Programmazione a Oggetti
Modulo B

Lezione 4

Dott. Alessandro Roncato

11/02/2014

Riassunto

Pattern Null Object

Pattern Singleton

Esempi su come ridurre dipendenza

Polimorfismo

Come gestire comportamenti diversi basati sul tipo

Esempio: come gestire maniere diverse per stornare un'operazione bancaria.

Proviamo prima a utilizzare un approccio NON orientato agli oggetti

Esempio

```
public class Conto {
  double saldo;
  …
 public boolean stornaOperazione(Operazione o){
   switch (o.getTipo()) {
     case PRELIEVO:
        return false;
     case DEPOSITO:
        saldo-=o.getImpoto();
        return true;
     case BONIFICO:
        String iban = o.getIbanDestinatario();
        if (banca.annullaAccreditoAltraBanca(iban,o.getCRO())){
          saldo+=o.getImporto();
          return true;
        }
        else return false;
     case ...
```

Svantaggi

- Nello stesso oggetto devono esserci dei campi/metodi che non hanno interesse per quell'oggetto (es. `getIbanDestinatario`)
- Una modifica al codice per la gestione di un particolare tipo di operazione si ripercuote anche agli altri tipi.
- Bassa coesione

Esempio

```
public class Conto {
  double saldo;
  …
 public boolean stornaOperazione(Operazione o){
   switch (o.getTipo()) {
     case PRELIEVO:
        return false;
     case DEPOSITO:
        saldo-=o.getImpoto();
     case GIROCONTO:
        saldo+=o.getImporto();
        o.getAltroConto().saldo-=o.getImporto();
        return true;
     case BONIFICO:
        String iban = o.getIbanDestinatario();
        if (banca.annullaAccreditoAltraBanca(iban,o.getCRO())){
          saldo+=o.getImporto();
          return true;
        }
        else return false;
```

Svantaggi

- Per aggiungere la gestione del `Giroconto` abbiamo messo mano al codice del metodo `stornaOperazione` per tutti i tipi di operazione
- Togliendo per sbaglio il `break` alla fine del caso `Deposito`, abbiamo introdotto un bug su una cosa che funzionava.
- Polimorfismo permette approccio incrementale...

Svantaggi

- Ripetizione codice (tanti metodi con `switch`...)
- Approccio procedurale: invece di pensare al comportamento del singolo oggetto, penso a un singolo comportamento di tutti gli oggetti.
- I metodi molto lunghi e poco riutilizzabili (supponiamo di avere una nuova banca con operazioni diverse, dobbiamo modificare in più classi)

Polimorfismo

Ogni comportamento diverso è gestito da un oggetto di tipo diverso.

Quindi avremo tante sottoclassi, una per ogni tipo diverso: `Prelievo`, `Bonifico`, `Deposito`.

Ogni classe implementa solo i metodi con comportamento diverso dalla classe Operazione

Esempio

```
public class Operazione {
  Conto mandante;
  Calendar data;
  double importo;
  …
  public boolean storna(){
      return false;
  }

  public String getDescrizione(){
      Return ””;
  }
  public Conto getConto(){
    return conto;
  }
}
```

Esempio

```
public class Prelievo extends Operazione {
  Ricevuta ricevuta;


 //storno non serve sia implementato, va bene
 //quello ereditato da Operazione

 public String getDescrizione(){
    return ”prelevati ”+o.getImporto()+” euro”;
  }
}
```

Esempio

```
public class Bonifico extends Operazione {
  String iban;
  String cro;


 public boolean storna(){
    if (conto.getBanca().annullaAccreditoAltraBanca(iban,cro)){
        conto.saldo+=getImporto();
        return true;
      }
      else return false;
  }
 public String getDescrizione(){
    return ...;
  }
}
```

