Eco-design Digitale di Base per i servizi ICT

Programmazione in Java

Massimo Giaccone, Luglio 2025

Obiettivi della lezione

- Comprendere i meccanismi dell’astrazione in Java
- Acquisire competenze base per testing di classi

Cos'è l'astrazione

- L'astrazione è un principio dell'OOP per nascondere i dettagli implementativi e mostrare solo le funzionalità essenziali.
- Permette di progettare sistemi più chiari, flessibili e manutenibili.

L'astrazione ci consente di concentrarci su cosa un oggetto fa, e non su come lo fa. In pratica, definiamo il "contratto" di un oggetto, separandolo dalla sua implementazione concreta. Questo principio è alla base delle interfacce e delle classi astratte.

Classi Astratte in Java

- Una classe astratta è una classe che non può essere istanziata.
- Può contenere metodi astratti (senza corpo) e metodi concreti.

Le classi astratte servono quando vogliamo fornire una base comune a più classi, ma non ha senso creare oggetti direttamente da essa.

```
abstract class Veicolo {
    abstract double consumo();
    void accendi() {
        System.out.println("Motore acceso");
    }
}
```

Quando usare una classe astratta?

- Quando più classi condividono logica comune.
- Quando vogliamo evitare istanziazioni dirette di una superclasse.
- Quando serve un modello base per classi simili.

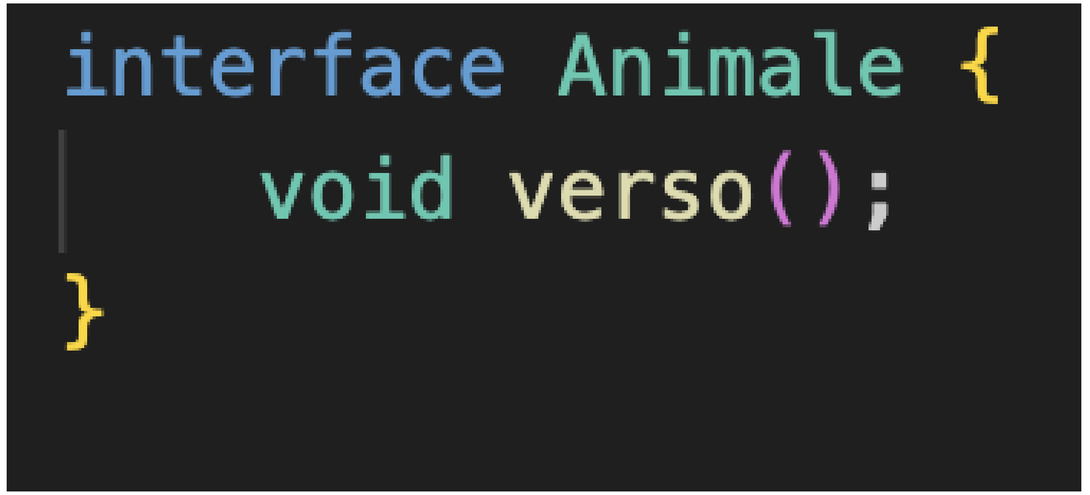
Esercizio 1

Creare una classe astratta Dipendente con i seguenti metodi:

- Metodo astratto calcolaStipendio()
- Metodo concreto infoGenerali() che stampa "Dipendente azienda XYZ"
  Creare due classi concrete Impiegato e Dirigente con implementazioni differenti di calcolaStipendio().

