**Factory design pattern**

If you ask someone to create an object in java, the most common answer you will get is using \*\*\*“new keyword”\*\*\*. Well we have lots of other ways also to create object but this is most easiest and used way to create an object.

\*\*\*But do you think in huge enterprise application it is better way to create object using new key word ??\*\*\*

Well No..

Reason behind not to create object using new is when we create object in one class using new key word we tightly coupled class with another class which is not at all the best practice in software industry. We create application loosely coupled so that we can scale our application as per our requirement.

**Cicrle class is tightly coupled in Main class**

public class Main{

**Circle c=new Circle();**

}

public class Circle{

int x;

int y

}

Fig1: tight coupling

![tight coupling]( https://github.com/PiyushMittl/java/blob/master/java\_designpattern\_factory\_im1\_tightcoupling.png)

\*\*\*how do we create object then ??\*\*

hold on guys.. don worry, we will create object using new keyword only but not in the way we have created earlier. We will use one design pattern ie Factory design pattern to create object in loosely coupled manner.

Factory design pattern comes under creational design pattern. And be used to create object in loosely coupled manner. If you have used Spring framework which provide spring bean factory(<http://piyushanandmittal.blogspot.com/2018/04/javaspringbeanfactory.html>) in our application which uses factory pattern internally and is the widely used framework which highly uses factory pattern.

**#### Implementation:**

In our example we will have one interface “Shape” and will create three implementations for the same.

We will create object by sending some of the information of the object we want and in return we will receive the object from factory.

asks

creates

Shape factory

+getShape():Shape

Factory pattern Demo

+main(): void

<<interface>>

Rectangle

+draw(): void

Circle

+draw(): void

Square

+draw(): void

Shape

+draw(): void

Fig2: loose coupling using Factory

![loose coupling]( https://github.com/PiyushMittl/java/blob/master/java\_designpattern\_adapter\_im2.png)

**#### Steps to create Factory**

**\*\*Step1:\*\*** Create Shape interface.

**\*\*Step2:\*\*** Create concrete classes Circle, Square, Rectangle implementing Shape interface.

**\*\*Step3:\*\*** Create factory to create concreate class object on the basis of provided information.

**\*\*Step4:\*\*** Use factory to get object of concrete class by passing information like shape type.

**\*\*Step1:\*\*** Create Shape interface.

``` java

public interface Shape{

public void drawShape();

}

```

**\*\*Step2:\*\*** Create concrete classes Circle, Square, Rectangle implementing Shape interface.

``` java

public class Circle implements Shape{

public void drawShape(){

System.out.println(“draw Circle”);

}

}

```

``` java

public class Square implements Shape{

public void drawShape(){

System.out.println(“draw Square”);

}

}

```

``` java

public class Rectangle implements Shape{

public void drawShape(){

System.out.println(“draw Rectangle”);

}

}

```

**\*\*Step3:\*\*** Create factory to create concreate class object on the basis of provided information.

``` java

Public class ShapeFactory{

public static Shape getShape(String type){

if(type.equals(“CIRCLE”))

{

Return new Circle();

}

if(type.equals(“SQUARE”))

{

Return new Square();

}

if(type.equals(“RECTANGLE”))

{

Return new Rectangle();

}

}

}

```

**\*\*Step4:\*\*** Use factory to get object of concrete class by passing information like shape type.

``` java

public class Main{

public static void main(String s[]){

Shape shape=ShapeFactory.getShape(“CIRCLE”);

Shape.draw();

Shape shape=ShapeFactory.getShape(“SQUARE”);

Shape.draw();

Shape shape=ShapeFactory.getShape(“RECTANGLE”);

Shape.draw();

}

}

```

**#### The need of Factories:**

There is lots of reason of using Factories in your project and some of them are below-

**\*\*Control over instantiation of object\*\***

Sometimes we require database connection, socket connection or a filesystem descriptor etc. which are very costly in terms of creating objects and because of that we want to limit the object creation like 1 or 2 or 3 instances per class.

The constructor approach is not at all appropriate to achieve the above problem because we it’s difficult to know if for a particular class the object is already created or not.

