Ý NGHĨA CỦA INTERFACE TRONG JAVA

Nguyễn Tô Sơn

(thầy Sơn: 091.333.2869)

Chúng ta đã biết: trong lập trình hướng cấu trúc để có thể sắp xếp tăng dần một dãy số nguyên và sắp xếp tăng dần một dãy xâu thì cần phải tới hai hàm để thực hiện công việc này. Rõ ràng việc viết hai hàm với nội dung cơ bản là giống nhau sẽ mất rất nhiều thời gian và công sức đặc biệt là với các dự án lớn. Với ngôn ngữ C++ thì việc này có thể thực hiện dễ dàng bằng Template, tuy nhiên cấu trúc sử dụng có hơi phức tạp. Còn trong Java, không có khái niệm Template, nhưng có khái niệm tương tự là Generic. Mặc dù vậy, trong tình huống sắp xếp mà ta không biết là sắp xếp tăng dần, hay sắp xếp giảm dần, hay sắp xếp theo một tiêu chí nào đó thì Java sử dụng một cấu trúc linh hoạt, đơn giản hơn đó là interface (giao diện).

Để thực hiện được điều này trong Java, trước tiên ta xây dựng giao diện *SoSanhChung* có nội dung gồm một nguyên mẫu so sánh hai đối tượng là *public boolean LonHon(SoSanhChung b)*. Tiếp tục, sử dụng lớp *SapXep* để sắp xếp tăng dần các đối tượng có kiểu *SoSanhChung* này. Cuối cùng, một lớp *thongminh* hiện thực giao diện *SoSanhChung* bằng cách định nghĩa phương thức *public boolean LonHon(SoSanhChung b)* trong lớp này (ở đây ta có sử dụng kĩ thuật ép kiểu) để so sánh thành phần z của các đối tượng lớp *thongminh* với nhau (z có thể là nguyên, xâu hoặc bất cứ kiểu dữ liệu nào). Và cũng tại đây, chúng ta gọi đến phương thức *public static void SapXepDoiChoTrucTiep(SoSanhChung Mang[])* của lớp *SapXep* để thực hiện công việc sắp xếp trên các đối tượng lớp *thongminh* (ép kiểu ngầm định khi truyền tham số bởi lớp thongminh đã hiện thực giao diện *SoSanhChung*).

Rõ ràng, kĩ thuật sử dụng giao diện trong trường hợp này giảm thiểu được khá nhiều công sức khi viết code.

```
package goichinh;
interface SoSanhChung
{
      public boolean LonHon(SoSanhChung b);
}
class SapXep
{
```

```
	public static void SapXepDoiChoTrucTiep(SoSanhChung Mang[])
	{
		int i, j;
		for (i = 1; i < Mang.length - 1; i++)
			for (j = i; j < Mang.length; j++)
			if (Mang.LonHon(Mang[j]))
			{
				SoSanhChung tg;
				tg = Mang;
				Mang = Mang[j];
				Mang[j] = tg;
			}
	}
}

public class thongminh implements SoSanhChung
{
	int z;

	public thongminh(int z)
	{
		this.z = z;
	}
	public boolean LonHon(SoSanhChung b)
	{
		thongminh tg = (thongminh) b;
		return z > tg.z;
		// Trong tình huống sắp xếp giảm dần thì chỉ cần đổi
		// dấu > thành dấu <
	}
	public String toString()
	{
		return new Integer(z).toString();
	}
	public static void main(String args[])
	{
		thongminh Mang[] = new thongminh[10];
```

```
            int i;
            System.out.println("Cac phan tu cua Mang la:");
            for (i = 1; i < Mang.length ; i++)
            {
                  Mang = new thongminh((int)(100*Math.random()));
                  System.out.print(Mang.toString()+" ");
            }
            System.out.println();
            SapXep.SapXepDoiChoTrucTiep(Mang);
            for (i = 1; i < Mang.length ; i++)
            {
                  System.out.print(Mang.toString()+" ");
            }
            System.out.println();
      }
}
```

Như phân tích ở trên, giao diện giúp cho một lớp nào đó thiết lập một phương thức tổng quát để xử lý một đối tượng tổng quát (ngay cả khi đối tượng tổng quát này chưa tồn tại) và các đối tượng tổng quát trừu tượng thành đối tượng của giao diện. Mỗi phương thức tổng quát của lớp này nêu ra cách thức xử lý đối tượng tổng quát thông qua việc gọi các nguyên mẫu của giao diện, chính vì vậy mà cần thiết phải định nghĩa các nguyên mẫu này trong lớp thể hiện để có thể sử dụng.

