Eco-design Digitale di Base per i servizi ICT

Programmazione in Java

Massimo Giaccone, Luglio 2025

Obiettivi della lezione

- Comprendere il paradigma OOP in Java
- Creare classi e oggetti
- Usare riferimenti e costruttori
- Capire l’uguaglianza tra oggetti

Dato vs Oggetto

- Un **dato** è un valore semplice (es. int, char)

- Un **oggetto** ha:
  - Stato (attributi)
  - Comportamento (metodi)

Classi come timbri

- Classe = timbro: definisce forma, struttura e comportamento, ma non è un oggetto concreto.
- Oggetto = impronta: istanza concreta della classe, con identità e stato propri, indipendente dalle altre.
- In Java, new Classe() crea un oggetto, cioè un’impronta del timbro.
- Possiamo avere molti oggetti della stessa classe, ognuno con valori differenti per gli attributi.

Perché serve la OOP?

- Riutilizzabilità del codice
- Incapsulamento dei dati
- Organizzazione modulare

Struttura classe Java

- **Struttura di una classe Java**
- public class Persona { String nome; int eta; }
- Attributi: nome, eta
- Visibilità predefinita: package-private

Istanziare un oggetto

- ***Persona p = new Persona();***
- ***p.nome = "Luca";***
- ***p.eta = 25;***

Costruttori

```
public Persona(String n, int e) {
    nome = n;
    eta = e;
}
```

Overloading costruttori

- L'**overloading dei costruttori** è una tecnica potente che permette di creare **più modi per istanziare gli oggetti** di una classe. Questo significa che si possono definire diversi costruttori, ognuno con un numero o tipo di parametri differente.

- Java capisce automaticamente quale costruttore usare in base ai parametri passati alla creazione di un nuovo oggetto. Questa **flessibilità** rende le classi più versatili e facili da utilizzare, fornendo diverse "interfacce" per la stessa funzionalità e rendendo il codice più intuitivo.

Uso keyword this

- La parola chiave **this** è uno strumento cruciale nella programmazione orientata agli oggetti, impiegata per risolvere l'ambiguità tra i nomi dei parametri (o variabili locali) e gli attributi di una classe.
- **Gestione dell'Ambiguità**
- Quando un parametro di un costruttore o di un metodo ha lo stesso nome di un attributo della classe, sorge un conflitto.
- La keyword **this** supera questa ambiguità fornendo un **riferimento esplicito all'oggetto corrente**.

Uso keyword this

Il Concetto di this

- Oltre alla risoluzione dell'ambiguità, **this** incarna l'identità dell'oggetto corrente. Agisce come un riferimento interno che consente all'oggetto di accedere in modo univoco ai propri attributi e metodi.

Chiarezza e Leggibilità del Codice

- Sebbene non sia sempre obbligatorio in assenza di ambiguità, l'uso di **this** è spesso preferito per aumentare l'esplicitezza e la leggibilità del codice.

Classe Contatore

- public class Contatore { private int valore = 0; public void incrementa() { valore++; } public void azzera() { valore = 0; } public int getValore() { return valore; } }

- public class TestContatore { public static void main(String[] args) { Contatore c = new Contatore(); c.incrementa(); System.out.println(c.getValore()); } }

Introduzione ai riferimenti

- I **riferimenti** sono un concetto cardine in Java e la loro comprensione è essenziale per capire come gli oggetti vengono gestiti in memoria.

Come Java Gestisce la Memoria e gli Oggetti

- Quando creiamo un oggetto usando la parola chiave new, Java riserva uno spazio dedicato nella **memoria heap** per tutti gli attributi di quell'oggetto. È importante notare che la variabile che dichiariamo (ad esempio, Persona p) non contiene l'oggetto stesso. Contiene invece un **riferimento** a quell'oggetto. Questo riferimento è come un indirizzo di memoria che ci permette di "trovare" l'oggetto quando ne abbiamo bisogno.
- Questa distinzione è cruciale: a differenza di altri linguaggi in cui le variabili possono contenere direttamente i valori, in Java ogni accesso a un oggetto tramite una variabile implica seguire un riferimento che porta all'oggetto reale nella memoria.

