UNIVERSITÀ PARTHENOPE
DiST
Dipartimento di Scienze e Tecnologie

Programmazione 3
e
Laboratorio di Programmazione 3

Ereditarietà e interfacce

Proff. Angelo Ciaramella – Emanuel Di Nardo

Ereditarietà

- Il termine ereditarietà è inteso in senso *darwiniano*
- Nel mondo reale noi classifichiamo tutto con classi e sottoclassi
  - un cane è un animale, un aereo è un veicolo, la chitarra è uno strumento musicale

Parola chiave extends

```
public class Libro {
public String titolo;
public String autore;
public String editore;
public int numeroPagine;
public int prezzo;
. . .
}
```

superclasse

erediterà tutti i campi pubblici della classe che estende

```
public class LibroSuJava extends Libro {
public final String ARGOMENTO_TRATTATO = "Java";
. . .
}
```

sottoclasse

Riassumendo…

```
class NomeSottoclasse extends NomeSuperclasse
{
      metodi
      variabili istanza
}
```


SavingsAccount

- Costruiamo una sottoclasse `SavingsAccount` per descrivere un conto bancario che garantisce un *tasso fisso sui depositi*

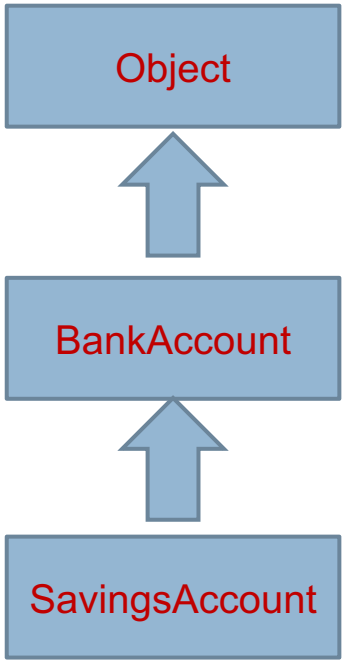


Diagramma di ereditarietà

`addInterest()`

```
interest = Balance * interestRate / 100
```

Metodo aggiuntivo

Codice di riferimento

`SavingsAccount.java`

Esercizio

Modificare `BankAccount` o `SavingAccounts` inserendo un array di conti corrente. Tenere in considerazione la possibilità di aggiungere una nuova classe `Bank` e che tipo di relazione è necessaria con tra gli elementi.

Calcolare:

- Bilancio totale della banca;
- Contare il numero di depositi totali della banca;
- Stampare il numero di conto corrente usando sempre 10 cifre;
  - Ed. accountNumber = 3 -> 0000000003;
- Tassare i conti corrente con più di 5000 € (una tantum);
- Convertire la valuta di un singolo conto corrente (`switch`);

Variabili istanza e metodi

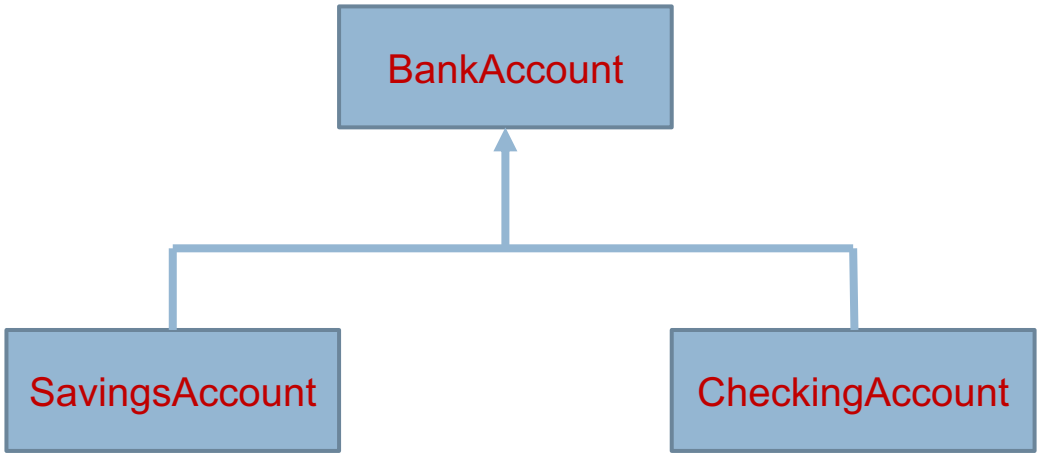
- Per definire i metodi di una sottoclasse, esistono tre possibilità
  - sovrascrivere o ridefinire metodi della superclasse
    - Se specifichiamo un metodo con la stessa firma viene sovrascritto quello della superclasse
  - possiamo ereditare metodi della superclasse
  - possiamo definire nuovi metodi

