Il riuso delle classi in Java

Il riuso delle classi in Java…

Una delle caratteristiche principali di Java è la possibilità di riusare il codice.

Anziché creare delle classi completamente nuove si possono usare delle classi esistenti che sono già implementate e testate.

Esistono due modi di procedere:

…Il riuso delle classi in Java.

- Nel primo modo semplicemente si creano oggetti per la classe esistente all'interno della nuova classe. Questa modalità è chiamata *composizione* perché la nuova classe è composta di oggetti delle classi esistenti in modo tale che si riusa la funzionalità del codice.
- Il secondo approccio si basa sul meccanismo di *ereditarietà*, una dei principi fondamentali della programmazione object-oriented.

Il meccanismo di composizione…

Tale meccanismo è molto diffuso. Come già detto consiste nell'inserire i riferimenti agli oggetti in una nuova classe.

Si può notare che mentre i tipi primitivi sono automaticamente inizializzati a 0, i riferimenti agli oggetti sono inizializzati a *null*.

Il compilatore, infatti, non deve creare degli oggetti di default altrimenti si potrebbero avere costi computazionali aggiuntivi indesiderati.

Il meccanismo di composizione…

```
// Riuso per composizione
class WaterSource {
private String s;
WaterSource() {
System.out.println("WaterSource()");
s = new String("Constructed");
}
public String toString() { return s; }
}
```

L'output di questo programma sarà il seguente:

```
valve1 = null
valve2 = null
valve3 = null
valve4 = null
i = 0
f = 0.0
source = null
```

```
public class SprinklerSystem {
private String valve1, valve2, valve3, valve4;
WaterSource source;
int i;
float f;
void print() {
System.out.println("valve1 = " + valve1);
System.out.println("valve2 = " + valve2);
System.out.println("valve3 = " + valve3);
System.out.println("valve4 = " + valve4);
System.out.println("i = " + i);
System.out.println("f = " + f);
System.out.println("source = " + source);
}
public static void main(String[] args) {
SprinklerSystem x = new SprinklerSystem();
x.print();
}
}
```

…Il meccanismo di composizione…

Se si vuole un riferimento inizializzato, si può fare nei seguenti punti:

- nel punto in cui gli oggetti sono definiti, quindi saranno sempre inizializzati immediatamente appena è richiamato il costruttore;
- nel costruttore della classe;
- giusto prima di quando si ha bisogno di usare l'oggetto. Questo approccio è spesso chiamato *lazy initialization*. Esso può ridurre i costi aggiuntivi in situazioni dove l'oggetto non deve essere creato molte volte.

…Il meccanismo di composizione.

```
// Inizializzazione nella
      costruzione.
class Soap {
private String s;
Soap() {
System.out.println("Soap()");
s = new String("Constructed");
}
public String toString() {
return s; }
}
```

Il programma avrà il seguente output:

```
Inside Bath()
Soap()
s1 = Happy
s2 = Happy
s3 = Joy
s4 = Joy
i = 47
toy = 3.14
castille = Constructed
```

```
public class Bath {
private String s1 = new String("Happy"), s2 = "Happy", s3, s4;
Soap castille;
int i; float toy;
Bath() {
System.out.println("Inside Bath()");
s3 = new String("Joy"); i = 47; toy = 3.14f;
castille = new Soap();
}
void print() {// Inizializzazione pigra
if(s4 == null) s4 = new String("Joy");
System.out.println("s1 = " + s1);
System.out.println("s2 = " + s2);
System.out.println("s3 = " + s3);
System.out.println("s4 = " + s4);
System.out.println("i = " + i);
System.out.println("toy = " + toy);
System.out.println("castille = " + castille); }

public static void main(String[] args) {
Bath b = new Bath();
b.print();
}
}
```

Il meccanismo di ereditarietà …

L'ereditarietà è uno dei concetti fondamentali di Java (e di tutti i linguaggi object-oriented in generale).

Esso entra in gioco ogni qualvolta che si crea una classe perché ogni classe in Java eredita dalla classe standard di root *Object* (a meno di esplicitarne una in particolare).

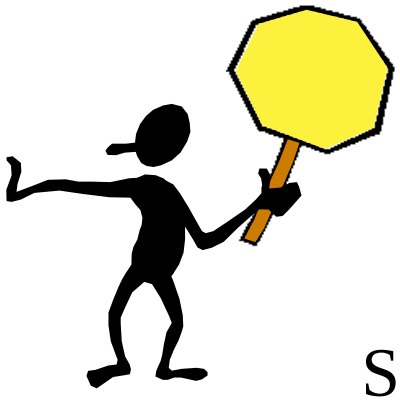
Per indicare che una classe eredita i membri da un'altra classe (chiamata *classe base*) si è già detto che Java mette a disposizione la parola chiave *extends*.

