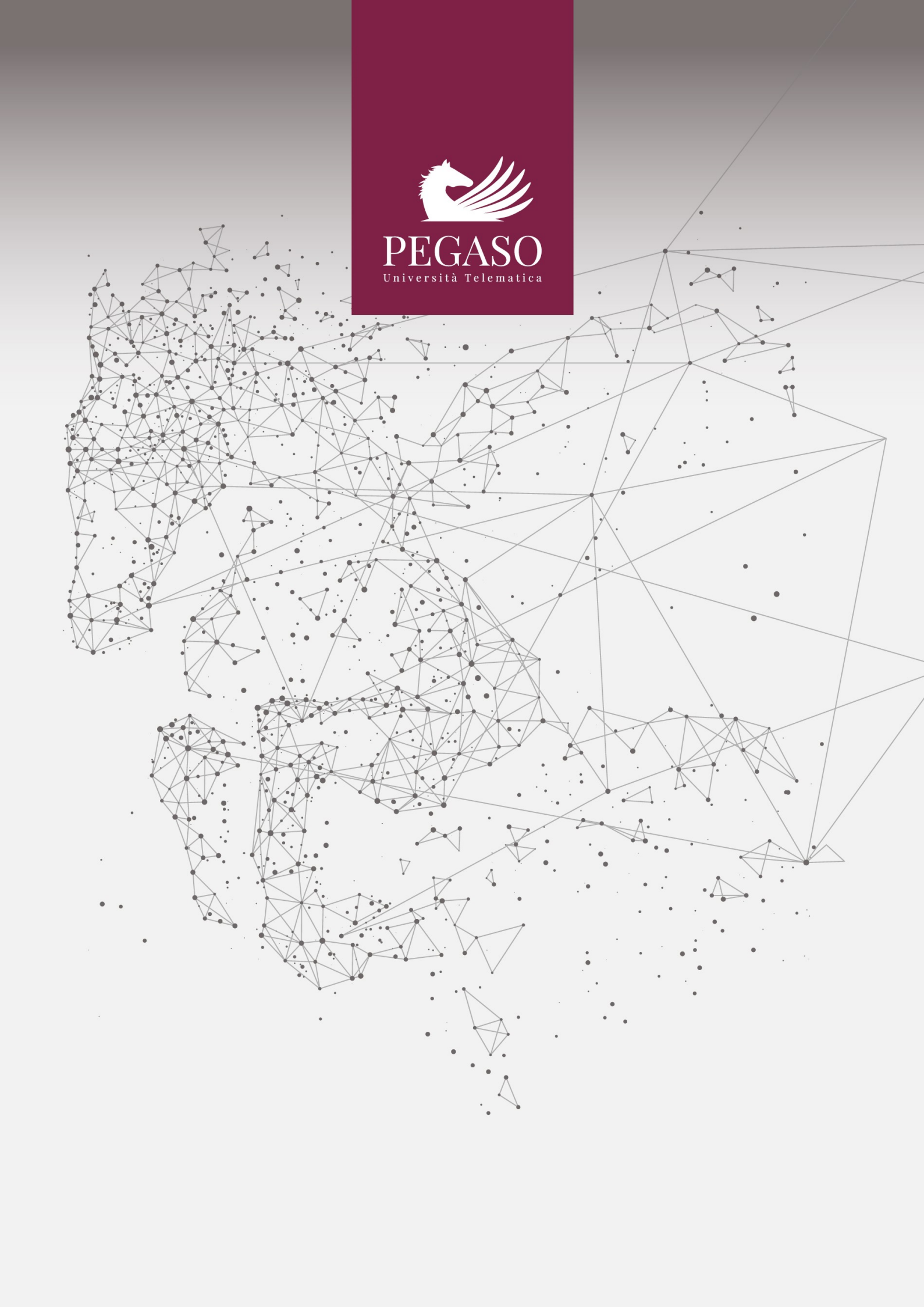
PEGASO
Università Telematica

Indice

1. Package e protezione

I package in Java sono un meccanismo che consente di raggruppare le classi. La Libreria Standard di Java (Java API) è organizzata in packages:

- String è una classe del package java.lang
- Scanner è una classe del package java.util

Un package riunisce classi logicamente correlate tra loro, ad esempio:

- Il package java.lang riunisce classi fondamentali del linguaggio Java (String, Math,)
- Il package java.util riunisce classi di frequente utilizzo (Scanner, Random, Timer,)
- I packages java.awt e java.swing riuniscono classi per costruire interfacce grafiche

I package possono tornare utili quando il programma inizia a diventare complesso e un raggruppamento in packages può consentire di fare ordine e dare un'organizzazione logica alle molte classi. Un package in Java permette di raggruppare in un'unica entità complessa classi Java logicamente correlate. Fisicamente il package non è altro che una cartella del nostro sistema operativo, ma non tutte le cartelle sono package. Per eleggere una cartella a package, una classe Java deve dichiarare nel suo codice la sua appartenenza a quel determinato package e inoltre deve risiedere fisicamente all'interno di essa. Per esempio se volessimo far appartenere la classe Auto a un package veicoliamotore, dovremmo semplicemente inserire come prima istruzione del nostro file sorgente la seguente dichiarazione di package:

```
package veicoliamotore;
```

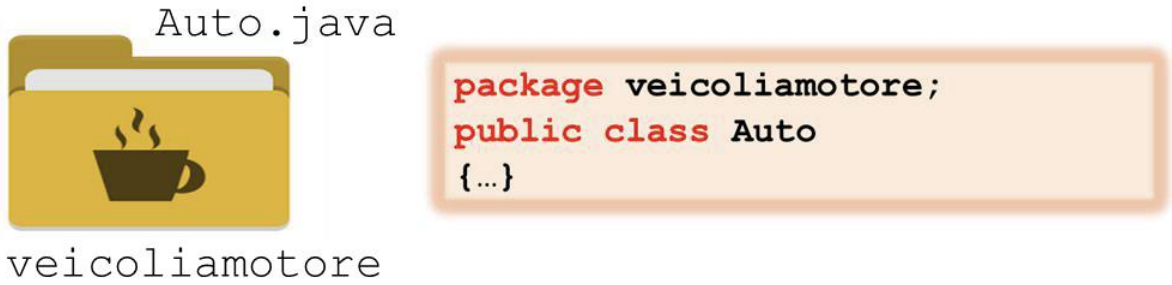

Fig. 12: *Uso dei package in Java*

Questo comporterebbe che il file compilato .class dovrà risiedere in una cartella chiamata proprio «veicoliamotore».

Se invece volessimo creare un albero di package più strutturato possiamo tranquillamente usare la sintassi sfruttando l'operatore dot (che anche in questo caso significa appartenenza). In sostanza di forma una struttura gerarchica analoga a quella delle directory in un file system.

package asset.veicoli.veicoliamotore;

In questo caso il file compilato dovrebbe risiedere all'interno di una cartella «veicoliamotore» a sua volta contenuta in una cartella «veicoli» a sua volta contenuta in una cartella «asset».

Il problema con i package sono poi le regole di visibilità delle classi contenute in esse. Come fa una classe che si trova in una cartella a «vedere» una classe che si trova in un'altra cartella? Si usa la direttiva import.

Sicuramente la classe che vuole usare la classe Auto (supponiamo si chiami TestAuto) se non si trova nello stesso package asset.veicoli.veicoli_a_motore deve importare la classe Auto con il comando import che va dichiarato dopo la dichiarazione di package ma prima della dichiarazione della classe , come nell'esempio in Fig. 2

```
package asset.veicoli.veicoliamotore;
import asset.veicoli.veicoliammotore.Auto;

public class TestAuto{

public static void main (String args[]) {
Auto auto = new Auto();

…}
}
```

Fig. 2: *Visibilità fra classi in diversi package*

Il modificatore è una parola chiave (es. public, private, final o static) capace di cambiare il significato di un componente di un'applicazione Java, fra cui classi (e in generale tutti i tipi Java), metodi, variabili d'istanza, variabili locali, attributi di metodi e costruttori, e altri.

