EBook Gratuito

APPRENDIMENTO

oop

Free unaffiliated eBook created from
Stack Overflow contributors.

#oop

Sommario

Di

Capitolo 1: Iniziare con oop

Osservazioni

La programmazione orientata agli oggetti (OOP) è un paradigma di programmazione basato sul concetto di "oggetti", che può contenere dati, sotto forma di campi, spesso noti come attributi; e codice, sotto forma di procedure, spesso conosciute come metodi.

Examples

introduzione

OOP - Object Oriented Programming è un paradigma di programmazione ampiamente utilizzato in questi giorni. In OOP, modelliamo i problemi del mondo reale usando Oggetti e i loro comportamenti, al fine di risolverli, a livello di programmazione.

Esistono quattro **concetti OOP** principali

1. Eredità
2. Polimorfismo
3. Astrazione
4. incapsulamento

Questi quattro concetti insieme sono usati per sviluppare programmi in OOP.

Esistono varie lingue che supportano la programmazione orientata agli oggetti. Le lingue più popolari sono

- C ++
- Giava
- C #
- Python (Python non è completamente orientato agli oggetti, ma ha la maggior parte delle sue caratteristiche OOP)

Introduzione OOP

intoduction

La programmazione orientata agli oggetti (principalmente indicata come OOP) è un paradigma di programmazione per risolvere i problemi.
La bellezza di un programma OO (orientato agli oggetti) è che pensiamo al programma come a un gruppo di oggetti che comunicano tra loro, anziché come uno script sequenziale che segue ordini specifici.

Ci sono molti linguaggi di programmazione che supportano OOP, alcuni dei più popolari sono:

- Giava
- C ++
- c #

Python è anche noto per supportare OOP ma manca alcune proprietà.

Terminologia OOP

Il termine più elementare in OOP è una `classe` .
Una classe è fondamentalmente un *oggetto* , che ha uno stato e funziona in base al suo stato.

Un altro termine importante è `un'istanza` .
Pensa a una classe come a un modello usato per creare istanze di se stesso. La classe è un modello e l'istanza (s) è gli oggetti concreti.

Un'istanza creata dalla classe *A* viene solitamente definita come dal 'tipo A', esattamente come il tipo di *5* è *int* e il tipo di *"abcd"* è una *stringa* .

> Un esempio di creazione di un'istanza denominata *insance1* di tipo (classe) *ClassA* :
>
> ### Giava
>
> ```
> ClassA instance1 = new ClassA();
> ```
>
> ### C ++
>
> ```
> ClassA instance1;
> ```
>
> o
>
> ```
> ClassA *instance1 = new ClassA(); # On the heap
> ```
>
> ### Pitone
>
> ```
> instance1 = ClassA()
> ```

Come potete vedere nell'esempio sopra, in tutti i casi è stato menzionato il nome della classe e dopo di esso c'erano parentesi vuote (eccetto per il C ++ dove se sono vuote le parentesi possono essere eliminate). In queste parentesi possiamo passare `arguments` al `costruttore` della nostra classe.

Un costruttore è un metodo di una classe che viene chiamato ogni volta che viene creata un'istanza. Può o prendere argomenti o no. Se il programmatore non specifica alcun costruttore

per una classe che costruiscono, verrà creato un costruttore vuoto (un costruttore che non fa nulla).
Nella maggior parte delle lingue il costruttore è definito come un metodo senza definire il suo tipo di ritorno e con lo stesso nome della classe (esempio in alcune sezioni).

> Un esempio di creazione di un'istanza denominata *b1* di tipo (classe) *ClassB* . Il costruttore di *ClassB* accetta un argomento di tipo *int* :
>
> ### Giava
>
> ```
> ClassA instance1 = new ClassA(5);
> ```
>
> o
>
> ```
> int i = 5;
> ClassA instance1 = new ClassA(i);
> ```
>
> ### C ++
>
> ```
> ClassA instance1(5);
> ```
>
> ### Pitone
>
> ```
> instance1 = ClassA(5)
> ```

Come puoi vedere, il processo di creazione di un'istanza è molto simile al processo di chiamata di una funzione.

