Capitolo 8

Estensione delle classi

Sommario: Estensione delle classi

Estensione di una classe

È possibile estendere una classe senza conoscere nulla dell'implementazione della classe estesa, ma solo il contratto

Estensione di una classe

È possibile estendere una classe senza conoscere nulla dell'implementazione della classe estesa, ma solo il contratto

Esempio

```
class Quadrato extends Rettangolo {
    ...
}
```

`extends` indica che:

- la classe `Quadrato` è ottenuta dalla classe `Rettangolo` estendendone stato e comportamento

Estensione di una classe

È possibile estendere una classe senza conoscere nulla dell'implementazione della classe estesa, ma solo il contratto

Esempio

```
class Quadrato extends Rettangolo {
    ...
}
```

`extends` indica che:

- la classe `Quadrato` è ottenuta dalla classe `Rettangolo` estendendone stato e comportamento
- `Quadrato` è una sottoclasse di `Rettangolo`

Estensione di una classe

È possibile estendere una classe senza conoscere nulla dell'implementazione della classe estesa, ma solo il contratto

Esempio

```
class Quadrato extends Rettangolo {
    ...
}
```

`extends` indica che:

- la classe `Quadrato` è ottenuta dalla classe `Rettangolo` estendendone stato e comportamento
- `Quadrato` è una sottoclasse di `Rettangolo`
- `Rettangolo` è una superclasse di `Quadrato`

Ereditarietà

Esempio

```
class Quadrato extends Rettangolo {
    ...
}
```

- La classe `Quadrato` eredita (e quindi dispone) di tutti
  - i campi
  - i metodi

  definiti nella classe `Rettangolo`

Ereditarietà

Esempio

```
class Quadrato extends Rettangolo {
    ...
}
```

- La classe `Quadrato` eredita (e quindi 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)

Ereditarietà

Esempio

```
class Quadrato extends Rettangolo {
    ...
}
```

- La classe `Quadrato` eredita (e quindi 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

I costruttori della sottoclasse

Costruzione di un oggetto

La costruzione di un oggetto di una sottoclasse avviene costruendo un oggetto della superclasse e adattandolo alle esigenze della sottoclasse

I costruttori della sottoclasse

Costruzione di un oggetto

La costruzione di un oggetto di una sottoclasse avviene costruendo un oggetto della superclasse e adattandolo alle esigenze della sottoclasse

Esempio

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

- `super`
  Per invocare un costruttore della superclasse nel costruttore di una sottoclasse diretta

I costruttori della sottoclasse

Costruzione di un oggetto

La costruzione di un oggetto di una sottoclasse avviene costruendo un oggetto della superclasse e adattandolo alle esigenze della sottoclasse

Esempio

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

- `super`
  Per invocare un costruttore della superclasse nel costruttore di una sottoclasse diretta
- In caso di overloading, il costruttore invocato viene riconosciuto in base alla lista degli argomenti fornita a `super`

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();
  }

}
```

Costruttori e gerarchia delle classi

Generalmente un costruttore non opera da solo, ma si avvale dell'ausilio di:

(1) un altro costruttore della stessa classe

```
public Frazione(int x) {
  this(x, 1);
}
```

Costruttori e gerarchia delle classi

Generalmente un costruttore non opera da solo, ma si avvale dell'ausilio di:

(1) un altro costruttore della stessa classe

```
public Frazione(int x) {
  this(x, 1);
}
```

(2) un costruttore della superclasse

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

Costruttori e gerarchia delle classi

Generalmente un costruttore non opera da solo, ma si avvale dell'ausilio di:

(1) un altro costruttore della stessa classe

```
public Frazione(int x) {
  this(x, 1);
}
```

(2) un costruttore della superclasse

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

L'invocazione di un costruttore tramite `this` o `super` dev'essere la prima istruzione del costruttore

Costruttori

Se in una classe non si definiscono costruttori

Costruttori

Se in una classe non si definiscono costruttori

viene automaticamente aggiunto un costruttore privo di argomenti (e privo di codice)

Costruttori

Se in una classe non si definiscono costruttori

viene automaticamente aggiunto un costruttore privo di argomenti (e privo di codice)

Se un costruttore non ne invoca esplicitamente un altro tramite `this` o `super`

Costruttori

Se in una classe non si definiscono costruttori

viene automaticamente aggiunto un costruttore privo di argomenti (e privo di codice)

Se un costruttore non ne invoca esplicitamente un altro tramite `this` o `super`

come prima istruzione viene automaticamente invocato il costruttore privo di argomenti della superclasse

Costruttori

Se in una classe non si definiscono costruttori

viene automaticamente aggiunto un costruttore privo di argomenti (e privo di codice)

Se un costruttore non ne invoca esplicitamente un altro tramite `this` o `super`

come prima istruzione viene automaticamente invocato il costruttore privo di argomenti della superclasse

Eccezione

Il costruttore della classe `Object` predispone l'oggetto senza alcun ausilio.

Esempio

Nell'implementazione della classe `A` non è stato scritto esplicitamente alcun costruttore...

Esempio

Nell'implementazione della classe `A` non è stato scritto esplicitamente alcun costruttore...

...il compilatore aggiunge:

```
public A() {
  super();
}
```

Esempio

Nell'implementazione della classe `A` non è stato scritto esplicitamente alcun costruttore...

...il compilatore aggiunge:

```
public A() {
  super();
}
```

Importante

Se la superclasse di `A` non dispone di un costruttore privo di argomenti, il compilatore segnalerà un errore

Esempio

- La classe `Quadrato` ha un costruttore con un argomento.

  Quindi il costruttore privo di argomenti non viene aggiunto automaticamente

Esempio

- La classe `Quadrato` ha un costruttore con un argomento.

  Quindi il costruttore privo di argomenti non viene aggiunto automaticamente

- Se aggiungiamo il costruttore:

```
public Quadrato() {
}
```

  il compilatore segnalerà un errore

Esempio

- La classe `Quadrato` ha un costruttore con un argomento.

  Quindi il costruttore privo di argomenti non viene aggiunto automaticamente

- Se aggiungiamo il costruttore:

```
public Quadrato() {
}
```

  il compilatore segnalerà un errore

> questo costruttore invoca (implicitamente) `super()`, ma `Rettangolo` non ha un costruttore privo di argomenti

Esempio: `Quadrato`

Una possibile definizione del costruttore privo di argomenti

Esempio: Quadrato

Una possibile definizione del costruttore privo di argomenti

```
public Quadrato() {
  this(0);
}
```

Esempio: Quadrato

Una possibile definizione del costruttore privo di argomenti

```
public Quadrato() {
  this(0);
}
```

oppure

Esempio: Quadrato

Una possibile definizione del costruttore privo di argomenti

```
public Quadrato() {
  this(0);
}
```

oppure

```
public Quadrato() {
  super(0, 0);
}
```

super: due utilizzi...

(1) Come costruttore

Per richiamare un costruttore della superclasse nel codice del costruttore della sottoclasse

super: due utilizzi...

(1) Come costruttore

Per richiamare un costruttore della superclasse nel codice del costruttore della sottoclasse

(2) Come pseudovariabile

(a) per invocare un metodo della superclasse nel codice della sottoclasse

super: due utilizzi...

(1) Come costruttore

Per richiamare un costruttore della superclasse nel codice del costruttore della sottoclasse

(2) Come pseudovariabile

(a) per invocare un metodo della superclasse nel codice della sottoclasse

la ricerca del metodo da eseguire non parte dalla classe effettiva dell'oggetto, ma dalla superclasse della classe in cui è scritta l'invocazione stessa

super: due utilizzi...

(1) Come costruttore

Per richiamare un costruttore della superclasse nel codice del costruttore della sottoclasse

(2) Come pseudovariabile

(a) per invocare un metodo della superclasse nel codice della sottoclasse

> la ricerca del metodo da eseguire non parte dalla classe effettiva dell'oggetto, ma dalla superclasse della classe in cui è scritta l'invocazione stessa

(b) per accedere ai campi della superclasse

Esempio

Potremmo aggiungere a `Quadrato` i metodi:

```
public String getDescrizione1() {
  return this.toString();
}
```

Esempio

Potremmo aggiungere a Quadrato i metodi:

```
public String getDescrizione1() {
  return this.toString();
}


public String getDescrizione2() {
  return super.toString();
}
```

Overloading

Overloading

La possibilità di avere metodi o costruttori con lo stesso nome ma segnatura diversa.

Overloading

Overloading

La possibilità di avere metodi o costruttori con lo stesso nome ma segnatura diversa.

- Segnatura = nome del metodo e lista dei tipi dei suoi argomenti

Overloading

Overloading

La possibilità di avere metodi o costruttori con lo stesso nome ma segnatura diversa.

- Segnatura = nome del metodo e lista dei tipi dei suoi argomenti
- L'overloading viene risolto in fase di compilazione

Esempio

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

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);
}
```

Compilazione: scelta della segnatura

In compilazione viene scelta la segnatura del metodo da eseguire in base:

(1) al tipo del riferimento utilizzato per invocare il metodo

Compilazione: scelta della segnatura

In compilazione viene scelta la segnatura del metodo da eseguire in base:

(1) al tipo del riferimento utilizzato per invocare il metodo

(2) al tipo degli argomenti indicati nella chiamata

Compilazione: scelta della segnatura

In compilazione viene scelta la segnatura del metodo da eseguire in base:

(1) al tipo del riferimento utilizzato per invocare il metodo

(2) al tipo degli argomenti indicati nella chiamata

Esempio

```
A r;
...
r.m(2)
```

Il compilatore cerca fra tutte le segnature di metodi di nome `m` disponibili per il tipo `A` ...

