Nascondere le implementazioni

File e classi Java …

Un file che contiene codice sorgente Java, detto comunemente *unità di compilazione* (talvolta anche *unità di traduzione*), deve sempre avere estensione *.java*

Esso può contenere più definizioni di classi ma solo una può essere *public*. Tale classe deve avere lo stesso nome del file (rispettando le maiuscole ma escludendo l'estensione .java). Pertanto se il file contiene il programma principale di un'applicazione, questo dovrà necessariamente essere il main della classe public.

Le altre classi sono **nascoste** al mondo esterno perché si intendono solo di supporto alla classe *public*.

… File e classi Java.

Quando si compila un file Java si otterrà in output un file (contenente esattamente lo stesso nome ma con estensione .*class*) per ogni classe presente nel file .*java*.

Una programma funzionante è praticamente un insieme di file .*class* che possono essere "impacchettati" e eventualmente compressi in un JAR file (usando l'applicazione ***jar*** fornita insieme al JDK).

Sarà compito dell'interprete Java occuparsi del ritrovamento, caricamento e interpretazione di questi file.

Una sola classe *public* per file. Perché?

Il fatto che solo una classe sia pubblica, permette al programmatore di modificare tutte le classi di supporto, senza che l'utente della classe pubblica debba modificare il proprio codice.

In questo modo si soddisfa uno dei principi chiave della progettazione object-oriented, quello di "separare ciò che muta nel tempo da ciò che resta invariato".

Creazione dei package…

Come già osservato, per raggruppare delle classi Java è possibile ricorrere alla loro inclusione in un unico file.

In alternativa, Java permette di raggruppare delle classi mediante clausola *package*.

Ad esempio, per raggruppare un insieme di classi in un package denominato `mypackage` basterà inserire la clausola:

```
package mypackage;
```

all'inizio di ogni file Java contenente le classi da raggruppare.

N.B.: Per convenzione, i nomi dei package si definiscono con le lettere minuscole.

Package predefiniti della JDK

Package	***Fornisce il supporto per***
`java.applet`	creare applet
`java.swing`	creare interfacce grafiche
`java.io`	eseguire funzioni di I/O
`java.lang`	funzioni generiche
`java.math`	eseguire calcoli matematici
`java.net`	comunicare attraverso una rete
`java.rmi`	creare applicazioni distribuite
`java.security`	imporre restrizioni sulla sicurezza
`java.sql`	interagire con basi di dati in SQL
`java.text`	formattare il testo per la visualizzazione
`java.util`	Servizi generali

La clausola `import`...

Per importare una intera libreria predefinita di Java si usa la clausola import.

Esempio: `import java.util.*;`

Volendo invece importare solo una classe della libreria, ad esempio *ArrayList*, si dovrebbe scrivere

```
import java.util.ArrayList;
```

Se all'interno del codice ci si vuole riferire a tale classe, occorre specificare solo `ArrayList` se si è precedentemente importato il *package* altrimenti si deve indicare l'intero percorso `java.util.ArrayList;`

…La clausola `import`

Nota: le classi del package `java.lang` sono automaticamente rese disponibili durante la scrittura di un programma, cioè esse sono importate automaticamente.

Lo spazio dei nomi delle classi

La ragione alla base della scelta di importare i *package* è fornire un meccanismo di gestione dello "*spazio dei nomi delle classi*".

Sappiamo che i nomi di tutti i membri delle classi sono isolati gli uni dagli altri. Un metodo *f()* della classe *A* non entrerà mai in conflitto con un metodo *f()* (avente la stessa lista di argomenti) della classe B. A distinguerli sarà l'oggetto sul quale sono invocati.

Ma come ottenere un risultato analogo per i nomi delle classi?

Questo problema è particolarmente avvertito per quelle applicazioni che possono essere ottenute via Web. Infatti può accadere facilmente che si abbia un "conflitto di nomi" fra classi prese automaticamente da Internet e classi disponibili localmente.

La soluzione sta appunto nel concetto di package.

Lo spazio dei nomi delle classi

Per esempio, si supponga di aver creato un package di un solo file **MyClass.java**. Questo significa che ci deve essere un'unica classe **public** in quel file e che il nome della classe dev'essere **MyClass**:

```
package mypackage;
public class MyClass {
// . . .
```

Se qualcuno volesse usare **MyClass** nel proprio programma potrebbe usare il nome in modo completamente specificato

```
mypackage.MyClass m = new mypackage.MyClass();
```

La clausola **import** può rendere ciò più semplice:

```
import mypackage.*;
// . . .
MyClass m = new MyClass();
```

Lo spazio dei nomi delle classi

Anche se **MyClass** fosse stata definita nel contesto di un'altra applicazione presa da Web, essa non entrerebbe in conflitto con quella importata in **mypackage**.

