Programmazione a Oggetti

Interfacce e sottotipi

Sommario

Classi e Interfacce

Subtyping/polimorfismo

Tipi Statici e Tipi Dinamici

Dynamic Method Lookup

Sviluppo del Software

Suddivisa in due fasi:

1. ***Specifica:*** *determinazione e descrizione di* ***cosa*** *fanno le singole componenti;*
2. ***Implementazione:*** *determinazione e descrizione di* ***come*** *lo fanno.*

Classi*: specifica + implementazione*

Interfacce*: costrutto per* ***separare*** *la specifica dall'implementazione*

Esempio: Pila

```
class Stack{
  private final int size=100;
  private int[] data = new int[size];
  private int top = -1;
  public void push(int el)
    { if (top<size-1) data[++top]=el;}
  public int pop()
    { if (!empty()) return data[top--];
                 else return -1;}
  public int top()
    { if (!empty()) return data[top];
                 else return -1;}
  public bool empty() {return top<0;}
}
```

Interfaccia delle pile

```
interface StackSpec
{void push(int);
 int pop(void);
 int top(void);
 bool empty(void);
}
```

***Nasconde l'implementazione**: **variabili di istanza** e **metodi non pubblici** e codice;*

*Osservate le analogie coi **prototipi del C**. Vedremo anche le differenze.*

Interfacce

*Forniscono una **specifica astratta** delle funzionalità.*

*Dichiara **i tipi dei metodi disponibili** (**pubblici**).*

***Non definisce costruttori** (quindi **non è possibile creare istanze**).*

*Un'interfaccia dichiara un **nuovo tipo.***

Interfacce: utilizzo

*È possibile associare **una o più** **implementazioni** ad una interfaccia*

```
class Stack implements StackSpec
    {. . .}
class anotherStack implements
    StackSpec {. . .}
```

Tutti i metodi *definiti nell'interfaccia devono essere dichiarati **pubblici** nella classe che la implementa e **avere lo stesso tipo**.*

Interfacce: esempio

```
interface I { int m1(int); B m2(String);}

class A implements I {
   public int m1(int i){return i;}
   public B m2(String s){return new B();}
   public void m3(){return;}
}

class B implements I {
   public int m1(int i){return i+1;}
   public B m2(String s){return this;}
}
```

Interfacce e sottotipi

Il costrutto **`implements`** ***genera una gerarchia di sottotipaggio***, `<:`, *nello spazio dei tipi.*

Se `C` *implementa* `I`, `C<:I`

`<:` *è riflessiva e transitiva*:

`T<:T`

`T<:U` **`e`** `U<:V` ⇒ `T<:V`

`<:` *è un ordine parziale*.

Sostitutività

Dati due tipi **`T<:U`** *ovunque sia atteso un riferimento di tipo U possiamo usare (**safely**) un riferimento di tipo T.*

Quindi:

***Assegnamento**: se* **`t:T`** *ed* **`<exp>`** *è un'espressione che torna un valore di tipo* **`S<:T`**, **`t=<exp>`** *è legale;*

***Parametri**: se* **`p`** *è un parametro di tipo* **`T`**, *posso passare un valore di tipo* **`S<:T`**

***Return**: se un metodo dichiara di tornare un tipo* **`T`**, *può tornare un valore di tipo* **`S<:T`**

Interfacce: riassunto

Le interfacce definiscono un tipo, quindi ***posso dichiarare variabili di tipo interfaccia****.*

Non posso creare istanze *di tipo interfaccia.*

Grazie al ***subtyping*** *una variabile di tipo interfaccia può riferire ad un oggetto istanza di una classe che la implementa.*

Quali vantaggi? ***astrazione, polimorfismo***

Tipo Statico e Tipo Dinamico

```
I ref = new C
```

`ref` *ha* ***due tipi***:

statico: **`I`** *dipende dalla* ***dichiarazione***

dinamico: **`C`** *dipende dall'****inizializzazione***

Durante la computazione può avere più tipi, purchè di ***classi sottotipi di I***.

Type checking: ***verifica staticamente***.

Goal*: evitare errori del tipo* ***message-not-understood***.

Tipo Statico e Tipo Dinamico

```
interface I {. . .}
class C implements I {. . .}
class D implements I {. . .}

I ref;
if (a<b) ref=new C();
    else ref=new D();
```

*Il tipo **dinamico** di ref **non** è prevedibile **staticamente**.*

*Tuttavia il type-checker evita il verificarsi di **errori di tipo**. **Perché?***

Type Soundness

Il type checker assicura che una variabile dichiarata di tipo **`I`** *può avere tipo dinamico* **`C`** *solo se* **`C<:I`**.

