Ereditarietà in C++ e in Java

1. Meccanismo base in C++
2. Esempi d'uso
3. Cosa si ereditata e cosa no
4. Meccanismo in Java
5. Il riferimento "super"

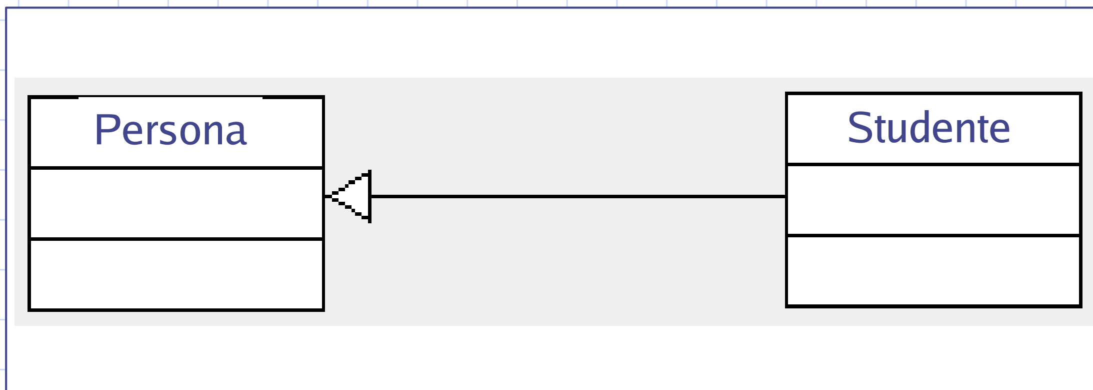
Ereditarietà: *definizione*

- Cosa fare nel caso sia necessario progettare una classe **B** e, allo stesso tempo, fosse disponibile una classe **A** quasi identica a **B** tranne per alcuni attributi e operazioni aggiuntivi?
- *Soluzione banale*: duplicare attributi e operazioni di **A** per collocarli in **B.** Difetti: (1) lavoro aggiuntivo di duplicazione, (2) lavoro di manutenzione
- *Soluzione migliore*: far sì che la classe **B** "chieda di utilizzare le operazioni" della classe **A.** Questa soluzione prende il nome di ereditarietà.

- L'**ereditarietà** (di una classe **B** da una classe **A**) è il meccanismo tramite il quale **B** ha implicitamente definito su di essa ciascuno degli attributi e delle operazioni della classe **A** come se tali attributi e operazioni fossero stati definiti per **A** stessa.
- **A** è **superclasse** di **B**, mentre **B** è una **sottoclasse** di **A**

Esempio: *concetti*

- Una *persona* è modellata da un oggetto della classe Persona. Un oggetto Persona è caratterizzato da due valori di tipo string (il nome e il cognome)
- Uno *studente* è modellato da un oggetto della classe Studente. Un oggetto Studente è caratterizzato da due valori (nome e cognome) e da una matricola (di tipo string)
- Conseguenza:



Esempio: *implementazione in C++*

- Come implementare il meccanismo dell'ereditarietà in C++?
- Analizziamo il codice delle classi Studente e Persona come se dovessero essere progettate indipendentemente una dall'altra
- Si rimarcano le differenza tra le due classi e si cerca di capire perché (in base al codice), la classe Studente può essere derivata da Persona
- Si progetta Studente derivandola da Persona

Class “Persona”

```
class Persona{
public:
   Persona(string = "", string = "");      // costruttore
   void setDati( string, string);          // set dei dati
   string getNome() const { return nome; }                // get nome
   string getCognome() const { return cognome; }  //e cognome
private:
   string nome, cognome;       // dati della persona
};
```


Class “Studente”

```
class Studente {
public:
   Studente(string, string, string ); // costruttore
    void setDati(string, string);      // set dati
    string getNome() const { return nome; } // get nome
    string getCognome() const { return cognome; } //e cognome
    void setMatricola( string );     // set matricola
    string getMatricola() const;     // get matricola
    void print() const;            // stampa i dati
private:
   string nome, cognome;          //dati studente
   string matricola;
};
```

Confronto tra “Persona” e “Studente”

```
class Studente {
public:
   Studente(string, string, string ); // costruttore
   void setDati(string, string );     // set nome e cognome
   string getNome() const { return nome; } // get nome
   string getCognome() const { return cognome; } // get cognome
   void setMatricola( string );    // set matricola
   string getMatricola() const;    // get matricola
   void print() const;             // stampa i dati
private:
   string nome, cognome;           //dati studente
   string matricola;
};
```