Variabili istanza e metodi

- Per definire variabili istanza in una sottoclasse
  - possiamo ereditare variabili della superclasse
    - tutte le variabili istanza della superclasse sono ereditate automaticamente
    - possiamo definire nuove variabili visibili solo nelle sottoclassi
    - *non possiamo sovrascrivere* variabili istanza
- Una sottoclasse non ha accesso ai campi privati della sua superclasse
- Per modificare una variabile privata di una superclasse
  - bisogna usare un metodo pubblico della stessa

CheckingAccount

- Costruiamo una sottoclasse `CheckingAccount` che addebita commissioni ogni 3 transizioni



Gerarchia di ereditarietà

Codice di riferimento

`CheckingAccount.java`

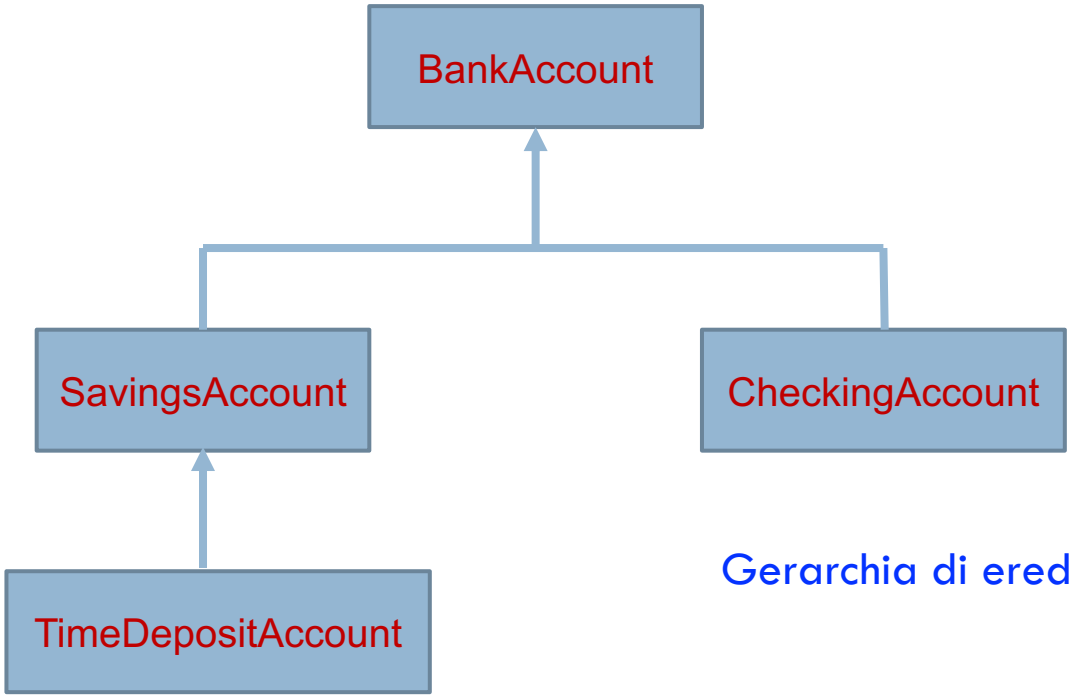
CheckingAccount

- Ha un metodo aggiuntivo `deductFees`
- Ha una ulteriore variabile istanza `transactionCount`
- Sovrascrive i metodi `deposit` e `withdraw`

```
FREE_TRANSACTIONS = 3
TRANSACTION_FEE = 2.0

fees = TRANSACTION_FEE *
(transactionCount - FREE_TRANSACTIONS)
```