Un modificatore sta ad un componente di un'applicazione Java come un aggettivo sta ad un sostantivo nel linguaggio umano. Vedremo in particolare i modificatori di accesso e il modificatore static.

2. Modificatori di accesso: private, public, protected

I modificatori di accesso specificano da quali classi è possibile utilizzare una classe, una variabile di istanza o un metodo.

I modificatori public e private vengono meglio definiti come modificatori di accesso in quanto regolano essenzialmente la visibilità e l'accesso ad un componente Java.

Esiste poi un altro modificatore di accesso protected.

Il modificatore public può essere applicato sia ad un membro (variabile di istanza o metodo) di una classe sia ad una classe stessa.

Un membro dichiarato pubblico sarà accessibile da una qualsiasi classe situata in qualsiasi package.

Per l'incapsulamento le variabili di istanza non dovrebbero mai essere dichiarate pubbliche. Una classe dichiarata pubblica sarà anch'essa visibile da un qualsiasi package (a meno di dichiarazioni di moduli).

Il modificatore private limita la visibilità di una variabile di istanza o di un metodo all'interno della classe in cui sono definiti.

Il modificatore private restringe la visibilità di un membro di una classe alla classe stessa.

Esso rappresenta la chiave dell'incapsulamento per le variabili di istanza.

Si usano anche metodi privati che fanno parte dell'implementazione del comportamento e che sono invocati da altri metodi della stessa classe.

Infine diversamente dal modificatore public, il modificatore private non è applicabile alla dichiarazione di una classe.

Il modificatore d'accesso protected è legato anche al concetto di ereditarietà (che vedremo dopo). Riguarda infatti l'accessibilità dei membri nelle sottoclassi.

Una sottoclasse è una classe che estende un'altra classe ereditandone le caratteristiche ed estendendo e al contempo specializzandone il comportamento, ad esempio la classe Dipendente e la classe Programmatore (il programmatore è un dipendente che svolge delle mansioni particolari).

I metodi pubblici sono ereditati dalla sottoclasse e quindi possono essere usati senza limiti. Lo stesso dicasi per i metodi protetti.

Di conseguenza il modificatore protected definisce per un membro di una classe il grado più accessibile dopo quello definito da public.

Un membro protetto infatti sarà accessibile all'interno delle stesso package, e verrà anche ereditato in tutte le sottoclassi della classe in cui è definito, anche se non appartenenti allo stesso package di dichiarazione. Infatti i membri dichiarati come protected saranno ereditati come fossero pubblici anche nelle sottoclasse esterne al package.

Diversamente dal modificatore public il modificatore protected non è applicabile alla dichiarazione di una classe.

È possibile non anteporre alcun modificatore di accesso ad un membro (variabile di istanza o metodo) di una classe o relativamente alla classe stessa. Se non si antepone alcun modificatore di accesso ad un membro di una classe esso sarà accessibile solo da classe appartenenti al package dove è definito.

3. Modificatore static

Altro modificatore connesso alla visibilità è static. Questo modificatore si applica ai metodi e attributi (ovvero variabili di istanza) di una classe e sta ad indicare la proprietà di essere condiviso da tutte le istanze della classe in cui è dichiarato.

Quindi le regole dell'incapsulamento quando impostiamo questo modificatore sono alterate.

Tecnicamente è possible incapsulare una variabile statica, ma quella variabile verrà comunque condivisa da tutti gli oggetti istanziati dalla stessa classe.

Quindi l'incapsulamento non si applica alle variabili statiche. Non è possibile applicare questo modificatore a variabili locali, parametri di metodi, costruttori o classi.

Anche senza istanziare la classe, l'utilizzo di un membro statico provocherà il caricamento in memoria della classe contenente il membro in questione che condividerà il ciclo di vita con quello della classe.

Un metodo statico essendo condiviso da tutti gli oggetti istanziati dalla classe in cui è dichiarato appartiene alla classe stessa e non ad un'istanza particolare.

Per questo motivo un membro statico ha la caratteristica di poter essere utilizzato mediante una sintassi del tipo:

NomeClasse.nomeMetodo (OK)

invece di

nomeOggetto.nomeMetodo (NO!)

Lo stesso vale per una variabile di istanza.

Un esempio di metodo statico è il metodo sqrt() della classe Math appartenente al package java.lang e che svolge la radice quadrata del numero fornito in input.

Come mostrato in Fig. 6 può essere chiamato tramite la sintassi Math.sqrt(numero).

Math è quindi il nome della classe e non il nome di un'istanza di quella classe (infatti si usa l'identificatore con la lettera maiuscola).

La ragione per cui la classe Math dichiara tutti i suoi metodi statici è facilmente comprensibile.

Trattandosi di operazioni matematiche, il funzionamento dipende dai parametri passati in ingresso piuttosto che dal valori di variabili di istanza. Infatti dichiarando due oggetti diversi obj1 e obj2 della classe Math, invocando i seguenti comandi:

obj1.sqrt(4);

obj2.sqrt(4);

si avrebbe lo stesso risultato ovvero 2. Questo perché sqrt calcola il risultato basandosi solo sull'input e non sugli attributi valorizzati di un particolare oggetto della classe Math.

Un metodo dichiarato static e public può considerarsi una sorta di funzione e non a caso la radice quadrata è definita come funzione matematica.

Per tale ragione questo tipo di approccio ai metodi è limitato solo ai casi in cui l'esecuzione del metodo non dipende dalle caratteristiche dell'oggetto su cui viene invocato il metodo.

```
import java.lang.*;

public class MathDemo {

   public static void main(String[] args) {

      // get two double numbers numbers
      double x = 9;
      double y = 25;

      // print the square root of these doubles
      System.out.println(Math.sqrt(x));
      System.out.println(Math.sqrt(y));
   }
}
```

Fig. 3: *Esempio di uso di un metodo statico sqrt*

Uno degli utilizzi più importanti dei modificatori in Java riguarda l'incapsulamento. L'incapsulamento è la proprietà per cui un oggetto contiene ("incapsula") al suo interno gli attributi (dati) e le operazioni (procedure) che agiscono sui dati stessi.

Tramite l'incapsulamento un oggetto appartenente ad una classe è considerato costituito da due parti distinte:

- un'interfaccia visibile all'esterno, interfaccia pubblica, che serve per l'interazione dell'oggetto con l'esterno.
- una parte interna nascosta all'esterno e costituita dai dati e dai metodi (che operano su tali dati) dando luogo così ad un certo comportamento.

La filosofia dell'incapsulamento si basa sull'accesso controllato ai dati mediante metodi speciali che possano prevenirne l'utilizzo non consono e gli errori.

L'incapsulamento è leggermente diverso dal concetto di astrazione.

L'incapsulamento corrisponde a separare l'interfaccia dall'implementazione degli oggetti. L'astrazione invece si riferisce al processo che porta ad estrarre le proprietà rilevanti di un'entità, ignorando i dettagli inessenziali.

Per applicare l'incapsulamento ad una classe occorre:

- Dichiarare privati gli attributi al fine di renderli inaccessibili al di fuori dalla classe stessa. A tale scopo si usa il modificatore private.
- Creare dei metodi pubblici (dichiarati con il modificatore public) che permettono l'accesso agli attributi anche da altre classi in maniera controllata. Tali metodi potrebbero realizzare controlli prima di confermare l'accesso ai dati privati evitando che essi assumano valori non validi.