Il meccanismo di ereditarietà …

```
// Esempio di ereditarietà.
class Cleanser {
private String s = new String("Cleanser");
public void append(String a) { s += a; }
public void dilute() { append(" dilute()"); }
public void apply() { append(" apply()"); }
public void scrub() { append(" scrub()"); }
public void print() { System.out.println(s); }
public static void main(String[] args) {
Cleanser x = new Cleanser();
x.dilute(); x.apply(); x.scrub();
x.print();
}
}
```

```
public class Detergent extends Cleanser {
// Change a method:
public void scrub() {
append(" Detergent.scrub()");
super.scrub(); // Call base-class version
}
// Aggiunta di metodi
public void foam() { append(" foam()"); }
// Test della nuova classe
public static void main(String[] args) {
Detergent x = new Detergent();
x.dilute();
x.apply();
x.scrub();
x.foam();
x.print();
System.out.println("Testing base class:");
Cleanser.main(args);
}
}
```

…Il meccanismo di ereditarietà…



Si può notare che le classi *Cleanser* e *Detergent* hanno entrambe un main. Ciò può essere fatto per testare il codice di ogni singola classe. Quando da linea di comando si vorrà lanciare il metodo *main* della classe *Cleanser* si digiterà: ***java Cleanser.***

Se si vorrà invocare il metodo *main* della classe *Detergent* si digiterà: ***java Detergent***.

Inoltre si può notare che la classe *Detergent* ha impostato dei metodi nella sua interfaccia: *append()*, *dilute()*, *apply()*, *scrub()* e *print()*.

…Il meccanismo di ereditarietà…

Ciò è dovuto al fatto che *Detergent* è derivata da *Cleanser* (mediante la parola chiave *extends*) quindi prende automaticamente tutti i metodi della sua interfaccia anche se questi non sono definiti esplicitamente in *Detergent*.

Si può pensare alla *ereditarietà* come ad una sorta di *riuso della interfaccia*.

È possibile ridefinire (override) i metodi ereditati dalla classe base (vedi metodo *scrub()* dell'esempio). In tal caso se volessimo richiamare dalla sottoclasse il metodo *scrub()* definito nella classe base, dobbiamo utilizzare la parola chiave *super* che si riferisce alla superclasse da cui è stata derivata la classe corrente.

…Il meccanismo di ereditarietà.

È sempre possibile definire nella classe derivata dei nuovi membri, non esistenti nella classe base (vedi il metodo *foam()* dell'esempio).

Inizializzazione della classe di base…

Quando si crea un oggetto di una classe derivata, esso contiene al suo interno un *sotto-oggetto* della classe di base. Questo sotto-oggetto è identico a quello che si avrebbe se istanziassimo la classe base.

Pertanto è estremamente importante che il sotto-oggetto sia inizializzato correttamente attraverso l'invocazione del costruttore della classe base che possiede la conoscenza e privilegi opportuni per farlo.

Java inserisce automaticamente la chiamata al costruttore della classe base nel costruttore della classe derivata.

Esempio

```
// Chiamate di costruttori nell'ereditarietà
class Art {
Art() {
System.out.println("Art constructor");
}}

class Drawing extends Art {
Drawing() {
System.out.println("Drawing constructor");}}

public class Cartoon extends Drawing {
Cartoon() {
System.out.println("Cartoon constructor");}
public static void main(String[] args) {
Cartoon x = new Cartoon();
}}
```

L'output :

Art constructor
Drawing constructor
Cartoon constructor11

…Inizializzazione della classe di base.

Si può notare come le chiamate avvengano "dall'esterno verso l'interno" ovvero a partire dalla radice della gerarchia di ereditarietà fino ad arrivare alla chiamata dell'ultimo costruttore. L'oggetto istanza della classe base viene creato prima di quello della sottoclasse.

Anche se non si crea un costruttore per la classe *Cartoon*, il compilatore ne prevedrà uno di default (a zero argomenti) che invocherà il costruttore della classe base. Questa aggiunta non si ha se l'utente specifica un qualche costruttore.

Inizializzazione della classe di base: costruttori con argomenti

Se il costruttore della classe base ha degli argomenti in input allora bisogna esplicitare nella sua invocazione la lista di tali argomenti attraverso l'uso della parola chiave *super*.

