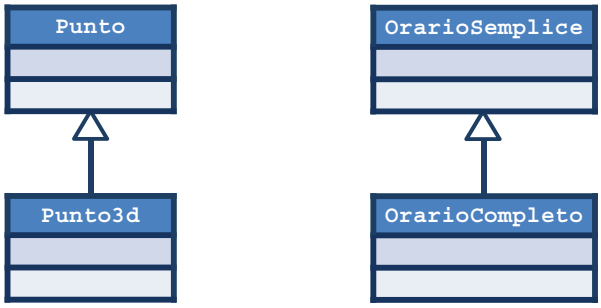
OOP

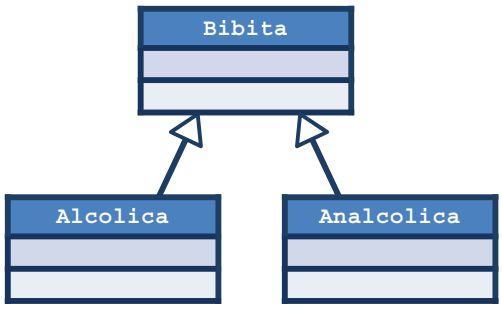
Ereditarietà tra più classi

EREDITARIETÀ TRA DUE CLASSI

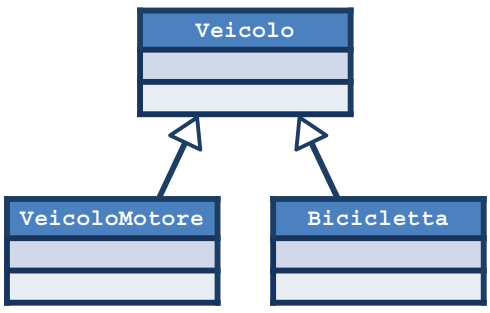


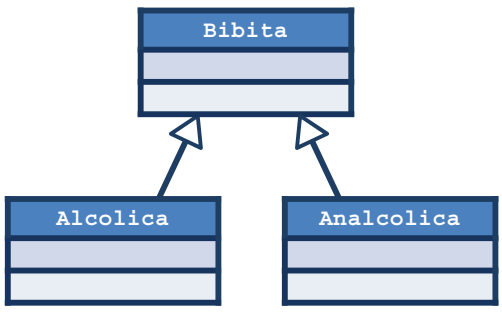
EREDITARIETÀ TRA PIÙ CLASSI

Una situazione banale
si verifica quando da una superclasse
estendiamo più di una sottoclasse

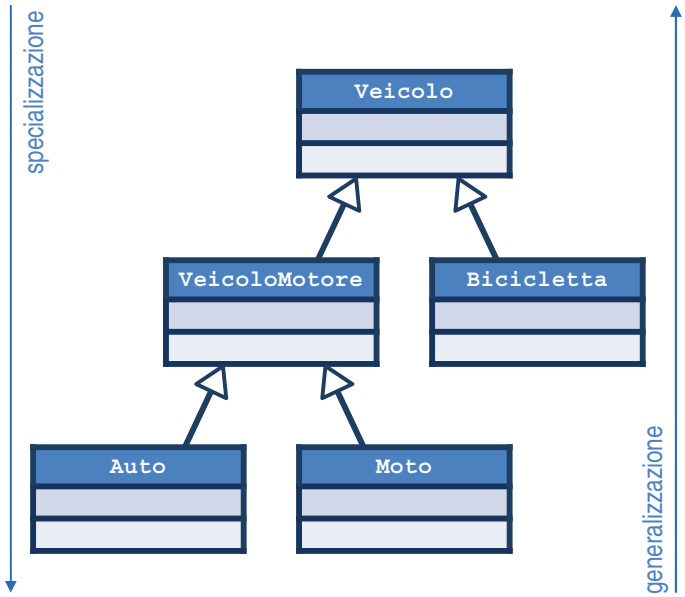


EREDITARIETÀ TRA PIÙ CLASSI





EREDITARIETÀ TRA PIÙ CLASSI



Una situazione già più interessante si verifica quando una classe è contemporaneamente sottoclasse di qualcuno e superclasse per qualcun altro.

Questa è una condizione molto diffusa e definisce una **gerarchia di classi**

Visitando una gerarchia in senso ascendente operiamo un processo di **generalizzazione**; in senso discendente una **specializzazione**

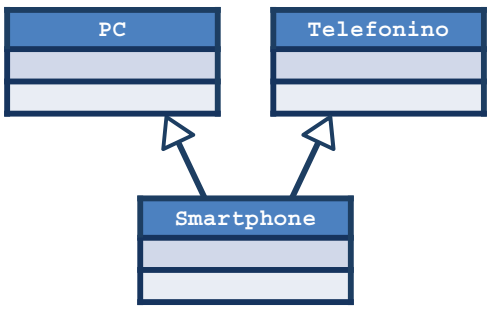
ALESSANDRO URSOMANDO

EREDITARIETÀ TRA PIÙ CLASSI

Una situazione molto particolare invece si verifica quando una classe ha due (o più) superclassi

In questo caso si parla di **ereditarietà multipla**

Java non consente (se non con un sotterfugio) **l'ereditarietà multipla**



ALESSANDRO URSOMANDO

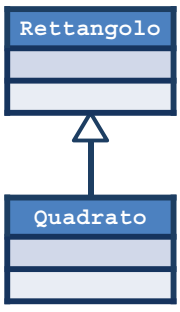
OOP

Ereditarietà VS appartenenza

EREDITARIETÀ VS APPARTENENZA

Anche a livelli molto alti,
l'ereditarietà può essere confusa con l'appartenenza,
ma queste sono due condizioni molto diverse tra loro.