Quindi le clausole **package** e **import** permettono al progettista di librerie di suddividere il singolo spazio globale dei nomi in modo da non avere conflitti, indipendentemente da quante persone mettono a disposizione classi Java su Internet.

Package e cartelle

Poiché un *package* si compone di più file *.class* è preferibile organizzarli in una cartella distinta, sfruttando l'organizzazione gerarchica in cartelle disponibile nei moderni sistemi operativi.

Questa soluzione risolve altri due problemi:

1. la creazione di nomi di package unici (poiché associati alla cartella) e
2. il ritrovamento di quelle classi che potrebbero essere disperse in qualche cartella imprecisata.

Questo è ottenuto codificando il percorso della locazione del file *.class* nel nome del *package*.

Package e cartelle

L'interprete Java procede nell'ordine seguente:

1. trova la variabile d'ambiente CLASSPATH (impostata manualmente oppure attraverso i tool Java-based sulla macchina). CLASSPATH contiene una o più cartelle che sono usate come root per la ricerca dei file *.class*
2. partendo dalla root, prende il nome del *package* e sostituisce ogni punto con uno *slash* per generare un nome di percorso per la root del CLASSPATH (in modo tale che ***package mypackage.util*** diventi ***mypackage\util*** dipendentemente dal sistema operativo)
3. Concatena la stringa ottenuta ai percorsi contenuti nel CLASSPATH dove accede per cercare il file *.class* con il nome corrispondente alla classe che si sta cercando di creare.

Nascondere le implementazioni

Abbiamo appena visto che per realizzare una libreria si può ricorrere al costrutto package del Java.

Ma come si realizza effettivamente il concetto di "information hiding"?

Per chiarire questo aspetto occorre spiegare il concetto di modificatore (o specificatore) di accesso, che si applica tanto alle proprietà di una classe (dati e funzioni membro) quanto alle classi stesse.

I modificatori di accesso in Java

I modificatori di accesso in Java sono i seguenti:

- *public*
- *private*
- *friendly* (o *package access*)
- *protected*

Questi modificatori vanno inseriti subito prima della definizione di ogni membro della classe (sia esso funzione o dato) in modo che ciascun modificatore controlla l'accesso ad un solo membro.

Questo è diverso da quello che accade in C++ in cui un modificatore controllava l'accesso di tutti i membri che seguivano fino a che non si indicava un altro modificatore diverso.

Il modificatore di accesso "friendly"

Se non si indica alcun modificatore per un determinato membro, Java gli assegna un accesso di default che non è rappresentato da alcuna parola chiave sebbene ci si riferisce ad esso come "*friendly*" o "*package access*".

Ciò significa che tutte le classi del package corrente hanno accesso al membro *friendly* che invece appare come *private* alle classi fuori del package.

…Il modificatore di accesso "friendly".

Di conseguenza si dice che questi elementi hanno un *package access*.

L'accesso *friendly* permette di raggruppare insieme delle classi correlate in un package cosicché possano facilmente interagire tra loro.

In Java il codice di un package non può autonomamente accedere a quello di un altro.

L'accesso ad un membro.

Il solo modo di concedere l'accesso ad un membro è:

- rendere il membro *public* in modo che tutti da qualsiasi parte possano accedervi;
- rendere il membro *"friendly"* non specificando alcun modificatore e inserire le altre classi nello stesso package in modo tale che queste possano accedervi;
- una sottoclasse può accedere, oltre ai membri pubblici, ai membri *protected* delle superclassi e ai "friendly" di un'altra classe solo se le due classi sono nello stesso package;
- Nel caso di membri dato, fornendo dei *metodi "accessor/mutator"* (noti anche come *metodi "get/set*) che leggono e cambiano il valore.

Il package di default

Se i file sorgente Java non importano alcun package particolare e risiedono nella stessa cartella sono implicitamente trattati come un "*package di default*" per quella cartella e pertanto sono *friendly* tra loro (se non viene specificato alcun modificatore di accesso).

Esempio

Supponendo che i file distinti *Cake.java* e *Pie.java* siano nella stessa cartella, sarà possibile eseguire correttamente il metodo *main* di *Cake.java*

Package di default

Il package di default

```
//Cake.java
// Accede a una classe in una unita' di compilazione.
class Cake {
public static void main(String[] args) {
      Pie x = new Pie();
      x.f();
      }
}
//Pie.java
//L'altra classe.
class Pie {
void f() {
      System.out.println("Pie.f()");
}
}
```

Il modificatore di accesso *public*

Se un membro della classe è preceduto dalla parola chiave *public*, significa che è accessibile a chiunque.