I metodi pubblici nel loro insieme costituiscono l'interfaccia pubblica della classe perché visibili (e invocabili) dall'esterno, garantendo un accesso appropriato e al contempo semplice, chiaro e indipendente dall'implementazione interna.

In particolare, l'interfaccia pubblica non è costituita dalla implementazione dei metodi «interni» in quanto questi non sono visibili né modificabili all'esterno e quindi fanno parte della implementazione interna insieme ai dati. Quest'ultimi sono definiti con parola «private».

Si consideri l'esempio di codice Java mostrato in Fig. 4.

Supponiamo di voler scrivere un'applicazione che utilizza la classe Data, la quale astrae in maniera semplice il concetto di data. La classe data ha 3 variabili di istanza: giorno, mese, anno.

Dichiariamo la classe Data pubblica per renderla utilizzabile da altre classi e la definiamo in particolare anche con le variabili di istanza pubbliche.

Immaginiamo di usare tale classe da un'altra classe oppure dal main di un'applicazione software dove serve istanziare un oggetto di tipo Data inizializzato al valore 14 aprile 2004. Per fare questo inizializziamo le variabili di istanza giorno, mese, anno rispettivamente a 14, 4, 2004

```
…
public class Data {
        public int giorno;
        public int mese;
        public int anno;
} \\ definizione della classe in Java
…
\\ uso della classe nel programma con assegnazione diretta di valori
Data unaData = new Data();
unaData.giorno = 14;
unaData.mese = 4;
unaData.anno = 2004;
…
\\ uso della classe nel programma con assegnazione da interfaccia
grafica
Data unaData = new Data();
unaData.giorno = interfaccia.dammiGiornoInserito();
unaData.mese = interfaccia.dammiMeseInserito();
unaData.anno = interfaccia.dammiAnnoInserito();
\\ e se i valori inseriti fossero 32, 13, 2087?
```

Fig. 4: *Esempio di classe Data senza incapsulamento*

Qual è il problema? Il programma così fatto compilerebbe senza errori e funzionerebbe correttaemente.

Quindi sembrerebbe nessun problema. Supponiamo però che l'applicazione permetta all'utente di inserire la data tramite un'interfaccia grafica.

In tal caso l'inizializzazione avviene invocando dei metodi denominati dammiGiornoInserito, dammiMeseInserito, dammiAnnoInserito dell'oggetto interfaccia, metodi che restituiscono un numero intero inserito dall'utente mediante l'interfaccia grafica dell'applicazione.

Supponiamo che i valori inseriti siano 32 per il giorno, 13 per il mese e 2087 per l'anno; ecco che i problemi di questo codice cominciano ad emergere.

Come possiamo evitare che l'utente inserisca dei dati errati ovvero che delle variabili di istanza pubbliche assumano valore non opportuni?

Ci potrebbero essere le seguenti due opzioni per controllare l'accesso alle variabili di istanza.

1. si potrebbe limitare la possibilità degli inserimenti sull'interfaccia grafica all'utente e quindi prevedere dei controlli nella classe corrispondente all'oggetto interfaccia.
2. si potrebbe delegare ai metodi dammiGiornoInserito, dammiMeseInserito, dammiAnnoInserito

dell'oggetto interfaccia i controlli necessari per una corretta impostazione della data.

In entrambi i casi il problema sarebbe risolto ma solo nel caso in cui l'impostazione della data avvenga sempre e comunque tramite l'interfaccia grafica e quindi solo per quella specifica funzionalità.

Ma se volessimo riutilizzare in un'altra funzione la classe Data senza riutilizzare la stessa interfaccia grafica, saremmo costretti a scrivere nuovamente del codice di controllo per gestire il problema.

Come procedere allora? Facendo uso dell'incapsulamento e definendo diversamente la classe Data con gli attributi giorno, mese, anno definiti private e poi dei metodi public per accedere a tali dati sia in lettura che in scrittura come mostrato in Fig. 5.

```
…
public class Data {
        private int giorno, mese, anno;
        public void setGiorno (int g) {
                if (g>0 && g<=31) giorno=g;
                else System.out.println(«Giorno non valido»)
        }
        public int getGiorno () {
                return giorno;
        }
        public void setMese (int m) {
                if (m>0 && m<=12) mese=m;
                else System.out.println(«Mese non valido»)
        }
        public int getMese () {
                return mese;
        }
        public int setAnno (int a) {
                anno=a;
        }
        public int getAnno () {
                return anno;
        }
```

Fig. 5: *Esempio di classe Data con incapsulamento*

Questi metodi seguono una convenzione utilizzata anche nella libreria standard. In presenza di un variabile di istanza privata, chiamiamo questi metodi con la sintassi setNomeVariabile e getNomeVariabile.

Esiste un'unica eccezione a questa convenzione, quando incontriamo una variabile di tipo booleano. In questo caso il metodo getNomeVariabile si chiamerà isNomeVariabile.

Anche se all'inizio potrà risultare noioso (la seconda versione della classe Data è decisamente più estesa della prima) implementare l'incapsulamento non richiede la grossa inventiva da parte dello sviluppatore.

Non c'è niente di concettuale, solo da definire dei metodi banali. Tuttavia sono da scrivere e possono allungare la lista dei metodi significativamente come in questo caso. Fortunatamente tutti gli IDE offrono strumenti per la generazione automatica dei metodi get e set.

4. Reference this

Java introduce una parola chiave che per definizione coincide con il reference dell'oggetto corrente: this.

In generale, nella definizione di un metodo all'interno di una classe la variabile di istanza viene riferita direttamente. Per esempio, nella definizione del metodo getGiorno di una classe Data che astrae il concetto di data.

Si ha return giorno; dove giorno è una variabile di istanza della classe. Lo stesso vale per metodi di istanza. Ovvero si può fare riferimento a variabili e metodi d'istanza direttamente con il loro nome come mostrato in Fig. 6.

```
...
public int getGiorno () {
        return giorno;
}
...
```

Fig. 6: *Uso variabile di istanza possibile attraverso il loro nome senza indicare il nome dell'istanza dell'oggetto*

D'altra parte ogni variabile di istanza, come giorno, appartiene ad un oggetto.

Ma il nome dell'istanza della classe (oggetto) su cui è invocata non potrebbe essere usato visto che siamo nella parte di definizione della classe e non nel main.

Quindi in genere si sottintende l'oggetto omettendone il nome. Tuttavia, l'oggetto (a cui appartiene la variabile di istanza) lo si può esplicitamente referenziare se si vuole usando il reference this.

Il nome sottinteso di questo oggetto è this (letteralmente "questo"). Per esempio, l'assegnamento precedente è equivalente a return this.giorno come mostrato in Fig. 7.

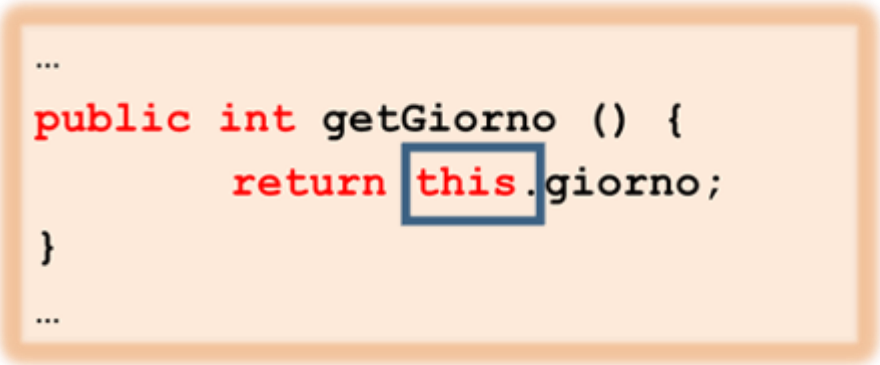

Fig. 7: *Uso variabile di istanza tramite reference this*

Il reference this viene implicitamente aggiunto nel bytecode compilato, per referenziare esplicitamente l'oggetto a cui si riferisce la variabile di istanza.

