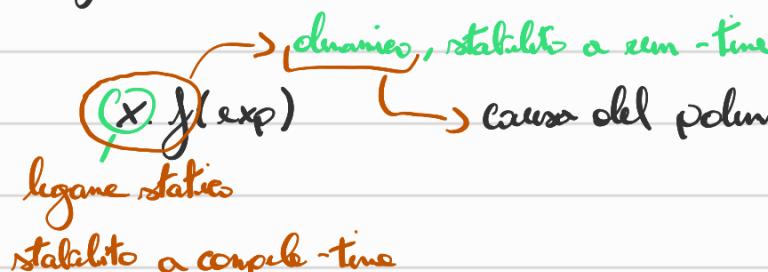


## Risoluzione dell'overloading e dell'overriding

Binding dinamico  $\rightarrow$  Reclamo per cui non è il compilatore, né lo JVR ad avere l'ultima parola per l'invocazione del metodo.

Si stabilisce il legame di locazione di un nome di metodo durante l'invocazione

Esempio:   
dinamico, stabilito a run-time  
causa del polimorfismo e dell'overriding  
legato statico  
stabilito a compile-time

Polimorfismo, ogni riferimento può puntare ad oggetti di tipo diverso.

Overriding, ognuno dei seguenti tipi può prendere una versione diversa dello stesso metodo

Fasi del Binding  $\rightarrow$  due fasi:

Early binding, il compilatore risolve l'overloading (seleziona la funzione)

Late binding, lo JVR risolve l'overriding (seleziona il metodo)  
non necessario per metodi privi di overriding

Come non viene scelta la funzione più specifica

I manuali di Java semplificano le regole dell'overloading, dichiarando che viene scelto il metodo con meno conversioni

Esempio  $\rightarrow$   $x.f(1, 2)$  e  $x$  ha i seguenti metodi

pubblici void  $f$  (double  $x$ , long  $y$ )

pubblici void  $f$  (int  $x$ , double  $y$ )

Abbiamo due modi diversi:

- ① Contano i parametri da convertire;
- ② Contano il totale di possibili conversioni richiesti (quanti archi vengono percorsi nel graph che rappresenta le conversioni implicite)

	# parametri da convertire	# possibili conver.
pubblici void $f$ (double $x$ , long $y$ )	2	3
pubblici void $f$ (int $x$ , double $y$ )	1	2

Sembra proprio che ricade fine, ma c'è ambiguità (errore di compilazione)

Vedremo presto il perché

Fasi dell'early binding  $\rightarrow$  Anche questa fase ha due fasi:

① Individuazione delle forme candidat

② Scelta della forma più specifica

## Approfondimento fase 1

Consideramus case generis Invocatioe  $X.f(a_1, \dots, a_n)$

Per forma si intende il nome e l'elenco dei tipi dei parametri formalii di un metodo.

Una generica funzione  $f(T_1, \dots, T_n)$  è candidata se:

- Si trova nelle classi dichiarata di  $x$  o in una superclasse;
  - È visibile nelle chiamate
  - È compatibile con le chiamate, cioè
 
$$x.f(a_1, \dots, a_n)$$

$$f(T_1, \dots, T_n)$$
C'è corrispondenza di tipo

Se nessuno firma i candidati  $\Rightarrow$  errore di compilazione

## Confronto tra le mie candidate

Se  $f(T_1, \dots, T_n)$  e  $f(U_1, \dots, U_n)$  avuti stesso n. e num. di argomenti

Si dice che la prima forma è più specifica della seconda se, vi compreso tra 1 ed  $N$ , il tipo  $T_i$  è assegnabile a  $V_i$ .

 relazione riflessiva, antimermetrica e transitiva  
come le relazioni di omogeneità

Relazione d'ordine **parziale** sull'insieme delle firme

## Approfondimento fase 2

Ricerca delle **PIÙ** specifiche. È possibile visualizzare il meccanismo con un **diagramma a grafo**.

5 nodi rappresentano le firme e l'arco orientato da **a** a **b** ci dice che **a** è più specifica di **b**

Se nel grafo c'è un nodo che ha archi **orientati verso tutte le altre firme**, era sarà la **Scelta del compilatore**

Se nessuna firma è specifica, **errore di compilazione** (ambiguità)

**Late Binding** → fase di risoluzione dell'overriding, a carico della JVR

Venne presa in input la firma scelta dal **compilatore**.

Si ricorda nuovamente  $x.f(a_1, \dots, a_n)$

La JVR cerca il metodo giusto con il seguente algoritmo:

- Cerca nella classe di  $x$ ;
- Cerca le forme **identiche** a quelle scelte nell'**early binding**
- Se è presente, punta alle **superclasse**, altrimenti si punta ad **object**

- 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

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

Prima chiamata

Visto che alfa è di tipo A, cerca le firme nella classe A.

```

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"; }
}

```

Sono tutte e 3 possibili  
conducibili

La seconda è più specifica della prima, ma la terza non è confrontabile.

Risultato = errore di compilazione

**Nota** → la scelta della firma più specifica non dipende dal tipo dei parametri attuali

✓  
nella memoria

*grazie m  
Dolce*

System.out.println(alpha.f(3.0,beta));

Seconda chiamata

```
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"; }  
}
```

- $f(\text{double}, A)$  candidata perché variabile e compatibile
- $f(\text{double}, B)$  candidata perché variabile e compatibile
- $f(\text{int}, \text{Object})$  non è candidata, non compatibile

la seconda è la più specifica e viene scelta

■ (operando max)

grazie all'assegnazione  $A \alpha = \beta$ ;  $\alpha$  può interagire con i metodi di  $B$  → ma solo con quelli presenti anche in  $A$  e per overrule, viene scelto un suo metodo

Risultato = B1

System.out.println(beta.f(3.0,alpha));

Terza chiamata

```
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"; }  
}
```

- $f(\text{double}, B)$  non è candidata, secondo argomento non compatibile
- $f(\text{float}, \text{Object})$  " " " primo " " "
- $f(\text{int}, \text{Object})$  " " " primo " " "
- $f(\text{double}, A)$  è candidata (sol esiste l'unica, viene scelta)

Per il late binding, cerchiamo il metodo in B  $\Rightarrow$  non è presente  $\Rightarrow$  Passo ad A

Tra le firme . Risultato = A1

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

Le firme vanno create in C, in A e in Object

```
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 C extends A {
    public final String f(int n, Object x) { return "C1"; }
}
```

- $f(\text{int}, \text{Object})$  è candidata
- $f(\text{double}, A)$  è candidata

non confrontabili, errore di compilazione

- $f(\text{double}, \beta)$  non è candidata, secondo argomento non compatibile

## Early binding e autoboxing

Nell'overloading, l'autoboxing e unboxing non sono sempre presi in considerazione.  
(di solito quando non ci sarebbero altre candidate)

Primo tentativo → Il compilatore cerca le forme senza **auto(un)boxing**. Se non vengono trovate, il compilatore abilita le conversioni da Wrapper a primitive e viceversa.

Secondo tentativo → Il compilatore **riesamina tutte le forme**

Questa scelta è stata presa per mantenere la compatibilità con il codice scritto prima dell'introduzione dell'**auto(un)boxing**.

## Late binding e autoboxing

Dopo aver trovato diverse forme, il compilatore cerca la più specifica. L'**auto(un)boxing** non influenza la scelta delle forme.

Di conseguenza, l'**autoboxing** non influenza il **late binding**.