Ma ciò significa che **`C`** ***implementa*** *dichiarandoli public* ***tutti*** *i metodi definiti in* **`I`**.

La ***correttezza*** *di una invocazione di metodo* **`ref.m()`** ***dipende solo dal tipo statico*** *di* **`ref`**.

Per evitare l'eccezione ***message-not-understood*** *è sufficiente il tipo statico.*

Method Lookup

*Il **method lookup** dipenderà comunque da chi è **run-time** l'oggetto ricevente, e quindi **dal tipo dinamico**.*

```
interface I {void test();}
class C implements I {
    void test(){System.out.println("C");}
}
class D implements I {
    void test(){System.out.println("D");}
}

I ref = new C(); ref.test();   stampa C
ref = new D(); ref.test();     stampa D
```

Interfacce Multiple

Una classe può ***implementare più interfacce***

`class C implements` $I_1 \ldots I_n${…}

$I_1 \ldots I_n$ *possono dichiarare* ***lo stesso metodo*** *con stesso tipo e stessi parametri*

`C` *deve implementare e definire pubblici* ***tutti i metodi*** *(e tutte le versioni) dichiarati da*

$I_1 \ldots I_n$

`C` *è* ***sottotipo di ciascuna*** *interfaccia* I_K

Interfacce e polimorfismo

Un metodo di ordinamento ***è indipendente dal tipo degli oggetti che si ordinano****.*

*Sarebbe elegante e utile scrivere un'****unica*** *procedura* ***polimorfa*** *(****riuso del software****)*

In C questo è possibile usando alcuni trucchi (tipo `void*`*)*

In JAVA si può elegantemente ***riassumere in un interfaccia il comportamento degli oggetti ordinabili*****.**

Ogni classe `C` *di cui prevedo sia utile ordinare array contenente oggetti di tipo* `C`*, viene dichiarata implementare l'interfaccia degli oggetti ordinabili.*

Interfacce e polimorfismo

```
class OrderUtility {
  void static quicksort(OrderedType [] v)
     {...}
  void static binSearch (OrderedType [] v)
     {...}
}
interface OrderedType {
  boolean lessOrEqual(OrderedType);
  boolean less(OrderedType);
}
```

Ogni classe `C` ***che dichiara di implementare*** `OrderedType` ***dovrà definire i metodi*** `less` ***e*** `lessOrEqual`. ***Sarà legale chiamare:***

```
OrderedUtility.quicksort(a)
```

Con `a` ***vettore di istanze di*** `C`.

Esempio: sistema grafico

Le componenti di una finestra possono essere di vari tipi: **`Buttons, Label, Text, Panel, Form`**…

Ogni componente è istanza di una diversa classe che definisce almeno il metodo **`draw()`**

Una finestra ha le seguenti caratteristiche:

- *Posizione*
- *Dimensione*
- *Metodo per disegnarla a video*
- *Un metodo per aggiungere nuove componenti*

Button, Label, Window

```
class Button {
  public void draw(Window w, Position where){…}
  public void click(){…}
}

class Label {
  public void draw(Window w, Position where){…}
}

class Window {
  private int width, height;
  public void addButton(Button b, Position where)
    {b.draw(this, where);}
  public void addLabel(Label l, Position where)
    {l.draw(this, where);}
```

Window client

```
class Client {
  public static void main(String[] args)
    { Window w = new Window(50,100);
      w.addButton(new Button(),new Position(0,0));
      w.addLabel(new Label(), new Position(0,0));
     }
}
```

***Limiti e Problemi*:**

La classe window contiente molte ripetizioni di codice

Contiene solo un numero prefissato di componenti

Se aggiunge nuovi tipi di componenti dovremmo definire nuovi metodi

Overloading?

