Classi e Oggetti

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

Strumenti a disposizione

- **tipi di dato predefiniti**: rappresentare e memorizzare informazione
- **espressioni**, **comandi**, **metodi**: trasformare dati

Problemi

- definire nuovi **tipi di dato**
- strutturare e organizzare i programmi (di medie/grandi dimensioni) in unità logiche il più possibile indipendenti tra loro per favorire
  - lo **sviluppo modulare** dei programmi
  - la **verifica di correttezza** delle unità logiche
  - il **riuso** di unità logiche in programmi diversi (librerie)

(una proposta di) soluzione

- **OOP** (Object-Oriented Programming, programmazione orientata agli oggetti)

Esempio: il tipo “recipiente”

Caratteristiche (stato)

- materiale
- etichetta
- volume
- quantità di liquido contenuto
- capacità (volume - contenuto)
- aperto o chiuso
- ...

Comportamenti (operazioni)

- aprire
- chiudere
- versare in
- versare da
- svuotare
- riempire
- ...

Oggetto = identità + stato + comportamento

Stato

- informazioni che caratterizzano l'oggetto in un momento della sua vita
- solo alcune informazioni sono effettivamente rappresentate (**astrazione**)
- lo stato può variare nel tempo (oggetti **mutabili**)

Comportamento

- operazioni che possono essere eseguite con l'oggetto
  (es. leggi l'etichetta, riempi il recipiente)
- le operazioni possono **dipendere** dallo stato dell'oggetto
- le operazioni possono **modificare** lo stato dell'oggetto

Identità

- due recipienti nello stesso stato non sono necessariamente lo stesso recipiente, ciascuno ha una propria identità che lo distingue dagli altri

Classe = famiglia di oggetti

Una **classe** è un tipo di dato che rappresenta una famiglia omogenea di oggetti

- il cui stato è *rappresentato* nello stesso modo
- che supportano le *stesse operazioni*

Gli oggetti che “appartengono” a una certa classe `C` sono detti **istanze** di `C`

Esempio

Supponiamo di voler definire una classe `Recipiente`, preoccupandoci solo di rappresentarne **volume**, **contenuto** e **capacità**

- volume, contenuto e capacità sono dipendenti, due di queste quantità determinano la terza
- lo stato di un recipiente può essere rappresentato con due valori di tipo `int`, uno per il volume e l’altro per il contenuto
- le operazioni supportate da un recipiente includono la lettura di volume, contenuto e capacità, l’aggiunta e la rimozione di contenuto

Definire una classe in Java

Una classe Java `C` deve essere definita in un file con nome `C.java`

`Recipiente.java`

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

Note

- `Recipiente` è il nome della classe scelto dal programmatore
- è convenzione (ma non è obbligatorio) dare alle classi nomi che iniziano con una lettera maiuscola
- `public` e `class` sono parole chiave di Java e non possono essere usate come identificatori
- `public` è un **modificatore di accesso** e indica che la classe definita è **pubblica**, ovvero è visibile ovunque nel programma

Campi

I **campi** (o **variabili di istanza**) definiscono lo stato di ogni istanza della classe

```
public class Recipiente {
    private int volume;     // 0 <= volume
    private int contenuto;  // 0 <= contenuto <= volume
    ...
}
```

Note

- `private` è una parola chiave, non può essere usata come identificatore
- `private` è un altro **modificatore di accesso** e indica che i campi sono **privati**, ovvero sono visibili solo all'interno della classe in cui sono definiti
- i commenti a lato dei campi indicano eventuali **invarianti di classe**, ovvero proprietà che i campi devono rispettare durante tutta la vita di ogni istanza della classe

Principio di incapsulamento

Sebbene sia possibile definire campi pubblici, nella maggior parte dei casi i campi di una classe sono **privati**, rendendoli di fatto inaccessibili (direttamente) dall'esterno della classe

Conseguenze

- se si vuole fornire accesso a un campo (privato), occorre predisporre una o più operazioni (pubbliche) che lo riguardano
- solo il codice della classe (che è circoscritto al file in cui la classe è definita) è autorizzato a modificare i campi privati

Nota

La prassi di definire campi privati il cui accesso è regolamentato da operazioni pubbliche prende il nome di **principio di incapsulamento** ed è uno dei cardini su cui si fonda la OOP

Costruttore

Il **costrutture** di una classe viene eseguito ogni volta che si crea una istanza della classe e prepara l'istanza al primo utilizzo

```
public class Recipiente {
    private int volume;    // 0 <= volume
    private int contenuto; // 0 <= contenuto <= volume

    public Recipiente(int quantita) { // quantita >= 0
        volume = quantita;
        contenuto = 0;
    }
    ...
}
```

