Creating Objects

- …it's more than just `new`
- A key tenet
  - Constructor usages often lead to unintended *coupling*
- Remember the note in the Strategy pattern?
  - When we say "`new`" to create a new object by calling a constructor, we're directly programming to an implementation
  - E.g., `Duck duck = new MallardDuck()`

We really WANT to use the interface…

But we're forced to create an instance of a concrete class!

```
Duck duck;
if (picnic){
  duck = new MallardDuck();
} else if (hunting) {
  duck = new DecoyDuck();
} else if (inBathTub) {
  duck = new RubberDuck();
}
```

- … especially when you think about the fact that things might change
  - E.g., you add a new type of duck and have to figure out when/how to instantiate it
  - And you make new kinds of Ducks in all different parts of your code

Open for Extension, Closed for Modification

- A key design goal:
  - Allow classes to be easily extended to incorporate new behavior
  - Without modifying existing code
    - Because everytime you modify it, you risk introducing new bugs
- This results in designs that are resilient to change but also flexible enough to accept new functionality to meet changing requirements

Information Hiding?

- What is it we're supposed to do with the stuff that changes?
- Encapsulate it!
- Practically, since the thing that's changing is **object creation**, we need an object that encapsulates object creation
- This object is called a **factory**
  - Then the `orderPizza` method is a **client** of the factory
    - Anytime it needs a pizza, it goes to the factory to request that one is created

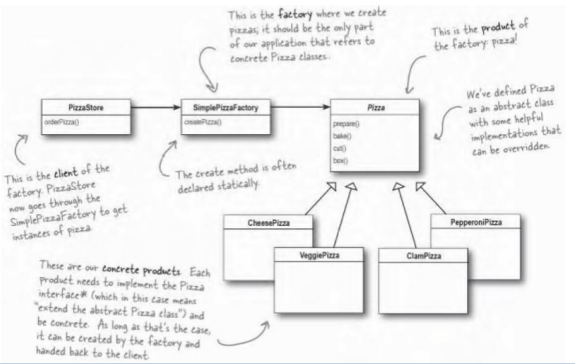
The Pizza Factory

```
public class SimplePizzaFactory {
  public Pizza createPizza(String type) {
    Pizza pizza = null;
    if (type.equals("cheese")) {
      pizza = new CheesePizza();
    } else if (type.equals("pepperoni") {
      pizza = new PepperoniPizza();
    } else if (type.equals("clam") {
      pizza = new ClamPizza();
    } else if (type.equals("veggie") {
      pizza = new VeggiePizza();
    }
    return pizza;
  }
}
```

- Well, that seems silly. All we did is copy the code out of the `orderPizza` method, but it still has all of the same problems…
- Does it?
- The `SimplePizzaFactory` might have lots of clients (not just the `orderPizza` method)
  - That was why we want to encapsulate the thing that changes!
- Also, the `orderPizza` method no longer needs to know anything at all about concrete `Pizza`s!

Simple Factory: Not Quite a Pattern

- But it is a **programming idiom**, and it's commonly used



Franchising the Pizza Store

- Now you want to spread your successful business
  - We want to localize the pizza making activities to the `PizzaStore` class
    - For quality control
  - But we want to give regional franchises the liberty to have their own pizza styles
- General framework:
  - Make the `PizzaStore` abstract
  - Put the `createPizza` method back in `PizzaStore`, but make it abstract
  - Create a `PizzaStore` subclass for every regional type of pizza

The Abstract Method

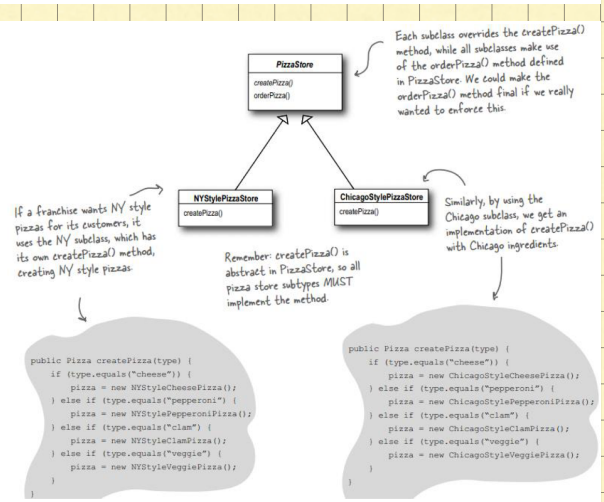
```
public abstract class PizzaStore {
  public Pizza orderPizza(String type) {
    Pizza pizza;
    pizza = createPizza(type);
    pizza.prepare();
    pizza.bake();
    pizza.cut();
    pizza.box();
    return pizza;
  }
  abstract Pizza createPizza(String type);
}
```

This is the "factory method"

如何设计这个方法将由子类决定.

Delegating to the Subclasses

- We've perfected the pizza ordering method, and it stays the same across all of the subclasses
- But now the regional franchises can differ in the style of pizza they make
  - E.g., thin crust in New York, thick crust in Chicago
- While the `orderPizza` method *looks* like it's defined in the `PizzaStore` class, this class is abstract
  - It can't *actually* do anything
  - So when it is executed, it is actually executing in the context of a concrete subclass
  - This context gets determined when the (abstract) method `createPizza` gets called



What's a Franchise Look Like?

