

Programmazione Orientata agli Oggetti

Estensione (Seconda Parte)

Esercizio

```
public class C {  
    public void dim(C c) {  
        System.out.println("C.dim(C) ");  
    }  
  
    public void dim(L l) {  
        System.out.println("C.dim(L) ");  
    }  
  
    public void dim(K k) {  
        System.out.println("C.dim(K) ");  
    }  
}
```

```
public class K extends C {  
    public void dim(C c) {  
        System.out.println("K.dim(C) ");  
    }  
  
    public void dim(L l) {  
        System.out.println("K.dim(L) ");  
    }  
  
    public void dim(K k) {  
        System.out.println("K.dim(K) ");  
    }  
}
```

```
public class L extends C {  
    public void dim(C c) {  
        System.out.println("L.dim(C) ");  
    }  
  
    public void dim(L l) {  
        System.out.println("L.dim(L) ");  
    }  
  
    public void dim(K k) {  
        System.out.println("L.dim(K) ");  
    }  
  
    public static void main(String args[]) {  
        C a = new K();  
        C b = new L();  
        a.dim(b);  
        L a1 = new L();  
        a1.dim(a);  
    }  
}
```

Sommario

- Estensione nel caso di studio
- Chiamate a metodi della superclasse
- Accesso protetto ai membri
- Overriding di metodi
- Ancora sull'ereditarietà multipla
- La gerarchia dei tipi
- Considerazioni finali sui tipi
- Esercizio: *Overriding for Overloading*

Introduzione

- Riprendiamo lo studio di caso **diadia**
 - supponiamo di voler inserire nel labirinto delle stanze particolari, stanze "magiche", il cui comportamento differisce da quello usuale
- Una stanza magica, esattamente come la stanza ordinaria, ha una descrizione, possiede una collezione di uscite, e può ospitare una collezione di attrezzi

La Stanza Magica (1)

- Una stanza magica ha delle particolarità, che la rendono diversa dalla stanza ordinaria:
 - dopo N volte che in tale stanza viene posato (aggiunto) un qualsiasi attrezzo da parte del giocatore, la stanza inizierà a comportarsi «magicamente»
 - quando la stanza si comporta magicamente, ogni volta che posiamo un attrezzo, la stanza "inverte" il nome dell'attrezzo e ne raddoppia il peso. Ad esempio: se posiamo (togliamo dalla borsa e aggiungiamo alla stanza) l'attrezzo con nome '**chiave**' e peso 2, la stanza memorizza un attrezzo con nome '**evaihc**' e peso 4
 - quando la stanza non si comporta magicamente, il comportamento rimane quello usuale

La Stanza Magica (2)

- Vogliamo introdurre questa *variante* nel nostro gioco
- Una stanza magica deve poter essere usata in qualsiasi punto in cui usiamo oggetti **Stanza**
- La classe della stanza magica pur essendo molto simile a quella della stanza ordinaria, differisce per:
 - alcuni dati in più da gestire
 - alcuni metodi in più che può offrire
 - il comportamento di un metodo
- ✓ Possiamo usare l'estensione e sfruttare il polimorfismo per introdurre questa caratteristica nel gioco

Definizione di Classe Estesa (1)

- La classe **StanzaMagica** rispetto alla classe **Stanza**
 - possiede nuove variabili:
 - **contatoreAttrezziPosati**: memorizza il numero di attrezzi posati (aggiunti)
 - **sogliaMagica**: memorizza il numero di attrezzi da posare prima che si attivi il comportamento «magico» della stanza
 - **SOGLIA_MAGICA_DEFAULT**: valore di default per la soglia
 - possiede un nuovo metodo privato
private Attrezzo modificaAttrezzo(Attrezzo attrezzo) che restituisce un attrezzo a partire dall'attrezzo passato come parametro
 - ridefinisce il metodo **addAttrezzo(Attrezzo attrezzo)** per implementare l'effetto magico
 - ha due costruttori: uno prende nome e soglia, l'altro solo il nome (e imposta la soglia al valore di default)

Esempio: StanzaMagica

```
class StanzaMagica extends Stanza {  
    final static private int SOGLIA_MAGICA_DEFAULT = 3;  
    private int contatoreAttrezziPosati;  
    private int sogliaMagica;  
  
    public StanzaMagica(String nome) {  
        this(nome, SOGLIA_MAGICA_DEFAULT);  
    }  
  
    public StanzaMagica(String nome, int soglia) {  
        super(nome);  
        this.contatoreAttrezziPosati = 0;  
        this.sogliaMagica = soglia;  
    }  
  
    @Override  
    public boolean addAttrezzo(Attrezzo attrezzo) {  
        ...  
    }  
  
    private Attrezzo modificaAttrezzo(Attrezzo attrezzo) {  
        ...// (>>)  
    }  
}
```


Esempio: StanzaMagica

- Un nuovo metodo privato `modificaAttrezzo()` :

```
class StanzaMagica extends Stanza {  
  
    ... ..  
  
    private Attrezzo modificaAttrezzo(Attrezzo attrezzo) {  
        StringBuilder nomeInvertito;  
        int pesoX2 = attrezzo.getPeso() * 2;  
        nomeInvertito = new StringBuilder(attrezzo.getNome());  
        nomeInvertito = nomeInvertito.reverse();  
        attrezzo = new Attrezzo(nomeInvertito.toString(),  
                                pesoX2);  
        return attrezzo;  
    }  
}
```

Definizione di Classe Estesa (2)

- Con oggetti **StanzaMagica** è comunque possibile usare i metodi pubblici definiti nella superclasse quali **getDescrizione()**, **getStanzeAdiacenti()**

```
StanzaMagica labIA = ...
```

```
String s = labIA.getDescrizione();
```

- Questi metodi non sono definiti esplicitamente in **StanzaMagica**, ma vengono *ereditati*
- Allo stesso modo vengono *ereditate* le variabili di istanza
 - (ma solo quelle non private solo visibili nella sottoclasse)

Overriding

- Alcuni metodi della classe base possono essere *ridefiniti* (sovrascritti) nella classe estesa
- Per modificare il comportamento (l'implementazione) di un metodo si effettua un *overriding* (*sovrascrittura*) del metodo
- Nel nostro esempio:

@Override

```
public boolean addAttrezzo(Attrezzo attrezzo)
```

Overriding (Tentativo 1)

```
class StanzaMagica extends Stanza {  
    private int contatoreAttrezziPosati;  
    private int sogliaMagica;  
    ...  
    @Override  
    public boolean addAttrezzo(Attrezzo attrezzo) {  
        this.contatoreAttrezziPosati++;  
        if (this.contatoreAttrezziPosati > this.sogliaMagica)  
            attrezzo = this.modificaAttrezzo(attrezzo);  
        if (this.numeroAttrezzi < this.attrezzi.length) {  
            this.attrezzi[this.numeroAttrezzi] = attrezzo;  
            this.numeroAttrezzi++;  
            return true;  
        }  
        else return false;  
    }  
  
    private Attrezzo modificaAttrezzo(Attrezzo attrezzo) {  
        ...  
    }  
}
```

le variabili `attrezzi` e `numeroAttrezzi` sono ereditate dalla classe base ma non sono accessibili (in quanto private)

Estensione: Overriding

- La soluzione precedente *non* compila: i metodi della classe estesa **StanzaMagica** non possono accedere ai campi *privati* della classe base **Stanza**
 - N.B. anche se ogni oggetto **StanzaMagica** implicitamente possiede le variabili di istanza per rappresentare gli attrezzi contenuti nella stanza
- La classe estesa può accedere solo ai membri pubblici della classe base come tutte le altre classi *esterne*
 - ✓ Coerentemente con il principio dell'*information hiding*
- Nel nostro caso possiamo però pensare di limitarci a riutilizzare il metodo pubblico reso disponibile dalla superclasse **Stanza**:
boolean addAttrezzo(Attrezzo attrezzo)

Overriding (Tentativo 2)

```
class StanzaMagica extends Stanza {  
    private int contatoreAttrezziPosati;  
    private int sogliaMagica;  
    ...  
  
    @Override  
    public boolean addAttrezzo(Attrezzo attrezzo) {  
        this.contatoreAttrezziPosati++;  
        if (this.contatoreAttrezziPosati > sogliaMagica) {  
            attrrezzo.modificaAttrezzo(attrezzo);  
            return true;  
        }  
        return false;  
    }  
  
    private Attrezzo modificaAttrezzo(Attrezzo attrezzo) {  
        ...  
    }  
}
```

NON FUNZIONA: viene chiamato
ricorsivamente (all'infinito) il metodo
addAttrezzo(Attrezzo attrezzo) !

