Lecture 04: The Factory Method Pattern

SE313, Software Design and Architecture

Damla Oguz

Chapter 4: The Factory Pattern

- "Get ready to bake some loosely coupled OO designs."
- There is more to making objects than just using the **new** operator.
- In this lecture, we will learn that instantiation is an activity that shouldn't always be done in public and can often lead to coupling problems.
- We will find out how Factory Pattern can help save us from embarrassing dependencies.
    - Simple Factory (not an actual design pattern)
    - Factory Method Pattern
    - Abstract Factory Pattern (next lecture)

A question

Q: We are not supposed to program to an implementation but every time we use **new**, that’s exactly what we are doing, right?

When you see “new”, think “concrete”

- When we use **new** we are certainly instantiating a concrete class, so that is definitely an implementation, not an interface.
- And it’s a good question; we’ve learned that tying our code to a concrete class can make it more fragile and less flexible.

```
Duck duck = new MallardDuck();
```

We want to use interfaces to keep code flexible.

But we have to create an instance of a concrete class!

When you see “new”, think “concrete” (cont.)

- When we have a whole set of related concrete classes, often we’re forced to write code like this:

```
Duck duck;

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

We have a bunch of different duck classes, and we don’t know until runtime which one we need to instantiate.

- Here we’ve got several concrete classes being instantiated, and the decision of which to instantiate is made at runtime depending on some set of conditions.

When you see “new”, think “concrete” (cont.)

- When it comes time for changes or extensions, we’ll have to reopen this code and examine what needs to be added (or deleted). Often this kind of code ends up in several parts of the application making maintenance and updates more difficult and error-prone.

```
Duck duck;

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

We have a bunch of different duck classes, and we don’t know until runtime which one we need to instantiate.

What’s wrong with “new”?

- Technically there’s nothing wrong with **new**, it’s a fundamental part of Java.
- The real culprit is CHANGE and how change impacts our use of **new**.
- If our code is written to an interface, then it will work with any new classes implementing that interface through polymorphism.
- However, when we have code that makes use of lots of concrete classes, we’re looking for trouble because that code may have to be changed as new concrete classes are added.
- So, in other words, our code will not be “**closed for modification**.”
- To **extend** it with new concrete types, we’ll have to reopen it.

What’s wrong with “new”? (cont.)

- So what can we do?
- Remember, our first principle deals with change and guides us to ***identify the aspects that vary and separate them from what stays the same***.

PizzaStore example

- Let’s say we have a pizza shop.
- PizzaStore class includes orderPizza() method that *determines* the appropriate type of pizza and then goes about *making* the pizza:

```
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();
      }

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

We're passing in the type of pizza to orderPizza.

- Based on the type of pizza, we instantiate the correct concrete class and assign it to the pizza instance variable.
- Note that each pizza here has to implement the Pizza interface.

- Once we have a Pizza, we prepare it, then we bake it, cut it and box it!
- Each Pizza subtype (CheesePizza, GreekPizza, etc.) knows how to prepare itself.

But the pressure is on to add more pizza types

- We want to add Clam Pizza and the Veggie Pizza to our menu.
- And we want to take off Greek Pizza from the menu.

```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("clam") {
            pizza = new ClamPizza();
        } else if (type.equals("veggie") {
            pizza = new VeggiePizza();
        }

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

This code is NOT closed for modification. If the Pizza Shop changes its pizza offerings, we have to get into this code and modify it.

This is what varies. As the pizza selection changes over time, we'll have to modify this code over and over.

Creation

This is what we expect to stay the same. We don't expect this code to change, just the pizzas it operates on.

Preperation

Encapsulating object creation

- We know what is varying and what isn’t.
- We can encapsulate the varying part (object creation) in a separate class.
- In other words, we can move the object creation out of the orderPizza() method.

Encapsulating object creation (cont.)

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



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

First we pull the object creation code out of the orderPizza Method

What's going to go here?

```
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();
}
```

Encapsulating object creation (cont.)

```
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();
}
```

Then we place that code in an object that is only going to worry about how to create pizzas. If any other object needs a pizza created, this is the object to come to.

Encapsulating object creation (cont.)

- **We've got a name for this new object: we call it a Factory.**
- Factories handle the details of object creation.
- Once we have SimplePizzaFactory, our orderPizza() method just becomes a client of that object. Any time it needs a pizza it asks the pizza factory to make one.
- Now the orderPizza() method just cares that it gets a pizza, which implements the Pizza interface so that it can call prepare(), bake(), cut(), and box().
- Let's implement a simple factory for the pizza store.

Building a simple 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;
    }
}
```

SimplePizzaFactory has one job in life: creating pizzas for its clients.

First we define a createPizza() method in the factory. This is the method all clients will use to instantiate new objects.

Here's the code we plucked out of the orderPizza() method.

A question

Q: What’s the advantage of this? It looks like we are just pushing the problem off to another object.

- SimplePizzaFactory may have many clients. We’ve only seen the orderPizza() method; however, there may be a PizzaShopMenu class that uses the factory to get pizzas for their current description and price.
- So, by encapsulating the pizza creating in one class, **we now have only one place to make modifications when the implementation changes**.
- Don’t forget, we are also just about to remove the concrete instantiations from our client code!

Reworking the PizzaStore class

- Now it’s time to fix up our client code.
- What we want to do is rely on the factory to create the pizzas for us.

```java
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.cut();
        pizza.box();
        return pizza;
    }

