Linguaggi di Programmazione

a.a. 13/14

docente: Gabriele Fici

gabriele.fici@unipa.it

7 - Ereditarietà e Polimorfismo

7 - Ereditarietà e Polimorfismo

- In Java è fondamentale riutilizzare codice già scritto
- Ad esempio, spesso si vuole estendere una classe esistente aggiungendo nuove funzionalità (nuovi attributi, nuovi metodi)
- In Java si può definire una nuova classe come sottoclasse di un'altra
- Ad esempio, la classe `Studente` potrebbe essere sottoclasse della classe `Persona`, immaginando che tutti gli attributi e i metodi di `Persona` vadano bene anche per `Studente`

7 - Ereditarietà e Polimorfismo

- Quando si vuole dichiarare una classe come sottoclasse di un'altra si usa la sintassi seguente:

```
class nomeSottoclasse extends nomeSuperclasse{
    ...
}
```

- La sottoclasse (o classe derivata) eredita allora *tutti* gli attributi e *tutti* i metodi della superclasse (o classe base), che sono quindi inclusi di default nella sottoclasse

7 - Ereditarietà e Polimorfismo

Esempio:

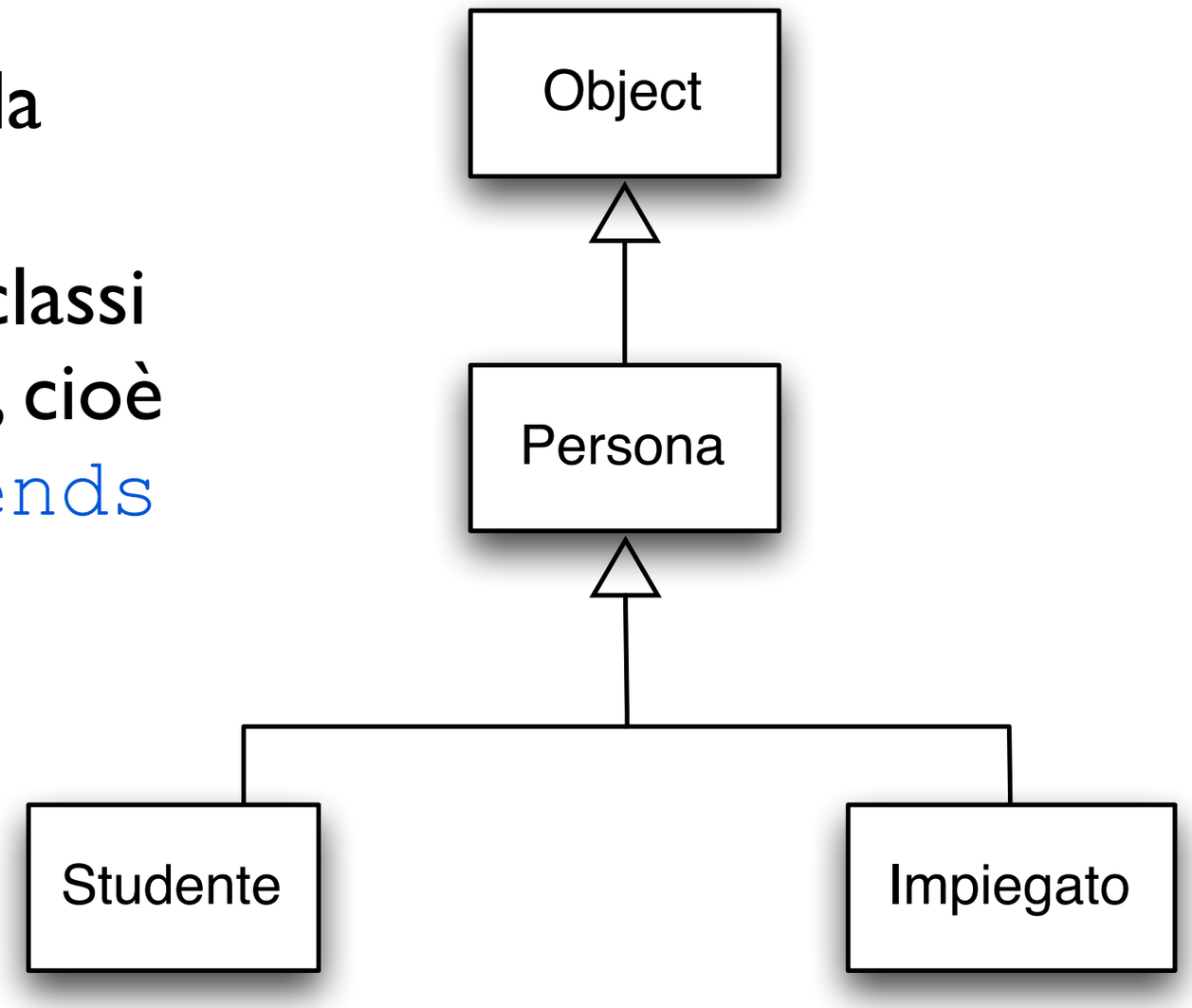
```
public class Persona {
    private String nome;
    private String cognome;
}
```

```
public class Studente extends Persona{
    private String universita;
    private String matricola;
}
```

