Ereditarietà

Permette di *estendere* classi esistenti, aggiungendo metodi e componenti.

Principi e tecnica

Differenza fra concetto e implementazione:

implementazione:
: prendo una classe, e ci metto dentro nuovi metodi e componenti

concetto:
: creo una classe che rappresenta un *sottoinsieme* della classe di partenza.

Sulla seconda cosa ci torneremo.

Aggiungere componenti

Data la classe `Studente`

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

Aggiungere la componente `int stipendio` per studenti borsisti.

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

Classe e sottoclasse

Nomenclatura:

`Studente`
: classe

`Borsista`
: sottoclasse

Si dice *estendere la classe*

In effetti: *creare una nuova classe* che estende quella esistente.

La classe di partenza non cambia:

Classe	Componenti
Studente	`nome`, `anno`
Borsista	`nome`, `anno`, `stipendio`

Aggiungere metodi

Stesso sistema:

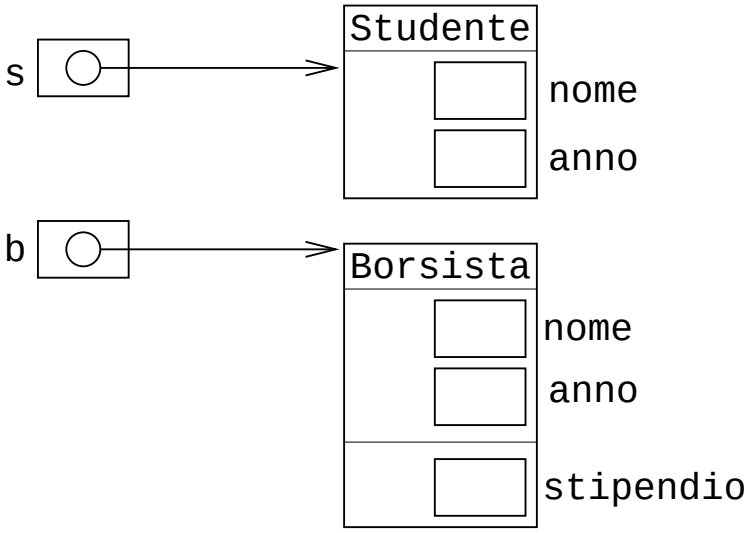
```
class Borsista extends Studente {
  int stipendio;

  int getStipendio() {
    return this.stipendio;
  }
}
```

La classe di partenza `Studente` rimane la stessa (non ha il metodo `getStipendio`)

La sottoclasse `Borsista` ha tutti i metodi di `Studente` più quelli nuovi.

Rappresentazione grafica degli oggetti

Per gli oggetti delle sottoclassi, usiamo una rappresentazione che evidenzia le parti aggiunte.



Poi vedremo il perchè

Perchè il nome "sottoclasse"?

Esempio specifico: i borsisti sono una particolare categoria di studenti.

In generale: gli oggetti di una sottoclasse possono avere proprietà che non tutti gli oggetti della classe di partenza hanno.

Usare l’ereditarietà in questi casi.

Errore metodologico

```
class Point3D extends Point {
  int z;
}
```

I punti nello spazio non sono un sottoinsieme dei punti del piano!

Uso metodologicamente sbagliato dell'ereditarietà

metodologicamente:
usare `extends` per caratterizzare sottoinsiemi;

tecnicamente:
si possono aggiungere componenti e metodi a una classe, anche se il risultato non è la rappresentazione di un sottoinsieme.

Giustificazione della regole

Classe B rappresenta un sottoinsieme di A

Tutti gli oggetti di B sono oggetti di A

Gli oggetti di B hanno (almeno) le stesse componenti e gli stessi metodi di quelli di A

Le regole sull'ereditarietà derivano dal concetto di sottoinsieme.

