3. Oggetti e Classi Pt. 2: Inizializzazione, Accessi, Distruzione

Questo capitolo completa la trattazione dei meccanismi fondamentali della programmazione orientata agli oggetti in Java, concentrandosi su **come gli oggetti vengono creati**, come si **accede** ai loro membri e come la **memoria** viene gestita. Verranno inoltre discusse le **convenzioni** di programmazione e l'importanza di un codice efficace.

3.1 Codice Statico

In Java, i membri di una classe (campi e metodi) possono essere dichiarati come `static`. Questa parola chiave ha un significato particolare e distingue i membri di istanza (che appartengono a ogni singolo oggetto) dai membri di classe (che appartengono alla classe stessa).

```
class Dog {
    String name;
    Integer age;

    public String bark() {
        System.out.println("Bau Bau");
    }

    public static boolean canDogBark() {
        true
    }
}

Dog pippo = new Dog("pippo", 12);
pippo.bark(); // Bau Bau

canDogBark(); // true
```

3.1.1 Meccanismo dei Membri Statici

- **Campi Statici**:
    - Un campo dichiarato `static` appartiene alla **classe**, non a una specifica istanza di essa.
    - Esiste una sola copia di un campo statico, condivisa da tutte le istanze della classe.
    - Viene inizializzato una sola volta, quando la classe viene caricata in memoria dalla JVM.
    - Sono spesso usati per rappresentare costanti (`final static`) o dati che devono essere condivisi tra tutti gli oggetti di una classe (es. un contatore di istanze).
    - *Esempio*: `java.lang.System.out` è un campo statico che rappresenta lo stream di output standard.
- **Metodi Statici**:
    - Un metodo dichiarato `static` appartiene alla classe e può essere invocato direttamente sulla classe, senza la necessità di creare un'istanza dell'oggetto.
    - I metodi statici non possono accedere direttamente ai campi o ai metodi non statici (di istanza) della stessa classe, perché non operano su una specifica istanza. Possono accedere solo ad altri membri statici.
    - Sono spesso usati per funzioni di utilità che non dipendono dallo stato di un oggetto (es. funzioni matematiche, metodi di conversione).
    - *Esempio*: Il metodo `java.util.Arrays.toString()` è un metodo statico che converte un array in una stringa. Il metodo `main` di un programma Java è sempre statico, perché è il punto di ingresso del programma e deve poter essere invocato senza che esista un'istanza della classe.

Tabella 3.1: Differenze tra Membri di Istanza e Membri Statici

Caratteristica	Membri di Istanza (non statici)	Membri Statici
Appartenenza	All'**istanza** dell'oggetto	Alla **classe** stessa
Memoria	Una copia per ogni oggetto	Una sola copia, condivisa da tutte le istanze
Accesso	Tramite un oggetto (`oggetto.campo` , `oggetto.metodo()`)	Tramite la classe (`Classe.campo` , `Classe.metodo()`)
Dipendenza	Dipende dallo stato dell'oggetto	Indipendente dallo stato dell'oggetto
Esempio	`p.x` (dove `p` è un `Point`)	`Math.PI` , `System.out.println()`

(Informazioni basate su `04-objects-2_slides.pdf` , Pag. 5 e conoscenza generale di Java)

3.2 Costruttori

I costruttori sono metodi speciali utilizzati per **inizializzare** lo stato di un oggetto appena creato.

3.2.1 Meccanismo dei Costruttori

- Un costruttore ha lo stesso nome della classe e non ha un tipo di ritorno (nemmeno `void`)
- Viene invocato automaticamente quando si crea una nuova istanza di una classe utilizzando l'operatore `new` .
- Il suo scopo principale è assicurare che l'oggetto sia in uno stato valido e coerente subito dopo la sua creazione.
- Se non viene definito alcun costruttore esplicito, Java fornisce un **costruttore di default** senza argomenti, che inizializza i campi ai loro valori predefiniti (0 per numerici, `false` per booleani, `null` per riferimenti a oggetti).

