CORSO JAVA AESYS
Parte Prima

2022

www.aesystech.it

Ereditarietà ed interfacce
prima parte

- Ereditarietà
- La parola chiave extends
- La classe Object
- Rapporto ereditarietà-incapsulamento
- Ereditarietà e costruttori
- La parola chiave super

Ereditarietà

Chiunque può programmare, basta conoscere come si usa l'if e il for, e si può fare tutto! In teoria è così, in pratica bisogna avere delle regole e le idee chiare prima di mettersi a programmare a testa bassa. Quello che fa la differenza tra un bravo smanettone e un bravo sviluppatore è la qualità del suo lavoro. Uno sviluppatore dovrebbe saper analizzare e progettare la sua soluzione, creando codice semplice e manutenibile.

Come tutti i paradigmi che caratterizzano l'Object Orientation, anche il concetto di ereditarietà è ispirato a qualcosa che esiste nella realtà. Il termine è inteso in senso darwiniano. Nel mondo reale noi classifichiamo tutto con classi e sottoclassi. Per esempio un cane è un animale, un aereo è un veicolo, la chitarra è uno strumento musicale. In Java l'ereditarietà è la caratteristica che mette in relazione di estensibilità più classi che hanno caratteristiche comuni. Per esempio la classe Cane estenderà la classe Animale. Il risultato immediato è la possibilità di ereditare codice già scritto, e quindi di gestire insiemi di classi collettivamente giacché accomunate da alcune caratteristiche. Le regole per utilizzare correttamente l'ereditarietà sono semplici e chiare. Ma sebbene l'ereditarietà sia un argomento agevole da comprendere, non è sempre utilizzata in maniera corretta.

La parola chiave extends 1/2

La parola chiave extends ci permetterà di implementare l'ereditarietà. Consideriamo le seguenti classi che non incapsuliamo per semplicità:

```
public class Libro {     public String titolo;     public String autore;
public String editore;     public int numeroPagine;     public int prezzo;
   //... }
public class LibroSuJava {     public String titolo;     public String autore;
public String editore;     public int numeroPagine;
public int prezzo;     public final String ARGOMENTO_TRATTATO = "Java";
//... }
```

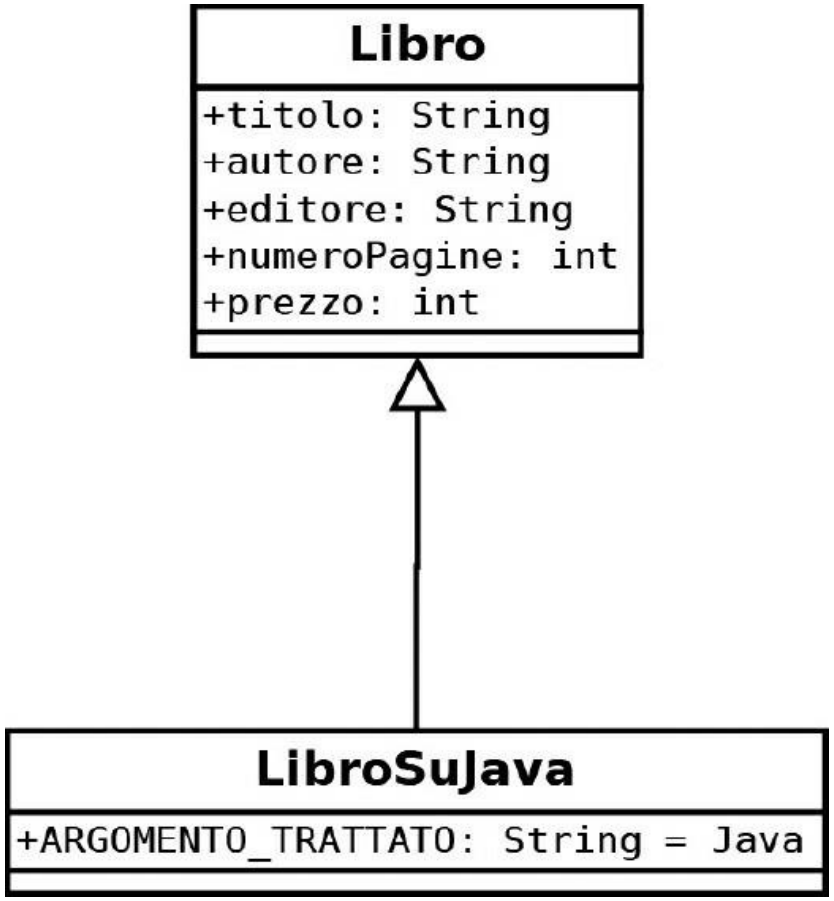
Notiamo che le classi Libro e LibroSuJava rappresentano due concetti in relazione tra loro e quindi dichiarano campi in comune. L'ereditarietà consentirà di mettere in relazione di estensione le due classi con la seguente sintassi:

```
public class LibroSuJava extends Libro {     public final String ARGOMENTO_TRATTATO = "Java";
   //... }
```