La classe `Studente` ha quattro attributi: nome, cognome, universita e matricola

7 - Ereditarietà e Polimorfismo

- In Java è supportata solo l'ereditarietà singola, cioè una classe può estendere solo *una* altra classe
- Si crea così un albero delle classi
- La radice di quest'albero è la classe `Object`, di cui sono dunque sottoclassi *tutte* le classi (ciò avviene implicitamente, cioè non occorre scrivere `extends Object`)



7 - Ereditarietà e Polimorfismo

- Quando si crea una sottoclasse è possibile:
  - aggiungere nuovi attributi
  - aggiungere nuovi metodi
  - aggiungere metodi della superclasse ma con signature diversa (*overloading* di metodi, trasversale all'ereditarietà)
  - ridefinire i metodi della superclasse (*overriding* di metodi)

- Mentre invece *non* è possibile:
  - accedere ad attributi `private` della superclasse (ecco perché è sempre consigliabile fornire metodi di accesso)
  - cancellare attributi e metodi della superclasse

7 - Ereditarietà e Polimorfismo

- Esempio di sovraccaricamento (overloading):

```
public class Persona {
    public void presentazione(){
      System.out.print("Mi chiamo" + this.nome);
    }
}
```

```
public class Studente extends Persona{
   public void presentazione(String s){
       System.out.print("Mi chiamo " + this.nome);
       System.out.print(" e mi piace " + s);
   } // overloading del metodo presentazione
}
```

La classe `Studente` contiene due versioni di `presentazione`

7 - Ereditarietà e Polimorfismo

- Esempio di ridefinizione di metodo (overriding):

```
public class Persona {
    public void presentazione(){
      System.out.print("Mi chiamo" + this.nome);
    }
}
```

```
public class Studente extends Persona{
   public void presentazione(){
       System.out.print("Ciao, sono " + this.nome);
   } // overriding del metodo presentazione
}
```

La classe `Studente` contiene una versione di `presentazione`

7 - Ereditarietà e Polimorfismo

- Per quanto riguarda gli attributi, è possibile usare il modificatore `protected`, che permette l'accesso da tutte le sottoclassi
- In definitiva, abbiamo i seguenti modificatori di accesso:

Modifier	Class	Package	Subclass	World
public	✔	✔	✔	✔
protected	✔	✔	✔	✘
no modifier	✔	✔	✘	✘
private	✔	✘	✘	✘

Nota Bene: Attributi e metodi `public` e `protected` sono citati nella `Javadoc` della classe, mentre quelli `private` e `default` invece no

7 - Ereditarietà e Polimorfismo

- Attenzione: se si ridefinisce un attributo della superclasse nella sottoclasse, si sta adombrando l'attributo della superclasse!

```
public class Persona {
    private String nome;
}
```

```
public class Studente extends Persona{
    private String nome;
}
```

La classe `Studente` ha 2 attributi nome, di cui uno adombrato.
In genere è meglio evitare questo comportamento!

7 - Ereditarietà e Polimorfismo

- Un costruttore della sottoclasse invoca implicitamente quello della superclasse che ha gli stessi parametri
- L'invocazione esplicita si fa col riferimento `super`
- L'invocazione esplicita al costruttore della superclasse, se presente, deve essere la prima istruzione del costruttore nella sottoclasse

7 - Ereditarietà e Polimorfismo

Esempio:

```
public class Persona {
   public Persona (String nome, String cognome){
       this.nome = nome;
       this.cognome = cognome;
   }
}
```

```
 public class Studente extends Persona{
    public Studente (String nome, String cognome,
String universita, String matricola){
        super(nome, cognome);
        this.universita = universita;
        this.matricola = matricola;
    }
 }
```

7 - Ereditarietà e Polimorfismo

- Oltre al riferimento al costruttore delle supeclasse mediante la parola chiave `super`, esiste la possibilità di richiamare un costruttore della classe corrente mediante la parola chiave `this`
- Quale sia il costruttore richiamato dipende dai parametri

Esempio:

```
Serbatoio (int livello){
    this.livello = livello;
}

Serbatoio (int livello, int capacita){
  this(livello); // chiama il costruttore sopra
  this.capacita = capacita;
}
```

7 - Ereditarietà e Polimorfismo

- Come ci si potrebbe aspettare, gli attributi e i metodi della superclasse possono essere acceduti attraverso il riferimento `super`

Esempio:

```
public class Studente extends Persona{
   ...
   public String toString(){
      return super.toString()
      + " e studio a " + this.universita;
   } // overriding del metodo toString
}
```

7 - Ereditarietà e Polimorfismo

- In Java è possibile usare un riferimento a un oggetto della superclasse come riferimento a un oggetto di una sottoclasse (*subtyping*)

