

# Riuso Di Classi

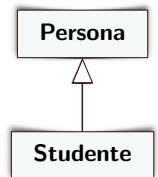
- Spesso si ha bisogno di classi simili
- Si vuole cioè **riusare** classi esistenti per implementare attributi e metodi leggermente diversi
- **Non** è pratico copiare la classe originaria e modificarne attributi o metodi, si avrebbe una proliferazione di classi e tanto lavoro per il programmatore
- Il riuso delle classi esistenti deve avvenire
  - Senza dover modificare codice esistente (e funzionante)
  - In modo semplice per il programmatore

1

Prof. Tramontana - Marzo 2020

# Ereditarietà Classi

- Attraverso l'ereditarietà è possibile
    - Definire una nuova classe indicando solo cosa ha in più rispetto ad una classe esistente: ovvero attributi e metodi nuovi, e modificando i metodi esistenti
  - Esempio: una classe Persona ha nome e cognome (più vari metodi)
  - La classe Studente dovrebbe avere tutto ciò che Persona ha (attributi e metodi) e nuovi attributi e metodi
  - Studente aggiunge esami, voti, etc.
  - La classe Studente eredita da Persona
- ```
public class Studente extends Persona { ... }
```
- Studente è sottoclasse di Persona e Persona è superclasse di Studente

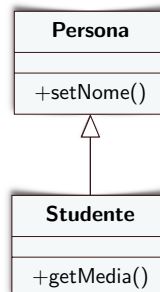


2

Prof. Tramontana - Marzo 2020

# Ereditarietà Classi

- La sottoclasse
  - Eredita tutti i metodi e gli attributi della superclasse e può usarli come se fossero definiti localmente
  - Aggiunge altri metodi
  - Può ridefinire i metodi della superclasse
  - Non può eliminare metodi o attributi della superclasse
- Esempio: la classe Studente
  - Può usare tutti i metodi della classe Persona, es. setName()
  - Può aggiungere metodi, es. getMedia()



3

Prof. Tramontana - Marzo 2020

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

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());
    }
}
```

```
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;
    }
}
```

```
classDiagram
    class Persona {
        +setName()
        +printAll()
    }
    class Studente {
        +setMatr()
        +nuovoEsame()
        +getMedia()
        +printAll()
    }
    class Esame {
        +getMat()
        +getVoto()
    }
    Studente --|> Persona
    Studente --> Esame
```

MainEsami

4

Prof. Tramontana - Marzo 2020

# Ereditarietà

- Visibilità
  - Ciò che è `private` è visibile solo alla classe, non alla sottoclasse
  - Ciò che è `public` è visibile a tutti, anche alla sottoclasse
  - Ciò che è `protected` è visibile alle sottoclassi ma non a tutte le altre classi
- Il nome della sottoclasse deve comunicare a quale classe è simile e come è diversa
- Classi che servono come radici di una gerarchia devono avere nomi concisi, altrimenti si può dare un nome più lungo

5

Prof. Tramontana - Marzo 2020

# 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 void setBalance();  
}
```

6

Prof. Tramontana - Marzo 2020

# Classi Astratte

- Una classe astratta è una classe parzialmente implementata. Alcuni metodi sono implementati, altri no (e quest'ultimi sono `abstract`)
- Un metodo `abstract` (senza implementazione) è utile poiché
  - Altri metodi della stessa classe (implementati) possono invocarlo
  - I client si aspettano di poterlo invocare
- Forza le sottoclassi (concrete) ad implementare il metodo `abstract`
- La classe astratta non può essere istanziata

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

7

Prof. Tramontana - Marzo 2020

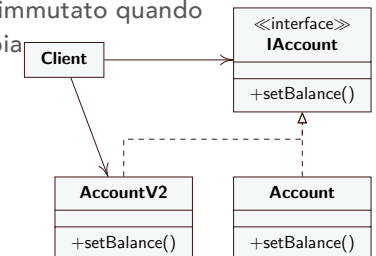
# Classi Ed Interfacce

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

```
public class AccountV2 implements IAccount {  
    public void setBalance() { }  
}  
  
public class M {  
    public void main(String[] args) {  
        IAccount a = new AccountV2();  
        a.setBalance();  
    }  
}
```