La parola chiave this rappresenta l'oggetto che riceve l'invocazione del metodo e quindi a cui si riferisce una variabile di istanza.

In generale non si usa this, perché si può fare riferimento a variabili e metodi d'istanza direttamente con il loro nome.

Quindi non importa specificarlo ed è aggiunto in automatico dalla JVM. Occorre usarlo in casi di ambiguità ed è per questo che ne parliamo.

Vediamo quali sono.

In Java si fa la distinzione tra variabili di istanza e variabili locali le quali vengono memorizzate in due aree di memoria differenti, rispettivamente la Heap Memory e la Stack Memory.

Questo fa sì che il compilatore consente di dichiarare nella stessa classe, una variabile locale (o un parametro di un metodo) ed una variabile d'istanza anche usando lo stesso identificatore.

Tuttavia spesso capita di fare passaggi di parametri nei metodi al fine di inizializzare variabili d'istanza. Sino ad ora, per il parametro passato siamo stati spinti ad inventare un identificatore differente da quello della variabile d'istanza da inizializzare.

Invece possiamo usare il reference this, per risolvere eventuali ambiguità. Per esempio consideriamo la seguente classe Cliente definita come mostrato in Fig. 8:

```
public class Cliente {

        private String nome, indirizzo, numeroDiTelefono;

        public void setNumeroDiTelefono (String numeroDiTelefono) {
        numeroDiTelefono = numeroDiTelefono;
        }
        public void setIndirizzo (String indirizzo) {
        indirizzo = indirizzo;
        }
        public void setNome (String nome) {
        nome = nome;
        }
//metodi get omessi…
}
```

Fig. 8: *Esempio di ambiguità nell'uso delle variabili di istanza*

Consideriamo per esempio il metodo setNome.

Chiaramente l'intenzione è quella di assegnare a nome, variabile di istanza, il valore dì nome parametro del metodo.

A prima vista può sembrare illegale, poiché all'interno del corpo del metodo si hanno due variabili con lo stesso identificativo: la variabile di istanza e la variabile locale al metodo. In realtà il compilatore non produce nessun messaggio di errore.

Infatti, all'interno di un blocco è possibile avere due variabili con lo stesso identificativo, a patto che una sia una variabile di istanza e l'altra una variabile locale al metodo (o anche un parametro) come si ha in questo caso.

Se però si tenta di eseguire il seguente codice:

```
…
Cliente unCliente = new Cliente();
unCliente.setNome(Carlo);
System.out.println(unCliente.getNome());
// il risultato è nome=«»
…
```

L'output prodotto sarà la stringa vuota «». In altre parole, non viene assegnato il valore Carlo alla variabile di istanza nome e il suo valore rimane «», il valore assegnato per default ai tipi di tipo String.

Tale comportamento è dovuto al fatto che nell'istruzione di assegnamento: nome=nome entrambe le variabili vengono considerate come la variabile locale al metodo non trovando riferimenti espliciti.

Praticamente il codice non fa altro che riassegnare alla variabile locale il suo valore.

Poiché l'intento era quello di assegnare alla variabile di istanza dell'oggetto invocante il metodo (setNome) il valore passato come argomento al metodo, occorre esplicitare che la prima variabile è in realtà la variabile di istanza e non la variabile locale.

Quindi this serve per distinguere tra una variabile di istanza e una variabile locale (o parametro) all'interno di un metodo.

Tramite la parola chiave this, intendiamo che la variabile referenziata appartiene all'istanza dell'oggetto che riceve l'invocazione del metodo.

Di conseguenza, per indicare che la prima variabile dell'assegnamento è la variabile di istanza si modifica il codice del metodo anteponendo alla variabile di istanza il reference this come mostrato nel codice rivisto per la classe Cliente.

Di conseguenza la variabile non referenziata sarà il parametro del metodo senza che vi sia ambiguità come mostrato nella classe Cliente rivista come mostrato di seguito:

```
…
public class Cliente {

private String nome, indirizzo, numeroDiTelefono;
public void setNumeroDiTelefono (String numeroDiTelefono) {
        this.numeroDiTelefono = numeroDiTelefono;
}
public void setIndirizzo (String indirizzo) {
        this.indirizzo = indirizzo;
}
public void setNome (String nome) {
        this.nome = nome;
}
//metodi get omessi…
}
```

Con questa modifica, a fronte dell'esecuzione delle istruzioni:

```
…
Cliente unCliente = new Cliente();
unCliente.setNome(Carlo);
System.out.println(unCliente.getNome());
// il risultato è Carlo
…
```

L'output sarà Carlo, che è esattamente quello che si voleva ottenere.

In generale per riferirsi a una variabile di istanza occorre utilizzare la seguente sintassi: this.nome_della_variabile_di_istanza dove la parola chiave this rappresenta l'oggetto che riceve l'invocazione del metodo.

Per quanto detto this rappresenta l'oggetto corrente e può quindi referenziare le variabili d'istanza. Naturalmente con this è anche possibile invocare metodi sull'oggetto corrente. Ad esempio, sempre nella classe Cliente vediamo di seguito come nel metodo costruttore sono stati invocati i metodi set sfruttando il reference this.

```
…
public class Cliente {

private String nome, indirizzo, numeroDiTelefono;

public Cliente (String nome, String indirizzo, String numeroDiTelefono) {
        this.setNome (nome);
        this.setIndirizzo(indirizzo);
        this.setNumeroDiTelefono(numeroDiTelefono)
}

//metodi pubblici set delle variabili di istanza come precedenti
//metodi get omessi
}
```

In questo caso però se non avessimo anteposto il reference this ai metodi set, avremmo comunque ottenuto lo stesso risultato visto che il compilatore lo avrebbe inserito implicitamente (e non siamo nel caso di ambiguità visto prima).

Più interessante è l'uso che si fa solitamente del reference this con i costruttori. La sintassi cambia visto che per un costruttore l'operatore dot non è utilizzabile.

Consideriamo sempre la classe Cliente modificata come segue:

```
…
public class Cliente {

private String nome, indirizzo, numeroDiTelefono;

public Cliente (String nome, String indirizzo) {
        this(nome, indirizzo, «sconosciuto»);
}

public Cliente (String nome, String indirizzo, String numeroDiTelefono) {
        this.setNome (nome);
        this.setIndirizzo(indirizzo);
        this.setNumeroDiTelefono(numeroDiTelefono)
}

//metodi pubblici set delle variabili di istanza come precedenti
//metodi get omessi
}
```

con il costruttore che prende in ingresso 3 valori corrispondenti alle 3 variabili di istanza e li imposta invocando i relativi metodi set.

In aggiunta definiamo un altro costruttore che prende in ingresso 2 valori per nome ed indirizzo e li imposta invocando l'altro costruttore con tre parametri di tipo stringa dopo la parola chiave this che coincidono con i parametri del secondo costruttore.

In questo caso si è riusato il codice del secondo costruttore senza scrivere di nuovo le stesse istruzioni. Questo tipo di pratica è raccomandato.

L'utilizzo della parla chiave this per invocare un costruttore è consentito solo all'interno di un altro costruttore. L'invocazione di un altro costruttore è consentita come prima istruzione.