Compilazione: scelta della segnatura

In compilazione viene scelta la segnatura del metodo da eseguire in base:

(1) al tipo del riferimento utilizzato per invocare il metodo

(2) al tipo degli argomenti indicati nella chiamata

Esempio

```
A r;
...
r.m(2)
```

Il compilatore cerca fra tutte le segnature di metodi di nome `m` disponibili per il tipo `A` ...

... quella "più adatta" per gli argomenti specificati

Esempio

```
A r;
...
r.m(2)
```

Se le segnature disponibili per il tipo `A` sono:

```
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.

Overriding

Overriding

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

L'overriding viene risolto in fase di esecuzione

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 stabilisce la segnatura del metodo da eseguire (*early binding*)

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 stabilisce la segnatura del metodo da eseguire (*early binding*)

Esecuzione
scelta del metodo

Il metodo da eseguire, tra quelli con la segnatura selezionata, viene scelto al momento dell'esecuzione, sulla base del tipo dell'oggetto (*late binding*)

Fase di compilazione

(1) Scelta delle segnature "candidate"

Il compilatore individua le segnature che possono soddisfare la chiamata

Fase di compilazione

(1) Scelta delle segnature "candidate"

Il compilatore individua le segnature che possono soddisfare la chiamata

(a) 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

(b) accessibile al codice chiamante

Fase di compilazione

(1) Scelta delle segnature "candidate"

Il compilatore individua le segnature che possono soddisfare la chiamata

(a) 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

(b) accessibile al codice chiamante

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

Fase di compilazione

(1) Scelta delle segnature "candidate"

Il compilatore individua le segnature che possono soddisfare la chiamata

(a) 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

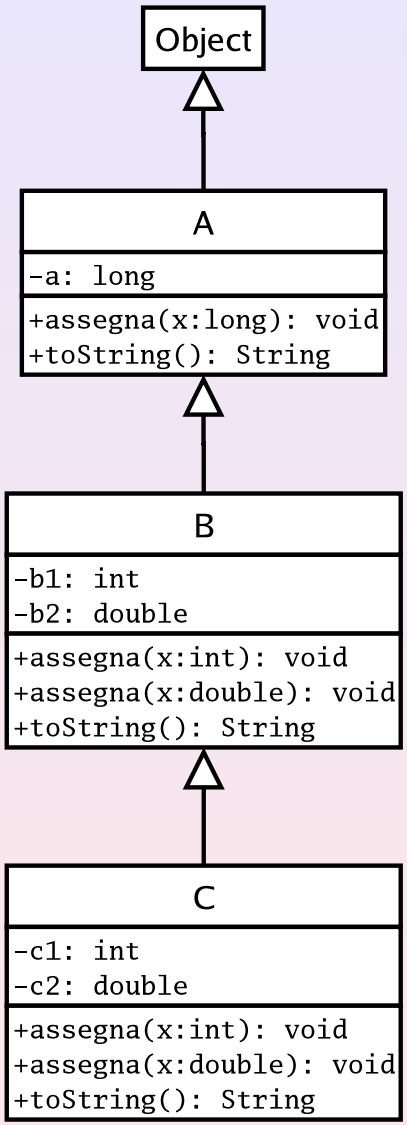
(b) accessibile al codice chiamante

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

(2) Scelta della segnatura "più specifica"

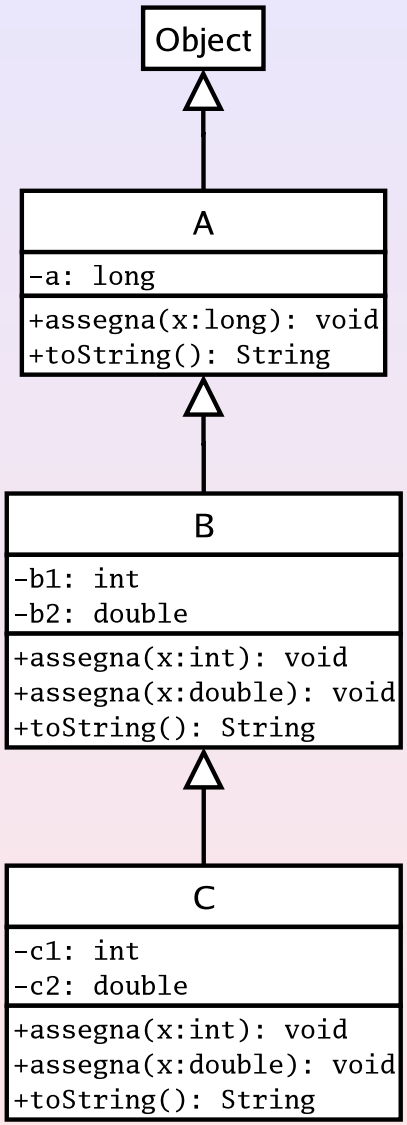
Tra le segnature candidate, il compilatore seleziona quella che richiede il minor numero di promozioni

Fase di compilazione: scelta della segnatura




```
C gamma;
```

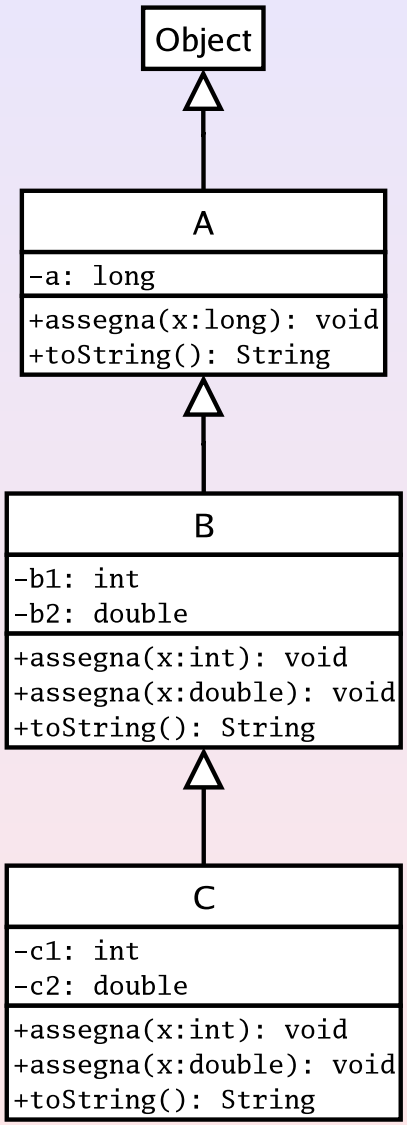
Fase di compilazione: scelta della segnatura




```
C gamma;
```

- `gamma.toString()`

Fase di compilazione: scelta della segnatura

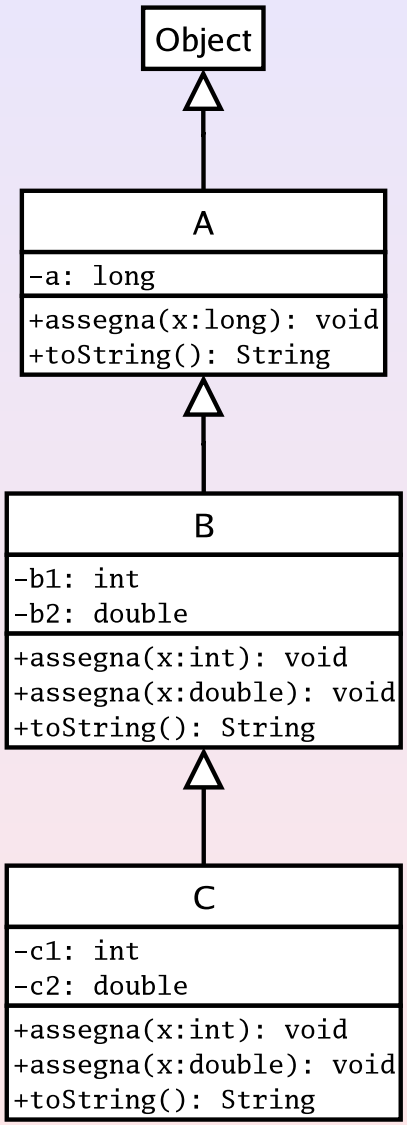



