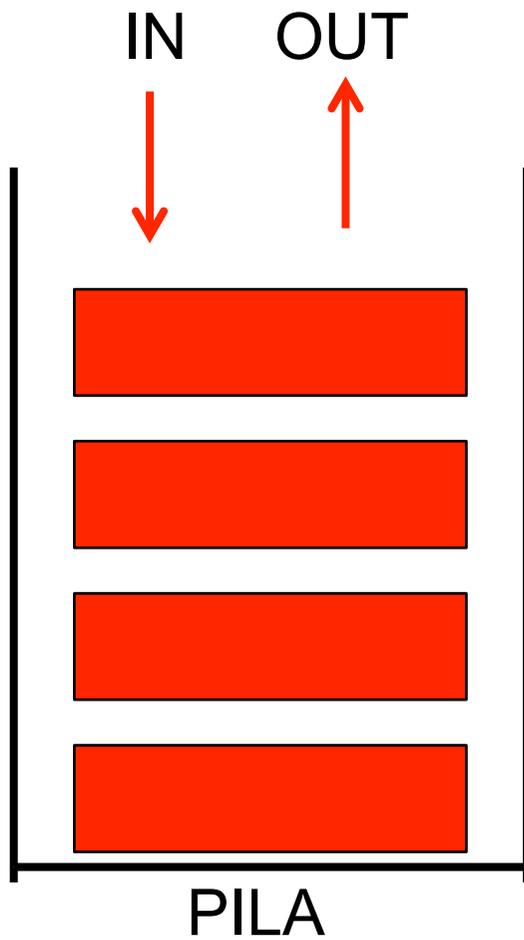
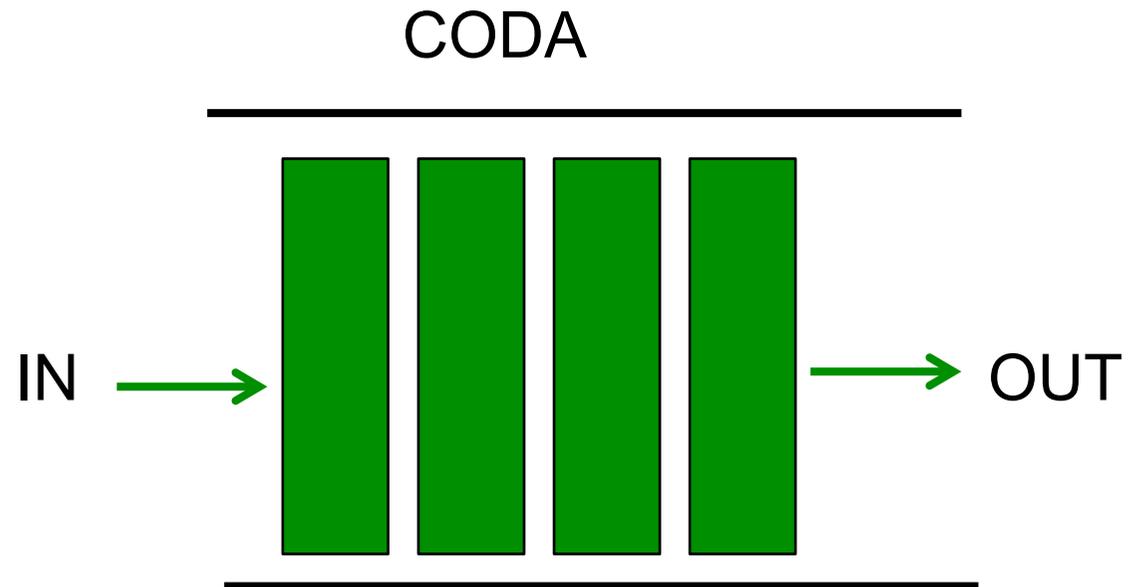


Ereditarietà e polimorfismo

Pila e coda



Sono molto simili... cosa cambia nel codice?



Trasformare la pila in coda

```
int estrai() {  
    assert(marker>0) : "Invalid marker";  
    int retval = contenuto[0];  
    for(int k=1; k<marker; k++)  
        contenuto[k-1] = contenuto[k];  
    marker--;  
    return retval;  
}
```

coda

Il resto del codice è identico:
possiamo evitare di riscriverlo?

```
int estrai() {  
    assert(marker>0) : "Invalid marker";  
    return contenuto[--marker];  
}
```

pila

Trasformare la Pila in Coda

```
package strutture;
public class Coda extends Pila {
    int estrai() {
        assert(marker>0) : "Invalid marker";
        int retval=contenuto[0];
        for (int k=1; k<marker; k++ )
            contenuto[k-1]=contenuto[k];
        marker--;
        return retval;
    }
}
```