    // other methods here
}
```

Now we give PizzaStore a reference to a SimplePizzaFactory.

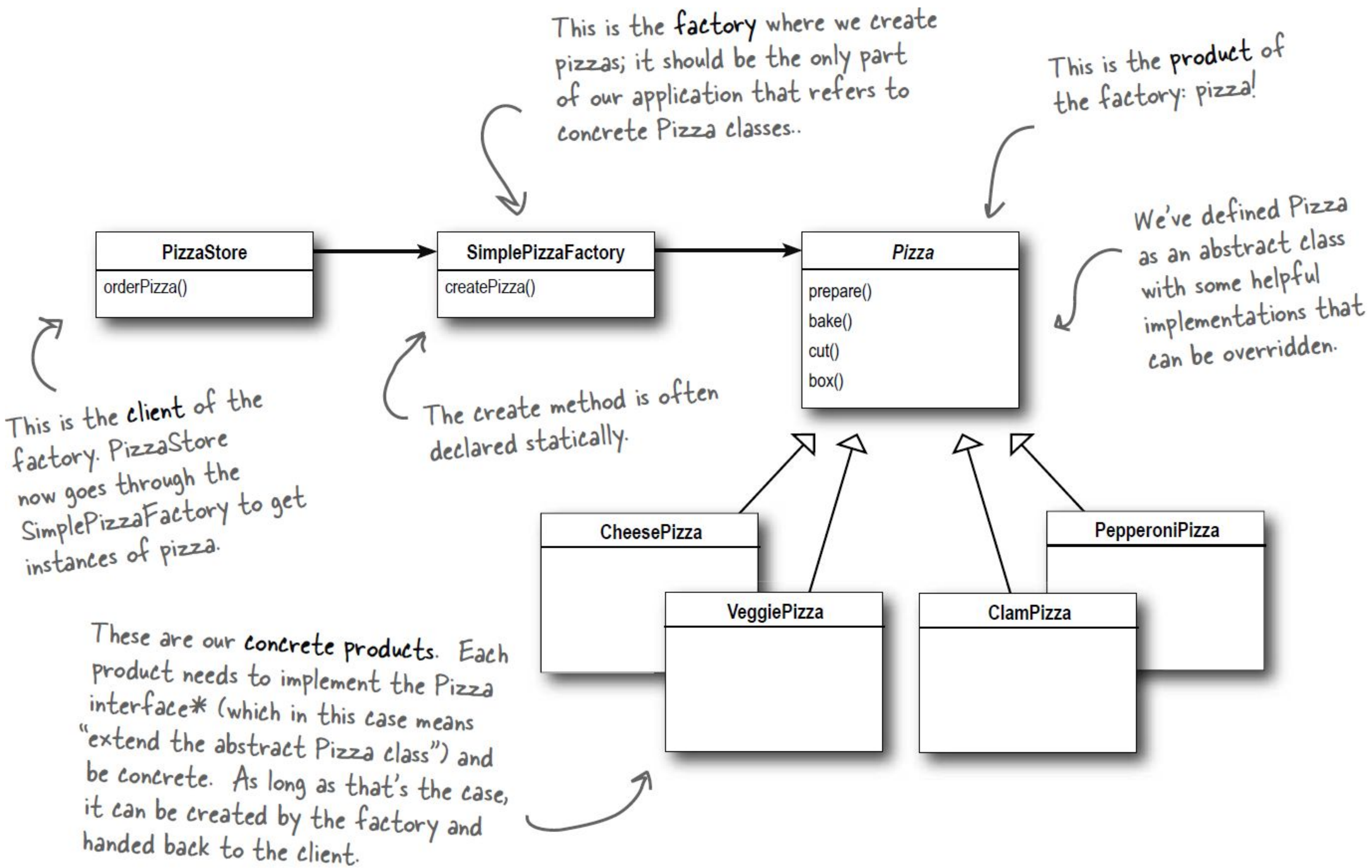
PizzaStore gets the factory passed to it in the constructor.

And the orderPizza() method uses the factory to create its pizzas by simply passing on the type of the order.

Notice that we've replaced the **new operator** with a create **method** on the factory object. No more concrete instantiations here!

The Simple Factory defined

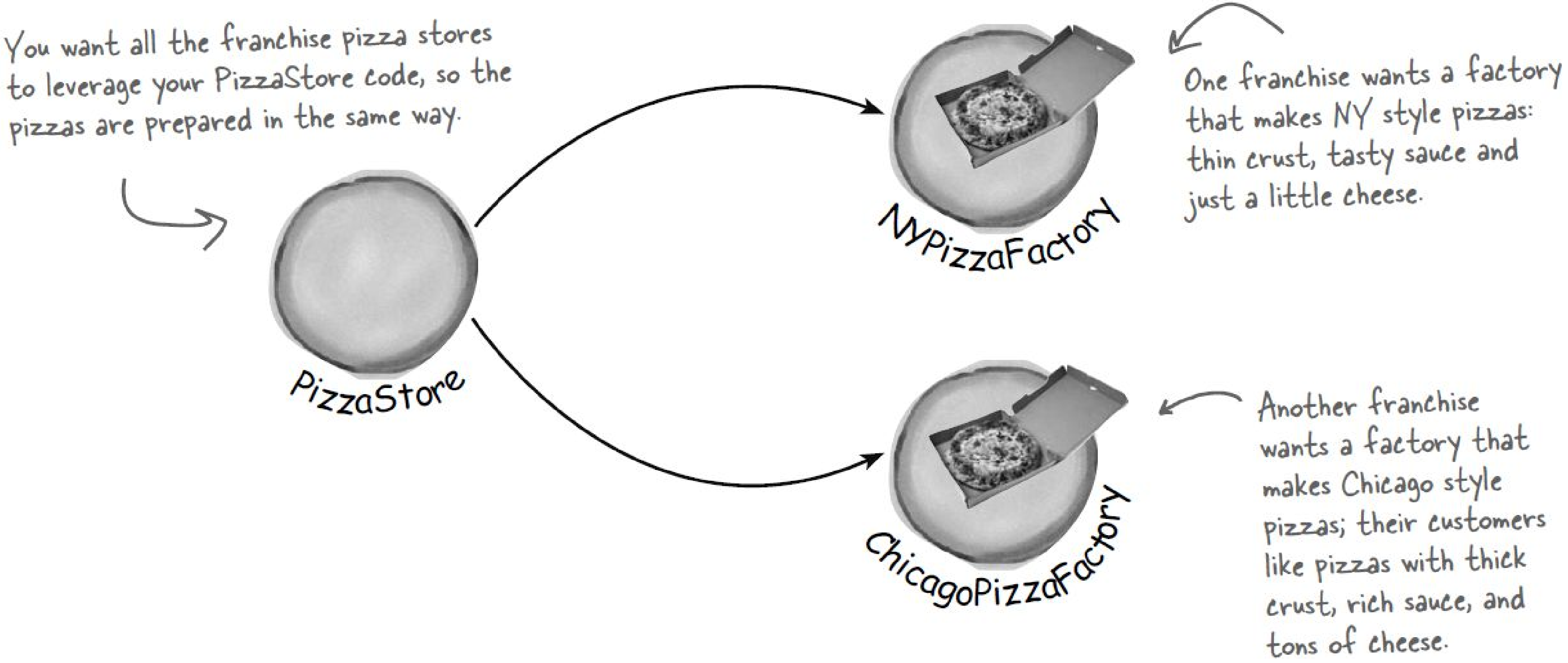
- The Simple Factory isn't actually a Design Pattern; it's more of a programming idiom.
- But it is commonly used.
- Let's take a look at the class diagram of our new Pizza Store.


This is the factory where we create pizzas; it should be the only part of our application that refers to concrete Pizza classes..
This is the product of the factory: pizza!
PizzaStore
orderPizza()
SimplePizzaFactory
createPizza()
Pizza
prepare()
bake()
cut()
box()
We've defined Pizza as an abstract class with some helpful implementations that can be overridden.
This is the client of the factory. PizzaStore now goes through the SimplePizzaFactory to get instances of pizza.
The create method is often declared statically.
CheesePizza
VeggiePizza
ClamPizza
PepperoniPizza
These are our concrete products. Each product needs to implement the Pizza interface* (which in this case means "extend the abstract Pizza class") and be concrete. As long as that's the case, it can be created by the factory and handed back to the client.

Franchising the pizza store

- As the franchiser, we want to ensure the quality of the franchise operations and so we want them to use our time-tested code.
- But what about regional differences? Each franchise might want to offer different styles of pizzas (New York, Chicago, and California, etc.) depending on where the franchise store is located.

Franchising the pizza store (cont.)



We’ve seen one approach...

- If we take out SimplePizzaFactory and create three different factories, NYPizzaFactory, ChicagoPizzaFactory and CaliforniaPizzaFactory, then we can just compose the PizzaStore with the appropriate factory and a franchise is good to go. That’s one approach.
- Let’s see what that would be like...

```
NYPizzaFactory nyFactory = new NYPizzaFactory();
PizzaStore nyStore = new PizzaStore(nyFactory);
nyStore.orderPizza("Veggie");
```

Here we create a factory for making NY style pizzas.

Then we create a PizzaStore and pass it a reference to the NY factory.

...and when we make pizzas, we get NY-styled pizzas.