Overriding (Tentativo 2)

- La soluzione precedente non funziona perché il metodo `addAttrezzo(Attrezzo attrezzo)` chiama se stesso!
 - `java.lang.StackOverflowError`
 - ✓ come implicato dal late-binding: per l'esecuzione si usa l'implementazione del tipo dinamico, ovvero il corpo dello stesso metodo che si sta definendo
- E' necessario indicare che vogliamo usare il metodo della superclasse
- Questo è reso possibile usando nuovamente la parola chiave **super**
 - ✓ ma con un nuovo significato

Estensione: Overriding

```
class StanzaMagica extends Stanza {  
    private int contatoreAttrezziPosati;  
    private int sogliaMagica;  
    ...  
  
    @Override  
    public boolean addAttrezzo(Attrezzo attrezzo) {  
        this.contatoreAttrezziPosati++;  
        if (this.contatoreAttrezziPosati > this.sogliaMagica)  
            attrezzo = this.modificaAttrezzo(attrezzo);  
        return super.addAttrezzo(attrezzo);  
    }  
  
    private Attrezzo modificaAttrezzo(Attrezzo attrezzo) {  
        ...  
    }  
}
```


Esercizio

- Scrivere una classe di test **StanzaMagicaTest** per testare la classe **StanzaMagica** che preveda anche test-case per verificarne il comportamento “magico”
- Implementare **StanzaMagica** usando l'ereditarietà secondo le linee guida discusse in queste dispense

Visibilità protected (1)

- I membri **private** della classe base non sono accessibili dall'esterno nemmeno da una sottoclasse
- In Java esiste un altro modificatore di accesso, che consente di definire campi a cui sia consentito l'accesso da sottoclassi
 - È il modificatore di accesso **protected**
- Un membro di una superclasse con accesso protetto è visibile a tutte le sottoclassi
 - ✓ Indipendentemente dal package di appartenenza

Visibilità protected (2)

```
class Stanza {  
    private String descrizione;  
    protected Attrezzo[] attrezzi;  
    protected numeroAttrezzi;  
    ...  
}
```

- In questo modo, qualsiasi classe che estenda `Stanza` può accedere alle *sue* variabili di istanza `attrezzi` e `numeroAttrezzi`
- Si rende possibile una definizione alternativa del metodo `addAttrezzo(Attrezzo attrezzo)` di `StanzaMagica`, con accesso diretto ai campi ereditati da `Stanza`

Accesso ai Membri dalla Classe Estesa

```
class StanzaMagica extends Stanza {  
    private int contatoreAttrezziPosati;  
    private int numeroPassaggi;  
    ...  
    @Override  
    public boolean addAttrezzo(Attrezzo attrezzo) {  
        this.contatoreAttrezziPosati++;  
        if (this.contatoreAttrezziPosati > this.sogliaMagica)  
            attrezzo = this.modificaAttrezzo(attrezzo);  
        if (this.numeroAttrezzi < this.attrezzi.length) {  
            this.attrezzi[this.numeroAttrezzi] = attrezzo;  
            this.numeroAttrezzi++;  
            return true;  
        }  
        else return false;  
    }  
    ...  
}
```

La classe estesa `StanzaMagica` ha la possibilità di accedere ai membri *protetti* (`attrezzi` e `numeroAttrezzi`) della classe base `Stanza`

Visibilità protected (3)

- Più precisamente, è possibile accedere ad un membro **protected**:
 - da tutte le classi estese
 - da tutte le classi dello stesso package!
- I membri protetti sono una violazione (seppur controllata e voluta) dell'information hiding
 - vanno pertanto usati con molta accortezza
 - **se possibile, meglio evitare il loro utilizzo**
- L'utilizzo più appropriato è nella progettazione di framework, ovvero librerie che consentano agevolmente l'estensione da parte di sviluppatori esperti
 - ✓ ... ben oltre i nostri obiettivi formativi

Livelli di visibilità in Java

Access Levels

Modifier	Class	Package	Subclass	World
<code>public</code>	Y	Y	Y	Y
<code>protected</code>	Y	Y	Y	N
<i>no modifier</i>	Y	Y	N	N
<code>private</code>	Y	N	N	N

scontato.

comprensibile...

da ricordare!

- Il livello di visibilità ottenuto senza modificatore viene anche denominato «package-private»
- Il livello di visibilità «protected» è più permissivo del livello «package-private» ed è il livello più permissivo in assoluto dopo «public»
- Il livello di visibilità «package-private» è il meno permissivo in assoluto dopo «private»

Esercizio

- Reimplementare **StanzaMagica** anche utilizzando l'estensione tra classi, impostando il modificatore di accesso delle variabili **attrezzi** e **numeroAttrezzi** della classe base **Stanza** a **protected** (anziché **private**)
- Le successive evoluzioni del nostro studio di caso saranno un'occasione per vedere quali problemi la violazione dell'information hiding può comportare durante la manutenzione del codice...

Overriding ed Information Hiding

- Solo i metodi pubblici e protetti della classe base possono essere sovrascritti
- Se proviamo a ridefinire un metodo **private** della classe base quello che otteniamo è un nuovo metodo
 - N.B. il nuovo metodo «nasconde» l'omonimo metodo offerto nella superclasse ma *non* lo sovrascrive affatto
- In generale è possibile sovrascrivere un metodo e cambiare il modificatore di accesso, ma solo mantenendo od *ampliandone* la visibilità
 - Perché? Sugg.: pensare al principio di sostituzione
- Quindi per sovrascrivere un metodo, i livelli di visibilità permessi sono:
 - *sovrascritto* → *sovrascrivente*
 - **protected** → **protected**
 - **protected** → **public**
 - **public** → **public**

Visibilità dei Metodi Sovrascritti

- Pertanto il seguente codice è corretto:

```
public class Superclasse {  
    protected void metodo() {}  
}  
  
public class Sottoclasse extends Superclasse {  
    @Override  
    public void metodo() {} // da protected a public OK  
}
```

- mentre invece il seguente non compila:

```
public class Superclasse {  
    public void metodo() {}  
}  
  
public class Sottoclasse extends Superclasse {  
    @Override  
    protected void metodo() {} // da public a protected ERRORE
```

Cambiare Segnatura in Override

- Permettere di sovrascrivere un metodo solo mantenendone od *ampliandone* la visibilità, è la scelta più sensata?
 - ✓ Rispondere applicando il p.d.s.:
 - un metodo che sovrascrive e rimpiazza un altro metodo *deve* certamente permettere tutti gli utilizzi originali
 - Al più, può consentirne *anche* di nuovi
 - ✓ Ampliare la visibilità è coerente con il p.d.s.!
- Finora abbiamo solo visto casi di metodi sovrascritti con ampliamento della visibilità, ma della stessa identica segnatura. Ad es. in `Stanza` e `StanzaMagica`:

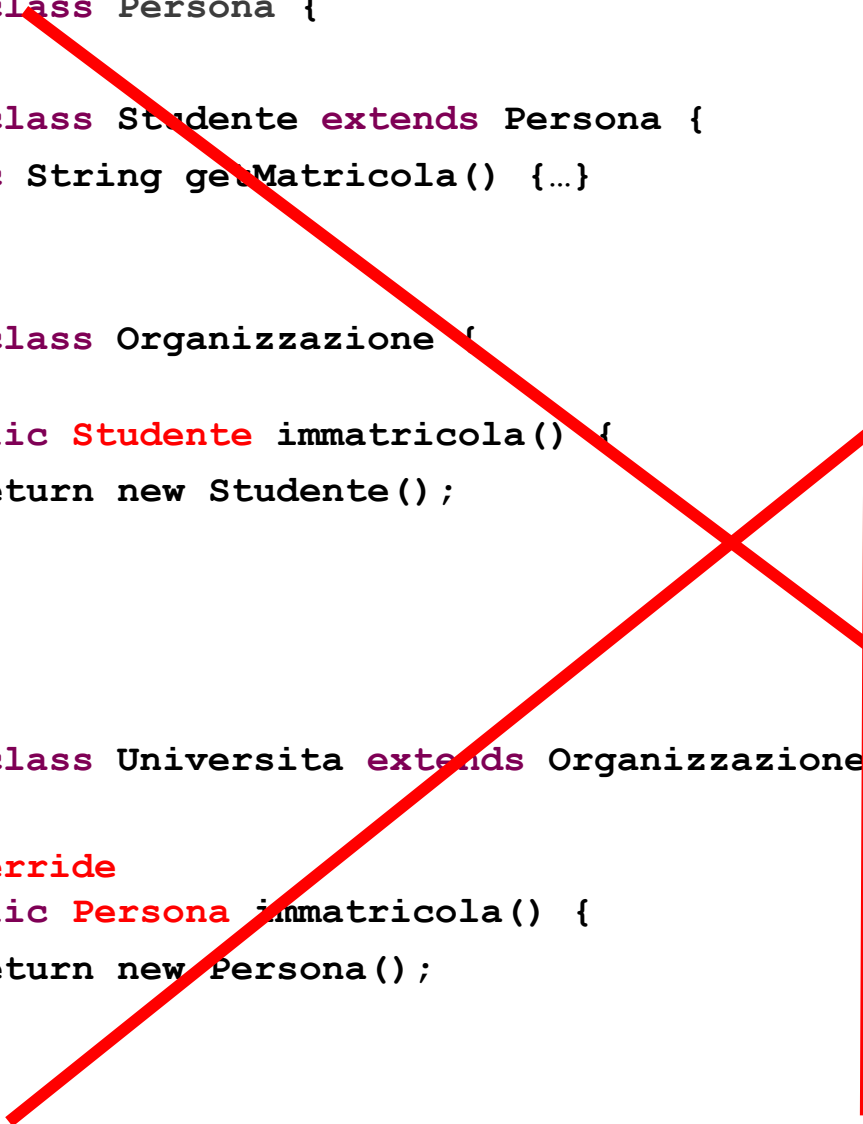
```
public boolean void addAttrezzo(Attrezzo a);
```
- Appliciamo lo stesso ragionamento per capire se ha senso cambiare il tipo restituito e/o quello dei parametri dei metodi sovrascritti

Overriding: Metodi di Risultato Polimorfo (1)

- Consideriamo questi due metodi
 - `public Studente immatricola()`
 - `public Persona immatricola()`con **Persona** supertipo di **Studente**
- Quale di questi due metodi in una classe estesa può sovrascrivere l'altro collocato nella classe base? Perché?
 - Sugg.: applicare il principio di sostituzione che *deve* restare valido
- ✓ Scriviamo due classi **Organizzazione** ed **Universita**, con la classe **Universita** che estende la classe **Organizzazione**...

Overriding: Metodi di Risultato Polimorfo (2)

```
public class Persona {  
}  
  
public class Studente extends Persona {  
    public String getMatricola() {...}  
}  
  
public class Organizzazione {  
  
    public Studente immatricola() {  
        return new Studente();  
    }  
  
}  
  
public class Università extends Organizzazione {  
  
    @Override  
    public Persona immatricola() {  
        return new Persona();  
    }  
  
}
```



NON COMPILA:

```
$ javac *.java
```

```
Università.java:6: error:  
immatricola() in Università cannot  
override immatricola() in  
Organizzazione
```

```
    public Persona immatricola() {  
        ^
```

return type Persona is
not compatible with
Studente

```
Università.java:5: error: method  
does not override or implement a  
method from a supertype
```

```
    @Override  
    ^
```

2 errors

Overriding: Metodi di Risultato Polimorfo (3)

```
public class Persona {  
}  
  
public class Studente extends Persona {  
    public String getMatricola() {...}  
}  
  
public class Organizzazione {  
    public Persona immatricola() {  
        return new Persona();  
    }  
}  
  
public class Universita extends Organizzazione {  
    @Override  
    public Studente immatricola() {  
        return new Studente();  
    }  
}
```

COMPILA!

Un metodo può sovrascrivere un metodo di identica segnatura ma tipo restituito più generale

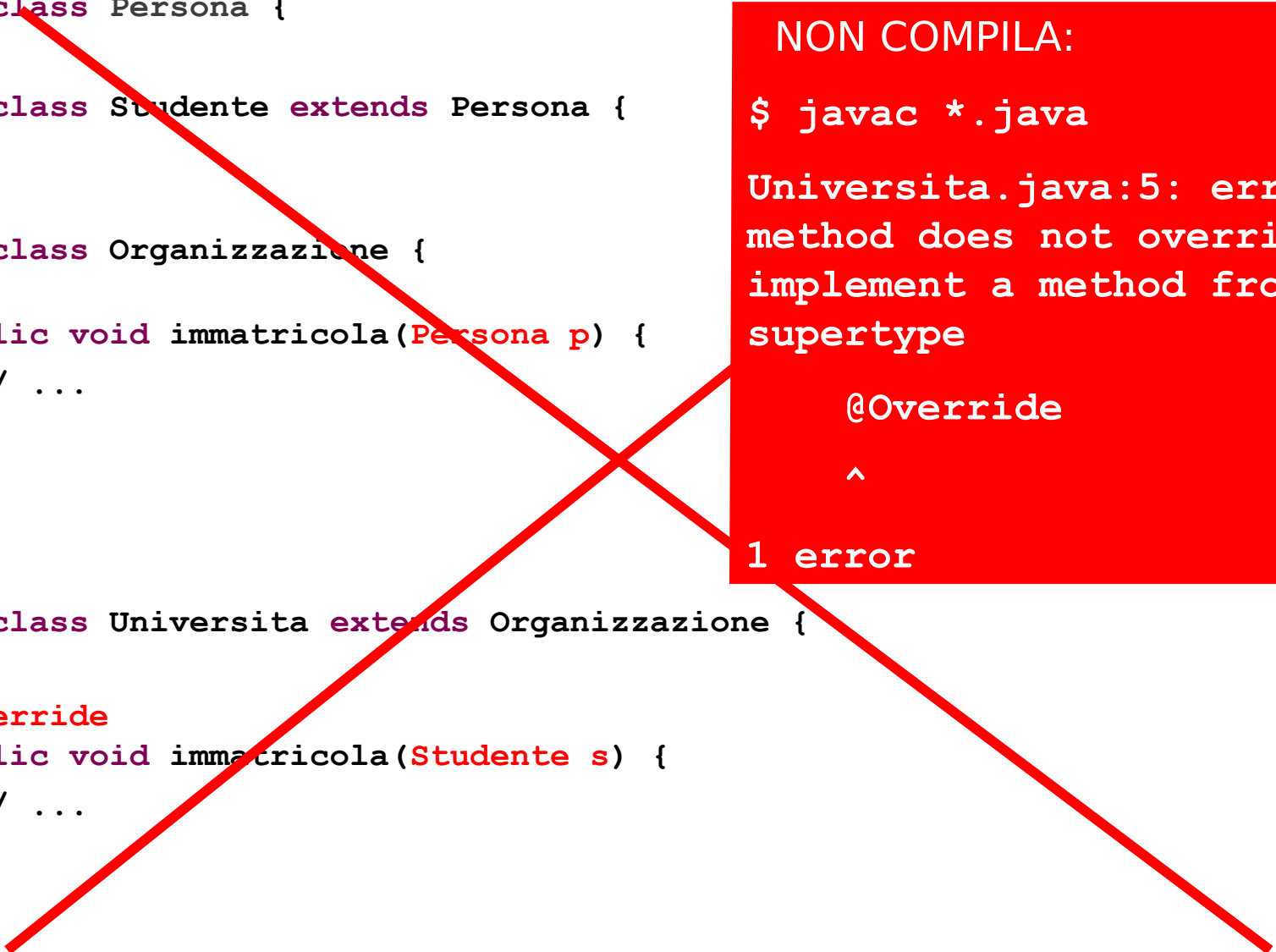
Si usa dire che il tipo restituito dai metodi sovrascritti è *covariante* con la derivazione tra classi: entrambi tendono a generalizzarsi od a specializzarsi muovendosi in su od in giù lungo la gerarchia dei tipi