La parola chiave extends 2/2

In questo modo la classe LibroSuJava erediterà tutti i campi pubblici della classe che estende. Quindi nella classe LibroSuJava sono presenti anche le variabili pubbliche numeroPagine, prezzo, titolo, autore ed editore definite nella classe Libro, anche se non sono state codificate esplicitamente. La costante ARGOMENTO_TRATTATO invece, appartiene solo alla classe LibroSuJava. In particolare diremo che LibroSuJava è **sottoclasse** di Libro, e Libro è **superclasse** di LibroSuJava. Il primo vantaggio è evidente, in questo esempio abbiamo riusato del codice evitando un "copia-incolla", ma l'ereditarietà non si riduce al solo scrivere meno righe.

La notazione UML che definisce la relazione di estensione, è una freccia che va dalla sottoclasse alla superclasse. Nella figura è rappresentato un diagramma delle classi che mostra la relazione di ereditarietà tra le due classi



Il modificatore final

Abbiamo già visto come questo modificatore venga usato per dichiarare le costanti in Java. Ma final può essere utilizzato non solo con le variabili, ma anche con classi e metodi per influire sulle regole dell'ereditarietà. Potremmo tradurre il termine final proprio con "finale", nel senso di "non modificabile". Infatti:

- una variabile (sia d'istanza che locale) dichiarata final diviene una costante;
- un metodo dichiarato final non può essere riscritto in una sottoclasse (non è possibile applicare l'override). Questo significa che si eredita così come è stato dichiarato nella superclasse;
- una classe dichiarata final non può essere estesa.

Per esempio le classi String e la classe Math del package java.lang sono esempi di classi dichiarate final. Queste classi quindi non si possono estendere perché otterremmo un errore in compilazione.
Il modificatore final si può utilizzare anche per variabili locali e parametri locali di metodi. In tali casi, i valori di tali variabili non saranno modificabili localmente.
Dichiarare final i parametri di un metodo è considerato da alcuni come una buona pratica di programmazione. Il modificatore in questo caso enfatizza il fatto che i parametri non debbano essere modificati. Infatti essi vengono valorizzati dall'esterno, e cambiare loro il valore all'interno del metodo significa snaturare la loro definizione.

La classe Object

Come abbiamo già osservato più volte, il compilatore Java inserisce spesso, nel bytecode compilato, alcune istruzioni che il programmatore non ha inserito. Questo al fine di agevolare lo sviluppatore sia nell'apprendimento che nella codifica. Abbiamo inoltre già asserito più volte che la programmazione ad oggetti si ispira a concetti reali. Tutta la libreria standard di Java è stata pensata ed organizzata in maniera tale da soddisfare la teoria degli oggetti. Siccome la realtà è composta da oggetti (tutto può considerarsi un oggetto, sia elementi concretamente esistenti, sia concetti astratti), nella libreria standard di Java esiste una classe chiamata Object che astrae il concetto di oggetto generico. Esso appartiene al package java.lang ed è di fatto la superclasse di ogni classe. È in cima alla gerarchia delle classi e quindi tutte le classi ereditano i membri di Object (è possibile verificare quest'affermazione leggendo la descrizione dei metodi di una qualsiasi classe nella documentazione). Se definiamo una classe che non estende altre classi, essa automaticamente estenderà Object. Ciò significa che se scrivessimo:

```
public class Arte {//... }
```

il compilatore tradurrebbe questa classe nel seguente modo:

```
public class Arte extends Object {//... }
```

Nel mondo reale tutto è un oggetto, quindi in Java tutte le classi estenderanno Object.