```
C gamma;
```

- `gamma.toString()`

  Una segnatura candidata: `toString`

Fase di compilazione: scelta della segnatura



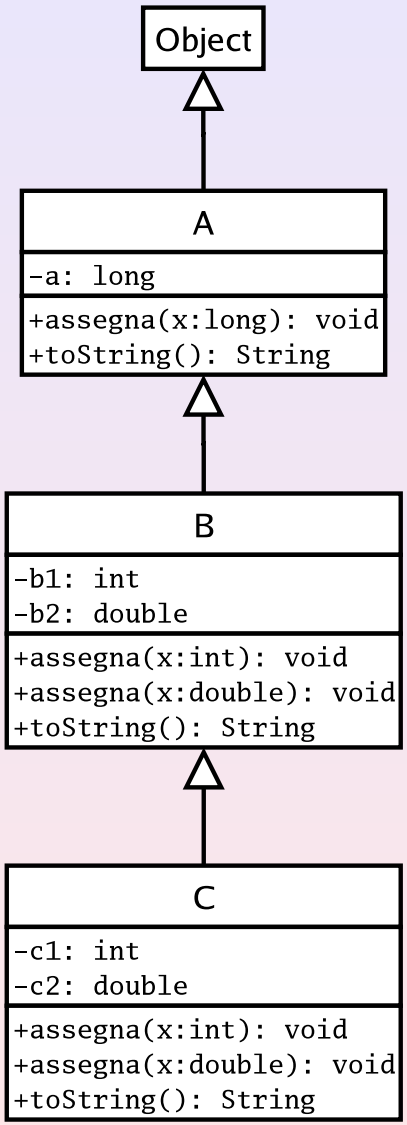

```
C gamma;
```

- `gamma.toString()`

  Una segnatura candidata: `toString`

- `gamma.equals(alfa)`

Fase di compilazione: scelta della segnatura




```
C gamma;
```

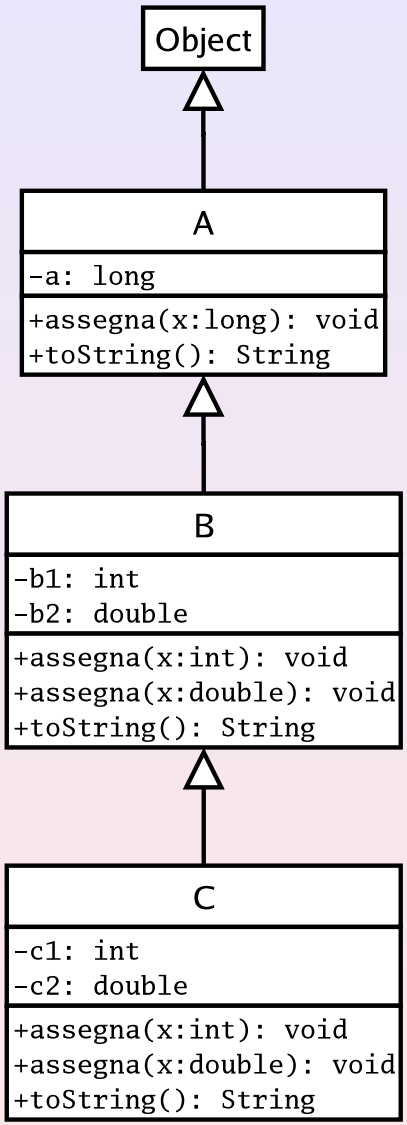
- `gamma.toString()`

  Una segnatura candidata: `toString`

- `gamma.equals(alfa)`

  `equals` con parametro di tipo `Object` (ereditato da `Object`)

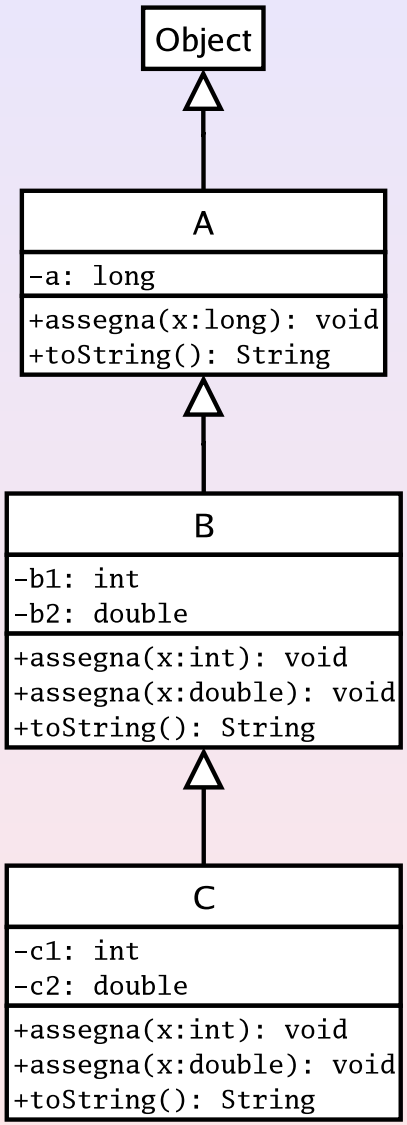
Fase di compilazione: scelta della segnatura




```
A alfa;
```

- `alfa.assegna(2)`

Fase di compilazione: scelta della segnatura

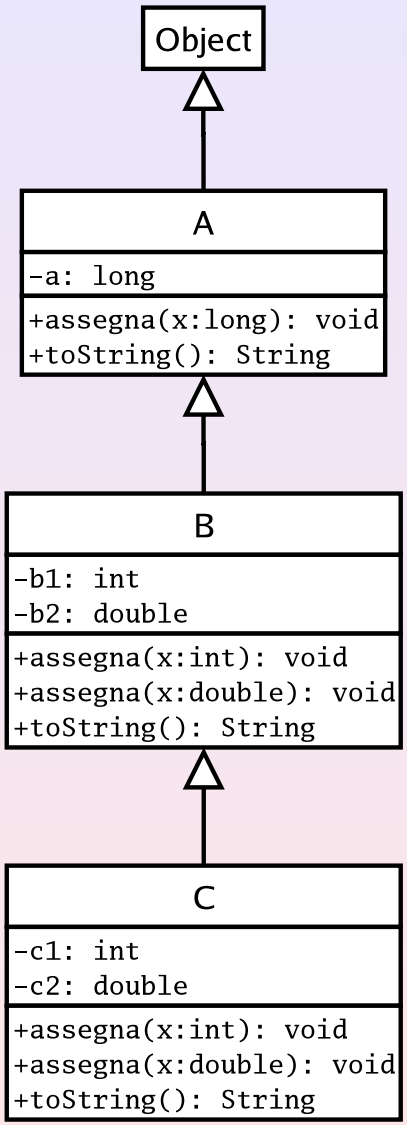



```
A alfa;
```

- `alfa.assegna(2)`

  Una segnatura candidata: `assegna(long x)`

Fase di compilazione: scelta della segnatura




```
A alfa;
```

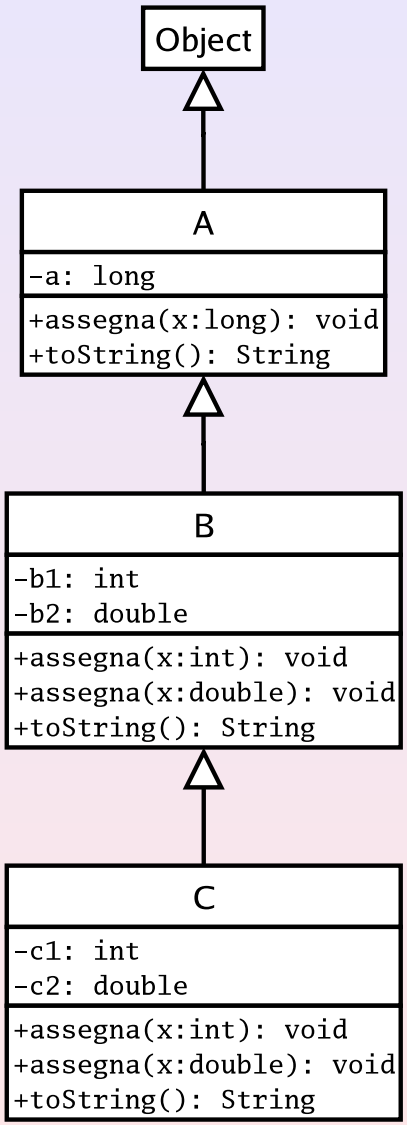
- `alfa.assegna(2)`

  Una segnatura candidata: `assegna(long x)`

- `alfa.assegna(2.0)`

  Nessuna segnatura candidata (errore)

Fase di compilazione: scelta della segnatura



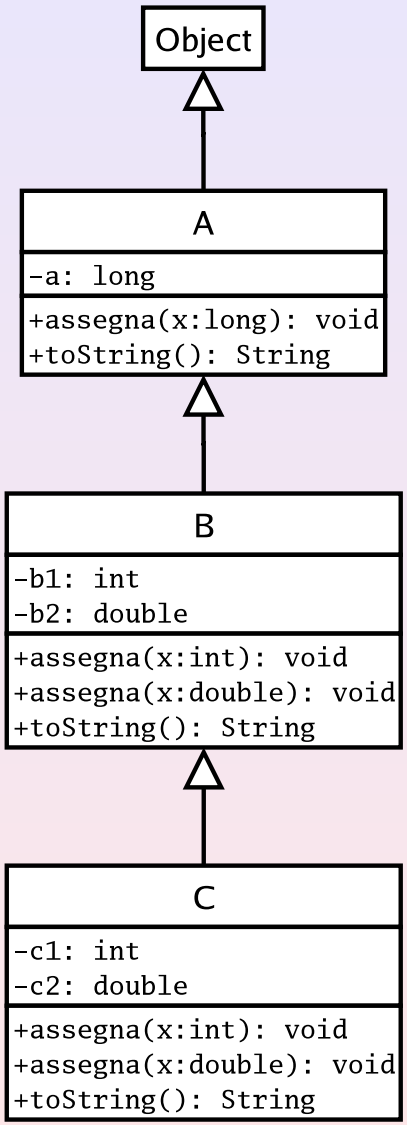