Using a static factory method, you could easily do that:

``` java

public class Singleton {

private static final Singleton INSTANCE = new Singleton();

private Singleton(){}

public static Singleton getInstance(){

return INSTANCE;

}

...

}

...

```

``` java

public class ClassXXX{

...

public static void someFunctionInClassXXX(){

Singleton instance = Singleton.getInstance();

//some stuff

}

}

...

```

``` java

public class ClassYYY{

...

public static void someFunctionInClassYYY(){

Singleton instance = Singleton.getInstance();

//some stuff

}

}

```

In this code, we’re using a factory that limits the number of instance of the class Singleton to one. By limiting the number of objects we’re creating a pool of instances and this Pool Pattern is based on a factory.

Note: Instead of limiting the number of instances, we could have modified the way an instance is created (for example by using a prototype pattern instead of creating a new object from scratch each time).

**\*\*Loose coupling\*\***

Another advantage of factories is the loose coupling.

Let’s assume you write a program that computes stuff and needs to write logs. Since it’s a big projet, one of your mate codes the class that writes the logs into a filesystem (the class FileSystemLogger) while you’re coding the business classes. Without factories, you need to instanciate the FileSystemLogger with a constructor before using it:

``` java

public class FileSystemLogger {

...

public void writeLog(String s) {

//Implemation

}

}

...

public void someFunctionInClassXXX(some parameters){

FileSystemLogger logger= new FileSystemLogger(some paramters);

logger.writeLog("This is a log");

}

```

But what happens if there is a sudden change and you now need to write logs in a database with the implememtation DatabaseLogger? Without factories, you’ll have to modify all the functions using the FileSystemLogger class. Since this logger is used everywhere you’ll need to modify hundreds of functions/classes whereas using a factory you could easiliy switch from one implementation to another by modifying only the factory:

``` java

//this is an abstraction of a Logger

public interface ILogger {

public void writeLog(String s);

}

```

``` java

public class FileSystemLogger implements ILogger {

...

public void writeLog(String s) {

//Implemation

}

}

```

``` java

public class DatabaseLogger implements ILogger {

...

public void writeLog(String s) {

//Implemation

}

}

```

``` java

public class FactoryLogger {

public static ILogger createLogger() {

//you can choose the logger you want

// as long as it's an ILogger

return new FileSystemLogger();

}

}

```

``` java

////////////////////some code using the factory

public class SomeClass {

public void someFunction() {

//if the logger implementation changes

//you have nothing to change in this code

ILogger logger = FactoryLogger.createLogger();

logger.writeLog("This is a log");

}

}

```

If you look at this code, you can easily change the logger implementation from FileSystemLogger to DatabaseLogger. You just have to modify the function createLogger() (which is a factory). This change is invisible for the client (business) code since the client code use an interface of logger (ILogger) and the choice of the logger implementation is made by the factory. By doing so, you’re creating a loose coupling between the implementation of the logger and the parts of codes that uses the logger.

**\*\*Encapsulation\*\***

Sometimes, using a factory improves the readibility of your code and reduces its complexity by encapsulation.

Let assume you need to use a business Class CarComparator that compares 2 cars. This class needs a DatabaseConnection to get the features of millions of cars and a FileSystemConnection to get a configuration file that parametrizes the comparison algorithm (for example: adding more weigth to the fuel consumption than the maximum speed).

Without a factory you could code something like:

``` java

public class DatabaseConnection {

DatabaseConnection(some parameters) {

// some stuff

}

...

}

```

``` java

public class FileSystemConnection {

FileSystemConnection(some parameters) {

// some stuff

}

...

}

```

``` java

public class CarComparator {

CarComparator(DatabaseConnection dbConn, FileSystemConnection fsConn) {

// some stuff

}

public int compare(String car1, String car2) {

// some stuff with objets dbConn and fsConn

}

}

...

public class CarBusinessXY {

public void someFunctionInTheCodeThatNeedsToCompareCars() {

DatabaseConnection db = new DatabaseConnection(some parameters);

FileSystemConnection fs = new FileSystemConnection(some parameters);

CarComparator carComparator = new CarComparator(db, fs);

carComparator.compare("Ford Mustang","Ferrari F40");

}

...

}

```

``` java

public class CarBusinessZY {

public void someOtherFunctionInTheCodeThatNeedsToCompareCars() {

DatabaseConnection db = new DatabaseConnection(some parameters);

FileSystemConnection fs = new FileSystemConnection(some parameters);

CarComparator carComparator = new CarComparator(db, fs);

carComparator.compare("chevrolet camaro 2015","lamborghini diablo");

}

...

}

```

This code works but you can see that in order to use the comparison method, you need to instanciate

```

\* a DatabaseConnection,

\* a FileSystemConnection,

\* then a CarComparator.