Rapporto ereditarietà-incapsulamento

Dal momento che l'incapsulamento si può praticamente considerare obbligatorio e l'ereditarietà un prezioso strumento di sviluppo, bisognerà chiedersi che cosa provocherà l'utilizzo combinato di entrambi i paradigmi. Ovvero: che cosa erediteremo da una classe incapsulata? Abbiamo già affermato alla fine del capitolo precedente che estendere una classe significa ereditarne i membri non privati. Consideriamo quindi una classe Ricorrenza ottenuta specializzando la classe Data definita nel capitolo precedente come esempio di incapsulamento. È escluso che la classe Ricorrenza possa accedere alle variabili giorno, mese e anno direttamente, giacché queste non saranno ereditate. Infatti tali variabili erano state dichiarate private all'interno della superclasse Data. Ma essendo tutti i metodi set e get dichiarati pubblici nella superclasse, verranno ereditati e quindi resi utilizzabili nella sottoclasse. Quindi anche se la classe Ricorrenza non possiede esplicitamente le variabili private di Data, è come se le possedesse, visto che può comunque usufruirne tramite l'incapsulamento.
Un errore comune è quello di pensare che per poter utilizzare una certa variabile in una sottoclasse, sia necessario dichiararla protected, quando è più che sufficiente avere a disposizione i metodi mutator (set) ed accessor (get) nelle sottoclassi.
Attenzione che, inoltre, dichiarare protetta una variabile d'istanza significa renderla pubblica a tutte le classi dello stesso package. Questo significa che la variabile protected è accessibile direttamente a tutte le classi appartenenti allo stesso package, e tranne in rari casi non è questo il livello di incapsulamento desiderato

Quando utilizzare l'ereditarietà

Quando si parla di ereditarietà si è spesso convinti che per implementarla basti avere un paio di classi che dichiarino campi in comune. Ciò potrebbe essere interpretato solo come un indizio di ereditarietà. Il test decisivo deve però essere effettuato mediante la cosiddetta "is a" relationship (in italiano la relazione "è un").
Per poter fare un uso corretto dell'ereditarietà, il programmatore deve porsi solo una semplice domanda: "un oggetto della candidata sottoclasse è un oggetto della candidata superclasse?". Se la risposta alla domanda è positiva, l'ereditarietà si deve applicare, se la risposta è negativa, l'ereditarietà non si deve applicare.Effettivamente, se l'applicazione dell'ereditarietà dipendesse solamente dai campi in comune tra due classi, potremmo trovare relazioni d'estensione tra classi quali Triangolo e Rettangolo. Per esempio:

```
public class Triangolo {     public final int
NUMERO_LATI = 3;     public float lunghezzaLatoUno
public float lunghezzaLatoDue;     public float
lunghezzaLatoTre;
   //...
} public class Trapezio extends Triangolo {




   public final int NUMERO_LATI = 4;     public float
lunghezzaLatoQuattro;
   //... }
```

Ma un trapezio non è un triangolo! E per la relazione "is a" quest'estensione non è valida. Il problema è che la costruzione di un programma avviene passo dopo passo. Se mettiamo in relazione di ereditarietà in maniera errata due classi, in un primo momento risparmieremmo del codice, ma i problemi sorgerebbero dopo, quando tenteremo di sfruttare un trapezio così come sfruttiamo un triangolo. La relazione "è un" ci mette al sicuro da questi inconvenienti, ed è molto semplice da testare

Ereditarietà e costruttori 1/2

L'ereditarietà non è applicabile ai costruttori. Anche quando sono dichiarati pubblici, i costruttori non sono ereditati per un motivo molto semplice: il loro nome. Ricordiamo che un costruttore è stato definito nel capitolo 2 come metodo speciale, in quanto possiede le seguenti proprietà:

1. ha lo stesso nome della classe cui appartiene;
2. non ha tipo di ritorno;
3. è chiamato automaticamente (e solamente) ogni volta che viene istanziato un oggetto della classe cui appartiene, relativamente a quell'oggetto;
4. è presente in ogni classe.

