

Pacchetti, Contratti ed Ereditarietà

Programmazione

Corso di Laurea in Informatica

Corso di Laurea in Informatica per la Comunicazione Digitale

Università di Camerino

Prof. Michele Loreti

Anno Accademico 2024/25

Classi, Pacchetti e file

In Java le classi fanno parte di un *pacchetto (package)*.

Un package viene usato per raggruppare classi che condividono finalità.

Ad esempio, se guardiamo le librerie standard in Java abbiamo:

- `java.math`
- `java.net`
- `java.io`

Classi, Pacchetti e file

Il nome di un pacchetto dovrebbe individuare in modo *univoco* chi ha sviluppato quel codice.

Per questa ragione, la struttura dei pacchetti segue lo stesso schema delle URL, ma in modo invertito:

- org.apache.commons.math3

Dichiarazione del pacchetto

La dichiarazione del pacchetto deve essere inserita all'inizio del file dove si trova la classe:

```
package nome.pacchetto;  
public class NomeClasse {  
}
```

Classi, Pacchetti e file

Per fare riferimento ad una classe `ClasseA` dichiarata nel pacchetto `a.b.c` viene univocamente identificata come `a.b.c.ClasseA`.

Se vogliamo usare questa classe in un pacchetto diverso da `a.b.c`, sarà conveniente importare la classe, o il pacchetto all'inizio del file, dopo la dichiarazione del package:

```
import a.b.c.ClasseA;
```

Classi, Pacchetti e file

Il nome del pacchetto viene utilizzato anche per ricercare la classe.

Per questa ragione, la classe `a.b.c.ClasseA` dichiarata nel pacchetto `a.b.c` deve essere dichiarata nel file `ClasseA.java` che si trova all'interno del path `a/b/c/` nella cartella dei sorgenti.

Inoltre, per poter essere trovata, la classe deve essere dichiarata `public`.

Lo stato di un oggetto

Domanda: Chi può modificare lo stato di un oggetto?

-Risposta_: Solo l'oggetto!

Perchè?

Per garantire che la struttura delle informazioni interne all'oggetto rimangano sempre coerenti con le assunzioni fatte durante la fase di progettazione!

Lo stato di un oggetto

La regola generale nella definizione di una classe è:

- usare solo campi `private` o `protected`;
- fornire metodi *getter* per i campi a cui vogliamo dare accesso in *lettura*;
- fornire metodi *setter* per i campi a cui vogliamo dare accesso in *scrittura*.

Anche internamente ad una classe dovremmo sempre usare i metodi *get* e *set* per modificare i valori dei campi!

Questo approccio ci permette di controllare:

- chi più accedere alle informazioni;
- la *correttezza* dei dati inseriti.

Lo stato di un oggetto

Quando progettiamo una classe, infatti, abbiamo in mente una serie di funzionalità che questa dovrà fornire.

I metodi inclusi nella classe forniranno un'implementazione per le funzionalità che la classe mette a disposizione.

Il funzionamento di un metodo dipende da:

- stato dell'oggetto;
- parametri passati al metodo.

Assunzioni su stato e parametri...

Quando sviluppiamo del codice, quindi, facciamo *sempre* delle assunzioni:

- sullo stato di un oggetto;
- sui parametri che un dato metodo riceve;
- sul risultato dell'esecuzione del metodo.

Le assunzioni fatte intorno ad una classe sono fondamentali per poter garantire il corretto funzionamento.

Quando usiamo un oggetto, quindi, dovremmo conoscere tali assunzioni!

Alcune domande:

- quali tipologie di assunzioni esistono?
- come possiamo esprimerle?

Assunzioni su stato e parametri...

Possiamo individuare due tipologie di assunzioni:

- le *invarianti di classe*;
- i *contratti* dei metodi.

Le invarianti di classe

Una *invariante* è una proprietà che lo stato di un'oggetto verificherà per tutta la sua esistenza.

Quando sviluppiamo il metodo di una classe potremo:

- *assumere* che l'invariante sia rispettata al momento dell'invocazione;
- *garantire* che l'invariante sarà rispettata dopo l'esecuzione del metodo.

I contratti dei metodi

Possiamo associare ad ogni metodo un *contratto*.