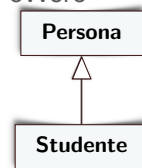
8

Prof. Tramontana - Marzo 2020



# Compatibilità Fra Tipi

- L'ereditarietà permette di definire una classificazione di tipi. Una sottoclasse è un sottotipo compatibile con la superclasse, ovvero una sottoclasse è anche ciò che è la superclasse
- Esempio, si abbia *Studiante* sottoclasse di *Persona*
  - Il tipo *Studiante* è compatibile con il tipo *Persona*
  - La classe *Studiante* fa tutto ciò che fa *Persona*, ed altre cose oltre quelle che fa *Persona*
- Una sottoclasse può prendere il posto della superclasse
- Esempio: si può usare un'istanza di *Studiante* al posto di una di *Persona*. Dove compare `p.setNome()` con `p` di tipo *Persona* posso sostituire `s.setNome()` con `s` di tipo *Studiante*
- Attenzione non vale il contrario, non si può usare la superclasse dove si usava la sottoclasse



9

Prof. Tramontana - Marzo 2020

## Considerazioni

- La classe *Studiante* eredita tutto ciò che ha *Persona*, inoltre ridefinisce il metodo `printAll()` (ovvero fa override), quindi modifica il comportamento di `printAll()` ereditato
  - `super.printAll()` permette di chiamare `printAll()` di *Persona* da *Studiante*, `super` permette di accedere ai metodi della superclasse
- Nella classe *MainEsami*, la variabile `p` è di tipo *Persona*, ma punta a un'istanza di *Studiante*
  - Chiamando su `p` il metodo `printAll()`, sarà eseguito `printAll()` di *Studiante*
  - Su `p` non si può chiamare `nuovoEsame()`, poiché il tipo di `p` a compile time (*Persona*) non ha `nuovoEsame()`
  - Quindi, il compilatore non può far chiamare `nuovoEsame()` su `p` (nonostante `p` punterà a runtime ad un'istanza di *Studiante*)

11

Prof. Tramontana - Marzo 2020

```

public class Persona {
    public void setName(String n, String c) { ... }
    public void printAll() { ... }
}
  
```

```

public class Studiante extends Persona {
    public void setMatr(String m) { ... }
    public void nuovoEsame(String m, int v) { ... }
    public float getMedia() { ... }
    public void printAll() { ... }
}
  
```

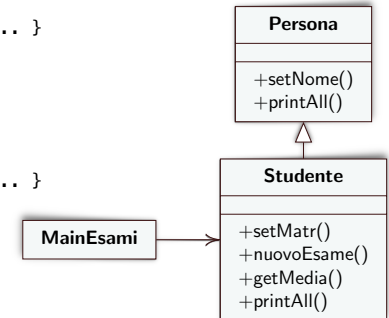
```

public class MainEsami {
    public static void main(String[] args) {
        Studiante s = new Studiante();
        s.setName("Alan", "Rossi"); // metodo della superclasse di s
        s.setMatr("M12345");
        s.nuovoEsame("Italiano", 8); // metodo della classe di s
        s.printAll(); // metodo della classe di s

        s.nuovoEsame("Fisica", 7);
        Persona p = s; // p e' dichiarato di tipo Persona
        p.printAll(); // a runtime p punta all'istanza s
    }
}
  
```

10

Prof. Tramontana - Marzo 2020

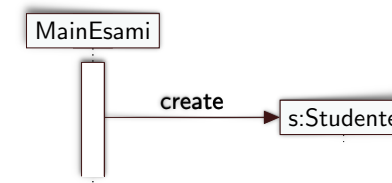


## Diagramma UML Di Sequenza

- Esempio di creazione di un'istanza della classe *Studiante*

```

public class MainEsami {
    public static void main(String[] args) {
        Studiante s = new Studiante();
    }
}
  