Esempio:

```
Persona anna = new Persona("Anna", "Cusimano");
 Persona luca = new Studente("Luca", "Maselli",
"Unipa", "0055121314");
 ...
 System.out.println(luca.toString());
```

Il principio è che uno Studente è una Persona, ma una Persona non necessariamente è uno Studente (ad es. non ha matricola)

```
Studente anna = new Persona("Anna", "Cusimano");
//Errore di compliazione!
```

7 - Ereditarietà e Polimorfismo

```
 Persona luca = new Studente("Luca", "Maselli",
"Unipa", "0055121314");
 ...
 System.out.println(luca.toString());
```

Quale metodo `toString` verrà invocato su `luca`? Quello di `Persona` o quello di `Studente`? In altre parole, prevale il tipo del riferimento o il tipo dell'oggetto?

Risposta: prevale il tipo dell'oggetto (questo fenomeno si chiama *dynamic binding*, o *late binding*)

Questo permette il meccanismo del *polimorfismo*

7 - Ereditarietà e Polimorfismo

- Il polimorfismo è il principio secondo cui il comportamento di un metodo si adatta al tipo di oggetto su cui è invocato
- Permette quindi di invocare un metodo su un riferimento a un oggetto della superclasse, e poi applicare (a runtime) la versione (ri)definita nella sottoclasse a cui appartiene l'oggetto effettivamente riferito
- E' un costrutto fondamentale della programmazione orientata agli oggetti (insieme a incapsulamento ed ereditarietà)

7 - Ereditarietà e Polimorfismo

```
 Persona luca = new Studente("Luca", "Maselli",
"Unipa", "0055121314");
 ...
 System.out.println(luca.toString());
```

Attenzione! `luca` è visto come oggetto di tipo `Studente` solo a runtime (grazie al *dynamic binding*), mentre in fase di compilazione è visto come oggetto `Persona` (si ha infatti *static type checking*)

Pertanto, non è possibile invocare su `luca` i metodi specifici della classe `Studente`, ma solo quelli della classe `Persona`, perché altrimenti si ha un errore di compilazione

Se però nella sottoclasse `Studente` il metodo è stato ridefinito (*overriding* del metodo), allora viene applicata la versione ridefinita del metodo (si ha cioè il *dynamic method lookup*)

7 - Ereditarietà e Polimorfismo

Esempio di array polimorfo:

Supponiamo di avere la classe `Persona`, che ha due sottoclassi: `Studente` e `Impiegato`; a sua volta, `Impiegato` ha una sottoclasse `Dirigente`. Se ogni classe ridefinisce il metodo `toString`, cosa stamperà questo codice?

```
Persona[] gente = new Persona[4];
gente[0] = new Persona("Mario", "Rossi");
gente[1] = new Studente("Luca", "Ferrari", "Unipa");
gente[2] = new Impiegato("Gianni", "Bianchi", "Poste");
gente[3] = new Dirigente("Carlo", "Marini", "Poste");

for ( int i = 0 ; i < gente.length ; i++ )
    System.out.println( gente[i] );
```

7 - Ereditarietà e Polimorfismo

<u>Risposta:</u>

Stamperà qualcosa di questo tipo:

```
Mi chiamo Mario Rossi

Mi chiamo Luca Ferrari e studio a Unipa

Mi chiamo Gianni Bianchi e sono impiegato presso
Poste italiane

Mi chiamo Carlo Marini e sono impiegato presso Poste
italiane come dirigente!
```

7 - Ereditarietà e Polimorfismo

- Per accedere ad attributi e metodi (non ridefiniti) della sottoclasse si può ricorrere a un casting esplicito

Esempio:

```
 Persona tizio;
 Studente luca = new Studente(“Luca”, “Maselli”,
“Unipa”, “0055121314”);
 tizio = luca;
 System.out.println(tizio.universita); //Errore

 Studente caio = (Studente) tizio;
 System.out.println(caio.universita); //OK
```

7 - Ereditarietà e Polimorfismo

- Per conoscere il tipo effettivo di un oggetto si può usare il costrutto `instanceof`, che restituisce un boolean
- Si può quindi usare insieme ad un casting esplicito per accedere agli attributi e ai metodi della sottoclasse

Esempio:

```
 Persona tizio = new Studente(“Luca”, “Maselli”,
“Unipa”, “0055121314”);
 if (tizio instanceof Studente) {
    Studente studente = (Studente) tizio;
    System.out.println(studente.universita);
 }
```

7 - Ereditarietà e Polimorfismo

- Se si vuole che una classe non possa avere sottoclassi la si può dichiarare `final`

Esempio:

```
public final class Quadrato extends Rettangolo{
    ...
}
```

7 - Ereditarietà e Polimorfismo

- Se si vuole che un metodo non possa essere ridefinito lo si può dichiarare `final`

Esempio:

```
public final void getNome(){
    return this.nome;
}
```