```

If you need to use the comparison in multiple functions, you will have to duplicate your code which means if the construction of the CarComparator changes, you will have to modifiy all the duplicated parts. The use of a factory could factorize the code and hide the complexity of the construction of the CarComparator class.

``` java

...

public class Factory {

public static CarComparator getCarComparator() {

DatabaseConnection db = new DatabaseConnection(some parameters);

FileSystemConnection fs = new FileSystemConnection(some parameters);

CarComparator carComparator = new CarComparator(db, fs);

}

}

```

``` java

//////////////////////////////some code using the factory

public class CarBusinessXY {

public void someFunctionInTheCodeThatNeedsToCompareCars() {

CarComparator carComparator = Factory.getCarComparator();

carComparator.compare("Ford Mustang","Ferrari F40");

}

...

}

...

```

``` java

public class CarBusinessZY {

public void someOtherFunctionInTheCodeThatNeedsToCompareCars() {

CarComparator carComparator = Factory.getCarComparator();

carComparator.compare("chevrolet camaro 2015","lamborghini diablo");

}

...

}

```

If you compare both codes, you can see that using a factory:

\* Reduces the number of line of code.

\* Avoid code duplication.

\* Organise the code: the factory has the responsibility to build a CarComparator and the business class just uses it.

The last point is important (in fact, they’re all important!) because a it’s about separation of concerns. A business class shouldn’t have to know how to build a complex object it needs to use: the business class needs to focus only on business concerns. Moreover, it also increases the division of work among the developers of the same project:

\* One works on the CarComparator and the way it’s created.

\* Others work on business objects that use the CarComparator.

**\*\*Disambiguation\*\***

Let’s assume that you have a class with multiple constructors (with very different behaviors). How can you be sure that you won’t use the wrong constructor by mistake?

Let’s look at the following code:

``` java

class Example{

//constructor one

public Example(double a, float b) {

//...

}

//constructor two

public Example(double a) {

//...

}

//constructor three

public Example(float a, double b) {

//...

}

}

```

Though constructor one and two doesn’t have the same number of arguments, you can quicky fail to choose the right one, especially at the end of a busy day using the nice autocomplete from you favorite IDE (I’ve been there). It’s even more difficult to see the difference between constructor one and constructor three. This example looks like a fake one but I saw it on legacy code (true story !).

The question is, how could you implement different constructors with the same type of parameters (while avoiding a dirty way like the contructors one and three) ?

Here is a clean solution using a factory:

``` java

class Complex {

public static Complex fromCartesian(double real, double imag) {

return new Complex(real, imag);

}

public static Complex fromPolar(double rho, double theta) {

return new Complex(rho \* Math.cos(theta), rho \* Math.sin(theta));

}

private Complex(double a, double b) {

//...

}

}

```

In this example, using a factory adds a description of what the creation is about with the factory method name: you can create a Complex number from cartesian coordinates or from polar coordinates. In both cases, you know exactly what the creation is about.

Types of factory pattern:

```

the **factory method pattern**,

the **abstract factory pattern**,

the **static factory method**,

the **simple factory (also called factory)**.

```

**\*\*static factory method\*\***

Note: If you read this article and don’t know a lot about Java, a static method is a class method.

The static factory method was described by Joshua Bloch in “Effective Java”:

“A class can provide a public static factory method, which is simply a static method that returns an instance of the class.”

