

Java: Risoluzione dell'overloading e dell'overriding

Marco Faella

Dip. Ing. Elettrica e Tecnologie dell'Informazione
Università di Napoli “Federico II”

Corso di Linguaggi di Programmazione I

- Per “binding dinamico” (letteralmente, “bind” significa “legare”) si intende il meccanismo per cui non è il compilatore, ma la JVM ad avere l'ultima parola su **quale metodo invocare** in corrispondenza di ciascuna chiamata a metodo
- In altri termini, si tratta di stabilire il *legame di locazione* di un nome di metodo nel contesto di una invocazione

- Esempio:

$x.f(\text{exp})$

- Il binding di x è statico
 - L'identità di x viene stabilita a tempo di compilazione
- Il binding di $x.f$ è dinamico
 - L'identità di $x.f$ viene stabilita a tempo di esecuzione
- Il fatto che il binding di $x.f$ sia dinamico è una diretta conseguenza del polimorfismo e dell'overriding
- Ovvero, ciascun riferimento può puntare ad oggetti di tipo effettivo diverso (polimorfismo) e ciascuno di questi tipi effettivi può prevedere una versione diversa dello stesso metodo (overriding)
- Inoltre, il compilatore non può prevedere di che tipo effettivo sarà una variabile nel corso dell'esecuzione del programma (tecnicamente, questo problema è *indecidibile*)

- In Java, il binding dei metodi (collegare ciascuna chiamata ad un metodo vero e proprio) avviene in due fasi:
 - **Early binding**, in cui il compilatore risolve l'*overloading* (scegliendo la firma più appropriata alla chiamata)
 - **Late binding**, in cui la JVM risolve l'*overriding* (scegliendo il metodo vero e proprio)
- Il late binding non è necessario per quei metodi che non ammettono overriding: i metodi **privati**, **statici** o **final**
- Per questi metodi si parla di "binding statico", perché la scelta del metodo da eseguire viene fatta già dal compilatore

Come ***non*** viene scelta la firma più specifica

- Molti manuali di Java semplificano le regole dell'overloading, suggerendo che venga scelto il metodo che richiede *il minor numero di conversioni*
- Ad esempio, valutiamo l'invocazione `x.f(1, 2)`, supponendo che la classe di `x` offra i seguenti metodi:

```
public void f(double x, long y)
public void f(int x, double y)
```

Come ***non*** viene scelta la firma più specifica

- Molti manuali di Java semplificano le regole dell'overloading, suggerendo che venga scelto il metodo che richiede *il minor numero di conversioni*
- Ad esempio, valutiamo l'invocazione `x.f(1, 2)`, supponendo che la classe di `x` offra i seguenti metodi:

```
public void f(double x, long y)
public void f(int x, double y)
```

- Contiamo il numero di conversioni richieste, in due modi diversi:
 - Primo modo: contiamo quanti parametri richiedono una conversione
 - Secondo modo: contiamo il numero totale di “passi di conversione” richiesti (cioè, quanti archi dobbiamo percorrere nel grafo che rappresenta le conversioni implicite)

- Otteniamo i seguenti conteggi:

	# parametri da convertire	# passi di conversione
<code>public void f(double d, long l)</code>	2	3
<code>public void f(int d, double l)</code>	1	2

- In entrambi i modi, sembra prevalere la seconda firma
- Invece, questo è un caso di **ambiguità** (errore di compilazione)
- Le prossime slide spiegano il perché

- L'early binding si divide a sua volta in due fasi:
 - 1) Individuazione delle firme candidate
 - 2) Scelta della firma più specifica tra quelle candidate
- Le prossime slide approfondiscono ciascuna di queste fasi

- Consideriamo una generica invocazione

$$x.f(a_1, \dots, a_n)$$

- Si ricorda che per “firma” di un metodo si intende il suo nome e l'elenco dei tipi dei suoi parametri formali
- Una generica firma

$$f(T_1, \dots, T_n)$$

è *candidata* per la chiamata in questione se:

- Si trova nella classe dichiarata di x o in una sua superclasse
 - E' *visibile* dal punto della chiamata, rispetto alle regole di visibilità Java
 - E' *compatibile* con la chiamata; ovvero, per ogni indice i compreso tra 1 ed n , il tipo (dichiarato) del parametro attuale a_i è **assegnabile** al tipo T_i
- Se nessuna firma risulta candidata per una data chiamata, il compilatore segnala un errore (accompagnato dal messaggio: “cannot find symbol”)

- Date due firme con lo stesso nome e numero di argomenti
 $f(T_1, \dots, T_n)$ e $f(U_1, \dots, U_n)$
- Si dice che la prima firma è **più specifica** della seconda se, per ogni indice i compreso tra 1 ed n , il tipo T_i è assegnabile al tipo U_i
- ATTENZIONE: notate che questo confronto tra firme non dipende dal tipo dei parametri attuali passati alla chiamata
- E' facile verificare che essere "più specifico" è una relazione riflessiva, antisimmetrica e transitiva, proprio come la relazione di assegnabilità
- Quindi, essa è una **relazione d'ordine** sull'insieme delle firme
- Tale ordine è **parziale**, in quanto alcune firme non sono confrontabili tra loro
- Ad esempio, le firme $f(\text{int}, \text{double})$ e $f(\text{double}, \text{int})$ non sono confrontabili quanto a specificità

- L'early binding si conclude individuando, tra le firme candidate, una che sia **più specifica di tutte le altre**
- Per simulare "a mano" questo meccanismo, nei casi complessi può essere conveniente realizzare un diagramma a **grafo**, in cui ci sia un nodo per ciascuna firma candidata ed un arco orientato da un nodo "a" ad un nodo "b" quando la firma "a" è più specifica della firma "b"
- Se nel grafo c'è un nodo che ha **archi uscenti diretti verso tutte le altre firme**, quella sarà la firma scelta dal compilatore
- Se nessuna firma è più specifica di tutte le altre, il compilatore segnala un errore e termina (parleremo di *chiamata ambigua*)
- Domanda: E' possibile che si trovi più di una firma più specifica di tutte le altre? Perché?
- Attenzione: in questa discussione sull'overloading sono state tralasciate la programmazione generica e l'autoboxing

Rivalutiamo l'invocazione `x.f(1, 2)`, supponendo che la classe di `x` offra i seguenti metodi:

```
public void f(double x, long y)
public void f(int x, double y)
```

- Il late binding è la fase di risoluzione dell'**overriding**, a carico della **JVM**
- Questa fase **riceve in input la firma** scelta dal compilatore durante l'early binding
- Consideriamo nuovamente la generica invocazione

$$x.f(a_1, \dots, a_n)$$

La JVM cerca un metodo da eseguire, con il seguente algoritmo:

- si parte dalla classe effettiva di x
 - si cerca un metodo che abbia la firma **identica** a quella scelta dall'early binding
 - se non lo si trova, si passa alla superclasse
 - così via, fino ad arrivare ad Object
-
- Questo procedimento può fallire, cioè non trovare alcun metodo, solo in casi molto particolari
 - Ad esempio, se una classe A dipendeva da una classe B e la classe B è cambiata da quando è stata compilata A

- Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```
class A {  
    public String f(double n, A x) { return "A1"; }  
    public String f(double n, B x) { return "A2"; }  
    public String f(int n, Object x) { return "A3"; }  
}  
class B extends A {  
    public String f(double n, B x) { return "B1"; }  
    public String f(float n, Object y) { return "B2"; }  
}  
class C extends A {  
    public final String f(int n, Object x) { return "C1"; }  
}
```