Questo rappresenta una *garanzia* sull'effetto derivante dall'esecuzione di un metodo quando questo viene invocato con parametri che *soddisfano* certe condizioni.

Si parla di *contratto* perché...

- chi usa l'oggetto *si impegna* a passare i *parametri corretti*;
- chi implementa il metodo *si impegna* a garantire i *risultati attesi*.

Esempio: i metodi di Object

Possiamo trovare un esempio esempio di *contratto* nei metodi della classe Object:

- `equals(Object o): bool`
- `hashCode(): int`
- `toString(): String`

Questi metodi sono messi a disposizione da oggetto Java e possono essere *riscritti* in ogni classe!

Il contratto di equals

Il *contratto* di equals richiede che il metodo deve implementare una *relazione di equivalenza* tra riferimenti ad oggetti che non sono null:

- E' riflessiva: per ogni riferimento non-null x, `x.equals(x)` dovrebbe restituire sempre true.
- E' simmetrica: per ogni coppia di riferimenti non-null x e y, `x.equals(y)` deve restituire true se e solo se `y.equals(x)` restituisce true.
- E' transitiva: per ogni tripla di riferimenti non-null x, y, e z, se `x.equals(y)` restituisce true e `y.equals(z)` restituisce true, allora `x.equals(z)` dovrebbe restituire true.
- E' consistente: per ogni coppia di riferimenti non-null x e y, più invocazioni di `x.equals(y)` dovranno restituire consistentemente il valore true o false se durante le operazioni di confronto lo stato dei due oggetti non viene modificato.
- Per ogni riferimento non-null x, `x.equals(null)` dovrebbe restituire false.

Il metodo hashCode

Un *hash code* è un intero derivato dallo stato di un oggetto.

Gli hash code dovrebbero essere ottenuti in modo tale che se *x* e *y* non sono uguali allora *x.hashCode()* e *y.hashCode()* dovrebbero restituire valori differenti *con alta probabilità*.

La classe `Objects` mette a disposizione dei metodi di utilità per il calcolo degli hash code:

```
public int hashCode() {
    return Objects.hash(description, price);
}
```

Il contratto di hashCode

Il contratto di hashCode richiede che:

- Ogni volta che il metodo è invocato durante l'esecuzione il risultato del metodo hashCode restituisce in modo consistente lo stesso valore intero.
- Se due oggetti sono uguali rispetto al metodo equals, allora il metodo hashCode invocato sui due oggetti dovrà produrre lo stesso risultato.

Come specifichiamo i contratti?

Purtroppo Java, come la quasi totalità dei linguaggi di programmazione, non consente di specificare esplicitamente i contratti.

Per questa ragione tali informazioni vengono normalmente aggiunte nel *javadoc* di una classe.

Sia le *invarianti di classe* che i *contratti di un metodo* possono essere descritti:

- attraverso frasi nel linguaggio naturale;
- mediante *predicati del prim'ordine* le cui variabili sono i campi dell'oggetto, i parametri ed il valore di ritorno.

La programmazione difensiva

Nella scrittura di un metodo ci troviamo quindi davanti al problema di verificare se, chi sta invocando il nostro metodo, stia rispettando o meno il *contratto*.

Con il termine *programmazione difensiva* si intende quello stile di coding dove all'inizio di ogni metodo vengono inseriti dei controlli per verificare se le condizioni di uso del metodo sono rispettate.

Tali controlli possono essere realizzati mediante:

- il costrutto `if then else`;
- il costrutto `assert`,

La programmazione difensiva

L'uso estensivo della programmazione difensiva incide significativamente sulla qualità del codice.

I controlli devono essere fatti *solo* nei metodi *pubblici*.

Per tale ragione è *buona norma* limitare il numero di metodi pubblici forniti da una classe.

Violazione dei contratti

Cosa dobbiamo fare, però, quando un metodo identifica una *violazione del contratto*?

Una possibilità è quella di restituire un *codice di errore*.

Questo è l'approccio usato in molti linguaggi di programmazione come, ad esempio, il C.

Questa soluzione è però insoddisfacente per diverse ragioni

- il valore di ritorno di un metodo è già usato per scopi diversi;
- il chiamante deve gestire l'errore ed, eventualmente, restituire un errore diverso al suo contesto;
- non è possibile verificare *staticamente* se tutti gli errori sono stati gestiti adeguatamente!