Overriding: Metodi con Parametri Polimorfi (1)

- Consideriamo questi due metodi
 - `public void immatricola(Studiante s)`
 - `public void immatricola(Persona p)`con `Persona` supertipo di `Studiante`
- Quale di questi due metodi in una classe derivata può sovrascrivere l'altro collocato nella classe base e... perché?
- ✓ Scriviamo due classi `Organizzazione` ed `Universita`, con la classe `Universita` che estende la classe `Organizzazione`...

Overriding: Metodi con Parametri Polimorfi (2)

```
public class Persona {  
}  
public class Studente extends Persona {  
}  
  
public class Organizzazione {  
    public void immatricola(Persona p) {  
        // ...  
    }  
}  
  
public class Università extends Organizzazione {  
  
    @Override  
    public void immatricola(Studente s) {  
        // ...  
    }  
}
```



NON COMPILA:

```
$ javac *.java
```

Università.java:5: error:
method does not override or
implement a method from a
supertype

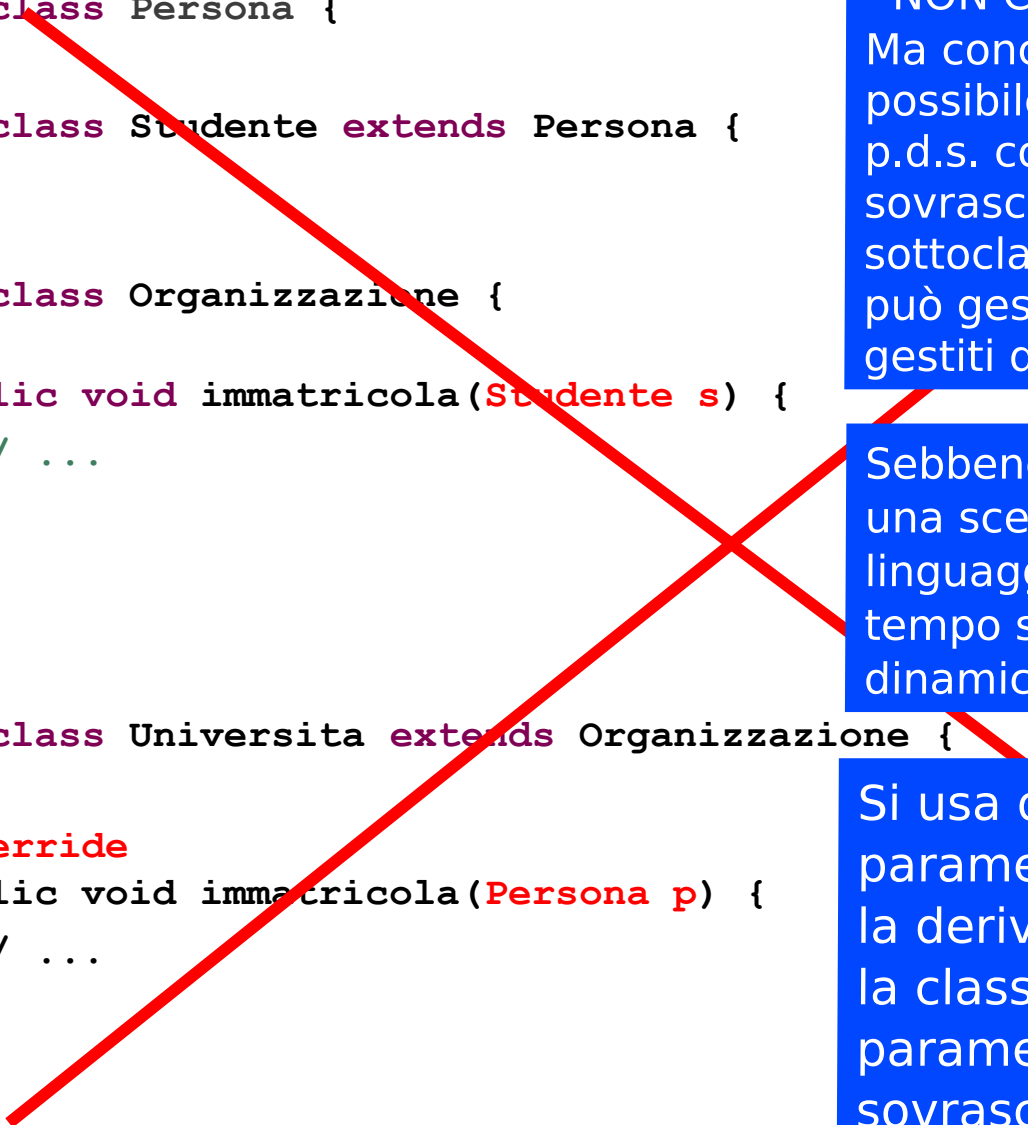
@Override

^

1 error

Overriding: Metodi con Parametri Polimorfi (3)

```
public class Persona {  
}  
  
public class Studente extends Persona {  
}  
  
public class Organizzazione {  
    public void immatricola(Studente s) {  
        // ...  
    }  
}  
  
public class Universita extends Organizzazione {  
  
    @Override  
    public void immatricola(Persona p) {  
        // ...  
    }  
}
```



NON COMPILA lo stesso!
Ma concettualmente dovrebbe essere possibile: ovvero si rispetterebbe il p.d.s. considerandoli uno la versione sovrascritta dell'altro perché la sottoclasse ospita un metodo che può gestire un sovrainsieme dei casi gestiti dal metodo sovrascritto.

Sebbene sorprendente, è comunque una scelta coerente con il resto del linguaggio che risolve l'*overloading* a tempo statico e non a tempo dinamico. >>

Si usa dire che il tipo dei parametri è *controvariante* con la derivazione tra classi: quando la classe viene specializzata, i parametri (dei metodi sovrascritti) tendono a generalizzarsi e viceversa

Overriding: Metodi con Parametri Polimorfi (4)

```
public class Persona {  
}  
  
public class Studente extends Persona {  
}  
  
public class Organizzazione {  
    public void immatricola(Studente s) {  
        System.out.println("Organizzazione.immatricola(Studente)");  
    }  
}  
  
public class Università extends Organizzazione {  
  
    public void immatricola(Persona p) {  
        System.out.println("Università.immatricola(Persona)");  
    }  
}
```

COMPILA ma non è un overriding!

