Qual è l'output?

```
public class F {
   int x = 3;
   F(int x) {
      f(x);
      f();
      System.out.println(x);
   }
   void f() { x++; System.out.print(x); }
   void f(int x) {
      this.x++; x--;
      System.out.print(x);
   }
   public static void main(String arg[]) {
      F x = new F(3);
   }
}
```

Qual è l'output?

```
public class C {
   public static int x;
   C(int s) { x = s; }
   void f() { System.out.print(x); }
   public static void main(String a[]){
      C b = new C(2);
      C c = new C(4);
      b.f();
      c.f();
   }
}
```

In quale linea è l'errore?

```
1. public class Test {
2.    public static void main(String[] args) {
3.       C obj = new C();
4.       int a = C.m1();
5.       int b = obj.m3();
6.    }
7. }
8. class C {
9.    static int x = 5;
10.   int y = 3;
11.   static public int m1() { return y + m2(); }
12.   static private int m2() { return x; }
13.   public int m3() { return m1(); }
14. }
```

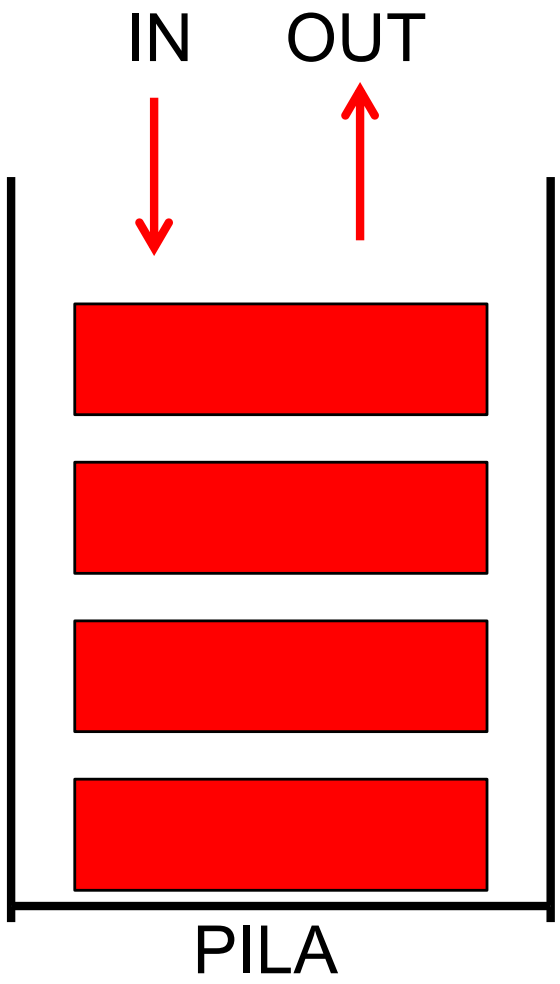
Ereditarietà e polimorfismo

Marco Patrignani

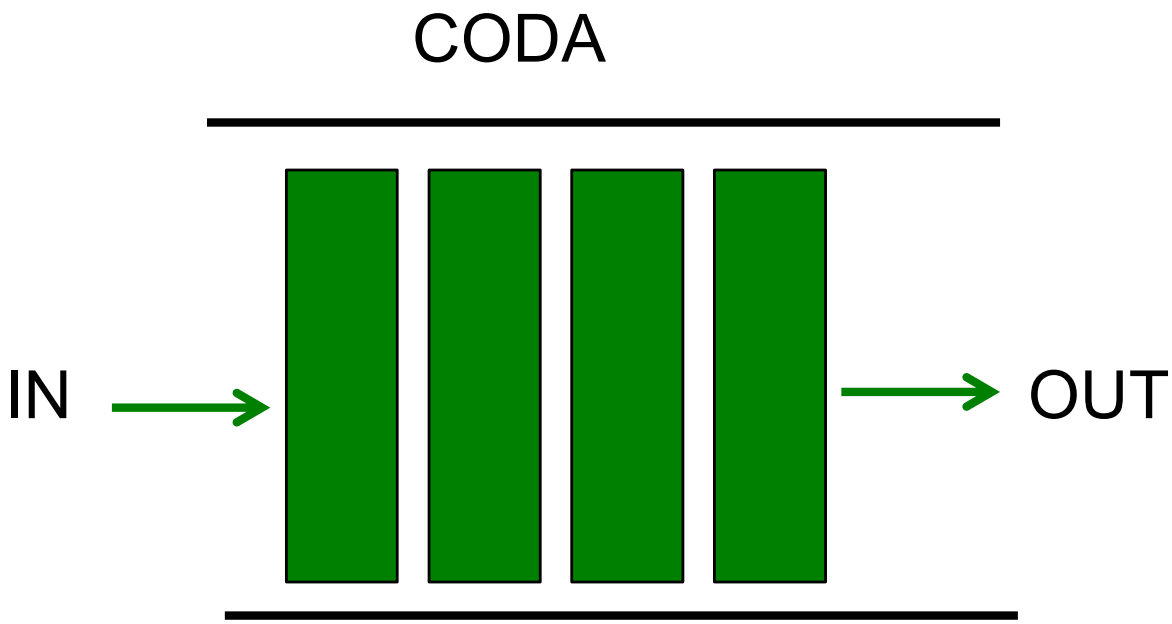
mailto: marco.patrignani@unitn.it

(basato sulle slides di Picco, Ronchetti, Marchese)

Pila e coda



Sono molto simili... cosa cambia nel codice?



Trasformare la pila in coda

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

