Il polimorfismo

Polimorfismo…

Il polimorfismo è un altro caposaldo della programmazione object-oriented.

Esso fornisce un'altra dimensione di separazione dell'interfaccia dall'implementazione per dividere il *'cosa'* dal *'come'*.

Pertanto il polimorfismo permette di migliorare l'organizzazione del codice e la sua leggibilità, oltre alla creazione di programmi estensibili.

Upcasting

Per spiegare il polimorfismo per inclusione, tipico della programmazione orientata a oggetti, dobbiamo rifarci al concetto di upcasting.

L'*upcasting* è un meccanismo che permette di prendere un riferimento ad un oggetto e trattarlo come riferimento al suo tipo base.

```
// upcasting
class Note {
private int value;
private Note(int val) { value = val; }
public static final Note{
    MIDDLE_C = new Note(0),
    C_SHARP = new Note(1),
    B_FLAT = new Note(2);
} // Etc.

class Instrument {
public void play(Note n) { System.out.println("Instrument.play()");}
}
class Wind extends Instrument {
// Redefine interface method:
public void play(Note n) {  System.out.println("Wind.play()");}
}
public class Music {
   public static void tune(Instrument i) { i.play(Note.MIDDLE_C);}
public static void main(String[] args) {
     Wind flute = new Wind();
     tune(flute); // Upcasting as Wind extends Instrument}
}
```

Il binding…

Il *binding* è il meccanismo di collegamento di una invocazione di metodo al corpo del metodo.

Se il *binding* viene eseguito prima che il programma sia lanciato (dal compilatore o da un linker) allora è denominato *early binding* (o *static binding*)

Nei linguaggi procedurali, come il C, non esistevano delle opzioni di binding differenti perché, per default, avevano solo l'*early bindin*g.

Esempio

```
public static void tune(Instrument i) {
// ...
i.play(Note.MIDDLE_C);}
```

…Il binding…

Se Java supportasse solo l'*early binding*, come farebbe il compilatore a sapere che tipo di oggetto tra *Wind*, *Brass*, e *Stringed* il metodo potrebbe avere in input a run-time?

La soluzione la troviamo nel cosiddetto *late binding* (o *binding dinamico*).

Quando un linguaggio supporta il *late binding*, ci deve essere un meccanismo che determina a *run-time* il tipo dell'oggetto passato e selezionare di conseguenza il metodo appropriato. Il meccanismo di *late-binding* varia da linguaggio a linguaggio.

…Il binding.

In Java tale meccanismo è impostato per default a meno che non si dichiara esplicitamente un metodo con la parola chiave ***final***.

Il compilatore genera del codice più efficiente per i metodi dichiarati *final* anche se nella maggior parte dei casi conviene usarlo solo per una decisione di progetto anziché per migliorare le prestazioni del programma.

…Il binding.

Un'altra eccezione è rappresentata dai metodi static per i quali vale il binding statico.

```
class Base {
    public static String whoAmI() {
        return "classe base";
    }
}
class Derived extends Base {
    public static String whoAmI() {
        return "classe derivata";
    }
    public static void main(String[] argv) {
        Base prova = new Derived();
         System.out.println(prova.whoAmI());
    }
}
```

Il programma risponde "classe base".

Binding dinamico…

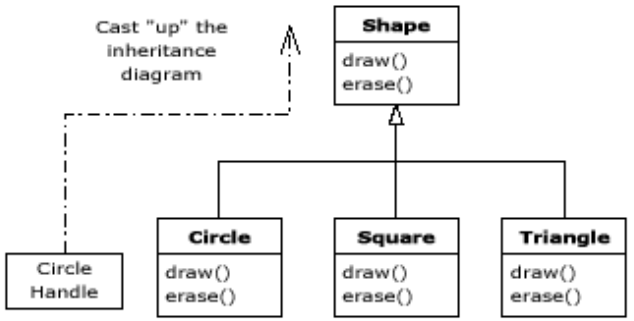
Per comprendere meglio le potenzialità offerte dal polimorfismo mediante il binding dinamico consideriamo il seguente esempio:

```
Shape s = new Circle();
```

L'oggetto *Circle* è creato e il suo riferimento viene associato immediatamente a *Shape* (ciò è possibile perché la classe *Circle* è derivata da *Shape*).