```
ChicagoPizzaFactory chicagoFactory = new ChicagoPizzaFactory();
PizzaStore chicagoStore = new PizzaStore(chicagoFactory);
chicagoStore.orderPizza("Veggie");
```

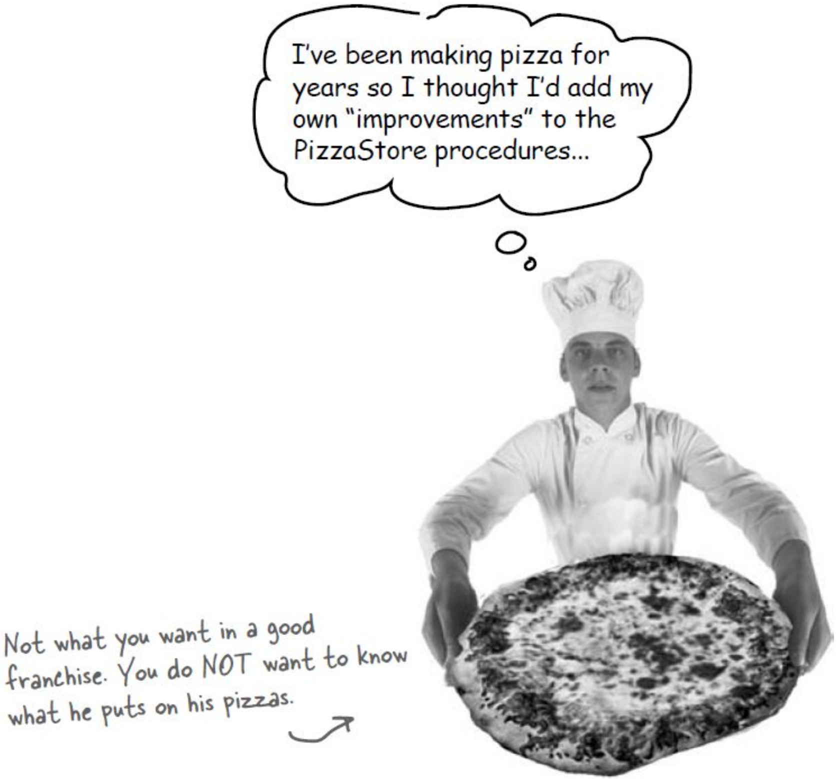
Likewise for the Chicago pizza stores: we create a factory for Chicago pizzas and create a store that is composed with a Chicago factory. When we make pizzas, we get the Chicago flavored ones

But we’d like a little more quality control...

- So we test marketed the SimpleFactory idea and we found that the franchises were using our factory to create pizzas.
- But they started to employ their own procedures for the rest of the process:
  - they’d bake things a little differently, they’d cut the pizza in different shapes and they’d use third-party boxes.

But you'd like a little more quality control… (cont.)

- What we'd really like to do is create a framework that ties the store and the pizza creation together, yet still allows things to remain flexible.
- In our early code, before the SimplePizzaFactory, we had the pizza-making code tied to the PizzaStore, but it wasn't flexible.



A framework for the pizza store

- There is a way to localize all the pizza making activities to the PizzaStore class, and yet give the franchises freedom to have their own regional style.
- What we're going to do is put the createPizza() method back into PizzaStore, but this time as an **abstract method**, and then create a PizzaStore subclass for each regional style.

PizzaStore is now abstract.

```
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);
}
```

Now createPizza is back to being a call to a method in the PizzaStore rather than on a factory object.

All this looks just the same...

Now we've moved our factory object to this method.

Our "factory method" is now abstract in PizzaStore.

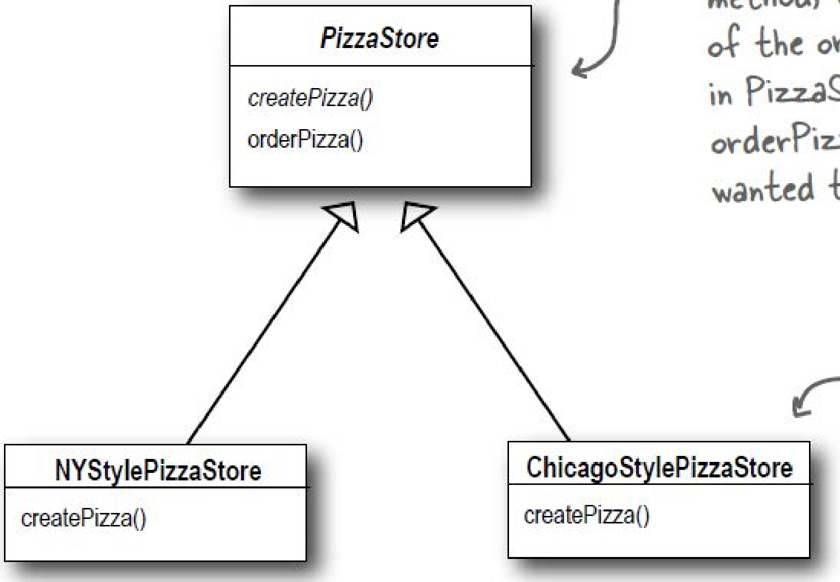
A framework for the pizza store (cont.)

- Now we’ve got a store waiting for subclasses; we’re going to have a subclass for each regional type (NYPizzaStore, ChicagoPizzaStore, CaliforniaPizzaStore) and each subclass is going to make the decision about what makes up a pizza.
- Let’s take a look at how this is going to work.

Allowing the subclasses to decide

- We want to ensure that orderPizza() method is consistent across all franchises.
- What varies among the regional PizzaStores is the style of pizzas they make
  - New York Pizza has thin crust, Chicago Pizza has thick, and so on.
- We are going to push all these variations into the createPizza() method and make it responsible for creating the right kind of pizza.
- The way we do this is by letting **each subclass of PizzaStore define what the createPizza() method looks like.**

Each subclass overrides the createPizza() method, while all subclasses make use of the orderPizza() method defined in PizzaStore. We could make the orderPizza() method final if we really wanted to enforce this.



If a franchise wants NY style pizzas for its customers, it uses the NY subclass, which has its own createPizza() method, creating NY style pizzas.

Similarly, by using the Chicago subclass, we get an implementation of createPizza() with Chicago ingredients.

If a franchise wants NY style pizzas for its customers, it uses the NY subclass, which has its own createPizza() method, creating NY style pizzas.

NYStylePizzaStore
createPizza()

ChicagoStylePizzaStore
createPizza()

Similarly, by using the Chicago subclass, we get an implementation of createPizza() with Chicago ingredients.

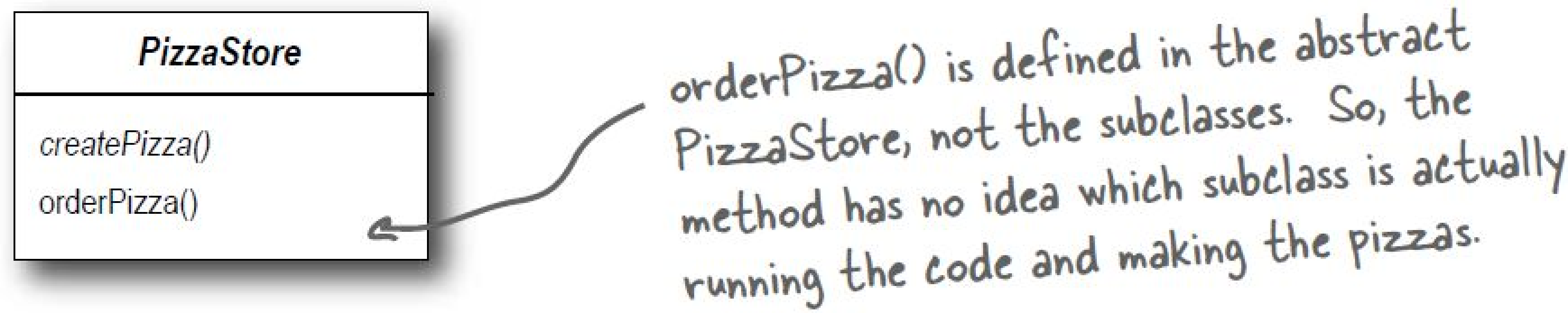
Remember: createPizza() is abstract in PizzaStore, so all pizza store subtypes MUST implement the method.

