

Appunti di Progettazione del Software

Canale 2 · Prof. Massimo Mecella

Capitolo 7: Ereditarietà

Autore: Tommaso C.

Appunti tratti dalle lezioni in aula e dal materiale del corso

Argomenti: Derivazione fra classi · Casting · Il livello **protected**
Costruttori delle classi derivate · Overriding · Late binding · Campi nascosti

Indice

1	Derivazione fra classi	1
2	Casting	3
3	Il livello di accesso protected	4
4	Costruttori delle classi derivate	5
5	Overriding	8
6	Late binding	10
7	Campi nascosti	11
8	Esercizi	13
9	Soluzioni	14

1 Derivazione fra classi

Definizione – Classe derivata

Una classe **derivata** si costruisce a partire da una classe esistente, la **classe base**, di cui riusa tutto, aggiungendo ciò che le serve:

```
public class D extends B { ...}
```

La classe base si chiama anche **superclasse** o classe padre. Il meccanismo si chiama **derivazione** fra classi, o **ereditarietà**.

È l'ereditarietà del Capitolo 2, Sezione 6, scritta in Java.

```
public class B {    // classe base
    int x;
    void G() { x = x * 20; }
}

public class D extends B {    // classe derivata
    void H() { x = x * 10; }
}
```

Proposizione – Principi della derivazione

1. **Eredita:** le proprietà (campi dati e metodi) della classe base sono definite implicitamente anche nella derivata.
2. **Aggiunge:** la derivata può avere proprietà ulteriori.
3. **È compatibile:** ogni oggetto della derivata è anche un oggetto della base, quindi si può usare dove serve un oggetto della base.
4. **Non il viceversa:** un oggetto della base non si può usare dove serve un oggetto della derivata.

Nell'esempio **D** ha il campo **x** e il metodo **G()**, anche se nel suo codice non compaiono, e in più il metodo **H()**. Anche i campi **private** della base fanno parte dell'oggetto derivato, ma il codice della derivata non li può nominare (Sezione 4).

Il terzo principio si riassume nella relazione «è un»: uno studente è una persona, quindi dove il programma si aspetta una persona gli si può dare uno studente. Si dice che **D** è **compatibile** con **B**.

```
public static void stampa(B bb) {
    System.out.println(bb.x);
}
...
D d = new D();
d.G();      // corretto: G() è ereditato da B
stampa(d);  // corretto: D è compatibile con B
```

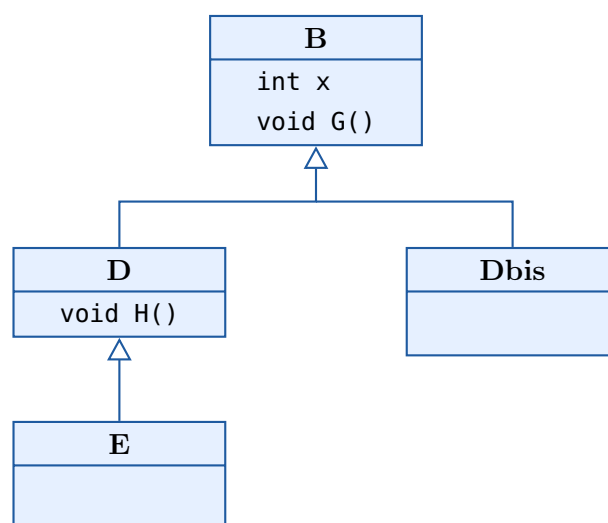
Il quarto principio vale perché una persona non è necessariamente uno studente.

```
B b = new B();
b.H();      // errore: B non ha il metodo H()
D d = b;    // errore: B non è compatibile con D
b = new D(); // corretto: D è compatibile con B
```

Il compilatore segnala *cannot find symbol, symbol: method H()* e *incompatible types: B cannot be converted to D*.

Gerarchie di classi. Le derivazioni si combinano in due modi:

- **in profondità:** una derivata fa da base per un'altra. Se **E extends D**, **E** eredita tutto da **D**, compreso ciò che **D** ha ereditato da **B**, ed è compatibile con **D** e con **B**.
- **in ampiezza:** una classe ha più derivate. **D** e **Dbis**, entrambe **extends B**, sono sorelle, cioè hanno tutto ciò che c'è in **B**, ma nessuna è compatibile con l'altra.