Eseguendo

s.draw()



```
class Shape {
 void draw() {}
 void erase() {}}

class Circle extends Shape {
 void draw() {
 System.out.println("Circle.draw()");}
 void erase() {
 System.out.println("Circle.erase()");}}

class Square extends Shape {
 void draw() {
 System.out.println("Square.draw()");
 }
 void erase() {
 System.out.println("Square.erase()");}}

class Triangle extends Shape {
 void draw() {
 System.out.println("Triangle.draw()");
 }
 void erase() {
 System.out.println("Triangle.erase()");}
 }
```

```
public class Shapes {
 public static Shape randShape() {
 switch((int)(Math.random() * 3)) {
 default:
 case 0: return new Circle();
 case 1: return new Square();
 case 2: return new Triangle();}
}
public static void main(String[]
args)
{
Shape[] s = new Shape[9];
for(int i = 0; i < s.length; i++)
        s[i] = randShape();
for(int i = 0; i < s.length; i++)
        s[i].draw();
}
}
```

possibile output:

```
Circle.draw()
Triangle.draw()
Circle.draw()
Circle.draw()
Circle.draw()
Square.draw()
Triangle.draw()
Square.draw()
```

…Binding dinamico.

ci aspetteremmo di invocare il metodo draw() definito all'interno della classe Shape, in realtà viene invocato il metodo draw() della classe Circle.

La classe base *Shape* stabilisce una interfaccia comune per le classi derivate che a loro volta ridefiniscono i metodi fornendo un unico comportamento per ogni specifico tipo di figura geometrica.

Il fatto di avere generato automaticamente le figure sta ad evidenziare che il compilatore non è a conoscenza del tipo di oggetto che viene passato ai metodi.

Classi e metodi astratti…

In alcune situazioni, si potrebbe desiderare definire solo l'interfaccia di una classe base senza fornire l'implementazione dei metodi.

Nell'esempio relativo agli strumenti musicali la classe base *Instrument* aveva dei *metodi fittizi* perché il suo scopo è quello di creare una *interfaccia comune* per tutte le sue sottoclassi.

Quindi lo scopo di una interfaccia comune è quello di esprimere cosa è in comune alle classi derivate.

In tal caso la classe *Instrument* viene chiamata *classe astratta*.

…Classi e metodi astratti…

Quando si definisce una classe astratta, tutti i metodi delle classi da essa derivate, la cui dichiarazione corrisponde con quelli della classe astratta, sono invocati mediante il meccanismo di binding dinamico.

Visto che una classe astratta esprime solo l'interfaccia e non una particolare implementazione, ha senso non istanziarla.

Java fornisce la parola chiave ***abstract*** per indicare che una classe o un metodo sono astratti.

…Classi e metodi astratti…

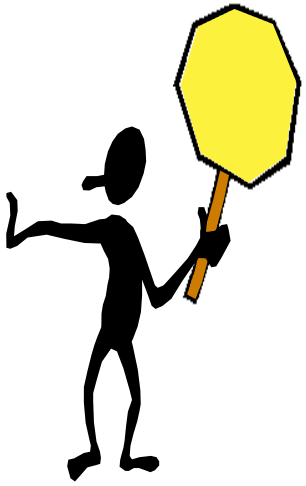
Un metodo astratto è incompleto: è comunque dichiarato ma è privo di corpo. La sintassi per la dichiarazione di un metodo astratto è la seguente:

```
abstract typeResult MethodName();
```

Se una classe contiene uno o più metodi astratti allora tale classe deve essere indicata con ***abstract*** (altrimenti viene segnalato un errore di compilazione).

Il compilatore si preoccupa di segnalare degli errori se si tenta di istanziare una classe dichiarata ***abstract***.

…Classi e metodi astratti…



Per ogni sottoclasse di una classe astratta devo ridefinirne i metodi per tutti i metodi astratti, altrimenti le classi derivate saranno considerate astratte e il compilatore richiederà che siano dichiarate come ***abstract***.

E' possibile anche avere classi astratte che non contengono alcun metodo astratto (per esempio classi di cui non si desidera alcuna istanza).

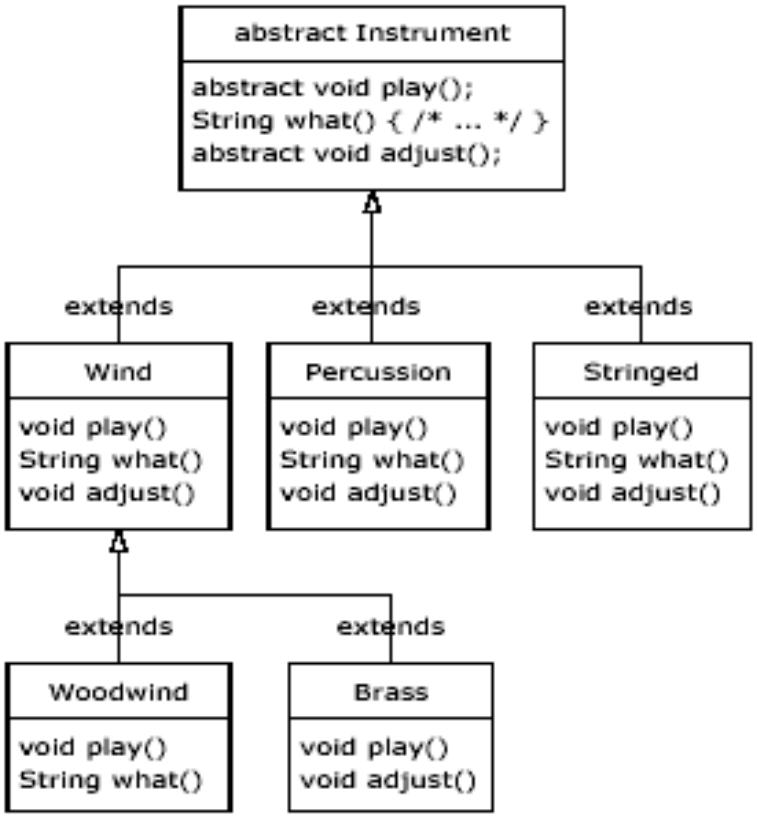
I programmatori C++ troveranno il concetto *di metodo astratto* analogo alle *funzioni virtuali pure* del loro linguaggio .