La gerarchia di ereditarietà

- Tutte le classi di un sistema OO sono legate in una **gerarchia di ereditarietà**
- Definita mediante la parola chiave **extends**
- La sottoclasse eredita tutti gli attributi ed i metodi della superclasse
 - Es., **Coda** eredita tutti gli attributi e metodi di **Pila**
- **Java supporta solo ereditarietà semplice**
 - In altre parole: una classe non può ereditare da più di una superclasse

La classe **Object**

- Se la clausola **extends** non è specificata nella definizione di una classe, questa *implicitamente* estende la classe **Object**
- ... che dunque è la *radice* della gerarchia di ereditarietà
- **Object** fornisce alcuni metodi importanti, che useremo nel seguito:
 - **public boolean equals(Object) ;**
 - **protected void finalize() ;**
 - **public String toString() ;**

Ereditarietà

Le estensioni possono essere:

strutturali

(aggiunta di variabili di istanza)

e/o

comportamentali

(aggiunta di nuovi metodi)

e/o

modifica di metodi esistenti)

Overriding

- Una sottoclasse può aggiungere nuovi attributi e metodi ma anche ...
- ... **ridefinire i metodi della sua superclasse**

```
class AutomobileElettrica extends Automobile {
    boolean batterieCariche;
    void ricarica() { batterieCariche = true; }
    void accendi() {
        if(batterieCariche) accesa = true;
        else accesa = false;
    }
}
```

La pseudo variabile **super**

- All'interno di un metodo che ne ridefinisce uno della superclasse diretta, ci si può riferire al metodo ridefinito tramite la notazione:

super.<nome metodo>(<lista par. attuali>)

```
class AutomobileElettrica extends Automobile {
    ...
    void accendi() {
        if(batterieCariche) super.accendi();
        else System.out.println("Batterie scariche");
    }
}
```

Ereditarietà e costruttori

- I costruttori **non** vengono ereditati
- All'interno di un costruttore è possibile richiamare il costruttore della superclasse tramite la notazione:
super (<lista di par. attuali>)
posta come **prima istruzione** del costruttore
- Se il programmatore non chiama esplicitamente un costruttore della superclasse, il compilatore inserisce automaticamente il codice che ne invoca il costruttore di default (senza parametri)

Trasformare la **Pila** in **Coda**

```
public static void main(String args[]) {  
    int dim = 5;  
    Coda s = new Coda(dim) ;  
    for(int k=0; k<2*dim; k++)  
        s.inserisci(k) ;  
    for(int k=0; k<3*dim; k++)  
        System.out.println(s.estrai() ) ;  
}
```

Nella definizione di **Coda**:

```
Coda(int size) {  
    super(size) ;  
}
```

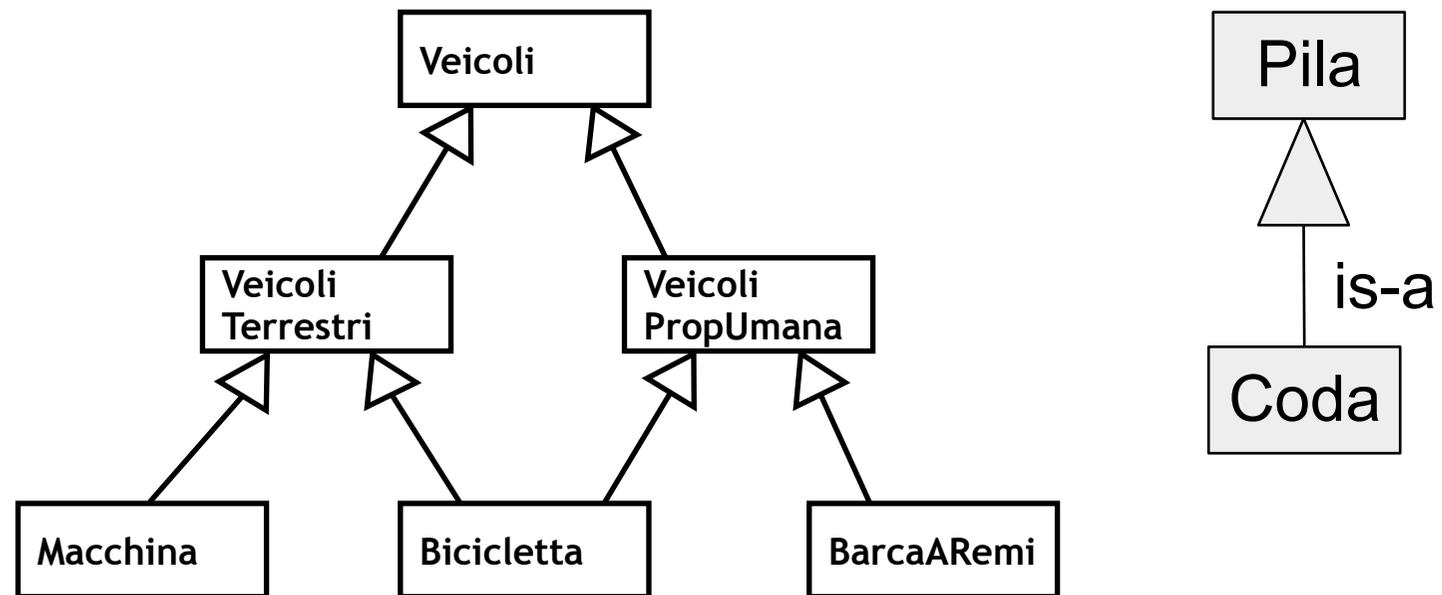
Ereditarietà (Generalizzazione) - UML

Esplicita strutture e comportamenti comuni

È possibile:

aggiungere nuovi attributi alle classi

ridefinire/modificare metodi esistenti



Nelle assegnazioni ...