Esempio

Modificatore di accesso public

Il modificatore di accesso *public*

```
// Crea una libreria.
package c05.dessert;
public class Cookie {
public Cookie() {
System.out.println("Cookie constructor");
}
void bite() { System.out.println("bite"); }
}
// Usa la libreria.
import c05.dessert.*;
public class Dinner {
public Dinner() {
System.out.println("Dinner constructor");
}
public static void main(String[] args) {
Cookie x = new Cookie();
//! x.bite(); // Non si puo' accedere
}}
```

Si fa notare che il metodo *bite()* della classe *Cookie* non è accessibile perché è stato dichiarato *friendly* (quindi è accessibile solo tra le classi del package) e non *public*.

Il modificatore di accesso *private*…

La parola chiave *private* è usata per rendere il membro accessibile solo dai metodi della classe in cui è stato dichiarato.

Visto che un package potrebbe essere creato da più programmatori, rendendo alcuni membri (metodi o attributi) della propria classe *private* si ha la certezza che le eventuali modifiche che potrebbero riguardare la classe non andranno a coinvolgere le altre classi presenti nel package.

Esempio

Modificatore di accesso private

Il modificatore di accesso *private*...

```
// Dimostrazione di dichiarazioni private.
class Sundae {
private Sundae() {}
static Sundae makeASundae() {
return new Sundae();
}
}
public class IceCream {
public static void main(String[] args) {
//! Sundae x = new Sundae();
Sundae x = Sundae.makeASundae();
}
}
```

…Il modificatore di accesso *private*…

Potrebbero essere resi *private* tutti quei metodi che fungono solo da supporto per quella classe in modo tale da impedirne l'uso accidentale all'interno del package e di conseguenza proibire la modifica o l'eliminazione del metodo.

Discorso analogo vale per gli attributi *private* delle classi.

Se non si hanno particolari esigenze, si dovrebbero rendere *private* tutti i membri dato della classe.

L'uso di *private* è importante specialmente nel *multithreading*.

Il modificatore di accesso *protected*…

Se un metodo o un attributo sono preceduti dalla parola chiave *protected*, significa che tali caratteristiche sono accessibili sia nella classe in cui sono stati dichiarati che in una qualsiasi sottoclasse della stessa in qualunque package sia, nelle classi che sono nello stesso package della classe in cui è stata dichiarata.

Per esprimere la relazione di ereditarietà tra una sottoclasse e la classe base, Java introduce la parola chiave *extends*.

Pertanto per indicare che una classe eredita metodi e attributi da una classe esistente si dirà che la nuova classe *estende* una classe esistente.

Il modificatore di accesso *protected*…

```
// La parola chiave protected.
package example;
import java.util.*;
public class A {
private int i;
protected int read() { return i; }
protected void set(int ii) { i = ii; }
public A(int ii) { i = ii; }
public int value(int m) { return m*i; }
}
package example2;
import example.A;
public class B extends A {
private int j;
public B(int jj) { super(jj); j = jj; }
public void change(int x) { set(x); }
}
```

…Il modificatore di accesso *protected*…

Se si crea un nuovo package e si eredita da una classe in un altro package, i soli membri accessibili sono quelli pubblici del pacchetto originale e quelli *protected* della classe base.

…Il modificatore di accesso *protected*…

```
// Non accede a membri friendly in altre classi.
package example;
import c05.dessert.*;
public class ChocolateChip extends Cookie {
public ChocolateChip() {
System.out.println("ChocolateChip constructor");
}

void exampleMethod{
//bite(); // non accede a bite
}
}
```

Se consideriamo il file *Cookie.java*, la classe precedente non può accedere al membro friendly

…Il modificatore di accesso *protected.*

Si fa notare che il metodo *bite()* pur essendo stato ereditato dalla sottoclasse *ChocolateChip*, non è da questa accessibile perché è friendly solo per le classi che appartengono allo stesso package del file *Cookie.java*.

Rendendo il metodo in questione *public* nella classe *Cookie* lo si renderebbe accessibile a tutti.

…Il modificatore di accesso *protected.*

La soluzione è quella di rendere il metodo *protected* andando a modificare la classe *Cookie* nel modo seguente:

```
public class Cookie {
public Cookie() {
System.out.println("Cookie constructor");
}
protected void bite() {
    System.out.println("bite");
}
}
```

Il modificatore di accesso *final*

I membri preceduti dalla parola chiave *final* non possono essere estesi ovvero non possono essere ridefiniti nelle sottoclassi.