Esempio:

```
@NoArgsConstructor
@AllArgsConstructor
class Point {
    double x;
    double y;

    public Point() {
        this(0.0, 0.0);
    } // new Point();

    public Point(x, y) {
        this.x = x;
        this.y = y;
    } // new Point(12.3, 41.4);
}

Point p2 = new Point();
// Point {
//   x: 0.0,
//   y: 0.0,
// }
```

3.2.2 Chiamate `this(...)`

Un costruttore può invocare un altro costruttore della stessa classe utilizzando la parola chiave `this()` . Questa chiamata deve essere la prima istruzione all'interno del costruttore. È utile per evitare la duplicazione di codice quando si hanno più costruttori con diverse liste di parametri.

Esempio (come sopra, il costruttore `Point()` chiama `Point(double, double)`):

```
Point() {
    this(0.0, 0.0); // Inizializza un punto alle coordinate (0,0)
}
```

3.3 Overloading di Metodi e Costruttori

L'overloading (o sovraccarico) è la capacità di avere **più metodi** (o **costruttori**) con lo **stesso nome** all'interno della stessa classe, purché abbiano liste di parametri diverse.

3.3.1 Meccanismo dell'Overloading

- **Regola**: Due metodi (o costruttori) sono considerati sovraccaricati se hanno lo stesso nome ma differiscono per il **numero** o il **tipo** dei parametri, o per l'**ordine** dei tipi dei parametri
- **Tipo di ritorno**: Il tipo di ritorno del metodo *non* è sufficiente per distinguere metodi sovraccaricati.
- **Scopo**: L'overloading migliora la leggibilità del codice, permettendo di usare un nome intuitivo per operazioni simili che accettano input diversi.

Esempio di overloading di metodi:

```
class Calculator {
    int add(int a, int b) {
        return a + b;
    }

    double add(double a, double b) { // Overload: tipi di parametri diversi
        return a + b;
    }

    int add(int a, int b, int c) { // Overload: numero di parametri diverso
        return a + b + c;
    }
}
```

Esempio di overloading di costruttori (da `04-objects-2_slides.pdf`, Pag. 10):

```
class Point {
    double x, y;
    Point(double x, double y) { this.x = x; this.y = y; } // Costruttore 1
    Point() { this(0.0, 0.0); } // Costruttore 2 (overload del primo)
}
```

3.4 Controllo d'Accesso (Access Modifiers)

I modificatori di accesso in Java controllano la **visibilità** dei campi, dei metodi, dei costruttori e delle classi. Sono fondamentali per implementare il principio di **incapsulamento** e **information hiding.**

3.4.1 Livelli di Accesso

Java fornisce quattro livelli di controllo d'accesso:

1. `public`:
   - Massima visibilità.
   - Membri `public` sono accessibili da qualsiasi classe, in qualsiasi package.
   - Usato per l'interfaccia pubblica di una classe.
2. `protected`: (Verrà approfondito nel Capitolo 7 sull'ereditarietà)
   - Accessibile all'interno dello **stesso package** e dalle **sottoclassi** (anche se in un package diverso).
   - Usato per membri che devono essere accessibili alle **classi derivate**.
3. `default` **(package-private)**:
   - Nessun modificatore esplicito.
   - Accessibile solo all'interno dello stesso package.

- Non è visibile alle classi in altri package.

4. `private`:
    - Minima visibilità.
    - Membri `private` sono accessibili **solo all'interno della classe in cui sono dichiarati**.
    - Essenziale per **l'incapsulamento**, nascondendo i dettagli di implementazione e proteggendo lo stato interno dell'oggetto.