In other words, instead of using a constructor to create an instance, a class could provide a static method that returns an instance. If this class has subtypes, the static factory method can return a type of the class or its subtypes. Though I hate UML, I said in the beginning of the article that I’d use UML to give a formal description. Here it is:

ObjectWithStaticFactory

getObject() : ObjectWithStaticFactory

SubType2

SubType1

Fig3: Static Factory Method

![static factory method]( https://github.com/PiyushMittl/java/blob/master/java\_designpattern\_factory\_im3\_staticfactorymethod.png)

simplified UML version of static factory

In this diagram, the class ObjectWithStaticFactory has a static factory method (called getObject()). This method can instanciate any type of class ObjectWithStaticFactory, which means a type ObjectWithStaticFactory or a type SubType1 or a type SubType2. Of course, this class can have other methods, properties and static factory methods.

Let’s look at this code:

``` java

public class MyClass {

Integer a;

Integer b;

MyClass(int a, int b){

this.a=a;

this.b=b;

};

public static MyClass getInstance(int a, int b){

return new MyClass(a, b);

}

public static void main(String[] args){

//instanciation with a constructor

MyClass a = new MyClass(1, 2);

//instanciation with a static factory method

MyClass b = MyClass.getInstance(1, 2);

}

}

```

This code shows 2 ways to create an instance of MyClass:

\* a static factory method getInstance() inside MyClass

\* a constructor of MyClass

But this concept can go deeper. What if a class with a static factory method could instanciate another class? Joshua Bloch described this possibility:

“Interfaces can’t have static methods, so by convention, static factory methods for an interface named Type are put in a noninstantiable class (Item 4) named Types.”

Here is the associated UML:

Types

Type

getObject: Type

SubType1

SubType1

Fig4:

![fig4]( https://github.com/PiyushMittl/java/blob/master/java\_designpattern\_factory\_im4.png)

simplified UML version of static factory

In this case the factory method getObject is inside an abstract class name Types. The factory method can create instances of class Type or any subtype of class Type (SubType1 or SubType2 in the diagram). The getObject() method can have parameters so that it returns a SubType1 for a given parameter and a SubType2 otherwise.

Let’s got back in Java and assume that we have 2 classes: Ferrari and Mustang that implements an interface Car. The static factory method can be put in an abstract class named “CarFactory” (using Joshua Boch’s conventions the name of the class should be “Cars” but I don’t like it):

``` java

/////////////////////////the products

public interface Car {

public void drive();

}

```

``` java

public class Mustang implements Car{

public void drive() {

// some stuff

}

...

}

```

``` java

public class Ferrari implements Car{

public void drive() {

// some stuff

}

...

}

```

``` java

///////////////////////// the factory

public abstract class CarFactory{

public static Car getCar() {

// choose which car you want

}

}

...

/////////////////////////some code using the factory

public static void someFunctionInTheCode(){

Car myCar = CarFactory.getCar();

myCar.drive();

}

```

The power of this pattern compared to the other factory patterns is that you don’t need

\* to instanciate the factory in order to use it (you’ll understand what I mean in a few minutes),

\* the factory to implement an interface (same comment).

It’s easy to use but works only for langages that provides class methods (i.e the static java keyword).

\*\*Note:\*\* When it comes to factories, many posts on the net are wrong , like this post on stackoverflow that was upvoted 1.5k times. The problem with the given examples of factory method patterns is that they are static factory methods. If I quote Joshua Bloch:

“a static factory method is not the same as the Factory Method pattern from Design Patterns [Gamma95, p. 107]. The static factory method described in

this item has no direct equivalent in Design Patterns.”

If you look at the stackoverflow post, only the last example (the URLStreamHandlerFactory) is a factory method pattern by the GoF (we’ll see this pattern in a few minutes)

\*\*Real examples\*\*

Here are some examples of static factory methods in Java frameworks and Java APIs. Finding examples in the Java APIs is very easy since Joshua Bloch was the main architect for many Java APIs.

\*\*Logging frameworks\*\*

The java logging frameworks slf4j, logback and log4j use an abstract class, LoggerFactory. If a developper wants to write logs, he needs to get an instance of Logger from the static method getLogger() of LoggerFactory.

The Logger implementation returned by getLogger() will depend on the of getLogger() implementation (and also the configuration file writen by the developer that is used by getLogger()).

``` java

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

public class Example{

public void example() {

//we're using the static factory method to get our logger

Logger logger = LoggerFactory.getLogger(Example.class);

logger.info("This is an example.");

}

}

```

Note: the name of the factory class and its static factory method is not exactly the same whether you’re using slf4j or log4j or slf4j.