```
public Pizza createPizza(type) {
    if (type.equals("cheese")) {
        pizza = new NYStyleCheesePizza();
    } else if (type.equals("pepperoni") {
        pizza = new NYStylePepperoniPizza();
    } else if (type.equals("clam") {
        pizza = new NYStyleClamPizza();
    } else if (type.equals("veggie") {
        pizza = new NYStyleVeggiePizza();
    }
}
```

```
public Pizza createPizza(type) {
    if (type.equals("cheese")) {
        pizza = new ChicagoStyleCheesePizza();
    } else if (type.equals("pepperoni") {
        pizza = new ChicagoStylePepperoniPizza();
    } else if (type.equals("clam") {
        pizza = new ChicagoStyleClamPizza();
    } else if (type.equals("veggie") {
        pizza = new ChicagoStyleVeggiePizza();
    }
}
```

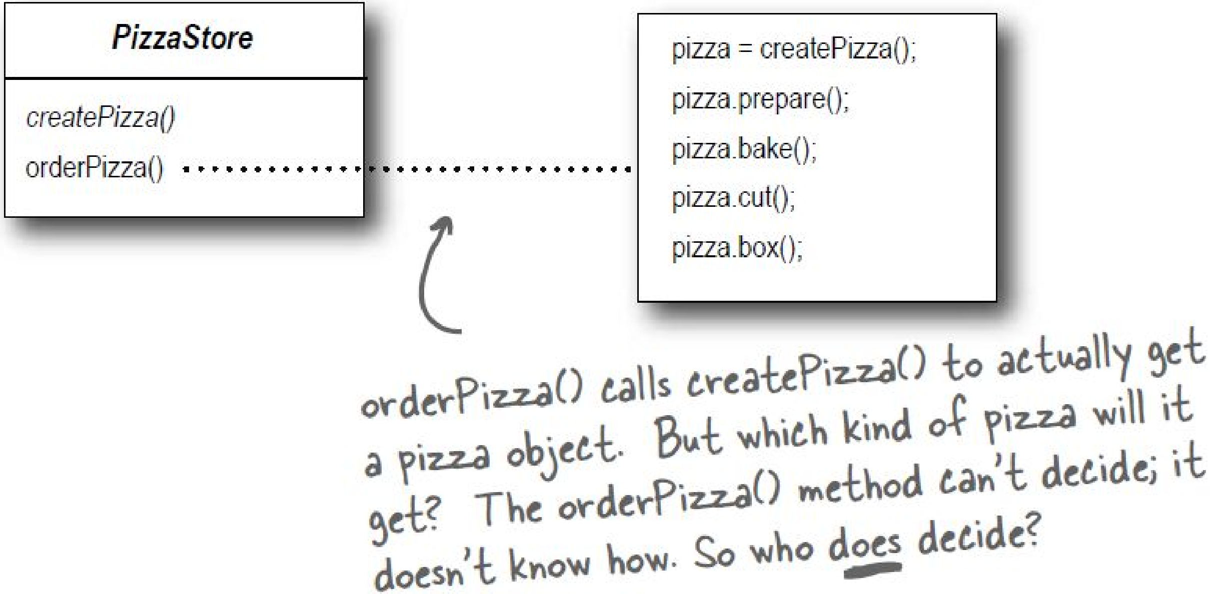
How do subclasses decide?

- Think about it from the point of view of the PizzaStore's orderPizza() method: it is defined in the abstract PizzaStore, but concrete types are only created in the subclasses.



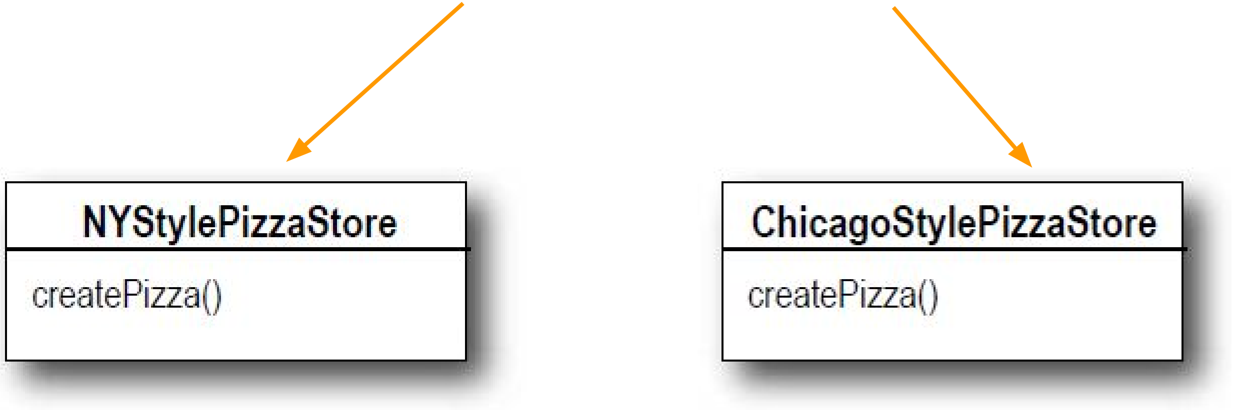
How do subclasses decide? (cont.)

- orderPizza() method does a lot of things with a Pizza object (like prepare, bake, cut, box), but because Pizza is abstract, orderPizza() has no idea what real concrete classes are involved.



How do subclasses decide? (cont.)

- When orderPizza() calls createPizza(), one of our subclasses will be called into action to create a pizza.
- Which kind of pizza will be made? Well, that's decided by the choice of pizza store we order from, NYStylePizzaStore or ChicagoStylePizzaStore.



How do subclasses decide? (cont.)

- There isn't a real-time decision that subclasses make, but from the perspective of orderPizza(), if we chose a NYStylePizzaStore, that subclass gets to determine which pizza is made.
- So the subclasses aren't really "deciding" – it was us who decided by choosing which store we wanted – but they do determine which kind of pizza gets made.

Let’s make a PizzaStore

- Being a franchise has its benefits. You get all the PizzaStore functionality for free.
    - All the regional stores need to do is subclass PizzaStore and supply a createPizza() method that implement their style of Pizza.
- We’ll take care of the big three pizza styles for the franchisees.
- Here’s the New York regional style…

createPizza() returns a Pizza, and the subclass is fully responsible for which concrete Pizza it instantiates

The NYPizzaStore extends PizzaStore, so it inherits the orderPizza() method (among others).

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

We've got to implement createPizza(), since it is abstract in PizzaStore.

Here's where we create our concrete classes. For each type of Pizza we create the NY style.

Declaring a factory 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;
    }

    protected abstract Pizza createPizza(String type);

    // other methods here
}
```

The subclasses of PizzaStore handle object instantiation for us in the createPizza() method.

NYStylePizzaStore
createPizza()

ChicagoStylePizzaStore
createPizza()

All the responsibility for instantiating Pizzas has been moved into a **method** that acts as a **factory**.

Code up close

- A factory method handles object creation and encapsulates it in a subclass. This decouples the client code in the superclass from the object creation code in the subclass.

A factory method may be parameterized (or not) to select among several variations of a product.

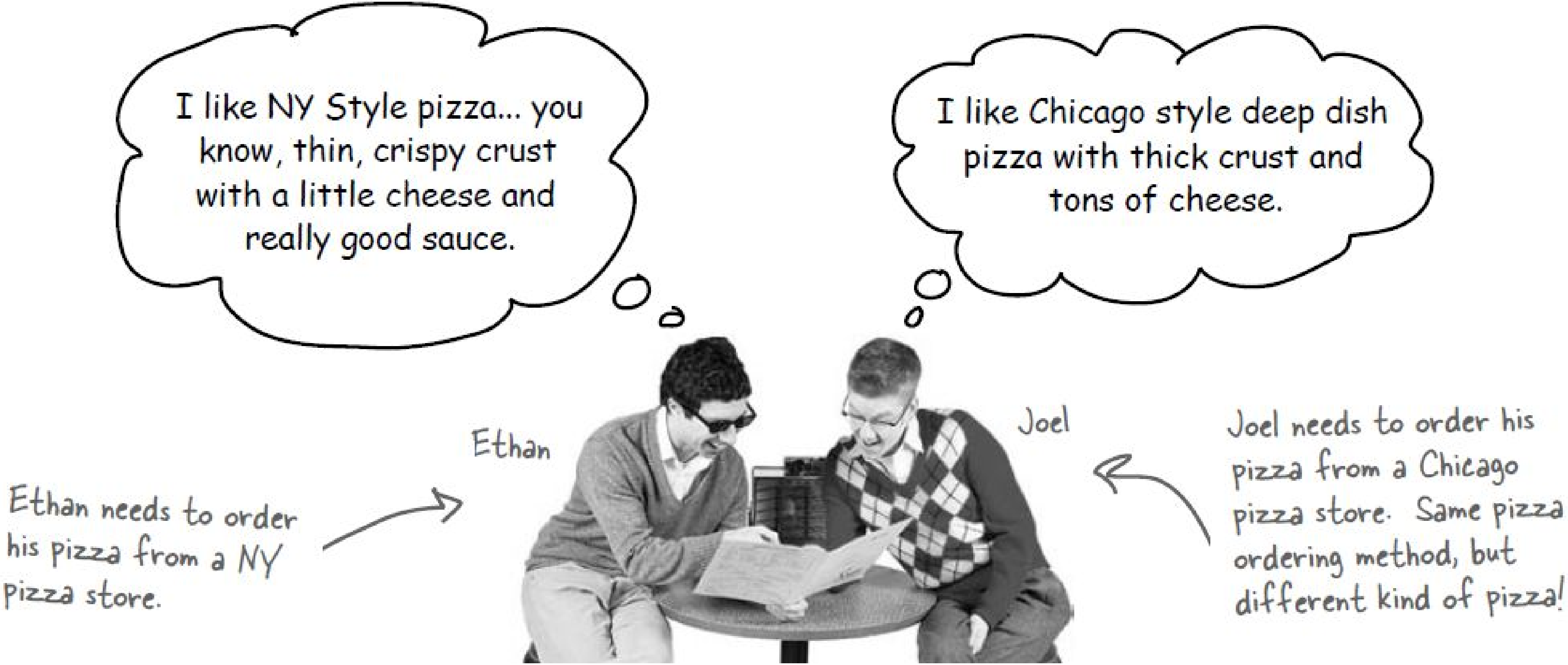
```
abstract Product factoryMethod(String type)
```

