Teoria: Ereditarietà e Polimorfismo in Java

Polimorfismo

Il **polimorfismo** è uno dei quattro pilastri della programmazione orientata agli oggetti (insieme a incapsulamento, ereditarietà e astrazione). Significa letteralmente "molte forme" e si riferisce alla capacità di oggetti di classi diverse di rispondere allo stesso messaggio in modi diversi.

Tipi di Polimorfismo in Java

Tipo	Quando viene risolto	Meccanismi
Polimorfismo Statico	Compile-time	• Overloading (sovraccarico) • Polimorfismo dei costruttori
Polimorfismo Dinamico	Runtime	• Overriding (sovrascrittura)

1. Polimorfismo dei Costruttori

È una forma particolare di **overloading** applicata ai costruttori. Permette di definire più costruttori all'interno della stessa classe, ognuno con una lista di parametri diversa.

Caratteristiche

- **Stesso nome** (nome della classe)
- **Parametri diversi** (numero, tipo o ordine)
- Permette di inizializzare gli oggetti in modi differenti
- Offre **flessibilità** nella costruzione degli oggetti

Esempio:

```
Person p1 = new Person();                                          // 0 parametri
Person p2 = new Person("Mario", "Rossi");                          // 2 parametri
Person p3 = new Person("Luigi", "Verdi", "01-01-1990", "M");       // 4 parametri
```

Vantaggi

- **Flessibilità**: creare oggetti con diverse quantità di informazioni
- **Leggibilità**: il codice è più chiaro e intuitivo
- **Convenienza**: non servono valori predefiniti quando non si hanno tutti i dati
- **Manutenibilità**: centralizza la logica di inizializzazione

2. Overloading (Sovraccarico)

È la possibilità di definire **più metodi con lo stesso nome** all'interno della stessa classe, purché abbiano una **firma diversa**.

Firma del Metodo

La firma è determinata da:

Elemento	Include nella firma?
Nome del metodo	SÌ
Numero di parametri	SÌ
Tipo dei parametri	SÌ
Ordine dei parametri	SÌ
Tipo di ritorno	**NO**

Caratteristiche

- Avviene nella **stessa classe**
- Risoluzione a **tempo di compilazione** (polimorfismo statico)
- Il compilatore sceglie quale metodo invocare in base agli argomenti passati
- Permette di usare lo stesso nome per operazioni concettualmente simili ma che operano su dati diversi

3. Overriding (Sovrascrittura)

È il meccanismo che permette a una **sottoclasse** di fornire una propria implementazione di un metodo già definito nella **superclasse**.

Requisiti per l'Overriding

1. Il metodo deve avere la **stessa firma**:
   - Stesso nome
   - Stessi parametri (numero, tipo, ordine)
   - Stesso tipo di ritorno (o un sottotipo - covariant return)
2. Il metodo deve essere **accessibile** (non può essere `private`)
3. Il modificatore di accesso **non può essere più restrittivo** nella sottoclasse
4. Il metodo **non può essere `final`** nella superclasse
5. Il metodo **non può essere `static`** (i metodi statici vengono nascosti, non sovrascritti)

Caratteristiche

- Avviene nella **gerarchia di classi** (superclasse → sottoclasse)
- Risoluzione a **tempo di esecuzione** (polimorfismo dinamico)

- Permette alle sottoclassi di **specializzare** il comportamento ereditato
- Mantiene la stessa **interfaccia**
- Fondamentale per il **polimorfismo** e il **principio di sostituzione di Liskov**
- Si usa l'annotazione `@Override` per indicare l'intenzione

Overloading vs Overriding

Aspetto	Overloading	Overriding
Dove avviene	Stessa classe	Gerarchia di classi
Firma del metodo	Parametri DIVERSI	Firma IDENTICA
Quando si risolve	Compile-time (statico)	Runtime (dinamico)
Tipo di polimorfismo	Statico	Dinamico