5. Ereditarietà

Come tutti i paradigmi che caratterizzano l'Object Oriented anche l'ereditarietà è ispirata a qualcosa che esiste nella realtà.

Nel mondo reale classifichiamo i concetti sia fisici che astratti in classi e sottoclassi.

Per esempio un cane è un animale così come lo è un moscerino, un aereo è un veicolo così come lo è un'auto, la chitarra è uno strumento musicale così come lo è un flauto.

In OO (e quindi anche Java) l'ereditarietà è la caratteristica che mette in relazione di estensibilità più classi che hanno caratteristiche comuni.

Per esempio la classe Cane e la classe Moscerino saranno sottoclassi ed estenderanno entrambe la classe Animale andando a definirne delle proprietà specifiche.

Al tempo stesso la classe Animale sarà la superclasse ed esprimerà le sole caratteristiche comuni a tutti gli animali, es. categoria (mammifero, rettile, volatile….).

Si parla di generalizzazione se a partire da un certo numero di classi si definisce una superclasse che ne raccoglie le caratteristiche comuni. Viceversa si parla di specializzazione quando, partendo da una classe si definiscono una o più sottoclassi allo scopo di ottenere oggetti più specializzati.

Si definisce una classe derivata, o sottoclasse, partendo dalla definizione di un'altra classe già definita e aggiungendo o modificando i metodi e le variabili di istanza necessari.

La classe di partenza è detta classe base o superclasse.

La classe derivata eredita tutte le variabili di istanza, le variabili statiche e tutti i metodi pubblici della classe base e può aggiungere variabili proprie e metodi propri.

Il riuso anche facilitato da ereditarietà. Il risultato immediato infatti è la possibilità di ereditare codice già scritto, e quindi di gestire insiemi di classi collettivamente, giacché accomunate da alcune caratteristiche.

Le caratteristiche della superclasse siano automaticamente disponibili per la sottoclasse.

La superclasse è riusata nelle funzionalità base (a comune), la sottoclasse definisce poi operazioni e funzionalità più specifiche.

Anche il codice a comune fra più sottoclassi può essere messo a comune nella superclasse senza riscriverlo due volte uguale o quasi nelle due sottoclassi.

Sebbene l'ereditarietà sia un argomento agevole da comprendere nei principi di base, non è semplice utilizzarla in modo corretto e al pieno delle sue potenzialità.

La parola chiave extends ci permette di implementare l'ereditarietà:

public class nome_sottoclasse extends nome_superclasse

{

// variabili aggiuntive

// metodi aggiuntivi e/o modificati

}

C'è un modo molto semplice per capire quando si deve implementare l'ereditarietà.

È necessario porsi la semplice domanda «un oggetto della candidata sottoclasse è un oggetto della candidata superclasse»? In altre parole occorre capire se c'è una relazione del tipo «è un» («is a» in inglese) tra le due classi candidate. Se la risposta alla domanda è positiva l'ereditarietà si deve applicare, altrimenti no.

Questo test permette di capire una vera relazione di ereditarietà tra due classi proprio a livello concettuale così come deve essere o, in altri termini, di smascherare situazioni che potrebbero portarci a pensare che ci sia una relazione di ereditarietà quando invece non c'è.

Ad esempio la presenza di attributi comuni fra due classi potrebbe far pensare che ci sia una relazione di ereditarietà ma non sempre è così.

Prendiamo come esempio di classi Studente e Professore (mostrate di seguito) pensate per una ipotetica applicazione per l'archiviazione di informazioni di un'università.

```
public class Studente {
        public String nome;
        public String cognome;
        public String dataDiNascita;
        public int numeroEsamiSostenuti;
        //…
}
public class Professore extends Studente {
        public String materia;
        //…
}
```

Fig. 9: *Esempio di classe e sottoclasse*

La classe Studente definisce fra gli altri attributi come nome, cognome, data di nascita.

Dal momento che anche la classe Professore ha nome, cognome e data di nascita come attributi si potrebbe pensare di derivare la classe Professore dalla classe Studente.

Ma un professore non è uno studente (né viceversa), quindi non è rispettata la relazione «è un» e quindi non è opportuna una relazione di ereditarietà fra di essi.

Se mettessimo in relazione di ereditarietà in modo errato due classi, in un primo momento risparmieremmo del codice, in seguito tuttavia sorgerebbero problemi quando tenteremmo di usare un oggetto professore in modo analogo ad un oggetto studente proprio perché non c'è la relazione a livello concettuale.

Inoltre la classe Studente potrebbe evolvere in modo divergente dal concetto Professore (ad esempio con aggiunta sulle info di voto, media etc etc), caratteristiche che sarebbero automaticamente ereditate dalla classe Professore, creando confusione e inconsistenze nel codice.

Un modo di procedere concettualmente in modo corretto nell'utilizzo della ereditarietà è il seguente.

Prendiamo ad esempio le classi Studente e Professore definite prima. Il test «è un» fallisce e quindi la classe Professore non è una specializzazione della classe Studente.

D'altro canto le due classi hanno dei campi in comune e non sembra un evento casuale.

In effetti, sia uno studente sia un professore sono entrambe delle persone. La soluzione a questo problema risulta quindi naturale.

Basta generalizzare le due astrazioni in una classe Persona, affinché le classi Studente e Professore possano estenderla come segue.

```
public class Persona {
        public String nome;
        public String cognome;
        public String dataDiNascita;
        //…
}
public class Studente extends Persona {
        public int numeroEsamiSostenuti;
        //…
}
public class Professore extends Persona {
        public String materia;
        //…
}
```

Fig. 10: *Esempio di classe e più sottoclassi*

Se fossimo partiti dalla classe Persona per poi definire le due sottoclassi avremmo parlato invece di specializzazione.

È possibile che più classi ne estendano un'altra.

Ad esempio le classi Segretario, Direttore, Bibliotecario estendono tutte la classe Persona.

Non è possibile che una classe ne estenda più di una per volta ma solo che più sottoclassi esistano di una superclasse diretta.

Tuttavia l'ereditarietà può generare gerarchie a più livelli, ovvero una classe ProfessoreAssociato potrebbe estendere Professore che estende a sua volta estende Persona.

In tal caso Persona è, per la proprietà transitiva, una superclasse anche di ProfessoreAssociato. Infatti un professore associato, oltre ad essere un professore è anche una persona.

La gerarchia può anche estendersi ulteriormente.

E allora potremo dire che oltre ai professori lavorano in università anche il personale amministrativo, a sua volta specializzazione di un'altra astrazione che è il dipendente dell'università rappresentato dalla classe Dipendente che insieme a studente sono messe a comune tramite la classe Persona.

A prescindere dal fatto che sia uno studente o un dipendente (sia esso professore o amministrativo) una scheda gestisce comunque informazioni su una persona.

Le schede gestite dal sistema sono quindi tutte schede di persone.

Troviamo anche una specializzazione della classe Professore rappresentato dal ProfessoreAssociato e della classe Studente rappresentato dalla classe StudenteDiDottorato.

Quest'ultimo oltre all'informazione sulla matricola e il numero di esami sostenuti presenta le caratteristiche peculiari date dal numero di articoli pubblicati su riviste scientifiche e l'informazione booleana se ha trascorso il periodo all'estero oppure no.

La gerarchia completa è mostrata in Fig. 8. Sebbene il programma possa non aver bisogno di classi che corrispondono a persone o a dipendenti, pensare in termini di tali classi può essere utile.