Una gerarchia di classi: *E* deriva da *D* (profondità), *D* e *Dbis* derivano da *B* (ampiezza). Le frecce vanno dalla derivata alla base.

Proposizione – Ereditarietà singola

Ogni classe ha una sola classe base, cioè dopo **extends** si scrive un solo nome. In Java non c'è ereditarietà multipla, e la gerarchia è un albero.

Così una proprietà ereditata viene da una sola classe, e non nasce ambiguità quando due classi definiscono proprietà con lo stesso nome.

2 Casting

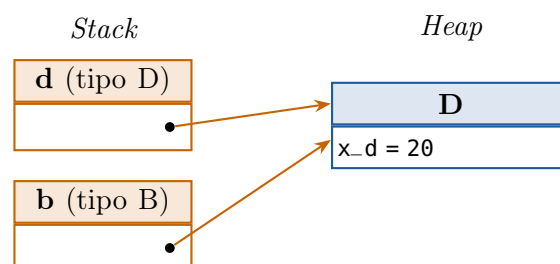
Definizione – Tipo statico e tipo dinamico

- Il **tipo statico** di un riferimento è la classe con cui è dichiarato, e il compilatore guarda solo questo.
- Il **tipo dinamico** è la classe dell'oggetto che il riferimento denota durante l'esecuzione.

Riferimenti di classi diverse possono denotare lo stesso oggetto, e il compilatore permette solo ciò che prevede il tipo statico.

```
public class D extends B { int x_d; }
...
D d = new D();
d.x_d = 10;
B b = d;      // corretto: b e d denotano lo stesso oggetto
b.x_d = 20;   // errore: x_d non è un campo di B
```

L'oggetto ha il campo `x_d`, ma `b` ha tipo statico `B`, e il compilatore segnala *cannot find symbol, symbol: variable x_d, location: variable b of type B*. È il **type checking statico**.



Due riferimenti allo stesso oggetto: il tipo dinamico è `D` per entrambi, il tipo statico di `b` è `B`.

Definizione – Cast fra classi

Il **cast (D) b** è un riferimento di tipo statico `D` allo stesso oggetto denotato da `b`. L'oggetto non cambia, perché cambia solo il modo in cui il compilatore vede il riferimento.

```
((D) b).x_d = 20;    // corretto: (D) b è un riferimento di tipo D
```

Le parentesi esterne servono perché il punto ha precedenza sul cast, quindi `(D) b.x_d` significa `(D) (b.x_d)` e non compila.

Proposizione – Correttezza del cast

- Un cast fra classi sullo **stesso cammino** della gerarchia (una classe e un suo antenato o discendente) compila sempre. Durante l'esecuzione la JVM controlla il tipo dinamico, e se l'oggetto non è della classe indicata solleva una `ClassCastException`, che interrompe il programma.
- Un cast fra classi che **non** stanno sullo stesso cammino, come le sorelle `D` e `Dbis`, è un errore di compilazione.

Il compilatore si fida del programmatore: che l'oggetto sia davvero della classe indicata è responsabilità di chi scrive il cast.

```
B b = new B();
D d = (D) b;    // compila, ma durante l'esecuzione: ClassCastException
d.x_d = 10;    // l'oggetto denotato da d non ha il campo x_d
```

L'esecuzione si ferma con `ClassCastException: class B cannot be cast to class D`.

3 Il livello di accesso **protected**

Definizione – Accesso **protected**

Un campo **protected** è accessibile alle classi dello stesso pacchetto e alle classi derivate, anche se stanno in un altro pacchetto. Ai clienti generici di un altro pacchetto è invece bloccato.

È il livello intermedio fra non qualificato e **public** nella tabella dei livelli di accesso del Capitolo 6, Sezione 6: una derivata in un altro pacchetto vede i campi **public** e **protected** della base.

