Estensione delle classi

Estensione di una classe

- È possibile estendere una classe senza conoscere nulla dell'implementazione della classe estesa, ma solo il contratto.
- ```
  class Quadrato extends Rettangolo {
      ...
  }
  ```
- `extends` (*estende*) indica che la classe `Quadrato` è ottenuta dalla classe `Rettangolo` estendendone stato e comportamento, cioè aggiungendo campi o metodi.
- `Quadrato` è una sottoclasse di `Rettangolo`.
- `Rettangolo` è una superclasse di `Quadrato`.
  ```

Estensione di una classe

- La classe `Quadrato`, per il solo fatto di essere definita come estensione della classe `Rettangolo`, dispone di tutti
  - i campi
  - i metodi

  definiti nella classe `Rettangolo`.
- Ciò non implica che questi membri siano accessibili nel codice di `Quadrato` (dipende dalla visibilità).
- I costruttori non vengono ereditati.

La classe `Quadrato`

```
public class Quadrato extends Rettangolo {

  public Quadrato(double x) {
    super(x, x);
  }

  public double getLato() {
    return getBase();
  }

  public String toString() {
    return "lato = " + getLato();
  }

}
```

Metodi di un oggetto

I metodi eseguibili dagli oggetti di una classe possono essere suddivisi in tre gruppi:

- definiti per la prima volta nella classe (`getLato`)
- ridefiniti (`toString`)

  prototipo uguale a quello di un metodo della superclasse, ne ridefiniscono il comportamento
- ereditati dalla superclasse (`getArea`, `getPerimetro`,...).

Overloading e overriding

Overloading

- Overloading
  La possibilità di avere metodi o costruttori con lo stesso nome ma segnatura diversa.
  - Segnatura = nome e tipi dei suoi argomenti
  - Non è possibile definire più metodi con la stessa segnatura e tipo restituito diverso.
- L'overloading viene risolto in fase di compilazione.

Esempio

```
public static double valoreAssoluto(double x) {
   if (x > 0)
      return x;
   else
      return -x;
}

public static int valoreAssoluto(int x) {
  return (int) valoreAssoluto((double) x);
}
```

Esempio

- In compilazione viene scelta la segnatura del metodo da eseguire sulla base:
  - del tipo del riferimento utilizzato per invocare il metodo
  - degli argomenti indicati nella chiamata

- `r.m(2)`

  il compilatore cerca fra tutte le segnature di metodi di nome `m` disponibili per il tipo del riferimento `r` quella "*più adatta*" per gli argomenti specificati.

  Se le segnature disponibili sono ad esempio:

  ```
  int m(byte b)
  int m(long l)
  int m(double d)
  ```

  il compilatore sceglie la seconda.

Overriding

- Overriding
  Quando si riscrive in una sottoclasse un metodo della superclasse con la stessa segnatura.

- L'overriding viene risolto in fase di esecuzione.
  - Compilazione: scelta della segnatura
    il compilatore può stabilire solo la segnatura del metodo da eseguire
    (early binding)
  - Esecuzione: scelta del metodo
    La decisione relativa al metodo effettivo, tra quelli con la segnatura selezionata, viene rimandata all'esecuzione
    (late binding)

Fase di compilazione

(1) Scelta delle segnature “candidate”

Il compilatore individua le segnature che possono soddisfare la chiamata.

Un metodo può soddisfare la chiamata se è

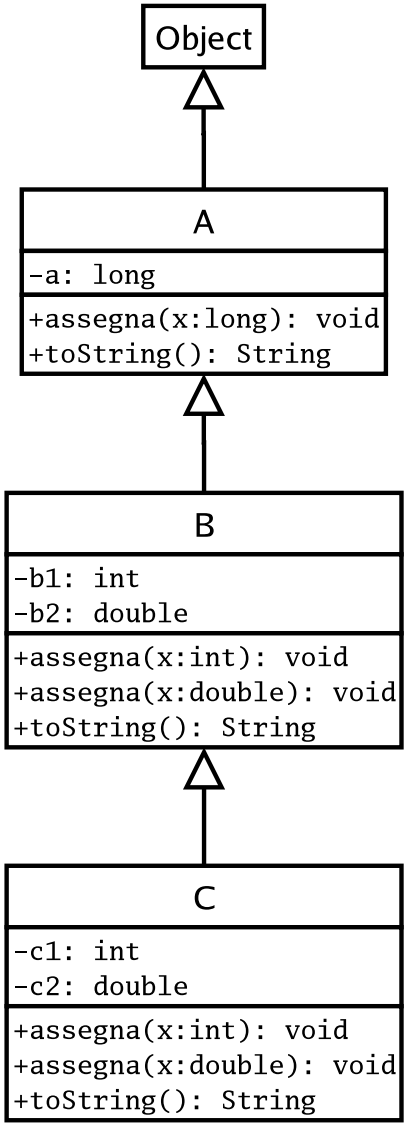
- compatibile con gli argomenti utilizzati nella chiamata
  - il numero dei parametri nella segnatura è uguale al numero degli argomenti utilizzati
  - ogni argomento è di un tipo assegnabile al corrispondente parametro
- accessibile al codice chiamante

Se non esistono segnature candidate, il compilatore segnala un errore.

(2) Scelta della segnatura “più specifica”

Tra le segnature candidate, il compilatore seleziona quella che richiede il minor numero di promozioni.

Fase di compilazione: scelta della segnatura

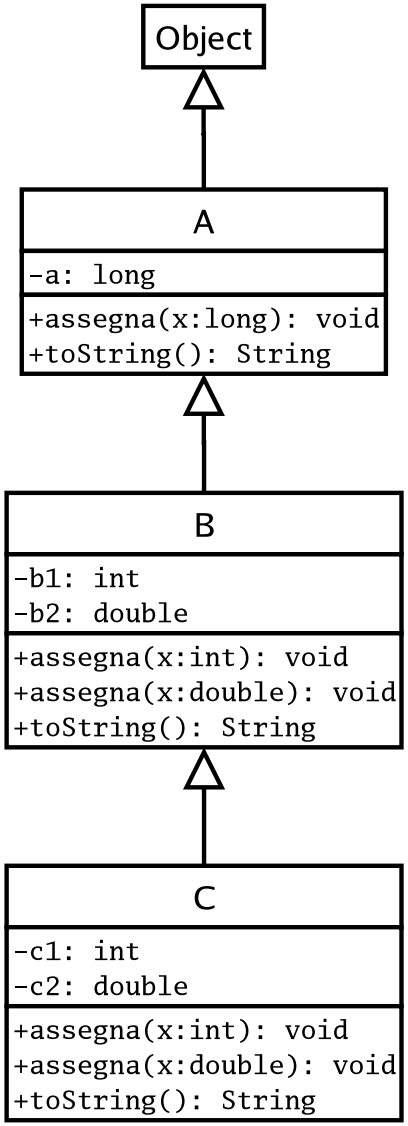



```
A alfa;
B beta;
C gamma;
```

- Il compilatore controlla che esista almeno un metodo invocabile per il tipo corrispondente al riferimento.
- `gamma.toString()`
  `gamma.equals(alfa)`

  sono lecite in quanto la classe `C` dispone di:
  - `toString` (definito localmente) senza parametri
  - `equals` (ereditato da `Object`) con parametro di tipo `Object` (cui verrà convertito l'argomento `alfa` di tipo `A`).

Fase di compilazione: scelta della segnatura




```
A alfa;
B beta;
C gamma;
```

- `alfa.assegna(2)`

  C'è una sola segnatura candidata.

  ```
  assegna(long x)
  ```

- `alfa.assegna(2.0)`

  Non viene accettata dal compilatore.

- `beta.assegna(2)`

  Ci sono tre segnature candidate:

  ```
  assegna(int x)
  assegna(double x)
  assegna(long x)
  ```

  La più specifica è `assegna(int x)`.

Ambiguità

- Se per la chiamata:

```
z(1, 2)
```

le segnature candidate sono:

```
z(double x, int y)
z(int x, double y)
```

il compilatore non riesce a individuare una segnatura più specifica.

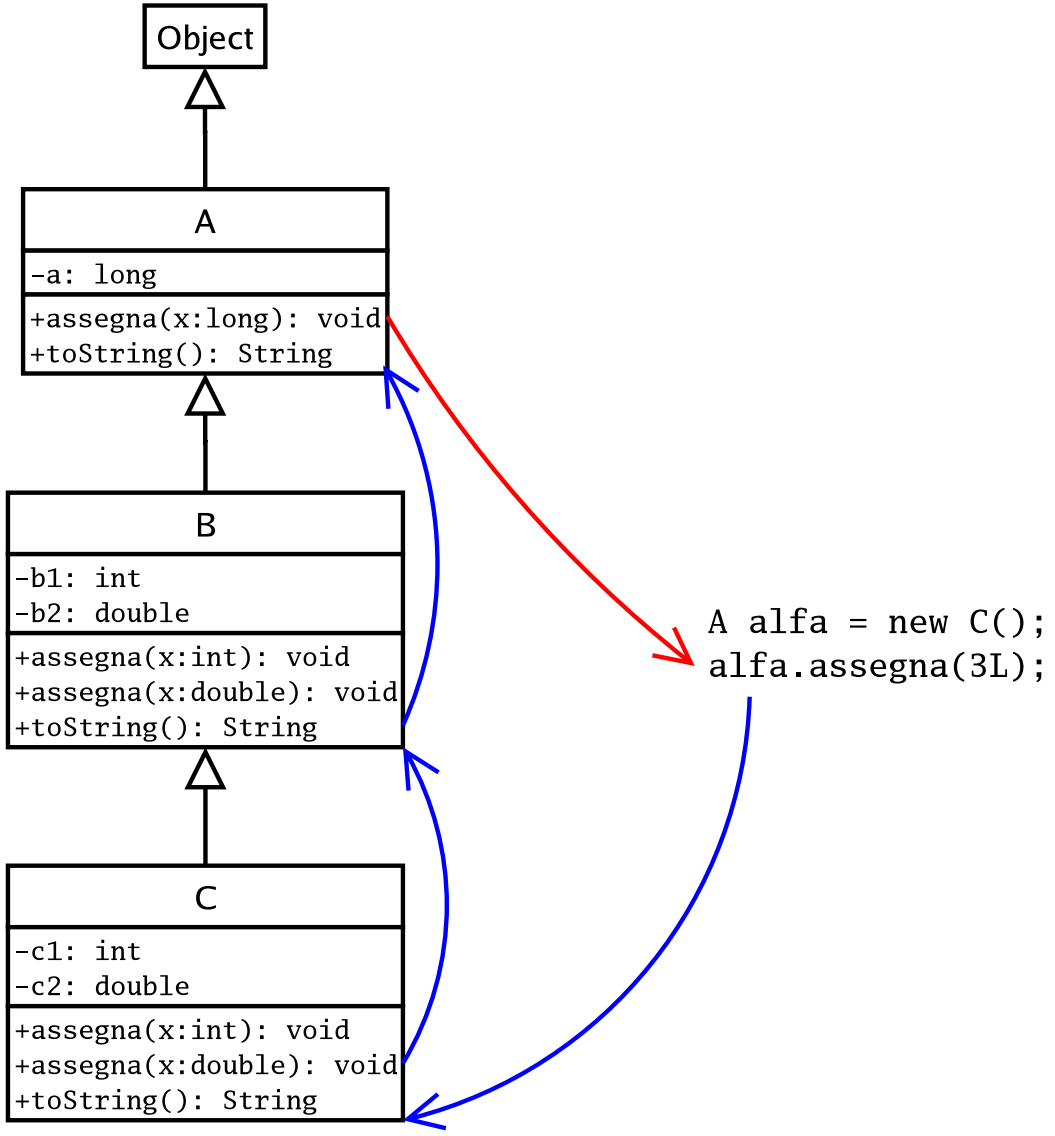
- In questi casi il compilatore segnala un messaggio di errore.

Fase di esecuzione: scelta del metodo

La JVM sceglie il metodo da eseguire in base al tipo dell'oggetto.

- Cerca un metodo la cui segnatura sia esattamente quella selezionata in fase di compilazione.
- Cerca tale metodo risalendo la gerarchia delle classi a partire dalla classe corrispondente all'oggetto che deve eseguire il metodo.