Funzioni vs metodi

Entrambe le funzioni e i metodi sono molto simili, ma in Object Oriented Design (OOD) ognuno di essi ha il proprio significato.
Un metodo è un'operazione eseguita su un'istanza di una classe. Il metodo stesso di solito utilizza lo stato dell'istanza per operare.
Nel frattempo, una funzione appartiene a una classe e non a un'istanza specifica. Ciò significa che non utilizza lo stato della classe o qualsiasi dato memorizzato in un'istanza.

D'ora in poi mostreremo i nostri esempi solo in *Java* poiché OOP è molto chiaro in questa lingua, ma gli stessi principi funzionano in qualsiasi altro linguaggio OOP.

In Java, una funzione ha la parola `static` nella sua definizione, in questo modo:

```
// File's name is ClassA
public static int add(int a, int b) {
    return a + b;
```

```
}
```

Ciò significa che puoi chiamarlo da qualsiasi punto dello script.

```
// From the same file
System.out.println(add(3, 5));

// From another file in the same package (or after imported)
System.out.println(ClassA.add(3, 5));
```

Quando chiamiamo la funzione da un altro file, usiamo il nome della classe (in Java questo è anche il nome del file) a cui appartiene, questo dà l'intuizione che la funzione appartiene alla classe e non a nessuna delle sue istanze.

Al contrario, possiamo definire un metodo in *ClassA in questo* modo:

```
// File's name is ClassA
public int subtract(int a, int b){
    return a - b;
}
```

Dopo questa declinazione possiamo chiamare questo metodo in questo modo:

```
ClassA a = new ClassA();
System.out.println(a.subtract(3, 5));
```

Qui abbiamo avuto bisogno di creare un'istanza di *ClassA* per chiamare il suo metodo sottrarre. Si noti che **NON POSSIAMO** fare quanto segue:

```
System.out.println(ClassA.subtract(3, 5));
```

Questa riga produrrà un errore di compilazione lamentando che abbiamo chiamato questo metodo *non statico* senza un'istanza.

Usando lo stato di una classe

Supponiamo di voler implementare nuovamente il nostro metodo di *sottrazione* , ma questa volta vogliamo sempre sottrarre lo stesso numero (per ogni istanza). Possiamo creare la seguente classe:

```
class ClassB {

    private int sub_amount;

    public ClassB(int sub_amount) {
        this.sub_amount = sub_amount;
    }
```

```
    public int subtract(int a) {
        return a - sub_amount;
    }

    public static void main(String[] args) {
        ClassB b = new ClassB(5);
        System.out.println(b.subtract(3)); // Ouput is -2
    }
}
```

Quando eseguiamo questo codice, viene creata una nuova istanza denominata *b* di classe *ClassB* e il suo costruttore viene alimentato con il valore *5* .
Il costruttore ora prende il *sub_amount* dato e lo memorizza come il suo campo privato, chiamato anche *sub_amount* (questa convenzione è molto conosciuta in Java, per denominare gli argomenti come i campi).
Dopodiché, stampiamo sulla console il risultato della chiamata della *sottrazione* del metodo su *b* con il valore di *3* .

Si noti che nell'implementazione della *sottrazione* non la usiamo `this.` come nel costruttore.
In Java, `this` deve essere scritto solo quando esiste un'altra variabile con lo stesso nome definito in quell'ambito. Lo stesso funziona con il `self` di Python.
Quindi, quando usiamo *sub_amount* in sottrazione, facciamo riferimento al campo privato che è diverso per ogni classe.

Un altro esempio da sottolineare.
Cambiamo semplicemente la funzione principale nel codice sopra al seguente:

```
ClassB b1 = new ClassB(1);
ClassB b2 = new ClassB(2);

System.out.println(b1.subtract(10)); // Output is 9
System.out.println(b2.subtract(10)); // Output is 8
```

Come possiamo vedere, *b1* e *b2* sono indipendenti e ognuno ha il proprio *stato* .