- Il costruttore della classe è un **metodo speciale** senza tipo di ritorno e riconoscibile dal nome che coincide con quello della classe
- Il fatto che il costruttore sia pubblico significa che chiunque può creare istanze della classe `Recipiente`

Metodi “getter”

I **metodi getter** sono operazioni di una classe che consentono la **lettura** del valore di (alcuni) campi

```
public class Recipiente {
    private int volume;    // 0 <= volume
    private int contenuto; // 0 <= contenuto <= volume
    public Recipiente(int quantita) { ... }

    public int get_volume()    { return volume; }
    public int get_contenuto() { return contenuto; }
    public int capacita()      { return volume - contenuto; }
    ...
}
```

- il nome di un metodo getter solitamente (ma non sempre) ha la forma `get_nome_campo` (o `getNomeCampo` in camel case)
- **non è detto** che debba esserci un metodo getter per ogni campo
- il metodo `capacita` non è propriamente un metodo “getter”, è comodo per ottenere la capacità di un recipiente definendola una volta per tutte

Altre operazioni di modifica del contenuto

Realizziamo operazioni per aggiungere e rimuovere contenuto

```
    public void aggiungi(int quantita) { // 0 <= quantita
        contenuto = Math.min(contenuto + quantita, volume);
    }

    public void rimuovi(int quantita) { // 0 <= quantita
        contenuto = Math.max(contenuto - quantita, 0);
    }
```

Note

- i commenti indicano **precondizioni** che l'utilizzatore dell'oggetto deve rispettare per eseguire le operazioni richieste
- una versione "robusta" della classe `Recipiente` può segnalare la violazione di queste condizioni con un'**eccezione**
- i metodi `Math.min` e `Math.max` sono definiti nella libreria standard di Java e calcolano minimo e massimo di due numeri

Il principio di incapsulamento in azione

Volume di un recipiente

Il volume di un recipiente può essere letto ma non modificato, il suo valore è determinato una volta per tutte al momento della creazione del recipiente

Contenuto di un recipiente

Il contenuto di un recipiente è inizialmente 0, può essere sia letto che modificato, ma le modifiche possono essere fatte solo in maniera incrementale e nel rispetto degli invarianti

In particolare, non è possibile riempire un recipiente oltre alla sua capacità

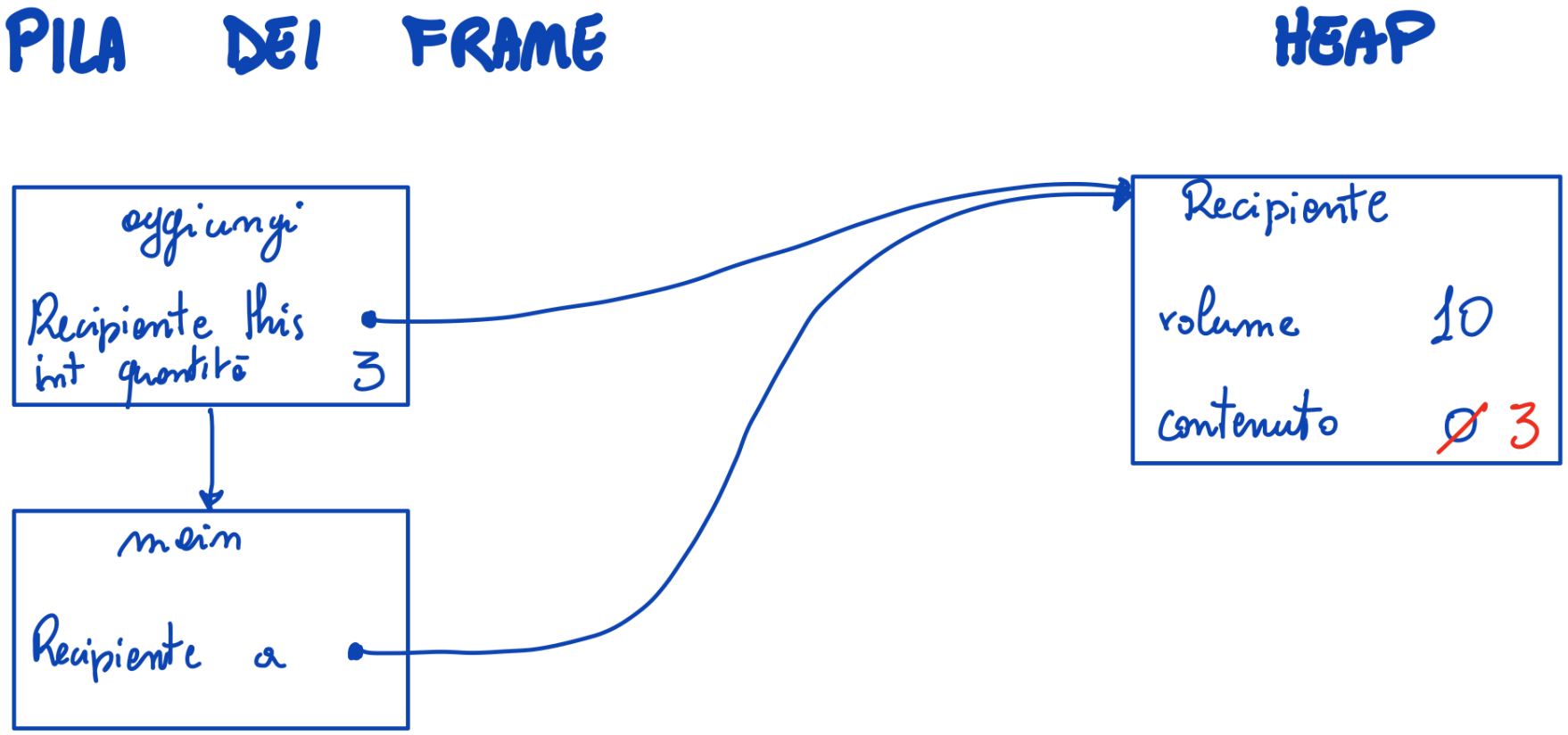
Esempio: creazione e uso di un recipiente

```
public class Main {
    public static void main(String[] args) {
        Recipiente a = new Recipiente(10);
        System.out.println("Capacità A = " + a.capacita());
        a.aggiungi(3);
        System.out.println("Capacità A = " + a.capacita());
    }
}
```