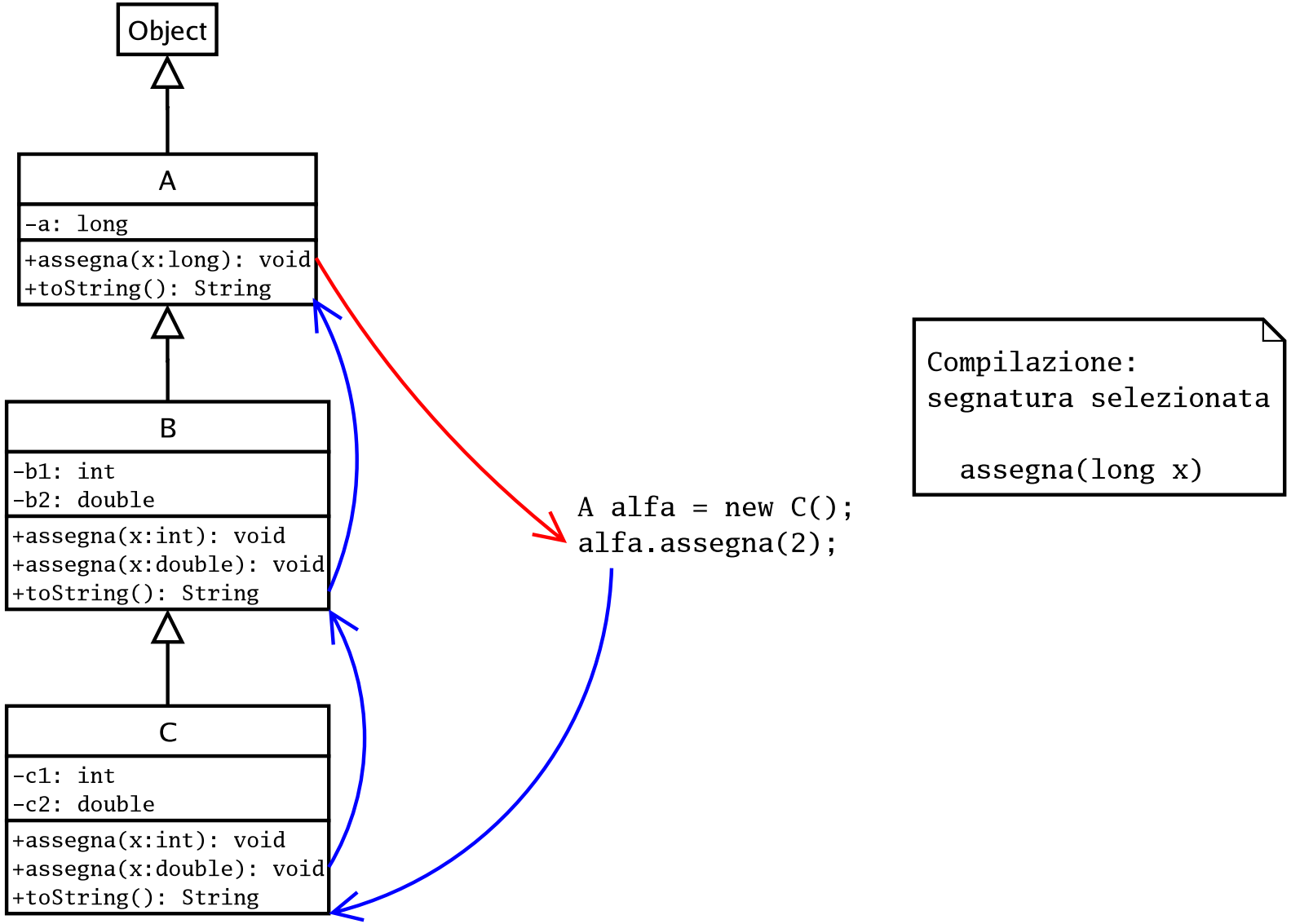
Esempio



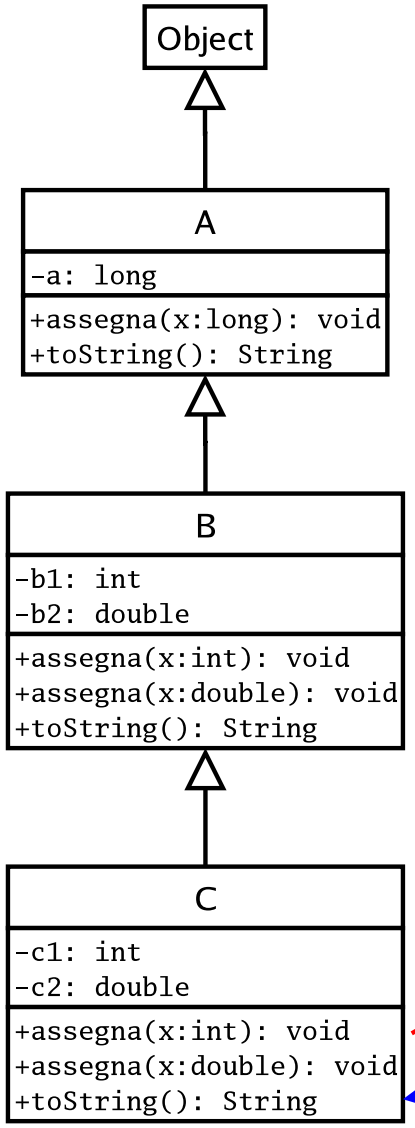
Compilazione:
segnatura selezionata

assegna(long x)

Esempio


Object
A
-a: long
+assegna(x:long): void
+toString(): String
B
-b1: int
-b2: double
+assegna(x:int): void
+assegna(x:double): void
+toString(): String
C
-c1: int
-c2: double
+assegna(x:int): void
+assegna(x:double): void
+toString(): String
A alfa = new C();
alfa.assegna(2);
Compilazione:
segnatura selezionata
assegna(long x)

Esempio

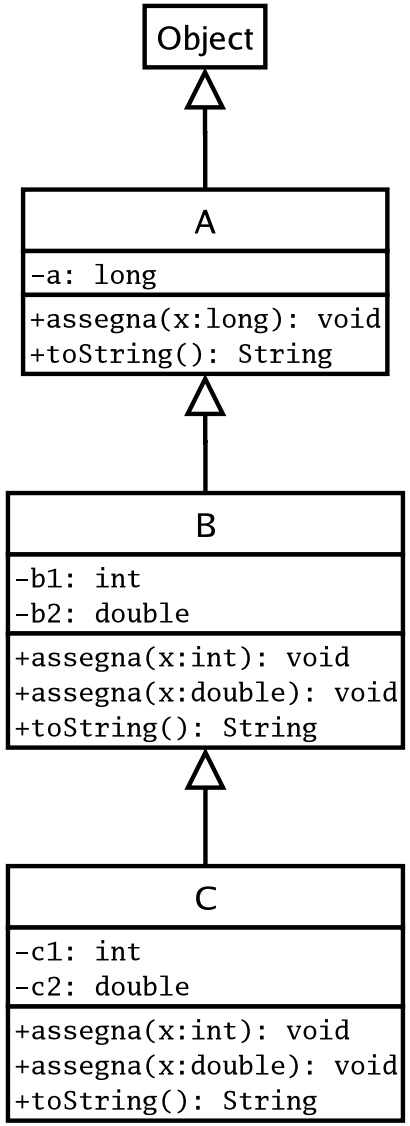