Introduzione ai riferimenti

I riferimenti in Java sono intrinsecamente più **sicuri** rispetto ai puntatori tradizionali che si trovano in linguaggi come il C++. Java gestisce automaticamente molti aspetti che potrebbero altrimenti portare a errori comuni:

- **Niente aritmetica sui riferimenti**: Non è possibile eseguire operazioni matematiche sui riferimenti.
- **Accesso indiretto alla memoria**: Non si può accedere direttamente alla memoria tramite i riferimenti.
- **Garbage collection automatica**: Java si occupa autonomamente di liberare la memoria quando gli oggetti non sono più in uso, evitando perdite di memoria.
- Nonostante questa sicurezza integrata, comprendere il funzionamento dei riferimenti è fondamentale. Molti comportamenti di Java dipendono da questa conoscenza, in particolare quando si tratta di confrontare oggetti, passare oggetti come parametri ai metodi o gestire collezioni.

Riferimenti

- L'esempio Contatore c1 = new Contatore(); Contatore c2 = c1; c2.incrementa(); dimostra una delle caratteristiche più significative dei riferimenti in Java: l'**assegnazione tra variabili di tipo oggetto copia il riferimento, non l'oggetto stesso**.
- Quando eseguiamo Contatore c2 = c1;, non creiamo un nuovo oggetto Contatore. Invece, sia c1 che c2 diventano **due riferimenti distinti che puntano allo stesso, unico oggetto** nella memoria heap. Questo significa che c1 e c2 sono essenzialmente due etichette diverse per lo stesso "contenitore" di dati.
- Di conseguenza, qualsiasi azione intrapresa tramite uno dei riferimenti influenzerà l'oggetto condiviso.

- Questa caratteristica è potente ma può generare confusione. Se l'intenzione fosse quella di creare una **copia indipendente** dell'oggetto, sarebbe necessario invocare esplicitamente new Contatore() per creare un nuovo oggetto e poi copiare manualmente i valori degli attributi dal primo al secondo.
- La comprensione dei riferimenti condivisi è cruciale anche per capire come gli oggetti vengono **passati ai metodi** in Java: si passa una copia del riferimento, non una copia dell'oggetto. Questo implica che un metodo può modificare l'oggetto originale attraverso il riferimento che riceve.

null e inizializzazione

- La parola chiave **null** in Java significa che una variabile di tipo oggetto non punta a nessun oggetto valido in memoria.
- Quando una variabile è null, tentare di usarla come se fosse un oggetto (ad esempio, chiamando un suo metodo) provoca un **NullPointerException (NPE)**.

Per evitare i NullPointerException, è fondamentale adottare pratiche di **programmazione difensiva**:

- **Inizializzare le variabili**: Quando possibile, assegna un oggetto valido a una variabile non appena la dichiari.
- **Controllare null**: Se non puoi inizializzare subito una variabile, verifica sempre che non sia null prima di tentare di usarla (ad esempio, con un if (variabile != null)).
- Comprendere null è anche un modo per distinguere tra l'esistenza di una variabile in sé (che c'è sempre) e l'esistenza dell'oggetto a cui dovrebbe riferirsi (che potrebbe non esserci se la variabile è null).

Identità vs Uguaglianza

- La distinzione tra **identità** e **uguaglianza** è un concetto cruciale nella programmazione orientata agli oggetti.

- L'operatore == in Java confronta l'**identità**: verifica se due riferimenti puntano esattamente allo **stesso oggetto fisico** in memoria. L'identità di un oggetto è la sua posizione unica in memoria e non può cambiare.

- L'**uguaglianza**, d'altra parte, è un concetto **logico** che può essere definito in base alle nostre esigenze.

Metodo equals

- Mentre l'operatore == è fisso e confronta sempre l'identità, il **metodo equals()** ci permette di definire il nostro concetto personalizzato di **uguaglianza logica** per gli oggetti di una classe. Questo significa che possiamo specificare quando due oggetti devono essere considerati equivalenti.

