**Factory Method**

**Definition**

Provides an abstraction or an interface and lets subclass or implementing classes decide which class or method should be instantiated or called, based on the conditions or parameters given.

**Where to use & benefits**

* Connect parallel class hierarchies.
* A class wants its subclasses to specify the object.
* A class cannot anticipate its subclasses, which must be created.
* A family of objects needs to be separated by using shared interface.
* The code needs to deal with interface, not implemented classes.
* Hide concrete classes from the client.
* Factory methods can be parameterized.
* The returned object may be either abstract or concrete object.
* Providing hooks for subclasses is more flexible than creating objects directly.
* Follow naming conventions to help other developers to recognize the code structure.
* Related patterns include
  + [Abstract Factory](http://www.javacamp.org/designPattern/abstractfactory.html) , which is a layer higher than a factory method.
  + [Template method](http://www.javacamp.org/designPattern/template.html), which defines a skeleton of an algorithm to defer some steps to subclasses or avoid subclasses
  + [Prototype](http://www.javacamp.org/designPattern/prototype.html), which creates a new object by copying an instance, so it reduces subclasses.
  + [Singleton](http://www.javacamp.org/designPattern/singleton.html), which makes a returned factory method unique.

**Examples**

To illustrate such concept, let's use a simple example. To paint a picture, you may need several steps. A shape is an interface. Several implementing classes may be designed in the following way.

interface Shape {

public void draw();

}

class Line implements Shape {

Point x, y;

Line(Point a, Point b) {

x = a;

y = b;

}

public void draw() {

//draw a line;

}

}

class Square implements Shape {

Point start;

int width, height;

Square(Point s, int w, int h) {

start = s;

width = w;

height = h;

}

public void draw() {

//draw a square;

}

}

class Circle implements Shape {

....

}

class Painting {

Point x, y;

int width, height, radius;

Painting(Point a, Point b, int w, int h, int r) {

x = a;

y = b;

width = w;

height = h;

radius = r;

}

Shape drawLine() {

return new Line(x,y);

}

Shape drawSquare() {

return new Square(x, width, height);

}

Shape drawCircle() {

return new Circle(x, radius);

}

....

}

...

Shape pic;

Painting pt;

//initializing pt

....

if (line)

pic = pt.drawLine();

if (square)

pic = pt.drawSquare();

if (circle)

pic = pt.drawCircle();

From the above example, you may see that the Shape pic's type depends on the condition given. The variable pic may be a line or square or a circle.

You may use several constructors with different parameters to instantiate the object you want. It is another way to design with Factory pattern. For example,

class Painting {

...

Painting(Point a, Point b) {

new Line(a, b); //draw a line

}

Painting(Point a, int w, int h) {

new Square(a, w, h); //draw a square

}

Painting(Point a, int r){

new Circle(a, r); //draw a circle

}

...

}

You may use several methods to finish the drawing jobs. It is so-called factory method pattern. for example,

class Painting {

...

Painting(Point a, Point b) {

draw(a, b); //draw a line

}

Painting(Point a, int w, int h) {

draw(a, w, h); //draw a square

}

Painting(Point a, int r){

draw(a, r); //draw a circle

}

...

}

The above draw() methods are overloaded.

Here is a popular example of Factory design pattern. For example, you have several database storages located in several places. The program working on the database is the same. The user may choose local mode or remote mode. The condition is the choice by the user. You may design your program with Factory pattern. When the local mode is set, you may instantiate an object to work on the local database. If the remote mode is set, you may instantiate an object which may have more job to do like remote connection, downloading, etc.

interface DatabaseService {

public DataInfo getDataInfo() throws Exception;

public FieldInfo getFieldInfo() throws Exception;

public void write(FieldInfo fi) throws Exception;

public void modify(FieldInfo fi) throws Exception;

public void delete(FieldInfo fi) throws Exception;

//...

}

class Data implements DatabaseService {

public Data(String fileName) {...};

public Data(URL url, String fileName) {....};

public DataInfo getDataInfo() throws Exception {...};

public FieldInfo getFieldInfo() throws Exception {...};

public void write(FieldInfo fi) throws Exception {...};

public void modify(FieldInfo fi) throws Exception {...};

public void delete(FieldInfo fi) throws Exception {...};

//....

}

class DataManager{

Data data = null;

...

if (local) {

data = new Data(localFile);

...

}

if (remote){

data = new Data(connectRemote, databaseFile);

...

}

data.write(someInfo);

data.modify(someInfo);

....

}

To illustrate how to use factory design pattern with class level implementation, here is a real world example. A company has a website to display testing result from a plain text file. Recently, the company purchased a new machine which produces a binary data file, another new machine on the way, it is possible that one will produce different data file. How to write a system to deal with such change. The website just needs data to display. Your job is to provide the specified data format for the website.

Here comes a solution. Use an interface type to converge the different data file format. The following is a skeleton of implementation.

//Let's say the interface is Display

interface Display {

//load a file

public void load(String fileName);

//parse the file and make a consistent data type

public void formatConsistency();

}

//deal with plain text file

class CSVFile implements Display{

public void load(String textfile) {

System.out.println("load from a txt file");

}

public void formatConsistency() {

System.out.println("txt file format changed");

}

}

//deal with XML format file

class XMLFile implements Display {

public void load(String xmlfile) {

System.out.println("load from an xml file");

}

public void formatConsistency() {

System.out.println("xml file format changed");

}

}

//deal with binary format file

class DBFile implements Display {

public void load(String dbfile) {

System.out.println("load from a db file");

}

public void formatConsistency() {

System.out.println("db file format changed");

}

}

//Test the functionality

class TestFactory {

public static void main(String[] args) {

Display display = null;

//use a command line data as a trigger

if (args[0].equals("1"))

display = new CSVFile();

else if (args[0].equals("2"))

display = new XMLFile();

else if (args[0].equals("3"))

display = new DBFile();

else

System.exit(1);

//converging code follows

display.load("");

display.formatConsistency();

}

}

//after compilation and run it

C:\>java TestFactory 1

load from a txt file

txt file format changed

C:\>java TestFactory 2

load from an xml file

xml file format changed

C:\>java TestFactory 3

load from a db file

db file format changed

In the future, the company may add more data file with different format, a programmer just adds a new class in accordingly. Such design saves a lot of code and is easy to maintain.