Calcolo commissione

TimeDepositAccount



Gerarchia di ereditarietà

Codice di riferimento

`TimeDepositAccount.java`

Modificatore final

- Usando il modificatore `final`
  - una variabile dichiarata `final` diviene una costante
  - un metodo dichiarato `final` non può essere riscritto in una sottoclasse (non è possibile applicare l'override)
    - si eredita così come è stato dichiarato nella superclasse
  - una classe dichiarata `final` non può essere estesa
    - Per esempio le classi `String` e la classe `Math` sono classi dichiarate `final`
  - le variabili locali e i parametri locali di metodi non saranno modificabili localmente

Relazione "is a"

- Fondamentale domanda
  - un oggetto della candidata sottoclasse "è un" oggetto della candidata superclasse?

- In Java non esiste propriamente quella che viene definita "ereditarietà multipla"
  - In Java 8 è possibile applicarla alle interfacce

Costruttori

- L'ereditarietà non è applicabile ai costruttori
  - *per il loro nome*

- Un qualsiasi costruttore (anche quello di default), come prima istruzione, invoca sempre un costruttore della superclasse

```
super(parametri)
```

- Per invocare un metodo della superclasse

```
super.nomeMetodo(parametri)
```


Costruttori

```
public class Libro {
// . . .
public Libro (String titolo, String autore) {
this(titolo); // Chiamata al secondo costruttore
setAutore(autore);
}
public Libro (String titolo) {
this.titolo = titolo;
}
}

public class LibroSuJava extends Libro {
public LibroSuJava (String titolo) {
super(titolo);

//super è un'istruzione implicita se non fornita esplicitamente

//deve essere la prima istruzione di un costruttore e non potrà
essere inserita all'interno di un metodo che non sia un
costruttore
}
...
}
```


Incapsulamento

```
public class Persona {

private String nome, cognome;
public String toString(){
return nome + " " + cognome;
}
. . .
//accessor e mutator methods (set e get)
}
public class Cliente extends Persona {
private String indirizzo, telefono;
public String toString() {
return super.toString()+ "\n"+
indirizzo + "\nTel:" + telefono;
}
. . .
//accessor e mutator methods (set e get)
}
```

Ridefinito per avere nuovi campi

Chiamata alla superclasse

La classe Object

- La classe `Object` astrae il concetto di *oggetto generico*
  - Appartiene al package `java.lang`
  - È la superclasse di ogni classe
  - Tutte le classi estenderanno `Object` ed ereditano i suoi membri

Conversioni

- I riferimenti a sottoclasse possono essere convertiti a riferimenti a superclasse

```
SavingsAccount collegeFund = new SavingsAccount(10);
BankAccount anAccount = collegeFund;
Object anObject = CollegeFund;
```

- Le tre variabili si riferiscono allo stesso oggetto
  - Non è possibile però avere tutte le informazioni
    - e.g., in `anAccount` non è possibile invocare `addInterest`

Codice di riferimento

`AccountTest.java`

Superclasse universale Object

- I metodi più utili della classe `Object` sono
  - `String toString()`
    - restituisce una rappresentazione dell'oggetto in forma di stringa
  - `boolean equals(Object otherObject)`
    - verifica se un oggetto è uguale ad un altro
  - `Object clone()`
    - crea una copia completa dell'oggetto

Override

- Per essere applicati alle nostre classi i metodi devono essere *sovrascritti*

```
public class BankAccount {
...
public String toString() {
return getClass().getName()+ "[balance=" + balance + "]";
}
```

```
public boolean equals (Object otherObject) {

Coin other = (Coin) otherObject
return name.equals(other.name) && value == other.value
}
```

Metodo per ottenere un oggetto di tipo classe per gestire l'ereditarietà

Codice di riferimento

Coin.java

Override

- Per essere applicati alle nostre classi i metodi devono essere *sovrascritti*

```
public class BankAccount {
...
public Object clone() {

BankAccount cloned = BankAccount();
cloned.balance = balance;
return cloned;
}
```