```
B beta = new C();
beta.assegna(2);
```

```
Compilazione:
segnatura selezionata

  assegna(int x)
```

Esempio




```
G gamma = new C();
gamma.assegna(2.0);
```

```
Compilazione:
segnatura selezionata

  assegna(double x)
```

Il metodo `equals`

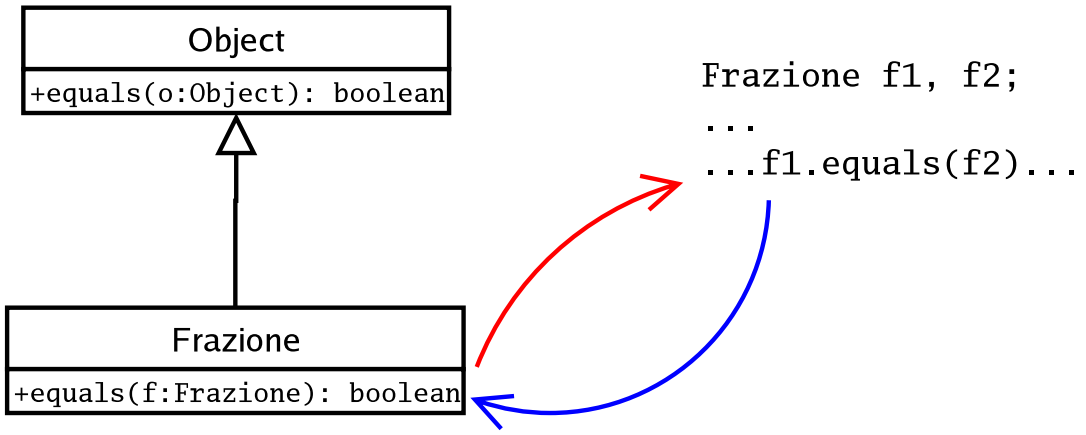
Il metodo `equals` di `Object`

Il metodo `equals` di `Object` ha un comportamento “banale” basato sull’assunzione che un oggetto è senz’altro uguale a se stesso.