```
B beta;
```

- `beta.assegna(2)`

Tre segnature candidate:

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

Fase di compilazione: scelta della segnatura




```
B beta;
```

- `beta.assegna(2)`

Tre segnature candidate:

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

La più specifica è `assegna(int x)`

Ambiguità

Se per l'invocazione:

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

le segnature candidate sono:

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

Il compilatore non è in grado di individuare la segnatura più specifica e segnala un messaggio di errore

Fase di esecuzione: scelta del metodo

La JVM sceglie il metodo da eseguire sulla base del tipo dell'oggetto usato nell'invocazione

Fase di esecuzione: scelta del metodo

La JVM sceglie il metodo da eseguire sulla base del tipo dell'oggetto usato nell'invocazione

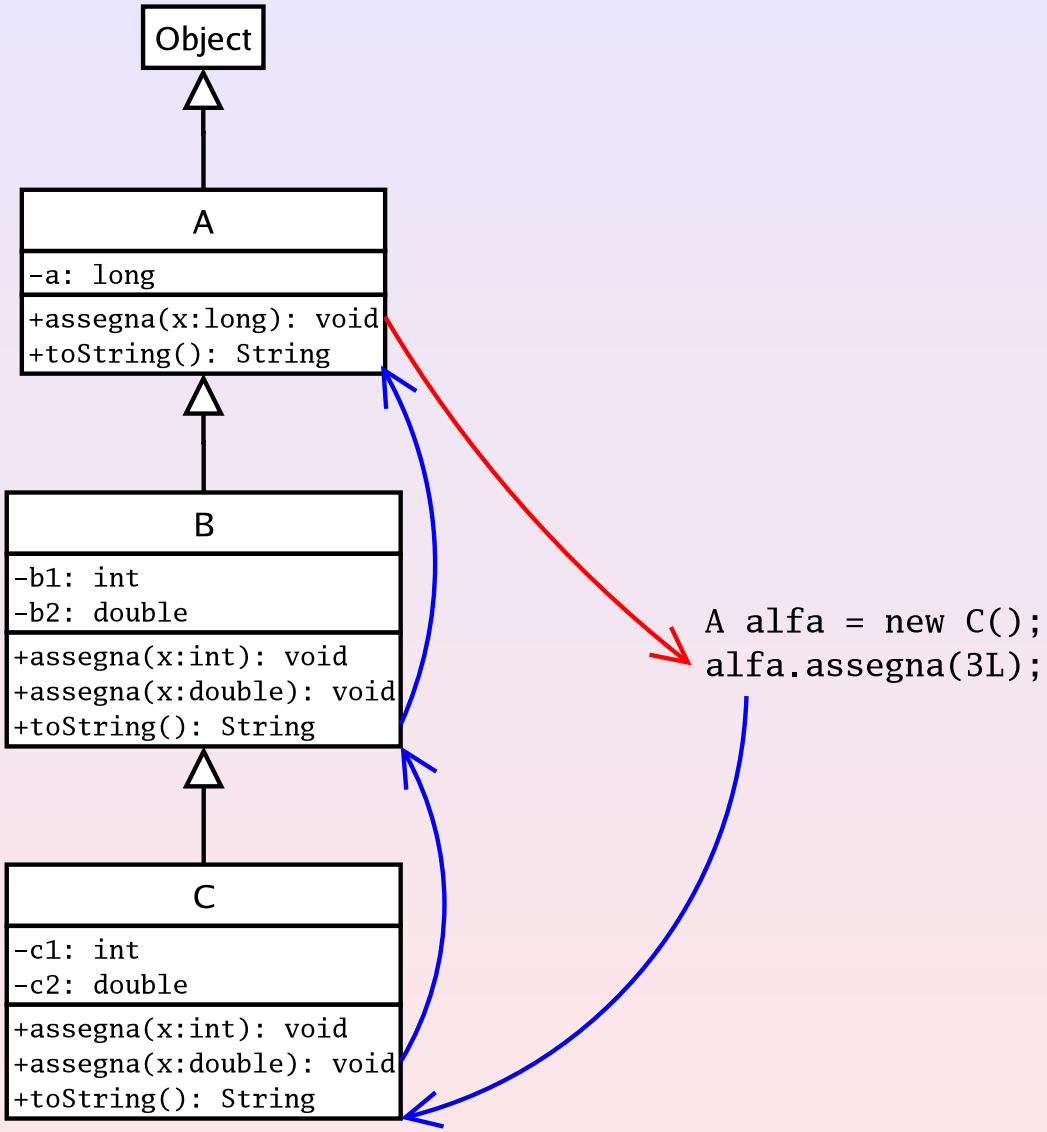
- cerca un metodo con la segnatura selezionata in fase di compilazione

Fase di esecuzione: scelta del metodo

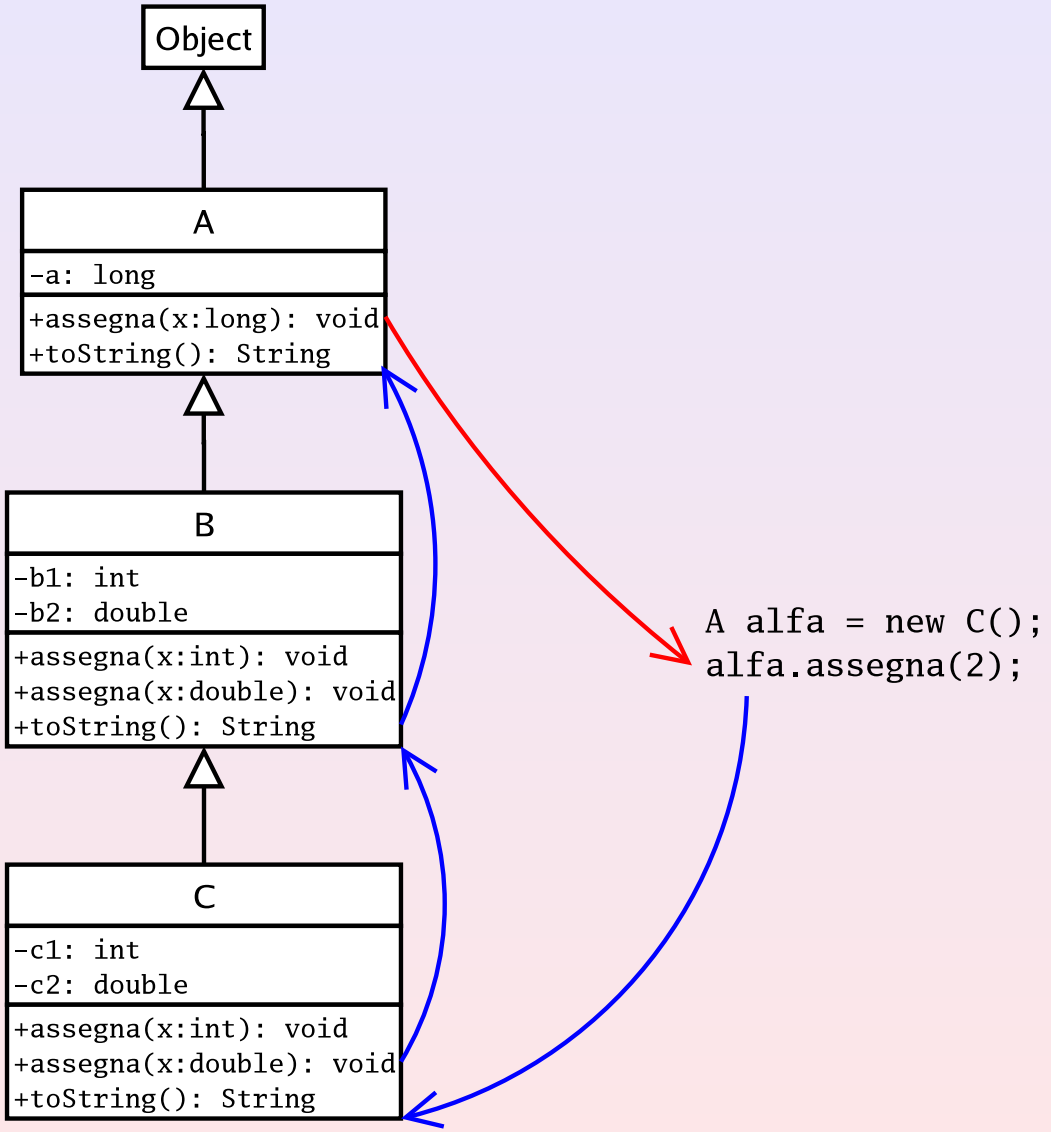
La JVM sceglie il metodo da eseguire sulla base del tipo dell'oggetto usato nell'invocazione

- cerca un metodo con la segnatura selezionata in fase di compilazione
- risalendo la gerarchia delle classi a partire dalla classe dell'oggetto che deve eseguire il metodo

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(3L);
Compilazione:
segnatura selezionata
assegna(long x)

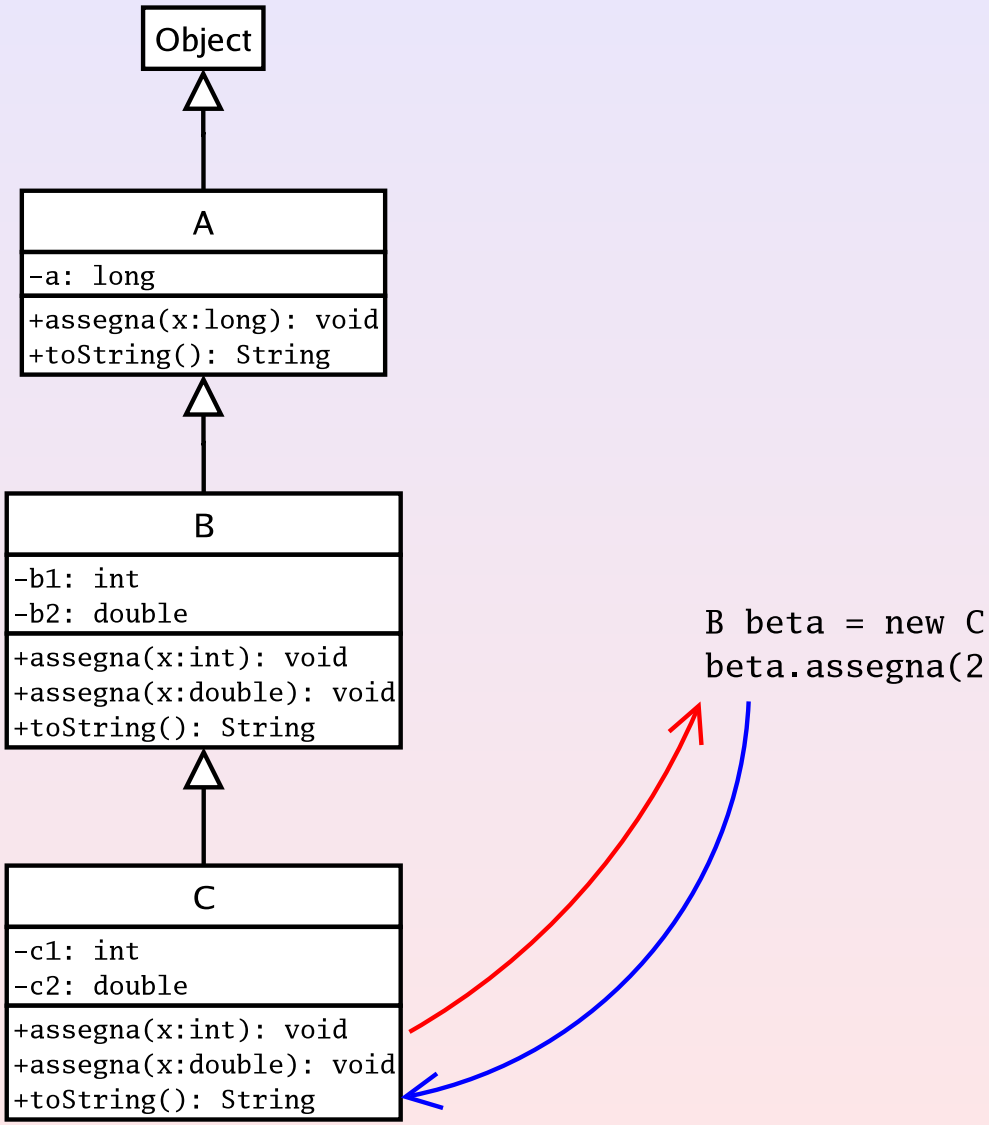
Esempio



Compilazione:
segnatura selezionata

assegna(long x)

Esempio

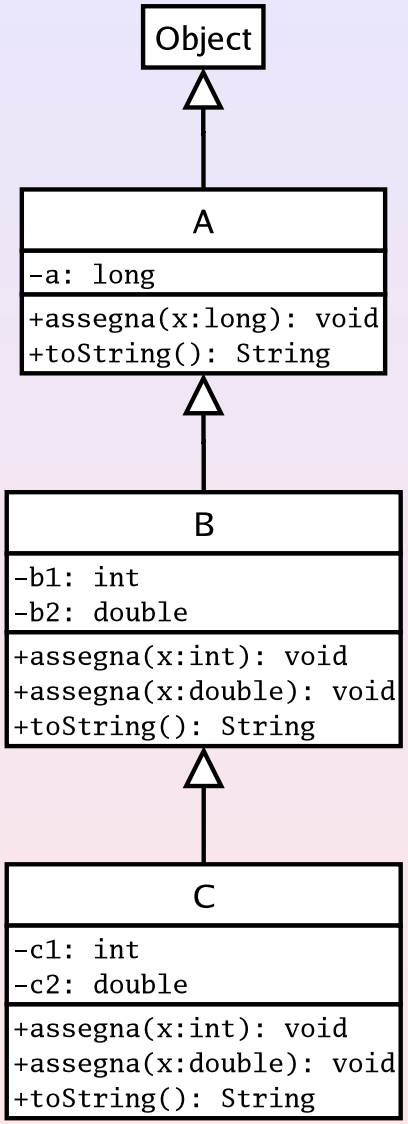