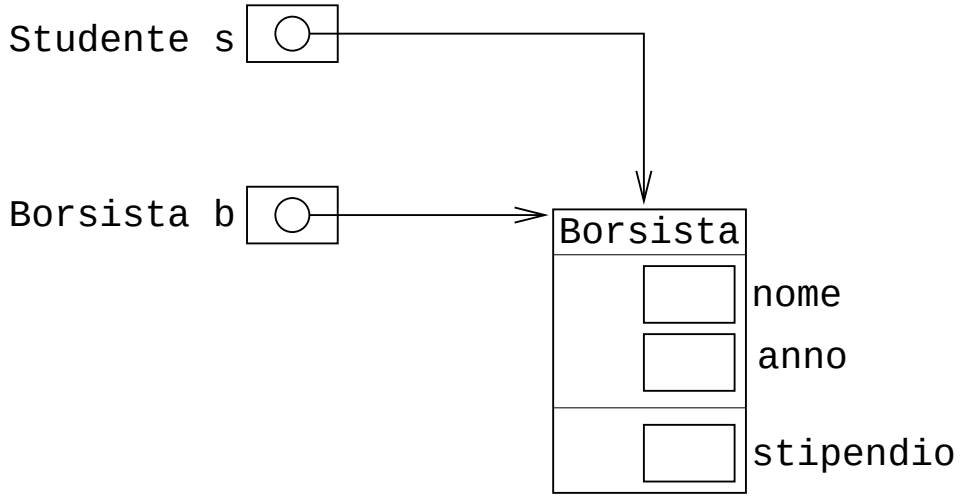
Variabili e oggetti

Tutti i Borsisti sono Studenti.

Quindi, uno `Studente` può *anche* essere un `Borsista`.

In una variabile `Studente` posso mettere il riferimento a un oggetto `Borsista`:

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

    s=b;
  }
```



Tecnicamente: da una variabile `Studente` mi aspetto un oggetto che abbia almeno le componenti `nome` e `anno`

Variabili e oggetti

Non tutti gli studenti sono borsisti:

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

    b=s; // errore: incompatible type
  }
```

Estensione e aggiunta di componenti

Mettere un riferimento a un `Point3D` in una variabile `Point`: si può fare, ma non ha senso.

La classe `Point3D` va definita da zero, anche se tecnicamente si potrebbe realizzare come sottoclasse di `Point`:

```
class Point3D {
  int x;
  int y;
  int z;
}
```

Variabili e oggetti

Alcuni studenti sono borsisti.

Si può trasferire il contenuto di una variabile `Studente` in una variabile `Borsista`, ma solo se la variabile `Studente` contiene effettivamente un `Borsista`.

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

  s=b;

  c=(Borsista) s;
}
```

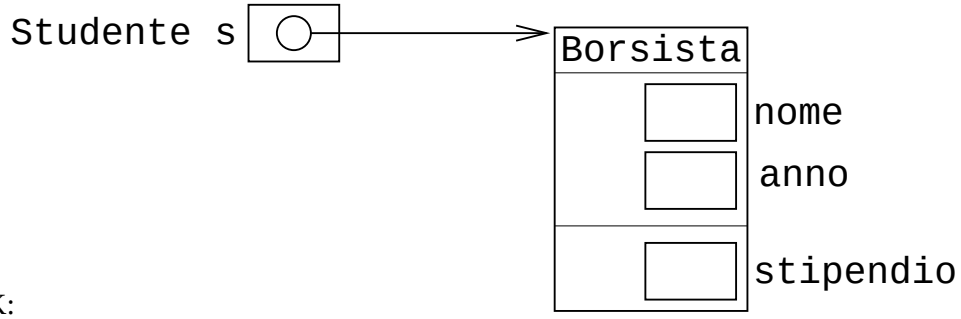
Si può sempre fare?

Funziona sempre?

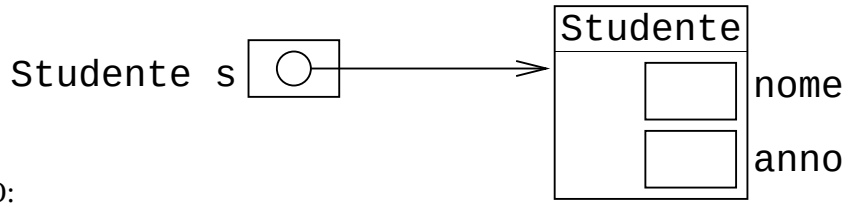
Sintatticamente, `borsista1=(Borsista) studente1;` si può sempre fare.

A run time:

OK:



NO:



Esempio:

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

  c=(Borsista) s;
}
```

Il compilatore non dà errori.

Quando si esegue:

```
Exception in thread "main" java.lang.ClassCastException
        at Runtime.main(Runtime.java:6)
```

Perchè il compilatore non se ne accorge?

```
Studente s;

if(metodo())
  s=new Studente();
else
  s=new Borsista();

Borsista b=(Borsista) s;
```

Il contenuto di `s` dipende dal risultato del metodo...

...che può dipendere dall'input dell'utente, dal contenuto di un file, ecc.