```



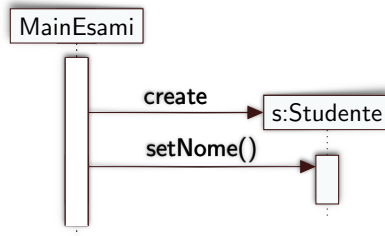
12

Prof. Tramontana - Marzo 2020

# Diagramma UML Di Sequenza

- Esempio di chiamata di metodo su un'istanza di Studente

```
public class MainEsami {
    public static void main(String[] args) {
        Studente s = new Studente();
        s.setNome("Alan", "Rossi"); // chiama metodo di s
    }
}
```



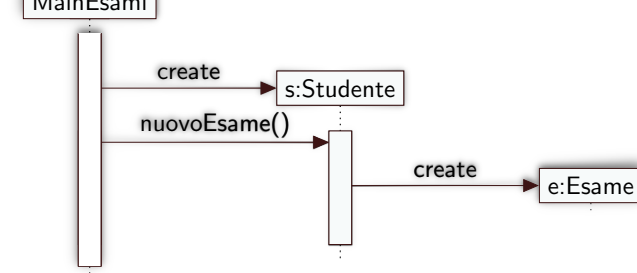
13

Prof. Tramontana - Marzo 2020

# Diagramma UML Di Sequenza

- Esempio di chiamate di metodo a cascata

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

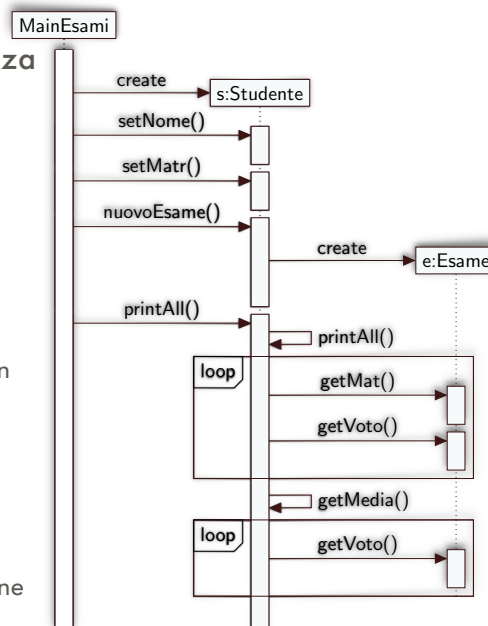


14

Prof. Tramontana - Marzo 2020

## Diagramma UML Di Sequenza

- Mostra interazioni fra oggetti
- L'asse temporale è inteso in verticale verso il basso
- In alto in orizzontale ci sono vari oggetti
- In ciascuna colonna se l'oggetto esiste è indicato con una linea tratteggiata, detta linea della vita, e se è attivo con una barra di attivazione
- Una chiamata di metodo è indicata da una freccia piena che va dalla barra di attivazione di un oggetto ad un altro

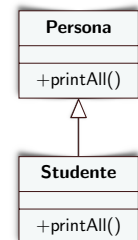


15

Prof. Tramontana - Marzo 2020

## Late Binding E Polimorfismo

```
public void m() {
    Persona p = new Persona();
    Studente s = new Studente();
    Persona px;
    if (i > 10)
        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, ed il comportamento di printAll() è polimorfo

16

Prof. Tramontana - Marzo 2020

# Polimorfismo

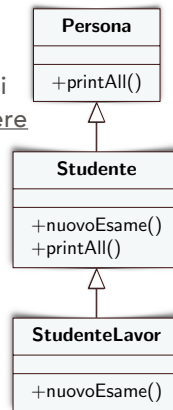
- Nei sistemi ad oggetti possono esistere metodi con lo stesso nome e la stessa signature (in classi diverse)
- Quando si usa l'ereditarietà e sono stati definiti metodi con lo stesso nome, la chiamata ad un metodo può avere effetti diversi, ovvero il comportamento è polimorfo