Accesso alle classi…

In Java i modificatori di accesso sono utilizzati anche per determinare quali classi nella libreria sono accessibili agli utenti di quella libreria.

Poiché una unità di compilazione (file) può appartenere ad un singolo package, tutte le classi in una singola unita di compilazione sono automaticamente *friendly* tra loro.

Se si vuole che la classe sia accessibile al programmatore bisogna dichiararla *public* nel seguente modo:

```
public class MyClass {
//body   }
```

…Accesso alle classi.

Si nota che una classe non può essere dichiarata *private* o *protected*.

Se si vuole che nessuno possa accedere alla classe si possono rendere tutti i costruttori *private*

Esempio

Accesso alle classi

Esempio

```
//Classe privata mediante costruttori privati

class Soup {
private Soup() {}

// (1) Permette la creaz. mediante un metodo
//     di classe
public static Soup makeSoup() {
return new Soup();
}

// (2) Crea un oggetto e restituisce un
// riferimento su richiesta.
// Un esempio di classe “singoletto”
private static Soup ps1 = new Soup();

public static Soup access() {
return ps1;
}
public void f() {}
}
```

```
class Sandwich {
// Uso di Lunch
void f() { new Lunch(); }
}
```

```
// Solo una classe pubblica ammessa nel file:
public class Lunch {
void test() {
// Can't do this! Private constructor:
//! Soup priv1 = new Soup();
Soup priv2 = Soup.makeSoup();
Sandwich f1 = new Sandwich();
Soup.access().f();
}
}
```

Le interfacce…

Spesso ci si riferisce al controllo dell'accesso come *implementation hiding*.

Il *wrapping* di dati e metodi nelle classi in aggiunta all'*implementation hiding* viene chiamato *incapsulamento*.

Il risultato di questa operazione è un tipo di dato fornito di caratteristiche e comportamento.

Il controllo dell'accesso permette essenzialmente di:

- stabilire cosa possono e non possono fare i programmatori utente;
- separare l'interfaccia dall'implementazione;

…Le interfacce.

Attraverso la parola chiave ***interface*** Java permette di astrarre completamente una interfaccia dalla relativa implementazione specificando che cosa una classe deve fare ma non come deve farlo. Esse forniscono dei "template" degli schemi per la definizione delle classi.

Le interfacce sono simili alle classi ma privi di variabili di istanza e i loro metodi sono privi di corpo.

Per implementare una interfaccia, una classe deve creare l'insieme dei metodi definiti in tale interfaccia con ampia libertà di implementazione.

Le interfacce aggiungono gran parte delle funzionalità necessarie per molte applicazioni, che solitamente utilizzerebbero invece l'ereditarietà multipla in C++ .

Esempio

```
access interface InterfaceName {
    returnType NameOfMethod1(argument-list);
    returnType NameOfMethod2(argument-list);
    //……
    returnType NameOfMethodN(argument-list);
    final static type finlNameVar1 = value;
    final static type finlNameVar1 = value;
    //……
    final static type finlNameVarN = value;
}
```

Definizione di una interfaccia…

access può essere public oppure può essere l'accesso default per cui l'interfaccia è disponibile solo nel pacchetto in cui è definita.

I metodi specificati in una interfaccia sono sempre considerati public, e quindi vanno implementati come public.

I metodi dichiarati sono privi di corpo ovvero sono *astratti*. Toccherà alla classe che implementerà l'interfaccia, definire i rispettivi metodi.

Le variabili possono essere indicate nella dichiarazione di interfaccia e sono implicitamente *final* e *static* (non possono essere modificate dalla classe di implementazione) e devono essere inizializzate ad un valore costante.

…Definizione di una interfaccia.

Se l'interfaccia viene dichiarata *public* allora anche tutti i metodi e le variabili sono considerate implicitamente *public*.

Esempio

```
public interface drive {
void firstgear();
void secondgear();
void thirdgear();
}
```

Implementazione delle interfacce…

Affinché una o più classi possano implementare una interfaccia, Java mette a disposizione la parola chiave *implements*.

La sintassi generale è la seguente:

```
access class ClassName[extends BaseClass]
      [implements interface1[,interface2..]]
{
  // body
};
```

…Implementazione delle interfacce.

- I metodi che implementano una interfaccia devono essere dichiarati *public* e inoltre devono corrispondere esattamente a quelli dichiarati nella definizione dell'interfaccia.