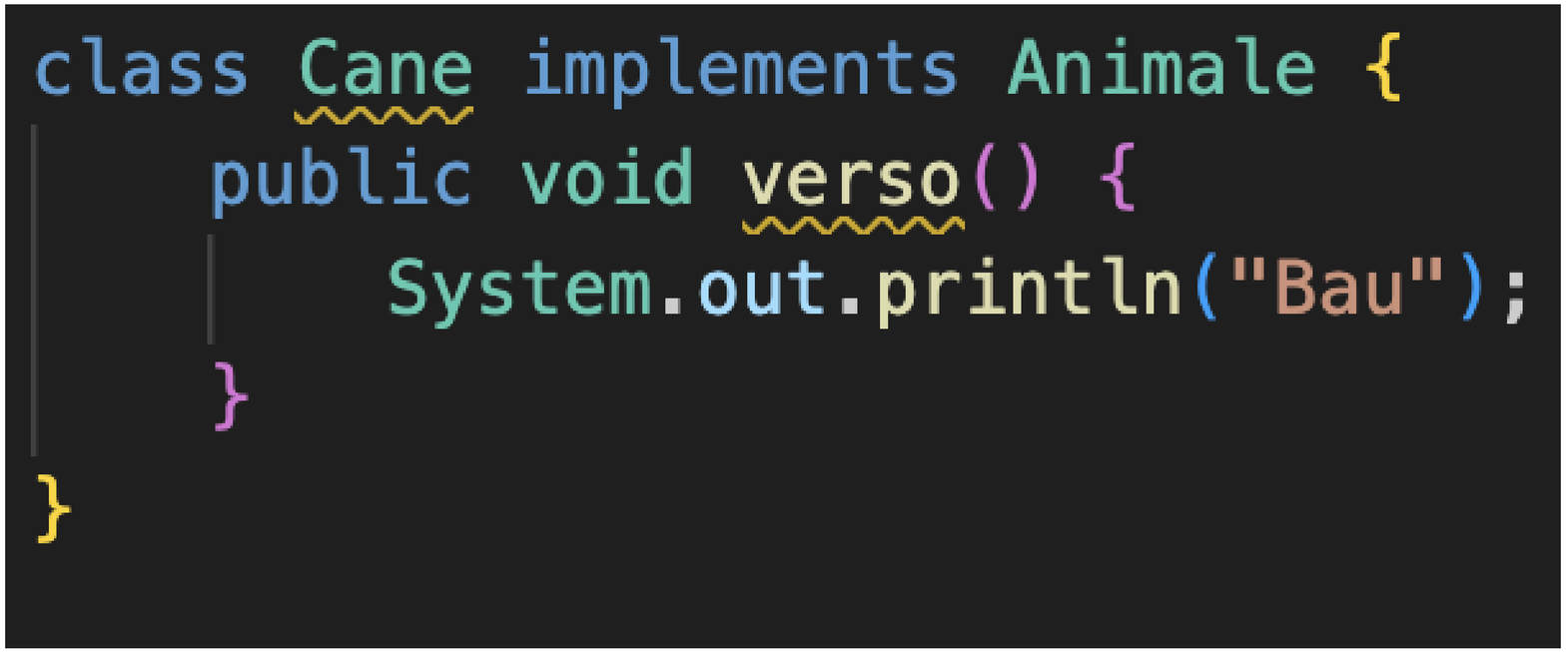
Le interfacce in Java

Un'interfaccia definisce un insieme di metodi che una classe deve implementare. Un'interfaccia è una sorta di "scheletro" di una classe. Non ha implementazione, ma obbliga le classi che la implementano a definire i metodi indicati. Questo garantisce coerenza nel comportamento delle classi, anche se sono diverse tra loro.



Le interfacce in Java

Per usare un'interfaccia, una classe deve implementarla con la parola chiave implements. Deve quindi fornire una definizione concreta per tutti i metodi dell'interfaccia.



Implementazione multipla

Per usare un'interfaccia, una classe deve implementarla con la parola chiave implements. Deve quindi fornire una definizione concreta per tutti i metodi dell'interfaccia.

```
class Cane implements Animale {
    public void verso() {
        System.out.println("Bau");
    }
}
```

Esercizio 2

Definire un'interfaccia Operazione con metodo esegui(int a, int b).
Implementare due classi:

- Somma che restituisce la somma dei due numeri
- Moltiplicazione che restituisce il prodotto Scrivere un metodo che accetta un oggetto Operazione e due numeri, ed esegue l'operazione.

Interfacce vs Classe astratte

Le interfacce sono ideali per descrivere un comportamento comune non legato all'implementazione, mentre le classi astratte sono più adatte a modellare relazioni gerarchiche con codice condiviso. La scelta dipende dallo scenario: comportamenti indipendenti = interfacce; relazioni strutturali = classi astratte.