Một ví dụ minh hoạ cuộc sống tự nhiên của chúng ta "phương tiện giao thông có thể là ô tô" được áp dụng trong cách sử dụng của giao diện như sau:

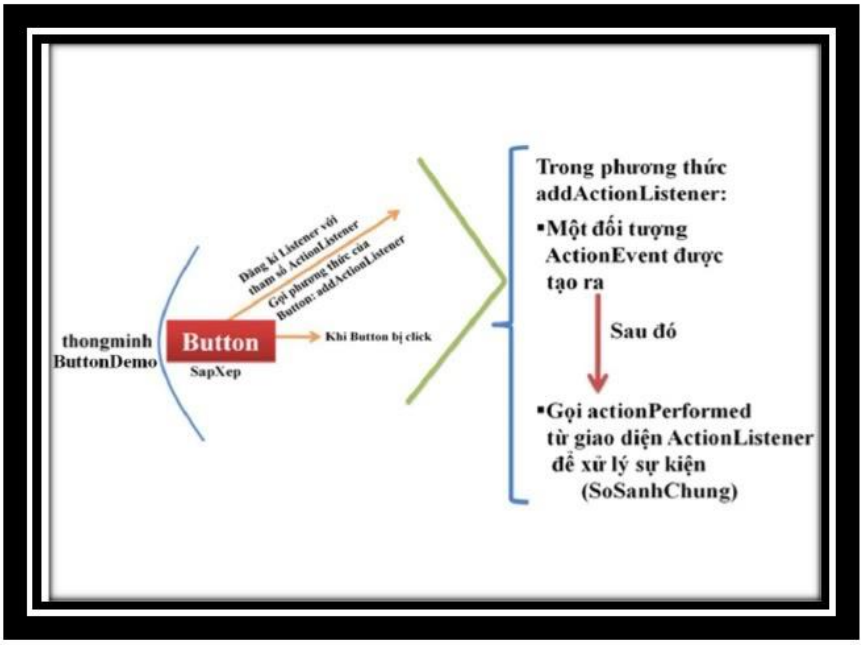
```
SoSanhChung x = new thongminh();
thongminh y = new thongminh();
if (x.LonHon(y)) System.out.println("x > y");
      else System.out.println("x <= y");
```

Điều này cũng giải thích cơ chế xử lý sự kiện *Event Handler*: cơ chế này hoạt động tương tự như chương trình mà chúng ta vừa tạo ra. Để thấy rõ điều này, ta xét thêm ví dụ về xử lý sự kiện:

```
package goichinh;
import java.awt.*;
import java.awt.event.*;
```

```
public class ButtonDemo extends Frame implements ActionListener
{
      Button KetThuc = new Button("Ket Thuc");
      public void actionPerformed(ActionEvent ae)
      {
            if (ae.getSource() == KetThuc) System.exit(0);
      }
      public ButtonDemo(String Title)
      {
            super(Title);
            setLayout(new FlowLayout());
            add(KetThuc);
            KetThuc.addActionListener(this);
      }
      public static void main(String args[])
      {
            ButtonDemo d = new ButtonDemo("Button Demo");
            d.setSize(200, 300);
            d.setVisible(true);
      }
}
```

So sánh hai ví dụ để thấy sự tương đồng của hai chương trình cũng như từng lớp và giao diện ta xây dựng sơ đồ sau:



Trong sơ đồ trên ta cần chú ý thêm rằng: Java sử dụng chế độ đa luồng. Còn thế nào là chế độ đa luồng, ta có thể lấy ví dụ thực tế như trình soạn thảo văn bản Micrsoft Word mà chúng ta vẫn thường sử dụng vừa có thể kiểm tra lỗi chính tả vừa có thể soạn thảo được văn bản cùng một lúc.

Phân tích cơ chế xử lý sự kiện *Event Handler* dựa vào sơ đồ trên ta nhận thấy: Khi *Button* đăng kí xử lý sự kiện trong chế độ đa luồng của Java nó sẽ lắng nghe đến khi có một sự kiện xảy ra bằng lời gọi phương thức *addButtonListener(this)*. Ý nghĩa của lời gọi này như sau: Khi có tình huống "nhấn nút" xảy ra hãy nhờ đối tượng *ActionListener* được truyền vào thông qua tham số *this* của lớp thể hiện của *ActionListener* là *ButtonDemo* để xử lý giúp tôi, chỉ thế thôi.