Principio di Sostituzione di Liskov (LSP)

L'overriding è fondamentale per rispettare questo principio che afferma:

> **"Gli oggetti di una sottoclasse devono poter sostituire gli oggetti della superclasse senza alterare la correttezza del programma"**

Esempio Pratico

```
public void printPersonInfo(Person person) {
    System.out.println(person.toString());
    // Funziona con qualsiasi sottoclasse di Person!
}

// Tutti questi funzionano perfettamente!
printPersonInfo(new Person());
printPersonInfo(new Student());          // Student IS-A Person
printPersonInfo(new Employee());         // Employee IS-A Person
printPersonInfo(new ForeignEmployee());  // ForeignEmployee IS-A Person
```

Ereditarietà

L'ereditarietà è un meccanismo fondamentale della programmazione orientata agli oggetti (OOP) che permette a una classe (classe figlia o derivata) di ereditare proprietà e metodi da un'altra classe (classe padre o superclasse).

Sintassi Base

```
public class ClasseFiglia extends ClassePadre {
    // proprietà e metodi aggiuntivi
}
```

La Parola Chiave `extends`

La parola chiave `extends` stabilisce una relazione di ereditarietà tra due classi: - La classe figlia **eredita** tutti i membri (proprietà e metodi) non privati della classe padre - In Java, una classe può estendere **solo una classe** (ereditarietà singola)

```
public class Student extends Person {
    protected int year;
    protected String section;
}
```

Visibilità delle Proprietà: `protected`

Modificatori di Accesso

Modificatore	Accesso dalla classe	Accesso dalle sottoclassi	Accesso dall'esterno
`private`	SÌ	NO	NO
`protected`	SÌ	SÌ	Solo stesso package
`public`	SÌ	SÌ	SÌ

Quando Usare `protected` Le proprietà `protected` sono ideali per l'ereditarietà perché: - Permettono alle classi derivate di accedere direttamente alle proprietà - Mantengono l'incapsulamento rispetto a classi esterne - Sono un compromesso tra `private` (troppo restrittivo) e `public` (troppo permissivo)

```
public class Employee extends Person {
    protected double salary;        // accessibile nelle sottoclassi
    protected String department;
}
```

La Parola Chiave `super`

`super` è un riferimento alla classe padre e ha due utilizzi principali:

1. Chiamare il Costruttore della Classe Padre

```
public Student(String name, String surname, String dateOfBirth, String gender, int year, Str
    super(name, surname, dateOfBirth, gender);  // chiama il costruttore di Person
```

```
    this.year = year;
    this.section = section;
}
```

Regole importanti: - La chiamata a `super()` deve essere la **prima istruzione** nel costruttore - Se non viene chiamato esplicitamente, Java chiama automaticamente `super()` (costruttore vuoto) - Se la classe padre non ha un costruttore vuoto, è **obbligatorio** chiamare `super(parametri)`

2. Chiamare Metodi della Classe Padre

```
@Override
public String toString() {
    return super.toString() + " Year: " + year + " Section: " + section;
}
```

Alternative a `super`

Usare i Setter invece di super() nei Costruttori

Se le proprietà della classe padre sono `private`, puoi usare i setter pubblici:

```
public Student(String name, String surname, String dateOfBirth, String gender, int year, Str
    // Invece di super(name, surname, dateOfBirth, gender);
    setName(name);
    setSurname(surname);
    setDateOfBirth(dateOfBirth);
    setGender(gender);
    this.year = year;
    this.section = section;
}
```

Chiamare Direttamente i Getter invece di super.toString()

```
@Override
public String toString() {
    // Invece di super.toString()
    return getName() + " " + getSurname() + " " + getDateOfBirth() + " " + getGender() +
           " Year: " + year + " Section: " + section;
}
```

Svantaggi: - Duplicazione del codice - Se il `toString()` della classe padre cambia, devi aggiornare tutte le classi figlie - Viola il principio DRY (Don't Repeat Yourself)