Le eccezioni

Per questa ragione in Java, come in altri linguaggi di programmazione moderni, gli errori vengono gestiti per mezzo delle *eccezioni*.

Un metodo segnala un *problema serio* andando a *sollevare una eccezione*.

Uno dei metodi nella *catena delle chiamate* sarà deputato a *gestire* tale eccezione.

Verificare i contratti

Come facciamo a garantire che una determinata classe soddisfi tutti i suoi contratti?

Scoprirete con il proseguire degli studi che non esiste un modo per garantire che *ogni tipologia di contratto* sia verificata.

Un contratto potrebbe essere del tipo: “il metodo termina”!

Esistono comunque due approcci:

- l'*analisi statica del codice* ed il *model checking*
 - Java Path Finder
 - Java Modelling Language
- l'uso dei test.

Javadoc

Javadoc è uno strumento distribuito con il JDK per generare, a partire del codice, la documentazione relativa ad una libreria/applicazione in formato HTML.

La documentazione generata verrà poi utilizzata dai vari IDE per mostrare il funzionamento dei metodi.

Il formato di Javadoc si basa su *commenti* opportunamente arricchiti per fornire informazioni.

La documentazione della libreria standard Java è costruita mediante [Javadoc](#).

Javadoc

Un commento Javadoc è un commento che inizia con un doppio *asterisco*:

```
/** Commento Javadoc */
```

Il commento javadoc deve essere inserito immediatamente prima dell'elemento che si vuole commentare.

All'interno del blocco di commento è possibile inserire dei *tag* che aggiungono *semantica* alle informazioni.

```
/**  
 * Le istanze di questa classe vengono utilizzate  
 * per rappresentare le coordinate di una casella.  
 *  
 * @author Michele Loreti  
 */  
public class Position {
```

Javadoc: i metodi

Il Javadoc di ogni metodo deve avere:

- una descrizione sommaria del significato del metodo (e del suo contratto);
- una descrizione del ruolo di ogni parametro;
- una descrizione del valore di ritorno;
- una descrizione delle eccezioni sollevate.

```
/**  
 * Esegue il movimento secondo la direzione passata come parametro.  
 * Restituisce {@code true} se la mossa è valida, {@code false} altrimenti.  
 *  
 * @param dir direzione del movimento  
 * @return {@code true} se la mossa è valida, {@code false} altrimenti.  
 */  
public boolean move(SlidingDirection dir) {
```

Javadoc: i tag

Tag	Descrizione
@author	Nome dello sviluppatore.
@deprecated	Indica che l'elemento potrà essere eliminato da una versione successiva del software.
@link	Crea un collegamento ipertestuale alla documentazione locale o a risorse esterna (tipicamente internet).
@param	Definisce i parametri di un metodo. Richiesto per ogni parametro.
@return	Indica i valori di ritorno di un metodo. Questo tag non va usato per metodi o costruttori che restituiscono void.
@see	Indica un'associazione a un altro metodo o classe.
@throws	Indica eccezioni lanciate da un metodo.

Javadoc

Maggiori dettagli sono disponibili sul sito [Oracle di Java](#).

Nello stesso link è possibile trovare le *linee guida* per la redazione della documentazione.

Alcune considerazioni:

- la documentazione deve essere aggiunta ad ogni elemento pubblico;
- è più conveniente scrivere la documentazione in inglese.

Un esempio...

Supponiamo di voler implementare un serie di voler implementare un *canale* di comunicazione.

Questo può essere utilizzato per inviare, o ricevere, un array di *byte*.

Dobbiamo...

- definire l'interfaccia che descrive un canale;
- fornire una (o più) implementazioni di un canale.

I canali...

Potremmo immaginare di avere due interfacce:

- `InputChannel`, per gestire la comunicazione in ingresso;
- `OutputChannel`, per gestire la comunicazione i uscita.

```
public interface InputChannel {  
    int available();  
    int receive(byte[] data);  
}  
  
public interface OutputChannel {  
    void send(byte[] data);  
}
```

Quali classi?