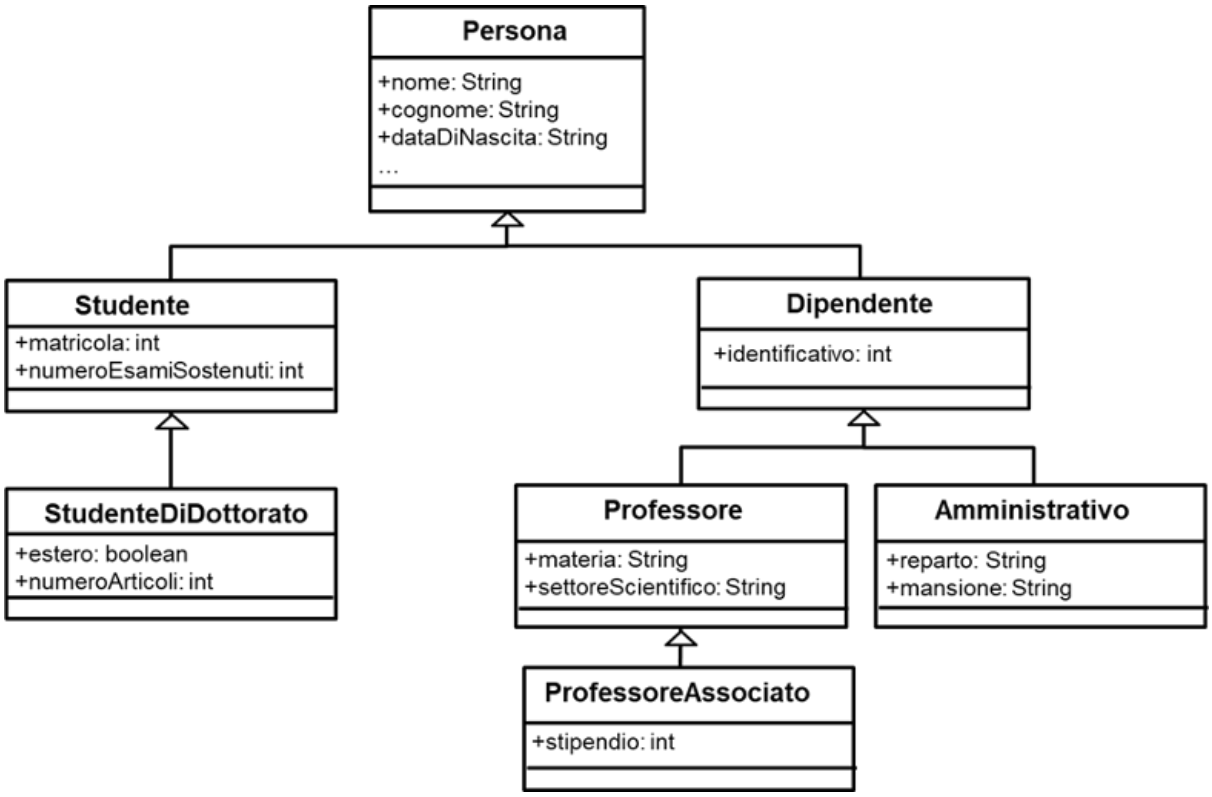

Fig. 11: *Gerarchia di classi derivate da Persona*

Per esempio tutte le persone possiedono un nome e i metodi di inizializzazione, visualizzazione e modifica del nome saranno gli stessi per studenti, professori e amministrativi come mostrato nella Figura 8.

6. Struttura

Co Dopo aver visto i concetti base di ereditarietà vediamo come questa si combina con il concetto di incapsulamento.

Cosa viene ereditato da una classe incapsulata?

Non vengono ereditati né i metodi privati né le variabili di istanza private, vengono ereditati solo i metodi pubblici.

Una sottoclasse non può accedere direttamente alle variabili di istanza private della sua classe base.

La sottoclasse conosce infatti il comportamento pubblico della classe base, e si presuppone che non conosca (e che non vi sia interessata) il modo in cui la classe base gestisce i propri dati che è alla base del concetto di incapsulamento che vale anche per la sottoclasse.

Tuttavia per l'accesso alle variabili di istanza private questo può essere reso possibile dalla classe derivata attraverso dei metodi pubblici (fra cui i metodi get e set realtivi ad una specifica variabile di istanza privata) che contengono riferimenti alle variabili di istanza private e che la classe derivata eredita in quanto pubblici.

I metodi privati questi sono totalmente e in ogni caso sempre inaccessibili direttamente.

Tuttavia la sottoclasse può chiamare metodi pubblici cha a loro volta chiamano metodi privati a patto che entrambi i metodi siano stati definiti nella classe base (ovvero i metodi privati possono essere usati indirettamente tramite metodi pubblici che li invoca).

Il fatto che una classe derivata non sia in grado di usare metodi privati della classe base non dovrebbe rappresentare un problema.

I metodi privati dovrebbero essere utilizzati come metodi di supporto ad altri metodi, quindi il loro utilizzo dovrebbe essere limitato alla classe nella quale sono definiti (uso interno).

Se si desidera utilizzare nei metodi delle classi derivate un metodo di supporto definito nella classe base, significa che non è un semplice metodo di supporto per la classe base, ma deve essere dichiarato pubblico affiché sia accessibili dalle sottoclassi.

Infine vengono ereditati i metodi e variabili di istanza protetti, che sono visibili anche a tutte le classi del package.

Riprendiamo l'esempio della classe Persona estesa qui di seguito con anche il costruttore e i metodi get e set.

```
public class Persona {
        private String nome;
        private String cognome;
        private String dataDiNascita;

        public Persona (String nomeIniziale) { nome = nomeIniziale; }
        public setNome (String nuovoNome) { nome = nuovoNome; }
        public getNome () { return nome; }
        public void scriviOutput () {
                System.out.println («Nome: «» + nome);
        }
//altri metodi…
}
```

In questo frammento di codice della classe Persona si hanno le definizioni di tutte quelle variabili di istanza che rappresentano le proprietà comuni a tutte le sottoclassi della classe Persona ovvero le variabili di istanza nome, cognome e data di nascita.

Per motivi di incapsulamento tali variabili sono definite private e poi sono forniti anche i metodi pubblici che permettono l'accesso (controllato) a tali variabili: get e set.

Le sottoclassi la estenderanno con variabili di istanza e metodi specifici senza bisogno di ridefinire quelli pubblici dalla superclasse perché le eredita dalla classe Persona.

Consideriamo la classe Studente modificata come segue:

```
public class Studente extend Persona {
        public int matricola;
        public int numeroEsamiSostenuti;
        // altre variabili di istanza qui……
        // costruttori qui…
        }
        public getMatricola () { return nome; }
        public setMatricola (int nuovaMatricola) { matricola = nuovaMatricola; }
        public reimposta (String nuovoNome, int nuovaMatricola) {
                setNome(nuovoNome);
                matricola = nuovaMatricola;
        }
        public void scriviOutput () {
                System.out.println («Nome: «» + getNome);
                System.out.println («Matricola: «» + matricola);
        }
//altri metodi…
}
```

La classe Studente come visto aggiunge la variabile di istanza matricola e numeroEsamiSostenuti alle variabili di istanza della classe Persona che in questo esempio sono definiti pubblici per semplicità ma

per il principio di incapsulamento dovrebbe essere definiti privati e poi fornire i metodi get e set corrispondenti (qui fornito solo per Matricola).

Inoltre definisce i metodi pubblici reimposta e scriviOutput. Inoltre la classe Studente ha tutti i metodi pubblici della classe Persona.

Per questo motivo può invocare il metodo setNome all'interno del metodo reimposta. Lo stesso dicasi per il metodo getNome invocato dentro il metodo scriviOutput.