- Ulteriori aspetti saranno approfonditi successivamente

Controllo di accesso

- Java fornisce quattro livelli per il controllo di accesso
  - `public`
    - metodi di *tutte le classi*
  - `private`
    - una sottoclasse *non* può accedere a campi private ereditati dalla superclasse
  - `protected`
    - possono accedervi *tutte le sottoclassi* e tutte le classi dello stesso pacchetto
  - `accesso di pacchetto` (predefinita)
    - possono accedervi i metodi di *classi nello stesso pacchetto*
    - per le variabili viola l'incapsulamento

Metodi e classi astratte

- Un metodo astratto è un metodo di cui *non viene specificata l'implementazione*
  - obbligando i programmatori di sottoclassi a fornirne una
  - Non è possibile costruire oggetti di classi aventi metodi astratti

```
public abstract void method_name();
```

- Vantaggio
  - "obbliga" le sue sottoclassi ad implementare un comportamento

- Il modificatore `abstract`
  - per le classi potrebbe essere considerato l'opposto del modificatore final
    - classe `final`, non può essere estesa, mentre una classe dichiarata `abstract` deve essere estesa

Classi astratte

- Una classe dichiarata astratta *non può essere istanziata*
  - sono utilizzate per poter dichiarare caratteristiche comuni fra classi di una determinata gerarchia
- Una classe astratta può avere
  - campi non statici e *final*
  - metodi non pubblici (*protected* e *private*)
  - un costruttore
  - può contenere o meno metodi astratti
    - una classe che contiene metodi astratti deve essere dichiarata necessariamente astratta

Classi astratte

```
public abstract class Pittore {
//. . .
public abstract void dipingiQuadro();
//. . .
}
```

- Questa classe ha senso se inserita in un sistema in cui l'oggetto `Pittore` può essere considerato troppo generico per definire un nuovo tipo di dato da istanziare
  - obbligare i programmatori a *creare sottoclassi*

Classe strumento

```
public abstract class Strumento { //Classe astratta
public String nome;
public String prezzo;
public abstract void suonaFaDiesis(); /*Ogni strumento
suona in modo diverso! Impossibile definire questo
metodo!*/
//. . .
}
```

```
public class Chitarra extends Strumento { // Classe
concreta
public void suonaFaDiesis() { // Override (riscrittura) del
metodo
//Implementazione del metodo per la chitarra.
}
//. . .
}
```


Classe Flauto

```
public abstract class StrumentoAFiato extends Strumento {
//Classe di
//nuovo astratta che estende Strumento
//metodo suonaFaDiesis ereditato ancora astratto e
//non riscritto perché troppo generico!
//. . .
}

public class Flauto extends StrumentoAFiato {
// Classe concreta che
//estende StrumentoAFiato
public void suonaFaDiesis() {
//Implementazione del metodo per il flauto.
}
//. . .
}
```


Classi astratte

- Quando estendiamo (o deriviamo) da una classe astratta
  - la classe derivata deve fornire un'implementazione per tutti i metodi astratti
  - se non forniamo un'implementazione dei metodi astratti la sottoclasse deve essere dichiarata astratta

Interfacce

- Un'interfaccia è un'evoluzione del concetto di classe astratta

- *Sino a Java 7* un'interfaccia per definizione poteva possedere solo
  - metodi dichiarati implicitamente `public` e `abstract`
  - variabili dichiarate implicitamente `public`, `static` e `final`

- Le interfacce si devono scrivere all'interno di file che hanno esattamente lo stesso nome dell'interfaccia che definiscono

- Un'interfaccia *non si può istanziare* (non è una classe)
  - per essere utilizzata ha bisogno *di essere in qualche modo estesa*

Interfacce

```
//Definizione di interfaccia

public interface Nomeinterfaccia
      {
       firma dei metodi
      }


//Implementazione di interfaccia

public class NomeClasse implements Nomeinterfaccia,
Nomeinterfaccia, ...
      {
        metodi
        variabili istanza
      }
```


Interfacce

- Caratteristiche
  - tutti i metodi sono astratti
    - hanno un nome, un elenco di parametri, un tipo restituito ma non hanno implementazione
  - tutti i metodi sono pubblici
  - non ha variabili istanza

