

## Javatecture II

### Factories for Making Classes

James W. Cooper

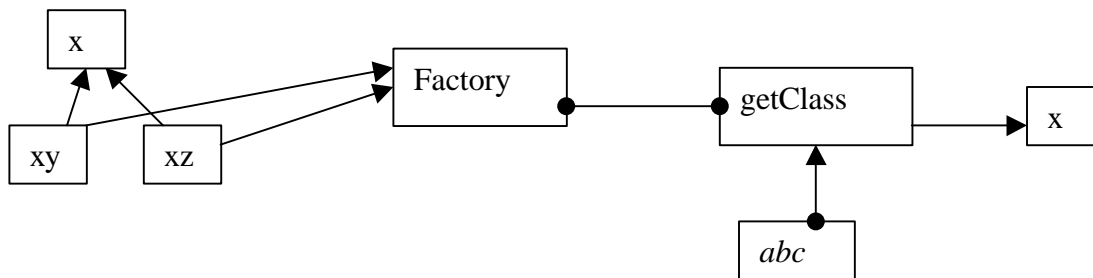
In our last column we began talking about how Java makes us write code using objects, but doesn't require us to write good object-oriented (OO) code. While it is pretty easy to construct Java objects (since you have to), the real crux of OO programming is how those objects communicate among each other. Riel (1) quoted an acquaintance of his as noting that while OO programming precludes writing "spaghetti code," you can still write "ravioli code."

One of the most interesting developments in the past few years has been the cataloguing and discussion of *Design Patterns*. As we noted in our last column, Design Patterns are not cutesy computer science abstractions, but catalogs of useful, tested ways to write effective OO programs.

#### Factory Patterns

One type of pattern that we see again and again in OO programs is the Factory pattern or class. A Factory pattern is one that returns an instance of one of several possible classes depending on the data which it is presented with. Usually all of the classes it returns have a common parent class and common methods, but each of them performs a task differently and is optimized for different kinds of data.

To understand a Factory pattern, lets look at Figure 1



**Figure 1**

In this figure **x** is a base class and classes **xy** and **xz** are derived from it. The Factory is a class that decides which of these subclasses to return depending on the arguments you give it. On the right, we define a *getClass* method which passes in some value *abc*, and which returns some instance of the class **x**. Which one it returns doesn't matter to the programmer since they all have the same methods, but different implementations. How it decides is entirely up to the factory. It could be some very complex function but it is often quite simple.

## A Name Splitting Factory

Let's consider a simple case where we could use a Factory class. Suppose we have an entry form and we want to allow the user to enter his name either as "firstname lastname" or as "lastname, firstname". We'll make the further simplifying assumption that we will always be able to decide the name order by whether there is a comma between the last and first name.

Now this is a pretty simple sort of decision to make, and you could make it with a simple if statement in a single class, but let's use it here to illustrate how a factory works and what it can produce. We'll start by defining a simple base class that takes a String and splits it (somehow) into two names:

```
class Namer {
//a simple class to take a string apart into two names
    protected String last;    //store last name here
    protected String first;   //store first name here

    public String getFirst()   {
        return first;        //return first name
    }
    public String getLast()    {
        return last;        //return last name
    }
}
```

In this base class we don't actually do anything, but we do provide implementations of the **getFirst** and **getLast** methods. We'll store the split first and last names in the Strings **first** and **last**, and, since the derived classes will need access to these variables, we'll make them *protected*.

## The Two Derived Classes

Now we can write two very simple derived classes which split the name into two parts in the constructor. In the FirstFirst class, we assume that everything before the last space is part of the first name:

```
class FirstFirst extends Namer {           //split first last
    public FirstFirst(String s) {
        int i = s.lastIndexOf(" ");      //find sep space
        if (i > 0) {
            //left is first name
            first = s.substring(0, i).trim();
            //right is last name
            last =s.substring(i+1).trim();
        }
        else {
            first = "";                   // put all in last name
            last = s;                     // if no space
        }
    }
}
```

And, in the LastFirst class, we assume that the last name is delimited by a comma. In both classes, we also provide error recovery in case the space or comma does not exist.

```

class LastFirst extends Namer {           //split last, first
    public LastFirst(String s)           {
        int i = s.indexOf(",");          //find comma
        if (i > 0) {
            //left is last name
            last = s.substring(0, i).trim();
            //right is first name
            first = s.substring(i + 1).trim();
        }
        else {
            last = s;                     // put all in last name
            first = "";                   // if no comma
        }
    }
}

```

### ***Building that Factory***

Now our Factory class is extremely simple. We just test for the existence of a comma and then return an instance of one class or the other:

```

class NameFactory {
    //returns an instance of LastFirst or FirstFirst
    //depending on whether a comma is found
    public Namer getNamer(String entry) {
        int i = entry.indexOf(",");      //comma determines name order
        if (i>0)
            return new LastFirst(entry); //return one class
        else
            return new FirstFirst(entry); //or the other
    }
}

```

### **Using the Factory**

Now let's see how we put this together. I've constructed a simple user interface to a Java application which allow you to enter the name in either order and see the results displayed. You can see this program in Figure 2, and can download it from this magazine's FTP site.



## Figure 2

You type in a name and then click on the **Compute** button, and the divided name appears in the text fields below. The crux of this program is the compute method which fetches the text, gets an instance of a Namer class and displays the results.