Esempio Point

- L'esempio della classe Point con costruttori multipli mette in mostra una tecnica elegante e avanzata: la **chiamata da un costruttore all'altro tramite this()**. Questa tecnica permette di evitare la duplicazione del codice e di stabilire una gerarchia logica tra i vari costruttori.
- Il costruttore più "completo", come public Point(double x, double y, double z), contiene tutta la logica necessaria per inizializzare un oggetto Point nello spazio tridimensionale. I costruttori "semplificati", come public Point(double x, double y) per punti bidimensionali, non duplicano questa logica. Invece, utilizzano **this(x, y, 0)** per chiamare il costruttore completo, passando un valore predefinito (come 0 per la coordinata z). Questo approccio, noto come **"constructor chaining" (concatenamento di costruttori)**, garantisce che tutta la logica di inizializzazione sia centralizzata in un unico punto, rendendo il codice più "DRY" (Don't Repeat Yourself) e coerente. È fondamentale ricordare che la chiamata this() deve essere la **prima istruzione** nel costruttore; questa regola di Java assicura che l'oggetto sia completamente inizializzato prima che qualsiasi altra operazione venga eseguita, migliorando la robustezza del linguaggio.

Alias di oggetti

- L'esempio Contatore a = new Contatore(); Contatore b = a; dimostra in modo pratico il concetto di **riferimenti condivisi**. Quando eseguiamo Contatore b = a;, non stiamo creando un nuovo oggetto Contatore; stiamo semplicemente creando un nuovo nome, b, che si riferisce allo **stesso oggetto** a cui si riferisce a. In questo scenario, a e b sono **alias** per la medesima entità in memoria.
- Qualsiasi operazione eseguita tramite uno di questi alias influenzerà l'unico oggetto sottostante.
- Comprendere gli alias è cruciale per la programmazione Java, poiché gli oggetti vengono passati ai metodi, inseriti in collezioni o assegnati a variabili sempre come riferimenti, non come copie complete.

Confronto tra == e equals

- L'operatore == è **rigido e immutabile**: esegue sempre un confronto di **identità**, verificando se due riferimenti puntano esattamente allo stesso oggetto in memoria. Questo comportamento non può essere modificato o personalizzato; è come chiedere "questi due riferimenti si riferiscono fisicamente alla stessa cosa?". La risposta è un semplice sì o no.

- Il metodo equals(), al contrario, è **flessibile e personalizzabile**. Ogni classe può definire il proprio criterio di **uguaglianza logica** sovrascrivendo questo metodo. La risposta dipende interamente dalla logica implementata nel metodo.

Esercizio 1

Si crei una classe LIBRO con:

- Attributi: titolo, autore
- Metodi: costruttore, stampa(), equals(Libro), toString()

Si testi la classe con due oggetti e si confrontino con i metodi equals o ==.

Ereditarietà in Java

- L'ereditarietà risolve brillantemente questo problema permettendo il **riutilizzo del codice esistente senza duplicazioni**.

Come Funziona l'Ereditarietà in Java: extends e Specializzazione

- L'ereditarietà è un meccanismo fondamentale della programmazione orientata agli oggetti che consente di definire una nuova classe basandosi su una esistente, acquisendone automaticamente tutti gli attributi e i metodi. In Java, questo si realizza con la parola chiave **extends**, che stabilisce una **relazione gerarchica** tra due classi: la **classe derivata (o sottoclasse)** e la **classe base (o superclasse)**.

La classe derivata può:

- **Aggiungere** nuovi metodi e attributi per funzionalità specifiche.
- **Ridefinire** metodi esistenti per modificarne il comportamento, mantenendo però la stessa interfaccia pubblica.

Questo meccanismo permette di creare **gerarchie di classi** che riflettono relazioni naturali, dove concetti più specifici ereditano caratteristiche da concetti più generali, specializzandoli in base alle proprie esigenze.

Counter -> Counter2

- L'esempio della classe Counter2 che estende Counter illustra i concetti fondamentali dell'ereditarietà e, al contempo, mette in luce una sfida comune che si presenta quando si lavora con gerarchie di classi. La classe Counter2 viene definita in modo semplice, utilizzando la parola chiave **extends** seguita dal nome della classe da cui si vuole ereditare (Counter). L'obiettivo è creare un contatore che, oltre alle funzionalità di incremento ereditate, possa anche decrementare il proprio valore tramite un nuovo metodo dec().
- Tuttavia, l'implementazione proposta di val-- all'interno del metodo dec() evidenzia subito un problema cruciale: l'attributo **val** è stato dichiarato come **private** nella classe Counter. Questo significa che val non è direttamente accessibile dalle sottoclassi. Questo esempio è didatticamente efficace perché mostra come l'**incapsulamento** e l'ereditarietà possano entrare in conflitto se non gestiti correttamente. La variabile val, essendo privata, rimane completamente nascosta alla classe derivata, impedendo qualsiasi accesso diretto. Questo comportamento non è un difetto del linguaggio, ma una caratteristica progettuale che preserva l'integrità dell'incapsulamento anche in presenza di ereditarietà.