Bởi vì trong lớp *ButtonDemo* đã hiện thực nguyên mẫu *actionPerformed* của giao diện *ActionListener* nên hoạt động lắng nghe của *Button* đã được cụ thể hoá để trình biên dịch có thể hiểu và dịch được.

Từ đây ta càng thấy được sự cần thiết của việc phải định nghĩa lại các nguyên mẫu của giao diện trong lớp hiện thực. Khi đã hiện thực giao diện, Java cho phép bạn sử dụng phương thức xử lý tổng quát của lớp khác trong lớp hiện thực để tiết kiệm thời gian bằng cách chỉ cần định nghĩa lại các nguyên mẫu của giao diện. Nếu không sử dụng cách này, bạn sẽ phải sao chép toàn bộ phương thức xử lý tổng quát đó sang lớp của bạn. Rõ ràng công việc này rất dễ phát sinh lỗi. Đặc biệt trong môi trường xử lý sự kiện nếu không sử dụng giao diện thì bạn phải xây dựng tất cả lại từ đầu vì Java chủ yếu dựa vào khái niệm giao diện, công việc này hết sức khó khăn (Bạn không biết khi click chuột thì làm thế nào để nhận biết chẳng hạn). Java sử dụng giao diện để ứng phó cho những tình huống luôn luôn biến đổi khi thừa kế, giảm thiểu thời gian viết code.

Như vậy, khi có tình huống "nhấn nút" thì trong thân phương thức *addButtonListener* sẽ có một đối tượng *ActionEvent* được tạo ra. Sau đó, cũng trong thân phương thức này gọi *actionPerformed* để xử lý sự kiện này.

Trong ví dụ về xử lý sự kiện ở trên ta nhận xét: nếu có nhiều *Button* và khi click vào các *Button* này sẽ làm thay đổi màu nền của *Frame* thì các công việc này cơ bản là giống nhau, nếu xử lý như ở trên thì rất rườm rà. Để khắc phục được điều này ta xây dựng thêm một lớp *DoiMau* để có thể đổi màu một đối tượng *Frame*.

Chúng ta có chương trình như sau:

```
package goichinh;
import java.awt.*;
import java.awt.event.*;

class DoiMau extends Frame implements ActionListener
{
	Frame frame;
	Color color;
	public void actionPerformed(ActionEvent ae)
	{
		frame.setBackground(color);
	}
	public DoiMau(Frame frame, Color color)
	{
		this.frame = frame;
		this.color = color;
	}
}
public class ButtonDemo2 extends Frame implements ActionListener
{
	Button KetThuc = new Button("Exit");
	public void actionPerformed(ActionEvent ae)
	{
		if (ae.getSource() == KetThuc) System.exit(0);
	}
	public ButtonDemo2(String Title)
	{
		super(Title);
		add(KetThuc);
		KetThuc.addActionListener(this);
		setLayout(new FlowLayout());
		Button red = new Button("Red");
		add(red);
		red.addActionListener(new DoiMau(this, Color.red));
		Button blue = new Button("Blue");
		add(blue);
```

```
            blue.addActionListener(new DoiMau(this, Color.blue));
            Button black = new Button("Black");
            add(black);
            black.addActionListener(new DoiMau(this, Color.black));
      }
      public static void main(String args[])
      {
            ButtonDemo2 d = new ButtonDemo2("Button Demo 2");
            d.setSize(200, 300);
            d.setVisible(true);
      }
}
```

Chúng ta chỉ có thể thừa kế duy nhất một lớp cha nhưng có thể hiện thực hàng loạt giao diện. Giao diện khắc phục nhược điểm của đa thừa kế trong C++ khi hai lớp cha có phương thức với tên giống nhau bởi trong Java thì các nguyên mẫu được định nghĩa duy nhất ở lớp thể hiện. Điều đó dẫn tới việc muốn thực hiện được đa thừa kế Java yêu cầu người lập trình phải xây dựng các lớp với các phương thức tổng quát thông qua giao diện. Khi đó lớp con hiện thực sẽ sử dụng được các phương thức tổng quát này. Và chỉ có thể sử dụng được các phương thức public không tĩnh mà thôi (yêu cầu của giao diện).