```
B beta = new C();
beta.assegna(2);
```

Compilazione:
segnatura selezionata

```
assegna(int x)
```

Esempio




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

Compilazione:
segnatura selezionata

assegna(double x)

Il metodo equals

- Classe `Object`: `public boolean equals(Object o)`

Il metodo equals

- Classe `Object`: `public boolean equals(Object o)`
  Basato sull'assunzione che un oggetto è uguale a se stesso.

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

Il metodo equals

- Classe `Object`: `public boolean equals(Object o)`
  Basato sull'assunzione che un oggetto è uguale a se stesso.

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

- Classe `Frazione`: `public boolean equals(Frazione f)`

Il metodo equals

- Classe `Object`: `public boolean equals(Object o)`
  Basato sull'assunzione che un oggetto è uguale a se stesso.

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

- Classe `Frazione`: `public boolean equals(Frazione f)`

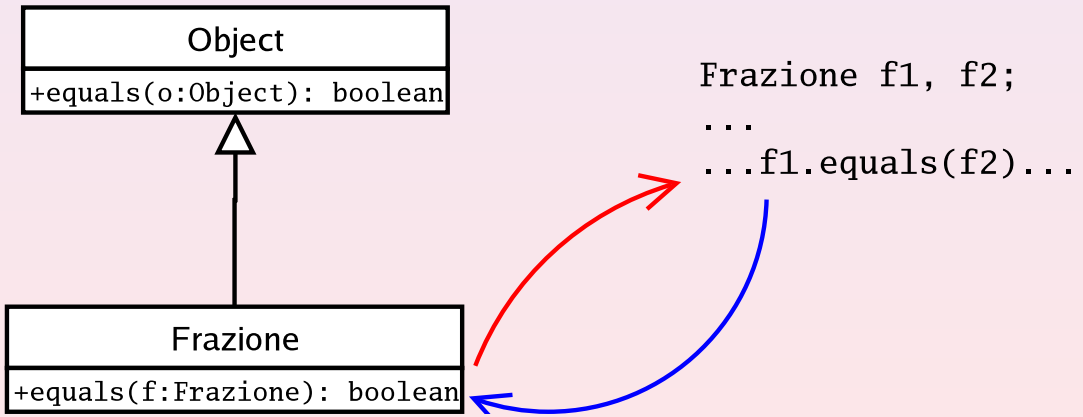
```
public boolean equals(Frazione f) {
  return this.num == f.num && this.den == f.den;
}
```

Esempio

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

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

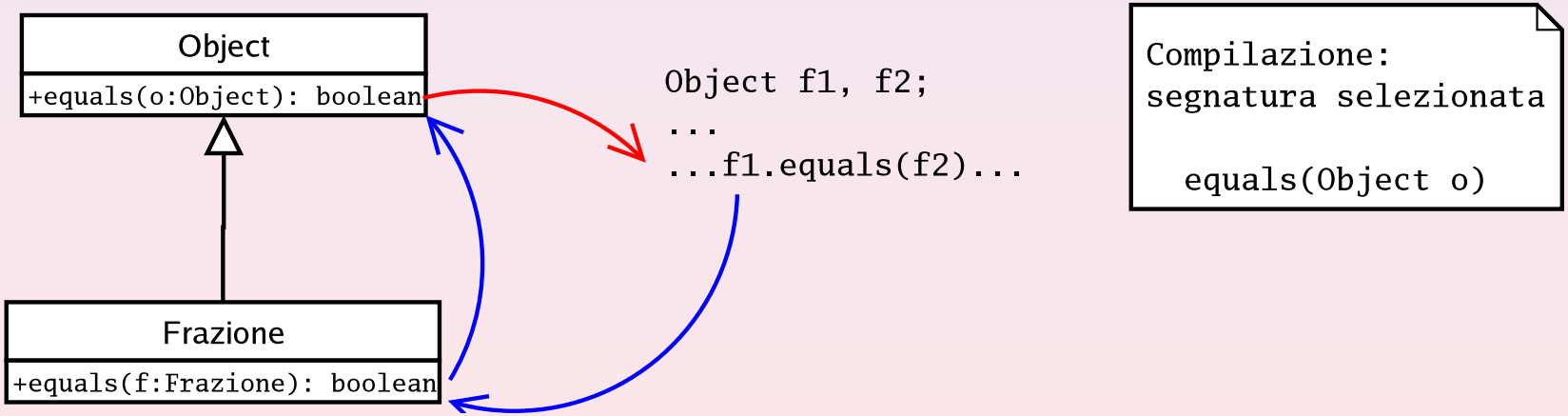
equals(Frazione f)

Esempio

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

Esempio

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

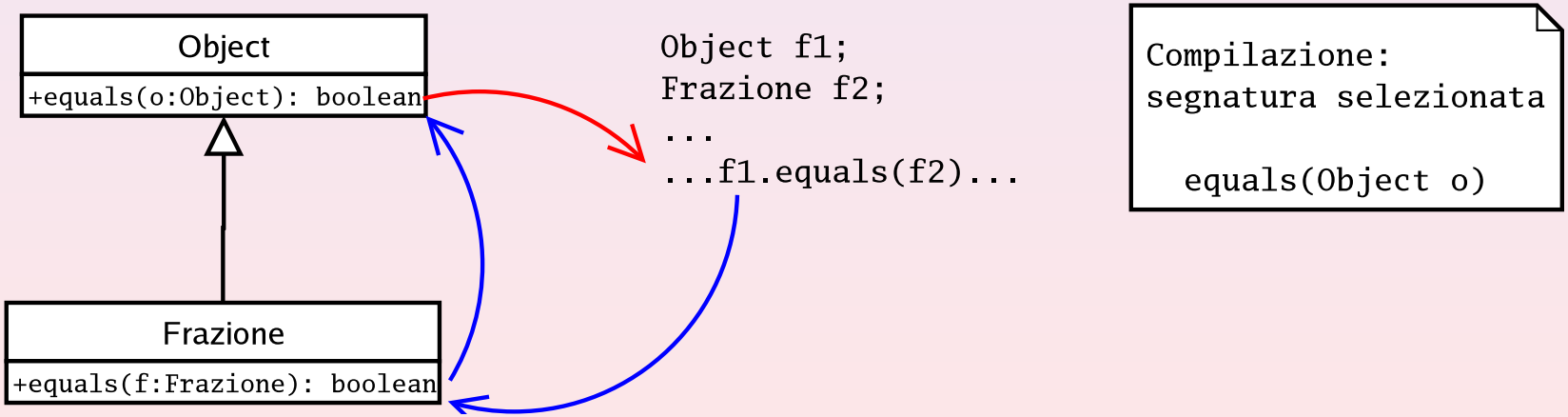


Esempio

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

Esempio

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

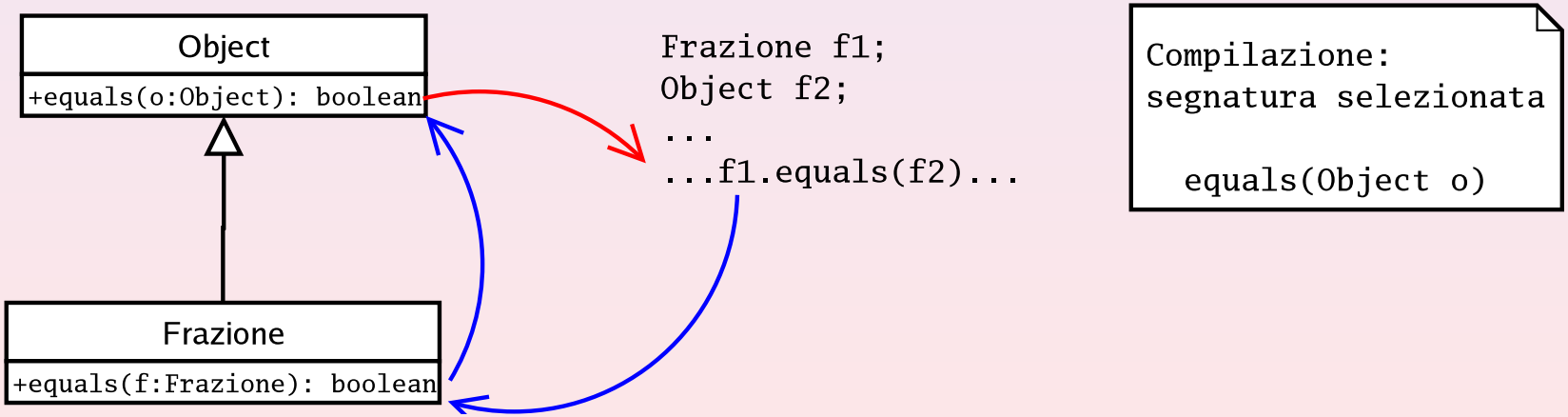


Esempio

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

Esempio

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

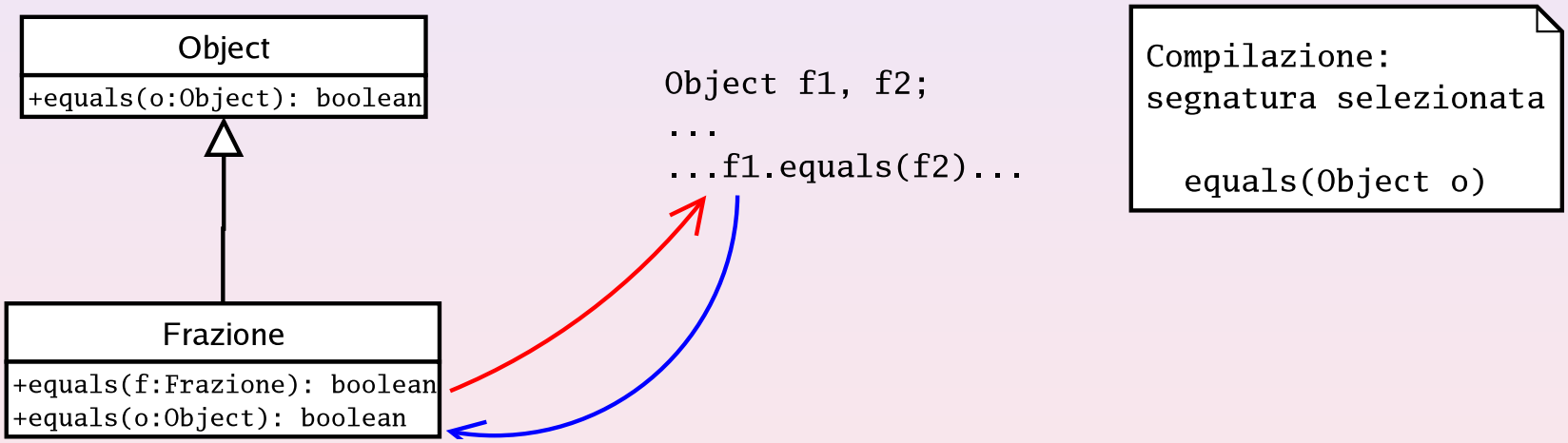


Esempio

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

Esempio

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



I metodi equals di Frazione

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

I metodi equals di 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;
}
```

Sommario: Estensione delle classi

Variabili e adombramento

```
public class Sopra {
  int k = 1;

  public String toString() {
    return String.valueOf(k);
  }
}
```

Variabili e adombramento

```
public class Sopra {
  int k = 1;

  public String toString() {
    return String.valueOf(k);
  }
}
```

```
public class Sotto extends Sopra {
  int k = 2;

  public String toString() {
    return k + ", " + super.toString();
  }
}
```

Variabili e adombramento

```
public class Sopra {
  int k = 1;

  public String toString() {
    return String.valueOf(k);
  }
}
```

```
public class Sotto extends Sopra {
  int k = 2;

  public String toString() {
    return k + ", " + super.toString();
  }
}
```