\*\*Java String class\*\*

The String class in Java represents a string. Sometimes, you need to get a string from a boolean or an integer. But String doesn’t provide constructors like String(Integer i) or String(Boolean b). Instead, it provides multiple static factory methods

```

String.valueOf(…).

int i = 12;

String integerAsString = String.valueOf(i);

```

**\*\*simple factory\*\***

This pattern is not a “real one” but I ‘ve seen it many times on the Internet. It doesn’t have a formal description but here is mine: A simple factory (or factory) is a tool

whose job is to create/instantiate objects,

and is neither a factory method pattern (we’ll see this pattern after),

nor an abstract factory pattern (same comment).

You can see it has a generalization of the static factory pattern but this time the factory can be instantiated (or not) because the “factory method” is not a class method (but it can). For a Java developer using the simple factory in its non-static form is rare. So, this pattern is most of the time equivalent to the static one. Here is an UML for the non-static form:

Type

SubType1

SubType1

Types

getObject: Type

Fig5: Simple Factory

![simple factory]( https://github.com/PiyushMittl/java/blob/master/java\_designpattern\_factory\_im5\_simplefactory.png)

simplified UML version of simple factory

In this case the factory method getObject() is inside a class named Factory. The factory method is not class method so, you need to instanciate the Factory before using it. The factory method can create instances of class Type or any of its subtypes .

Here is the previous example from the static factory method but this time I instantiate the factory before using it

``` java

/////////////////////////the products

public interface Car {

public void drive();

}

```

``` java

public class Mustang implements Car{

public void drive() {

// some stuff

}

...

}

```

``` java

public class Ferrari implements Car{

public void drive() {

// some stuff

}

...

}

```

``` java

/////////////////////////The factory

public class CarFactory{

//this class is instantiable

public CarFactory(){

//some stuff

}

public Car getCar() {

// choose which car you want

}

}

...

/////////////////////////some code using the factory

public static void someFunctionInTheCode(){

CarFactory carFactory = new CarFactory();

Car myCar = carFactory.getCar();

myCar.drive();

}

```

As you see, this time I need to instanciate the Factory in order to use it. I didn’t find real example in java since it’s better to use a static factory method than a simple factory. Still, you can use this pattern in its non-static form if your factory method needs some instances to work. For example, if you need a database connection, you could first instanciate your factory (that would instanciate the database connection) and then use the factory method that requieres this connection. Personnally, in this case I’d still use a static factory with lazy initialization (and a pool of database connections).

Tell me if you know a Java framework that uses a simple factory in its non-static form.

**\*\*factory method pattern\*\***

The factory method pattern is a more abstract factory. Here is the definition of the pattern given by the “Gang of Four”:

“Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses”

Here is a simplified UML diagram of the factory method pattern:

<<interface>>

Factory

getObject: Type

Type

SubType1

SubType1

<<Realize>>

Concrete Factory

getObject: Type

Fig6: Factory Method Pattern

![factory method pattern]( https://github.com/PiyushMittl/java/blob/master/java\_designpattern\_factory\_im6\_factorymethodpattern.png)

simplified UML version of factory method pattern

This diagram looks like the simple factory one (in its non-static form). The only (and BIG !) difference is the interface Factory:

\* the Factory represents the “interface for creating an object”. It describes a factory method: getObjects().

\* the ConcreteFactory represents one of the “subclasses [that] decide which class to instantiate”. Each ConcreteFactory has its own implementation of the factory method getObjects().

In the diagram getObjects() has to return a Type (or its subtypes). Which means that one conctrete factory could return a Subtype1 whereas another could return a SubType2.

Why using a factory method pattern instead of a simple factory?

Only when your code requires multiple factory implementations. This will force each factory implemantion to have the same logic so that a developer that uses one implementation can easily switch to another one without wandering how to use it (since he just have to call the factory method that has the same signature).