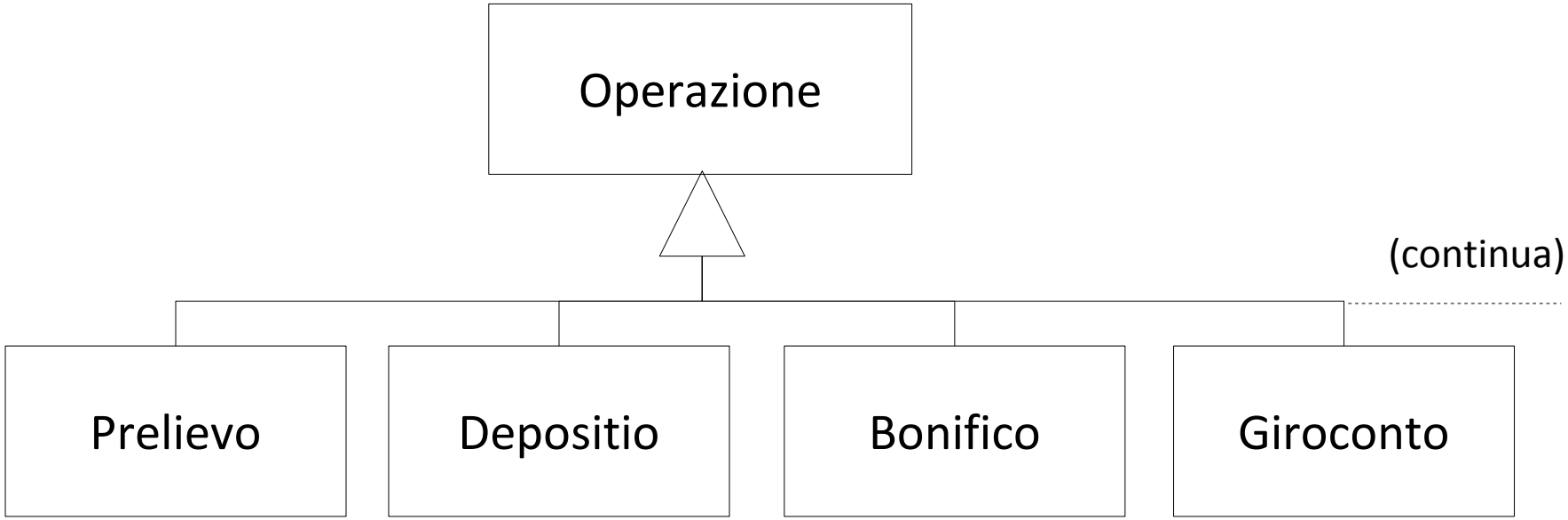
Esercizio

Scrivere le classi Deposito e Giroconto

Pro e contro

- PRO e CONTRO: Una classe per ogni tipo (aumentano i file ma la contempo sappiamo dove trovare il codice)
- PRO: oggetti più semplici e coesi
- PRO: metodi più semplici
- PRO: Estensibile
- CONTRO: molte classi => applicare solo se il comportamento è diverso

Nel Diagramma



Altre applicazioni Poly.

- I ruoli Cassiere e Direttore sono tutti dello stesso tipo quindi il loro comportamento deve essere lo stesso.
- Il Conto potrebbe avere tipi diversi: ContoDepositio, ContoImpresa, ContoInvestimento etc.
- Altri...

Classi o Interfacce?

- Talvolta parliamo di classi e interfacce come fossero la stessa cosa
- Talvolta chiamiamo interfaccia la parte pubblica di una classe
- Una classe ha qualcosa di più di una interfaccia