Il motivo sta nei principi della derivazione:

- una derivata `D` non è un cliente qualsiasi di `B`, perché i suoi oggetti si usano al posto di quelli di `B`, e deve conoscere `B` più a fondo.
- `D` non coincide con `B`, e non è giusto che veda tutto come `B` stessa.

```
package base;
public class B {
    protected int p;
}

package derivata;
import base.B;
public class D extends B {
    void prova() { p = 1; }    // corretto: D deriva da B
}

package cliente;
import base.B;
public class Cl {
    void prova(B b) { b.p = 1; }    // errore: p has protected access in B
}
```

4 Costruttori delle classi derivate

Il costruttore della derivata deve inizializzare anche i campi ereditati, che fanno parte dell'oggetto, ma quelli **private** nella base non li può nemmeno nominare.

Definizione – **super(...)**

Nel costruttore della derivata, **super(argomenti)** ; invoca un costruttore della classe base, che inizializza la parte dell'oggetto definita nella base. Va scritto come **prima istruzione** del costruttore: prima si costruisce la parte della base, poi quella della derivata.

```
public class B {
    private int x_b;
    public B(int a) {    // costruttore della classe base
        x_b = a;
    }
    public int getXb() { return x_b; }
}

public class D extends B {
    private int x_d;
    public D(int b, int c) {    // costruttore della classe derivata
        super(b);              // invoca il costruttore di B
        x_d = c;
    }
    public int getXd() { return x_d; }
}
```

Con `D d = new D(3, 4);` la stampa di `d.getXb() + ", " + d.getXd()` dà `3, 4`. Al posto di `super(b)`, l'assegnamento `x_b = b` non compila, perché il compilatore segnala *x_b has private access in B*.

Proposizione – I costruttori non si ereditano

Un costruttore di **B** non è un costruttore di **D**. Anche se **B** ha `B(int a)`, la derivata non ha `D(int a)` se non lo scrive.

Proposizione – Invocazione implicita di `super()`

- Se un costruttore della derivata non invoca esplicitamente un costruttore della base, Java inserisce come prima istruzione `super()`, cioè il costruttore della base senza parametri.
- Se la derivata non ha costruttori, Java genera il **costruttore standard**, senza parametri, che invoca `super()` e lascia i campi della derivata ai valori predefiniti.

In entrambi i casi, se la base non ha un costruttore senza parametri, è un errore di compilazione.

L'errore è frequente, perché una base che definisce un costruttore con parametri non ha più il costruttore standard (Capitolo 6, Sezione 2). Con la prima **B** il compilatore segnala *constructor B in class B cannot be applied to given types; required: int, found: no arguments*.

Se **B** ha anche il costruttore senza parametri, entrambi i casi compilano:

```
public class B {
    private int x_b;
    public B(int a) { x_b = a; }
    public B() { x_b = 10; } // costruttore senza parametri
    public int getXb() { return x_b; }
}
```

Costruttori di D	Che cosa succede	getXb(), getXd()
D(int b, int c) con super(b), e new D(3, 4)	super(b) invoca B(int a)	3, 4
D(int c) senza super, e new D(4)	è invocato B()	10, 4
nessuno, e new D()	il costruttore standard invoca B(), x_d resta 0	10, 0

Nota – Istruzioni prima di **super(...)**

Da Java 25 prima di **super(...)** possono stare istruzioni che non usano l'oggetto in costruzione, per esempio un controllo sui parametri o l'assegnamento di un campo della derivata. Non si possono invocare metodi dell'oggetto, e il compilatore segnala *reference to getXb() may only appear after an explicit constructor invocation*.

Definizione – **this(...)**

In un costruttore, **this(argomenti);** invoca un altro costruttore della **stessa classe**. Come **super(...)**, si scrive come prima istruzione, quindi un costruttore contiene **super(...)** oppure **this(...)**, non entrambi.

Le inizializzazioni si scrivono così una volta sola, e gli altri costruttori vi rimandano. La **Persona** del Capitolo 6, Sezione 2, diventa:

```
public Persona() {
    this("Mario Rossi");
}
public Persona(String n) {
    this(n, null);
}
public Persona(String n, String r) { // l'unico che assegna i campi
    nome = n;
    residenza = r;
}
```

`new Persona()` crea la persona Mario Rossi con residenza `null`. Se si aggiunge un campo, si modifica un solo costruttore.