```
public boolean equals(Object altro) {
  return this == altro;
}
```

Esempio

```
Frazione f1, f2;
...
if (f1.equals(f2))
  out.println("Le due frazioni sono uguali");
else
  out.println("Le due frazioni sono diverse");
}
```

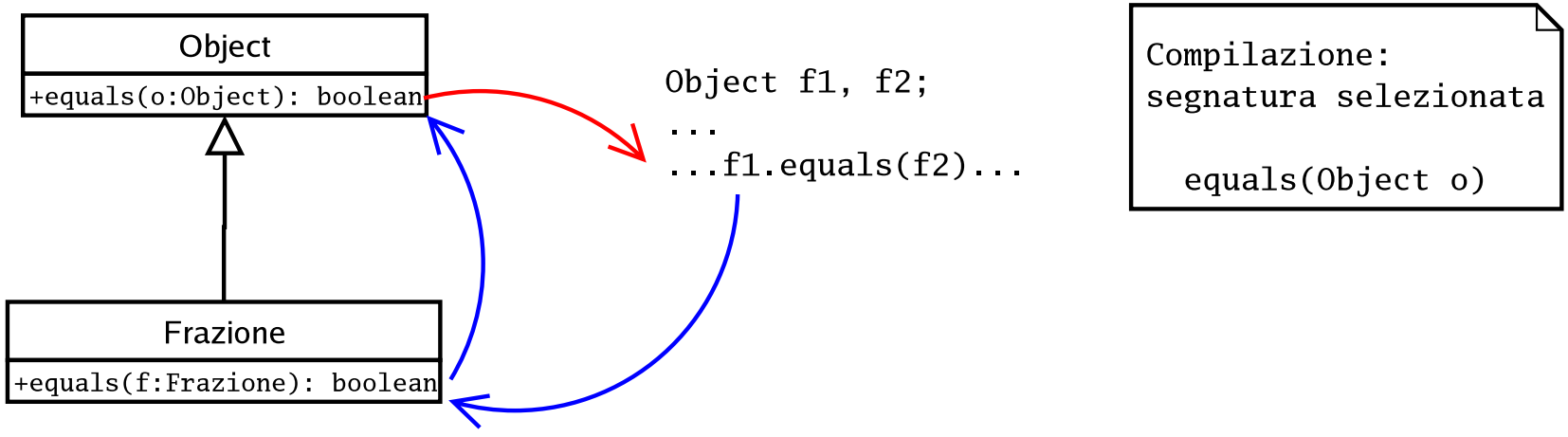


Compilazione:
segnatura selezionata

equals(Frazione f)

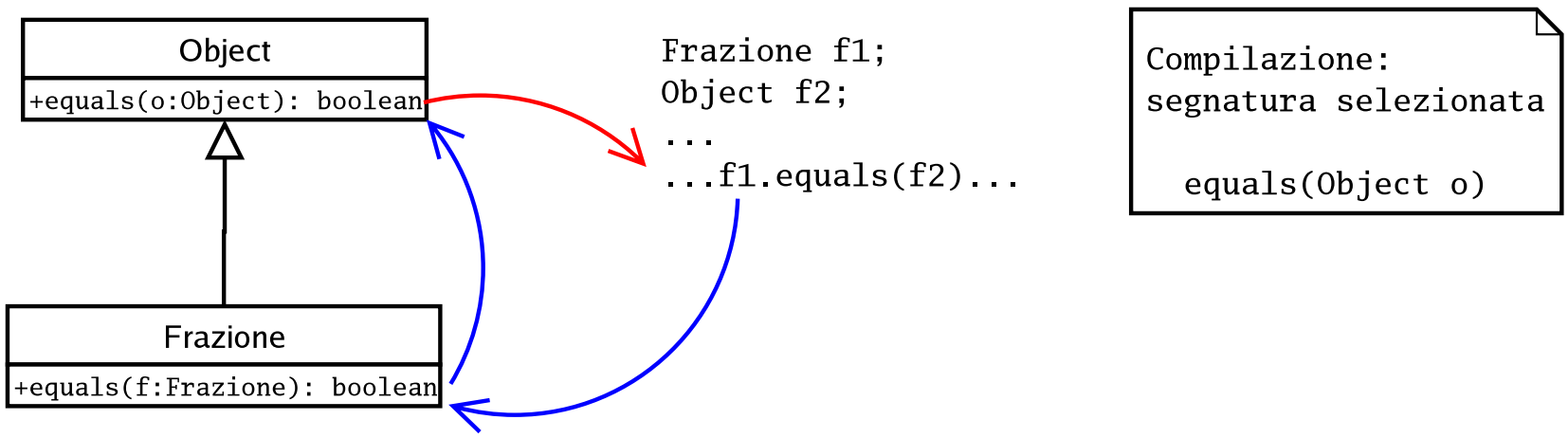
Esempio