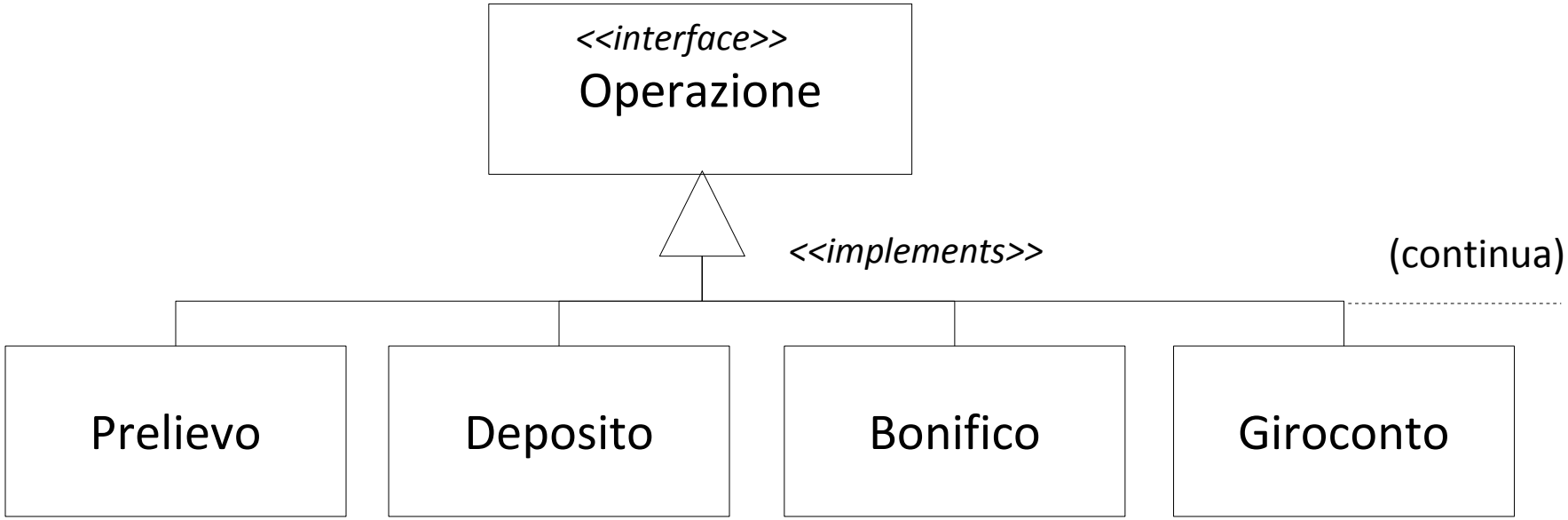
Cosa ha in più la classe

- Istanziabile (= creare oggetti)
- Istruzioni (implementazione metodi)
- Parte privata
- Variabili d'istanza

Interfaccia

- Deve essere implementata da una classe
- I metodi sono astratti
- E' tutto pubblico
- Non ha variabili d'istanza

Nel Diagramma



Meglio interfaccia se ...

- Il comportamento in comune tra le sottoclassi è minimo
- Una interfaccia può essere facilmente promossa a classe, mentre il viceversa no (dove metto implementazione metodi e attributi, serve una classe astratta)
- Nel dubbio usare interfaccia
- Ereditarietà multipla in Java

Meglio interfaccia se ...

- Fin'ora abbiamo visto come fare una classe e fare in modo che chi la usa la usi come diciamo noi
- Con un'interfaccia possiamo fare in modo di obbligare gli altri a fare le loro classi come vogliamo noi!

Pilotare futuro

Supponiamo di volere implementare una libreria per gestire la persistenza dei dati nel DB. Dato che non sappiamo come saranno fatte tutti gli oggetti che dovranno essere memorizzati nel DB come facciamo a scrivere la libreria in modo che possa essere usata in tutti i casi?

Pilotare Futuro

Scriviamo la nostra libreria assumendo che gli oggetti di cui gestire la persistenza implementino l'interfaccia `Persistente` e definiamo solo l'interfaccia stessa.

- Chi vorrà usare la libreria dovra' implementare l'interfaccia negli oggetti da salvare nel DB.
- Per fare in modo che la libreria sia facile da usare, l'interfaccia dovrà essere semplice (pochi metodi)

Interfaccia Persistenza

```
public interface Persistente {
    String [] getProperties();
    void setProperties(String[] props);
}

public class DBManager {
    …
    public static DBManager getSingleton(){ … }

    public boolean load(int key, Persistente p){
      …
      String[] props = …; //from DB
      p.setProperties(props);
    }
    public boolean store(int key, Persistente p){
     String[] props = p.getProperties();
      … // to DB;
    }
    …
```

