The Simple Factory Pattern
and
Factory Method Pattern
New is an implementation

- Calling “new” is certainly coding to an implementation
- In fact, it’s always related to a concrete class
- (Open for Modification)
- That’s fine when things are simple, but ...
Look Out For Change

• When you have several related classes, that’s probably a good sign that they might change in the future

```java
Duck duck;
if (picnic) {
    duck = new MallardDuck();
} else if (hunting) {
    duck = new DecoyDuck();
} else if (inBathTub) {
    duck = new RubberDucky();
}
```
Instantiating Concrete Classes

- Using `new` instantiates a concrete class.
- This is programming to implementation instead of interface.
- Concrete classes are often instantiated in more than one place.
- Thus, when changes or extensions are made all the instantiations will have to be changed.
- Such extensions can result in updates being more difficult and error-prone.
Example

Suppose that you are required to develop a system that accepts orders for pizzas. There are three types of pizzas, namely, cheese, Greek, and pepperoni. The pizzas differ according to the dough used, the sauce used and the toppings.

Draw a class diagram for the system.
Design Principle

• What should you try to do with code that changes?

Pizza orderPizza() {
    Pizza pizza = new Pizza();

    pizza.prepare();
    pizza.bake();
    pizza.cut();
    pizza.box();

    return pizza;
}

Pizza order

Pizza(String type) {
    Pizza pizza;

    if (type.equals("cheese")) {
        pizza = new CheesePizza();
    } else if (type.equals("greek")) {
        pizza = new GreekPizza();
    } else if (type.equals("pepperoni")) {
        pizza = new PepperoniPizza();
    }

    pizza.prepare();
    pizza.bake();
    pizza.cut();
    pizza.box();

    return pizza;
}
Example: Class Diagram

PizzaStore
  orderPizza()

Pizza
  name
  dough
  sauce
  toppings: 1..*
  prepare()
  bake()
  cut()
  box()
  get_name()
  toString()

CheesePizza

PepperoniPizza

GreekPizza

Problems with the design
Pizza order

Pizza(String type) {
    Pizza pizza;

    if (type.equals("cheese")) {
        pizza = new CheesePizza();
    } else if (type.equals("greek")) {
        pizza = new GreekPizza();
    } else if (type.equals("pepperoni")) {
        pizza = new PepperoniPizza();
    } else if (type.equals("sausage")) {
        pizza = new SausagePizza();
    } else if (type.equals("veggie")) {
        pizza = new VeggiePizza();
    }

    pizza.prepare();
pizza.bake();
pizza.cut();
pizza.box();
return pizza;
}
Pizza order

```java
Pizza orderPizza(String type) {
    Pizza pizza;

    if (type.equals("cheese")) {
        pizza = new CheesePizza();
    } else if (type.equals("greek")) {
        pizza = new GreekPizza();
    } else if (type.equals("pepperoni")) {
        pizza = new PepperoniPizza();
    } else if (type.equals("sausage")) {
        pizza = new SausagePizza();
    } else if (type.equals("veggie")) {
        pizza = new VeggiePizza();
    }

    pizza.prepare();
pizza.bake();
pizza.cut();
pizza.box();
return pizza;
}
```
Revised System

• Suppose that the Greek pizza is not popular and must be removed and two new pizzas, Clam and Veggie, must be added to the menu.
• Programming to implementation like makes such changes difficult.
• Creating a SimpleFactory to encapsulate the code that changes will make the design more flexible.
• Remove the code that creates a pizza – forms a factory.
public class SimplePizzaFactory {
    public Pizza createPizza(String type) {
        Pizza pizza;

        if (type.equals("cheese")) {
            pizza = new CheesePizza();
        } else if (type.equals("pepperoni")) {
            pizza = new PepperoniPizza();
        } else if (type.equals("sausage")) {
            pizza = new SausagePizza();
        } else if (type.equals("veggie")) {
            pizza = new VeggiePizza();
        }

        return pizza;
    }
}

Now orderPizza() is tidy
public class PizzaStore {
    SimplePizzaFactory factory;

    public PizzaStore(SimplePizzaFactory factory) {
        this.factory = factory;
    }

    public Pizza orderPizza(String type) {
        Pizza pizza;
        pizza = factory.createPizza(type);

        pizza.prepare();
        pizza.bake();
        pizza.cute();
        pizza.box();

        return pizza;
    }
}
Example: Revised Class Diagram

PizzaStore
  orderPizza()

SimpleFactory
  createPizza()