Come possiamo implementare tale interfaccia?

Ci sono più *opzioni*.

Una possibile opzione è quella di usare un *buffer circolare* per salvare i dati nel canale.

Quando necessario la dimensione del buffer viene *aumentata*.

Il buffer...

Un *buffer* è utilizzato per memorizzare i dati ricevuti in un *array*.

Due indici vengono utilizzati per memorizzare

- il primo dato disponibile da leggere;
- la prima posizione vuota da utilizzare.

Se necessario la dimensione dell'array viene incrementata.

La classe Buffer...

```
public class Buffer {  
    public final static int DEFAULT_INITIAL_SIZE;  
    private byte[] buffer;  
    private int firstDataElement;  
    private int firstEmptyElement;  
    public Buffer() {  
        this(DEFAULT_INITIAL_SIZE);  
    }  
    public BufferedChannel(int initialSize) {  
        this.buffer = new byte[initialSize];  
        this.firstDataElement = 0;  
        this.firstEmptyElement = 0;  
    }  
    private byte get(int i) {  
        return buffer[i%buffer.length];  
    }  
    private byte get() {  
        return get(this.firstDataElement++);  
    }  
    public int available() {  
        return this.firstEmptyElement-this.firstDataElement;  
    }  
}
```

La classe Buffer...

```
public int get(byte[] data) {
    int counter;
    for(counter = 0;
        (counter<data.length)&&(firstDataElement<firstEmptyElement));
        counter++) {
        data[counter] = get();
    }
    return counter;
}

private void set(int i, byte b) {
    this.buffer[i%buffer.length] = b;
}

private void set(byte b) {
    set(this.firstEmptyElement++, b);
}

public void put(byte[] data) {
    extendIfNeeded(data.length);
    for(int i=0; i<data.length; i++) {
        set(data[i]);
    }
}
```

La classe Buffer...

```
private void extendIfNeeded(int size) {
    if (available() + size > buffer.length) {
        extend(2 * (buffer.length + size))
    }
}
private void extend(int newSize) {
    int elements = available();
    int counter = 0;
    byte[] newBuffer = new byte[newSize];
    for(int i=firstDataElement; i<firstEmptyElement; i++) {
        newBuffer[counter++] = get(i);
    }
    buffer = newBuffer;
    firstDataElement = 0;
    firstEmptyElement = counter;
}
}
```

Canale di input...

```
public class BufferedInputChannel implements InputChannel {  
    private final Buffer buffer;  
  
    public BufferedInputChannel(Buffer buffer) {  
        this.buffer = buffer;  
    }  
  
    @Override  
    public int available() {  
        return buffer.size;  
    }  
  
    @Override  
    public int receive(byte[] data) {  
        return buffer.get(data);  
    }  
}
```

Canale di output...

```
public class BufferedOutputChannel implements OutputChannel {  
    private final Buffer buffer;  
  
    protected BufferedOutputChannel(Buffer buffer) {  
        this.buffer = buffer;  
    }  
  
    @Override  
    public void send(byte[] data) {  
        this.buffer.put(data);  
    }  
}
```

Altre funzionalità...

Le classi `BufferedInputChannel` e `BufferedOutputChannel` ci permettono di implementare canali per scambiare singoli byte.

Supponiamo ora di voler implementare altre tipologie di canali che possano essere usato per inviare *dati*.

Come possiamo fare? Possiamo *riusare* quanto già fatto?

Ovviamente sì... dobbiamo *estendere* le classi!

Estendere una classe...

Consideriamo una nuova classe DataOutputChannel che mantiene alcune delle funzionalità di BufferedInputChanel ma che introduce nuove funzionalità.

Ad esempio la possibilità di inviare dati come int, byte, float, double o String.

```
public class DataOutputChannel extends BufferedOutputChannel {  
    // Nuovi campi...  
    // Nuovi metodi...  
}
```

Estendere una classe...

La keyword `extends` viene utilizzata per definire una nuova classe che deriva da una esistente.

La classe esistente è chiamata *superclasse* mentre quella che estende è chiamata *sottoclasse*.

Una sottoclasse ha più funzionalità delle sue superclassi!

Estendere una classe...