Since this is abstract, let’s go back to the car example. It’s not a great example but I use it so that you can see the difference with the simple factory, (we’ll see the real examples to understand the power of this pattern):

``` java

/////////////////////////the products

public interface Car {

public void drive();

}

```

``` java

public class Mustang implements Car{

public void drive() {

// some stuff

}

...

}

```

``` java

public class Ferrari implements Car{

public void drive() {

// some stuff

}

...

}

```

``` java

///////////////////////// the factory

//the definition of the interface

public interface CarFactory{

public Car getCar() {}

}

//the real factory with an implementation of the getCar() factory method

public class ConcreteCarFactory implements CarFactory{

//this class is instantiable

public CarFactory(){

//some stuff

}

public Car getCar() {

// choose which car you want

return new Ferrari();

}

}

...

/////////////////////////some code using the factory

public static void someFunctionInTheCode(){

CarFactory carFactory = new ConcreteCarFactory();

Car myCar = carFactory.getCar();

myCar.drive();

}

```

If you compare this code with the simple factory, I added this time an interface (CarFactory). The real factory (ConcreteCarFactory) implements this interface.

As I said this is not a great example because in this example you shouldn’t use a factory method pattern since there is only one concrete factory. It would be useful only if I have multiple implementations like SportCarFactory, VintageCarFactory, LuxeCarFactory, CheapCarFactory … . In this case, a developper could easily switch from one implementation to another since the factory method is always getCar().

Real examples

Java API

In java, a common example is the iterator() function in the collection API. Each collection implements the interface Iterable<E> . This interface describes a function iterator() that returns an Iteractor<E>. An ArrayList<E> is a collection. So, it implements the interface Iterable<E> and its factory method iterator() that returns a subclass of Iterator<E>

``` java

//here is a simplified definition of an iterator from the java source code

public interface Iterator<E> {

boolean hasNext();

E next();

void remove();

}

```

``` java

//here comes the factory interface!

public interface Iterable<T> {

Iterator<T> iterator();

}

//here is a simplified definition of ArrayList from the java source code

//you can see that this class is a concrete factory that implements

//a factory method iterator()

//Note : in the real Java source code, ArrayList is derived from

//AbstractList which is the one that implements the factory method pattern

public class ArrayList<E> {

//the iterator() returns a subtype and an "anonymous" Iterator<E>

public Iterator<E> iterator()

{

return new Iterator<E>()

{

//implementation of the methods hasNext(), next() and remove()

}

}

...

}

And here is a standard use of the ArrayList

import java.util.ArrayList;

import java.util.Iterator;

import java.util.List;

public class Example {

public static void main(String[] ars){

//instantiation of the (concrete factory) ArrayList

List<Integer> myArrayList = new ArrayList<>();

//calling the factory method iterator() of ArrayList

Iterator<Integer> myIterator = myArrayList.iterator();

}

}

```

I showed an ArrayList but I could have shown a HashSet, a LinkedList or a HashMap since they are all part of the collection API. The strength of this pattern is that you don’t need to know what type of collections you’re using, each collection will provide an Iterator through the factory method iterator().

Another good example is the stream() method in the new Java 8 collection API.