Pizza
  name
dough
  sauce
toppings: 1..*
  prepare()
bake()
cut()
box()
generate()
toString()

CheesePizza

ClamPizza

PepperoniPizza

VeggiePizza
Simple Factory

• Pull the code that builds the instances out and put it into a separate class
  – Identify the aspects of your application that vary and separate them from what stays the same
Simple Factory Pattern

**Client**
- orderProduct()

**SimpleFactory**
- createProduct()

**Product**
- productMethod()

**ConcreteProductA**
- 

**ConcreteProductB**
- 

**ConcreteProductC**
- 

16
Why would we do this?

• Multiple clients needing same types of object
• Ensure consistent object initialization
Example: System Revision Again

Franchises in different parts of the country are now adding their own special touches to the pizza. For example, customers at the franchise in New York like a thin base, with tasty sauce and little cheese. However, customers in Chicago prefer a thick base, rich sauce and a lot of cheese. Some franchises also cut the pizza slices differently (e.g. square)

You need to extend the system to cater for this.
The Factory Method

Definition:

• The factory Method Pattern defines an interface for creating an object, but lets the subclasses decide which class to instantiate.

• Factory Method lets a class defer instantiation to subclasses.
Creator Class

PizzaStore

createPizza()
orderPizza()

NYPizzaStore
createPizza()

ChicagoPizzaStore
createPizza()
Product Class

Pizza
- name
dough
- sauce
toppings: 1..*
- prepare()
- bake()
- cut()
- box()
- get_name()
- toString()

NYStyleCheesePizza
- NYStyleClamPizza
- NYStylePepperoniPizza
- NYStyleVeggiePizza

ChicagoStyleCheesePizza
- ChicagoStyleClamPizza
- ChicagoStylePepperoniPizza
- ChicagoStyleVeggiePizza
public abstract class PizzaStore {

    public Pizza orderPizza(String type) {
        Pizza pizza;
        pizza = createPizza(type);

        pizza.prepare();
        pizza.bake();
        pizza.cute();
        pizza.box();

        return pizza;
    }

    abstract Pizza createPizza(String type);
}
public class NYPizzaStore extends PizzaStore{
    Pizza createPizza(String item) {
        if (type.equals("cheese")) {
            return new NYStyleCheesePizza();
        } else if (type.equals("pepperoni")) {
            return new NYStylePepperoniPizza();
        } else if (type.equals("veggie")) {
            return new NYStyleVeggiePizza();
        } else return null;
    }
}


Factory Method Pattern

```
<<abstract>>
Product

<<abstract>>
Creator
  factoryMethod()
  operation()

<<abstract>>
ConcreteCreator
  factoryMethod()

<<abstract>>
ConcreteProduct

IS-A
HAS-A

abstract!

No more SimpleFactory class
Object creation is back in our class, but ... delegated to concrete classes
```
Applicability

• Use Factory Method when:
  – A class can’t anticipate the class of objects it must create
  – A class wants its subclasses to specify the objects it creates
  – Classes delegate to one of several helper subclasses, and you want to localize knowledge about which is the delegate
Participants

• Product
  – Defines interface of objects to be created

• ConcreteProduct
  – Implements Product interface

• Creator
  – Declares factory method
    • Returns object of type Product
    • May define default implementation
  – May call factory method to create Products
Consequences

- Eliminates need to bind creation code to specific subclasses
- May need to subclass Creator for each ConcreteProduct
- Provides hooks for subclasses
- Connects parallel class hierarchies
An Example

• Application framework
  – App subclasses “Application,” “Document”
  – Framework knows when to create Document, but Doesn’t know what type of Document to create
  – Factory Method moves knowledge about specific Document subclass out of framework
Guidelines

• No variable should hold a reference to a concrete class
• No class should derive from a concrete class
• No method should override an implemented method of its base classes.
Summary

• The creator gives you an interface for writing objects.
• The creator specifies the “factory method”, e.g. the createPizza method.
• Other methods generally included in the creator are methods that operate on the product produced by the creator, e.g. orderPizza.
• Only subclasses implement the factory method.
Summary

• **Simple Factory**
  - Use when you have only one family of objects

• **Factory Method Pattern**
  - Use when you have multiple families of objects

• **Abstract Factory (next time)**
  - Use when you have multiple families of object components
Thoughts...

• When you have code that instantiates concrete classes that may change, encapsulate the code that changes.
• The factory pattern allows you to encapsulate the behaviour of the instantiation.
• Duplication of code is prevented.
• Client code is decoupled from actual implementations.
• Programming to interface not implementation.