Azzolini Riccardo 2019-09-26

Classi e oggetti

1 Elementi di classe e di istanza

Le **classi** e gli **oggetti** contengono come **elementi**:

- dati, detti **dati membro**;
- metodi, detti **metodi membro**.

Quando ci si riferisce alla classe o a una sua istanza, si dicono **non membro** tutti i dati e i metodi esterni alla classe (ad esempio, quelli appartenenti ad altre classi).

Gli elementi si suddividono inoltre in:

- elementi **di istanza**, di cui esiste una copia indipendente per ogni oggetto che è istanza della classe;
- elementi **di classe** (`static`), di cui esiste un'unica copia, comune a tutte le istanze e accessibile anche direttamente dalla classe, senza bisogno di un riferimento a un'istanza.

Gli elementi di classe introducono quindi una dipendenza tra la classe e le sue istanze.

2 Visibilità

Ciascun elemento di una classe può avere visibilità:

- **Pubblica** (`public`): visibile a tutte le classi e a tutti gli oggetti. Essa è *assolutamente sconsigliata* per i dati membro.
- **Protetta** (`protected`): accessibile solo dai metodi
  - della classe (e delle sue istanze);
  - delle sue sottoclassi (e delle loro istanze);
  - delle classi appartenenti allo stesso package (e delle loro istanze).

In pratica, questa visibilità è leggermente più ristretta di `public`: le uniche classi che non possono vedere un membro protetto sono quelle di altri package che non sono sottoclassi. Ciò nonostante, è comunque accettabile definire dati membro protetti, poiché in questo modo si esplicita l'intenzione di renderli accessibili alle sottoclassi.

- **Limitata al package** (che è la visibilità di default, a cui non corrisponde una parola chiave): visibile a tutte le classi dello stesso package e alle loro istanze. Anche questa visibilità è *assolutamente sconsigliata* per i dati membro.
- **Privata** (`private`): accessibile solo all'interno della classe e delle sue istanze. È consigliato che i dati membro siano privati.

3 Categorie di metodi membro

- **Estensore** ("set"): è un metodo membro non privato che, almeno in alcune circostanze (ma non necessariamente sempre), modifica (o, più tecnicamente, definisce) il valore di almeno un dato membro.
- **Selettore** ("get"): è un metodo membro non privato che restituisce il valore di un dato membro, senza modificarlo.
- **Metodo di servizio** ("utility"): è un metodo membro privato.

Nel linguaggio C++ (ma non in Java) è possibile marcare esplicitamente i metodi selettori, mediante la parola chiave `const`. Ad esempio:

```
class Test {
private:
    int value;
public:
    Test(int v = 0) { value = v; }
    int getValue() const { return value; }
};
```

Il compilatore verifica che nel corpo di un metodo `const` non siano presenti istruzioni di modifica dei dati. In questo modo, l'interfaccia della classe garantisce agli utenti che il metodo non modifica mai i dati.

4 Relazione `friend` in C++

In C++, una classe può dichiarare delle funzioni `friend`: esse, pur essendo esterne alla classe stessa, possono accedere direttamente ai suoi elementi non pubblici. Ad esempio:

```
class Rectangle {
private:
    int width, height;
public:
    Rectangle() {}
    Rectangle(int x, int y) : width(x), height(y) {}
    int area() const { return width * height; }
    friend Rectangle duplicate(const Rectangle&);
};

Rectangle duplicate(const Rectangle& param) {
    Rectangle res;
    res.width = param.width * 2;
    res.height = param.height * 2;
    return res;
}

int main() {
    Rectangle foo(2, 3);
    Rectangle bar = duplicate(foo);
    std::cout << foo.area() << '\n';
}
```

È possibile dichiarare come `friend` anche un'intera classe, permettendo a tutti i suoi metodi di vedere gli elementi non pubblici. Ad esempio:

```
class Square {
    friend class Rectangle;
private:
    int side;
public:
    Square(int a) : side(a) {}
};

class Rectangle {
private:
    int width, height;
public:
    int area() const { return width * height; }
    void convert(Square a) {
        width = a.side;
        height = a.side;
    }
};
```

```
int main() {
    Square sqr(4);
    Rectangle rect;
    rect.convert(sqr);
    std::cout << rect.area() << '\n';
}
```

La relazione `friend`:

- non è simmetrica (ad esempio, `Rectangle` può accedere agli elementi privati di `Square`, ma non viceversa);
- non è transitiva;
- non viene ereditata (per esempio, eventuali sottoclassi di `Rectangle` *non* potrebbero accedere agli elementi privati di `Square`).

Essa può essere usata per:

- facilitare la scrittura del codice (permettendo di scrivere `x.y` invece di `x.getY()`, ecc.), ma questo non è un uso ideale perché rende visibile l'implementazione della classe, introducendo quindi del "debito tecnico";
- permettere un accesso più efficiente ai dati non pubblici, senza il peso aggiuntivo delle chiamate ai metodi selettori/estensori, che può essere eccessivo se ci sono requisiti particolari di performance (in tempo e/o spazio, perché la chiamata di un metodo richiede non solo un certo tempo, ma anche l'allocazione in memoria di un record d'attivazione).

In generale, è comunque meglio non abusarne.

5 Finalizzatore

Il **finalizzatore** è un metodo membro, `void finalize()`, che viene chiamato automaticamente subito prima che un oggetto venga distrutto dal *garbage collector*. Esso esprime le "ultime volontà" dell'oggetto.

Il garbage collector può distruggere un oggetto a partire dal momento in cui non ci sono più riferimenti che permettono di accedervi (perché si è usciti dall'ambito in cui tali riferimenti sono stati dichiarati, e/o perché i riferimenti sono stati sovrascritti, ad esempio assegnandovi il valore `null`). Il momento preciso in cui avvengono effettivamente l'invocazione di `finalize` e la distruzione dell'oggetto, però, non sono controllabili dal programmatore né prevedibili.[1]

[1] Per questo e altri motivi, a partire da Java 9 il metodo `finalize` è deprecato: https://docs.oracle.com/en/java/javase/12/docs/api/java.base/java/lang/Object.html#finalize()

In alternativa, il finalizzatore può essere chiamato esplicitamente dal programmatore per distruggere manualmente un oggetto.

5.1 Distruttori in C++

Nel linguaggio C++, i metodi finalizzatori sono chiamati *distruttori*.

A differenza dei finalizzatori Java, però, il distruttore di un oggetto viene invocato *immediatamente* quando si esce dall'ambito in cui tale oggetto è dichiarato.

Per questo, il funzionamento dei distruttori è completamente prevedibile.