Il trucco sta nel capire se la relazione è di tipo **is-a** o **has-a**



Un quadrato è un rettangolo
con una condizione particolare
(entrambi i lati uguali)
quindi sono in relazione **is-a**.

EREDITARIETÀ VS APPARTENENZA

Anche a livelli molto alti,
l'ereditarietà può essere confusa con l'appartenenza,
ma queste sono due condizioni molto diverse tra loro.

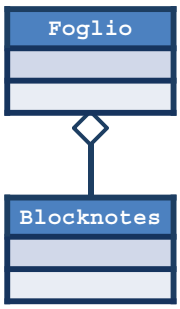
Il trucco sta nel capire se la relazione è di tipo **is-a** o **has-a**

Un cilindro non è un cerchio a cui abbiamo aggiunto qualcosa: un cilindro è caratterizzato da due basi a forma di cerchio e da un'altezza.
Una classe Cilindro dunque avrà due attributi: uno di tipo intero (l'altezza) e uno di tipo Cerchio (che rappresenta entrambe le basi del cilindro): **has-a**

EREDITARIETÀ VS APPARTENENZA

Anche a livelli molto alti,
l'ereditarietà può essere confusa con l'appartenenza,
ma queste sono due condizioni molto diverse tra loro.

Il trucco sta nel capire se la relazione è di tipo **is-a** o **has-a**

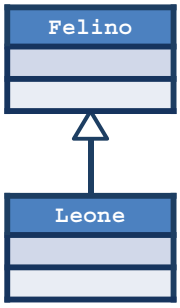


Un foglio è caratterizzato da dimensioni, grammatura, colore, ecc.
Un blocknotes ha un foglio di un certo tipo come copertina iniziale, un foglio di un altro tipo come copertina finale, una serie di fogli di un altro tipo in mezzo: **has-a**.

EREDITARIETÀ VS APPARTENENZA

Anche a livelli molto alti,
l'ereditarietà può essere confusa con l'appartenenza,
ma queste sono due condizioni molto diverse tra loro.

Il trucco sta nel capire se la relazione è di tipo **is-a** o **has-a**

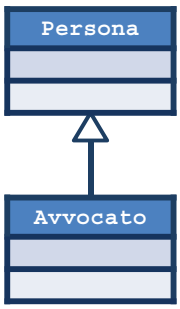


Un esemplare di leone sarà sempre e comunque anche un felino e quindi quanto definito per la classe Felino vale anche per la classe Leone: **is-a**.

EREDITARIETÀ VS APPARTENENZA

Anche a livelli molto alti,
l'ereditarietà può essere confusa con l'appartenenza,
ma queste sono due condizioni molto diverse tra loro.

Il trucco sta nel capire se la relazione è di tipo **is-a** o **has-a**

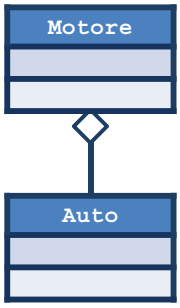


Per quanto le esperienze di qualcuno potrebbero indurre a pensare il contrario tutti gli avvocati in fondo non sono altro che persone: **is-a**.

EREDITARIETÀ VS APPARTENENZA

Anche a livelli molto alti,
l'ereditarietà può essere confusa con l'appartenenza,
ma queste sono due condizioni molto diverse tra loro.

Il trucco sta nel capire se la relazione è di tipo **is-a** o **has-a**



Sebbene il motore sia la cosa più importante di un'autovettura essa non può essere vista come un motore a cui si aggiunge qualcosa.
Un'auto ha un volante, quattro ruote e un motore. E un'infinità di altre cose: **has-a**.

OOP

Overloading e overriding

OVERLOADING

Per overloading si intende
la presenza di più metodi con lo stesso nome ma con firme diverse.

Per differenziare le firme
si tiene conto solo dell'elenco dei parametri e non del tipo del metodo

Quindi fare overloding di un metodo significa scrivere un metodo dello stesso tipo,
con lo stesso nome ma con parametri diversi per posizione, per tipo o per quantità.

```
public double somma(int a, int b)           {...}
public double somma(int a, int b, int c)    {...}
public double somma(float a, float b)       {...}
public double somma(int a, float b)         {...}
public double somma(float a, int b)         {...}
```

Non possiamo avere due metodi la cui firma differisce solo per il tipo restituito.