Interfacce ed eredità

`Un'interfaccia` è un contratto, esso definisce i metodi di una classe avrà e quindi le sue capacità.
Un'interfaccia non ha un'implementazione, ha solo definito ciò che deve essere fatto.
Un esempio in Java potrebbe essere:

```
interface Printalbe {
    public void print();
}
```

L'interfaccia *Printalbe* definisce un metodo chiamato *print* ma non ne fornisce l'implementazione (piuttosto strano per Java). Ogni classe che si dichiara di `implementing` questa interfaccia deve fornire un'implementazione al metodo draw. Per esempio:

```
class Person implements Printalbe {

    private String name;

    public Person(String name) {
        this.name = name;
    }

    public void print() {
        System.out.println(name);
    }
}
```

Se *Person* si dichiarasse come implementabile *Drawable* ma non fornisse un'implementazione per la *stampa* , ci sarebbe un errore di compilazione e il programma non si compilerebbe.

L'ereditarietà è un termine che indica una classe che *estende* un'altra classe. Ad esempio, diciamo che ora abbiamo una persona che ha un'età. Un modo per implementare una persona del genere sarebbe copiare la classe *Person* e scrivere una nuova classe chiamata *AgedPerson* che ha gli stessi campi e metodi ma ha un'altra proprietà -age.
Questo sarebbe terribile dato che duplichiamo il nostro **intero** codice solo per aggiungere una semplice funzionalità alla nostra classe.
Possiamo utilizzare l'ereditarietà per ereditare da *Persona* e quindi ottenere tutte le sue funzionalità, quindi migliorarle con la nostra nuova funzione, in questo modo:

```
class AgedPerson extends Person {

    private int age;

    public AgedPerson(String name, int age) {
        super(name);
        this.age = age;
    }

    public void print() {
        System.out.println("Name: " + name + ", age:" + age);
    }
}
```

Ci sono alcune novità in corso:

- Abbiamo usato la parola salvata si `extends` per indicare che stiamo ereditando da Person (e anche la sua implementazione in *Printable* , quindi non abbiamo bisogno di dichiarare di nuovo `implementing Printable`).
- Abbiamo usato la parola di salvataggio `super` per chiamare il costruttore di *Person* .
- Abbiamo sostituito il metodo di *stampa* di *Person* con uno nuovo.

Questo sta diventando piuttosto tecnico Java quindi non approfondirò questo argomento. Ma menzionerò che ci sono molti casi estremi che dovrebbero essere appresi sull'ereditarietà e le interfacce prima di iniziare ad usarli. Ad esempio quali metodi e funzioni sono ereditati? Cosa succede ai campi privati / pubblici / protetti quando si eredita da una classe? e così via.

Classe astratta

Una `classe astratta` è un termine piuttosto avanzato in OOP che descrive una combinazione di entrambe le interfacce e l'ereditarietà. Ti permette di scrivere una classe che ha sia metodi implementati che non implementati. In Java ciò avviene usando la parola chiave `abstract` e non lo spiegherò più di un rapido esempio:

```
abstract class AbstractIntStack {

    abstract public void push(int element);

    abstract public void pop();

    abstract public int top();

    final public void replaceTop(int element) {
        pop();
        push(element);
    }
}
```

Nota: la parola chiave `final` afferma che non è possibile sovrascrivere questo metodo quando si eredita da questa classe. Se una classe è dichiarata definitiva, nessuna classe può ereditarla affatto.

Leggi Iniziare con oop online: https://riptutorial.com/it/oop/topic/5081/iniziare-con-oop

Capitolo 2: Astrazione

Examples

Astrazione - Introduzione

L'astrazione è uno dei concetti principali in **Object Oriented Programming (OOP)** . Questo è il processo di nascondere i dettagli di implementazione per gli estranei mentre mostra solo dettagli essenziali. In altre parole, l'astrazione è una tecnica per organizzare la complessità di un programma.