Variabili e oggetti, in generale

Si può usare una espressione che ritorna un oggetto della sottoclasse in ogni punto del programma dove va messo un oggetto della sovraclasse.

Il contrario si può fare con un cast. Può dare errore a runtime.

Esempio: a un metodo con argomento `Studente` si può passare `new Borsista()`

Gerarchia di classi

Si può estendere una classe qualsiasi.

Anche una classe che è già il risultato di una estensione:

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

class BorsistaLaureando
  extends Borsista {
  String data_prevista_laurea;
}
```

Tutti i metodi e le componenti di `Studente` sono in `Borsista`, e quindi anche in `BorsistaLaureando`

Classi estese in più modi

Una stessa classe si può estendre in più modi:

```
class StudenteLavoratore extends Studente {
  String azienza;
}
```

Le classi `Borsista` e `StudenteLavoratore` sono entrambe sottoclassi di `Studente`

Entrambe le classi hanno tutte le componenti e i metodi di `Studente`

Le componenti aggiuntive di `Borsista` e `StudenteLavoratore` possono essere diverse

Ereditarietà multipla

Uno studente potrebbe essere sia borsista che studente lavoratore.

```
// errore
class BorsistaLavoratore
  extends Borsista, StudenteLavoratore {
  ...
}
```

Creare una classe che estende due classi date: ereditarietà multipla.

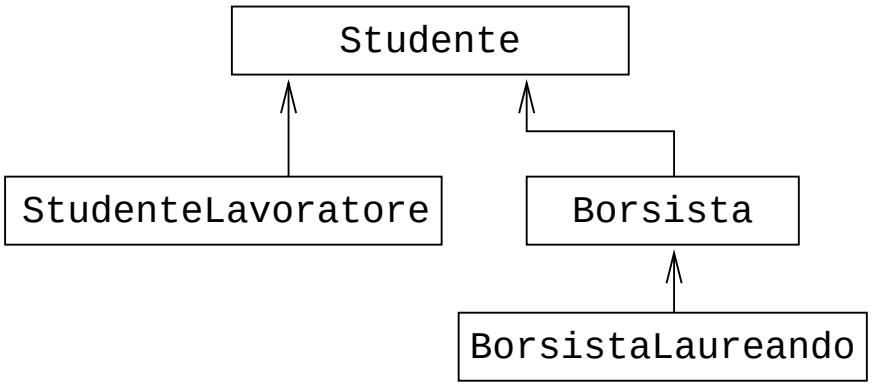
Non si può fare in Java.

Esistono linguaggio in cui si può fare

In Java, il problema è parzialmente risolto con le interfaccie.

Gerarchie di classi

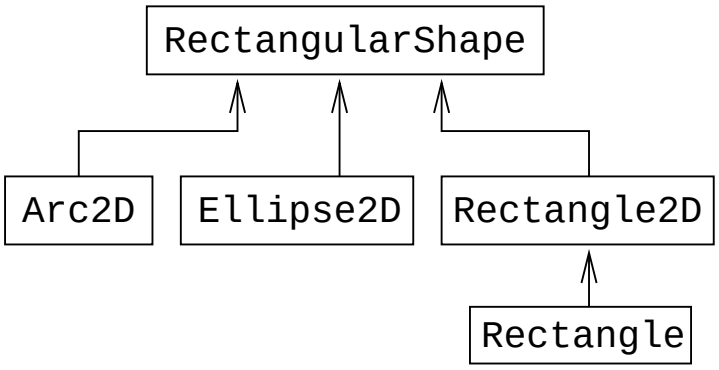
Rappresentazione grafica delle classi definite ora:



La freccia indica una relazione sottoclasse->classe

Gerarchie di classi predefinite

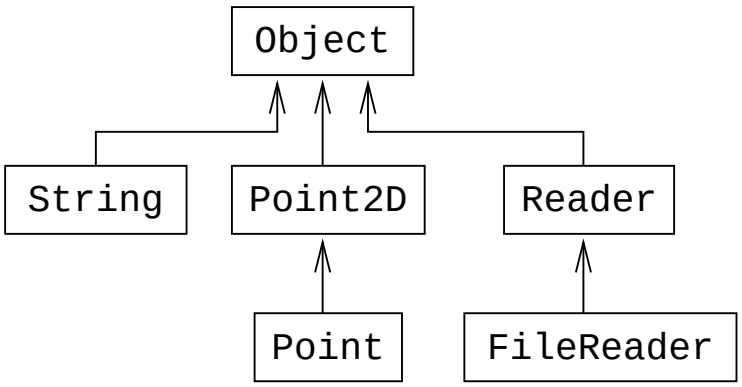
Esempio di gerarchia che esiste nelle classi predefinite di Java:



Quando possibile, si tende sempre a definire sottoclassi invece di definire classi da zero.

La classe `Object`

In cima alla gerarchia di tutte le classi c'è la classe predefinita `Object`:



È una classe predefinita del linguaggio.

Quando si crea una nuova classe senza mettere `extends SovraClasse`, è implicito `extends Object`

`class Studente {`
`...`
`}`
	=	`class Studente`
		`  extends Object {`
		`...`
		`}`

Ogni classe deriva da `Object`

Ogni classe è sottoclasse di `Object` (direttamente o indirettamente):

direttamente
: quando si omette `extends` oppure si scrive l'inutile `extends Object`

indirettamente
: si mette `extends SovraClasse`: a sua volta, `SovraClasse` è sottoclasse (direttamente o indirettamente) di `Object`

Cosa c'è nella classe `Object`?

Tutte le classi sono sottoclassi di `Object`

Tutte le classi hanno i metodi e componenti di `Object`, che sono:

componenti
: nessuna

metodi
: `equals`, `toString`, e altri

Tutte le classi hanno `equals`, `toString` ecc.

Regola mnemonica

L'indirizzo di un oggetto si può sempre mettere in una variabile di una... sovraclasse? sottoclasse?

Regola facile:

> *tutti gli oggetti si possono memorizzare in una variabile Object*

Ricordare poi che `Object` è sovraclasse di tutte le altre.

Attenzione!

Il controllo di esistenza di componenti e metodi viene fatto in fase di compilazione.

Si assume quindi che in una variabile `Classe x` ci sia un oggetto di tipo `Classe`:

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

  o=p;

  System.out.println(o.x); // errore!
}
```