- Sotto: la variabile k adombra la variabile k di Sopra

Variabili e adombramento

```
public class Sopra {
  int k = 1;

  public String toString() {
    return String.valueOf(k);
  }
}
```

```
public class Sotto extends Sopra {
  int k = 2;

  public String toString() {
    return k + ", " + super.toString();
  }
}
```

- Sotto: la variabile k adombra la variabile k di Sopra
- La variabile utilizzata da un'istruzione dipende dal contesto

`super` per accedere ai campi della superclasse

È possibile accedere al campo adombrato della superclasse mediante `super` (a patto che non sia `private`).

super per accedere ai campi della superclasse

È possibile accedere al campo adombrato della superclasse mediante super (a patto che non sia private).

```
public class Sopra {
  int k = 1;

  public String toString() {
    return String.valueOf(k);
  }
}
```

super per accedere ai campi della superclasse

È possibile accedere al campo adombrato della superclasse mediante `super` (a patto che non sia `private`).

```
public class Sopra {
  int k = 1;

  public String toString() {
    return String.valueOf(k);
  }
}
```

```
public class Sotto extends Sopra {
  int k = 2;

  public String toString() {
    return k + ", " + super.k;
  }
}
```

Variabili e adombramento

```
//CAMPI
private int ore;
private int min;

public Orario(int ore, int min) {
  this.ore = ore;
  this.min = min;
}
```

- I parametri formali adombrano i campi `ore` e `min` all'interno del costruttore

Variabili e adombramento

```
//CAMPI
private int ore;
private int min;

public Orario(int ore, int min) {
  this.ore = ore;
  this.min = min;
}
```

- I parametri formali adombrano i campi `ore` e `min` all'interno del costruttore
- `this.ore` specifica il campo dell'oggetto

Variabili e adombramento

```
//CAMPI
private int x;

public void m(double y) {
  int x = (int)y.;
  this.x = y;
}
```

- La variabile locale adombra il campo `x` all'interno del metodo

Variabili e adombramento

```
//CAMPI
private int x;

public void m(double y) {
  int x = (int)y.;
  this.x = y;
}
```

- La variabile locale adombra il campo `x` all'interno del metodo
- `this.x` specifica il campo dell'oggetto

Sommario: Estensione delle classi

Classi astratte

- Dichiarate mediante la parola chiave `abstract` prima di `class`

```
public abstract class Figura {
   ...
}
```

Classi astratte

- Dichiarate mediante la parola chiave `abstract` prima di `class`

```
public abstract class Figura {
  ...
}
```

- Non è possibile costruire oggetti di una classe astratta

Classi astratte

- Dichiarate mediante la parola chiave `abstract` prima di `class`

```
public abstract class Figura {
  ...
}
```

- Non è possibile costruire oggetti di una classe astratta
- Una classe astratta può avere tutti i membri di una classe concreta

Classi astratte

- Dichiarate mediante la parola chiave `abstract` prima di `class`

```
public abstract class Figura {
  ...
}
```

- Non è possibile costruire oggetti di una classe astratta
- Una classe astratta può avere tutti i membri di una classe concreta

I costruttori della classe astratta vengono utilizzati nella costruzione di un oggetto di una sottoclasse esattamente come per le classi concrete

Classi astratte

- Dichiarate mediante la parola chiave `abstract` prima di `class`

```
public abstract class Figura {
  ...
}
```

- Non è possibile costruire oggetti di una classe astratta
- Una classe astratta può avere tutti i membri di una classe concreta

  I costruttori della classe astratta vengono utilizzati nella costruzione di un oggetto di una sottoclasse esattamente come per le classi concrete

- Se una classe ha un metodo `abstract` allora deve essere dichiarata astratta

Classi astratte

- Dichiarate mediante la parola chiave `abstract` prima di `class`

```
public abstract class Figura {
  ...
}
```

- Non è possibile costruire oggetti di una classe astratta
- Una classe astratta può avere tutti i membri di una classe concreta

  I costruttori della classe astratta vengono utilizzati nella costruzione di un oggetto di una sottoclasse esattamente come per le classi concrete

- Se una classe ha un metodo `abstract` allora deve essere dichiarata astratta
- È possibile definire astratta una classe anche se non ha metodi astratti

La classe Figura

```
public abstract class Figura {

  public abstract double getArea();

  public abstract double getPerimetro();
```

La classe Figura

```
public abstract class Figura {

  public abstract double getArea();

  public abstract double getPerimetro();

  public boolean haAreaMaggiore(Figura altra) {
    return this.getArea() > altra.getArea();
  }

  public boolean haPerimetroMaggiore(Figura altra) {
    return this.getPerimetro() > altra.getPerimetro();
  }
}
```

La classe Figura

```
public abstract class Figura {

  public abstract double getArea();

  public abstract double getPerimetro();

  public boolean haAreaMaggiore(Figura altra) {
    return this.getArea() > altra.getArea();
  }

  public boolean haPerimetroMaggiore(Figura altra) {
    return this.getPerimetro() > altra.getPerimetro();
  }
}
```

- `Figura` estende implicitamente `Object`

La classe Figura

```
public abstract class Figura {

  public abstract double getArea();

  public abstract double getPerimetro();

  public boolean haAreaMaggiore(Figura altra) {
    return this.getArea() > altra.getArea();
  }

  public boolean haPerimetroMaggiore(Figura altra) {
    return this.getPerimetro() > altra.getPerimetro();
  }
}
```

- `Figura` estende implicitamente `Object`
- Ha un costruttore di default privo di argomenti

Sommario: Estensione delle classi

Rettangolo: campi e costruttore

```
public class Rettangolo extends Figura {
  //CAMPI
  private double base, altezza;
```

Rettangolo: campi e costruttore

```
public class Rettangolo extends Figura {
  //CAMPI
  private double base, altezza;


  //COSTRUTTORI
  public Rettangolo (double x, double y) {
    base = x;
    altezza = y;
  }


  //METODI
  ...
}
```

Implementazione dei metodi astratti di `Figura`

```
public double getArea() {
  return base * altezza;
}
```

Implementazione dei metodi astratti di `Figura`

```
public double getArea() {
  return base * altezza;
}

public double getPerimetro() {
  return 2 * (base + altezza);
}
```

Rettangolo: metodi di accesso e toString

```
public double getAltezza() {
  return altezza;
}
```

Rettangolo: metodi di accesso e toString

```
public double getAltezza() {
  return altezza;
}

public double getBase() {
  return base;
}
```

Rettangolo: metodi di accesso e toString

```
public double getAltezza() {
  return altezza;
}

public double getBase() {
  return base;
}

public String toString() {
  return "base = " + base + ", altezza = " + altezza;
}
```

Rettangolo: metodi equals

```
public boolean equals(Rettangolo altro) {
  return this.base == altro.base &&
         this.altezza == altro.altezza;
}
```

Rettangolo: metodi equals

```
public boolean equals(Rettangolo altro) {
  return this.base == altro.base &&
         this.altezza == altro.altezza;
}

public boolean equals(Object o) {
  if (o instanceof Rettangolo) {
    Rettangolo a = (Rettangolo) o;
    return equals(a);
  } else
    return false;
}
```

I metodi `equals` di `Quadrato`

- `Quadrato` eredita i metodi da `Rettangolo`

I metodi equals di Quadrato

- Quadrato eredita i metodi da Rettangolo

```
Rettangolo r = new Rettangolo(4, 4);
Quadrato q = new Quadrato(4);
if (q.equals(r))
  out.println("uguali");
else
  out.println("diversi");
```

I metodi equals di Quadrato

- Quadrato eredita i metodi da Rettangolo

```
Rettangolo r = new Rettangolo(4, 4);
Quadrato q = new Quadrato(4);
if (q.equals(r))
  out.println("uguali");
else
  out.println("diversi");
```

- Dal punto di vista della gerarchia gli oggetti riferiti da r e da q sono diversi, dal punto di vista geometrico rappresentano la stessa figura

I metodi equals di Quadrato

- Quadrato eredita i metodi da Rettangolo

```
Rettangolo r = new Rettangolo(4, 4);
Quadrato q = new Quadrato(4);
if (q.equals(r))
  out.println("uguali");
else
  out.println("diversi");
```

- Dal punto di vista della gerarchia gli oggetti riferiti da r e da q sono diversi, dal punto di vista geometrico rappresentano la stessa figura
- Privilegiamo il punto di vista geometrico

Rettangolo: cambiaBase

```
public void cambiaBase(double x) {
  base = x;
  return;
}
```

Rettangolo: cambiaBase

```
public void cambiaBase(double x) {
  base = x;
  return;
}
```

L'istruzione `return` può essere omessa

Rettangolo: cambiaBase

```
public void cambiaBase(double x) {
  base = x;
  return;
}
```

L'istruzione `return` può essere omessa

```
public void cambiaBase(double x) {
  base = x;
}

public void cambiaAltezza(double x) {
  altezza = x;
}
```

Quadrato: metodo cambiaLato

```
public void cambiaLato(double x) {
  cambiaBase(x);
  cambiaAltezza(x);
}
```