Esistono due tipi fondamentali di astrazione:

1. Controlla l'astrazione

   Questo viene fatto usando le sub-routine e il controllo del flusso. Possiamo chiamare un'altra funzione / metodo / routine (sotto-routine) da una funzione / metodo per eseguire un'attività specifica, in cui tale sotto-routine è astratta.

2. Astrazione dei dati

   Questo viene fatto attraverso varie strutture di dati e le loro implementazioni. Possiamo creare le nostre strutture di dati per archiviare i nostri dati, mantenendo l'implementazione astratta.

In *OOP* usiamo il mix di astrazione di controllo e funzione.

Modificatori di accesso

I modificatori di accesso sono usati per controllare l'accesso ad un oggetto o ad una funzione / metodo. Questa è una parte principale del concetto di *astrazione* .

Diversi linguaggi di programmazione utilizzano diversi modificatori di accesso. Ecco alcuni esempi:

- **Giava**

  Java ha 4 modificatori di accesso.

  1. `private` - Questi attributi sono accessibili solo all'interno della classe.
  2. `protected` : questi attributi sono accessibili da sottoclassi e classi dallo stesso pacchetto.
  3. `package` : questi attributi sono accessibili solo dalle classi all'interno dello stesso pacchetto.
  4. `public` : questi attributi sono accessibili a tutti.

- **C ++**

C ++ ha 3 modificatori di accesso.

1. `private` - Questi attributi sono accessibili solo all'interno della classe.
2. `protected` : è possibile accedere a questi attributi per classi derivate.
3. `public` : questi attributi sono accessibili a tutti.

- **C #**

  C # ha 5 modificatori di accesso

  1. `private` - Questi attributi sono accessibili solo all'interno della classe.
  2. `protected internal` : è possibile accedere a questi attributi dallo stesso assembly e dalle classi derivate.
  3. `protected` : è possibile accedere a questi attributi per classi derivate.
  4. `public internal` - Questi attributi sono accessibili dalle classi all'interno dello stesso assembly.
  5. `public` : questi attributi sono accessibili a tutti.

Leggi Astrazione online: https://riptutorial.com/it/oop/topic/7324/astrazione

Capitolo 3: Classe

Examples

introduzione

Class è il pezzo di codice in cui definiamo gli attributi e / oi comportamenti di un oggetto. È possibile definire variabili, costanti, metodi e costruttori dell'oggetto, all'interno della classe. In altre parole, la classe è il progetto di un oggetto.

Vediamo una classe di esempio in Java, che definisce una (semplice) auto:

```
public class Car {
  private Engine engine;
  private Body body;
  private Tire [] tire;
  private Interior interior;

  // Constructor
  public Car (Engine engine, Body body, Tire[] tires, Interior interior) {

  }

  // Another constructor
  public Car () {

  }

  public void drive(Direction d) {
    // Method to drive
  }

  public void start(Key key) {
    // Start
  }
}
```

Questo è solo per un esempio. Puoi modellare oggetti del mondo reale come questo, secondo le tue necessità.

Leggi Classe online: https://riptutorial.com/it/oop/topic/7374/classe

Capitolo 4: Eredità

Osservazioni

Nota: l'ereditarietà a più livelli è consentita in Java ma non in ereditarietà multipla. Scopri di più su http://beginnersbook.com/2013/04/oops-concepts/

Examples

Eredità - Definizione

L'ereditarietà è uno dei concetti principali della **programmazione orientata agli oggetti (OOP)** . Usando l'ereditarietà, possiamo modellare correttamente un problema e possiamo ridurre il numero di linee che dobbiamo scrivere. Vediamo l'ereditarietà usando un esempio popolare.

Considera che devi modellare il regno animale (regno animale semplificato, ovviamente, i biologi, scusami) usando OOP. Ci sono molte specie di animali, alcune hanno caratteristiche uniche, mentre altre condividono le stesse caratteristiche.

Ci sono le principali famiglie di animali. Diciamo, `Mammals` , `Reptiles` .

