7. Polimorfismo, Classi Astratte e Tipi a Runtime

Questo capitolo approfondisce il concetto di polimorfismo in Java, esplorando la sua connessione con l'ereditarietà e le interfacce, introducendo le classi astratte come meccanismo per definire comportamenti parziali e discutendo la gestione dei tipi a runtime.

7.1 Polimorfismo Inclusivo con le Classi

Il **polimorfismo** inclusivo (o subtyping) è un pilastro della programmazione orientata agli oggetti che permette a un oggetto di un sottotipo di essere **utilizzato ovunque sia atteso un oggetto del suo supertipo**.

In parole semplici, il polimorfismo inclusivo significa che:

- Puoi trattare oggetti di sottoclassi come se fossero oggetti della loro superclasse.
- Un riferimento a una superclasse può puntare a un oggetto di qualsiasi sottoclasse derivata da essa.
- Quando chiami un metodo su quel riferimento, il metodo effettivo che viene eseguito è quello definito nella classe dell'oggetto reale, non nella classe del riferimento. Questo è noto come *dispatch dinamico* o *legame tardivo*.

gli elementi chiave che abilitano il polimorfismo inclusivo in Java sono:

1. **Ereditarietà (Inheritance):** Il polimorfismo inclusivo si basa sull'ereditarietà. Una sottoclasse (classe figlia) eredita attributi e metodi da una superclasse (classe genitore). Questo crea una relazione "è un" (is-a): una `Cane` *è un* `Animale`, un `Quadrato` *è una* `Forma`.
2. **Sovrascrittura di Metodi (Method Overriding):** Questo è il meccanismo chiave. Una sottoclasse può fornire la propria implementazione specifica di un metodo che è già definito nella sua superclasse. La firma del metodo (nome e parametri) deve essere la stessa.

7.1.1 Ereditarietà e Principio di Sostituibilità

Come già accennato nel Capitolo 4, il **Principio di Sostituibilità di Liskov** è fondamentale: se B è un sottotipo di A, allora ogni oggetto di B deve poter essere utilizzabile ovunque ci si attenda un oggetto di A (da `12-polymorphism_slides.pdf`, Pag. 5).

Con l'ereditarietà (`class B extends A { ... }`):

- La classe B eredita tutti i membri (campi, metodi) di A e non può restringere la loro visibilità.
- Gli oggetti della classe B rispondono a tutti i messaggi previsti dalla classe A (e, in generale, a qualcuno in più).
- Di conseguenza, un oggetto di B può essere passato dove se ne aspetta uno di A senza problemi di tipo. In Java, B è considerato un sottotipo di A a tutti gli effetti (da `12-polymorphism_slides.pdf`, Pag. 5).

Esempio (da `12-polymorphism_slides.pdf`, Pag. 7):

```
class Switchable {
   public void foo();
}

class D {
   Switchable c;
   public void m(Switchable c) { c.foo(); }
}

class Fan extends Switchable {}
class Lamp extends Switchable {}

// Uso
D d = new D();
d.c = new Fan(); // OK: Fan è un sottotipo di Switchable
d.m(new Lamp());  // OK: Lamp è un sottotipo di Switchable
```

Le sottoclassi di `C` (come `C1`, `C2`) sono compatibili con `C` perché supportano lo stesso contratto, hanno lo stesso stato (o di più) e un comportamento auspicabilmente compatibile.

7.1.2 Layout degli Oggetti in Memoria

Comprendere come gli oggetti sono disposti in memoria aiuta a capire la sostituibilità (da `12-polymorphism_slides.pdf`, Pag. 8). Sebbene ogni JVM possa avere implementazioni leggermente diverse, alcuni elementi sono comuni:

- Un oggetto in memoria inizia con un'intestazione ereditata da `Object` (circa 16 byte), che include informazioni sul tipo a runtime e una tabella di puntatori ai metodi (per supportare il *late-binding* o *dynamic dispatch*).
- Seguono i campi della classe, a partire da quelli delle superclassi.
- Questo significa che un oggetto di una sottoclasse `Switchable` è simile a un oggetto della sua superclasse `A`: ha solo informazioni aggiuntive in fondo, il che semplifica la sostituibilità.

7.1.3 Esempio di Polimorfismo tra Classi: `Person`, `Student`, `Teacher`

Consideriamo una gerarchia di classi con `Person` come superclasse e `Student` e `Teacher` come sottoclassi (da `12-polymorphism_slides.pdf`, Pag. 9).