pila

Trasformare la Pila in Coda

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

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);
}
```

Information hiding in Java

- La visibilità degli attributi e metodi di una classe **`C`** può essere:
  - **`public`**
    - visibili a tutti
  - **`protected`**
    - visibili alle classi dichiarate nel package di **`C`**
    - visibili alle sottoclassi di **`C`**, anche se definite in altro package
  - **«package» (nessun modificatore specificato)**
    - visibili solo alle classi dichiarate nel package di **`C`**
  - **`private`**
    - visibili solo all'interno della classe in cui sono definiti
    - non visibili nelle sottoclassi

Modificatori di visibilità

	visibilità			
modificatore	**classe**	**package**	**sottoclasse**	**mondo**
`private`	Y	N	N	N
“package”	Y	Y	N	N
`protected`	Y	Y	Y	N
`public`	Y	Y	Y	Y

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

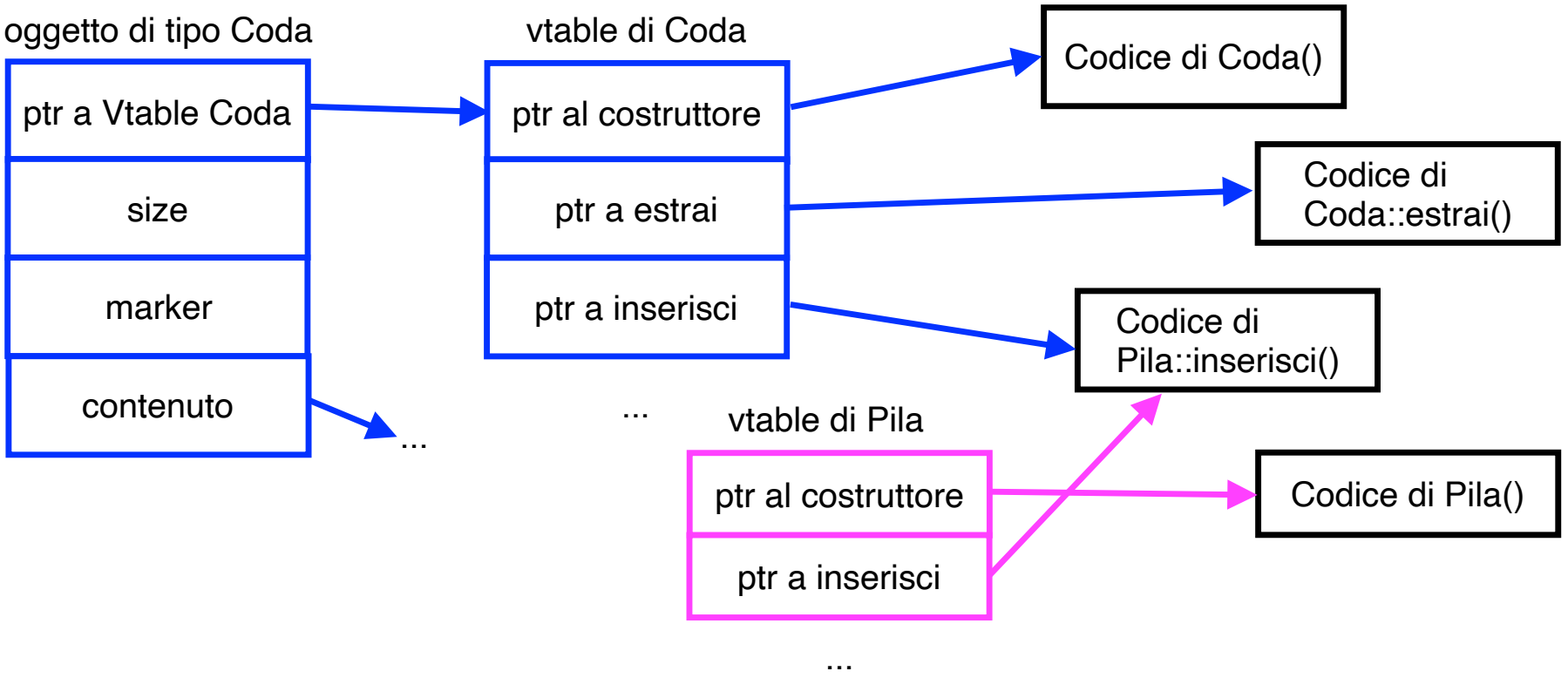
La classe `Object`

- 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();`

Memory Layout di oggetti di tipo Pila e Coda

ogni oggetto ha:
- i suoi campi
- un puntatore alla vtable della classe

ogni classe ha:
- una vtable che punta al corpo dei metodi



Ereditarietà

La 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");
  }
}
```