Quando usare questo approccio: - Raramente! È generalmente meglio usare `super.toString()` - Solo se hai bisogno di un formato completamente diverso

Gerarchia di Ereditarietà

Le classi possono formare una catena di ereditarietà:

```
Object (classe base di Java)
  ↓
Person
  ↓
Employee
  ↓
ForeignEmployee
```

Best Practices

3. Getter e Setter anche per Proprietà `protected`

Anche se le proprietà sono `protected`, è buona pratica fornire getter e setter:

```
public int getYear() { return year; }
public void setYear(int year) { this.year = year; }
```

Vantaggi: - Permette di aggiungere validazione in futuro - Mantiene l'incapsulamento - Facilita il debugging - Fornisce un'interfaccia pubblica stabile

---

4. Preferire la Composizione all'Ereditarietà (quando appropriato)

Non tutto deve essere risolto con l'ereditarietà. A volte la composizione è più appropriata.

Scenario	Usare Ereditarietà	Usare Composizione
Tipo di relazione	"IS-A" (Student IS-A Person)	"HAS-A" (Car HAS-A Engine)
Comportamento	Condivisione di comportamento comune	Cambiare comportamento a runtime
Struttura	Gerarchia naturale e stabile	Evitare gerarchie profonde

Annotazione `@Override`

L'annotazione `@Override` indica che un metodo sovrascrive un metodo della classe padre:

```
@Override
public String toString() {
```

```
    return super.toString() + " informazioniAggiuntive";
}
```

Vantaggi: - Il compilatore verifica che il metodo esista nella classe padre - Previene errori di battitura nel nome del metodo - Documenta l'intenzione del programmatore - Migliora la leggibilità del codice - Facilita il refactoring

Riepilogo Concetti Chiave

Polimorfismo

#	Concetto	Descrizione
1	**Polimorfismo dei costruttori**	Più costruttori nella stessa classe con parametri diversi
2	**Overloading**	Più metodi con lo stesso nome ma firma diversa, risoluzione statica
3	**Overriding**	Ridefinizione di un metodo della superclasse, risoluzione dinamica
4	**Distinzione chiave**	Overloading = stessa classe, parametri diversi; Overriding = gerarchia, firma identica

Ereditarietà

#	Concetto	Descrizione
5	`extends`	Parola chiave per stabilire l'ereditarietà
6	`protected`	Modificatore di visibilità ideale per proprietà ereditate
7	`super()`	Chiama il costruttore della classe padre
8	`super.metodo()`	Chiama un metodo della classe padre
9	**Gerarchia di tipi**	Un oggetto appartiene a tutti i tipi della sua catena di ereditarietà
10	`@Override`	Annota i metodi che sovrascrivono metodi del padre
11	**LSP**	Principio di Sostituzione di Liskov - le sottoclassi devono poter sostituire le superclassi
12	**Best practice**	Usare `super` per evitare duplicazione di codice (DRY)

Relazioni

Tipo	Simbolo	Esempio	Quando usare
IS-A	Ereditarietà	Student IS-A Person	Relazione di ereditarietà
HAS-A	Composizione	Car HAS-A Engine	Relazione di composizione

CONCORDANZA DEI TIPI ED EREDITARIETÀ

Il principio

Person (classe base/superclasse) ed **Employee** (classe derivata/sottoclasse) hanno una relazione di ereditarietà, dove Employee estende Person.