5 Overriding

Un metodo ereditato si può sovraccaricare come gli altri (Capitolo 5, Sezione 3).

```
public class B {  
    public void f(int i) { ... }  
}  
public class D extends B {  
    public void f(String s) { ... }    // overloading  
}  
...  
D d = new D();  
d.f(1);           // invoca B.f(int)  
d.f("prova");     // invoca D.f(String)
```

Definizione – Overriding

Si ha **overriding** (ridefinizione) quando la derivata definisce un metodo con esattamente la stessa segnatura di un metodo ereditato, cioè lo stesso nome e gli stessi tipi dei parametri, e lo stesso tipo restituito. Il nuovo metodo non si aggiunge a quello ereditato ma lo sostituisce: su un oggetto della derivata si esegue sempre il corpo ridefinito, qualunque sia il tipo del riferimento usato.

Nota – L'annotazione `@Override`

Scritta sopra il metodo, `@Override` chiede al compilatore di controllare che il metodo ridefinisca davvero un metodo ereditato. Senza, un errore nel nome o nel tipo di un parametro produce in silenzio un overloading. Con l'annotazione il compilatore segnala *method does not override or implement a method from a supertype*.

```

public class B {
    public void f(int i) {
        System.out.println(i * i);
    }
}

public class D extends B {
    public void f(String s) {    // overloading
        System.out.println(s);
    }
    @Override
    public void f(int n) {      // overriding
        System.out.println(n * n * n);
    }
}

public class Esempio {
    public static void main(String[] args) {
        B b = new B();
        b.f(5);                // stampa 25
        D d = new D();
        d.f("ciao");           // stampa ciao
        d.f(10);               // stampa 1000
        b = d;
        b.f(10);               // stampa 1000, non 100
    }
}

```

L'ultima chiamata passa per un riferimento di tipo **B**, ma l'oggetto è un **D**, quindi si esegue la versione ridefinita.

	Overloading	Overriding
Metodi	diversi, con lo stesso nome e parametri diversi	uno solo, con corpi diversi nella base e nella derivata
Chi sceglie	il compilatore, dagli argomenti	la JVM, dalla classe dell'oggetto

Osservazione – La versione della base

I clienti di **D** non possono più invocare **B.f(int)** su un oggetto **D**, ma solo **D.f(int)**. Dentro **D**, invece, **super.f(n)** invoca la versione della base, e così il metodo ridefinito può estendere il comportamento invece di rimpiazzarlo.

Proposizione – Regole dell'overriding

- Il livello di accesso del metodo ridefinito si può solo **allargare**: un metodo **protected** può diventare **public**, uno **public** non può diventare **protected** o **private**.
- Un metodo **final** non si può ridefinire, e una classe **final** non si può estendere. Per esempio **String** è **final**.

Il motivo della prima regola è il terzo principio: un oggetto **D** si deve poter usare dove serve un **B**. Se **B** promette un metodo **public** e **D** lo rendesse **private**, un cliente con un riferimento di tipo **B** a un oggetto **D** non potrebbe più invocarlo.

6 Late binding

Definizione – Late binding

Il **late binding** (collegamento ritardato) sceglie il corpo del metodo in due tempi:

1. durante la compilazione si controlla solo che esista un metodo con la segnatura richiesta nel tipo statico del riferimento, senza collegare la chiamata a un corpo.
2. durante l'esecuzione la chiamata si collega al corpo in base al tipo dinamico, cioè alla classe dell'oggetto di invocazione.