```
Object f1, f2;
//memorizzo in f1 e f2 degli oggetti di tipo Frazione
...
if (f1.equals(f2))
  out.println("Le due frazioni sono uguali");
else
  out.println("Le due frazioni sono diverse");
}
```



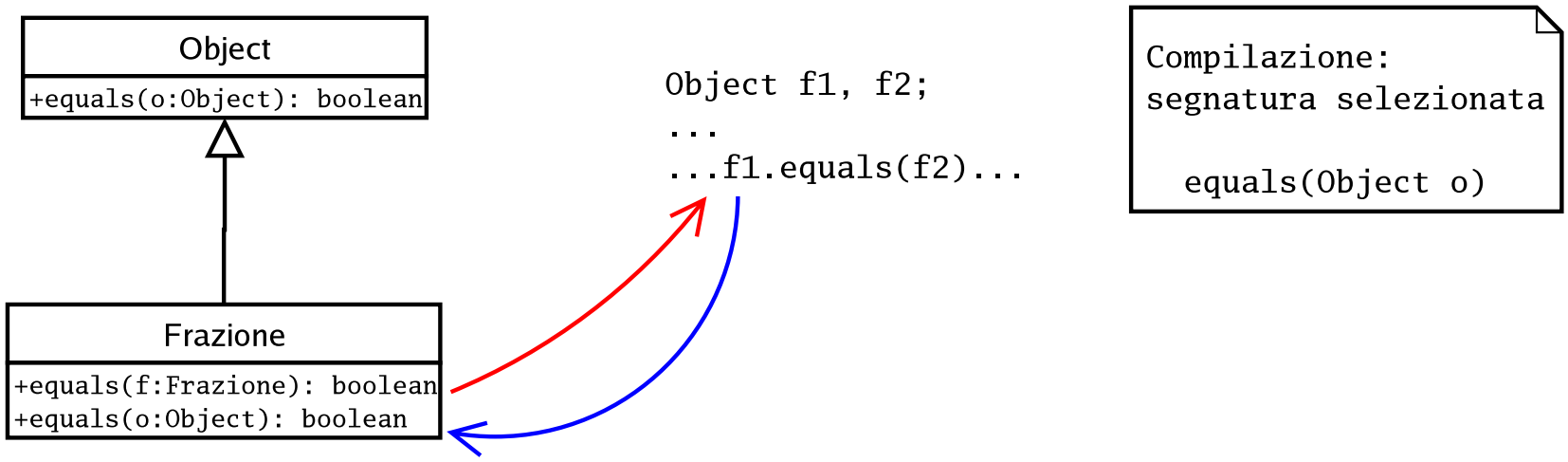
Esempio

```
Frazione f1;
Object f2;
//memorizzo in f2 un oggetto di tipo Frazione
...
if (f1.equals(f2))
  out.println("Le due frazioni sono uguali");
else
  out.println("Le due frazioni sono diverse");
}
```



Esempio

- Se vogliamo che anche in questo caso il comportamento sia corretto dobbiamo sovrascrivere il metodo `equals(Object o)` nella classe `Frazione`.



Esempio: il metodo `cerca`

```
public static int cerca(Object[] o, Object chiave) {
  boolean trovato = false;
  int i;
  for (i = 0; i < o.length && !trovato; i++)
    if (chiave.equals(o[i]))
      trovato = true;

  if (trovato)
    return i - 1;
  else return -1;
  }
```

- In fase di compilazione viene selezionata la segnatura `equals(Object o)`.
- Se vogliamo utilizzarlo su oggetti di tipo `Frazione` dobbiamo definire un metodo con tale segnatura in `Frazione`.

Classe `Frazione`

```
public boolean equals(Frazione f) {
  return this.num == f.num && this.den == f.den;
}

public boolean equals(Object o) {
  if (o instanceof Frazione) {
    Frazione altra = (Frazione) o;
    return this.equals(altra);
  } else
    return false;
}
```