# Il late binding

Quando si estende una classe, si possono aggiungere componenti e metodi.

Possono avere gli stessi nomi di componenti e metodi già esistenti

componenti:
si aggiungono a quelle che esistono

metodi:
sovrascrivono (overriding) quelli di prima

---

## Componenti con lo stesso nome

```
class Studente {
  String nome;
  int anno;
}
```

Esempio: nella classe `Borsista` metto il numero di anni da cui va avanti la borsa di studio.

```
class Borsista extends Studente {
  int stipendio;
  double anno;
}
```

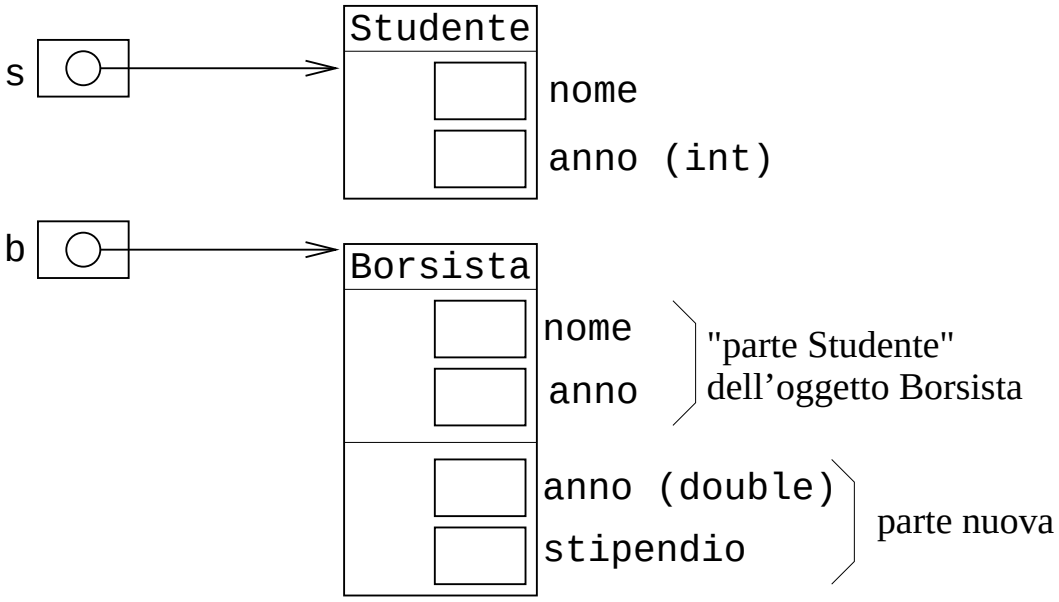
Si può fare

---

## Rappresentazione grafica

Gli oggetti `Borsista` hanno due componenti con lo stesso nome

Una componente sta nella "parte `Studente`", l'altra sta nella parte nuova



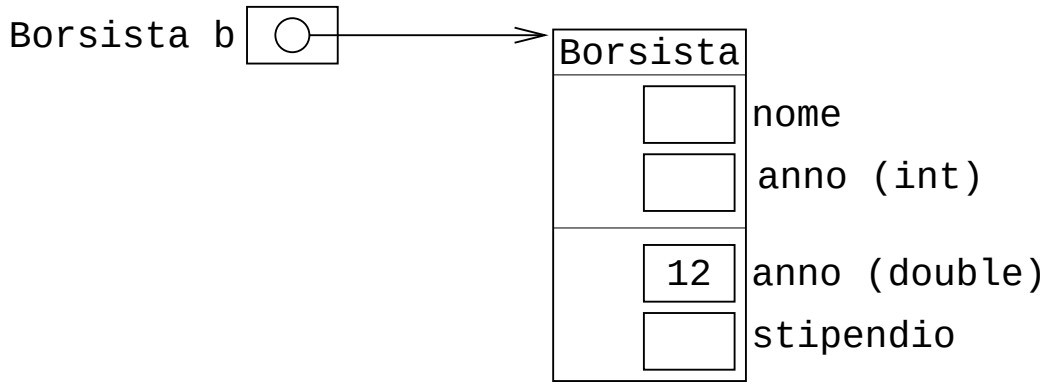

---

## Accesso alle componenti

```
public static void main(String args[]) {
  Borsista b=new Borsista();

  b.anno=12;
}
```