L'entità a sinistra deve sempre essere compatibile con ciò che le viene assegnato!

DEVE VALERE LA IS-A

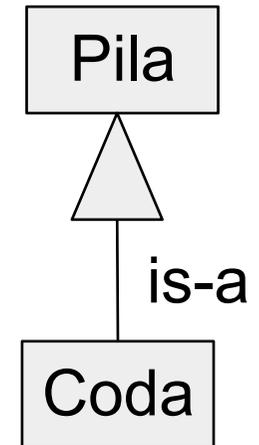
`Pila x = new Pila(3); // banale`

e anche

`Pila x = new Coda(3); // si perché a dx ho una Pila`

ma non

`Coda x = new Pila(3); // no perché a dx NON ho una Coda!`



Subclassing & overriding

```
public class Point {
    public int x = 0;
    public int y = 0;
    Point(int x,int y){
        this.x = x;
        this.y = y;
    }
    public String toString(){
        return "(" + x + "," + y + ")";
    }
    public static void main(String a[]){
        Point p = new Point(5,3);
        System.out.println(p);
    }
}
```

Output:
(5, 3)

Subclassing & overriding

```
public class NamedPoint extends Point {
    String name;
    public NamedPoint(int x,int y,String name) {
        super(x,y); //prima istruzione!
        this.name = name;
    }
    public String toString(){ //Overriding
        return name + " (" + x + "," + y + ")";
    }
    public static void main(String a[]){
        NamedPoint p = new NamedPoint(5,3,"A");
        System.out.println(p);
    }
}
```

Output:
A (5, 3)

Subclassing & overriding

```
public class NamedPoint extends Point {
    String name;
    public NamedPoint(int x,int y,String name) {
        super(x,y); //prima istruzione!
        this.name = name;
    }
    public String toString(){ //Overriding
        return name + super.toString();
    }
    public static void main(String a[]){
        NamedPoint p = new NamedPoint(5,3,"A");
        System.out.println(p);
    }
}
```

Output:
A (5, 3)

Overloading di metodi

- All'interno di una stessa classe possono esservi più metodi con lo **stesso nome** purché si distinguano per numero e/o tipo dei parametri
- **Attenzione:** Il tipo del valore di ritorno non basta a distinguere due metodi

Firma di `int f(int x, int y, String z) :`
`f(int, int, String)`

Overloading: un esempio

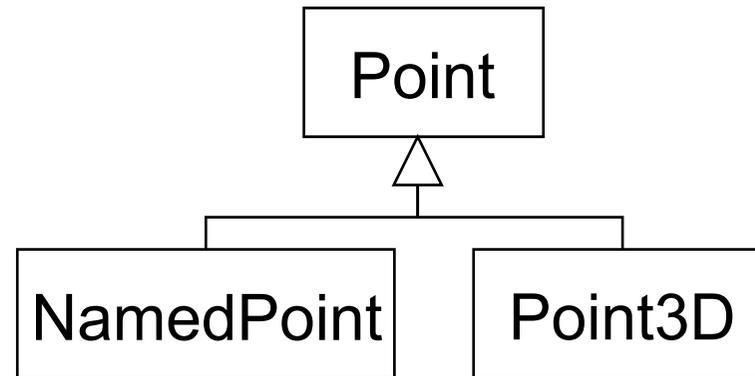
```
class C {  
    int f() {...}  
  
    int f(int x) {...}  
    // corretto  
  
    void f(int x) {...}  
    // errato  
}
```

```
C ref = new C();  
  
ref.f();  
// distinguibile  
  
ref.f(5);  
// ???
```

Overloading vs. overriding

- **Overloading**: funzioni con uguale nome e diversa firma possono coesistere, es.
`move(int dx, int dy)`
`move(int dx, int dy, int dz)`
- **Overriding**: ridefinizione di una funzione in una sottoclasse (mantenendo immutata la firma definita nella superclasse)
es., `estrai()` in **Coda** e **Pila**

Esercizio



a) Scrivere un metodo `move(int dx, int dy)` in **Point**

b) Estendere **Point** a **Point3D**
aggiungendo una coordinata **z**, e fornendo
un metodo `move(int dx, int dy, int dz)`

c) E' possibile definire la classe **NamedPoint3D**? Come?