Esempio

```
// Ereditarietà, costruttori, argomenti.
class Game {
Game(int i) {
System.out.println("Game constructor");
}
}

class BoardGame extends Game {
BoardGame(int i) {
super(i);
System.out.println("BoardGame constructor");
}
}

public class Chess extends BoardGame {
Chess() {
super(11);
System.out.println("Chess constructor");
}
public static void main(String[] args) {
Chess x = new Chess();
}
}
```

Inizializzazione della classe di base: costruttori con argomenti

Se non ci fosse la chiamata al costruttore della classe base in *BoardGame()*, il compilatore non troverebbe il costruttore *Game()*, segnalando un errore.

La chiamata al costruttore della classe base, deve essere la prima cosa da fare nel costruttore della classe derivata (altrimenti si verifica un errore di compilazione).

Combinazione di ereditarietà e composizione.

Nella maggior parte dei casi l'approccio per ereditarietà e per composizione verranno usati insieme.

Esempio

Ereditarietà e composizione

Esempio

```
// Combina composizione ed ereditarietà
class Plate {
Plate(int i) {
System.out.println("Plate constructor");
}
}
```

```
class DinnerPlate extends Plate {
DinnerPlate(int i) {
super(i);
System.out.println(
"DinnerPlate constructor");
}
}
```

```
class Utensil {
Utensil(int i) {
System.out.println("Utensil constructor");
}
}
```

```
class Spoon extends Utensil {
Spoon(int i) {
super(i);
System.out.println("Spoon constructor");
}
}
```

```
class Fork extends Utensil {
Fork(int i) {
super(i);
System.out.println("Fork constructor");
}
}
```

Esempio

```
class Knife extends Utensil {
Knife(int i) {
super(i);
System.out.println("Knife constructor");
}
}
```

```
// A cultural way of doing something:
class Custom {
Custom(int i) {
System.out.println("Custom constructor");
}
}
```

```
public class PlaceSetting extends Custom {
Spoon sp;
Fork frk;
Knife kn;
DinnerPlate pl;
PlaceSetting(int i) {
super(i + 1);
sp = new Spoon(i + 2);
frk = new Fork(i + 3);
kn = new Knife(i + 4);
pl = new DinnerPlate(i + 5);
System.out.println(
"PlaceSetting constructor"); }
public static void main(String[] args) {
PlaceSetting x = new PlaceSetting(9);
}
} ///:~
```

Derivazione protetta…

Nelle applicazione del mondo reale, nasce spesso l'esigenza di rendere i membri di una classe inaccessibili a tutte le altre classi tranne quelle da essa derivate.

A tale scopo Java prevede la parola chiave *protected* che applicata ai membri di una classe, li rende accessibili solo nel package e alle sue sottoclassi (anche di altri package).

Derivazione protetta…

```
// La parola chiave protected.
import java.util.*;
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; }
}

public class B extends A {
private int j;
public B(int jj) { super(jj); j = jj; }
public void change(int x) { set(x); }
}
```

Il metodo *change()* ha accesso al metodo *set()* perché è stato dichiarato *protected*.

La parola chiave ***final***

La parola chiave *final* ha un significato leggermente differente a seconda del contesto in cui viene inserita.

Si applica a dati, ai metodi e alle classi.

final applicato ai dati…

Molti linguaggi di programmazione hanno un modo particolare per indicare al compilatore che un dato è "costante".

Una costante è utile per due ragioni:

- può essere una costante a tempo di compilazione che non cambierà mai;
- può essere una valore inizializzato a run-time che non si vuole più cambiare in seguito;

Nel primo caso il compilatore userà subito il valore della costante nei calcoli in cui è richiesta in modo da eliminare alcuni sovraccarichi di lavoro a run-time.

…***final*** applicato ai dati…

In Java le costanti di questo genere devono essere di tipo primitivo e sono espresse usando la parola chiave *final*. Inoltre deve essere assegnato loro un valore al momento della definizione.

Quando si usa *final* con i riferimenti agli oggetti cambiano leggermente le cose, infatti mentre con i tipi primitivi *final* rende il valore costante, con i riferimenti agli oggetti accade che rende i riferimenti costanti. Ciò significa che se si è inizializzato un riferimento ad un oggetto, esso non può più essere modificato sebbene l'oggetto stesso possa esserlo.

…***final*** applicato ai dati…

Java infatti non fornisce un modo per rendere degli oggetti (array inclusi) costanti (anche se è possibile realizzare delle classi apposite).