```
public class Test {  
    public static void main(String[] args) {  
        C gamma = new C();  
        B beta = new B();  
        A alfa = beta;  
        System.out.println(alfa.f(3, beta));  
        System.out.println(alfa.f(3.0, beta));  
        System.out.println(beta.f(3.0, alfa));  
        System.out.println(gamma.f(3, gamma));  
        System.out.println(false ||  
            alfa.equals(beta));  
    }  
}
```

- Indicare l'output del programma
- Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione
- Per ogni chiamata ad un metodo (escluso System.out.println), indicare la lista delle firme candidate

- Esaminiamo le chiamate una per volta

1) `System.out.println(alfa.f(3, beta));`

- alfa è di tipo dichiarato A, quindi le firme candidate vanno cercate nella classe A (o tutt'al più in Object)
- i due parametri attuali della chiamata sono di tipo (dichiarato) int e B, rispettivamente
- la firma `f(double, A)` è candidata, in quanto visibile e compatibile
 - essa è compatibile perché int è assegnabile a double (conversione implicita) e B è assegnabile ad A (sottotipo)
- la firma `f(double, B)` è candidata, in quanto visibile e compatibile
- la firma `f(int, Object)` è candidata, in quanto visibile e compatibile
- non vi sono altre firme candidate
- Delle tre firme candidate, la seconda è più specifica della prima, ma non è confrontabile con la terza
- Quindi, nessuna firma è più specifica di tutte le altre
- Il risultato è un **errore di compilazione**
- ATTENZIONE: ricordate che la scelta della firma più specifica non dipende dal tipo dei parametri attuali della chiamata

- Esaminiamo la seconda chiamata:

2) `System.out.println(alfa.f(3.0, beta));`

- alfa è di tipo dichiarato A, quindi le firme candidate vanno cercate nella classe A (o tutt'al più in Object)
- i due parametri attuali della chiamata sono di tipo (dichiarato) double e B, rispettivamente
- la firma `f(double, A)` è candidata, in quanto visibile e compatibile
- la firma `f(double, B)` è candidata, in quanto visibile e compatibile
- la firma `f(int, Object)` *non* è candidata, in quanto non compatibile
- non vi sono altre firme candidate
- Delle due firme candidate, la seconda è più specifica della prima
- Quindi, l'early binding si conclude con la selezione della firma `f(double, B)`
- Per il late binding, cerchiamo il metodo da eseguire a partire dalla classe effettiva di alfa: B
- Nella classe B, troviamo un metodo visibile con quella firma
- Quindi, l'output di questa chiamata è

B1

- Esaminiamo la terza chiamata:

3) `System.out.println(beta.f(3.0, alfa));`

- beta è di tipo dichiarato B, quindi le firme candidate vanno cercate in B, in A e in Object
- i due parametri attuali della chiamata sono di tipo (dichiarato) double ed A, rispettivamente
- la firma `f(double, B)` non è candidata, in quanto non compatibile (secondo argomento)
- la firma `f(float, Object)` non è candidata, in quanto non compatibile (primo argomento)
- la firma `f(int, Object)` non è candidata, in quanto non compatibile (primo argomento)
- la firma `f(double, A)` è candidata, in quanto visibile e compatibile
- Essendoci una sola firma candidata, l'early binding si conclude con la selezione della firma `f(double, A)`
- Per il late binding, cerchiamo il metodo da eseguire a partire dalla classe effettiva di beta: B
- Nella classe B, non c'è alcun metodo con la firma scelta
- Passiamo alla classe A, in cui troviamo un metodo con la firma scelta
- Quindi, l'output di questa chiamata è

A1

- Esaminiamo l'ultima chiamata:

4) `System.out.println(gamma.f(3, gamma));`

- gamma è di tipo dichiarato C, quindi le firme candidate vanno cercate in C, in A e in Object
- i due parametri attuali della chiamata sono di tipo (dichiarato) int e C, rispettivamente
- la firma `f(int, Object)` è candidata, in quanto visibile e compatibile
- la firma `f(double, A)` è candidata, in quanto visibile e compatibile
- la firma `f(double, B)` non è candidata, in quanto non compatibile (secondo argomento)
- Le due firme candidate non sono confrontabili
- Quindi, l'early binding si conclude con un **errore di compilazione**

Esercizio 2 (esame 27/11/2009, #2)

- Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```
class A {  
    public String f(double n, Object x) { return "A1"; }  
    public String f(double n, A x)    { return "A2"; }  
    public String f(int n,   Object x) { return "A3"; }  
}  
class B extends A {  
    public String f(double n, Object x) { return "B1"; }  
    public String f(float n,  Object y) { return "B2"; }  
}  
class C extends B {  
    public final String f(double n, A x) { return "C1"; }  
}
```

```
public class Test {  
    public static void main(String[] args) {  
        C gamma = new C();  
        B beta  = new B();  
        A alfa  = gamma;  
        System.out.println(alfa.f(3.0, gamma));  
        System.out.println(beta.f(3, beta));  
        System.out.println(beta.f(3.0, null));  
        System.out.println(gamma.f(3.0, gamma));  
    }  
}
```