Di seguito un frammento di un main che usa la classe Studente:

```
public class DemoEreditarietàEIncapsulamento  {
        public static void main (String[] args) {
        Studente s = new Studente();
        s.setNome(«Mario»);
        //assegna il valore Mario alla variabile di istanza nome
        //dell'oggetto s
        // s.nome = «Giovanni» avrebbe dato errore
        s.setMatricola(1234);
        //s.matricola = 1234 permesso ma non raccomandato
        gio.scriviOutput();
        }
}
```

Si immagini di istanziare un oggetto della classe Studente come segue: Studente s = new Studente;

Uno dei membri che un oggetto di tipo Studente eredita dalla classe Persona è la variabile di istanza nome.

Dopo aver definito un oggetto s di tipo Studente assegnamo alla variabile di istanza nome dell'oggetto s il valore «Mario».

La variabile di istanza nome della classe Studente è stata ereditata dalla classe Persona, ma si tratta di una variabile di istanza privata della classe Persona.

Questo significa che si può accedere alla variabile di istanza nome solamente attraverso i metodi definiti nella classe Persona.

Non avremmo potuto usare lo statement s.nome in quanto le varaibili di istanza che sono dichiarati privati in una classe base non sono accessibili direttamente (ovvero per messo del loro identificativo) dai metodi di nessun'altra classe, incluse le classi derivate.

Quindi s.nome avrebbe dato errore ed necessario come fatto nell'esempio usare un metodo pubblico per accedere alla variabile di istanza nome, come appunto setNome per impostare un valore come mostrato.

Per contro matricola definito come variabile di istanza specifica della classe Studente può essere impostata sempre usando il relativo metodo set (con il nome setMatricola) ma in questo caso era pure accessibile direttamente e s.matricola non avrebbe dato errore perché permesso.

Tuttavia come stile di programmazione raccomandato è preferibile usare s.setMatricola. Infine si può stampare a video il nome e la matricola tramite il metodo scriviOutput.

Guardando meglio questo metodo era definito nella classe Persona ma anche poi ridefinito nella classe Studente.

Come detto una sottoclasse include automaticamente tutte le variabili di istanza, le variabili statiche e tutti i metodi pubblici della classe base.

I membri ottenuti dalla classe base si dicono ereditati. Le variabili di istanza, le variabili statiche e i metodi pubblici ereditati dalla classe base non sono dichiarati esplicitamente nella definizione della sottoclasse, ma diventano automaticamente suoi membri.

Esiste una solo eccezione a questa regola: in una sottoclasse derivata è possibile modificare un metodo ereditato.

Questa nuova definizione ridefinirà il comportamento del metodo solo nella classe derivata/sottoclasse. Questo è il caso del metodo scriviOutput ridefinito nella classe Studente.

Come mai se un oggetto di una classe derivata è anche un oggetto della classe base non riesce ad accedere ai suoi membri privati e in particolare alle variabili di istanza? Per preservare la correttezza del programma che potrebbe venir compromessa.

Se una variabile di istanza privata di una classe fosse accessibile dalla definizione di un metodo di una sottoclasse, ogniqualvolta si volesse accedere a una variabile di istanza privata basterebbe creare una classe derivata e accedervi da un metodo di questa classe.

Questo vorrebbe dire rendere accessibili (a chiunque voglia fare lo sforzo di definire una classe derivata) le variabili di istanza private definite in una classe.

Questo scenario illustra la problematica, ma il reale problema che si verrebbe a creare è

l'involontaria introduzione di errori.

Se le variabili di istanza private di una classe fossero accessibili dalle definizioni dei metodi delle classi derivate, il programmatore potrebbe modificarne il valore inavvertitamente o in modo inappropriato (mentre con i metodi get e set si proteggono le variabili di istanza da cambiamenti inappropriati).

A questo riguardo risulta importante introdurre il modificatore di accesso protected.

Si consideri ancora la classe Studente derivata dalla classe base Persona modificata come segue:

```
public class Persona {
        protected String nome;
        ….
        //altri variabili di istanza e metodi…
}

public class DemoEreditarietàEIncapsulamento  {
        public static void main (String[] args) {
        Studente gio = new Studente;
        gio.nome = «Giovanni»
        //assegna il valore Giovanni alla variabile di istanza nome
        //dell'oggetto gio
        //gio.setNome(«Giovanni»); pure possibile

        gio.matricola = 1234;
        //gio.setMatricola(1234) pure possibile

        gio.scriviOutput();
        }
}
```

Se la variabile di istanza nome fosse specificata come protected nella classe Persona, allora la variabile di istanza nome potrebbe essere acceduta direttamente, e ancora tramite il metodo set se ancora disponibile e come la variabile di istanza matricola definita nella classe derivata (anche qui possibile accedervi tramite metodo pubblico set corrispondente).

Il modificatore d'accesso protected garantisce un basso livello di protezione rispetto al modificatore d'accesso private perché qualsiasi programmatore può accedere direttamente a un membro protected definendo una classe derivata.

Per tale motivo L'utilizzo del modificatore protected è spesso sconsigliato e le variabili di istanza non dovrebbero essere specificate come protected.

Solo in rare occasioni si potrebbe voler dichiarare un metodo come protected.

Un errore in cui un programmatore neofita potrebbe incorrere è pensare che per utilizzare una variabile in una sottoclasse sia necessario dichiararla protected, invece è più che sufficiente avere a disposizione i metodi set e get nelle sottoclassi in virtù dell'incapsulamento.

D'altro canto dichiarare protetta una variabile di istanza significa renderla pubblica a tutte le classi dello stesso package.

Analizziamo ora il rapporto fra ereditarietà e costruttori.

Il costruttore è un metodo speciale presente in ogni classe. Infatti anche quando il programmatore non ne prevede uno esplicitamente è il compilatore che ne introduce uno al momento della compilazione, il «costruttore di default».

Serve solitamente ad inizializzare le variabili (o attributi) degli oggetti al momento dell'istanza ed ha le seguenti proprietà:

- Ha lo stesso nome della classe a cui appartiene;
- Non ha tipo di ritorno
- È chiamato automaticamente e solamente ogni volta che viene istanziato un oggetto della classe a cui appartiene, relativamente a quell'oggetto.

Cosa succede in caso di classe derivate e quindi quando entra in gioco l'ereditarietà?

L'ereditarietà non è applicabile ai costruttori. Anche quando sono dichiarati pubblici, i costruttori non sono ereditati per un motivo semplicissimo: il loro nome.

Il nome del costruttore è il nome stesso della classe di appartenenza e si invoca attraverso una invocazione new con il nome della classe stessa.

Avendo la superclasse e sottoclasse nomi diversi non si può ereditare il costruttore. Consideriamo il seguente esempio.

```
public class Libro {
        public String titolo, autore, editore;
        public int numeroPagine, prezzo;
        //…
}
public class LibroSuJava extends Libro
        public static final String ARGOMENTO = «Java»;
        //altre definizioni di variabili e metodi aggiuntivi
}
……
//nel main…
LibroSuJava libroSuJava = new LibroSuJava();
// viene invocato il costruttore LibroSuJava()

LibroSuJava libroSuJava = new Libro();
// viene invocato il costruttore Libro() ma non si istanzia la
classe LibroSuJava (oltre ad avere errore di compilazione)
….
```

Per esempio se la classe LibroSuJava ereditasse dalla classe Libro un costruttore ne erediterebbe uno che si chiama proprio Libro(). Ma un costruttore che si chiama Libro, in una classe che si chiama

LibroSuJava non potrà mai essere chiamato al momento in cui si istanzia una classe LibroSuJava quando invece viene invocato il costruttore di nome LibroSuJava().

Infatti per istanziare un oggetto dalla classe LibroSuJava è strettamente necessario chiamare un costruttore di nome LibroSuJava() in modo da inizializzare tutte le variabili di istanza.