A factory method is abstract so the subclasses are counted on to handle object creation.

A factory method returns a Product that is typically used within methods defined in the superclass.

A factory method isolates the client (the code in the superclass, like orderPizza()) from knowing what kind of concrete Product is actually created.

Let's see how it works: ordering pizzas with the pizza factory method



So how do they order?

1. First, Joel and Ethan need an instance of a PizzaStore. Joel needs to instantiate a ChicagoPizzaStore and Ethan needs a NYPizzaStore.

2. With a PizzaStore in hand, both Ethan and Joel call the orderPizza() method and pass in the type of pizza they want (cheese, veggie, and so on).

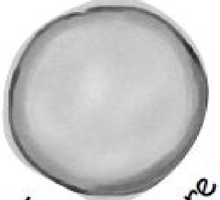
3. To create the pizzas, the createPizza() method is called, which is defined in the two subclasses NYPizzaStore and ChicagoPizzaStore. As we defined them, the NYPizzaStore instantiates a NY style pizza, and the ChicagoPizzaStore instantiates Chicago style pizza. In either case, the Pizza is returned to the orderPizza() method.

4. The orderPizza() method has no idea what kind of pizza was created, but it knows it is a pizza and it prepares, bakes, cuts, and boxes it for Ethan and Joel.

❶ Let's follow Ethan's order: first we need a NY PizzaStore:

```
PizzaStore nyPizzaStore = new NYPizzaStore();
```

Creates a instance of NYPizzaStore.



nyPizzaStore

❷ Now that we have a store, we can take an order:

```
nyPizzaStore.orderPizza("cheese");
```

The orderPizza() method is called on the nyPizzaStore instance (the method defined inside PizzaStore runs).

createPizza("cheese")

❸ The orderPizza() method then calls the createPizza() method:

```
Pizza pizza  = createPizza("cheese");
```

Remember, createPizza(), the factory method, is implemented in the subclass. In this case it returns a NY Cheese Pizza.

Pizza

3 **The orderPizza() method then calls the createPizza() method:**

```
Pizza pizza  = createPizza("cheese");
```

Remember, createPizza(), the factory method, is implemented in the subclass. In this case it returns a NY Cheese Pizza.

4 **Finally we have the unprepared pizza in hand and the orderPizza() method finishes preparing it:**

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

All of these methods are defined in the specific pizza returned from the factory method createPizza(), defined in the NYPizzaStore.

The orderPizza() method gets back a Pizza, without knowing exactly what concrete class it is.

We’re just missing one thing: PIZZA!

```
public abstract class Pizza {
    String name;
    String dough;
    String sauce;
    ArrayList <String> toppings = new ArrayList <String>();

    void prepare() {
        System.out.println("Preparing " + name);
        System.out.println("Tossing dough...");
        System.out.println("Adding sauce...");
        System.out.println("Adding toppings: ");
        for ( String topping : toppings) {
            System.out.println("   " + topping);
        }
    }
```

Each Pizza has a name, a type of dough, a type of sauce, and a set of toppings.

Preparation follows a number of steps in a particular sequence.

We’re just missing one thing: PIZZA! (cont.)

```
void bake() {
    System.out.println(“Bake for 25 minutes at 350”);
}

void cut() {
    System.out.println(“Cutting the pizza into diagonal slices”);
}

void box() {
    System.out.println(“Place pizza in official PizzaStore box”);
}

public String getName() {
    return name;
}
```

The abstract class provides some basic defaults for baking, cutting and boxing.

Now we just need some concrete subclasses

```
public class NYStyleCheesePizza extends Pizza {
    public NYStyleCheesePizza() {
        name = "NY Style Sauce and Cheese Pizza";
        dough = "Thin Crust Dough";
        sauce = "Marinara Sauce";

        toppings.add("Grated Reggiano Cheese");
    }
}
```

The NY Pizza has its own marinara style sauce and thin crust.

And one topping, reggiano cheese!

Now we just need some concrete subclasses (cont.)

```
public class ChicagoStyleCheesePizza extends Pizza {
    public ChicagoStyleCheesePizza() {
        name = "Chicago Style Deep Dish Cheese Pizza";
        dough = "Extra Thick Crust Dough";
        sauce = "Plum Tomato Sauce";

        toppings.add("Shredded Mozzarella Cheese");
    }

    void cut() {
        System.out.println("Cutting the pizza into square slices");
    }
}
```

The Chicago Pizza uses plum tomatoes as a sauce along with extra thick crust.

The Chicago style deep dish pizza has lots of mozzarella cheese!

The Chicago style pizza also overrides the cut() method so that the pieces are cut into squares.

Time for some pizzas!

```
public class PizzaTestDrive {

    public static void main(String[] args) {
        PizzaStore nyStore = new NYPizzaStore();
        PizzaStore chicagoStore = new ChicagoPizzaStore();

        Pizza pizza = nyStore.orderPizza("cheese");
        System.out.println("Ethan ordered a " + pizza.getName() + "\n");

        pizza = chicagoStore.orderPizza("cheese");
        System.out.println("Joel ordered a " + pizza.getName() + "\n");
    }
}
```

First we create two different stores.

Then use one one store to make Ethan's order.

And the other for Joel's.

```
%java PizzaTestDrive

Preparing NY Style Sauce and Cheese Pizza
Tossing dough...
Adding sauce...
Adding toppings:
   Grated Regiano cheese
Bake for 25 minutes at 350
Cutting the pizza into diagonal slices
Place pizza in official PizzaStore box
Ethan ordered a NY Style Sauce and Cheese Pizza

Preparing Chicago Style Deep Dish Cheese Pizza
Tossing dough...
Adding sauce...
Adding toppings:
   Shredded Mozzarella Cheese
Bake for 25 minutes at 350
Cutting the pizza into square slices
Place pizza in official PizzaStore box
Joel ordered a Chicago Style Deep Dish Cheese Pizza
```

