Chương 8

Xây dựng class tổng quát hóa bằng VC#

8.0 Dẫn nhập

Chương này giới thiệu một loại class đặc biệt : class tổng quát hóa, nó giúp người lập trình tối thiểu hóa việc viết họ các class có tính chất giống nhau.

Chương này cũng giới thiệu cách miêu tả các thông tin ràng buộc kèm theo từng tên kiểu hình thức được dùng trong class tổng quát hóa, cách dùng class tổng quát hóa để yêu cầu máy sinh mã tự động ra class cụ thể.

8.1 Tổng quát về interface và class tổng quát hóa

Trong phương pháp xây dựng chương trình hướng đối tượng, chương trình là tập các đối tượng sống và tương tác lẫn nhau để hoàn thành nhiệm vụ. Số lượng các đối tượng cấu thành phần mềm thường rất lớn, nhưng chúng thường thuộc 1 số loại xác định. Viết phần mềm hướng đối tượng là quá trình lặp đặc tả các loại đối tượng cấu thành chương trình.

Trong các chương trình lớn và phức tạp, số loại đối tượng cần đặc tả có thể lớn nên thời gian, công sức đặc tả chúng cũng sẽ lớn.

Để giảm nhẹ thời gian, công sức đặc tả các đối tượng, mô hình hướng đối tượng đã giới thiệu tính thừa kế : ta không đặc tả đối tượng từ đầu (zero) mà dùng lại đặc tả có sẵn rồi hiệu chỉnh/thêm các thành phần mới. Tuy nhiên, thừa kế cũng chỉ giúp giảm nhẹ công sức đặc tả interface/class, chứ chưa triệt tiêu việc đặc tả.

Trong chương này, chúng ta sẽ thấy được phương pháp khác, nó cũng cho phép ta giảm nhẹ và triệt tiêu việc đặc tả interface/class cho 1 số class cấu thành ứng dụng. Phương pháp này được gọi là tổng quát hóa.

Ta biết, 1 hàm không tham số chỉ có thể thực hiện 1 thuật giải cố định trên các dữ liệu cố định và cho kết quả cố định, cho dù ta gọi nó bao nhiêu lần. Thí dụ hàm Cos() chỉ có thể tính được Cos của góc nào đó (được xác định cứng trong thân hàm).

Nếu thêm tham số cho hàm, nó sẽ thực hiện 1 thuật giải nhưng trên các dữ liệu khác nhau mà những lần gọi khác nhau người ta truyền cho nó, như vậy kết quả cũng sẽ khác nhau. Thí dụ hàm Cos(x) có thể tính Cos của góc x bất kỳ, tùy thuộc mỗi lần gọi nó, người ta truyền góc nào.

Như vậy, ta nói hàm có tham số sẽ có tính năng tổng quát hơn hàm không tham số. Càng có nhiều tham số, hàm càng có tính tổng quát hơn.

Tương tự, nếu ta đặc tả 1 class bình thường như đã thấy trong các chương trước, ta nói class dạng này là class cụ thể. Class cụ thể chỉ có thể chứa và xử lý các dữ liệu xác định trước. Class cụ thể chỉ có thể tạo ra các đối tượng có dữ liệu được class xác định.

Trong lập trình, chúng ta mơ ước có ai đó viết dùm mình các class cụ thể mà chương trình cần. Class tổng quát hóa sẽ giúp ta điều này. Nhiệm vụ của class tổng quát hóa là viết dùm con người các class cụ thể mà chương trình cần dùng.

Sự khác biệt giữa class tổng quát hóa và class cụ thể cũng giống như sự khác biệt giữa hàm có tham số và hàm không có tham số. Cụ thể ta sẽ định nghĩa từ 1 đến n tên kiểu hình thức mà sẽ được dùng trong class tổng quát hóa. Trong thân của class tổng quát hóa, ta sẽ dùng các tên kiểu hình thức để đặc tả cho các dữ liệu. Như vậy class tổng quát hóa không thể tạo đối tượng cụ thể (điều này không có nghĩa), nó chỉ có thể tạo ra class cụ thể khi được truyền các tên kiểu cụ thể.

Để thấy rõ sự khác biệt giữa class tổng quát hóa và class cụ thể, trước tiên ta hãy xây dựng 1 class quản lý Stack các số nguyên có dung lượng tùy ý, nó có 2 tác vụ chức năng là push(int) và pop().