Per invocare il costruttore Libro() occorre istanziare con una new una classe di tipo Libro. Per esempio, dato il codice:

LibroSuJava libroSuJava = new LibroSuJava();

con cui si istanzia una classe LibroSuJava e quindi si invoca il costruttore LibroSuJava(), si considera al suo posto:

LibroSuJava libroSuJava = new Libro(); istanzieremmo un oggetto della classe Libro (con la chiamata al rispettivo costruttore) ma non un oggetto della classe LibroSuJava (ottenendo anche un errore in compilazione in quanto l'assegnazione non è corretta).

Quindi il fatto che i costruttori non siano ereditati dalle sottoclassi è coerente con la sintassi del linguaggio.

Tuttavia sembra violare i principi del riuso e dell'astrazione.

Ci si aspetterebbe di poter riutilizzare il costruttore della superclasse, essendo la sottoclasse anche un oggetto della superclasse dal momento che verifica la relazione «è un».

Un libro su Java essendo anche un libro deve avere tutte le caratteristiche di un libro.

In particolare abbiamo bisogno di inizializzare fra le variabili di istanza anche quelle ereditate dal Libro.

E allora perché non riutilizzare anche il costruttore della superclasse?

In realtà una proprietà del costruttore oltre ad avere lo stesso nome della classe a cui appartiene, a non avere un tipo di ritorno, a essere chiamato automaticamente (e solamente) ogni volta che è istanziato l'oggetto della classe a cui appartiene, è anche che: qualsiasi costruttore (anche di default) come prima istruzione invoca sempre un costruttore della superclasse.

Il principio di riuso e astrazione sono salvi.

Consideriamo il seguente esempio:

```
public class Libro {
        // variabili di istanza titolo, autore, editore, numeroPagine, prezzo
        public Libro() {
        System.out.println(«Costruito un Libro!»)
        }
        // costruttore
        // altri metodi
}
…
public class LibroSuJava extends Libro
        public static final String ARGOMENTO = «Java»;
        public LibroSuJava() {
        System.out.println(«Costruito un Libro su Java!»)
        }
        //altre definizioni di variabili e metodi aggiuntivi
}
…
//nel main…
new LibroSuJava();
// invocato costruttore di LibroSuJava che invoca come prima istruzione il
costruttore della superclasse Libro().
```

Consideriamo ancora le classi Libro e LibroSuJava con LibroSuJava che estende Libro andando a specializzarne alcuni aspetti come visto. Andiamo ad aggiungere dei costruttori.

Immaginiamo in un main di istanziare un oggetto LibroSuJava con una new LibroSuJava(); Si consideri che l'assegnazione di un reference non è obbligatoria per avere una istanza.

Avendo appreso che i costruttori non sono ereditati, ci si aspetterebbe un output con la scritta «Costruito un Libro su Java!» in realtà avremo entrambe le scritte «Costruito un Libro!» e «Costruito un Libro su Java!»

Il costruttore LibroSuJava ha prima invocato il costruttore della superclasse Libro() e poi ha eseguito la sua istruzione.

Quindi riepilogando….

Il costruttore è un metodo speciale presente in ogni classe che serve ad inizializzare le variabili (o attributi) degli oggetti al momento si istanza un oggetto:

- Ha lo stesso nome della classe a cui appartiene;
- Non ha tipo di ritorno
- È chiamato ogni volta che viene istanziato un oggetto
- Invoca sempre come prima istruzione un costruttore della superclasse

Quindi nel costruttore della sottoclasse è come se ci fosse implicitamente una chiamata al costruttore della superclasse Libro prima di eseguire le proprie istruzioni.

A sua volta questo vale per la superclasse qualora derivasse da un'altra classe dando luogo ad un comportamento ricorsivo.

È cose se si usasse un riferimento implicito alla superclasse e questo venisse usato per invocare il suo costruttore.

Questo un po' ricorda quanto succede con la parola chiave this usato come reference all'oggetto corrente.

La chiamata esplicita al costruttore della superclasse viene effettuata tramite la parola chiava super.

Il reference super consente di accedere ai componenti della superclasse e in particolare al suo costruttore.

Infatti la parola chiave super è strettamente connessa al concetto di costruttore. In ogni costruttore infatti è sempre presente una chiamata al costruttore della superclasse mediante una sintassi speciale che sfrutta il reference super.

Per esempio nella classe LibroSuJava, il costruttore sarà modificato dal compilatore nel seguente modo:

```
public class Libro {
        // variabili di istanza titolo, autore, editore,
        // variabili di istanza numeroPagine, prezzo
        public Libro() {
        System.out.println(«Costruito un Libro!»)
        }
        // costruttore
        // altri metodi
}
…
public class LibroSuJava extends Libro
        public static final String ARGOMENTO = «Java»;
        public LibroSuJava() {
        super();
        System.out.println(«Costruito un Libro su Java!»)
        }
        //altre definizioni di variabili e metodi aggiuntivi
}
…
//nel main…
new LibroSuJava();
// invocato costruttore di LibroSuJava che invoca come prima
istruzione il costruttore della superclasse Libro().
```

Ecco perché il costruttore della classe Libro (senza parametri) 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.

Il costruttore della classe Libro chiamerà il costruttore della sua superclasse Object mediante un comando super() implicito.

Supponiamo di voler dotare la superclasse Libro di un costruttore che imposta un valore alla variabile titolo.

D'altronde un oggetto di tipo Libro almeno un titolo lo dovrebbe avere.

Definiamo un costruttore che prende in ingresso una stringa da impostare come titolo nella variabile di istanza titolo. Questa classe compilerà ma LibroSuJava non compilerà più.

```
public class Libro {
        // variabili di istanza titolo, autore, editore,
        // variabili di istanza numeroPagine, prezzo
        public Libro(String titolo) {
        this.titolo = titolo
        }
        //costruttore
        // altri metodi
}
```

```
LibroSuJava.java: 1:error: constructor Libro in class Libro
     cannot be applied to given types;
Public class LibroSuJava extend Libro {
       ^
     required: String
     found: no arguments
     reason: actual and formal argument lists differ in length
1 error
```

Modificando la superclasse non compila la sottoclasse.

Infatti quando viene compilata la sottoclasse il costruttore della sottoclasse ha cercato di invocare il costruttore della superclasse.

La superclasse non ha più un costruttore senza parametri come invocato dalla chiamata super() e quindi il costruttore che si cerca di invocare è inesistente.

Infatti se un costruttore viene aggiunto esplicitamente, nessun altro costruttore di default viene inserito implicitamente.

Quindi l'istruzione super() inserita automaticamente nella sottoclasse LibroSuJava, prova a chiamare il costruttore della superclasse senza parametri (infatti non sono passati parametri tra le parentesi).

Per risolvere il problema, occorre modificare il costruttore di LibroSuJava in modo tale da fargli invocare il costruttore della superclasse Libro che prende come parametro di input il titolo nel seguente modo:

```
public class Libro {
        // variabili di istanza titolo, autore, editore,
        // variabili di istanza numeroPagine, prezzo
        public Libro(String titolo) {
        this.titolo = titolo
        }
        //costruttore
        // altri metodi
}

public class LibroSuJava extends Libro
        public LibroSuJava(String titolo) {
        super(titolo);
        }
        //…
}
```

Nella classe LibroSuJava abbiamo invocato esplicitamente il costruttore della superclasse che prendo in input una stringa.

Se avessimo più costruttori nella superclasse potremmo scegliere quale chiamare a seconda della necessità.

Bibliografia

- Il nuovo Java, Claudio De Sio Cesari, Hoepli Informatica
- Programmazione di base e avanzata con Java, Walter Savitch, Pearson