- Per realizzare un'interfaccia una classe deve fornire tutti i metodi richiesti dall'interfaccia

Soluzioni riutilizzabili

```
public class DataSet
{
...
   public void add(BankAccount x)
   {
      sum += x.getBalance();
      ...
   }

   public BankAccount getMaximum()
   {
      return maximum;
   }

  private double sum;
  private BanckAccount maximum;
  private int count;
}
```

Classe modificata per i conti bancari

Soluzioni riutilizzabili

```
public class DataSet
{
...
   public void add(Coin x)
   {
      sum += x.getValue();
      ...
   }

   public Coin getMaximum()
   {
      return maximum;
   }

  private double sum;
  private Coin maximum;
  private int count;
}
```

Classe modificata per le monete

Soluzioni riutilizzabili

```
public interface Measurable
{
 double getMeasure();
}
```

Interfaccia con il metodo per generalizzare la classe `DataSet`

Codice di riferimento

`DataSetTest.java, DataSet.java, BankAccount.java, Coin.java, Mearurable.java, Purse.java`

Interfacce

- Solo un'altra *interfaccia può estendere un'altra interfaccia*
  - è possibile creare *gerarchie costituite da sole interfacce*
- Abbiamo bisogno che le *classi ereditino dalle interfacce*
  - parola chiave `implements`
- Implementare un'interfaccia
  - implementare un numero indefinito di interfacce simulando di fatto l'ereditarietà multipla
- Una classe può implementare un'interfaccia e un'interfaccia può estendere un'altra interfaccia

Interfacce

- Prima dell'avvento di Java 8
  - Se una classe ereditava metodi da un'interfaccia, allora doveva "*onorare il contratto*" e ridefinire i metodi astratti ereditati

- Con Java 8 le interfacce hanno acquisito maggiore potenza
  - è possibile *definire all'interno delle interfacce anche metodi statici*
    - non sembrava in linea con la concezione di interfaccia come contratto da implementare
    - i metodi statici di un'interfaccia non vengono ereditati

Implementare un'interfaccia

```
public interface Saluto {
public static final String CIAO = "Ciao";
public static final String BUONGIORNO = "Buongiorno";
. . .
public abstract void saluta();
}
```

```
public class SalutoImpl implements Saluto {
public void saluta() {
System.out.println(CIAO);
}
}
```

Interfacce fino al Java 7

Metodi statici di un'interfaccia

```
public interface StaticMethodInterface {
static void metodoStatico() {
System.out.println("Metodo Statico Chiamato!");
}
}


// implements non fa ereditare i metodi statici
// bisogna chiamare metodi statici direttamente dalle
// interfacce

public class TestStaticMethodInterface {
public static void main(String args[]) {
StaticMethodInterface.metodoStatico();
}
}
```

Interfacce con Java 8

Metodi di default

- Altra novità di Java 8
  - possibilità di dichiarare metodi concreti all'interno delle interfacce (**metodi di default**, *default methods*)

```
public interface Solista {
default void eseguiAssolo() {
//Scala maggiore in DO
System.out.println("DO RE MI FA SOL LA SI");
}
}

// la classe eredita eseguiAssolo()
public class Musicista implements Solista {
}
```


Ereditarietà multipla

- In Java 8 esiste una manifestazione semplificata di ereditarietà multipla
  - una classe può implementare più interfacce
  - possono ereditare dalle interfacce solo la loro parte funzionale (i metodi) e non i dati (a parte le costanti statiche che un'interfaccia può dichiarare)

Metodi di default

```
public interface Lettore {
default void leggi(Libro libro) {
System.out.println("Sto leggendo: " + libro.getTitolo() + "
di " + libro.getAutore());
}
}

public interface Programmatore {
default void programma(String linguaggio) {
System.out.println("Sto programmando in " + linguaggio);
}
}

public class ChiStaLeggendo implements Lettore,
Programmatore {
}
```


Diamond Problem

```
public class TestEreditarietaMultipla {
public static void main(String args[]) {
ChiStaLeggendo tu = new ChiStaLeggendo();
Libro java8 = new Libro("Manuale di Java 8", "Claudio De
Sio Cesari");
tu.programma("Java");
tu.leggi(java8);
}
}
```