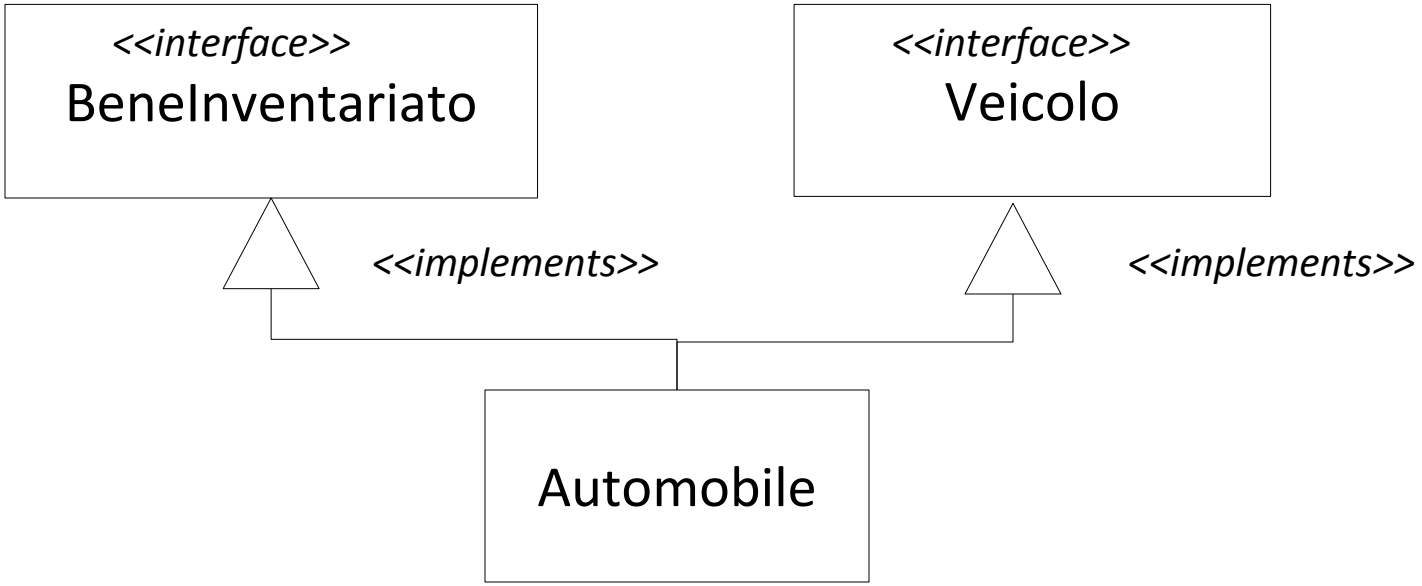
Nel futuro…

```
public class Conto implements Persistente {
   String cliente;
   int numero;
   double saldo;
   …
   public String [] getProperties(){
    String[] res = new String[3];
    res[0]=cliente;
    res[1]=””+numero;
    res[2]=””+saldo;
    Return res;
   }
   public void setProperties(String[] props){
    cliente= res[0];
    numero = Integer.parseInt(res[1]);
    saldo = Double.parseDouble(res[2]);
   }
```

Ereditarietà multipla

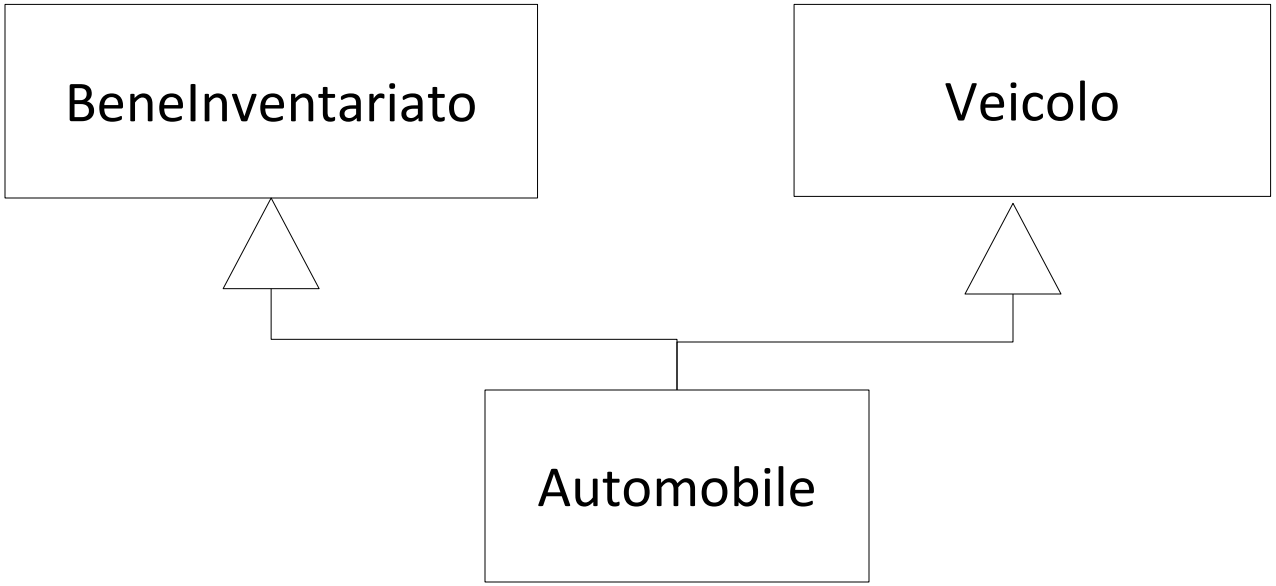
- In Java una classe può estendere una **sola** altra classe, mentre può implementare tante interfacce
- In altri linguaggio possiamo avere l'estensione multipla (ovvero la possibilità che una classe estenda **tante** altre classi)
- In ogni caso l'ereditarietà multipla è fonte di problemi