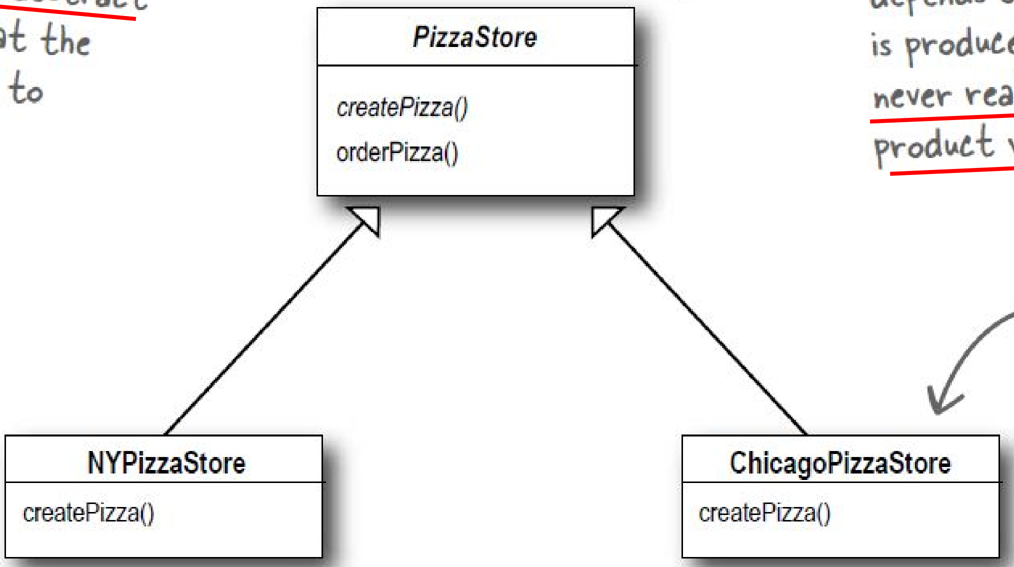
Both pizzas get prepared, the toppings added, and the pizzas baked, cut and boxed. Our superclass never had to know the details, the subclass handled all that just by instantiating the right pizza.

Meet the Factory Method Pattern

- All factory patterns encapsulate object creation.
- The Factory Method Pattern encapsulates object creation by letting subclasses decide what objects to create.
- The players in this pattern:
  - The Creator Classes
  - The Product Classes
- Let’s check out their class diagrams respectively.

This is our abstract creator class. It defines an abstract factory method that the subclasses implement to produce products.

Often the creator contains code that depends on an abstract product, which is produced by a subclass. The creator never really knows which concrete product was produced.



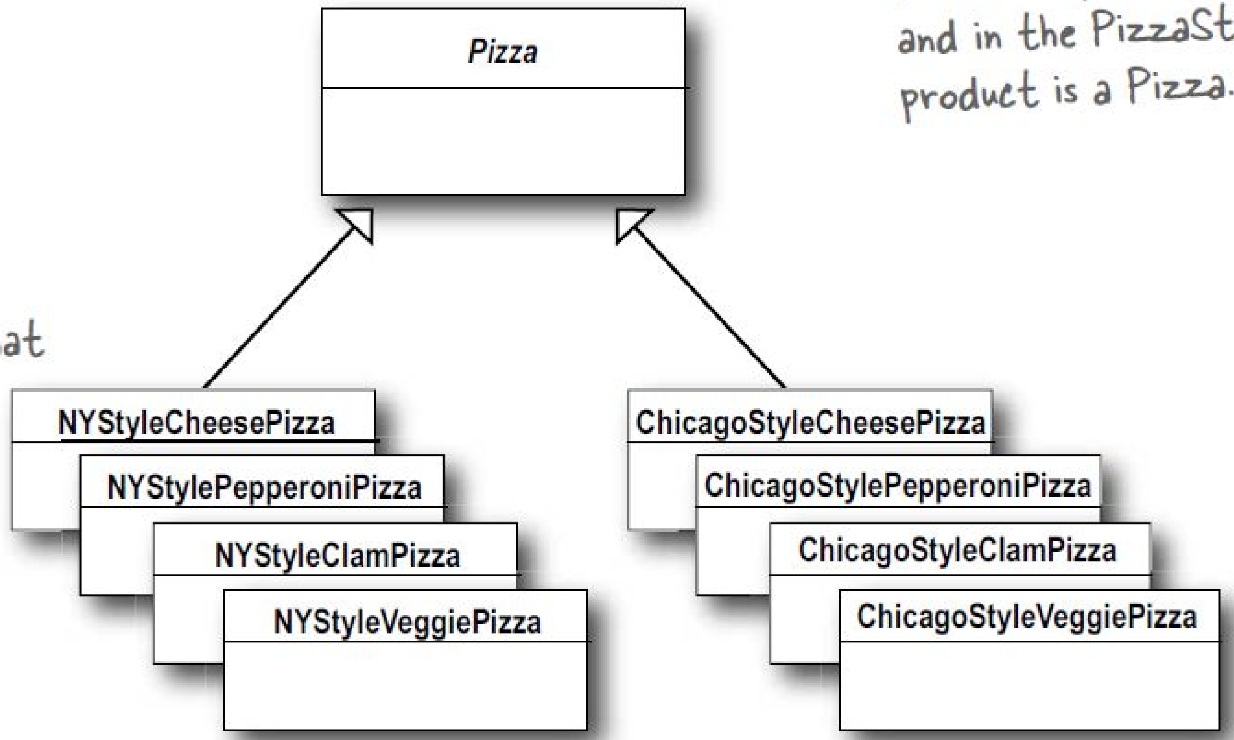
Since each franchise gets its own subclass of PizzaStore, it's free to create its own style of pizza by implementing createPizza().

The createPizza() method is our factory method. It Produces products.

Classes that produce products are called concrete creators

Factories produce products, and in the PizzaStore, our product is a Pizza.

These are the concrete products – all the pizzas that are produced by our stores.



Factory Method Pattern defined

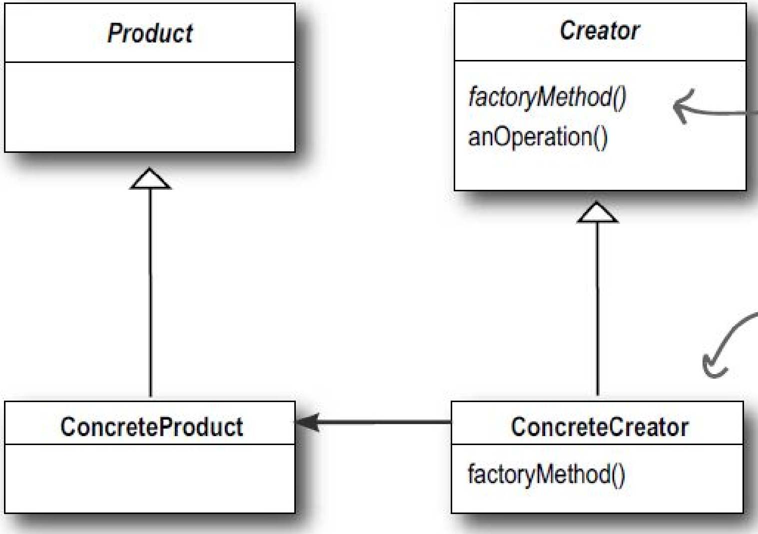
> **The Factory Method Pattern** defines an interface for creating an object, but lets subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

Because the creator class is written without knowledge of the actual products that will be created, which is decided purely by the choice of the subclass that is used.

The Creator is a class that contains the implementations for all of the methods to manipulate products, except for the factory method.

All products must implement the same interface so that the classes which use the products can refer to the interface, not the concrete class.

The abstract factoryMethod() is what all Creator subclasses must implement.



The ConcreteCreator implements the factoryMethod(), which is the method that actually produces products.

The ConcreteCreator is responsible for creating one or more concrete products. It is the only class that has the knowledge of how to create these products.