Togliendo l'annotazione `@Override`, compila e la sottoclasse `Università` finisce per disporre di due metodi (di cui uno ereditato) sovraccarichi (overloading) e distinti, ma NON sovrascritti (overriding)

Un Overriding che Diventa Overloading

```
static public void main(String args[]) {  
    Persona p = new Persona();  
    Persona ps = new Studente();  
    Studente s = new Studente();  
    Organizzazione org = new Organizzazione();  
    Organizzazione studi = new Università();  
    Università rm3 = new Università();  
//org.immatricola(p); // ERRORE: NON COMPILA  
//org.immatricola(ps); // ERRORE: NON COMPILA  
    org.immatricola(s);  
//studi.immatricola(p); // ERRORE: NON COMPILA  
//studi.immatricola(ps); // ERRORE: NON COMPILA  
    studi.immatricola(s);  
    rm3.immatricola(p);  
    rm3.immatricola(ps);  
    rm3.immatricola(s);  
}
```

Togliendo l'annotazione `@Override`
COMPILA ma non è un overriding!

Università dispone di due metodi
sovraccarichi (overloading): quale
invocare viene deciso direttamente
dal compilatore solo sulla base del
tipo statico

L'esecuzione stampa:

```
Organizzazione.immatricola(Studente)  
Organizzazione.immatricola(Studente)  
Università.immatricola(Persona)  
Università.immatricola(Persona)  
Organizzazione.immatricola(Studente)
```

Riassunto: Covarianza & Controvarianza

Organizzazione

`protected Persona immatricula(Studente)`

tipo + generale

Studente

Persona

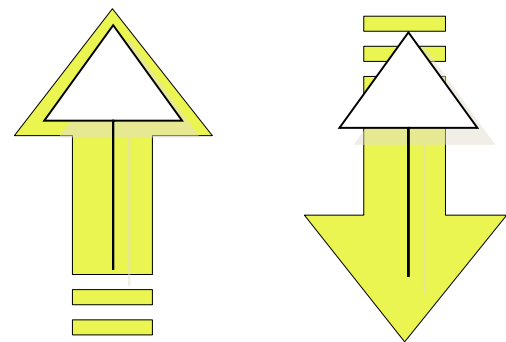
Università

`@Override`

`public Studente immatricula(Persona)`

tipo + speciale

Specializzando \uparrow il tipo che ospita un metodo,
affinché si possa sovrascrivere:
visibilità \downarrow – tipo restituito \uparrow – tipo parametro \downarrow



covarianza controvarianza

La Gerarchia delle Classi Java:

La Classe `Object`

- Abbiamo già visto che in Java tutte le classi estendono automaticamente la classe `Object`
- E' una classe predefinita, che viene automaticamente estesa da ogni nuova classe (direttamente o indirettamente)
- N.B. scrivere:

```
public class MiaClasse {  
}
```

è del tutto equivalente a scrivere:

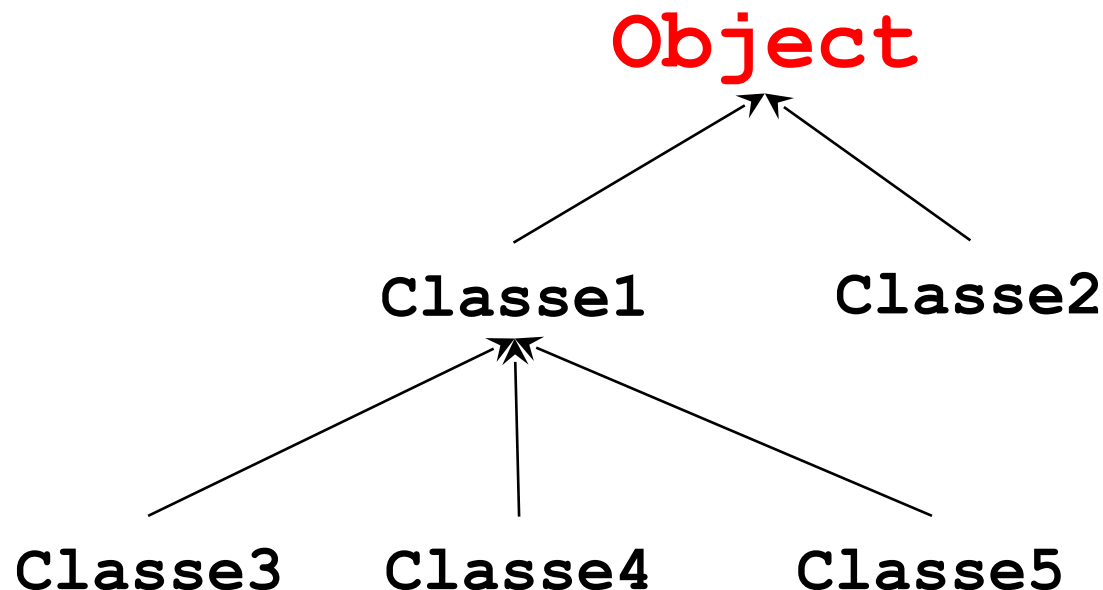
```
public class MiaClasse extends Object {  
}
```

Gerarchie di Classi

- In Java una classe può essere estesa da molte classi, ma ogni classe estende sempre *una ed una sola* classe
 - tranne `Object`, che è la radice predefinita della gerarchia di classi
- Non ci può essere «*ereditarietà multipla*» delle implementazioni

La Gerarchia delle Classi Java

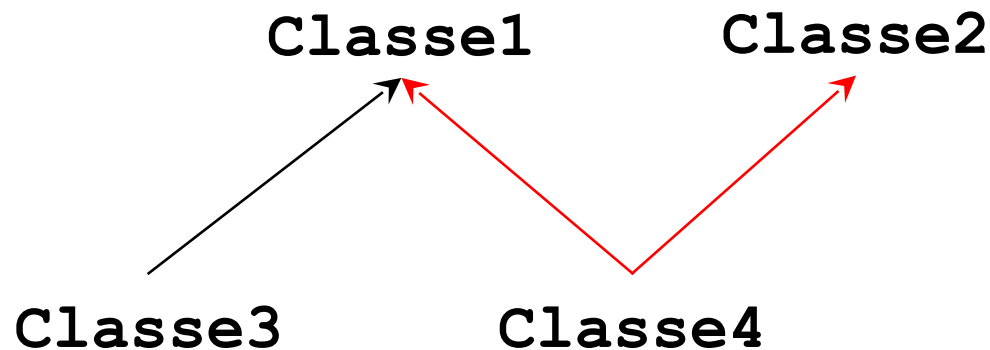
- Un'unica radice: `Object`
- Ogni classe* ha una e una sola superclasse
- Ogni classe può avere zero o più sottoclassi



* con l'eccezione di `Object`

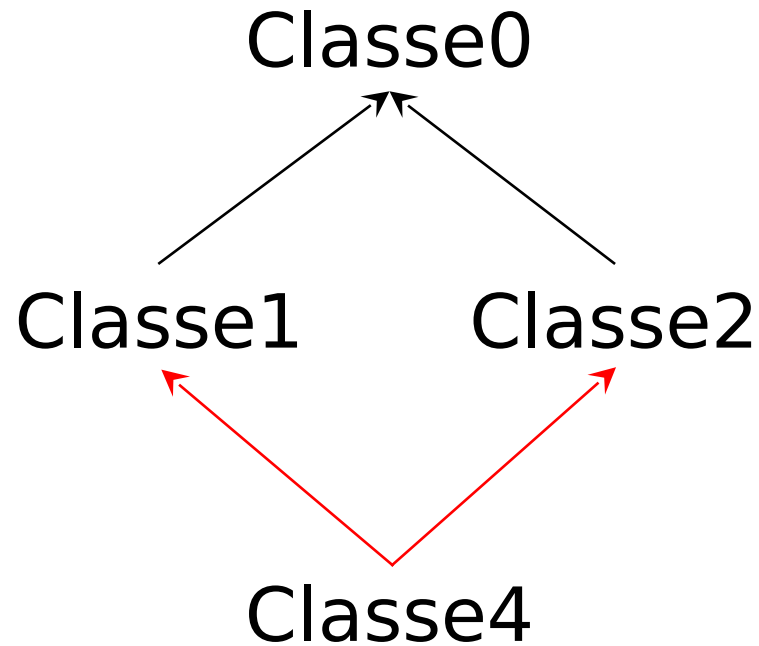
Ereditarietà Multipla non Ammessa

- Se `Classe4` ereditasse sia da `Classe1` che da `Classe2`
- ✓ N.B. In Java **non** è possibile!