I metodi definiti nella classe derivata potranno usare i metodi della classe base.

```
public void send(int v) {
    byte[] bytes = new byte[Integer.BYTES];
    for(int i = 0; i<bytes.length; i++) {
        bytes[i] = (byte) (value & 0xFF);
        value = value >> 8;
    }
    send(bytes);
}
```

L'implementazione dei seguenti metodi è lasciata per esercizio:

- void send(long v)
- void send(float f)
- void send(double d)
- void send(String str)

I costruttori...

Una sottoclasse potrà definire dei costruttori.

E' opportuno che questi invochino i costruttori della *classe base* per garantire l'inizializzazione dei campi privati.

```
public DataOutputChannel(Buffer buffer) {  
    super(buffer);  
}
```

Se la classe base mette a disposizione un *costruttore di default* l'invocazione del costruttore può essere omessa.

Se un *costruttore di default* non è disponibile, la classe derivata deve definire necessariamente un costruttore ed invocare il costruttore della classe base.

Assegnamento...

E' sempre possibile assegnare ad una *variabile* di tipo *classe base* un oggetto istanza della *classe derivata*:

```
BufferedOutputChannel channel = new DataOutputChannel();
```

Possiamo notare quindi che ogni *variabile* (o *campo*) in Java ha due tipi:

- il *tipo statico*, che è quello assegnato dal compilatore al momento della dichiarazione;
- il *tipo dinamico*, che è il tipo degli oggetti a cui fa riferimento la variabile.

Metodi sovraccaricati...

Possiamo notare come nella classe `DataOutputChannel` siano presenti più metodi con il medesimo nome ma con *signature* differente.

La *signature* di un metodo è costituita da:

- *nome del metodo*;
- *numero e tipo* dei parametri.

In questo caso diremo che tali metodi sono *sovraccaricati* (o *overloaded*).

Alcune considerazioni...

Consideriamo, ora, di voler implementare un canale *cifrato*.

Ossia un canale a cui alle informazioni inviate vengono applicate delle *codifiche*.

Come possiamo procedere?

Alcune considerazioni...

Potremmo, ad esempio, andare a definire una nuova *sottoclasse* chiamata, ad esempio, *SecretOutputChannel*.

Come consideriamo tale soluzione?

Tale soluzione non è *adeguata*!

Comporre oggetti...

L'ereditarietà non è sempre la soluzione più adatta.

E' molto più conveniente, talvolta, usare la *composizione* di oggetti.

Nel nostro esempio dei *canali*, quindi, sarebbe più conveniente usare un approccio più *composizionale*.

Comporre oggetti...

Possiamo considerare, ad esempio, la classe `FilteredOutputChannel` che permette di *operare* su un altro canale.

La classe `FilteredOutputChannel`...

- *implementa* l'interfaccia `OutputChannel`;
- *usa (per delega)* un oggetto istanza di `OutputChannel`.

```
public class FilteredOutputChannel implements OutputChannel {  
    private final OutputChannel output;  
  
    public FilteredOutputChannel(OutputChannel output) {  
        this.output = output;  
    }  
  
    @Override  
    public void send(byte[] data) {  
        output.send(data);  
    }  
}
```

DataCipher...

Consideriamo l'interfaccia DataCipher che possiamo usare per codificare/decodificare dei dati:

```
public interface DataCipher {  
    byte[] encode(byte[] data);  
    byte[] decode(byte[] data);  
}
```

CipherOutputChannel...

```
public class CipherOutputChannel extends FilteredOutputChannel {  
    private final DataCipher cipher;  
  
    public CipherOutputChannel(OutputChannel output, DataCipher cipher) {  
        super(output);  
        this.cipher = cipher;  
    }  
  
    @Override  
    public void send(byte[] data) {  
        super.send(cipher.encode(data));  
    }  
}
```

La sovrascrittura

Possiamo notare come nella classe `CipherOutputStream` il metodo `send(byte[] data)` venga *sovraffratto*.

Un metodo in una classe base viene *sovraffratto* da una classe *derivata* quando quest'ultima mette a disposizione un metodo con la medesima *signature*.

Un semplice esempio...