```

Studiante s;
// ...
s.nuovoEsame("Maths", 8);
s.printAll();

```

- La variabile s potrebbe tenere il riferimento ad un'istanza di *Studiante* o di *StudianteLavor*, quale metodo `nuovoEsame()` sarà chiamato è deciso a runtime, in base all'istanza. Lo stesso vale quando si ha una variabile di tipo *Persona* e per il metodo `printAll()`



17

Prof. Tramontana - Marzo 2020

# Polimorfismo

- Il polimorfismo è una caratteristica fondamentale dei sistemi ad oggetti
- Il late binding è tipico dei sistemi in cui esiste il polimorfismo
- Senza polimorfismo dovremmo inserire uno switch sul chiamante per valutare la classe di ciascuna istanza e chiamare il metodo corrispondente a tale classe

18

Prof. Tramontana - Marzo 2020

## Sottoclassi E Dispatch

```

public class Account {
    protected float balance;
    public void setBalance(float amount) {
        System.out.println("in account set-balance");
        if (check(amount))
            balance = amount;
    }
    public boolean check(float amount) {
        System.out.println("in account check");
        return (balance + amount) >= 0;
    }
}

```

```

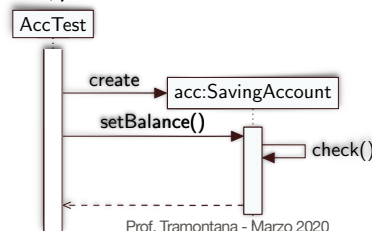
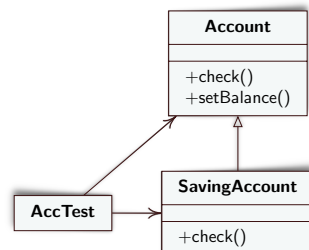
public class SavingAccount extends Account {
    public boolean check(float amount) {
        System.out.println("in saving-account check");
        return (balance + amount) >= 1000;
    }
}

```

```

public class AccTest {
    public static void main(String[] args) {
        Account acc = new SavingAccount();
        acc.setBalance(1234);
    }
}

```



19

Prof. Tramontana - Marzo 2020

## Dispatch

- Quale versione di `check()` è chiamata da `setBalance()`?
- Quando un metodo (`setBalance()`) è chiamato su un oggetto, viene controllato il suo tipo a runtime (`SavingAccount`)
- Si cerca il metodo `setBalance()` sul tipo a runtime: non trovato
- Se non trovato, si cerca la superclasse: `Account`
- Se trovato si esegue: `Account.setBalance()`
- Questo chiama `check()`, si cerca come prima
- Quindi si cerca prima su `SavingAccount`: trovato
- Si esegue `SavingAccount.check()`

20

Prof. Tramontana - Marzo 2020

# Tipi Di Variabili E Tipi A Runtime

- A compile time, si dichiara una variabile di un certo tipo, ed il tipo determina quali operazioni, quali signature, si possono usare. Tale tipo può essere una classe o un'interfaccia
- A runtime, l'oggetto riferito dalla variabile ha una sola implementazione che è sempre una classe, non un'interfaccia. Il tipo a runtime di un oggetto non cambia mai
- Una variabile di un tipo può tenere il riferimento ad un'istanza del sottotipo (la classe `Studente` è sottotipo di `Persona`)

`Persona p = new Studente();`

- Tramite cast forzo l'istanza `p` di tipo `Persona` sulla variabile `s` del sottotipo `Studente`

`Studente s = (Studente) p;`

- OK per il compilatore, OK a runtime solo se l'istanza `p` è del tipo di `s`
- Con `s = (Studente) new Persona()` si avrebbe a runtime `ClassCastException`