Looking at object dependencies

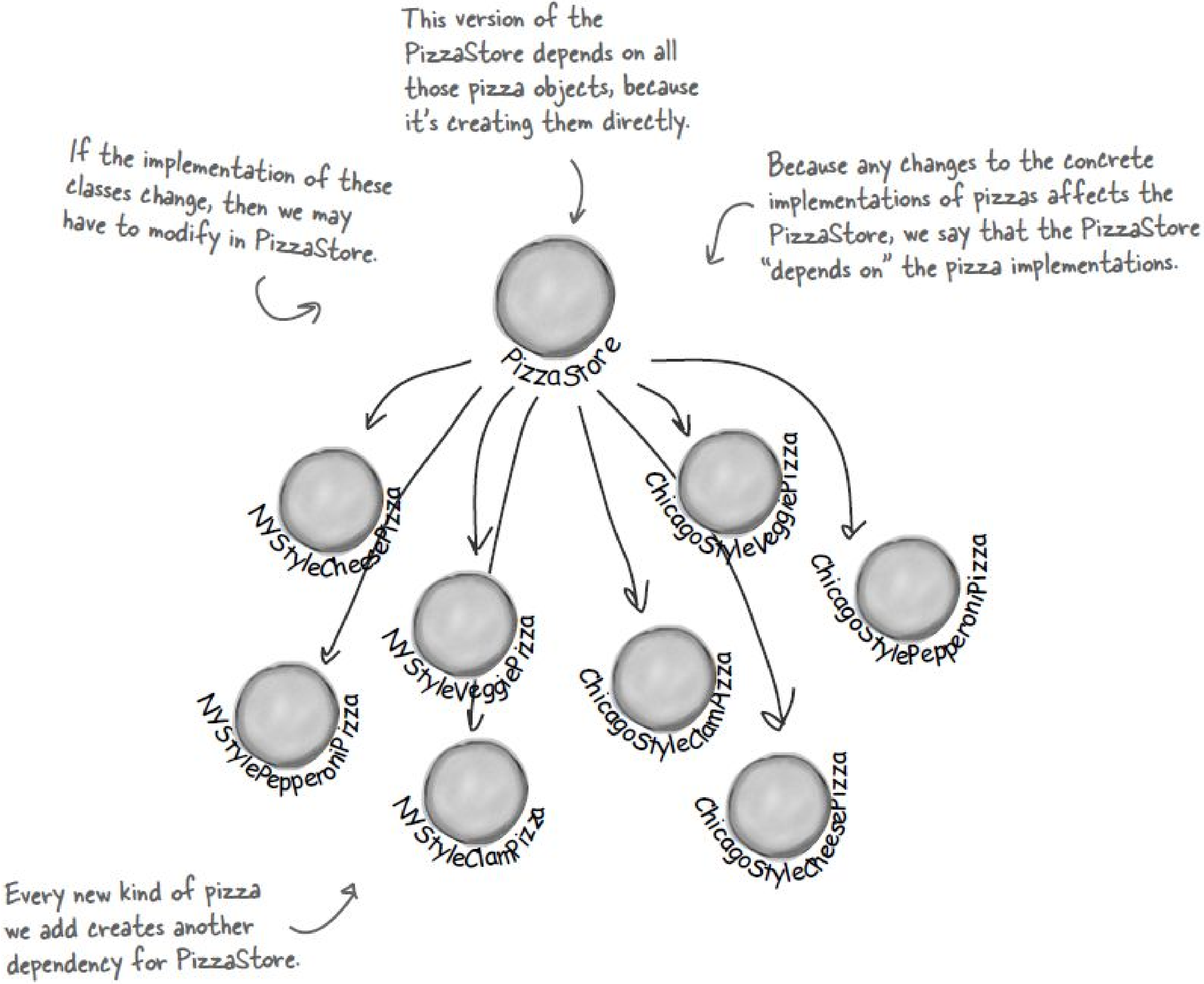
- When you directly instantiate an object, you are depending on its concrete class. Take a look at our very Dependent PizzaStore in the next page.
- It creates all the pizza objects right in the PizzaStore class instead of delegating to a factory.

```java
public class DependentPizzaStore {

    public Pizza createPizza(String style, String type) {
        Pizza pizza = null;
        if (style.equals(“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”)) {
            if (type.equals(“cheese”)) {
                pizza = new ChicagoStyleCheesePizza();
            } else if (type.equals(“veggie”)) {
                pizza = new ChicagoStyleVeggiePizza();
            } else if (type.equals(“clam”)) {
                pizza = new ChicagoStyleClamPizza();
            } else if (type.equals(“pepperoni”)) {
                pizza = new ChicagoStylePepperoniPizza();
            }
        } else {
            System.out.println(“Error: invalid type of pizza”);
            return null;
        }
        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();
        return pizza;
    }
```

Handles all the NY style pizzas

Handles all the Chicago style pizzas


This version of the PizzaStore depends on all those pizza objects, because it's creating them directly.
If the implementation of these classes change, then we may have to modify in PizzaStore.
Because any changes to the concrete implementations of pizzas affects the PizzaStore, we say that the PizzaStore "depends on" the pizza implementations.
PizzaStore
NYStyleCheesePizza
NYStyleVeggiePizza
ChicagoStyleVeggiePizza
ChicagoStylePepperoniPizza
NYStylePepperoniPizza
NYStyleClamPizza
ChicagoStyleClamPizza
ChicagoStyleCheesePizza
Every new kind of pizza we add creates another dependency for PizzaStore.

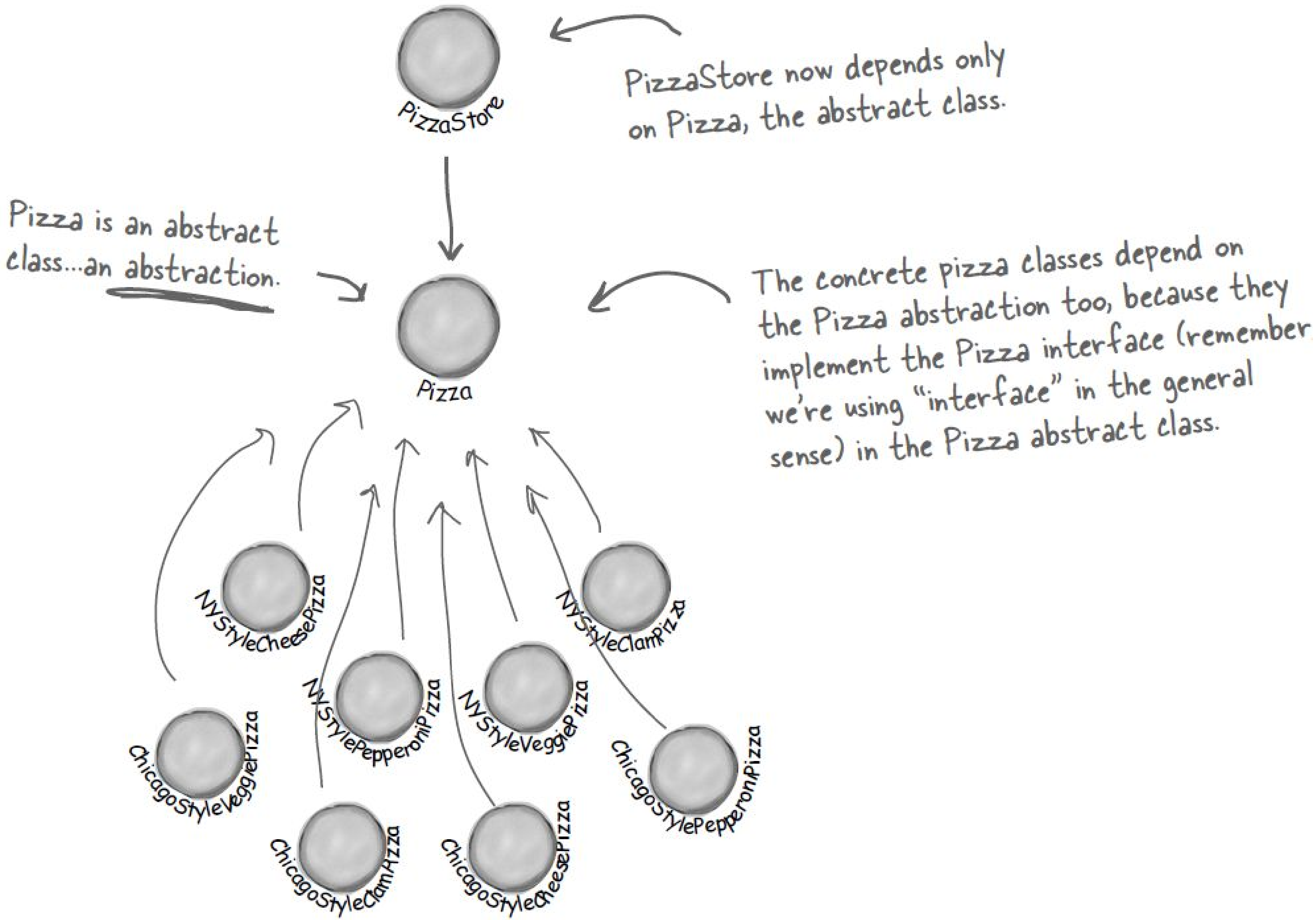
The Dependency Inversion Principle

Design Principle: **Depend upon abstractions. Do not depend upon concrete classes.**

- It suggests that our high-level components should not depend on our low-level components; rather, they should both depend on abstractions.
- PizzaStore is our “high-level component” and the pizza implementations are our “low-level components,” and clearly the PizzaStore is dependent on the concrete pizza classes.
- This principle tells us we should prefer to depend on abstractions rather than concrete classes.