Aspetto	Interfaccia	Classe Astratta
Istanziabile	No	No
Ereditarietà multipla	Sì	No
Metodi concreti	Solo default	Sì
Attributi	Solo static final	Sì, anche istanza
Costruttori	No	Sì

Esempio Forme geometriche

Progettare un sistema flessibile per rappresentare e calcolare le aree di diverse forme geometriche, utilizzando i concetti di interfaccia e classe astratta per massimizzare la riusabilità del codice e la manutenibilità del sistema.

Esercizio 3

Progettare un sistema per rappresentare diversi **dispositivi multimediali** (es. lettori audio, video, dispositivi combinati).
Ogni dispositivo può **riprodurre** un contenuto, ma non tutti supportano gli stessi formati o funzionalità aggiuntive (es. pausa, stop, avanzamento veloce).

Progetta una gerarchia che consenta di:

- rappresentare comportamenti comuni (es. riproduci())
- distinguere tra dispositivi audio, video o entrambi
- permettere l'estendibilità per aggiungere nuovi tipi in futuro

Obiettivo

Progetta il sistema sia:

- **usando interfacce** (es. Riproducibile, Pausabile, Avanzabile)
- **usando una classe astratta** base (es. DispositivoMultimediale)

Interfaccia vs Testing

Aspetto	Interfacce	Classe Astratta
Flessibilità	Alta	Più rigida
Codice riutilizzabile	No	Sì (metodi comuni)
Più comportamenti	Combinabili liberamente	Limitato a una gerarchia
Estendibilità	Aggiunta semplice di nuove interfacce	Facile estensione con override
Comportamenti opzionali	Deve essere implementato ovunque	Si può lanciare eccezione

Introduzione al testing

- Il testing serve a verificare che il codice funzioni come previsto
- Aiuta a prevenire bug e regressioni
- Supporta lo sviluppo iterativo e sicuro

Perché testare?

Scrivere test consente di scoprire errori subito, senza doverli rincorrere in produzione. Inoltre, i test ben scritti diventano una forma di documentazione sempre aggiornata. Infine, permettono di modificare e migliorare il codice con la sicurezza di non romperne il comportamento.

Intro a Junit 5

- JUnit è un framework per scrivere test automatizzati in Java
- JUnit 5 è l'ultima versione moderna e modulare
- Usa annotazioni per definire i test

Struttura test

```java
import static org.junit.jupiter.api.Assertions.assertEquals;
import org.junit.jupiter.api.Test;

public class MainTest {

    public static int somma(int n, int m) {
        return n + m;
    }

    @Test
    public void testSomma() {
        assertEquals(4, somma(n:2, m:2));
    }
}
```

Tipi di testing

- **Unit Test**: testano singole unità (metodi/classi)
- **Integration Test**: testano l'interazione tra componenti
- **System Test**: verificano l'intero sistema

Annotazioni principali

- @Test → indica un metodo di test
- @BeforeEach → eseguito prima di ogni test
- @AfterEach → eseguito dopo ogni test
- Queste annotazioni ci permettono di strutturare meglio i nostri test.

Asserzioni più comuni

- assertEquals(a, b) → verifica che a == b
- assertTrue(condizione)
- assertFalse(condizione)
- assertThrows(Exception.class, () -> {...})

Organizzazione dei test

Seguire una struttura ordinata nei test rende il progetto più manutenibile.

- Ogni classe test corrisponde a una classe di produzione
- Convenzione: nome ClasseTest.java
- Posizione in cartelle: src/test/java (con Maven/Gradle)

TDD (Test Driven Development)

Scrivere prima i test, poi il codice

- Ciclo: scrivi un test → fallisce → scrivi codice → passa il test → refactoring

Il TDD cambia il modo di scrivere software. Questo aiuta a chiarire i requisiti e produrre codice più semplice, focalizzato e testabile.

Buone pratiche nel testing

- Ogni test deve essere isolato e indipendente
- Scrivere test chiari e leggibili
- Usare nomi descrittivi: testSommaConNumeriPositivi()