Quadrato: metodo cambiaLato

```
public void cambiaLato(double x) {
  cambiaBase(x);
  cambiaAltezza(x);
}
```

- Quadrato eredita da Rettangolo anche i metodi cambiaBase e cambiaAltezza

Quadrato: metodo cambiaLato

```
public void cambiaLato(double x) {
  cambiaBase(x);
  cambiaAltezza(x);
}
```

- Quadrato eredita da Rettangolo anche i metodi cambiaBase e cambiaAltezza
- L'uso di questi metodi consente di costruire oggetti inconsistenti

```
Quadrato q = new Quadrato(4);
q.cambiaBase(3);
double area = q.getArea();  //area = 12
```

Quadrato: metodo cambiaLato

```
public void cambiaLato(double x) {
  cambiaBase(x);
  cambiaAltezza(x);
}
```

- Quadrato eredita da Rettangolo anche i metodi cambiaBase e cambiaAltezza
- L'uso di questi metodi consente di costruire oggetti inconsistenti

```
Quadrato q = new Quadrato(4);
q.cambiaBase(3);
double area = q.getArea();  //area = 12
```

- Li ridefiniamo in modo che questo non avvenga

Quadrato: getBase, getAltezza

```
public void cambiaBase(double x) {
  cambiaLato(x);
}
```

Quadrato: getBase, getAltezza

```
public void cambiaBase(double x) {
  cambiaLato(x);
}

public void cambiaAltezza(double x) {
  cambiaLato(x);
}
```

Quadrato: getBase, getAltezza

```
public void cambiaBase(double x) {
  cambiaLato(x);
}

public void cambiaAltezza(double x) {
  cambiaLato(x);
}

public void cambiaLato(double x) {
  cambiaBase(x);
  cambiaAltezza(x);
}
```

Quadrato: getBase, getAltezza

```
public void cambiaBase(double x) {
  cambiaLato(x);
}

public void cambiaAltezza(double x) {
  cambiaLato(x);
}

public void cambiaLato(double x) {
  cambiaBase(x);
  cambiaAltezza(x);
}
```

Problema

I metodi si richiamano tra loro all'infinito.

Quadrato: getBase, getAltezza

```
public void cambiaBase(double x) {
  cambiaLato(x);
}

public void cambiaAltezza(double x) {
  cambiaLato(x);
}

public void cambiaLato(double x) {
  super.cambiaBase(x);
  super.cambiaAltezza(x);
}
```

protected

Il suo significato è legato all'ereditarietà

- All'interno di un package il modificatore `protected` è equivalente ad *amichevole*

protected

Il suo significato è legato all'ereditarietà

- All'interno di un package il modificatore `protected` è equivalente ad *amichevole*
- All'esterno del package i membri `protected` di una classe `A` sono visibili solo:
  - nel corpo delle classi `B` che la estendono

protected

Il suo significato è legato all'ereditarietà

- All'interno di un package il modificatore `protected` è equivalente ad *amichevole*
- All'esterno del package i membri `protected` di una classe `A` sono visibili solo:
  - nel corpo delle classi `B` che la estendono
  - da un riferimento che sia al più del tipo di `B`

Sommario: Estensione delle classi

final

- Il modificatore `final` può essere applicato alle variabili, ai metodi e alle classi

final

- Il modificatore `final` può essere applicato alle variabili, ai metodi e alle classi
- Ha significati differenti a seconda del contesto in cui compare

final

- Il modificatore `final` può essere applicato alle variabili, ai metodi e alle classi
- Ha significati differenti a seconda del contesto in cui compare
- In generale stabilisce che ciò che è associato all'identificatore cui è applicato non può essere modificato

`final` applicato a variabili

- `final` può essere applicato alle variabili:
  - campi
  - campi statici
  - variabili locali
  - parametri formali

`final` applicato a variabili

- `final` può essere applicato alle variabili:
  - campi
  - campi statici
  - variabili locali
  - parametri formali
- Una variabile `final` è una variabile alla quale è possibile assegnare un valore una sola volta

`final` applicato a variabili

- `final` può essere applicato alle variabili:
  - campi
  - campi statici
  - variabili locali
  - parametri formali
- Una variabile `final` è una variabile alla quale è possibile assegnare un valore una sola volta
- Nel codice che la utilizza il valore della variabile è quindi costante

`final` applicato a variabili

- `final` può essere applicato alle variabili:
  - campi
  - campi statici
  - variabili locali
  - parametri formali
- Una variabile `final` è una variabile alla quale è possibile assegnare un valore una sola volta
- Nel codice che la utilizza il valore della variabile è quindi costante
- Un tentativo di modificare la variabile dopo che le è stato assegnato un valore dà luogo a un errore in fase di compilazione

`final` applicato a variabili

- `final` può essere applicato alle variabili:
  - campi
  - campi statici
  - variabili locali
  - parametri formali
- Una variabile `final` è una variabile alla quale è possibile assegnare un valore una sola volta
- Nel codice che la utilizza il valore della variabile è quindi costante
- Un tentativo di modificare la variabile dopo che le è stato assegnato un valore dà luogo a un errore in fase di compilazione
- Convenzione: per le variabili `final` si utilizzano nomi in maiuscolo, in modo da evidenziarne la peculiarità

`final`: variabili locali

- Si può assegnare loro il valore in fase di dichiarazione o in un momento successivo.

```
final int X;
final int Y = 2;
X = Y + 1;
```

`final`: variabili locali

- Si può assegnare loro il valore in fase di dichiarazione o in un momento successivo.

```
final int X;
final int Y = 2;
X = Y + 1;
```

- Nel caso delle variabili riferimento ciò che risulta immodificabile è il riferimento non l'oggetto.

Non viene compilato

```
final String STRINGA = "pippo";
STRINGA = STRINGA.toUpperCase();
```

final: variabili locali

- Si può assegnare loro il valore in fase di dichiarazione o in un momento successivo.

```
final int X;
final int Y = 2;
X = Y + 1;
```

- Nel caso delle variabili riferimento ciò che risulta immodificabile è il riferimento non l'oggetto.

Non viene compilato

```
final String STRINGA = "pippo";
STRINGA = STRINGA.toUpperCase();
```

Compilato

```
final Rettangolo RETTANGOLO = new Rettangolo(2,3);
RETTANGOLO.cambiaBase(4);
```

final: parametri formali

```
public void f(final int X) {
...
}
```

`final`: parametri formali

```
public void f(final int X) {
...
}
```

- A `X` viene assegnato un valore al momento dell'invocazione del metodo

`final`: parametri formali

```
public void f(final int X) {
...
}
```

- A `X` viene assegnato un valore al momento dell'invocazione del metodo
- All'interno del codice del metodo `X` risulta immodificabile

`final`: campi statici

- Il valore può essere assegnato solo in fase di definizione

`final`: campi statici

- Il valore può essere assegnato solo in fase di definizione
- Ai campi statici viene assegnato un valore all'atto della creazione anche in mancanza di un'inizializzazione esplicita

final: campi statici

- Il valore può essere assegnato solo in fase di definizione
- Ai campi statici viene assegnato un valore all'atto della creazione anche in mancanza di un'inizializzazione esplicita

```
class A {
  static final int MAX_OGGETTI = 10;

  ...
}
```

`final`: campi

- È possibile assegnare il valore in fase di definizione, oppure all'interno dei costruttori

`final`: campi

- È possibile assegnare il valore in fase di definizione, oppure all'interno dei costruttori
- I campi `final` privi di una inizializzazione esplicita sono chiamati blank-final

`final`: campi

- È possibile assegnare il valore in fase di definizione, oppure all'interno dei costruttori
- I campi `final` privi di una inizializzazione esplicita sono chiamati blank-final
- I blank-final devono essere obbligatoriamente inizializzati all'interno di tutti i costruttori della classe

Esempio

```
public class A {
  final int C = 0;
  final int D;

  public A() {
    D = 1;
  }

  public A(int x) {
    D = x;
  }

  public String toString() {
    return C + ", " + D;
  }
}
```

`final`: metodi

- Stabilisce che il metodo non può essere ridefinito dalle estensioni della classe

`final`: metodi

- Stabilisce che il metodo non può essere ridefinito dalle estensioni della classe
- Meccanismo utilizzato per impedire alle sottoclassi di cambiarne il significato

`final`: metodi

- Stabilisce che il metodo non può essere ridefinito dalle estensioni della classe
- Meccanismo utilizzato per impedire alle sottoclassi di cambiarne il significato
- I metodi `final` assicurano anche una maggiore efficienza, perché la garanzia che non possono essere ridefiniti consente al compilatore di trattarli in modo diverso dagli altri metodi

`final`: classi

- Specifica che non possono essere estese

`final`: classi

- Specifica che non possono essere estese
- Utilizzato per questioni di sicurezza ed efficienza

`final`: classi

- Specifica che non possono essere estese
- Utilizzato per questioni di sicurezza ed efficienza
- Esempi di classi `final` di Java sono la classe `String` e le classi involucro