the Factory Design Pattern

Eric McCreath
In this lecture we will:

- motivate and define the factory design pattern (simple factories, the method factory pattern, and the abstract factory pattern),
- provide a template for implementing your own classes that use this pattern, and
- have a "live code" example.
The simple approach for designing a program is to have a class that contains all the information of your program. Then to group together parts of this information and create other classes and their associated methods. Then we use "new" to construct objects of these classes and use fields to keep references to them in the containing classes. This approach produces simple robust designs.
However, we have also created a design in which there is coupling or dependency from the containing classes to the contained classes. Thus modifying or adding classes will often involve changing dependent classes. Also the strongly coupled classes are not easily reused for other situations or programs.
Now say we wish to maintain a different version of the program that uses "OtherSmallClass" rather than "SmallClass". Then we need to duplicate and modify both OtherClass and BigClassWhichHoldsProgram. This is not a good solution.
We could also place "if-else" statements within "AnotherClass" selecting between "SmallClass" and "OtherSmallClass", this would create duplicated code.
An improvement on using "if-else" statements would be to create an abstract class or interface and then within "AnotherClass" only refer to the abstract class or interface. This is a big step forward as we can change the "SmallClass" without having to change most of the code in "AnotherClass". The exception to this is the constructor, as we need to create concrete classes. So we still have a dependency between "AnotherClass" and the "SmallClass" / "OtherSmallClass". This is where factories help!
We now focus in on the "AnotherClass", "SmallInterface", "SmallClass", and "OtherSmallClass" part of this example and we gives these classes the more general names "Client", "Product", "ConcreteProductA", and "ConcreteProductB" respectively.
Factories

Factories produce objects. The "new" operator is a type of factory as it creates an object for you of a particular type. However when you use the "new" operator you tie your code into a particular class and a particular implementation of that class. You need to change the code using the "new" operator if you would like to use a different class or implementation. The factory approaches shift the code that determines which objects to construct away from the code using the objects. So now classes that implement the objects can change without requiring the code using the classes to change. So we go from:

```java
Product product = new ConcreteProductA();
product.method();
```

To:

```java
Product product = someSortOfFactorMethod();
product.method();
```
One simple approach is to create a method within a class that is responsible for constructing objects of a particular type. So following our example through we could add a method in another class that creates objects of type "Product" for us. Now whenever we want an object of this type we call this method rather than using the "new" operator.

```java
public Product createProduct(Info info) {
    if (info.wantProductA()) {
        return new ConcreteProductA();
    } else {
        return new ConcreteProductB();
    }
}
```

If we made the method static it would be a static factory method (still a simple factory).
Simple Factory

We have shifted all the code into one spot, which is good. We can easily add or change the class that is implementing the "Product" within our program. However, we still have a dependency between the class using this simple factory method and the classes implementing the Products. In this case via the "ClassWithSimpleFactoryMethod".

```
Client -> Product
    ^           ^
  |           |
ConcreteProductA -> ConcreteProductB
    |           |
ClassWithSimpleFactoryMethod
```
The **factory method pattern** uses an abstract method to describe the method that creates the objects. Clearly the class that extends the abstract class will be dependent on particular classes implementing the objects we wish to create. However, the abstract class which has the abstract constructor method is completely independent of particular product implementations.

So in our example we may make the Client abstract adding the abstract method:

```java
abstract Product createProduct(Info info);
```

The factory method pattern defers the code that constructs the concrete products to the implementing subclass. The great thing about this approach is the "Client" does not dependent in any way on the concrete product classes.
So the Client will have code which uses and depends on only the Product interface (or abstract class). For creating objects of type Product it will use the "createProduct" method. This method is abstract in the client yet implemented in the ConcreteClient. In effect we have used inheritance to make the Client independent from the implementations of the products. Hence we can change/modify/add new Products without changing the Client.
Abstract Factory Pattern

In some cases we may wish to group together individual factory methods. This can be done by creating an interface or abstract class that holds a collection of abstract methods for constructing the objects of our different products. We then create a concrete factory class which implements all these methods for constructing the required objects. By using composition the client using this factory is made independent from product implementations.

So this abstract factory class would have abstract methods:

```java
abstract Product1 createProduct1(Info1 info1);
abstract Product2 createProduct2(Info2 info1);
```

This class would be extended and the methods for creating the objects implemented.
The client would have a field (composition) that reference abstract factory. This factory object would be passed to the Client when the Client is constructed. Thus the code in the client would not be dependent on either the ConcreteFactory or any of the Concrete Products.
Explain the difference between: a static factory method, the factory method pattern, and the abstract factory pattern?
What are some key advantages of using a factory design pattern? What are some disadvantages?