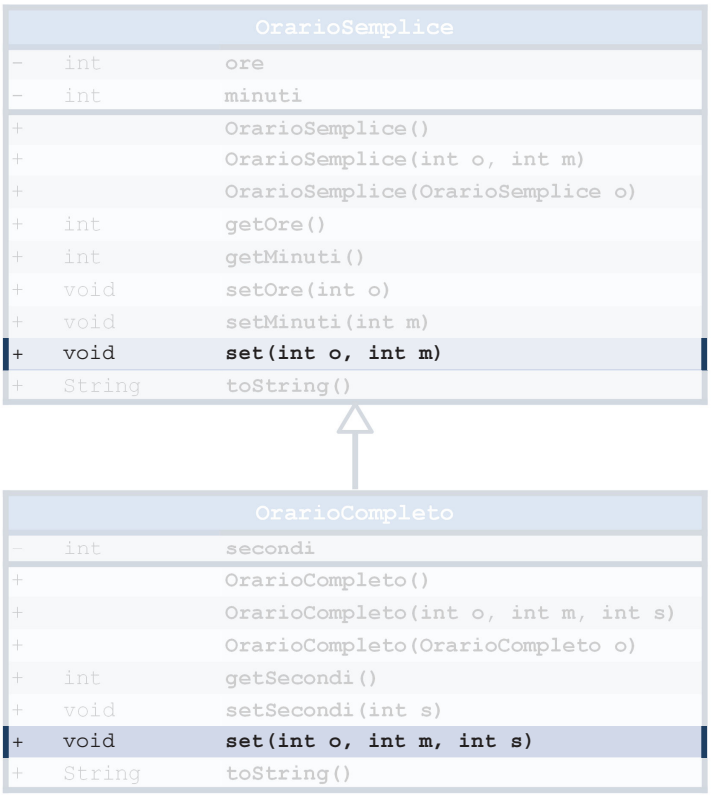
```
public double somma(int a, int b)           {...}
public int somma(int a, int b)              {...}
```


OVERLOADING

OrarioSemplice		
-	int	ore
-	int	minuti
+		OrarioSemplice()
+		OrarioSemplice(int o, int m)
+		OrarioSemplice(OrarioSemplice o)
+	int	getOre()
+	int	getMinuti()
+	void	setOre(int o)
+	void	setMinuti(int m)
+	void	set(int o, int m)
+	String	toString()

Ogni volta che scriviamo
una classe facciamo overloading
del costruttore

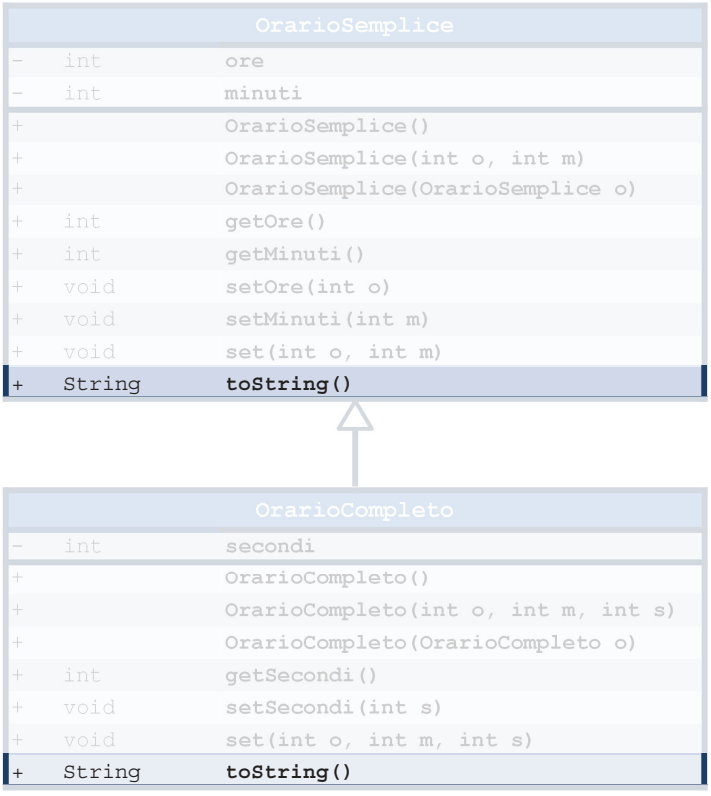
OVERLOADING



Ogni volta che scriviamo una classe facciamo overloading del costruttore

L'overloading si può fare anche in ereditarietà

OVERRIDING



Per overriding si intende la ridefinizione di un metodo in una sottoclasse.

OOP

Richiamare metodi delle superclassi

RICHIAMARE I METODI DELLE SUPERCLASSI

Nell'overriding è facile che si desideri invocare il metodo che si sta ridefinendo

Questa è la toString() di OrarioSemplice

```
public String toString() {
  return ore+":"+((minuti<10)?"0":"")+minuti;
}
```

Questa è la toString() di OrarioCompleto

```
public String toString() {
  return ore+":"+((minuti<10)?"0":"")+minuti+":"+((secondi<10)?"0":"")+secondi;
}
```

Sarebbe comodo poter usare la toString() della superclasse

```
public String toString() {
  return super.toString()+":"+((secondi<10)?"0":"")+secondi;
}
```