- Bonus. The franchises get all of the benefits of the perfected PizzaStore ordering process
- All they have to do is define how to create pizzas!

```
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("clam") {
      return new NYStyleClamPizza();
    } else if (type.equals("veggie") {
      return new NYStyleVeggiePizza();
    } else return null;
  }
}
```

A Generic Factory Method

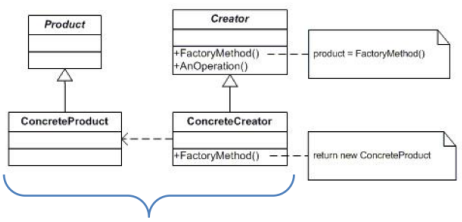
```
abstract Product factoryMethod(String type)
```

- A factory method is abstract so the subclasses are counted on to handle object creation
- The factory method isolates the client (the code in the superclass) from knowing what kind of concrete `Product` is created
- A factory method returns a `Product` that is typically used within methods defined in the superclass
- A factory method may be parameterized (or not) to select among several variations of a `Product`

Ordering a Pizza

1. First, the customer needs to get a NY PizzaStore:
   - `PizzaStore nyPizzaStore = new NYPizzaStore();`
2. Now the pizza store can accept our order
   - `nyPizzaStore.orderPizza("cheese");`
3. The `orderPizza` method calls the `createPizza` method
   - `Pizza pizza = createPizza("cheese");`
     - Remember the `createPizza` method is implemented in the subclass, so we're automagically getting a NY style cheese pizza here
4. The `orderPizza` method finishes preparing our pizza
   - `pizza.prepare();pizza.bake();pizza.cut();pizza.box();`
     - These methods are defined in the abstract `PizzaStore` class, which doesn't need to know which kind of pizza it is in order to follow the steps

- The whole thing requires some pizzas to tie everything together; you can check out the sample source code to see how it all fits



Parallel Class Hierarchies

The Factory Method Pattern

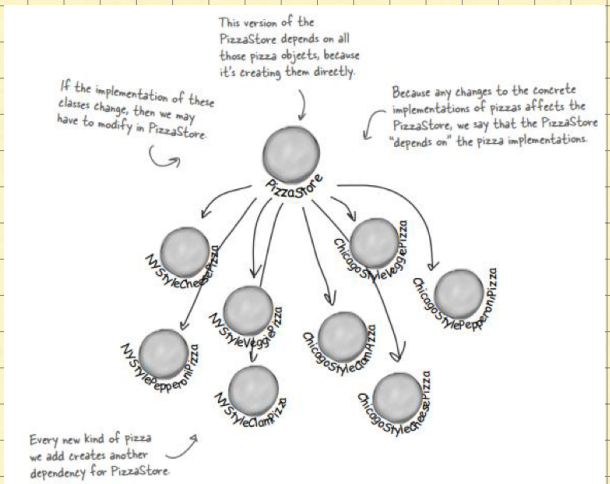
The Factory Method Pattern defines an interface for creating an object but lets subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses

工厂方法模式定义了一个创建对象的接口，由子类决定要实例化哪个类.

The Intuitive way of design

```
public class DependentPizzaStore{
    public Pizza createPizza (String style, String type){
        Pizza pizza = null;
        if (style.euqals("NY")){
            if (type.equals("cheese")){
                pizza = new NYStyleCheesePizza();
            } else if (type.equals("veggie")){
                pizza = new NYStyleVeggiePizza();
            } else if (type.equals("clam")){
                pizza = new NYStyleClamPizza();
            } else if (type.equals("pepperoni")){
                pizza = new NYStylePepperoniPizza();
            }
        } else if (style.equals("Chicago")){
            ...;
        }
        pizza.prepare();
        pizza.bake();
        ...;
        return pizza;
    }
}
```

What does this Look Like?

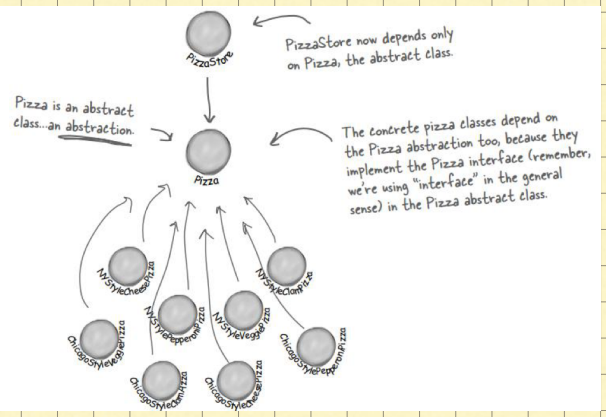


Another Design Principle

- This seems like a bad idea. We're definitely not encapsulating for change.
- If we change any of the concrete pizza classes, we have to change the PizzaStore because it **depends** on them
- Instead we should **depend upon abstractions. Do not depend upon concrete classes**
  - High level components should not depend on low-level components; instead, both should depend on abstractions
  - For example, in the previous pizza store, the store depended on all of the pizza types
  - Instead, the pizza store should depend on the abstract notion of Pizza, and the concrete pizza types should too
    - This is exactly what the Factory Method pattern we applied did!

依赖倒置原则：
要依赖抽象，不要依赖具体类.

Dependency Inversion



Guidelines that Help

- **NOT RULES TO FOLLOW**
- No variable should hold a reference to a concrete class
  - If you use new, you'll be holding a reference to a concrete class
  - Use a factory to get around that!
- No class should derive from a concrete class
  - If you do, you're depending on the concrete class
  - Instead, derive from an abstraction (like an interface or an abstract class)
- No method should override an implemented method of any of its base classes
  - If you do, then your base class wasn't really an abstraction
  - The methods implemented in the base class are meant to be shared by the derived classes

变量不能持有具体类的引用

类不能派生自具体类

不要覆盖基类中已实现的方法.