In our constructor for the program, we initialize an instance of the factory class with

```
NameFactory nfactory = new NameFactory();
```

Then, when we process the button action event, we call the **computeName** method, which calls the **getNamer** factory method and then calls the name methods of the class instance it returns:

```
private void computeName() {
    //send the text to the factory and get a class back
    namer = nfactory.getNamer(entryField.getText());

    //compute the first and last names
    //using the returned class
    txFirstName.setText(namer.getFirst());
    txLastName.setText(namer.getLast());
}
```

And that's all there is to using Factory patterns. You create an abstraction which decides which of several possible classes to return and returns one. Then you call the methods of that class instance without ever knowing which derived class you are actually using. And, as you can see, it's a neat trick to keep the issues of data dependence separated from the classes' useful methods.

### \*\*\*\*Sidebar\*\*\*\* *Factory Patterns in Math Computation*

Most people who use Factory patterns tend to think of them as tools for simplifying tangled programming classes. But it is perfectly possible to use them in programs which simply do mathematical computations. For example, in the Fast Fourier Transform (FFT), you evaluate the following four equations repeatedly for a large number of point pairs over many passes through the array you are transforming. Because of the way the graphs of these computations are drawn, these equations constitute one instance of the FFT "butterfly." These are shown as Equations 1-4.

$$R_1' = R_1 + R_2 \cos(y) - I_2 \sin(y) \quad (1)$$

$$R_2' = R_1 - R_2 \cos(y) + I_2 \sin(y) \quad (2)$$

$$I_1' = I_1 + R_2 \sin(y) + I_2 \cos(y) \quad (3)$$

$$I_2' = I_1 - R_2 \sin(y) - I_2 \cos(y) \quad (4)$$

However, there are a number of times during each pass through the data, where the angle  $y$  is zero. In this case, your complex math evaluation reduces to Equations (5-8)

$$R_1' = R_1 + R_2 \quad (5)$$

$$R_2' = R_1 - R_2 \quad (6)$$

$$I_1' = I_1 + I_2 \quad (7)$$

$$I_2' = I_1 - I_2 \quad (8)$$

So it is not unreasonable to package this computation in a couple of classes which do the simple or the expensive computation depending on the angle  $y$ . We'll start by creating a Complex class which allows us to manipulate real and imaginary number pairs:

```
class Complex {
    float real;
    float imag;
}
```

It also will have appropriate get and set functions.

Then we'll create our Butterfly class as an abstract class which we'll fill in with specific descendants:

```
abstract class Butterfly {
    float y;
    public Butterfly() {
    }
    public Butterfly(float angle) {
        y = angle;
    }
    abstract public void Execute(Complex x, Complex y);
}
```

Our two actual classes for carrying out the math are called **addButterfly** and **trigButterfly**. They implement the computations shown in equations (1-4) and (5-8) above.

```
class addButterfly extends Butterfly {
    float oldr1, oldi1;

    public addButterfly(float angle) {
    }
    public void Execute(Complex xi, Complex xj) {
        oldr1 = xi.getReal();
        oldi1 = xi.getImag();
        xi.setReal(oldr1 + xj.getReal()); //add and subtract R's and I's
        xj.setReal(oldr1 - xj.getReal());
        xi.setImag(oldi1 + xj.getImag());
        xj.setImag(oldi1 - xj.getImag());
    }
}
```

and for the trigonometric version:

```
class trigButterfly extends Butterfly {
```

```

float y;
float oldr1, oldi1;
float cosy, siny;
float r2cosy, r2siny, i2cosy, i2siny;

public trigButterfly(float angle)    {
    y = angle;
    cosy = (float) Math.cos(y);    //precompute sine and cosine
    siny = (float) Math.sin(y);

}

public void Execute(Complex xi, Complex xj)    {
    oldr1 = xi.getReal();           //multiply by cos and sin
    oldi1 = xi.getImag();
    r2cosy = xj.getReal() * cosy;
    r2siny = xj.getReal() * siny;
    i2cosy = xj.getImag()*cosy;
    i2siny = xj.getImag()*siny;
    xi.setReal(oldr1 + r2cosy +i2siny); //store sums
    xi.setImag(oldi1 - r2siny +i2cosy);
    xj.setReal(oldr1 - r2cosy - i2siny);
    xj.setImag(oldi1 + r2siny - i2cosy);
}
}
}

```

Finally, we'll can make a simple factory class which decides which class instance to return. Since we are making Butterflies, we'll call our Factory a Cocoon:

```

class Cocoon {
    public Butterfly getButterfly(float y)    {
        if (y !=0)
            return new trigButterfly(y);    //get mulitply class
        else
            return new addButterfly(y);    //get add/sub class
    }
}
-----end of sidebar-----

```

### **More Factories**

A Factory pattern is only the beginning. It is not hard to imagine factory classes that themselves make different kinds of factories, depending on the type and quantity of data. We can also think of cases where we might create a different user interface depending on what sort of data are presented to the factory. These more complicated factories are called Builder patterns and we'll look into one in detail in the next column.

### **References**

1. Gamma, Eric; Helm, Richard; Johnson, Ralph and Vlissides, John, *Design Patterns. Elements of Reusable Software.*, Addison-Wesley, Reading, MA, 1995
2. Riel, Arthur J., *Object-Oriented Design Heuristics*, Addison-Wesley, Reading, MA, 1996

James W. Cooper is a computer science researcher and the author of a number of books in the field. His most recent book is *Principles of Object-Oriented Programming in Java 1.1*, Ventana, 1997