Esempio

```
// L'effetto di final sugli attributi.
class Value {
int i = 1;}
public class FinalData {
// Costanti al compile-time
final int i1 = 9;
static final int VAL_TWO = 99;
// Tipiche costanti pubbliche:
public static final int VAL_THREE = 39;
// Costanti al run-time:
final int i4 = (int)(Math.random()*20);
static final int i5 = (int)(Math.random()*20);
Value v1 = new Value();
final Value v2 = new Value();
static final Value v3 = new Value();
// Array:
final int[] a = { 1, 2, 3, 4, 5, 6 };
public void print(String id) {
System.out.println(
id + ": " + "i4 = " + i4 +", i5 = " + i5);
}
public static void main(String[] args) {
FinalData fd1 = new FinalData();
//! fd1.i1++; // Errore: valore non modificabile
fd1.v2.i++; // L'oggetto non e' costante!
fd1.v1 = new Value(); // OK -- non final
for(int i = 0; i < fd1.a.length; i++)
fd1.a[i]++; // Object isn't constant!
//! fd1.v2 = new Value(); // Errore: non si puo'
//! fd1.v3 = new Value(); // cambiare il riferim.
//! fd1.a = new int[3];
fd1.print("fd1");
System.out.println("Creating new FinalData");
FinalData fd2 = new FinalData();
fd1.print("fd1");
fd2.print("fd2");}}
```

…***final*** applicato ai dati…

Si fa notare che i tipi primitivi preceduti da *final* e *static* insieme, con valori costanti (cioè costanti a tempo di compilazione) hanno per convenzione nomi con lettere maiuscole e parole separate da underscore (analogamente alle costanti in C) .

…***final*** applicato ai dati…

Java permette la creazione di *blank final* che sono dati dichiarati come *final* senza un valore iniziale. Questi devono essere inizializzati esplicitamente prima di essere utilizzati nelle classi e assicurano una maggiore flessibilità nell'uso di *final* visto che, per esempio, ora è possibile avere una valore diverso per ogni oggetto pur mantenendo le caratteristiche fornite da *final*.

…*final* applicato ai dati…

```
// "Blank" final.
class Poppet { }

class BlankFinal {
final int i = 0; // Final inizializzato
final int j; // Final vuoto
final Poppet p; // Riferimento final vuoto
// Gli attributi final vuoti devono essere
// inizializzati nel costruttore
BlankFinal() {
j = 1; // Inizializza final vuoto
p = new Poppet();}
BlankFinal(int x) {
j = x; // Inizializza final vuoto
p = new Poppet();}
public static void main(String[] args) {
BlankFinal bf = new BlankFinal();}}
```

…***final*** applicato ai dati…

Java permette di rendere *final* gli argomenti di un metodo. Ciò significa che all'interno del metodo non si possono cambiare i riferimenti a cui gli argomenti puntano.

…*final* applicato ai dati.

```
// Uso di “final” per gli argomenti.
class Gizmo {
public void spin() {}
}
public class FinalArguments {
void with(final Gizmo g) {
//! g = new Gizmo(); // Illegale -- g e’ final
}
void without(Gizmo g) {
g = new Gizmo(); // OK -- g non e’ final
g.spin();
}
// void f(final int i) { i++; } //Non modificabile
// Si puo’ solo leggere un final di tipo primitivo:
int g(final int i) { return i + 1; }
public static void main(String[] args) {
FinalArguments bf = new FinalArguments();
bf.without(null);
bf.with(null);
}
```

final applicato ai metodi.

Ci sono due ragioni per dichiarare *final* dei metodi:

- impedire alle classi derivate di cambiare il significato del metodo ereditato;
- i metodi *final* sono più efficienti perché si permette al compilatore di effettuare ogni chiamata al metodo come una chiamata *inline*.

I metodi *private* sono implicitamente *final* perché non è possibile accedere ad un metodo *private* ne "sovrascriverlo"(*overriding*).

Pertanto se si aggiunge *final* ad un metodo *private* non si associa alcun significato extra.

final applicato alle classi….

Quando si dichiara una intera classe *final*, si specifica che non si desidera derivare delle classi da quella corrente.

In altre parole si vuole, per questioni di sicurezza o altro, che non sia possibile specializzare tale classe.

Alternativamente si potrebbe desiderare che le attività in cui sono coinvolti gli oggetti di tale classe siano più efficienti possibili.

…*final* applicato alle classi.

```
// Making an entire class final.
class SmallBrain {}
final class Dinosaur {
int i = 7;
int j = 1;
SmallBrain x = new SmallBrain();
void f() {}
}
//! Una classe Further estende Dinosaur {}
// errore: Non si puo' estendere una classe final
public class Jurassic {
public static void main(String[] args) {
Dinosaur n = new Dinosaur();
n.f();
n.i = 40;
n.j++;
}
}
```