Note

- L'operazione `a.aggiungi(3)` **invoca** il metodo `aggiungi` sull'oggetto `a`, passando il valore 3 come argomento, `a` è detto **oggetto ricevente**
- in Java gli oggetti sono memorizzati nell'**heap** (come gli array)
- la variabile locale `a` contiene solo un **riferimento** all'oggetto (come gli array)
- il riferimento funge anche da **identificatore univoco** dell'oggetto
- la modifica allo stato dell'oggetto `a` altera l'heap, dunque la modifica è "visibile" anche all'esterno del metodo `aggiungi` (come per gli array)

Allocazione degli oggetti in memoria



Esempio: metodo statico di riempimento

In una classe `Main != Recipiente` realizzare una metodo `riempi` che consenta di riempire completamente un recipiente

```
public class Main {
    public static void riempi(Recipiente a) {
        a.aggiungi(a.capacita());
    }
    public static void main(String[] args) {
        Recipiente a = new Recipiente(10);
        riempi(a);
        System.out.println("Capacità A = " + a.capacita());
    }
}
```

Note

- i metodi **statici** non hanno un oggetto ricevente su cui sono invocati (tutti i metodi definiti nella prima parte del corso erano statici)
- il metodo `main` indica il **punto di ingresso** del programma, è il metodo eseguito per primo dall'interprete JVM

Esercizi

1. Definire una classe `Orologio` che consenta di tenere traccia di ora e minuti. Dotare la classe di metodi getter `get_ora` e `get_minuti` per leggere ora e minuti correnti. Dotare la classe di un metodo `tick` che ha come effetto il passaggio da un minuto al successivo.
2. Modificare la classe `Recipiente` in modo tale da rappresentare un recipiente che può essere aperto o chiuso. Dotare la classe di metodi `apri` e `chiudi` per cambiare lo stato del recipiente e modificare i metodi `aggiungi` e `rimuovi` in modo che abbiano un effetto solo se il recipiente è aperto.
3. Definire una classe `ContoCorrente` che supporti le seguenti operazioni: `get_saldo` per la lettura del saldo; `deposita` per effettuare un deposito; `preleva` per effettuare un prelievo; `get_saldo_massimo` per la lettura del saldo più alto mai raggiunto dal conto corrente nel corso della sua esistenza
4. Definire una classe `Primes` con un'unica operazione `next` che, a ogni invocazione, restituisce numeri primi via via crescenti. Ad esempio

```
Primes p = new Primes();
for (int i = 0; i < 10; i++)
    System.out.println(p.next());
```

deve stampare i numeri 2 3 5 7 11 13 17 19 23 29


TO
BE
CONTINUED!