- Quali problemi?

Problemi con l'Ereditarietà Multipla: ***L'Ereditarietà a Diamante***

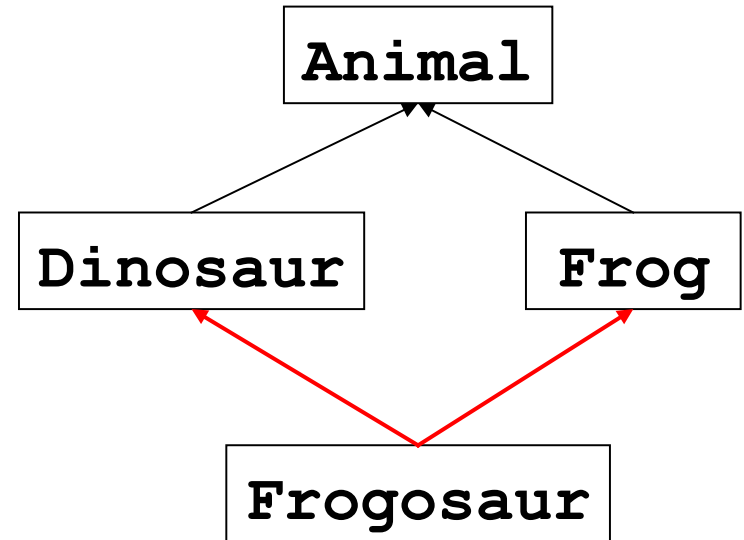


La gerarchia del *Frogosauro*

```
class Animal {  
    void talk() {  
        System.out.println("...");  
    }  
}
```

```
class Frog extends Animal {  
    void talk() {  
        System.out.println("Ribit, ribit.");  
    }  
}
```

```
class Dinosaur extends Animal {  
    void talk() {  
        System.out.println("I'm a dinosaur: I'm cool! ");  
    }  
}
```



II *Frogosau*so

// NON COMPILA

```
class Frogosaur extends Frog, Dinosaur {  
}
```

✓ Cosa dovrebbe fare la seguente chiamata a `talk()`?

```
Animal animal = new Frogosaur();  
animal.talk();
```

Problemi di Ereditarietà Multipla (1)

- Se un membro (metodo o campo) è definito in entrambe le classi base, da quale delle due la classe estesa «eredita» l'implementazione?
 - E se le due classi base a loro volta estendono una superclasse comune ?
 - Il problema è legato alle implementazioni, che vengono ereditate e potenzialmente «confuse»
- ✓ Per questo motivo, in Java si adotta una scelta molto conservativa:
- una classe può implementare *tante* interfacce
 - ma può estendere sempre e solo una *unica* superclasse

Problemi di Ereditarietà Multipla (2)

- In realtà il problema di quale implementazione scegliere, anche in questi casi, è perfettamente risolvibile e risolto in alcuni linguaggi di programmazione
 - (C++, Scala, in una forma molto limitata anche Java 8+)
- Il *Problema del Frogosauro* fa quasi parte del folklore oramai, quasi impossibile non parlarne!
- ✓ Ma solo lascamente chiarisce il vero problema:
 - “mischiare” due implementazioni diverse è estremamente complicato e quasi sempre non necessario
 - comodo in pochi e particolari casi (implementazioni che sono sostanzialmente ortogonali: per i più interessati vedere *mixin*) ben oltre gli obiettivi formativi di questo corso

Implementazione di più Interfacce

- Al contrario in Java è possibile che una classe implementi molteplici interface, una politica molto permissiva:

```
public interface Persona {  
    ...  
}  
  
public class Dirigente implements Impiegato, Persona {  
    ...  
}
```

- ✓ Una classe già ampiamente utilizzata implementa tre interfacce: `java.lang.String`

```
public final class String implements  
    Serializable, Comparable<String>, CharSequence
```

Java Interface ed Ereditarietà Multipla (1)

- Il problema dell'ereditarietà multipla (delle implementazioni) non sussiste affatto con le interface
- Consideriamo infatti una classe che implementa più di una interface
- Per essere concreta e risultare istanziabile:
 - è costretta a fornire direttamente tutte le implementazioni di tutti metodi implementati
 - ✓ si evitano così alla radice tutti i problemi legati all'eredità multipla delle implementazioni

Java Interface ed Ereditarietà Multipla (2)

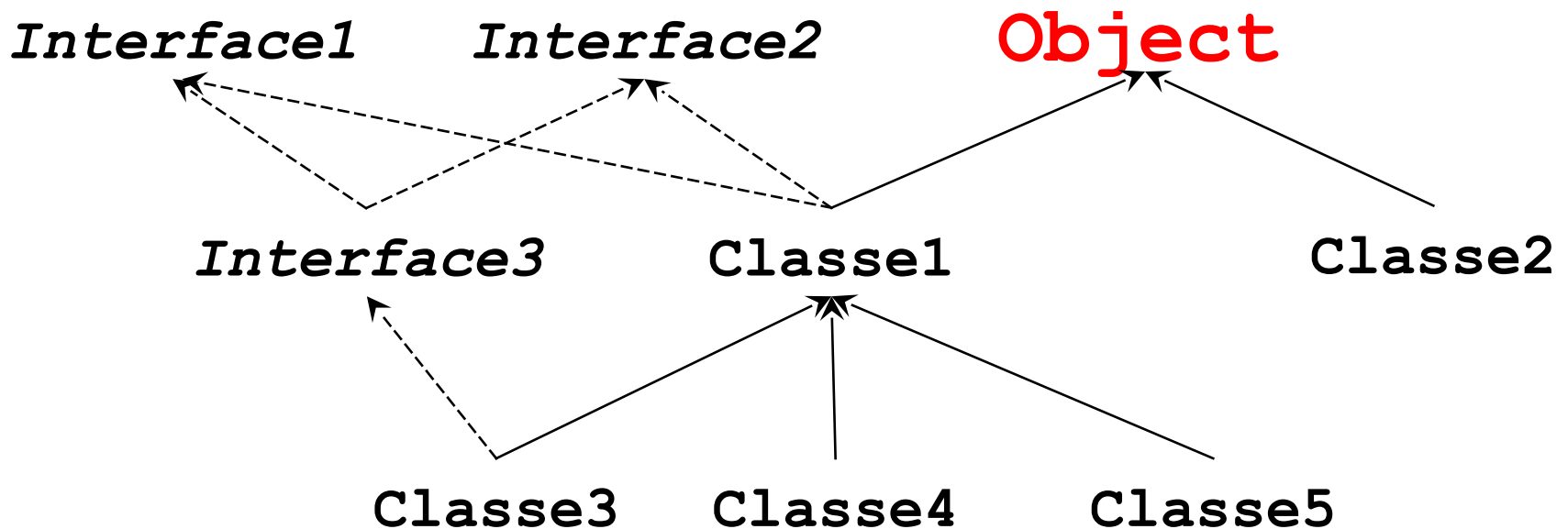
- Abbiamo anche visto che una interface può essere definita per estensione da un'altra interface
- In generale una interface può estendere anche più di una interface scrivendo direttamente:

```
public interface A {}  
public interface B {}  
public interface C {}  
  
public interface I extends A,B,C {}
```

- Pertanto la gerarchia dei *tipi* è più articolata di quella delle classi in quanto contempla anche i tipi definiti tramite le interface...