Relativamente all'ultimo punto abbiamo anche definito il "costruttore di default" come il costruttore che è introdotto in una classe dal compilatore al momento della compilazione, nel caso il programmatore non gliene abbia fornito uno in modo esplicito. Abbiamo anche affermato che solitamente un costruttore è utilizzato per inizializzare le variabili degli oggetti al momento dell'istanza. Per quanto detto, se per esempio la classe LibroSuJava ereditasse dalla classe Libro un costruttore, erediterebbe un costruttore che si chiama proprio Libro(). Un costruttore che si chiama Libro() in una classe che si chiama LibroSuJava non potrà mai essere chiamato! Infatti per istanziare un oggetto dalla classe LibroSuJava, bisogna obbligatoriamente chiamare un costruttore chiamato LibroSuJava(). Per esempio, dato il seguente codice:

LibroSuJava libroSuJava = new LibroSuJava(); (segue slide seguente)

Ereditarietà e costruttori 2/2

Se al suo posto scrivessimo:
LibroSuJava libroSuJava = new Libro();

istanzieremmo un oggetto della classe Libro, e non un oggetto della classe LibroSuJava (ottenendo anche un errore in compilazione).
Il fatto che i costruttori non siano ereditati dalle sottoclassi è assolutamente in linea con la sintassi del linguaggio, ma contemporaneamente è in contraddizione con i principi della programmazione ad oggetti. In particolare sembra violata la regola dell'astrazione. Infatti, nel momento in cui lo sviluppatore ha deciso di implementare il meccanismo dell'ereditarietà, ha dovuto testarne la validità. mediante la cosiddetta relazione "is a". Alla domanda: "un oggetto istanziato dalla candidata sottoclasse può considerarsi anche un oggetto della candidata superclasse?" ha infatti risposto affermativamente.
Un libro su Java, essendo anche un libro, deve avere tutte le caratteristiche di un libro. In particolare deve riutilizzarne anche il costruttore. Non potendolo ereditare però, l'astrazione sembra violata. Invece è proprio in una situazione del genere che Java dimostra la sua coerenza. Aggiungiamo infatti un'altra proprietà alla definizione di costruttore:
5. un qualsiasi costruttore (anche quello di default), come prima istruzione, invoca sempre un costruttore della superclasse

La parola chiave super

abbiamo definito la parola chiave this come “reference implicito all’oggetto corrente”. Possiamo definire la parola chiave super come “reference implicito all’intersezione tra l’oggetto corrente e la sua superclasse”. Questo reference ci permette di accedere ai componenti della superclasse ed in particolare al suo costruttore. La parola chiave super è strettamente legata al concetto di costruttore. In ogni costruttore, infatti, è sempre presente una chiamata al costruttore della superclasse, tramite una sintassi speciale che sfrutta il reference super. Per esempio nella classe LibroSuJava, il costruttore verrà modificato dal compilatore nel seguente modo:

public class LibroSuJava extends Libro { public LibroSuJava() { **super();** //istruzione implicita se non fornita esplicitamente.

```
        System.out.println("Costruito un Libro su Java!");
    }
}
```

Ecco come il costruttore della classe Libro viene invocato dal costruttore della classe LibroSuJava. Possiamo esplicitare la chiamata a super(), ma se non lo facciamo il compilatore considererà questa istruzione implicita. La chiamata ad un costruttore della superclasse è quindi inevitabile!

Ereditarietà e modificatori

Riassumiamo la situazione per quanto riguarda i modificatori d'accesso.
Il modificatore private può essere applicato a variabili e metodi, e impedisce a tali membri di essere ereditati in una sottoclasse.
Il modificatore protected, che pure può essere applicato solo a variabili e metodi, permette che i membri siano ereditati.
Quando invece non specifichiamo modificatori davanti a variabili d'istanza o metodi, questi saranno ereditati solo nelle sottoclassi appartenenti allo stesso package. Invece, se non specifichiamo modificatori quando dichiariamo una classe, essa sarà visibile (e quindi estendibile) solo da classi appartenenti allo stesso package.
Abbiamo già parlato del modificatore final e del suo rapporto con l'ereditarietà. Riassumendo una classe dichiarata final non può essere estesa, un metodo dichiarato final non può essere riscritto (override) in una sottoclasse, una variabile final diventa una costante. È però possibile ereditare una costante (a meno che non sia dichiarata private).
Il modificatore static può essere utilizzato con il comando import, per importare membri statici di una classe, ma soprattutto può essere applicato a variabili, metodi ed anche inizializzatori. Il comando import non ha nulla a che fare con l'ereditarietà. Invece variabili e metodi statici, vengono ereditati, ma siccome appartengono alla classe e non agli oggetti, si tratta di un'ereditarietà tra classi.