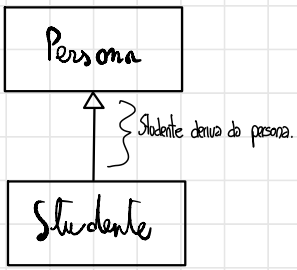
Spesso si necessita di riutilizzare il codice di classi già esistenti per creare classi simili ma con funzionalità diverse

Questo riuso di codice deve avvenire senza toccare il codice della classe base (già funzionante) ed in maniera semplice per il programmatore, che non deve ricopiare tutto il codice già scritto.

Attraverso l'ereditarietà è possibile definire una nuova classe che eredita tutti i contenuti di una classe esistente ed implementa solo attributi e metodi nuovi specifici della classe figlia.

es



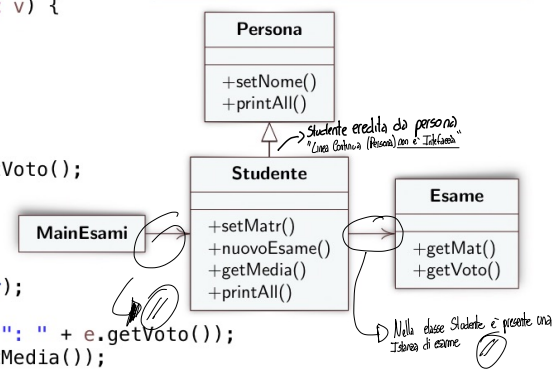

```
public class Studente extends Persona
{ ... }
```

La sottoclasse eredita tutti i metodi della superclasse e li può utilizzare come fossero definiti localmente. La sottoclasse aggiunge dei metodi che ampliano il funzionamento della classe madre ma può anche ridefinire il comportamento di alcuni metodi (non può eliminare dei metodi).

```java
public class Persona {
  private String nome, co;
  public void setNome(String n, String c) {
    nome = n; co = c;
  }
  public void printAll() {
    System.out.println(nome + " " + co);
  }
}
```

```java
public class Studente extends Persona {
  private String matr;
  private List<Esame> esami = new ArrayList<>();
  public void setMatr(String m) { matr = m; }
  public void nuovoEsame(String m, int v) {
    Esame e = new Esame(m, v);
    esami.add(e);
  }
  public float getMedia() {
    if (esami.isEmpty()) return 0;
    float sum = 0;
    for (Esame e : esami) sum += e.getVoto();
    return sum / esami.size();
  }
  public void printAll() {
    super.printAll();
    System.out.println("matr: " + matr);
    for (Esame e : esami)
      System.out.println(e.getMat() + ": " + e.getVoto());
    System.out.println("media: " + getMedia());
  }
}
```

```java
public class Esame {
  private String mat;
  private int voto;
  public Esame(String n,int v){
    mat = n;  voto = v;
  }
  public String getMat() {
    return mat;
  }
  public int getVoto() {
    return voto;
  }
}
```



La freccia che collega Studente a Persona esprime il fatto che la classe Studente eredita dalla classe Persona . Si indica con una linea continua perchè la classe base non è un'interfaccia (altrimenti tratteggiata) con sopra un triangolo vuoto.
La freccia che collega Studente ad Esame esprime invece che nella classe studente è presente un'istanza della classe esame, così come per MainEsami e Studente.

Dare il giusto livello di visibilità a metodi e attributi è essenziale quando si parla di gerarchie di classi :

- private : visibile solo all'interno della classe stessa
- public : visibile anche all'esterno della classe
- protected : visibile solo alle classi che fanno parte della gerarchia

↳ Non contiene COSTRUTTORI nella classe sono presenti metodi non "sviluppati"

Interfaccia —> keyword Interface

Un'interfaccia è una classe che contiene solo attributi inizializzati, non contiene un'implementazione per i metodi ma ne elenca solo le firme e non contiene costruttori.