Spring

The Spring Framework is based on a factory method pattern. The ApplicationContext implements the BeanFactory Interface. This interface describes a function Object getBean(param) that returns an Object. This example is interesting because, every Class in java are derived from Object. So, this factory can return an instance of any class (depending on the parameters).

``` java

public class Example{

public static void main(String[] args) {

//creation of the BeanFactory

ApplicationContext context = new ClassPathXmlApplicationContext("config.xml");

//creation totaly different type of objets with the factory

MyType1 objectType1 = context.getBean("myType1");

MyType2 objectType2 = context.getBean("myType2");

}

}

**```**

**\*\*The abstract factory\*\***

Here comes the fat one ! This factory was described by the Gang of Four with the following sentence:

“Provide an interface for creating families of related or dependent objects without specifying their concrete classes”

If you don’t understand this sentence, don’t worry it’s normal. It’s not called abstract factory for nothing!

If it can help you, I see the abstract factory pattern as a generalization of the factory method pattern execpt this time the factory interface have multiple factory methods that are related. When I say related, I mean conceptually linked so that they form a “familly” of factory methods. Let’s look at the UML diagram to see the difference with the factory method pattern:

<<interface>>

Factory

getObject1(): Type1

getObject2(): Type2

Type2

SubType2.1

SubType2.2

Type1

SubType1.1

SubType1.2

<<Realize>>

Concrete Factory

getObject1(): Type

getObject2(): Type2

Fig7: Abstract Factory

![abstract factory]( https://github.com/PiyushMittl/java/blob/master/java\_designpattern\_factory\_im7\_abstractfactory.png)

simplified UML version of abstract factory pattern

\* Factory is an interface that defines mulitple factory methods, 2 in our cases: getObject1() and getObject2(). Each method creates a type (or its subtypes).

\* ConcreteFactory implements the Factory interface and therefore has its own implementation of getObject1() and getObject2(). Now imagine 2 concrete factories: one could returns instances of SubType1.1 and SubType2.1 and the other SubType1.2 and SubType2.2

Since this is very absract, let’s go back to the CarFactory example.

With the factory method pattern, the factory interface had just one method, getCar(). An abstract factory could be an interface with 3 factory methods: getEngine(), getBody() and getWheel(). You could have multiple concrete factories:

\* SportCarFactory that could return instances of PowerfulEngine, RaceCarBody and RaceCarWheel

\* CheapCarFactory that could return instance of WeakEngine, HorribleBody and RottenWheel

If you want to build a sport car, you’ll need to instanciate a SportCarFactory then use it. And if you want to build a cheap car, you’ll need to instanciate a CheapCarFactory then use it.

The 3 factory methods of this abstract factory are related. They all belong to the car production concept.

Of course, the factory methods can have parameters so that they return different types. For example getEngine(String model) factory from SportCarFactory could return a Ferrari458Engine or a FerrariF50Engine or a Ferrari450Engine or … depending on the parameter.

Here is the same example in java (with only the SportCarFactory and 2 factory methods).

``` java

/////////////////////////the different products

public interface Wheel{

public void turn();

}

```

``` java

public class RaceCarWheel implements Wheel{

public void turn(){

// some stuff

}

...

}

```  
``` java

public interface Engine{

public void work();

}

```

``` java

public class PowerfulEngine implements Engine{

public void work(){

// some stuff

}

...

}

```

``` java

/////////////////////////the factory

public interface CarFactory{

public Engine getEngine();

public Wheel getWheel();

}

```

``` java

public class SportCarFactory implements CarFactory{

public Engine getEngine(){

return new PowerfulEngine();

}

public Wheel getWheel(){

return new RaceCarWheel();

}

}

```

``` java

/////////////////////////some code using the factory

public class SomeClass {

public void someFunctionInTheCode(){

CarFactory carFactory = new SportCarFactory();

Wheel myWheel= carFactory.getWheel();

Engine myEngine = carFactory.getEngine();

}

}

```

This factory is not an easy one. So, when should you use it ?

NEVER!!!! Hum, difficult to answer. I see this factory pattern as a way to organise code. If you end up with many factory method patterns in your code an you see a common theme between some a them, you can gather this group with an abstract factory. I’m not a big fan of the “let’s use an abstract factory because we could need one in the future” because this pattern is very abstract. I prefer building simple things and refactor them after if needed.

Yet, a common use case is when you need to create a user interface with different look and feel. This example was used by the Gang of Four to present this pattern. This UI will need some products like a window, a scroll bar, buttons … You can create a Factory with a concret factory for each look and feel. Of course, this example was written before the Internet era, now you could have one component and modify its look and feel using CSS (or some scripting languages) even for desktop applications. Which means a static factory method is most of the time enough.

But if you still want to use this pattern, here are some use cases from the GoF:

“a system should be configured with one of multiple families of products”

“you want to provide a class library of products, and you want to reveal just their interfaces, not their implementations”

Real examples

Most DAO (Data Access Object) frameworks use an abtract factory to specify the basic operations a concrete factory shoud do. Though the names of the factory methods depend on the framework, it’s often closed to:

```

\* createObject(…) or persistObject(…)

\* updateObject(…) or saveObject(…)

\* deleteObject(…) or removeObject(…)

\* readObject(…) or findObject(…)

```

For each type of object you manipulate, you’ll need a concrete factory. For example, if you need to manage people, houses, and contracts using a database. I’ll have a PersonFactory, a HouseFactory and a ContractFactory.

The CrudRepository from Spring is a good example of an abstract Factory.

If you want Java code, you can look for JPA, Hibernate or SpringData tutorials.