Diamond Problem

- Nel caso una classe erediti da due interfacce due metodi di `default` con la stessa firma (nome più lista di attributi) è necessario che la classe ridefinisca il metodo

Diamond Problem

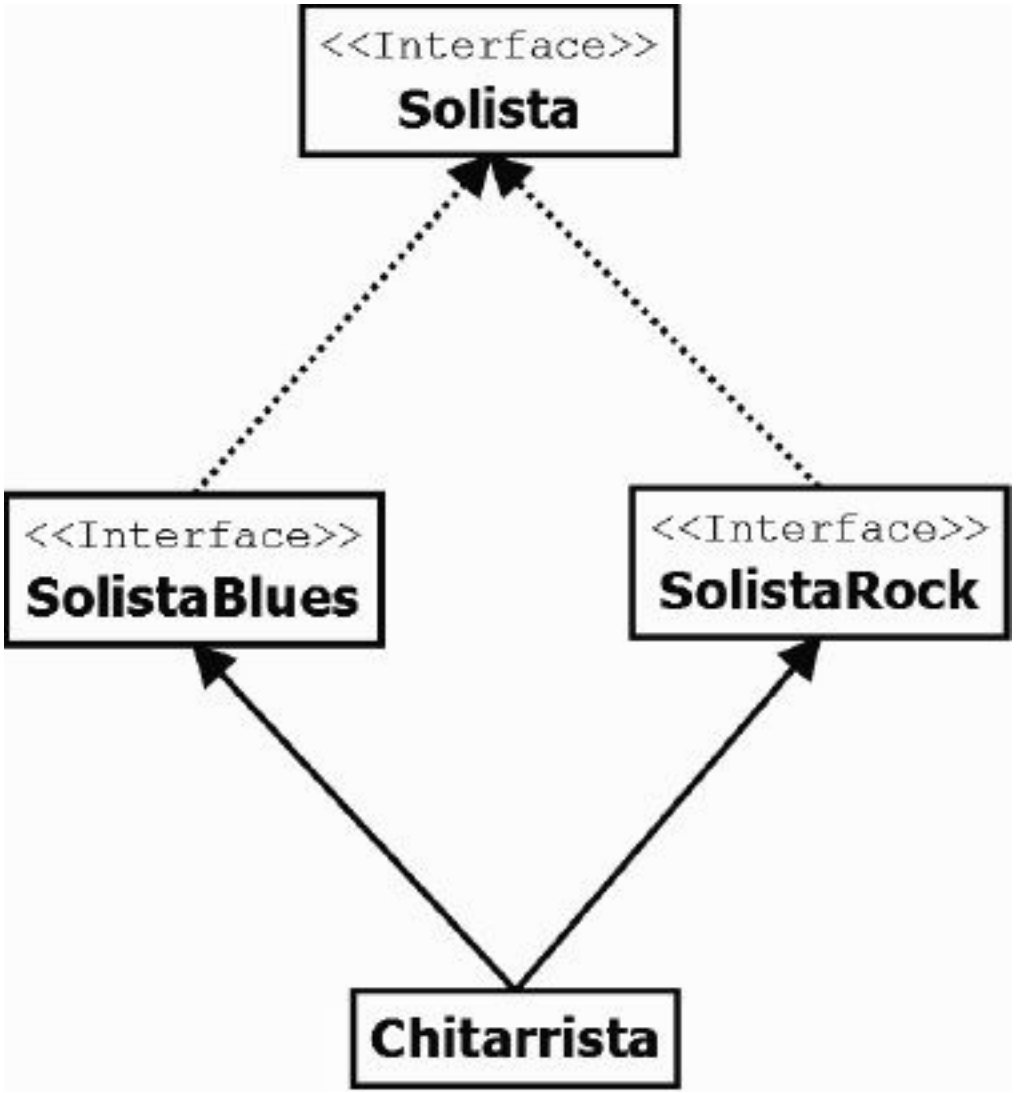
```
public interface Solista {
default void eseguiAssolo() {
//Scala maggiore in DO
System.out.println("DO RE MI FA SOL LA SI");
}
}

public interface SolistaBlues extends Solista {
default void eseguiAssolo() {
//Scala blues in DO
System.out.println("DO MIb FA SOLb SOL SIb DO");
}
}
public interface SolistaRock extends Solista {
default void eseguiAssolo() {
//Scala pentatonica in DO
System.out.println("DO RE MI SOL LA DO");
}
}
```