- Indicare l'output del programma
- Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione
- Per ogni chiamata ad un metodo (escluso `System.out.println`), indicare la lista delle firme candidate

Esercizio 3 (esame 25/1/2017, #1)

- Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```
class A {  
    public String f(A x, A[] y, B z) { return "A1"; }  
    public String f(A x, Object y, B z) { return "A2"; }  
}  
class B extends A {  
    public String f(B x, A[] y, B z) { return "B1:" + x.f((A)x, y, z); }  
    public String f(A x, B[] y, B z) { return "B2"; }  
}  
class C extends B {  
    public String f(A x, A[] y, C z) { return "C1:" + z.f(new C(), y, z); }  
}
```

```
public class Test {  
    public static void main(String[] args) {  
        C gamma = new C();  
        B beta = gamma;  
        A[] array = new A[10];  
        System.out.println(beta.f(gamma, array, gamma));  
        System.out.println(gamma.f(array[0], null, beta));  
        System.out.println(beta == gamma);  
    }  
}
```

- Indicare l'output del programma
- Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione
- Per ogni chiamata ad un metodo (escluso System.out.println), indicare la lista delle firme candidate

Binding dinamico e auto-(un)boxing

- Nella risoluzione dell'overloading, l'autoboxing e l'auto-unboxing entrano in gioco *soltanto se necessario*
 - Ovvero, soltanto se altrimenti non ci sarebbero firme candidate
- Quindi, come **primo tentativo**, il compilatore cerca le firme che sono candidate senza prendere in considerazione l'autoboxing e l'auto-unboxing
- **Solo se non ci sono firme candidate**, il compilatore abilita le conversioni da tipo primitivo a tipo wrapper, e viceversa, e *riesamina tutte le firme (secondo tentativo)*
- Questa scelta è stata fatta per mantenere la compatibilità con il codice scritto prima dell'introduzione dell'auto-(un)boxing
- Infatti, le invocazioni a metodo che funzionavano senza auto-(un)boxing continuano a funzionare con l'auto-(un)boxing, e *sono risolte nello stesso modo*
- Con l'auto-(un)boxing, alcune invocazioni che prima non erano consentite diventano lecite

- Una volta ottenuto un insieme non vuoto di firme candidate, il compilatore passa alla scelta della più specifica, con le regole descritte nelle slide precedenti
- Quindi, l'auto-(un)boxing **non influenza** in alcun modo **la scelta della firma** più specifica
- Analogamente, l'auto-(un)boxing **non influenza** in alcun modo **il late binding**

- Consideriamo i seguenti metodi:

```
public static int foo(int i, Object o) { return 1; }  
public static int foo(long i, String o) { return 2; }
```

- La chiamata

```
foo(new Integer(7), "ciao")
```

provoca un errore di **ambiguità**, perché il compilatore prima ottiene un insieme di firme candidate vuoto; poi, una volta attivato l'auto-(un)boxing, ottiene candidate **entrambe** le firme "foo", delle quali nessuna è più specifica dell'altra

- Consideriamo i seguenti metodi:

```
public static int foo(int i, Object o) { return 1; }  
public static int foo(long i, String o) { return 2; }
```

- La chiamata `foo(new Float(7), "ciao")` provoca un **errore** di compilazione, in quanto il compilatore non trova firme candidate neanche al secondo tentativo
- La chiamata `foo(new Long(7), "ciao")` ottiene output 2, in quanto quella è l'unica firma candidata, una volta attivato l'auto-(un)boxing

- Consideriamo i seguenti metodi:

```
public static int bar(double a, Integer b) { return 3; }  
public static int bar(Double a, Integer b) { return 4; }
```

- La chiamata `bar(1.0, 7)` provoca un errore di ambiguità, perché entrambe le firme saranno candidate (al secondo tentativo)
- La chiamata `bar(1, 7)` ottiene il risultato 3, perché avrà un'unica firma candidata (al secondo tentativo)
 - la seconda firma non è candidata perché un *int* non può trasformarsi in *Double* tramite autoboxing