Tabella 3.2: Riepilogo dei Modificatori di Accesso

Modificatore	Stessa Classe	Stesso Package	Sottoclasse (altro package)	Qualsiasi Classe (altro package)
`private`	Sì	No	No	No
`default`	Sì	Sì	No	No
`protected`	Sì	Sì	Sì	No
`public`	Sì	Sì	Sì	Sì

3.4.2 Metodologia di Progettazione con Incapsulamento

Una metodologia di progettazione standard per le classi in Java prevede:

1. **Campi `private`**: Dichiarare tutti i campi (variabili di istanza) come `private`. Questo nasconde lo stato interno dell'oggetto e impedisce accessi diretti e non controllati dall'esterno.
2. **Metodi `public` (Getter/Setter)**: Fornire metodi `public` per accedere e modificare lo stato dei campi, se necessario.
    - **Getter (Accessors)**: Metodi che restituiscono il valore di un campo (es. `int getAge()`).
    - **Setter (Mutators)**: Metodi che impostano il valore di un campo (es. `void setAge(int newAge)`).
    - Questi metodi permettono di controllare come i dati vengono letti e scritti, consentendo la validazione dell'input o l'esecuzione di logica aggiuntiva.
3. **Metodi di Servizio `public`**: Fornire altri metodi `public` che rappresentano il comportamento dell'oggetto e che operano sul suo stato interno.

Esempio di classe incapsulata:

```
class Lamp {
    private double intensity; // Campo privato

    public void setIntensity(double intensity) { // Setter pubblico
        if (intensity >= 0.0 && intensity <= 1.0) {
            this.intensity = intensity;
        } else {
            System.out.println("Intensità non valida.");
        }
    }

    public double getIntensity() { // Getter pubblico
        return this.intensity;
    }
}
```

Questo approccio garantisce che la classe sia ben progettata, manutenibile e robusta, poiché le modifiche interne non dovrebbero influenzare il codice esterno che interagisce solo con l'interfaccia pubblica.

3.5 Finalizzazione e Garbage Collection

La gestione della memoria in Java è automatica, grazie al Garbage Collector (GC), che si occupa di **liberare la memoria occupata da oggetti non più utilizzati**.

3.5.1 Garbage Collection

- **Meccanismo**: Il Garbage Collector è un processo automatico che identifica e distrugge gli oggetti che non sono più referenziati da nessuna parte nel programma.
- **Vantaggi**: Gli sviluppatori non devono preoccuparsi di allocare o deallocare esplicitamente la memoria, riducendo il rischio di memory leak e dangling pointer.
- **Quando avviene**: Il GC viene eseguito periodicamente dalla JVM. Non è possibile forzarne l'esecuzione o prevedere esattamente quando avverrà.

3.5.2 Finalizzazione (`finalize()` metodo)

- Il metodo `finalize()` è un metodo `protected` della classe `Object` (la superclasse di tutte le classi in Java) che può essere sovrascritto.
- Se un oggetto ha un metodo `finalize()` sovrascritto, la JVM lo invocherà *prima* che l'oggetto venga raccolto dal Garbage Collector.
- **Uso sconsigliato**: L'uso di `finalize()` è fortemente sconsigliato per la pulizia delle risorse. Non c'è garanzia che venga chiamato, o quando. Per la pulizia delle risorse (es. chiusura di file, connessioni di database), è preferibile usare blocchi `try-with-resources` o implementare l'interfaccia `AutoCloseable`.

3.6 Programmazione Strutturata e Convenzioni di Codice

Oltre ai meccanismi orientati agli oggetti, una buona pratica di programmazione richiede l'adesione a principi di programmazione strutturata e a convenzioni di stile.

3.6.1 Programmazione Strutturata

La programmazione strutturata si basa su **tre costrutti fondamentali** per controllare il flusso di esecuzione del programma:

1. **Sequenza**: Le istruzioni vengono eseguite nell'**ordine** in cui appaiono.
2. **Selezione (Condizione)**: Permette di eseguire blocchi di codice diversi in base a una condizione (`if-else`, `switch`).
3. **Iterazione (Ciclo)**: Permette di ripetere un blocco di codice più volte (`for`, `while`, `do-while`).

Questi costrutti, uniti all'uso di funzioni/metodi, permettono di scrivere codice chiaro, leggibile e manutenibile.