L'errore è che `Object` non ha la componente `x`.

In fase di compilazione, si sa solo che `o` è un `Object`

In alcuni casi, cosa c'è nella variabile si può sapere con certezza solo quando si esegue il programma:

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

  if(metodo())
    o=new Object();
  else
    o=new Point();

  System.out.println(o.x);
}
```

Inciso

```
  if(metodo())
    o=new Object();
  else
    o=new Point();
```

Il valore di ritorno di `metodo()` può dipendere dai dati di input, da valori su file, ecc.

Se `metodo()` ritorna sempre `false`, allora `o.x` esiste sempre.

Decidere se un metodo ritorna sempre `false` è *indecidibile*

Non esiste nessuna procedura algoritmica per fare questo controllo.

Indecidibilità

Variabili non inizializzate:

```
public static void main(String args[]) {
  int x=10;
  int y;
  int z;

  y=x;

  if((x==10)||(y!=10))
    z=20;

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

Errore: `z` *potrebbe* non essere stata inizializzata.

In questo caso, lo è sicuramente!

In generale, non c'è modo per sapere se una certa istruzione verrà sempre eseguita.

Il compilatore non ci prova nemmeno

---

if/else

Il compilatore usa solo alcune regole semplici, es. si esegue sempre o l'`if` oppure l'`else`:

```
  public static void main(String args[]) {
    int x=10;
    int y;
    int z;

    y=x;

    if(x==10)
      z=20;
    else
      z=40;

    System.out.println(z); // ok
  }
```

Parametri e valori di ritorno

Se un metodo ha un parametro o un valore di ritorno, lo mantiene anche quando il metodo viene ereditato.

Lo stesso vale se il parametro o il valore di ritorno sono del tipo della classe:

```
class Studente {
  Studente cambioCanale(Studente s) {
    ...
  }
}

class Borsista
  extends Studente {
}
```

Nella classe `Borsista` il metodo `cambioCanale` ha sempre come valore di ritorno uno `Studente` (non un `Borsista`!)

classe	metodo
`Studente`	`Studente cambioCanale(Studente s)`
`Borsista`	`Studente cambioCanale(Studente s)`

Il metodo `cambioCanale` della classe `Borsista` ha come parametro e valore di ritorno uno `Studente`, non un `Borsista`

Costruttori

I costruttori non si ereditano.

Si può vedere cosí:

- il costruttore di `Studente` ritorna uno `Studente`
- quando si eredita in `Borsista`, dovrebbe avere lo stesso tipo del valore di ritorno (`Studente`)
- invece, un costruttore di `Borsista` dovrebbe avere valore di ritorno `Borsista`

I costruttori delle sovraclassi si possono però riusare (poi vediamo come)