8.2 Class cụ thể : Stack các số nguyên

```
//định nghĩa class Stack các số nguyên
public class IntStack {
	//định nghĩa các thuộc tính cần dùng
	private int[] data;	//danh sách đặc các số nguyên trong stack
	private int top;  // chỉ số phần tử đỉnh stack
	private int max; // số lượng max hiện hành của stack
	private int GROWBY = 4; //bước tăng dung lượng stack
	//hàm constructor
	public IntStack() {
		top = 0;
		max =GROWBY;
		//lúc đầu, phân phối GROWBY phần tử
		data = (int[])new int[max];
	}
	//hàm push phần tử vào stack
	public bool push(int newVal) {
	int[] newdata;
	if (top==max) { //kiểm tra xem stack đầy chưa
		//tạo vùng nhớ chứa các phần tử stack
		//hơn GROWBY phần tử
		try {
			newdata = (int[])new int[GROWBY+max];
		} catch (Exception e) {
			return false; //nếu hết bộ nhớ thì báo lỗi
		}
		//copy các phần tử từ stack cũ vào stack mới
		for (int i = 0; i<max; i++) newdata[i] =data[i];
		//ghi nhớ vùng stack mới
		data = newdata;
		max += GROWBY;
	}
	//chứa phần tử mới vào đỉnh stack
```

```
        data[top++] = newVal;
        return true;        //báo thành công
        }
        //hiện thực hàm pop phần tử từ đỉnh stack
        public int pop() {
        if (top == 0) //kiểm tra hết stack chưa
              throw new Exception ("Cạn stack");
        else return data[--top];      //return phần tử ở đỉnh stack
        }
}
```

8.3 Class tổng quát hóa : Stack các phần tử kiểu T

Đặc tả class IntStack ở mục 8.2 miêu tả stack các số nguyên. Giả sử trong 1 chương trình nào đó, ta cần thêm stack các số thực, stack các chuỗi, stack các trị luận lý, stack các đối tượng,...

Nếu ta tự viết lấy các class còn lại (thường bằng cách dùng lại đặc tả class IntStack rồi hiệu chỉnh lại các chi tiết cho phù hợp với kiểu phần tử trong stack mới) thì cũng được, nhưng đây là cách làm tốn nhiều thời gian, công sức, nhưng không hiệu quả, không tin cậy, dễ gây lỗi,...

Do đó ta sẽ định nghĩa class tổng quát hóa miêu tả Stack các phần tử thuộc kiểu T nào đó bằng cách :

- dùng class IntStack cụ thể, tìm và thay thế kiểu phần tử cụ thể (int) thành tên kiểu hình thức (T).
- định nghĩa kiểu hình thức T trong danh sách tham số của phát biểu class.

```
//định nghĩa class tổng quát hóa : Stack các phần tử thuộc kiểu T
public class ValueStack <T> {
        //định nghĩa các thuộc tính cần dùng
        private T[] data;//danh sách đặc các phần tử T trong stack
        private int top;   // chỉ số phần tử đỉnh stack
        private int max; // số lượng max hiện hành của stack
        private int GROWBY = 4;  //bước tăng dung lượng stack
```

```
//hàm constrcutor
public ValueStack() {
	top = 0;
	max =GROWBY;
	//lúc đầu, phân phối GROWBY phần tử
	data = (T[])new T[max];
}
//hàm push phần tử vào stack
public bool push(T newVal) {
T[] newdata;
if (top==max) { //kiểm tra xem stack đầy chưa
	//tạo vùng nhớ chứa các phần tử stack
	//hơn GROWBY phần tử
	try {
		newdata = (T[])new T[GROWBY+max];
	} catch (Exception e) {
		return false; //nếu hết bộ nhớ thì báo lỗi
	}
	//copy các phần tử từ stack cũ vào stack mới
	for (int i = 0; i<max; i++) newdata[i] =data[i];
	//ghi nhớ vùng stack mới
	data = newdata;
	max += GROWBY;
}
//chứa phần tử mới vào đỉnh stack
data[top++] = newVal;
return true;	//báo thành công
}
//hiện thực hàm pop phần tử từ đỉnh stack
public T pop() {
if (top == 0) //kiểm tra hết stack chưa
	throw new Exception ("Cạn stack");
else return data[--top];	//return phần tử ở đỉnh stack
}
```

}

8.4 Ràng buộc về tham số kiểu hình thức

Trong class tổng quát hóa ValueStack được định nghĩa ở mục 8.3, ta có dùng kiểu hình thức có tên là T. Nếu không có thông tin gì khác ngoài tên hình thức T thì trong thân của class ValueStack, ta chỉ có thể gán các biến dữ liệu thuộc kiểu hình thức T chứ không thể thực hiện được gì khác trên các dữ liệu thuộc kiểu T này.

Trong trường hợp cần thực hiện nhiều hoạt động xử lý khác trên các dữ liệu thuộc kiểu hình thức T, thí dụ gởi thông điệp nhờ thực hiện 1 tác vụ nào đó, ta phải định nghĩa thông tin ràng buộc về kiểu T.