Il valore `12` viene messo nella componente nuova



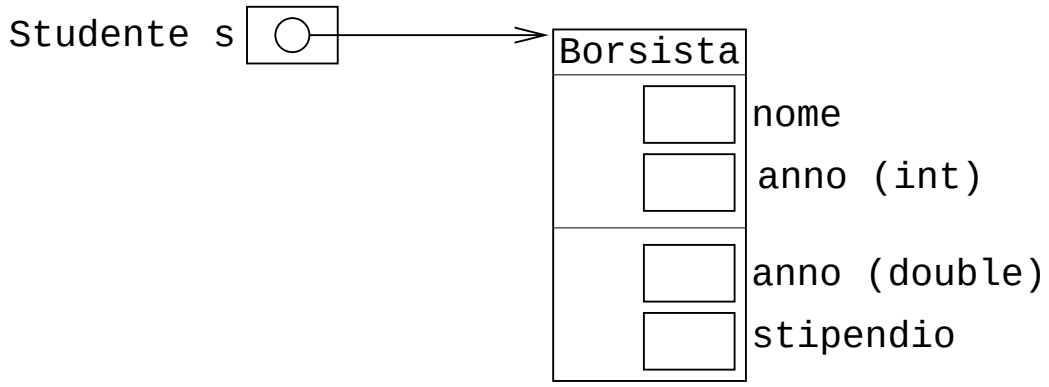

---

## La componente vecchia a che serve?

In una variabile `Studente` mi aspetto che ci sia un oggetto che abbia almeno tutte le caratteristiche di uno `Studente`

- ogni oggetto `Studente` ha una componente intera `anno`
- se `s` è di tipo `Studente`, mi aspetto che `s.anno` sia un intero

Questo deve valere anche se in `s` c'è un riferimento a un `Borsista`




---

## Variabili e oggetti: ripasso

In una variabile di una classe, ci può essere:

- `null`
- un riferimento a un oggetto della classe
- un riferimento a un oggetto di una qualsiasi sottoclasse
- nient'altro

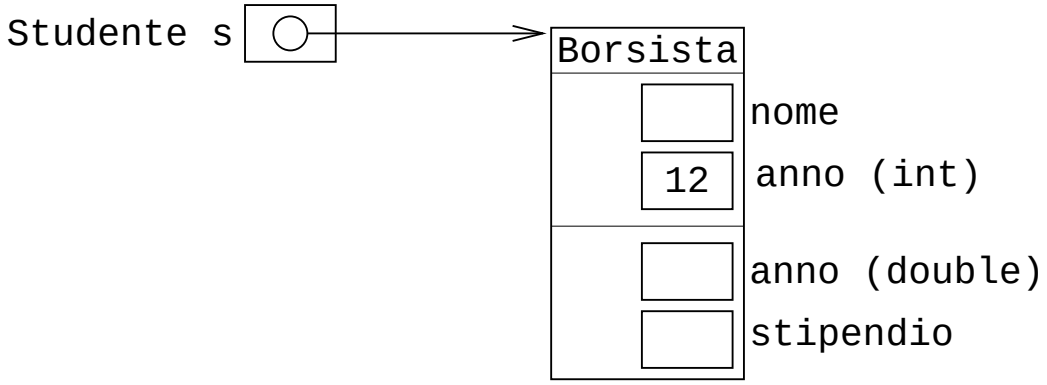
---

## Applicazione al caso di componenti aggiunte

```
public static void main(String args[]) {
  Studente s=new Borsista();

  s.anno=12;
}
```

- `s` è di tipo `Studente`
- mi aspetto che ci sia uno `Studente`
- mi aspetto che `s` abbia una componente **intera** che si chiama `anno`




---

## Regola generale

1. una sottoclasse può avere componenti in più rispetto alla sovraclasse
2. gli oggetti della sottoclasse sono comunque (concettualmente) oggetti della sovraclasse
3. gli oggetti della sottoclasse devono contenere tutte le componenti della sovraclasse, anche se sono ridefinite

---

## Nel caso specifico...?

In questi esempi, si capisce quale oggetto c'è in una variabile.

In generale, non si può sapere per certo:

```
public static void main(String args[]) {
  Borsista b=new Borsista();
  Studente s=new Studente();

  if(metodo())
    s=b;

  s.anno=12.0;
}
```

Non c’è modo di stabilire a priori se in `s` ci sarà uno `Studente` oppure un `Borsista`

So soltanto che c’è un oggetto `Studente` oppure un oggetto di una sottoclasse

---

## Nota

Concettualmente, ogni `Borsista` è uno `Studente`

In Java, sono comunque due classi diverse, e quindi due tipi diversi

Le regole sulle variabili, ecc. servono a "simulare" l’idea che ogni `Borsista` è uno `Studente` in un linguaggio in cui sono classi diverse

---

## Come si accede alle componenti?

**Regola:** la scelta della componente dipende dalla variabile

```
public static void main(String args[]) {
  Borsista b=new Borsista();
  Studente s=b;       // stesso oggetto

  s.anno=10;          // componente vecchia
  b.anno=10.2;        // componente nuova;
}
```

Come accedere alla componente intera di `Borsista`: mettere l’oggetto in una variabile `Studente`

---

## Nota

```
public static void main(String args[]) {
  Borsista b=new Borsista();
  Studente s=b;

  s.stipendio=12;     // errore
}
```

L’oggetto che sta in `s` ha una componente `stipendio`

Gli oggetti `Studente` no

Il controllo è statico: le variabili `Studente` possono non avere `stipendio`

---

## Ridefinizione di metodi

I metodi si possono ridefinire

Esempio di classe:

```
class Studente {
  String nome;
  int anno;

  void stampa() {
    System.out.println(nome+" "+anno);
  }
}
```

Tutte le sottoclassi hanno un metodo `stampa` ereditato.