Interfacce multiple



OK

Ereditarietà multipla



NO!

Codice (se si potesse)

```
public class B1 {
   public methodA(){...};
   public methodB(){...};
}

public class B2 {
    public method1(){...}
    public method2(){...}
    …
}

public class A extends B1 B2 {

   …
}
```

Uso classe

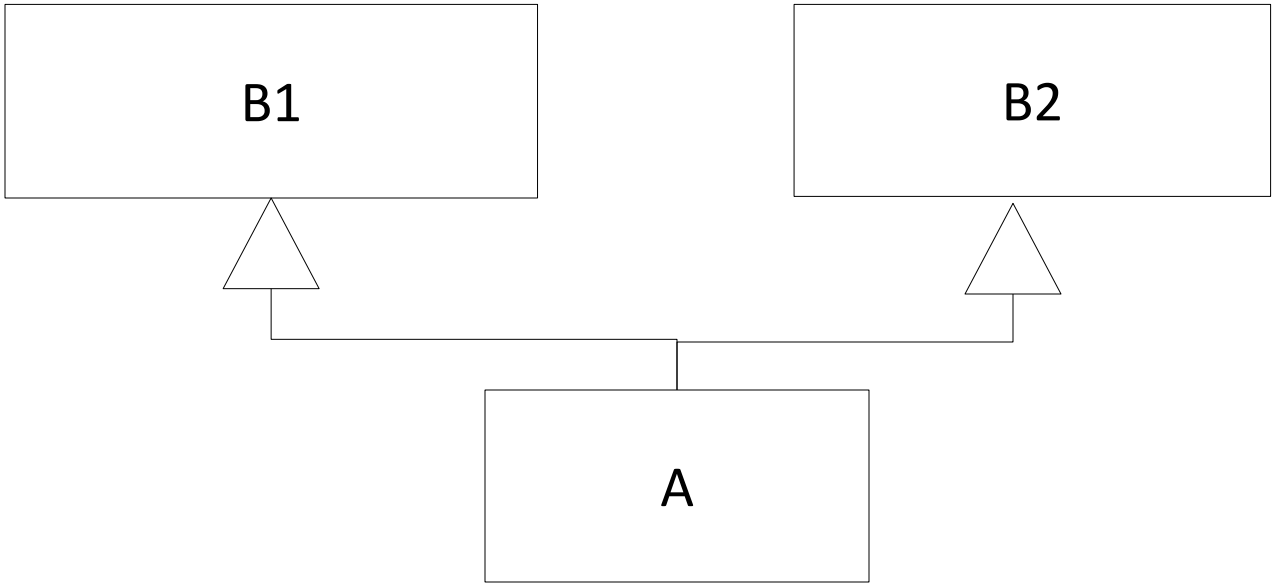
```
public class XXX {
    A a

  public void perEsempio(){
    a.methodA();
    …
    a.method1();
  }
   …
}
```