Poi abbiamo figli di quelle famiglie. Per un esempio,

- `Cat` , `Dog` e `Lion` sono mammiferi.
- `Cobra` e `Python` sono rettili.

Ogni animale condivide alcune caratteristiche di base come `eat` , `drink` , `move` . Quindi possiamo dire che possiamo avere un genitore chiamato `Animal` da cui possono ereditare quelle caratteristiche di base.

Quindi anche quelle famiglie condividono alcune caratteristiche. Per un esempio i rettili usano la *scansione* per spostarsi. Ogni mammifero viene `fed milk` nelle prime fasi della vita.

Poi ci sono alcune caratteristiche uniche per ogni animale.

Considera se dobbiamo creare queste specie animali separatamente. Dobbiamo scrivere più volte lo stesso codice in ogni specie animale. Invece di quello, usiamo l'ereditarietà. Possiamo modellare il regno animale come segue:

- Possiamo avere un oggetto genitore chiamato `Animal` , che ha caratteristiche di base di tutti gli animali.
- `Mammal` e `Reptile` (ovviamente anche le altre famiglie di animali) oggetti con le loro caratteristiche comuni mentre ereditano le caratteristiche di base dall'oggetto genitore, `Animal` .
- Oggetti di specie animali: `Cat` e `Dog` ereditano dall'oggetto `Mammal` , `Cobra` e `Python` ereditano dall'oggetto `Reptile` , e così via.

In questa forma possiamo ridurre il codice che scriviamo, poiché non è necessario definire le caratteristiche di base degli animali in ogni specie animale, poiché possiamo definirli nell'oggetto `Animal` e quindi ereditarli. La stessa cosa vale per le famiglie di animali.

Esempio di successione - Considerare sotto due classi

Classe dell'insegnante:

```
class Teacher {
       private String name;
       private double salary;
       private String subject;
       public Teacher (String tname)  {
           name = tname;
       }
       public String getName()  {
           return name;
       }
       private double getSalary()  {
           return salary;
       }
       private String  getSubject()  {
            return  subject;
       }
    }
```

Classe OfficeStaff:

```
class  OfficeStaff{
   private String name;
   private double salary;
   private String dept;
   public OfficeStaff (String sname)  {
      name = sname;
   }
   public String getName()  {
       return name;
   }
   private double  getSalary()  {
       return salary;
   }
   private String  getDept ()  {
       return dept;
   }
}
```

1. Entrambe le classi condividono poche proprietà e metodi comuni. Quindi ripetizione del codice.
2. Creazione di una classe che contiene i metodi e le proprietà comuni.
3. Le classi Teacher e OfficeStaff possono ereditare tutte le proprietà e i metodi comuni dalla classe Employee.

Classe dei dipendenti:

```
class Employee{
   private String name;
   private double salary;
   public Employee(String ename){
      name=ename;
   }
   public String getName(){
      return name;
   }
   private double getSalary(){
      return salary;
   }
}
```

4. Aggiungi metodi e proprietà individuali ad esso Una volta creata una super classe che definisce gli attributi comuni a un insieme di oggetti, può essere usata per creare qualsiasi numero di sottoclassi più specifiche
5. Classi simili come Engineer, Principal possono essere generati come sottoclassi dalla classe Employee.
6. La classe genitore è definita super classe e la classe ereditata è la sottoclasse
7. Una sottoclasse è la versione specializzata di una super classe: eredita tutte le variabili di istanza e i metodi definiti dalla super classe e aggiunge i propri elementi unici.
8. Sebbene una sottoclasse includa tutti i membri della sua super classe, non può accedere a quei membri della super classe che sono stati dichiarati come privati.
9. Una variabile di riferimento di una super classe può essere assegnata a un riferimento a qualsiasi sottoclasse derivata da quella super classe es. Employee emp = new Teacher ();

Leggi Eredità online: https://riptutorial.com/it/oop/topic/7321/eredita

Capitolo 5: incapsulamento

Examples

Nascondere informazioni

Lo stato di un oggetto in un dato momento è rappresentato dalle informazioni che contiene in quel punto. In un linguaggio OO, lo stato viene implementato come variabili membro.