---

## Ridefinizione di metodi

`Borsista` eredita il metodo `stampa` di `Studente`

Però può anche ridefinirlo:

```
class Borsista extends Studente {
  int stipendio;
  double anno;

  void stampa() {
    System.out.println(nome+" "+super.anno+" "+stipendio+" "+this.anno);
  }
}
```

Non è un errore!

In fondo, è quello che uno si aspetta: nella classe nuova ci sarà una nuova versione del metodo di stampa

`super.anno` è la componente `anno` della sovraclasse, `this.anno` quella nuova
(si può omettere il `this`)

In questo caso, `super.nome`, `this.nome` e `nome` sono la stessa cosa

---

## Metodi e componenti aggiuntive

Regola generale: se ho una variabile di una classe, posso solo accedere a metodi e componenti definiti nella classe

L'unico caso particolare è:

1. ho una componente o metodo *sia nella sottoclasse che nella sovraclasse*
   ed inoltre
2. ho una variabile della sovraclasse e un oggetto della sottoclasse

Cosa succede:

componenti:
: decide il tipo della variabile

metodi:
: decide il tipo dell'oggetto (late binding)

## Metodi di `Object`

La classe `Object` ha questi metodi (ne ha altri):

`String toString()`
: ritorna una stringa che rappresenta l'oggetto, in particolare "nomeClasse@indirizzo"

`boolean equals(Object o)`
: vede se l'oggetto passato è uguale a quello di invocazione (hanno lo stesso indirizzo);
è equivalente a `==`

`int hashCode()`
: ritorna un intero che è, se possible, diverso per oggetti diversi (ci torneremo)

## Ridefinizione dei metodi di `Object`

Tutte le classi hanno (ereditano) tutti i metodi di `Object`

Di solito, questi metodi vengono ridefiniti.

Esempio: per la classe `Point`:

`String toString()`
: ritorna una stringa con le coordinate del punto,
per esempio, `java.awt.Point[x=2,y=3]`

`boolean equals(Object o)`
: vede se l'oggetto di invocazione e il punto passato come argomento hanno le stesse coordinate

## Uso (implicito) del late binding

Perchè `System.out.println` stampa un oggetto qualsiasi?

`System.out` è un oggetto della classe `PrintStream` che ha un metodo `println`

```
class PrintStream {
  ...

  void println(Object o) {
    ...
  }
}
```

Cosa fa: stampa `o.toString()`

## La `println` di un `Point`

```
import java.awt.*;

class Punto {
  public static void main(String args[]) {
    Point p=new Point(2, 3);
```

```
    System.out.println(p);
  }
}
```

Cosa succede?

1. `println` ha un `Object` come argomento: si può passare un `Point` in quanto sottoclasse
2. l'esecuzione del metodo inizia con la copiatura del parametro attuale in quello formale (è come fare `o=p`)
3. il metodo `println(Object o)` invoca `o.toString()`;
   grazie al late binding, viene invocato il metodo di `Point`;
   la stringa risultante è quella specifica dei `Point`: `"java.awt.Point[x=2,y=3]"`
4. questa stringa viene stampata

---

## La `println` di uno `Studente`

Dato che la `println` ha un argomento `Object`, si può passare uno `Studente`:

```
  public static void main(String args[]) {
    Studente s=new Studente();

    System.out.println(s);
  }
```

Cosa succede:

1. viene invocato `println(Object o)`
2. copiatura dei parametri (equivalente a `o=s`)
3. viene invocato `o.toString()`
4. dato che `Studente` non ridefinisce `toString`, viene invocato il `toString` di `Object`
5. questo metodo ritorna il nome della classe e l'indirizzo dell'oggetto, es. `"Studente@e2fa2a"`
6. questo stringa viene stampata

Non è quello che si vorrebbe venisse stampato (i dati dello studente)

---

## Ridefinire `toString`

Basta inserire un metodo con questo nome nella classe

Cosa ci si aspetta: che ritorni una stringa che rappresenta l'oggetto

```
class Studente {
  String nome;
  int anno;

  public String toString() {
    return "["+nome+","+anno+"]";
  }
}
```

Sul `public` ci ritorneremo

---

## Attenzione!

Se `Studente` ha un metodo `stampa()` si può pensare di fare:

```
public static void main(String args[]) {
  Object o=new Studente();

  o.stampa();
}
```