UML Semplificato:
(da `12-polymorphism_slides.pdf`, Pag. 9)

Esempio di codice (da `12-polymorphism_slides.pdf`, Pag. 10-13):

```
public class Person {
    private final String name;
    private final int id;
    public Person(final String name, final int id) {
        this.name = name;
        this.id = id;
    }
    public String getName() { return this.name; }
    public int getId() { return this.id; }
    public String toString() { return "P [name=" + this.name + ", id=" + this.id + "]"; }
}

public class Student extends Person {
    final private int matriculationYear;
    public Student(final String name, final int id, final int matriculationYear) {
        super(name, id);
        this.matriculationYear = matriculationYear;
    }
    public int getMatriculationYear() { return matriculationYear; }
    @Override
    public String toString() {
        return "S [getName()=" + getName() + ", getId()=" + getId() + ", matriculationYear=" + matriculationYear + "]";
    }
}

public class Teacher extends Person {
    final private String[] courses;
    public Teacher(final String name, final int id, final String[] courses) {
        super(name, id);
        this.courses = java.util.Arrays.copyOf(courses, courses.length);
    }
    public String[] getCourses() {
        return java.util.Arrays.copyOf(courses, courses.length); // Copia difensiva
    }
    @Override
    public String toString() {
        return "T [getName()=" + getName() + ", getId()=" + getId() + ", courses=" + java.util.Arrays.toString(courses) +
"]";
    }
```

```
}

public class UsePerson {
    public static void main(String[] args) {
        final var people = new Person[]{
            new Student("Marco Rossi", 334612, 2013),
            new Student("Gino Bianchi", 352001, 2013),
            new Student("Carlo Verdi", 354100, 2012),
            new Teacher("Mirko Viroli", 34121, new String[]{"OOP", "PPS", "PC"})
        };
        for (final var p : people) {
            System.out.println(p.getName() + ": " + p); // Polimorfismo: toString() chiamato sul tipo runtime
        }
    }
}
```

In `UsePerson`, un array di `Person` può contenere oggetti `Student` e `Teacher`, e i metodi appropriati (`toString()`) vengono invocati in base al tipo effettivo a runtime.

7.1.4 Differenza con il Polimorfismo con le Interfacce

- **Polimorfismo con le Interfacce**: Riguarda solo un contratto. Una classe che implementa un'interfaccia `I` aderisce a un certo comportamento (`I.method()`), ma non eredita implementazioni di campi o metodi da una classe base. È possibile implementare più interfacce (ereditarietà multipla di contratto). `IMPLEMENTS`
- **Polimorfismo con le Classi (Ereditarietà)**: Riguarda sia il contratto che il comportamento. Una classe eredita implementazioni da una superclasse. In Java, non è possibile estendere più classi (ereditarietà singola), per evitare problemi come il "Diamond Problem". `EXTENDS`

7.1.5 Riassunto sul Polimorfismo Inclusivo

- **Polimorfismo**: Fornisce supertipi che raggruppano classi uniformi tra loro, utilizzabili da funzionalità o contesti ad alta riusabilità (es. collezioni omogenee di oggetti)
- **Con le Interfacce**: Relativo solo a un contratto. Facilita l'adesione al contratto per classi esistenti. Spesso porta a un alto numero di interfacce (principio di segregazione delle interfacce).
- **Con le Classi**: Relativo a contratto e comportamento. Di solito si aderisce per costruzione dall'inizio. Vincolato dall'ereditarietà singola.

7.1.6 Come Simulare l'Ereditarietà Multipla

Sebbene Java non supporti l'ereditarietà multipla di classi, è possibile simularla combinando ereditarietà singola e implementazione di interfacce, spesso con l'uso della **delegazione**

- Si definiscono interfacce per i diversi contratti.
- Si creano classi di implementazione concrete per ciascuna interfaccia.
- Una classe che deve combinare più comportamenti può estendere una delle classi di implementazione e delegare le altre funzionalità a istanze delle altre classi di implementazione (composizione).

7.2 Tipi a Runtime

In Java, ogni oggetto è un'istanza di `java.lang.Object`. Questo permette di creare funzionalità che operano su qualunque oggetto.

7.2.1 Everything is an Object

La radice comune `Object` per tutte le classi consente di fattorizzare il comportamento comune a ogni oggetto e di costruire funzionalità che lavorano su qualunque oggetto (da `12-polymorphism_slides.pdf`, Pag. 19).