Visibilità protected

- Il linguaggio offre diversi **modificatori di accesso** per controllare chi può vedere e usare i membri di una classe.
- **private: Incapsulamento Totale**

Il modificatore **private** assicura che i membri di una classe siano accessibili solo al suo interno. Questo significa che sono completamente nascosti a tutte le altre classi, comprese le sottoclassi. Questo livello di protezione è fondamentale per mantenere l'**integrità dei dati** e applicare un **incapsulamento rigoroso**.

- **protected: Il Compromesso nell'Ereditarietà**

Un membro dichiarato protected rimane nascosto al "mondo esterno" (cioè non è accessibile da classi non correlate), ma diventa **visibile a tutte le sottoclassi**, anche se si trovano in package diversi. Questo crea un'interfaccia "semi-pubblica" che facilita l'estensione delle classi, mantenendo comunque un controllo sull'accesso ai dati.

Costruttori ed ereditarietà

- I **costruttori** gestiscono l'inizializzazione degli oggetti e hanno un comportamento specifico nell'ereditarietà che li distingue dagli altri metodi. Un principio chiave è che i costruttori **non vengono mai ereditati automaticamente**.

Riutilizzo della Logica di Inizializzazione della Superclasse con super()

- Java offre un meccanismo per riutilizzare la logica di inizializzazione della superclasse tramite la parola chiave **super()**. Questa chiamata deve essere la **prima istruzione** nel costruttore della sottoclasse e può essere usata per invocare un costruttore specifico della classe genitore, passando i parametri necessari.

Counter -> Counter 2 pt2

- Counter2 estende la classe base Counter e aggiunge la nuova funzionalità di decremento tramite il metodo dec(). Ora, questo metodo può accedere all'attributo val della superclasse grazie al modificatore **protected** (o tramite un metodo pubblico fornito da Counter).
- La definizione di due costruttori nella sottoclasse mostra come gestire correttamente l'inizializzazione in presenza di ereditarietà:
- Il costruttore senza parametri (Counter2()) chiama **super()** per invocare il costruttore di default della classe genitore.
- Il costruttore con parametro (Counter2(int v)) chiama **super(v)** per passare il valore iniziale al costruttore parametrizzato della superclasse.
- Implementazione che si concentra esclusivamente sulla propria funzionalità aggiuntiva, delegando a Counter tutto ciò che riguarda l'implementazione del contatore base. Il risultato è un codice pulito, modulare e facilmente estendibile, dove ogni classe ha una responsabilità specifica e ben definita all'interno della gerarchia.

Estendere e ridefinire i metodi

- Una delle caratteristiche più potenti dell'ereditarietà è la possibilità di ridefinire i metodi ereditati dalla classe base, un meccanismo noto come **overriding**.
- Quando una sottoclasse ridefinisce un metodo della superclasse, deve usare esattamente la **stessa firma**: stesso nome, stessi parametri e stesso tipo di ritorno. Java ti incoraggia a usare l'annotazione **@Override** per indicare esplicitamente che stai ridefinendo un metodo esistente. Questo aiuta il compilatore a verificare la correttezza della ridefinizione e rende il codice più leggibile.

super: Accedere all'Implementazione Originale

- All'interno di un metodo ridefinito, spesso è utile poter accedere all'implementazione originale del metodo nella superclasse. La parola chiave **super** rende possibile questa operazione. Usare super.nomeMetodo() ti permette di estendere il comportamento esistente piuttosto che sostituirlo completamente, seguendo il principio di riutilizzo del codice.
- Esempio veicolo.

Polimorfismo

- Il **polimorfismo** è uno dei concetti più eleganti e potenti della programmazione orientata agli oggetti. Permette di trattare oggetti di diverse classi in modo **uniforme** attraverso un'interfaccia comune.