La Gerarchia dei Tipi Java

- ✓ Un'unica radice: **Object**
 - Ogni classe* ha una ed una sola superclasse
 - Ogni classe può avere zero o più sottoclassi
 - Ogni classe può implementare zero o più interface
 - Ogni interface può estendere zero o più interface



* con l'eccezione di **Object**

Gerarchia delle Classi Lineare

- Ogni classe estende sempre una ed una sola classe
- Tranne `Object`, che è la radice predefinita della gerarchia di classi (e tipi)
- Ma una classe può essere estesa da molte classi
- Non ci può essere ereditarietà multipla delle *implementazioni*
- Si usa dire che la gerarchia delle classi è *lineare*
 - ✓ le implementazioni di tutti i metodi di una classe si trovano in una sequenza lineare di classi e superclassi che conducono sino alla radice `Object`
 - ✓ Attenzione: non più perfettamente vero in Java 8+
 - metodi di interface con implementazioni di *default*

L'importanza dei Tipi (1)

- La scelta dei corretti tipi da definire in un programma è un delicato esercizio di modellazione
- Si può affrontare a ragion veduta solo dopo studio, pratica ed interi corsi specificatamente dedicati alla *Analisi & Progettazione* (come APS)
- E' opportuno cercare *subito* di prevenire alcuni degli errori più ricorrenti nei primi utilizzi dei meccanismi di definizione dei tipi
 - Non considerare gli aspetti *dinamici*
(il comportamento degli oggetti)
 - Ovvero considerare solo gli aspetti *statici*
(dati che modellano)

L'importanza dei Tipi (2)

- Su altri errori tipici dovuti alla mancanza di pratica meglio tornarci in seguito (>>)
- Tra i più rilevanti la *tipizzazione anemica*
 - possono scherzosamente chiamarsi sulla base del nome di un tipo di cui si ha un'irrefrenabile tendenza ad abusare, quasi fosse una «malattia»
 - La «*stringhite*» (>>)
 - La «*mappite*» (>>)
- Sono «sintomi» dell'incapacità di scegliere correttamente i tipi da definire

Considerazioni Finali: Tipi e Sottotipi

- Un frequente errore è sicuramente legato alla tendenza a fissare le relazioni tra tipi concentrandosi *solo* sugli aspetti *statici* delle classi e trascurando invece quelli *dinamici*
- Domanda: Ma... *Quadrato* è sottotipo di *Rettangolo*?
Farsi sempre guidare dal *principio di sostituzione*: ogni qualvolta mi aspetto un **Rettangolo** posso utilizzare al suo posto un **Quadrato**?
- Risposta: *dipende!* Se devo calcolarne il perimetro sì, ma se devo raddoppiarne l'altezza senza cambiarne la base sicuramente no!
 - ✓ Il p.d.s. deve valere anche a tempo dinamico, anche dopo i cambiamenti di stato degli oggetti coinvolti

Tipi e Sottotipi:

String VS StringBuilder (1)

- Altro esempio; cfr. **String** VS **StringBuilder**:
- **String** si usa per oggetti *immutabili*:
 - Una volta creato un oggetto `java.lang.String`, non è poi più possibile cambiarne lo stato
 - ✓ (E' tuttavia possibile creare *nuovi* oggetti stringhe sulla base del primo)
- Per costruire progressivamente stringhe particolarmente lunghe, è spesso preferibile utilizzare la versione modificabile delle stringhe: classe `java.lang.StringBuilder`
- **StringBuilder** si usa per oggetti *mutabili*:

```
public final class StringBuilder ...???... {  
    StringBuilder append(String str);  
  
    ...  
    String toString();  
}
```

- **StringBuilder** deve essere sottotipo di **String**???

Tipi e Sottotipi:

String VS StringBuilder (2)

✓ Cfr. **String/StringBuilder/CharSequence**:

```
public final class String extends Object
    implements
        Serializable, Comparable<String>, CharSequence {...}
```

```
public final class StringBuilder extends Object
    implements Serializable, CharSequence {...}
```

```
public interface CharSequence {
    public char charAt(int index);
    public int length();
    public CharSequence subSequence(int start, int end);
    public String toString();
}
```

✓ **StringBuilder** NON è un sottotipo (né potrebbe esserlo) di **String** ma di un loro supertipo comune **CharSequence**; si prevede esplicitamente la convertibilità degli oggetti **StringBuilder** a **String** tramite **toString()**

Esercizio:

Overriding for Overloading (1)

- Rafforziamo la comprensione del legame tra l'overloading e l'overriding in Java svolgendo un particolare esercizio
- Vogliamo invocare un metodo sovraccarico proprio sulla base del tipo dinamico del suo parametro e *NON* sulla base del suo tipo statico
- N.B. *NON* è possibile farlo con i meccanismi offerti direttamente dal linguaggio Java
 - In Java i metodi sovraccarichi sono risolti sulla base del tipo STATICO e mai sulla base del tipo DINAMICO!
- Mostriamo come mediante l'overriding di un metodo polimorfo ed il late-binding sia possibile «simulare» la risoluzione di un metodo sovraccarico sulla base del tipo dinamico di un parametro

Esercizio:

Overriding for Overloading (2)

- Consideriamo un semplice esempio: è data una gerarchia di tipi avente come radice l'interface ***Forma*** e sottotipi concreti ***Quadrato***, ***Cerchio***,... ma non è possibile cambiarne il codice
- ✓ Per quanto l'esempio sia volutamente stilizzato, nella pratica questa situazione si presenta non di rado
 - ad es. durante l'uso di gerarchie di tipi definite da librerie esterne il cui codice risulta imm modificabile
 - ad es. quelle che gestiscono composizioni di oggetti tutti di uno stesso supertipo comune ma di diversi sottotipi concreti

Esempio: Una Gerarchia di Tipi (Sorgente Non Modificabile)

```
public interface Forma {  
    }  
public class Cerchio implements Forma {  
    private int raggio;  
    public Cerchio(int r) { this.raggio = r; }  
    public int getRaggio() { return this.raggio; }  
}  
  
public class Quadrato implements Forma {  
    private int lato;  
    public Quadrato(int l) { this.lato = l; }  
    public int getLato() { return this.lato; }  
}
```

- Si vuole calcolare l'area delle forme, che dipende dal tipo, ma NON è possibile modificare l'interface *Forma* per aggiungere il metodo

```
public float getArea();
```

e quindi non è possibile implementarlo in *Quadrato*, *Cerchio*, ...

Risoluzione Tramite Overriding (1)

```
public class CalcolatoreDiArea {  
    public float areaDi(Cerchio c) {  
        int r = c.getRaggio();  
        return 3.14f * r * r;  
    }  
  
    public float areaDi(Quadrato q) {  
        int l = q.getLato();  
        return l * l;  
    }  
  
    public static void main(String args) {  
        CalcolatoreDiArea calcolatore = new CalcolatoreDiArea();  
        Cerchio cerchio = new Cerchio(1);  
        Quadrato quadrato = new Quadrato(2);  
        System.out.println(calcolatore.areaDi(cerchio));  
        System.out.println(calcolatore.areaDi(quadrato));  
    }  
}
```

COMPILA

Ma solo perché si è avuto cura di fare coincidere tipo statico e tipo dinamico delle variabili locali **cerchio** e **quadrato**.