VC# cho phép ta định nghĩa thông tin ràng buộc về kiểu hình thức T theo cú pháp sau :

```
//định nghĩa class có 3 tham số kiểu hình thức
class ValueStack <T, K, L>
where T : <thông tin ràng buộc về kiểu T>
where K : <thông tin ràng buộc về kiểu K>
where L : <thông tin ràng buộc về kiểu L>
```

Ta có thể dùng 1 trong 5 dạng ràng buộc sau đây :

1. where T: struct → kiểu T phải là kiểu giá trị (kiểu cổ điển)
2. where T: class → kiểu T phải là kiểu tham khảo (class, delegate,...)
3. where T: new( ) → kiểu T phải là kiểu đối tượng và phải có hàm contructor không tham số (contructor mặc định)
4. where T: <base class name> → kiểu T phải là kiểu đối tượng và phải tương thích với class <base class name>
5. where T: <interface name> → kiểu T phải là kiểu đối tượng và phải tương thích với interface <interface name>

Mặc định, nếu không khai báo gì thì được hiểu là where T: struct. Như vậy class ValueStack được định nghĩa ở mục 8.3 chỉ quản lý các dữ liệu cổ điển như số nguyên, số thực,...

Nếu muốn viết class tổng quát hóa miêu tả stack các đối tượng bất kỳ, ta chỉ cần hiệu chỉnh lại lệnh đặc tả class từ :

class ValueStack <T>

hay

class ValueStack <T> where T : struct

thành

class RefStack <T> where T : class

8.5 Sử dụng class tổng quát hóa

Sau khi có class cụ thể A, chương trình có thể tạo đối tượng cụ thể thuộc class cụ thể bằng cách gọi lệnh new :

```
A obj = new A();
```

Bản thân class tổng quát hóa không thể tạo ra đối tượng được dùng trong chương trình như các class thông thường. Nhiệm vụ của nó là tạo ra đặc tả class cụ thể. Thí dụ sau khi đã đặc tả được 2 class tổng quát hóa ValueStack, RefStack trong mục 8.3 và 8.4, nếu ta cần viết tự động class stack các nguyên, stack các thực,... thì ta chỉ cần viết lệnh như sau :

```
ValueStack <int> si;  //định nghĩa biến stack các số nguyên
ValueStack <double> sd;  //định nghĩa biến stack các số thực
RefStack <MyInt>  sri;       //biến stack các đối tượng nguyên
RefStack <MyDouble>  srd;      //biến stack các đối tượng thực
```

Chương trình sau demo việc dùng ValueStack để quản lý các số nguyên :

```
static void Main(string[] args) {
    int i;
    //định nghĩa class stack các số nguyên và biến thuộc class này
    ValueStack<int> si = new ValueStack<int> ();
    //push lần lượt các trị từ -5 tới 5
```

```
    for (i = -5; i <= 5; i++)  {
        if (!si.push(i)) {
            Console.WriteLine("Không push được nữa!!!");
            return;
        }
    }
    //pop ra từng phần tử cho đến khi hết stack
    try {
        while (true) {
                int ci = si.pop();
                Console.WriteLine("Trị vừa pop ra là : " + ci);
        }
    }
    //xử lý lỗi khi hết stack
    catch (Exception e) {
        Console.Write("Hết stack. Ấn Enter để đóng cửa sổ");
        Console.Read();
    }
}
```

Chương trình sau demo việc dùng RefStack để quản lý các đối tượng, mỗi đối tượng chứa 1 số nguyên :

```
 static void Main(string[] args) {
    int i;
    //định nghĩa class stack các đối tượng nguyên (class MyInt)
    //và biến thuộc class này
    RefStack<MyInt> si = new RefStack<MyInt> ();
    //push lần lượt các trị từ -5 tới 5
    for (i = -5; i <= 5; i++)  {
        if (!si.push(new MyInt(i))) {
            Console.WriteLine("Không push được nữa!!!");
            return;
        }
    }
    //pop ra từng phần tử cho đến khi hết stack
```

```
        try {
            while (true) {
                MyInt ci = si.pop();
                Console.WriteLine("Trị vừa pop ra là : " + ci.Value);
            }
        }
        //xử lý lỗi khi hết stack
        catch (Exception e) {
            Console.Write("Hết stack. Ấn Enter để đóng cửa sổ");
            Console.Read();
        }
}
```

Chương trình ở slide trước có sử dụng class MyInt để quản lý số nguyên được định nghĩa như sau :

```
 class MyInt {
        //định nghĩa thuộc tính vật lý chứa số nguyên
        private int m_value;
        //định nghĩa thuộc tính luận lý để truy xuất số nguyên
        public int Value {
              get { return m_value; }
              set { m_value = value; }
        }
        //định nghĩa hàm contructor
        public MyInt(int val) { m_value = val; }
}
```

8.6 Kết chương

Chương này đã giới thiệu một loại class đặc biệt : class tổng quát hóa, nó giúp người lập trình tối thiểu hóa việc viết họ các class có tính chất giống nhau.

Chương này cũng đã giới thiệu cách miêu tả các thông tin ràng buộc kèm theo từng tên kiểu hình thức được dùng trong class tổng quát hóa, cách dùng class tổng quát hóa để yêu cầu máy sinh mã tự động ra class cụ thể.