La classe `Object` non ha il metodo `stampa()`

Il discorso "il metodo è quello della sottoclasse" vale solo se il metodo sta *sia* nella sottoclasse che nella sovraclasse

---

## Ridefinizione di `equals`

Le classi "serie" hanno sia `toString` che `equals` ridefiniti

`Object`:
`equals` è uguale a `==`
`Point`:
`equals` vede se i due punti hanno le stesse coordinate
`Studente`:
se non ridefinisco `equals`, è lo stesso della classe `Object`

---

## Ridefinire `equals`

Vediamo un passo per volta

1. intestazione (non è ovvia!)
2. confronto componente per componente
3. confronto delle componenti oggetto
4. caso di argumento nullo
5. caso di componenti `null`
6. confronto di classi

---

## Ridefinire il metodo

Intanto, l’intestazione:

```
class Studente {
  String nome;
  int anno;

  ...

  public boolean equals(Object o) {
    ...
  }
}
```

Nella classe `Object` ha come parametro un `Object`

Quando si eredita, il tipo degli argomenti non può cambiare

Se voglio ridefinire il metodo, devo usare gli stessi parametri che ci sono nella sovraclasse

Se definisco un metodo `equals(Studente s)`, la classe contiene sia questo che `equals(Object o)`, che è stato ereditato da `Object`

---

## Attenzione al tipo!

```
  public boolean equals(Object o) {
    return (this.nome==o.nome)&&(this.anno==o.anno);
  }
```

La variabile `o` è un `Object`

Non ha le componenti `nome` e `anno`

Prima va fatto il cast:

```
  public boolean equals(Object o) {
    Studente s;

    s=(Studente) o;

    return (this.nome==s.nome)&&(this.anno==s.anno);
  }
```

---

## Componenti oggetto

Dato che la componente `nome` è una stringa, va confrontata usando `equals`

```
  public boolean equals(Object o) {
    Studente s;

    s=(Studente) o;

    return (this.nome.equals(s.nome))&&
           (this.anno==s.anno);
  }
```

---

## Valori `null`

`this` non può valere `null` (è l'oggetto di invocazione)

`o` può valere `null`

```
if(s.equals(null))
  ...
```

Va aggiunto il controllo:

```
public boolean equals(Object o) {
  Studente s;

  if(o==null)
    return false;

  s=(Studente) o;

  return (this.nome.equals(s.nome))&&
         (this.anno==s.anno);
}
```

Dato che `this` non può valere `null`, se `o` vale `null` i due oggetti non sono uguali

---

## Componenti nulle

Anche le componenti `nome` possono valere `null`

Se `this.nome` vale `null`, allora l'invocazione `this.nome.equals(...)` genera un errore

Non dovrebbe: se `s.nome` vale `null` allora i due oggetti potrebbero essere uguali, altrimenti non lo sono (ma non si tratta di una condizione di errore)

```
public boolean equals(Object o) {
  Studente s;

  if(o==null)
    return false;

  s=(Studente) o;

  if(this.nome==null) {
    if(s.nome!=null)
      return false;
  }
  else
    if(!this.nome.equals(s.nome))
      return false;

  if(this.anno!=s.anno)
    return false;

  return true;
}
```

---

## Verifica componenti nulle

L'idea: per verificare se un gruppo di condizioni sono tutte vere, le verifico una per volta.

Se una condizione è falsa mi fermo e ritorno `false`

Altrimenti, vado avanti: se arrivo alla fine, tutte le condizioni sono vere e quindi ritorno `true`

Per le componenti scalari: se sono diverse ritorno `false`

Per gli oggetti: se `this.nome` vale `null`, allora se `s.nome` è diverso da `null` ritorno `false`

Se `this.nome` è diverso da `null`, uso `equals` come al solito

---

## Verifica classi

Dato che `equals` ha un `Object` come argomento, si può anche invocare con un parametro attuale che non è uno `Studente`

Se `o` non contiene uno `Studente` (o un oggetto di una sottoclasse) allora il cast dà un errore a runtime

Se due oggetti sono di classi diverse, allora non sono uguali

Confronto fra classi:

```
    if(this.getClass()!=o.getClass())
      return false;
```

---

## Versione finale di `equals`

```
class Studente {
  String nome;
  int anno;

  public String toString() {
    return "["+nome+","+anno+"]";
  }

  public boolean equals(Object o) {
    Studente s;

    if(o==null)
      return false;

    if(this.getClass()!=o.getClass())
      return false;

    s=(Studente) o;

    if(this.nome==null) {
      if(s.nome!=null)
        return false;
    }
    else
      if(!this.nome.equals(s.nome))
```

```
        return false;

    if(this.anno!=s.anno)
      return false;

    return true;
  }
}
```