Proposizione – Tipo statico e tipo dinamico nelle chiamate

tipo statico → quali metodi si possono invocare tipo dinamico → quale corpo si esegue

Per questo **b.H()**, con **b** di tipo **B**, non compila neanche quando **b** denota un **D** (Sezione 1). Il late binding realizza il polimorfismo del Capitolo 2, Sezione 6: la stessa chiamata esegue il metodo della classe dell'oggetto.

```
public class B {
    protected int id;
    public B(int i) { id = i; }
    public boolean get() { return id < 0; }
}

public class D extends B {
    protected char ch;
    public D(int i, char c) {
        super(i);
        ch = c;
    }
    @Override
    public boolean get() { return ch != 'a'; }
}

public class LateBinding {
    public static void main(String[] args) {
        D d = new D(1, 'b');
        B b = d;
        System.out.println(b.get());    // true
        System.out.println(d.get());    // true
    }
}
```

Se si guardasse il riferimento, `b.get()` eseguirebbe la versione di `B`, e poiché 1 non è negativo stamperebbe `false`. L'oggetto invece è un `D`, quindi si esegue `D.get()`, e `'b'` è diverso da `'a'`: il programma stampa `true` due volte. Con un oggetto `B` la chiamata esegue `B.get()` (Esercizio 1).

7 Campi nascosti

Proposizione – L'overriding dei campi non esiste

Se la derivata dichiara un campo con lo stesso nome di un campo della base, il campo della derivata **nasconde** quello della base ma non lo sostituisce: l'oggetto li ha entrambi.

- Quale campo si usa dipende dal tipo statico del riferimento.
- Il campo della base resta accessibile con un riferimento di tipo base, o con `super.campo` dentro la derivata.

```
public class B { int i; }

public class D extends B {
    char i;
    void stampa() {
        System.out.println(i);           // il campo char di D
        System.out.println(super.i);     // il campo int di B
    }
}

public class SovrascritturaCampiDati {
    public static void main(String[] args) {
        D d = new D();
        d.i = 'f';                       // campo di D
        ((B) d).i = 9;                   // campo di B, attraverso un riferimento di tipo B
        d.stampa();                      // stampa f e poi 9
    }
}
```

D
int i = 9 (parte B)
char i = 'f' (parte D)

L'oggetto D ha due campi i: d.i denota quello di D, ((B) d).i e super.i quello di B.

Dare a un campo il nome di un campo ereditato rende il codice difficile da leggere, quindi conviene evitarlo.

Riepilogo

- **Derivazione:** `class D extends B`. `D` eredita tutto da `B`, può aggiungere proprietà e ha una sola classe base.
- **Compatibilità:** un oggetto `D` si usa dove serve un `B`, non il viceversa. Le classi sorelle non sono compatibili.
- **Cast:** `(D) b` cambia il tipo statico, non l'oggetto. Se l'oggetto non è un `D`, `ClassCastException`.
- **protected:** accesso alle classi del pacchetto e alle derivate, anche di altri pacchetti.
- **Costruttori:** non si eritano. `super(...)` invoca la base, se manca Java inserisce `super()`. `this(...)` invoca un costruttore della stessa classe.
- **Overriding:** stessa segnatura e tipo restituito, con `@Override`. L'accesso si può solo allargare, e i metodi `final` non si ridefiniscono.
- **Late binding:** il tipo statico decide quali metodi si invocano, il tipo dinamico quale corpo si esegue.
- **Campi nascosti:** un campo con il nome di uno ereditato non lo sostituisce, e conta il tipo statico.

8 Esercizi

Esercizio 1 – Late binding con un oggetto della base

Con le classi `B` e `D` del programma `LateBinding` (Sezione 6), che cosa stampa ciascuna istruzione, e perché?

```
B b1 = new B(1);
System.out.println(b1.get());
B b2 = new D(1, 'b');
System.out.println(b2.get());
```

Soluzione a pagina 14 →

9 Soluzioni

Soluzione dell'Esercizio 1 – Late binding con un oggetto della base

La prima stampa **false**, la seconda **true**. Le due variabili hanno lo stesso tipo statico **B**, quindi entrambe le chiamate compilano, perché **B** ha il metodo **get()**. Il corpo eseguito dipende dal tipo dinamico:

1. **b1** denota un oggetto **B**, quindi si esegue **B.get()**, e **1 < 0** è falso.
2. **b2** denota un oggetto **D**, quindi si esegue **D.get()**, e **'b' != 'a'** è vero.

← Torna all'Esercizio 1, pagina 13