Ereditarietà multipla

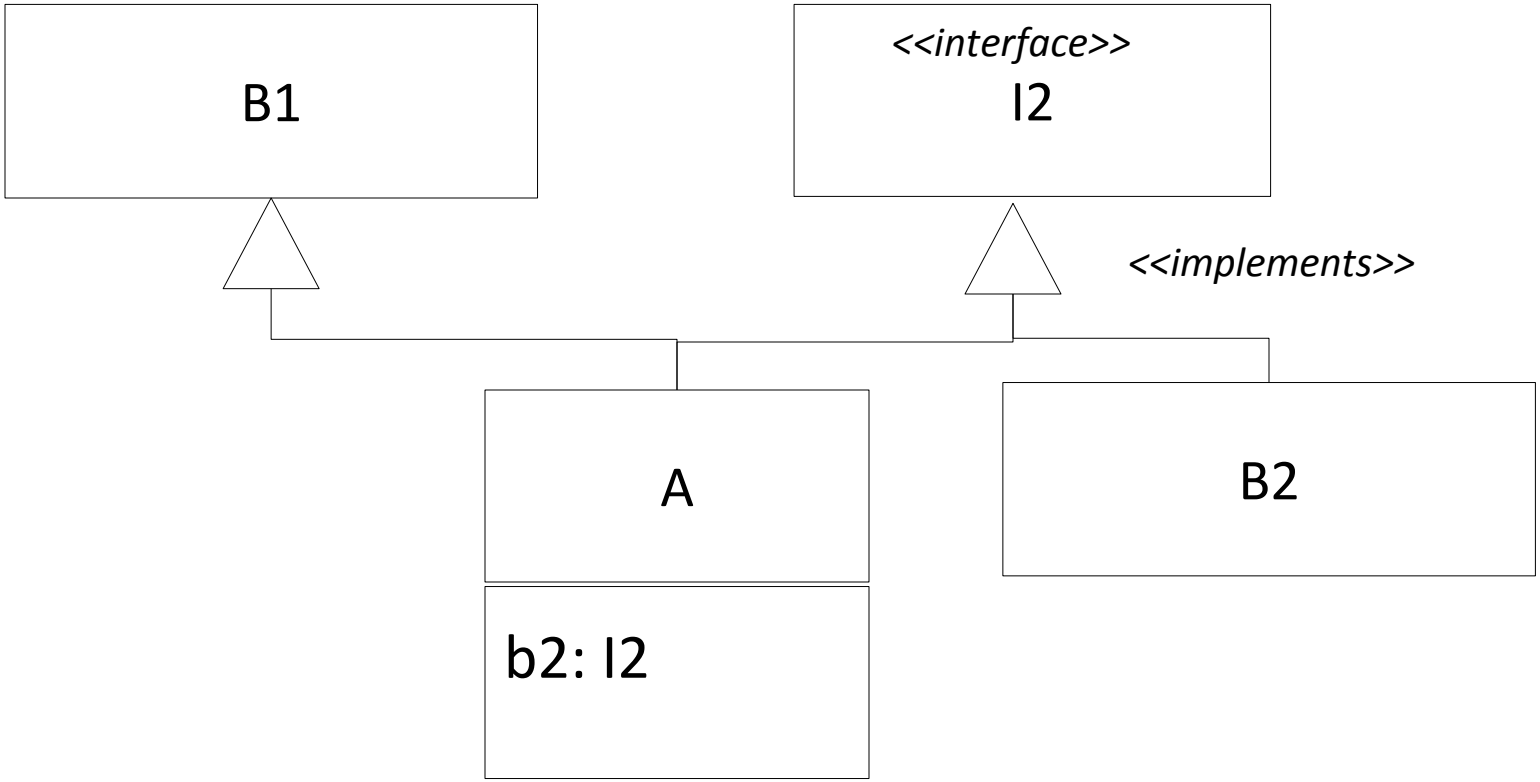
Come ottenere l'ereditarietà multipla in Java?

- Supponiamo che A erediti dalle classi B1, B2, B3 etc.
- A estende una delle classi Bi (meglio la classe Bi con più metodi pubblici)
- Le altre classi vengono “incorporate” nella classe A
- Servono delle interfacce I1, I2, etc

Ereditarietà multipla



Ereditarietà singola +...



Codice

```
public interface I2 {
    method1();
    method2();
    …
}

public class B2 implements I2 {
     public method1(){...}
     public method2(){...}
     …
}

public class A extends B1 implements I2 {

    I2 b2 = new B2();
    public method1(){b2.method1();}
    public method2(){b2.method2();}
    …
}
```

Interfacce

- In ogni caso il problema dell'ereditarietà multipla è secondario rispetto alla scelta dell'uso delle interfacce invece che delle classi
- Le interfacce rappresentano il contratto tra chi implementa l'oggetto e chi lo usa
- Chi usa la classe è interessato solo alla parte pubblica

domande