3.6.2 Convenzioni sul Codice (Code Style)

Seguire convenzioni di stile standard è cruciale per migliorare la leggibilità e la comprensione del codice, specialmente in team

- **Formattazione**: Utilizzare una formattazione consistente (es. 4 spazi) per migliorare la leggibilità della struttura del codice.
- **Lunghezza delle linee**: Mantenere le linee di codice entro una lunghezza ragionevole (es. non più di 90 caratteri) per evitare lo scorrimento orizzontale.
- **Formattazione delle parentesi graffe**: Adottare uno stile consistente per il posizionamento delle parentesi graffe (`{}`).
- **Una istruzione per linea**: Ogni istruzione dovrebbe idealmente occupare una singola linea per chiarezza.
- **Nomi significativi**: Usare nomi descrittivi per variabili, metodi e classi (es. `isValid`, `calculateTotal`).
- **Commenti**: Inserire commenti per spiegare logica complessa, decisioni di design o parti non ovvie del codice (da `03-structured_slides.pdf`, Pag. 38).
- **Esempio di convenzioni di naming in Java (da `07a-codestyle_slides.pdf`, Pag. 6):**

Elemento	Convenzione	Esempio
Classi	`PascalCase`	`MyClass`, `Point3D`
Metodi	`camelCase`	`myMethod()`, `calculateSum()`
Campi	`camelCase`	`myField`, `totalCount`
Costanti (`final static`)	`ALL_CAPS_WITH_UNDERSCORES`	`MAX_VALUE`, `PI`
Package	`lowercase`	`com.example.myapp`

3.6.3 CheckStyle

Strumenti come **CheckStyle** possono automatizzare la verifica del rispetto delle convenzioni di stile.

- **Funzionalità**: CheckStyle è un tool che analizza il codice sorgente Java per verificare la conformità a un insieme di regole di stile predefinite o personalizzate.
- **Integrazione**: Può essere integrato nei sistemi di build (es. Gradle, Maven) per far fallire la build se le regole di stile non sono rispettate, garantendo la coerenza del codice in tutto il progetto.
- **Output**: Genera report dettagliati sugli errori di stile, indicando la linea e la colonna del problema.

3.6.4 Programmazione Efficace (Effective Programming)

Esistono tecniche di programmazione che migliorano l'efficacia dello sviluppo. Molte di queste sono connesse alle best practice della programmazione OO e all'uso di **pattern di progettazione** (soluzioni riutilizzabili a problemi comuni). Libri come "Effective Java" e "Design Patterns" sono riferimenti importanti in questo campo. L'obiettivo è prediligere soluzioni semplici e compatte che non siano né più complesse né più lente del necessario.

3.7 Controllo del Flusso di Esecuzione (Programmazione Strutturata)

La programmazione strutturata fornisce i costrutti per controllare l'ordine in cui le istruzioni vengono eseguite.

3.7.1 Istruzioni Condizionali

- `if-else`: Esegue un blocco di codice se una condizione è vera, altrimenti (opzionalmente) un altro blocco (da `03-structured_slides.pdf`, Pag. 10).

```
if (condizione) {
    // codice se condizione è vera
} else {
    // codice se condizione è falsa
}
```

- `switch`: Permette di selezionare un blocco di codice da eseguire tra diversi casi possibili, basandosi sul valore di un'espressione (da `03-structured_slides.pdf`, Pag. 13).

```
switch (espressione) {
    case valore1:
        // codice per valore1
        break; // importante per uscire dallo switch
    case valore2:
        // codice per valore2
        break;
    default:
        // codice se nessun caso corrisponde
}
```

Java moderno (dalla 14 in poi) ha introdotto le **switch expressions**, che permettono di usare `switch` per restituire un valore e con una sintassi più compatta (`->`) (da `01-oo-abstraction_slides.pdf`, Pag. 34).