Consideriamo la seguente porzione di codice:

```
public void doSend(OutputChannel oc, DataCipher cipher, String message) {  
    OutputChannel output = new DataOutputChannel(new CipherOutputChannel(oc));  
    output.send(message);  
}
```

Cosa accade quando viene invocato il metodo doSend?

Il *dynamic lookup*

Ad ogni classe Java viene associata una *tabella dei metodi*.

I metodi in questa tabella vengono *indicizzati* sulla base della loro signature.

Di fatto, quando usiamo un metodo, il compilatore individua la specifica *signature* che stiamo utilizzando.

Tale selezione viene fatta per mezzo del *tipo statico* delle espressioni.

Il *dynamic lookup*

Quando un oggetto ricece un'invocazione il metodo da invocare viene *selezionato* sulla base della signature utilizzata.

Tale *ricerca* avviene usando il *tipo dinamico* del ricevente.

Questo meccanismo consente di implementare complessi comportamenti.

Esempio...

Consideriamo le seguenti classe:

```
class ClasseA {  
  
    public void m1( ) {  
        System.out.println("ClasseA->m1()");  
        m2();  
    }  
  
    public void m2( ) {  
        System.out.println("ClasseA->m2()");  
    }  
  
}  
  
class ClasseB extends ClasseA {  
    public void m2( ) {  
        System.out.println("ClasseB->m2()");  
    }  
}
```

Cosa accade eseguendo la seguente porsione di codice?

```
ClasseA a = new ClasseB();  
a.m1();
```

Verrà stampato a video

```
ClasseA->m1();  
ClasseB->m2();
```

Esempio...

Consideriamo le seguenti classe:

```
class ClasseA {  
  
    public void m1( ) {  
        System.out.println("ClasseA->m1()");  
        m2();  
    }  
  
    private void m2( ) { //<- ATTENZIONE!!!!  
        System.out.println("ClasseA->m2()");  
    }  
  
}  
  
class ClasseB extends ClasseA {  
    public void m2( ) {  
        System.out.println("ClasseB->m2()");  
    }  
}
```

Cosa accade eseguendo la seguente porsione di codice?

```
ClasseA a = new ClasseB();  
a.m1();
```

Verrà stampato a video

```
ClasseA->m1();  
ClasseA->m2();
```

Esempio...

Consideriamo le seguenti classe:

```
class ClasseA {  
  
    public void m1( ) {  
        System.out.println("ClasseA->m1()");  
        m2();  
    }  
  
    protected void m2( ) { //<- ATTENZIONE!!!!  
        System.out.println("ClasseA->m2()");  
    }  
  
}  
  
class ClasseB extends ClasseA {  
    public void m2( ) {  
        System.out.println("ClasseB->m2()");  
    }  
}
```

Cosa accade eseguendo la seguente porsione di codice?

```
ClasseA a = new ClasseB();  
a.m1();
```

Verrà stampato a video

```
ClasseA->m1();  
ClasseB->m2();
```

Esempio...

Consideriamo ora la seguente definizione:

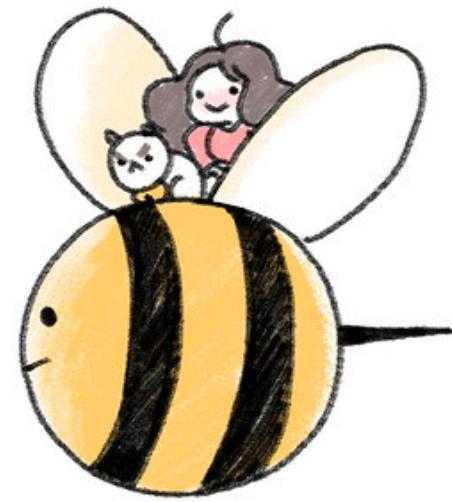
```
class ClasseC {  
    public void m( ClasseA a ) {  
        System.out.println("ClasseC->m()");  
    }  
}  
  
class ClasseD extends ClasseC {  
    public void m( ClasseB b ) {  
        System.out.println("ClasseD->m()");  
    }  
}
```

Cosa accade eseguendo la seguente porzione di codice?

```
ClasseA c = new ClasseB();  
ClasseC d = new ClasseD();  
d.m(c);
```

Verrà stampato a video

```
ClasseC->m();
```



TO
BE
CONTINUED!