- In Java, il polimorfismo si manifesta perché un oggetto di una classe derivata può essere usato ovunque sia richiesto un oggetto della classe base, seguendo il **principio di sostituibilità di Liskov**. Ciò significa che se Counter2 è una sottoclasse di Counter, ogni istanza di Counter2 può essere trattata come un'istanza di Counter senza che il codice client debba conoscere la differenza.

- Il polimorfismo non è solo una comodità sintattica; è un meccanismo fondamentale per scrivere codice più **generico, flessibile e facilmente estendibile**. Consente di progettare algoritmi e strutture dati che operano su tipi base, funzionando automaticamente con tutti i tipi derivati. Questo favorisce un codice che rimane valido anche con l'aggiunta di nuove sottoclassi, aderendo all'**Open/Closed Principle**: aperto all'estensione, chiuso alla modifica.

Binding dinamico

- Il **binding dinamico**, o **late binding**, è il meccanismo attraverso il quale Java risolve le chiamate ai metodi a **runtime** (durante l'esecuzione del programma) piuttosto che a compile-time.
- Quando il compilatore incontra una chiamata a un metodo su un oggetto genera un codice che permetterà di prendere questa decisione durante l'esecuzione del programma. Al momento dell'esecuzione, la Java Virtual Machine (JVM) esamina il **tipo effettivo dell'oggetto** (non il tipo dichiarato della variabile) e cerca il metodo appropriato nella gerarchia delle classi, partendo dalla classe più specifica e risalendo verso le classi più generiche fino a trovare l'implementazione corretta.
- Questo processo, chiamato **risoluzione dinamica dei metodi**, introduce un leggero sovraccarico computazionale rispetto al binding statico, ma offre un'enorme flessibilità nel design del software.

Gerarchia di Classi in Java: Object come Radice

In Java, tutte le classi fanno parte di una grande gerarchia che ha al suo vertice la classe **Object**. Questa scelta di design del linguaggio garantisce che tutti gli oggetti in Java condividano un insieme minimo di **funzionalità comuni**:

- **toString()**: Fornisce una rappresentazione testuale dell'oggetto.
- **equals()**: Permette di confrontare due oggetti per determinare se sono logicamente equivalenti.
- **hashCode()**: Calcola un codice hash per l'oggetto, utilizzato in strutture dati basate su hash come HashMap e HashSet.

Questa gerarchia universale ha implicazioni importanti per la programmazione in Java: significa che **qualsiasi oggetto può essere trattato come un Object**, permettendo la creazione di strutture dati e algoritmi completamente generici.

Ereditarietà e Correttezza Logica: Il Principio IS-A

- L'ereditarietà è un meccanismo potente, ma deve essere usato con attenzione, rispettando sempre la relazione **IS-A** tra le classi coinvolte. La relazione IS-A significa che ogni istanza della sottoclasse deve essere concettualmente e logicamente un'istanza della superclasse.

- Quando si progetta una gerarchia di classi, è fondamentale chiedersi se la relazione IS-A sia rispettata in tutti i contesti possibili. Una sottoclasse dovrebbe essere sempre utilizzabile ovunque sia richiesta un'istanza della superclasse, senza causare errori o comportamenti inattesi. Questo principio, noto come **principio di sostituibilità di Liskov**, è fondamentale per garantire che il polimorfismo funzioni correttamente e che il codice rimanga robusto e prevedibile. La violazione di questo principio può portare a gerarchie fragili che richiedono controlli di tipo e casting espliciti, vanificando i benefici dell'ereditarietà e del polimorfismo.

Esercizio

- Definisci una classe Animale che contenga un attributo nome e un metodo parla() che stampi un messaggio generico. Definisci poi una sottoclasse Cane che estende Animale, ridefinisce il metodo parla() e utilizza il costruttore della superclasse per inizializzare il nome. Usa l'annotazione @Override per la ridefinizione del metodo.

Esercizio

- Crea una classe Persona con attributi nome e cognome, e un metodo print(). Crea una classe Studente che estende Persona, aggiunge l’attributo matricola, e ridefinisce toString() e print() per includere la matricola. Usa super() per inizializzare la parte ereditata e super.toString() per estendere la rappresentazione testuale.

Esercizio

- Crea un array di oggetti Persona che contiene anche istanze di Studente. Itera sull’array e chiama il metodo print() su ogni elemento. Dimostra come il polimorfismo consenta di eseguire automaticamente il metodo corretto per ciascun oggetto.