Regole di compatibilità

Assegnazione	Compatibile?	Spiegazione
`Employee e = new Person()`	**NO**	Person è più generica e non ha tutte le caratteristiche di Employee
`Person p = new Employee()`	**SÌ**	Employee "è un" (IS-A) Person, ha tutte le caratteristiche di Person + altre

Il principio teorico

Questo è il **principio di sostituzione** (parte del polimorfismo):

> **Un oggetto di una sottoclasse può sempre essere trattato come un oggetto della superclasse, ma non viceversa.**

In termini formali:

```
Se B extends A, allora:
  A riferimento = new B();     // VALIDO
  B riferimento = new A();     // NON VALIDO (senza casting)
```

La sottoclasse eredita tutto dalla superclasse e può aggiungere funzionalità, quindi è compatibile verso l'alto nella gerarchia.

Questo principio garantisce la coerenza del sistema di tipi e permette il polimorfismo, consentendo di scrivere codice generico che opera su superclassi ma funziona con qualsiasi sottoclasse.

POLIMORFISMO DI OGGETTI

Scenario di partenza

Abbiamo creato un oggetto di tipo Teacher (nuovo Teacher()) e lo abbiamo assegnato a una variabile di tipo Person (p).

```
Person p = new Teacher();
```

Domande e Risposte

#	Domanda	Risposta	Spiegazione
1	L'oggetto in p è un insegnante?	**SÌ**	Abbiamo creato un oggetto di tipo Teacher, quindi p fa riferimento a un Teacher
2	L'oggetto in p è una persona?	**SÌ**	Teacher è una sottoclasse di Person (Teacher IS-A Person)
3	L'oggetto in p è uno studente?	**NO**	L'oggetto è un Teacher, non uno Student (sono sottoclassi distinte)
4	A che tipo appartiene l'oggetto in p?	**Person e Teacher**	L'oggetto appartiene a più tipi lungo la gerarchia di ereditarietà

Concetto chiave

Il **tipo dichiarato** della variabile (Person) determina quali metodi possiamo chiamare a tempo di compilazione, ma il **tipo effettivo** dell'oggetto (Teacher) determina quale implementazione viene eseguita a tempo di esecuzione.

Questo è il cuore del polimorfismo dinamico.

---

hashCode()

Il metodo **hashCode()** restituisce un valore intero che rappresenta un codice hash dell'oggetto, utilizzato principalmente per migliorare l'efficienza delle strutture dati basate su hash come HashMap, HashSet e Hashtable.

Il metodo hashCode() è definito nella classe Object ed è progettato per generare un identificatore numerico (hash) che rappresenti l'oggetto in modo rapido e compatto. Questo hash viene utilizzato dalle collezioni basate su hash per organizzare e recuperare gli oggetti in modo efficiente.

Il principio fondamentale è che il codice hash permette di dividere gli oggetti in "bucket" o categorie, riducendo drasticamente il numero di confronti necessari per trovare un elemento. Invece di confrontare un oggetto con tutti gli altri nella collezione, si confronta solo con quelli che hanno lo stesso hash.

Esiste una **relazione fondamentale** tra hashCode() ed equals(): se due oggetti sono uguali secondo equals(), devono necessariamente avere lo stesso hashCode(). Questa è chiamata "contratto hashCode-equals". Tuttavia, il contrario non è necessariamente vero: due oggetti con lo stesso hashCode possono essere diversi secondo equals() (questo è chiamato "collisione").

Quando si ridefinisce equals() in una classe, è obbligatorio ridefinire anche hashCode() per mantenere questo contratto. Se non lo si fa, gli oggetti non funzioneranno correttamente nelle collezioni basate su hash: potrebbero essere considerati uguali da equals() ma finire in bucket diversi, rendendo impossibile trovarli.

Un buon algoritmo di hashCode() dovrebbe distribuire gli oggetti uniformemente tra i vari valori hash possibili per minimizzare le collisioni. Tipicamente si calcolano gli hash combinando i valori hash dei campi significativi dell'oggetto, quelli usati anche nel metodo equals().

Il hashCode() non identifica univocamente un oggetto (non è una chiave primaria), ma è uno strumento di ottimizzazione che permette alle strutture dati di organizzare e accedere agli oggetti in tempo praticamente costante anziché lineare.