In un oggetto progettato correttamente, lo stato può essere modificato solo mediante chiamate ai suoi metodi e non tramite la manipolazione diretta delle sue variabili membro. Ciò si ottiene fornendo metodi pubblici che operano sui valori delle variabili dei membri privati. L'occultamento delle informazioni in questo modo è noto come *incapsulamento* .

Pertanto, l'incapsulamento garantisce che le informazioni private non siano esposte e non possano essere modificate se non attraverso chiamate rispettivamente a metodi e accessori.

Nell'esempio seguente, non puoi impostare un `Animal` per non aver più fame cambiando il campo privato `hungry` ; invece, devi invocare il metodo `eat()` , che altera lo stato `Animal` impostando il flag `hungry` su `false` .

```
public class Animal {
    private boolean hungry;

    public boolean isHungry() {
        return this.hungry;
    }

    public void eat() {
        this.hungry = false;
    }
}
```

Leggi incapsulamento online: https://riptutorial.com/it/oop/topic/5958/incapsulamento

Capitolo 6: Oggetto

Examples

introduzione

Object è il modulo di base in **Object Oriented Programming (OOP)** . Un oggetto può essere una variabile, una struttura dati (come un array, una mappa, ecc.) O anche una funzione o un metodo. In OOP modelliamo oggetti del mondo reale come animali, veicoli, ecc.

Un oggetto può essere definito in una classe, che può essere definita come il modello dell'oggetto. Quindi possiamo creare istanze di quella classe, che chiamiamo oggetti. Possiamo usare questi oggetti, i loro metodi e variabili nel nostro codice dopo.

Leggi Oggetto online: https://riptutorial.com/it/oop/topic/7377/oggetto

Capitolo 7: Polimorfismo

Examples

introduzione

Il polimorfismo è uno dei concetti di base in **OOP (Object Oriented Programming)** . L'idea principale del polimorfismo è che un oggetto ha la capacità di assumere forme diverse. Per ottenere ciò (polimorfismo), abbiamo due approcci principali.

1. Sovraccarico di metodi
    - Si verifica quando esistono due o più metodi con lo stesso nome, con parametri di input diversi. **Il tipo restituito dovrebbe essere lo stesso per tutti i metodi con lo stesso nome**
2. Metodo prioritario
    - Si verifica quando l'oggetto figlio utilizza la stessa definizione di metodo (stesso nome con gli stessi parametri), ma ha implementazioni diverse.

Usando questi due approcci possiamo usare lo stesso metodo / funzione per comportarci diversamente. Vediamo più dettagli su questo nei seguenti esempi.

Sovraccarico del metodo

L'overloading del metodo è il modo di utilizzare il polimorfismo all'interno di una classe. Possiamo avere due o più metodi all'interno della stessa classe, con diversi parametri di input.

La differenza tra i parametri di input può essere:

- Numero di parametri
- Tipo di parametri (tipo di dati)
- Ordine dei parametri

Diamo un'occhiata a loro separatamente (Questi esempi in Java, come io sono più familiarità con esso - Ci scusiamo per questo):

1. Numero di parametri

```
public class Mathematics {
    public int add (int a, int b) {
        return (a + b);
    }

    public int add (int a, int b, int c) {
        return (a + b + c);
    }
```

```
     public int add (int a, int b, int c, int c) {
         return (a + b + c + d);
     }
 }
```

Osserva attentamente, puoi vedere che il metodo restituito dal metodo è lo stesso - `int` , ma questi metodi hanno un numero diverso di input. Questo è chiamato come metodo sul caricamento con un numero diverso di parametri.

PS: questo è solo un *esempio* , non è necessario definire funzioni di aggiunta come questa.