- *Esempi*: Container polimorfici (es. `Object[]`), definizione di metodi con numero variabile di argomenti (`Object...`).

Uso di `Object[]` (da `12-polymorphism_slides.pdf`, Pag. 20):

```
public class AObject {
    public static void main(String[] s) {
        final Object[] os = new Object[5];
        os[0] = new Object();
        os[1] = "stringa";
        os[2] = Integer.valueOf(10);
        os[3] = new int[]{10, 20, 30}; // Array di primitivi, ma boxed come Object
        os[4] = new java.util.Date();
        printAll(os);
        System.out.println(java.util.Arrays.toString(os)); // toString() superficiale
        System.out.println(java.util.Arrays.deepToString(os)); // deepToString() per array di array/oggetti
    }
    public static void printAll(final Object[] array) {
        for (final Object o : array) {
            System.out.println("Oggetto:" + o.toString()); // Late-binding di toString()
        }
    }
}
```

7.2.2 Tipo Statico e Tipo a Runtime

- **Tipo Statico**: Il tipo di dato di una variabile dichiarato nel codice (es. `Object o;`).
- **Tipo Runtime (Dinamico)**: Il tipo di dato effettivo dell'oggetto a cui la variabile fa riferimento in un dato momento (es. `o` potrebbe essere un `String` o un `Integer`).
- In caso di polimorfismo, le chiamate di metodo sono fatte per *late-binding* (o *dynamic dispatch*), il che significa che l'implementazione del metodo invocata è determinata dal tipo runtime dell'oggetto, non dal suo tipo statico

7.2.3 Ispezione del Tipo a Runtime (`instanceof`) e Conversioni di Tipo (Cast)

In alcuni casi, è necessario ispezionare il tipo a runtime di un oggetto per eseguire operazioni specifiche. Questo si fa con l'operatore `instanceof`

- `instanceof`: Verifica se un oggetto è un'istanza di una certa classe o di una sua sottoclasse.
- **Conversioni di Tipo (Cast)**:
  - **Upcast**: Da sottoclasse a superclasse (spesso automatico e sempre sicuro).
  - **Downcast**: Da superclasse a sottoclasse (potrebbe fallire a runtime con una `ClassCastException` se l'oggetto non è effettivamente del tipo di destinazione)

```
public class AObject2 {
    public static void printAllAndSum(final Object[] array) {
        int sum = 0;
        for (final Object o : array) {
            System.out.println("Oggetto:" + o.toString());
            if (o instanceof Integer) { // Test a runtime
                final Integer i = (Integer) o; // Downcast (sicuro qui grazie a instanceof)
                sum = sum + i.intValue(); // intValue() è unboxing
            }
        }
        System.out.println("Somma degli Integer: " + sum);
    }
}
```

Java 14 ha introdotto il **Pattern Matching for** `instanceof`, che semplifica il codice combinando il test e il cast in un'unica espressione:

```
if (o instanceof Integer i) { // 'i' è automaticamente castato a Integer se la condizione è vera
    sum = sum + i.intValue();
}
```

7.2.4 Autoboxing dei Tipi Primitivi e Argomenti Variabili (`varargs`)

- **Autoboxing/Unboxing**: Java supporta la conversione automatica tra tipi primitivi (es. `int`) e i loro corrispondenti tipi wrapper (es. `Integer`). Questo permette di trattare i primitivi come oggetti quando necessario
  - `Integer i = 10;` (autoboxing: `int` a `Integer`)
  - `int j = i;` (unboxing: `Integer` a `int`)
- **Variable Arguments (`varargs`)**: Permette a un metodo di accettare un numero variabile di argomenti dello stesso tipo. L'ultimo (o unico) argomento di un metodo può essere dichiarato con `Type... argname`. All'interno del metodo, `argname` è trattato come un array `Type[]`

```
public class VarArgs {
    public static int sum(final Integer... args) { // Accetta 0 o più Integer
        int sum = 0;
        for (int i : args) { // Autoboxing/Unboxing implicito
            sum = sum + i;
        }
        return sum;
    }
    public static void printAll(final String start, final Object... args) { // Accetta 0 o più Object
        System.out.println(start);
        if (args.length < 10) {
            for (final Object o : args) {
                System.out.println(o);
            }
        }
    }
    public static void main(String[] s) {
        System.out.println(sum(10, 20, 30, 40)); // Passa singoli Integer
        printAll("inizio", 1, 2, 3.2, true, new int[]{10}, new Object()); // Passa tipi diversi
    }
}
```

7.3 Classi Astratte

Le classi astratte sono un costrutto intermedio **tra interfacce e classi concrete**, che permettono di definire classi con un **comportamento parziale**.