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(){ // overriding (da Object)
    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! (costruttore)
    this.name = name;
  }
  public String toString(){ // overriding (da Point)
    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);
      this.name = name;
   }
   public String toString(){
      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)

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 specifica nessun costruttore, il compilatore automaticamente inserisce un'invocazione al costruttore di default (senza parametri) della superclasse
  - Se questo non esiste, si verifica un errore in compilazione
  - **`Object`** definisce un costruttore di default …

Costruttori di default

```
class A {
  int x;
  int y;
  A(int x, int y){
    this.x = x;
    this.y = y;
  }
}
class B extends A {
}
```

```
class A {
  int x;
  int y;
  A(int x, int y){
    this.x = x;
    this.y = y;
  }
  A(){
    x = 0; y = 0;
  }
}
class B extends A {
}
```

```
class A {
  int x;
  int y;
  A(int x, int y){
    this.x = x;
    this.y = y;
  }
}
class B extends A {
  B(int x, int y){
    super(x,y);
  }
}
```

Costruttori di default

```
class A {
  int x;
  int y;
  A(int x, int y){
    this.x = x;
    this.y = y;
  }
}
class B extends A {
}
```

Non compila: la superclasse non ha un costruttore di default

```
class A {
  int x;
  int y;
  A(int x, int y){
    this.x = x;
    this.y = y;
  }
  A(){
    x = 0; y = 0;
  }
}
class B extends A {
}
```

Compila:
la superclasse ha un costruttore di default, che viene invocato automaticamente alla creazione di B

```
class A {
  int x;
  int y;
  A(int x, int y){
    this.x = x;
    this.y = y;
  }
}
class B extends A {
  B(int x, int y){
    super(x,y);
  }
}
```

Compila:
la sottoclasse richiama esplicitamente il costruttore con parametri della superclasse

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

Overloading: un esempio

```
class C {
   int f() {...}

   int f(int x) {...}
   // corretto

   void f(int x) {...}
   // errore
} // (in compilazione)
```

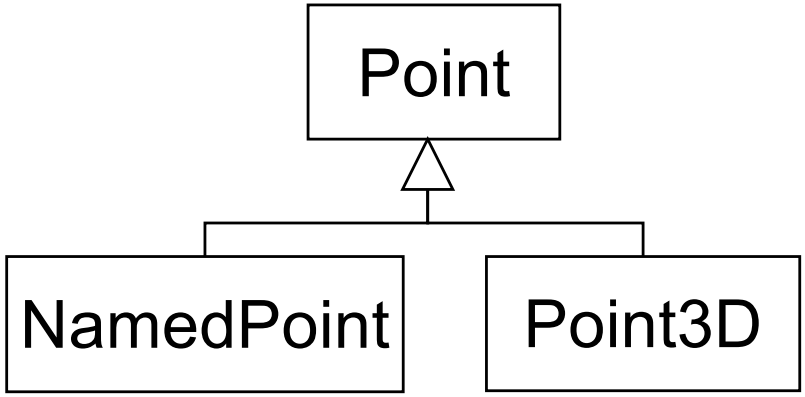
```
C ref = new C();



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)`**