2. Tipo di parametri

```
public class Mathematics {
    public void display (int a) {
        System.out.println("" + a);
    }

    public void display (double a) {
        System.out.println("" + a);
    }

    public void display (float a) {
        System.out.println("" + a);
    }
}
```

Nota che ogni metodo ha lo stesso nome e lo stesso tipo di ritorno, mentre hanno diversi tipi di dati di input.

PS: questo esempio serve solo a spiegare lo scopo.

3. Ordine dei parametri

```
public class Mathematics {
    public void display (int a, double b) {
        System.out.println("Numbers are " + a + " and " + b);
    }

    public void display (double a, int b) {
        System.out.println("Numbers are " + a + " and " + b);
    }
}
```

PS: questo esempio serve anche a spiegare lo scopo.

Metodo Overriding

L'override del metodo è il modo di utilizzare il polimorfismo tra le classi. se una classe è ereditata da un'altra, la prima (sottoclasse) può sovrascrivere i metodi di quest'ultima (super-classe) e modificare l'implementazione.

questo è usato dove la super classe definisce l'implementazione più generale del metodo mentre la sottoclasse usa uno più specifico.

Considera il seguente esempio:

Abbiamo una classe per i mammiferi:

```
class Mammal {
  void whoIam () {
    System.out.println("I am a Mammal");
  }
}
```

Poi abbiamo una classe per Cane, che è un Mammifero:

```
class Dog extends Mammal {
  @Override
  void whoIam () {
    super.whoIam();
    System.out.println("I am a Dog!");
  }
}
```

In questo esempio, definiamo il metodo `whoIam()` nella classe `Mammal` , dove il mammifero dice che è un Mammifero. Ma questo è un termine generale, dato che ci sono molti mammiferi là fuori. Quindi possiamo ereditare la classe `Dog` dalla classe dei `Mammal` , come Dog is a Mammal. Ma, per essere più specifici, Dog is a Dog e a Mammal. Quindi, Dog dovrebbe dire, `I am a Mammal` e anche `I am a Dog` . Quindi possiamo **sovrascrivere** il metodo `whoIam()` in super classe (classe `Mammal` , cioè) dalla sub class (la classe `Dog`).

Possiamo anche chiamare il metodo della super-classe usando `super.whoIam()` in Java. Quindi, il `Dog` si comporta come un cane, mentre si comporta anche come un mammifero.

Leggi Polimorfismo online: https://riptutorial.com/it/oop/topic/7323/polimorfismo

Capitolo 8: Problema del diamante

Examples

Diamond Problem - Esempio

Il problema del diamante è un problema comune nella programmazione orientata agli oggetti, mentre si utilizza `multiple-inheritance` .

Considera il caso in cui la `class C` è ereditata dalla `class A` e dalla `class B` Supponiamo che sia la `class A` che la `class B` abbiano un metodo chiamato `foo()` .

Quindi quando chiamiamo il metodo `foo()` , il compilatore non può identificare il metodo esatto che stiamo cercando di usare

- `foo()` dalla `class A`
- `foo()` dalla `class B`

Questo è chiamato fondamentalmente il problema dei diamanti. Ci sono alcune varianti di questo problema. Per evitare questo, ci sono diversi approcci. **Java** non consente l'ereditarietà multipla. Quindi il problema è evitato. Ma C ++ sta consentendo l'ereditarietà multipla, quindi devi stare attento all'utilizzo dell'ereditarietà multipla.

Leggi Problema del diamante online: https://riptutorial.com/it/oop/topic/7319/problema-del-diamante

Titoli di coda

S. No	Capitoli	Contributors
1	Iniziare con oop	Alon Alexander, Community, James, Thisaru Guruge
2	Astrazione	davioooh, Thisaru Guruge
3	Classe	Thisaru Guruge
4	Eredità	SRENG Khorn, Thisaru Guruge
5	incapsulamento	davioooh, jeyoung
6	Oggetto	Thisaru Guruge
7	Polimorfismo	Fantasy Pollock, murthy, Thisaru Guruge
8	Problema del diamante	Thisaru Guruge