differenza tra widening e principio di sostituzione

Il **widening** consente, ad esempio, di usare un `int` laddove è atteso un `double`, ma per fare questo effettua una **conversione** di tipo, mentre il principio di sostituzione **non comporta alcuna conversione**

# Esempio: riempimento

```
public class C {
    ...
    public static void riempi(Recipiente a) {
        a.aggiungi(a.capacita());
    }
    ...
}
```

- il metodo `riempi` si attende una istanza di `Recipiente`
- dunque, il metodo `riempi` usa (alcune) caratteristiche di `a` in quanto istanza di `Recipiente`, in particolare invoca i metodi `capacita` e `aggiungi`
- `RecipienteConTappo <: Recipiente`
- dunque, le istanze di `RecipienteConTappo` supportano tutte le operazioni supportate dalle istanze di `Recipiente`
- per il principio di sostituzione di Liskov, posso invocare `riempi` con una istanza di `RecipienteConTappo` anche se `riempi` si attende una istanza di `Recipiente`

# Esperimento 1

```
public class TestSottotipo1 {
    public static void riempi(Recipiente a) {
        a.aggiungi(a.capacita());
    }

    public static void main(String[] args) {
        RecipienteConTappo a = new RecipienteConTappo(5);
        a.set_aperto(true);
        System.out.println("Contenuto = " + a.get_contenuto());
        riempi(a); // qui viene usato il principio di sostituzione
        System.out.println("Contenuto = " + a.get_contenuto());
    }
}
```

```
Contenuto = 0
Contenuto = 5
```

**Nota**: a è una istanza di `RecipienteConTappo`, mentre `riempi` si aspetta una istanza di `Recipiente`, eppure il programma viene compilato ed eseguito correttamente

# Esperimento 2

```
public class TestSottotipo2 {
    public static void riempi(Recipiente a) {
        a.aggiungi(a.capacita());
    }

    public static void main(String[] args) {
        RecipienteConTappo a = new RecipienteConTappo(5);
        // a.set_aperto(true); // invocazione rimossa
        System.out.println("Contenuto = " + a.get_contenuto());
        riempi(a); // qui viene usato il principio di sostituzione
        System.out.println("Contenuto = " + a.get_contenuto());
    }
}
```

```
Contenuto = 0
Contenuto = 0
```

La classe `Recipiente` definisce un metodo `aggiungi` e la classe `RecipienteConTappo` fa override del metodo `aggiungi`. **Quale dei due metodi `aggiungi` viene invocato da `riempi`?**

# Late binding

```
public static void riempi(Recipiente a) {
    a.aggiungi(a.capacita());
}
```

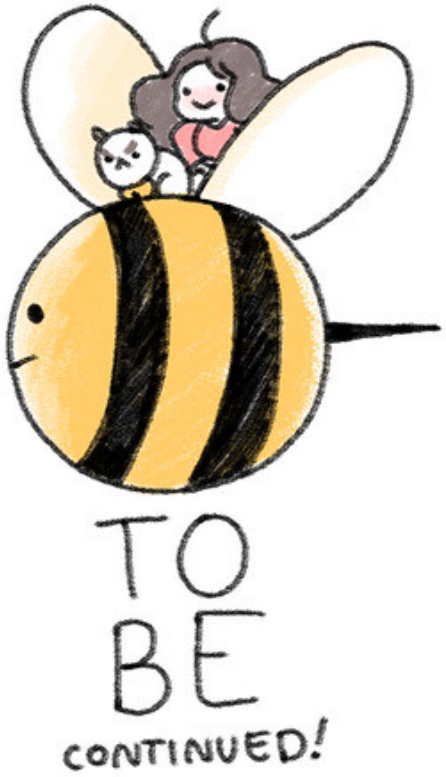
- il metodo `riempi` ha un argomento `a` di tipo `Recipiente`, questo è il **tipo apparente** di `a` (quello che appare al compilatore)
- il metodo `riempi` può essere invocato con istanze di `Recipiente` o di sotto classi di `Recipiente` come `RecipienteConTappo`. La classe di cui è istanza l'argomento in una particolare invocazione di `riempi` è detto **tipo reale** di `a` e può essere diverso da una invocazione all'altra
- il **late binding** è il meccanismo per cui, in presenza di una invocazione `o.m`, il metodo `m` che viene eseguito è quello corrispondente al **tipo reale** di `o`, non al suo tipo apparente

# Esercizi

1. Elencare tutte le relazioni di sottotipo che si possono dedurre date le seguenti definizioni (incomplete) di classi:

```
class A { ... }
class B extends A { ... }
class C extends B { ... }
class D extends A { ... }
class E { ... }
```

2. Definire una classe `Counter` dotata di un metodo `get_next` che ad ogni invocazione restituisce numeri interi via via crescenti a partire da `0`. Definire una classe `CounterBase` che estende `Counter` in cui il contatore è resettato a `0` ogni volta che raggiunge la "base" del contatore (la base è un numero intero positivo fornito come argomento al costruttore di `CounterBase`). Scrivere un programma di prova che mostri il meccanismo del late binding in azione usando le classi `Counter` e `CounterBase`.


TO
BE
CONTINUED!