Classe Astratta —> keyword Abstract

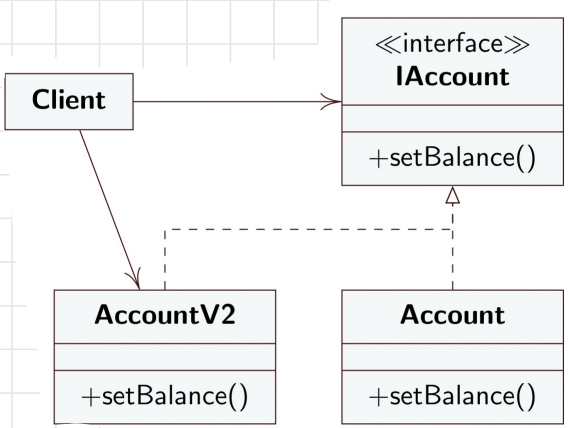
↳ Alcuni metodi verranno implementati nelle classi figlie, i metodi abstract possono essere invocati attraverso i metodi della classe stessa.

Una classe astratta è invece è una classe implementata solo in parte, infatti alcuni metodi sono definiti mentre di altri non è possibile dare un'implementazione e quindi si indicano con "abstract" per indicare che quel metodo verrà ridefinito in una classe figlia. Altri metodi della stessa classe possono invocare i metodi abstract. Il client (l'utilizzatore della classe) si aspetta di poter invocare dei metodi abstract poichè istanzierà delle sottoclassi della classe astratta che sono forzate nell'implementazione di questi metodi.

NB. Non possono essere istanziate interfacce e classi stratte ma posso dichiarare un oggetto di questo tipo da inizializzare con una qualche sottoclasse .

```
public abstract class Libro {
   private String autore;
   public abstract void insert();
   public String getAutore() {
      return autore;
   }
}
```

Un client che utilizza al suo interno l'interfaccia continua a funzionare quando viene cambiata l'implementazione di un metodo dell'interfaccia



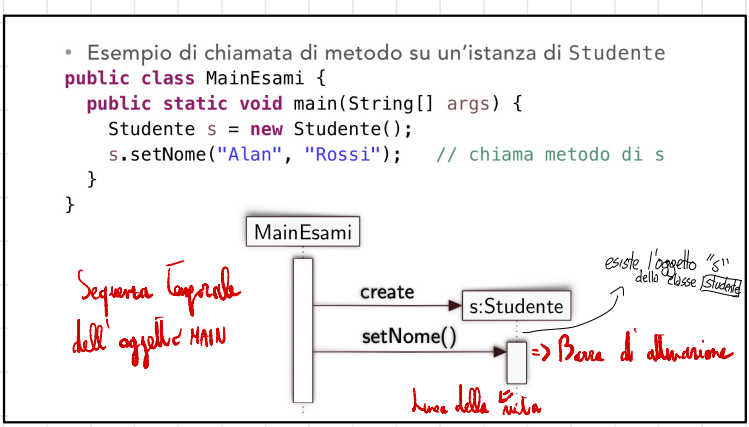
Una sottoclasse è un sottotipo compatibile con la superclasse, ovvero un oggetto di una classe figlia è anche un oggetto di classe madre. Posso sempre sostituire un istanza di una sottoclasse dove è presente un'istanza della superclasse lasciando inalterato il funzionamento del programma, non è possibile fare il viceversa. Nell'esempio di prima la classe Studente ridefinisce il metodo printAll() della classe Persona (fa override) : è possibile richiamare il metodo della superclasse Persona dalla sottoclasse Studente attraverso la seguente sintassi :
" super.printAll(); " → Richiamo il metodo della superclasse nella classe figlia
Su un oggetto di classe madre che punta ad un oggetto di classe figlia, non possono comunque essere invocati metodi specifici della classe figlia.