7.3.1 Motivazioni e Definizione

- **Scopo**: Le classi astratte sono usate per descrivere classi il cui comportamento è incompleto (alcuni metodi sono dichiarati ma non implementati)
- **Non istanziabili**: Non è possibile creare direttamente istanze di una classe astratta con l'operatore `new`. Devono essere estese da classi concrete che ne completano l'implementazione.
- **Membri**: Possono definire campi, costruttori, metodi concreti e metodi astratti. I metodi astratti non hanno un corpo e sono dichiarati con la parola chiave `abstract` (es. `public abstract int m();`).
- **Ereditarietà e Interfacce**: Possono estendere un'altra classe (astratta o concreta) e implementare interfacce. Se implementano un'interfaccia ma non tutti i suoi metodi, i metodi non implementati diventano automaticamente astratti nella classe astratta.
- Una sottoclasse di una classe astratta può essere concreta solo se fornisce implementazioni per tutti i metodi astratti ereditati.

7.3.2 Pattern Template Method

Una tipica applicazione delle classi astratte è il **Pattern Template Method**

- **Intento**: Definisce lo scheletro (template) di un algoritmo in un metodo (spesso `final` per impedirne l'override), lasciando l'implementazione di alcuni passaggi specifici alle sottoclassi tramite metodi astratti.
- **Vantaggio**: Garantisce che la struttura generale dell'algoritmo sia mantenuta, mentre le sottoclassi possono personalizzare parti specifiche.

Esempio `LimitedLamp:`
Consideriamo una `SimpleLamp` e una `LimitedLamp` astratta che introduce il concetto di esaurimento. Le sottoclassi concrete (`UnlimitedLamp`, `CountdownLamp`, `ExpirationTimeLamp`) implementano la logica specifica di esaurimento.

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

public abstract class LimitedLamp extends SimpleLamp {
    public LimitedLamp() { super(); }
    /* Questo metodo è finale: regola la coerenza con okSwitch() e isOver() */
    public final void switchOn() { // TEMPLATE METHOD
        if (!this.isSwitchedOn() && !this.isOver()) {
            super.switchOn();
            this.okSwitch(); // Metodo astratto chiamato dal template
        }
    }
    protected abstract void okSwitch(); // Da implementare nelle sottoclassi
    public abstract boolean isOver(); // Da implementare nelle sottoclassi
    public String toString() { return "Over: " + this.isOver() + ", switchedOn: " + this.isSwitchedOn(); }
}

// Esempio di sottoclasse concreta
public class UnlimitedLamp extends LimitedLamp {
    public UnlimitedLamp() { super(); }
    @Override protected void okSwitch() { /* non faccio nulla */ }
    @Override public boolean isOver() { return false; }
}
```

Il metodo `switchOn()` in `LimitedLamp` è un template method: definisce la logica generale di accensione, ma delega i dettagli specifici (`okSwitch()`, `isOver()`) alle sottoclassi concrete.

7.3.3 Classi Astratte vs. Interfacce (con Metodi Default)

Con l'introduzione dei metodi `default` nelle interfacce (Java 8), la distinzione tra classi astratte e interfacce si è leggermente attenuata, ma rimangono differenze cruciali

Caratteristica	Classe Astratta	Interfaccia (con metodi default)
Stato (Campi)	Può definire variabili d'istanza (stato)	Non può definire variabili d'istanza (solo costanti `public static final`)
Costruttori	Può definire costruttori	Non può definire costruttori
Visibilità Membri	Può definire membri con visibilità diverse (`private`, `protected`, `default`, `public`)	Tutti i metodi sono implicitamente `public` (anche i default), le costanti `public static final`
Override `Object`	Può fare overriding di metodi da `Object` (es. `equals`, `hashCode`)	I metodi `default` non possono fare overriding di metodi da `Object` (solo i metodi astratti possono essere implementati)
`final` sui metodi	I metodi concreti possono essere `final`	I metodi `default` non possono essere `final`
Ereditarietà	Ereditarietà singola (`extends`)	Ereditarietà multipla (`implements`)

In sintesi, le classi astratte sono più adatte quando si vuole condividere stato e comportamento di base con le sottoclassi, mentre le interfacce sono ideali per definire contratti di comportamento che possono essere implementati da classi non correlate.

</text/markdown>
Procediamo con il prossimo capitolo.