Classi astratte

- Una classe astratta è parzialmente implementata
  - Alcuni metodi sono implementati
  - Altri metodi non sono implementati (sono dichiarati `abstract`)
    - E' utile avere la definizione del metodo anche non avendone l'implementazione
      - I client si aspettano di poterlo invocare
      - Forza le sottoclassi (concrete) ad implementare quel metodo
  - La classe non può essere istanziata
  - Le sottoclassi ereditano implementazioni e attributi
- Es.

```
public abstract class Libro {
   public abstract void insert();
   public String getAutore() { … }
}
```


Interfacce

- In Java una interfaccia riprende il concetto di interfaccia di sistemi orientati ad oggetti
  - Non fornisce una implementazione per i metodi
  - Permette di definire un tipo
  - Elenca le signature dei metodi public (senza corpo dei metodi)
    - Posso solo dichiarare i metodi
  - Niente attributi non inizializzati, niente costruttori

```
public interface IAccount {
  public boolean deposito(int amount);
  public void setInitial();
  public boolean check();
}
```


Classi e Interfacce

- Una classe può implementare una interfaccia
  - Ovvero, la classe fornisce una implementazione per i metodi definiti dall'interfaccia
- Non è possibile istanziare interfacce
- Tramite l'interfaccia i client sanno cosa possono invocare
- Posso usare (per istanziazioni e invocazioni di metodo) una qualsiasi delle implementazioni disponibili per l'interfaccia
- Un client che usa una interfaccia rimane immutato quando l'implementazione dell'interfaccia cambia

```
public class Account implements IAccount {  …  }
public class AccountV2 implements IAccount {  …  }

…

IAccount a = new AccountV2();
a.setBalance();
```


Compatibilità tra classi

- L'ereditarietà permette una classificazione di tipi
- Una sottoclasse è un *sottotipo* della superclasse, ovvero
  - Una sottoclasse è anche ciò che è la superclasse
- Es. il tipo `Studente` è compatibile con il tipo `Persona`
  - La classe `Studente` fa tutto ciò che fa `Persona`
  - La classe `Studente` fa altre cose oltre quelle che fa `Persona`
- Una sottoclasse può prendere il posto della superclasse
  - Es. posso usare una istanza di `Studente` quando ne usavo una di `Persona`
  - Dove compare ad es. `p.setName()` con `p` di tipo `Persona` posso sostituire `s.setName()` con `s` di tipo `Studente`
- Attenzione: non vale il contrario

```java
public class Persona {
  protected String nome, cognome;
  public void setName(String nom,
                      String cog) {
    nome = nom;
    cognome = cog;
  }
  public void printAll() {
    System.out.println("Nome: "+
                  nome+" "+cognome);
  }
}
```

```java
public class Test {
  static void main(String[] args) {
    Studente s = new Studente();
    s.setName("Jeff","Riddle");
    s.nuovoEsame("Italiano", 8);
    s.nuovoEsame("Fisica", 7);
    s.printAll();
    Persona p = s;
    p.printAll();
  }
}
```

```java
public class Studente extends Persona {
  private int numEsami = 0;
  private String[] esami = new String[10];
  private int[] voti = new int[10];
  public void nuovoEsame(String e, int v) {
    esami[numEsami] = e;
    voti[numEsami] = v;
    numEsami++;
  }
  public float media() {
    if (numEsami == 0) return 0;
    float sum = 0;
    for (int i=0; i<numEsami; i++)
      sum = sum + voti[i];
    return sum/numEsami;
  }
  public void printAll() {
    super.printAll();
    for (int i=0; i<numEsami; i++)
      System.out.println(esami[i]+" "+
                                    voti[i]);
    System.out.println("media = "+media());
  }
}
```

Considerazioni sul codice

- La classe `Studente`
  - Eredita tutto ciò che fornisce `Persona`
  - Ridefinisce il metodo `printAll` (ovvero fa *override*), quindi modifica il comportamento del metodo `printAll` ereditato
    - `super.printAll` serve per invocare `printAll` di `Persona` da `Studente`
    - `super` permette di accedere metodi della superclasse
- Per la classe `Test`
  - La variabile `p` è di tipo `Persona`, ma punta una istanza di `Studente`
  - Posso invocare su `p` il metodo `printAll`
    - Quale sarà il risultato? Viene invocato `printAll` di `Studente`
  - Non posso invocare su `p` il metodo `nuovoEsame`
    - Il tipo di `p` (ovvero `Persona`) non fornisce il metodo `nuovoEsame`, quindi il compilatore non può far invocare `nuovoEsame` su `p` (nonostante `p` punterà a runtime ad una istanza di `Studente`)

Late binding e polimorfismo

```java
public class Test {
  public static void main(String[] args) {
    Persona p = new Persona();
    Studente s = new Studente();
    Persona px;
    ...
    if ( ... ) px = p;
    else px = s;
    px.printAll();
  }
}
```

- `printAll` invocato su `px` può assumere il comportamento definito in `Persona` o quello definito in `Studente`
- Il compilatore riconosce che `printAll` è definito per `px` (qualunque sia l'istanza puntata)
- A runtime si decide quale `printAll` eseguire, ovvero si ha late binding
  - Il comportamento di `printAll` è polimorfo

Polimorfismo

- Nei sistemi ad oggetti possono esistere metodi con lo stesso nome e la stessa signature (in classi diverse)
- Quando si usa l'ereditarietà e sono stati definiti metodi con lo stesso nome, la chiamata ad un metodo può avere effetti diversi, ovvero si ha un comportamento polimorfo
  - Es. ciascun elemento di `s` può indicare una qualsiasi istanza la cui classe ha come superclasse `Shape`

```java
public class TestShape {
  public static void main(String[] args) {
    Shape[] s = new Shape[3];
    s[0] = new Box(2,3);
    s[1] = new Circle(4);
    s[2] = new TriangleRect(3,4);
    for (int i=0; i<3; i++) s[i].show();
  }
}
```