Diamond Problem

```
public class Chitarrista implements SolistaBlues,
SolistaRock {
}
```

ERRORE DI COMPILAZIONE

Diamond Problem



Diamond Problem descritto con UML

Diamond Problem

- Non è in grado di decidere quale delle due implementazioni ereditate del metodo `eseguiAssolo()` debba essere considerata quella prioritaria

- Soluzione
  - risolvere il conflitto di nomi *ridefinendo il metodo* per specificarne l'implementazione

Diamond Problem

```
public class Chitarrista implements SolistaBlues,
SolistaRock {
public void eseguiAssolo() {
//Scala pentatonica + scala blues in DO
SolistaRock.super.eseguiAssolo();
SolistaBlues.super.eseguiAssolo();
}
}
```


Conflitto di nomi

- Classe che implementa più interfacce
  - conflitto di nomi tra un metodo astratto ereditato dalla prima interfaccia ed un metodo concreto ereditato da un'altra interfaccia
  - ridefinizione del metodo nella classe così come visto nel diamond problem

- Classe che implementa più interfacce
  - conflitto di nomi di soli metodi astratti
  - la soluzione è quella di ridefinire il metodo nella sotto classe, così come visto precedentemente

Conflitto di nomi

- Classe implementa due interfacce dove una estende l'altra
  - nella classe che le implementa c'è conflitto di nomi di metodi ereditati
  - viene ereditata l'implementazione della interfaccia più specifica

- Classe eredita un conflitto di firme tra un metodo ereditato da una classe estesa e un metodo dell'interfaccia implementata
  - viene ereditata l'implementazione della classe, anche se questa fosse astratta
    - "classe vince sempre", "class always win"

Classe astratta vs Interfacce

	Interfacce	Classi astratte
Istanziabile	no	no
Campi	solo `static final`	sì
Costruttore	no	sì
Metodi statici	Java8+	sì
Dichiarazione metodi (virtual)	no	sì
Implementazione metodi	Java8+ (`default`)	sì

Classe astratta vs Interfacce

- Una classe può implementare un numero arbitrario di interfacce, ma può ereditare al più da una classe astratta
- Una classe astratta può avere metodi non astratti. Tutti i metodi di un'interfaccia sono invece astratti
- Una classe astratta può dichiarare variabili istanza che saranno ereditate da tutte le sottoclassi. Un'interfaccia può definire solo campi static final
- Una classe astratta può definire dei costruttori; un'interfaccia no
- Una classe astratta può avere metodi di visibilità *protected*, *private*, o non specificata (*package*); ogni metodo di un'interfaccia dev'essere *public*
- Una classe astratta eredita da *Object*, anche metodi come *clone()* ed *equals()*

Stub dell'interfaccia

- Interfaccia che specifica una collezione di metodi di notifica
  - fornire uno *stub* per quell'interfaccia
    - classe che implementa i metodi tutti vuoti
  - è possibile ereditare dallo stub
    - overriding dei metodi importanti all'applicazione
  - Classe WindowsAdapter di java.awt.event

Stub dell'interfaccia

```
public interface InterfaceVideo {

    static String format="xyz";

    public void run();
    public void start();
    public void stop();
    public void rewind();
}
```

Implementare tutti i metodi

Fornire uno stub della classe e fare un esempio di utilizzo

Stub dell'interfaccia

```
public interface InterfaceVideo {

    static String format="xyz";

    public void run();
    public void start();
    public void stop();
    public void rewind();
}
```

Implementare tutti i metodi

```
public abstract class Video implements BaseVideo {

    @Override
    public void run() {
        // definizione del funzionamento del metodo
    }
}
```

Una classe astratta permette l'implementazione di un solo metodo