Possiamo definire un metodo add overloaded:

```
class Window {
  private int width, height;
  public void add(Button b, Position where)
    {b.draw(this, where);}
  public void add(label l, Position where)
    {l.draw(this, where);}
}
```

Possiamo usare la classe come segue:

```
w.add(new Label(), new Position(0,0));
w.add(new Button(), new Position(0,0));
```

Ma la sostanza del problema non cambia:

- ***ripetizioni di codice simile***
- ***struttura poco estensibile***

Soluzione con interfacce

Il metodo `add` ***sfrutta solo il fatto che il suo primo parametro risponda al metodo draw*****.**

Definiamo ***un'interfaccia*** `Drawable`**!**

```
interface Drawable{void draw(Window, Position);}

class Button implements Drawable{
  public void draw(Window w, Position where){…}
  public void click(){…}
}
class Label implements Drawable{
  public void draw(Window w, Position where){…}
}
```

Overloading vs Subtyping

```
class WindowOv {
  private int width, height;
  public void add(Button b, Position where)
    {b.draw(this, where);}
  public void add(Label l, Position where)
    {l.draw(this, where);}
}
```

Overloading: STATIC BINDING

```
void client(WindowOv w)
{w.add(new Button(), new Position(0,0);}
```

```
class WindowSub {
  private int width, height;
  public void add(Drawable d, Position where)
    {d.draw(this, where);}
}
```

Subtyping: DYNAMIC BINDING

```
void client(WindowSub w)
{w.add(new Button(), new Position(0,0);}
```

Raffiniamo il codice

```
interface Drawable{
    void draw(Window, Position);}
    Position getPos(void);
}
```

Ora un `drawable` deve saper restituire la sua posizione.

```
class Button{
    private Position pos;
    public void draw(Window w, Position where)
      {pos=where; ... }
    public Position getPos() {return pos;}
    public void click() {...}
}
```

Similmente per `Label, Text,` …

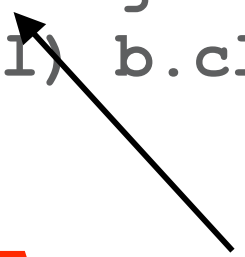
Nuova classe Window

```
class Window {
  private int width, height;
  private int numComp;
  private Drawable[] components;

  Window(int w, int h, int n)
    {width=w; hight=h;
     components=new Drawable[n];}
  public void addComp(Drawable d, Position where)
    {components[numComp++]=d;
     d.draw(this, where);}
  public Drawable getComp(Position where)
    {for (i=1; i<numComp; i++)
      {Drawable d = components[i];
       if (d.getPosition()==where) return d;}
     return null;
  }
}
```

Corretta indipendentemente dal tipo dinamico di `components[i]`

Aggiungiamo un metodo click()

```
public void click(Position where)
/* restituisce la componente che si trova
   su una certa posizione */

{Button b = w.getComp(where);
 if (b!=null) b.click();
}
```

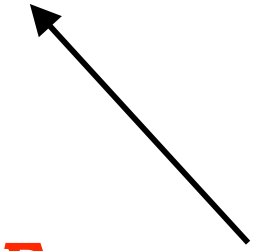


Errore di Tipo! `getComp()` ***restituisce un*** `Drawable` ***non un*** `Button` **(Button *è un sottotipo!*)**

Riproviamoci...

```
public void click(Position where)
/* restituisce la componente che si trova
   su una certa posizione */

{Drawable d = w.getComp(where);
 if (d!=null) b.click();
}
```

OK



Errore di Tipo! Un `Drawable` ***non ha un metodo*** `click()`***!***

Window: problemi

L'interfaccia **`Drawable`** *ci permette di usare un array per trattare* ***uniformemente*** *tutte le componenti.*

Il tipo ***statico*** *di una componente,* **`Drawable`***, è* ***meno informativo*** *del tipo dinamico. Da più informazione sapere che un oggetto è* **`Button`** *che non* `Drawable`*.*

E' utile un meccanismo per ***risalire*** *al tipo dinamico di un oggetto*

Downcast & InstanceOf

`ref instanceof T`

`true` *se il tipo dinamico di* `ref` *è* `T`

Downcast o UpCast: `(T) ref`

trasforma il ***tipo statico*** *di* `ref` *in* `T` *a patto che* `ref` *sia dichiarato* `S` *e* `T<:S` *o* `S<:T`

Il compilatore genera codice per controllare dinamicamente la correttezza dei cast. Altrimenti genera un eccezione (`ClassCastException`)

click() revisited

```
public void click(Position where)
/* restituisce la componente che si trova
   su una certa posizione */

{Drawable d = w.getComp(where);
 if (d!=null) && d instanceof Button)
    ((Button) d).click();
}
```

Controlla che il tipo Dinamico sia `Button`

Ha tipo statico `Button`