…Classi e metodi astratti.

Esempio

Se consideriamo la definizione della classe *Instrument* alla luce di quello appena visto, avremo:

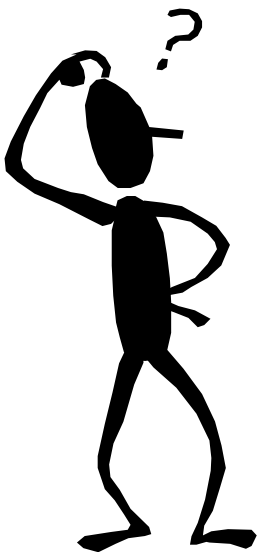
```
abstract class Instrument {
  int i; // storage allocated for each
  public abstract void play();
  public String what() {return "Instrument";}
  public abstract void adjust();}
```



Usando classi e metodi astratti si rende più esplicita l'astrazione di una classe.

Costruttori e polimorfismo…

Comportamento dei metodi polimorfici nei costruttori



Cosa accade se all'interno di un costruttore si ha una chiamata ad un metodo (con binding dinamico) dell'oggetto costruito?

Quando si invoca definito con binding dinamico in un costruttore si potrebbero avere degli effetti strani e indesiderati.

```
abstract class Glyph {
abstract void draw();
Glyph() {
    System.out.println("Glyph() before
    draw()");
    draw();
    System.out.println("Glyph() after
    draw()");
}
}
class RoundGlyph extends Glyph {
 int radius = 1;
 RoundGlyph(int r) {
      radius = r;
      System.out.println(
"RoundGlyph.RoundGlyph(), radius = "
      + radius);
 }
 void draw() {
     System.out.println(
 "RoundGlyph.draw(), radius = " +
 radius);}
 }
```

```
public class PolyConstructors {
 public static void main(String[] args) {
 new RoundGlyph(5);
 }
}
```

L'output del programma è il seguente:

```
Glyph() before draw()
RoundGlyph.draw(), radius = 1
Glyph() after draw()
RoundGlyph.RoundGlyph(), radius = 5
```

…Costruttori e polimorfismo…

Quando il costruttore di *Glyph* invoca il metodo *draw()*, il valore di *radius* non è quello inizializzato nella classe ma 0.

Il processo di inizializzazione che si è avuto è il seguente:

- la memoria allocata per l'oggetto è inizializzata a zero binario prima che accada tutto il resto;
- sono invocati i costruttori della classe base. A questo punto è invocato il metodo ridefinito *draw()* (in realtà prima che il costruttore di *RoundGlyph* sia chiamato) che scopre il valore di *radius* uguale a 0 (dovuto al passo precedente);

…Costruttori e polimorfismo.

- I membri sono inizializzati nell'ordine di dichiarazione;
- è invocato il corpo del costruttore della classe derivata

Morale: attenzione nella invocazione dei metodi all'interno di costruttori. Gli unici metodi sicuri da chiamare nei costruttori sono quelli dichiarati *final* (o *private*) nella classe base. Questi metodi non possono essere ridefiniti e non possono riservare pertanto alcun tipo di sorpresa.

Ereditarietà e finalizzazione.

Quando si usa la composizione per creare una nuova classe non ci si deve preoccupare della finalizzazione dei membri degli oggetti di tale classe.

Ogni membro è un oggetto indipendente e viene trattato dal *garbage collector* e finalizzato analogamente agli altri oggetti.

Con l'ereditarietà è necessario ridefinire il metodo *finalize()* nella classe derivata se si ha bisogno di compiere qualche particolare operazione prima che parti il *garbage collector*.

E' importante ricordare che bisogna sempre invocare il metodo *finalize()* della classe base altrimenti questa non sarà finalizzata.