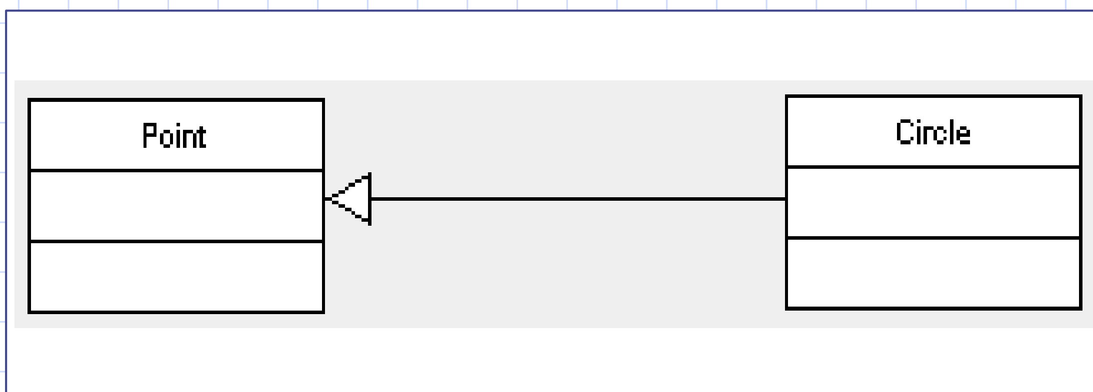
(nota: le parti in verde appartengono anche a Persona.
Della classe Persona manca il costruttore.)

Class “Studente” (derivata da Persona)

```
class Studente : public Persona {
public:
   Studente(string, string, string ); // costruttore
    void setMatricola( string );    // set matricola
    string getMatricola() const;    // get matricola
    void print() const;             // stampa i dati
private:
   string matricola;
};
```

Secondo esempio

- Un *punto* del piano è modellato da un oggetto della classe Point. Un oggetto point è caratterizzato da due valori interi (le coordinate)
- Un *cerchio* del piano è modellato da un oggetto della classe Circle. Un oggetto circle è caratterizzato da due valori interi (le coordinate del centro) e da un reale (il raggio)
- Conseguenza:



Class “Point”

```
class Point {
public:
   Point( int = 0, int = 0 );      // costruttore
   void setPoint( int, int );      // set coordinate
   int getX() const { return x; }  // get coordinata x
   int getY() const { return y; }  // get coordinata y
private:
   int x, y;      // coordinate x e y del punto
};
```

Class “Circle”

```
class Circle {
public:
   Circle(double = 0.0, int = 0, int = 0 ); // costruttore
    void setPoint( int, int );      // set coordinate
    int getX() const { return x; } // get coordinata x
    int getY() const { return y; } // get coordinata y
    void setRadius( double );     // set radius
   double getRadius() const;     // get radius
   double area() const;           // calcola area
private:
   int x, y;          // coordinate x e y del cerchio
    double radius; // raggio del cerchio
};
```


Confronto tra “Point” e “Circle”

```
class Circle {
public:
   Circle(double = 0.0, int = 0, int = 0 );   // costruttore
   void setPoint( int, int );        // set coordinate
   int getX() const { return x; } // get coordinata x
   int getY() const { return y; } // get coordinata y
   void setRadius( double );      // set radius
   double getRadius() const;      // get radius
   double area() const;           // calcola area
private:
   int x, y;          // coordinate x e y del cerchio
   double radius; // raggio del cerchio
};
```

(nota: le parti in verde appartengono anche a Point.
Della classe Point manca il costruttore.)

Class “Circle” (derivata da Point)

```
class Circle : public Point {
public:
   Circle(double = 0.0, int = 0, int = 0 );   // costruttore
    void setRadius( double );       // set radius
   double getRadius() const;      // get radius
   double area() const;            // calcola area
private:
double radius; // raggio del cerchio
};
```


Vedere esempio

- Vedere esempio del progetto “ereditarietà1”
- Consiste delle due classi Point e Circle appena descritte

Protected/Private

- I dati membro definiti protected di una classe A sono accessibili ai metodi ed alle funzioni friend di A. Sono inoltre accessibili ai metodi ed alle funzioni friend delle classi derivate da A.

- I dati membro definiti private di una classe A sono accessibili soltanto alle metodi ed alle funzioni friend di A. **Non sono** accessibili ai metodi ed alle funzioni friend delle classi derivate da A.

Cosa si eredita?