Non è sempre possibile...

Risoluzione Tramite Overriding (2)

```
public class CalcolatoreDiArea {  
    public float areaDi(Cerchio c) {  
        int r = c.getRaggio();  
        return 3.14f * r * r;  
    }  
  
    public float areaDi(Quadrato q) {  
        int l = q.getLato();  
        return l * l;  
    }  
  
    public static void main(String args) {  
        CalcolatoreDiArea calcolatore = new CalcolatoreDiArea();  
        Forma cerchio = new Cerchio(1);  
        Forma quadrato = new Quadrato(2);  
        System.out.println(calcolatore.areaDi(cerchio)); // ERRORE: NON COMPILA  
        System.out.println(calcolatore.areaDi(quadrato)); // ERRORE: NON COMPILA  
    }  
}
```

NON COMPILA:

The method `areaDi(Cerchio)` in the type `CalcolatoreDiArea` is not applicable for the arguments `(Forma)`

- Si supponga invece di voler calcolare l'area totale di una collezione di forme di cui non sia possibile prevedere il tipo a tempo dinamico...

Risoluzione Tramite Overriding (3)

```
public class CalcolatoreDiArea {  
    public float areaDi(Cerchio c) {  
        int r = c.getRaggio();  
        return 3.14f * r * r;  
    }  
    public float areaDi(Quadrato q) {  
        int l = q.getLato();  
        return l * l;  
    }  
    static public float sommaAll(CalcolatoreDiArea calcolatore, Forma[] forme) {  
        float acc = 0;  
        for(Forma forma : forme) {  
            acc += calcolatore.areaDi(forma); // ERRORE: NON COMPILA  
        }  
        return acc;  
    }  
    public static void main(String args) {  
        CalcolatoreDiArea calcolatore = new CalcolatoreDiArea();  
        Forma[] forme = { new Cerchio(1), new Quadrato(2) } ;  
        System.out.println(sommaAll(calcolatore, forme));  
    }  
}
```

NON COMPILA:

The method `areaDi(Cerchio)`
in the type `CalcolatoreDiArea` is
not applicable for the arguments
(`Forma`)

Overriding for Overloading (1)

- Una libreria che pubblica una gerarchia di tipi (con sorgente imm modificabile) può essere predisposta per la definizione di codice *cliente* che sappia specializzare i propri comportamenti sulla base del tipo *dinamico*
- L'interface **Forma** viene cambiata per ospitare un metodo che permetta a tutti i tipi di forme di «accettare» un generico **Calcolatore**:

```
public interface Forma {  
    public float accetta(Calcolatore c);  
}
```

- Questo deve saper distinguere tutti i tipi di forme della gerarchia

```
public interface Calcolatore {  
    public float calcola(Cerchio c);  
    public float calcola(Quadrato q);  
}
```

- ✓ Parliamo di «generici» calcoli; il calcolo dell'area è solo un esempio di uno dei loro possibili «istanziamenti»...

Overriding for Overloading (2)

- Il metodo `calcola()` trova questa implementazione nei sottotipi concreti `Cerchio` e `Quadrato`:

```
public class Cerchio    // lo stesso dicasi per Quadrato
    implements Forma {
    @Override
    public float accetta(Calcolatore c) { // N.B. il tipo statico
        // di this è quello della classe corrente, qui Cerchio
        return c.calcola(this); // tipo statico Cerchio
    }
}
```

- ✓ La chiamata al metodo polimorfo `float accetta(Calcolatore)` di `Forma` come sovrascritto (overload) in `Cerchio` finisce per servire a fissare il tipo statico dell'argomento alla chiamata al metodo sovraccarico (override) `Calcolatore.calcola(Cerchio)`!
- ✓ In `Quadrato` servirà per chiamare l'altro metodo sovraccarico `Calcolatore.calcola(Quadrato)`

Overriding for Overloading (3)

```
public interface Forma {  
    public float accetta(Calcolatore c);  
}  
  
public class Cerchio implements Forma {...  
    @Override  
    public float accetta(Calcolatore c) {  
        return c.calcola(this);  
    }  
}  
  
public class Quadrato implements Forma {...  
    @Override  
    public float accetta(Calcolatore c) {  
        return c.calcola(this);  
    }  
}  
  
public interface Calcolatore {  
    public float calcola(Cerchio c);  
    public float calcola(Quadrato q);  
}
```

E' possibile mettere a fattor comune
in una superclasse tutti i metodi
accetta(Calcolatore)???

Tipo statico Cerchio

Tipo statico Quadrato

Overriding for Overloading (4)

```
public class CalcolatoreDiArea implements Calcolatore {  
    @Override  
    public float calcola(Cerchio c) {  
        return 3.14f * c.getRaggio() * c.getRaggio();  
    }  
    @Override  
    public float calcola(Quadrato q) {  
        return q.getLato() * q.getLato();  
    }  
  
    static public float sommaAll(Calcolatore calc, Forma[] forme) {  
        float acc = 0;  
        for(Forma forma : forme) {  
            acc += forma accetta(calc); // COMPILA!  
        }  
        return acc;  
    }  
  
    public static void main(String args) {  
        Calcolatore calcAree = new CalcolatoreDiArea();  
        Forma[] forme = { new Cerchio(1), new Quadrato(2) } ;  
        System.out.println(sommaAll(calcAree, forme));  
    }  
}
```


Ulteriori Calcolatori (Dipendenti dal Tipo Dinamico)

- E' banalmente possibile implementare altre tipologie di Calcolatore. Ad es.:

```
public class CalcolatoreDiPerimetro implements Calcolatore {  
    @Override  
    public float calcola(Cerchio c) {  
        Return 2 * 3.14f * c.getRaggio();  
    }  
    @Override  
    public float calcola(Quadrato q) {  
        return 4 * q.getLato();  
    }  
}
```

- Il metodo statico `sommaAll()` continua a funzionare:

```
public static void main(String args) {  
    Calcolatore calcPerim = new CalcolatoreDiPerimetro();  
    Forma[] forme = { new Cerchio(1), new Quadrato(2) } ;  
    System.out.println(sommaAll(calcPerim, forme));  
}
```

Conclusioni: *Overriding for Overloading (1)*

- Riassumendo: si è aggiunto un livello di indirizzione affinché il corpo del metodo polimorfo **accetta (Calcolatore)** di **Forma** fissi il tipo statico dell'argomento della chiamata al metodo sovraccarico **calcola(...)** di **Calcolatore**
- Per ottenere una chiamata sovraccarica risolta (a tempo di esec.) sulla base del tipo *dinamico* del suo unico parametro vengono fatte:
 - ✓ *prima*, una chiamata *polimorfa* risolta *dinamicamente*
 - ✓ *poi*, una chiamata *sovraccarica* risolta *staticamente*

Conclusioni:

Overriding for Overloading (2)

- Lo scopo di questo esercizio era chiarire i legami (ma anche ribadire le significative differenze) esistenti tra due meccanismi tipici della programmazione orientata agli oggetti
 - ✓ *Overriding vs Overloading*
- Si potrebbe ripetere lo stesso esercizio anche per metodi che ricevono DUE o più parametri polimorfi (ad es. 2+ forme)
 - ✓ Il numero di varianti di metodo in overload si moltiplica ad ogni parametro aggiuntivo per ogni possibile tipo contemplato nella gerarchia
 - ✓ Sino a risolverli tutti tramite una sequenza di chiamate polimorfe, una per ciascun parametro
- Il risultato sarebbe significativamente più lungo e meno leggibile, ma non per questo concettualmente più complicato
- Nella pratica, anche grazie a questi meccanismi alternativi, la mancanza in Java della risoluzione dei metodi sovraccarichi sulla base del tipo dinamico di uno o più parametri non viene percepita come una significativa carenza