3.7.2 Cicli (Loop)

- `while`: Ripete un blocco di codice finché una condizione è vera. La condizione viene valutata prima di ogni iterazione (da `03-structured_slides.pdf`, Pag. 16).

```
while (condizione) {
    // codice da ripetere
}
```

- `do-while`: Simile a `while`, ma la condizione viene valutata dopo la prima iterazione, garantendo che il blocco di codice venga eseguito almeno una volta

```
do {
    // codice da ripetere
```

```
} while (condizione);
```

- `for`: Utilizzato per cicli con un numero predefinito di iterazioni, o quando si ha un contatore

```
for (inizializzazione; condizione; aggiornamento) {
    // codice da ripetere
}
```

- `for-each` **(Enhanced For Loop)**: Semplifica l'iterazione su array o collezioni

```
for (TipoElemento elemento : collezione) {
    // codice per ogni elemento
}
```

3.7.3 Istruzioni di Salto

- `break`: Termina immediatamente il ciclo o lo `switch` più interno (da `03-structured_slides.pdf`, Pag. 22).
- `continue`: Salta l'iterazione corrente di un ciclo e passa all'iterazione successiva (da `03-structured_slides.pdf`, Pag. 23).
- `return`: Termina l'esecuzione di un metodo e restituisce un valore (se il metodo non è `void`) (da `03-structured_slides.pdf`, Pag. 24).

3.8 Array

Gli array sono strutture dati che permettono di memorizzare una collezione di elementi dello stesso tipo in una sequenza contigua di memoria.

3.8.1 Dichiarazione e Inizializzazione di Array

- **Dichiarazione**: `TipoElemento[] nomeArray;` o `TipoElemento nomeArray[];` (da `03-structured_slides.pdf`, Pag. 25).
- **Creazione**: Gli array sono oggetti in Java, quindi vengono creati con `new`.
  - `nomeArray = new TipoElemento[dimensione];` (crea un array con `dimensione` elementi, inizializzati ai valori di default del tipo).
  - `TipoElemento[] nomeArray = {valore1, valore2, ...};` (crea e inizializza l'array con i valori specificati).
- **Accesso**: Gli elementi sono accessibili tramite un indice numerico che parte da 0 (es. `nomeArray[0]`).
- **Proprietà** `length`: Ogni array ha un campo `length` che indica il numero di elementi (da `03-structured_slides.pdf`, Pag. 27).

Esempio (da `03-structured_slides.pdf`, Pag. 25-27):

```
int[] numbers = new int[5]; // Array di 5 interi, inizializzati a 0
numbers[0] = 10;
System.out.println(numbers[0]); // Stampa 10

String[] names = {"Alice", "Bob", "Charlie"}; // Array di stringhe
System.out.println(names.length); // Stampa 3
```

3.8.2 Array Multidimensionali

Java supporta array multidimensionali, che sono essenzialmente array di array.

- **Dichiarazione**: `TipoElemento[][] nomeArray;` (da `03-structured_slides.pdf`, Pag. 28).
- **Esempio**:

```
int[][] matrix = new int[3][3]; // Matrice 3×3
matrix[0][0] = 1;

int[][] raggedArray = new int[2][]; // Array "frastagliato"
```

```
raggedArray[0] = new int[3];
raggedArray[1] = new int[5];
```

3.9 Performance e JVM

Un'ultima nota riguarda le performance in Java.

- **Critiche storiche**: I linguaggi orientati agli oggetti e le macchine virtuali (come la JVM) sono stati storicamente criticati per essere più lenti rispetto ai linguaggi imperativi/strutturati compilati direttamente in codice macchina.
- **Miglioramenti recenti**: Grazie a tecniche avanzate come la **JIT (Just-In-Time) compilation**, le JVM moderne hanno ridotto significativamente, e in alcuni casi annullato, le differenze di performance. La JIT compilation traduce il bytecode in codice macchina nativo durante l'esecuzione, ottimizzando le parti di codice più utilizzate.
- **Focus del corso**: In questo contesto, l'attenzione non è sui dettagli delle performance, ma sulla scelta di soluzioni di design più semplici e compatte, evitando soluzioni sia più complesse che più lente