- Se A eredita da B, allora A possiede tutti gli attributi e metodi di B tranne:
  - Costruttori
  - Distruttori
  - Operatore = (assegnamento)
  - Costruttore di copia
  - Funzioni friends (ovvio, ma va rimarcato!)

Overriding

- Reimplementazione, in una classe derivata, di un metodo presente nella classe base.
- Da non confondere con l'overloading di metodi (la reimplementazione mantiene **intatto** il prototipo).
- Vedere esempio del progetto "overriding"

Progettare la classe Cylinder da Circle

```
class Cylinder {
public:
   Cylinder(double =0.0, double = 0.0, int = 0, int = 0 );
   void setPoint( int, int );   // set coord. x e y centro della base
   int getX() const;     // get coord. x centro della base
   int getY() const;     // get coord. y centro della base
   void setRadius( double );   // set raggio
   double getRadius() const;  // get raggio
   void setHeight( double );   // set altezza
   double getHeight() const;  // get altezza
   double area() const;      // calcolo area esterna totale: overriding
   double volume() const; // calcolo volume
private:
   double height, radius;  // altezza e raggio di base
   int x, y;                        // coordinate centro della base
};
```

Tipi di ereditarietà

		Tipo di ereditarietà:		
		public	**protected**	**private**
Parti della classe base:	**public**	public	protected	private
	protected	protected	protected	private
	private	private	private	private

Nota: - studiamo solo l'ereditarietà di tipo public (la più usata)

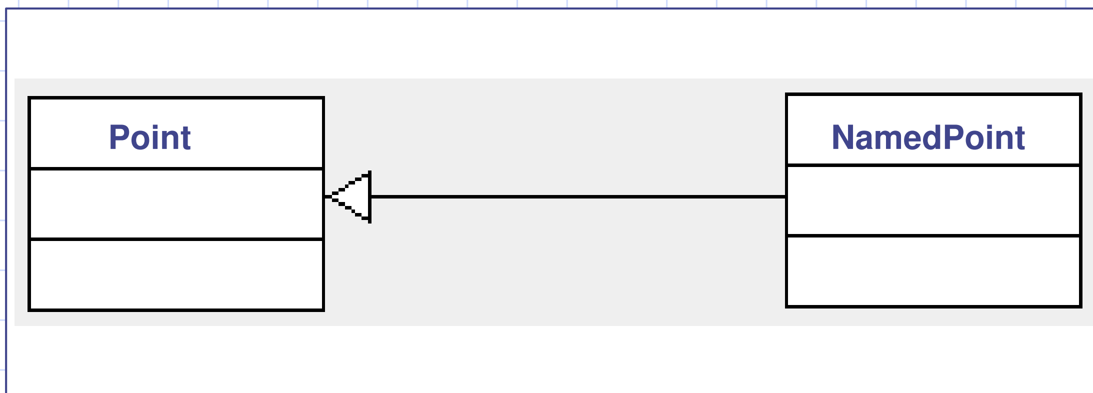
- l'ereditarietà public è la sola a rendere valido il teorema fondamentale (gli oggetti della classe derivata appartengono alla classe base)

Ereditarietà in Java

- In Java l'ereditarietà è un meccanismo fondamentale
- Esiste solo l'ereditarietà singola
- Esiste un'unica gerarchia di classi poiché ogni classe, anche se non specificato, eredita dalla classe **Object**

Un esempio

- Un *punto* del piano è modellato da un oggetto della classe java.awt.Point. Un oggetto Point è caratterizzato da due valori interi (le coordinate)
- Un *oggetto con nome* del piano è modellato da un oggetto della classe NamedPoint. Un oggetto NamedPoint è caratterizzato da due valori interi (le coordinate) e da una stringa (il nome)
- Conseguenza:



Esempio: *implementazione in Java*

```
import java.awt.Point;    //la classe Point è importata

class NamedPoint extends Point {  //derivazione
   String name;

   NamedPoint(int x, int y, String name) { //costruttore

      super(x,y); //chiamata al costruttore della superclasse
      this.name = name;

   }
}
```

this e super

- **this** e **super** sono due parole chiavi di Java
- **this** è un riferimento all'oggetto attuale
- La notazione `this(arg1, arg2, ...)`può essere usata per richiamare un costruttore dell'oggetto.
- Analogamente **super** permette, con `super.x` e `super.metodoX(),` di accede ad un attributo e un metodo della superclasse (è unica: non esiste ereditarietà multipla!)
- Con `super(arg1, arg2, ...)` si richiama il costruttore della superclasse: questa istruzione deve essere la prima di un costruttore della sottoclasse.