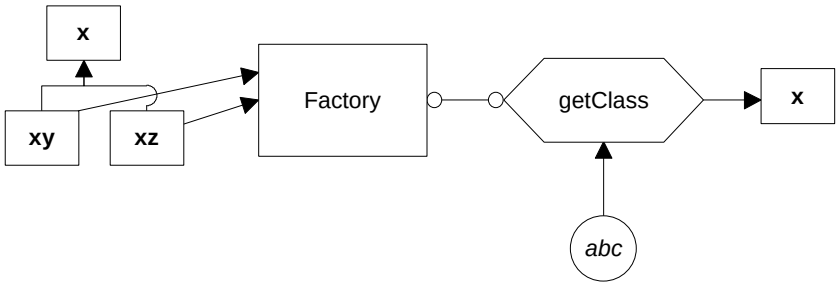
THE FACTORY PATTERN

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 provided to it. 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.

How a Factory Works

To understand a Factory pattern, let's look at the Factory diagram below.



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 to be one that passes in some value *abc*, and that 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 which one to return is entirely up to the factory. It could be some very complex function but it is often quite simple.

Sample Code

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.

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 that 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 a comma delimits the last name. 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 the 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

Let’s see how we put this together.

We have constructed a simple Java user interface that allows you to enter the names in either order and see the two names separately displayed. You can see this program below.

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 that fetches the text, obtains 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 first and last 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 the fundamental principle of 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. This approach keeps the issues of data dependence separated from the classes' useful methods. You will find the complete code for Namer.java on the example CD-ROM.

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 that simply perform 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 doing the simple or the expensive computation depending on the angle *y*. We'll start by creating a Complex class that 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 that 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
     xj.setReal(oldr1 - xj.getReal());
     xi.setImag(oldi1 + xj.getImag());
     xj.setImag(oldi1 - xj.getImag());
   }
}
```

and for the trigonometic 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 can make a simple factory class that 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 multiply class
      else
         return new addButterfly(y);    //get add/sub class
   }
}
```

You will find the complete FFT.java program on the example CDROM.

When to Use a Factory Pattern

You should consider using a Factory pattern when

- A class can't anticipate which kind of class of objects it must create.
- A class uses its subclasses to specify which objects it creates.
- You want to localize the knowledge of which class gets created.

There are several similar variations on the factory pattern to recognize.

1. The base class is abstract and the pattern must return a complete working class.
2. The base class contains default methods and is only subclassed for cases where the default methods are insufficient.
3. Parameters are passed to the factory telling it which of several class types to return. In this case the classes may share the same method names but may do something quite different.

Thought Questions

1. Consider a personal checkbook management program like Quicken. It manages several bank accounts and investments and can handle your bill