Diagramma UML di sequenza

Nel diagramma di sequenza, gli oggetti sono rappresentati da rettangoli verticali disposti in sequenza temporale. Gli eventi o le azioni sono rappresentati da frecce orizzontali che collegano gli oggetti. Le frecce indicano l'ordine temporale delle azioni e possono avere delle etichette che descrivono l'azione che viene eseguita. Il diagramma di sequenza è utile per comprendere le interazioni tra gli oggetti del sistema, l'ordine temporale delle azioni e i flussi di controllo. Inoltre, può essere utilizzato per identificare eventuali problemi di sincronizzazione o di coordinamento tra gli oggetti del sistema, facilitando così la comprensione del comportamento dinamico del sistema.

Un UML di sequenza mostra interazioni fra oggetti in un asse temporale crescente verso il basso. In ciascuna colonna se l'oggetto esiste, è riportata la linea della vita dell'oggetto, ovvero una linea tratteggiata . Se l'oggetto è attivo viene riportata la barra d'attivazione . Una chiamata di un metodo è indicata da una freccia piena che va dalla barra di attivazione di un oggetto ad un altro.

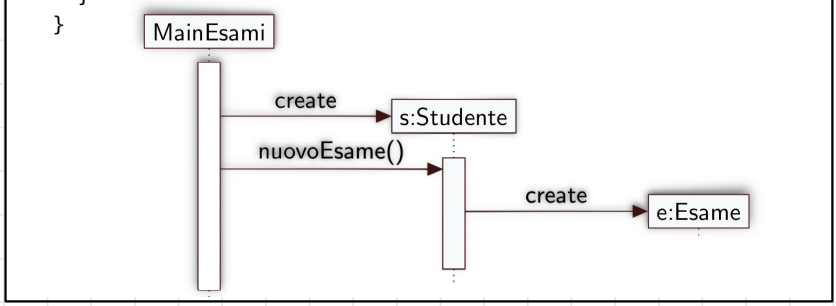
Dato che setNome() è collegato allo studente, nel main indico che viene invocato tale metodo e poi mi ricollego all'istanza s:Studente in quanto è quella l'istanza su cui viene invocato il metodo. Ovviamente setNome() , essendo invocato dopo la creazione di s, viene disegnato dopo nella barra di attivazione di s.



Potrebbe capitare che un metodo crea un'istanza di un'altra classe

```
public void nuovoEsame(String m, int v) {
  Esame e = new Esame(m, v);
  esami.add(e);
}
```

```
public class MainEsami {
  public static void main(String[] args) {
    Studente s = new Studente();
    s.nuovoEsame("Italiano", 8);
  }
}
```



Late Binding e Polimorfismo

Il polimorfismo è la proprietà in base alla quale oggetti differenti possono rispondere in maniera diversa ad uno stesso messaggio. Si manifesta quando in una gerarchia ereditaria vi sono ridefinizioni dello stesso metodo (stessa firma) che ha un comportamento diverso a seconda dell'oggetto chiamante(overriding).
Strettamente legato al concetto di "binding":
Con late binding intendiamo la connessione tra la chiamata di una funzione ed il codice che l'implementa avviene a run-time (in fase di esecuzione). Solo durante l'esecuzione del programma si determinerà il binding effettivo. Il principale vantaggio del Polimorfismo è il riutilizzo del codice, infatti permette di definire un'interfaccia comune per lavorare con oggetti di diverse classi.
Il meccanismo che permette l'associazione tra chiamata della funzione e blocco di codice che la implementa si chiama Dispatch : esso valuta la destinazione di salto.

Una variabile di un tipo può tenere un riferimento ad un'istanza di un sottotipo.
Siano Persona e Studente le classi riportate sopra, tali che Studente eredita da Persona.

```
Persona p = new Studente ();
```

Tramite cast forzo l'istanza p di tipo Persona sulla variabile s del tipo studente

```
Studente s = (Studente) p;
```

Per il compilatore va bene solo se p è di tipo Studente. Se si assegnasse un'istanza di tipo persona si avrebbe a runtime un ClassCastException

```
Studente s = (Studente) new Persona();
```

Il casting cambia il tipo a a compile-time ma non a run-time , in linea di massima è meglio evitare i casting.