Applying the Principle

- The main problem with the very Dependent PizzaStore is that it depends on every type of pizza because it actually instantiates concrete types in its createPizza() method.
- After we've applied the Factory Method, our diagram looks like this...


PizzaStore
PizzaStore now depends only on Pizza, the abstract class.
Pizza is an abstract class...an abstraction.
Pizza
The concrete pizza classes depend on the Pizza abstraction too, because they implement the Pizza interface (remember we're using "interface" in the general sense) in the Pizza abstract class.
NYStyleCheesePizza
NYStyleClamPizza
NYStylePepperoniPizza
NYStyleVeggiePizza
ChicagoStyleVeggiePizza
ChicagoStylePepperoniPizza
ChicagoStyleClamPizza
ChicagoStyleCheesePizza

Applying the Principle (cont.)

- After applying the Factory Method, you’ll notice that our high-level component, the PizzaStore, and our low-level components, the pizzas, both depend on Pizza, the abstraction.
- Factory Method is one of the more powerful techniques for adhering to the Dependency Inversion Principle.

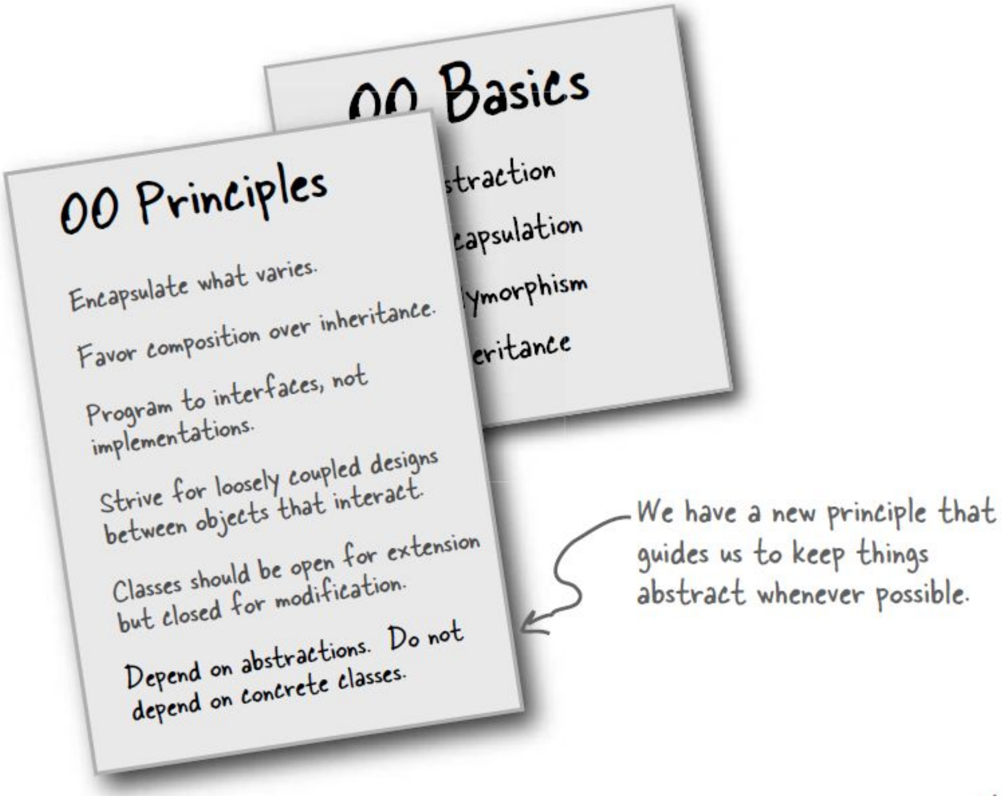
A few guidelines to help you follow the principle...

- 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 derive from a concrete class, you're depending on a concrete class. 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 override an implemented method, then your base class wasn't really an abstraction to start with. Those methods implemented in the base class are meant to be shared by all your subclasses.

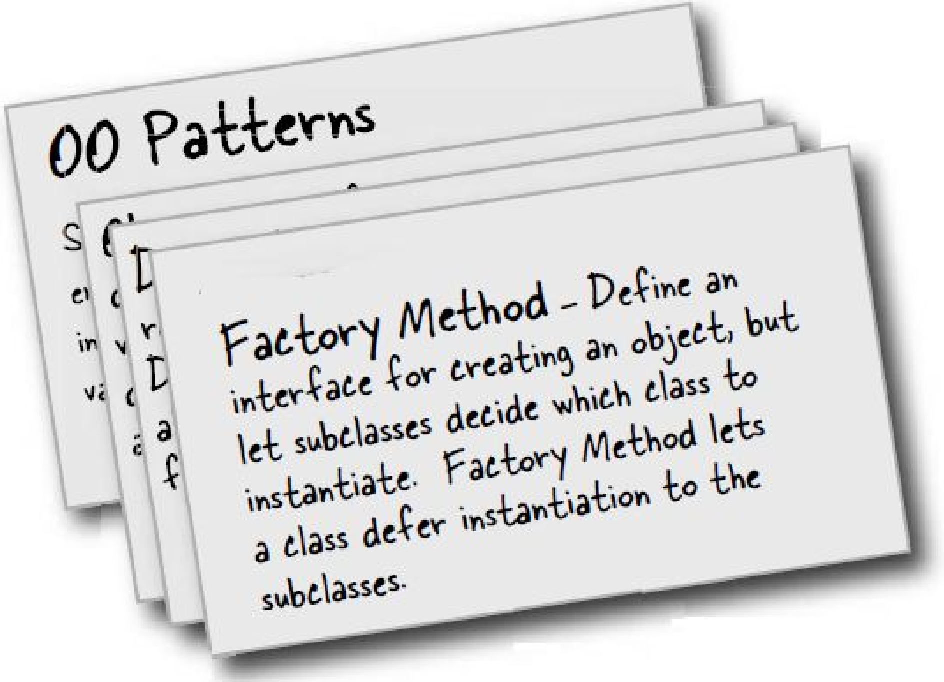
About the guidelines

- Like many of our principles, this is a guideline you should strive for, rather than a rule you should follow all the time.
- But, if you internalize these guidelines and have them in the back of your mind when you design, you'll know when you are violating the principle and you'll have a good reason for doing so.

Tools for your Design Toolbox



Tools for your Design Toolbox (cont.)



Review: Access modifiers

- The following table shows the access to members permitted by each modifier.
- The first data column indicates whether the class itself has access to the member defined by the access level. As you can see, a class always has access to its own members.

Access Levels

Modifier	Class	Package	Subclass	World
`public`	Y	Y	Y	Y
`protected`	Y	Y	Y	N
no modifier	Y	Y	N	N
`private`	Y	N	N	N

Review: Access modifiers (cont.)

- The second column indicates whether classes in the same package as the class (regardless of their parentage) have access to the member.
- The third column indicates whether subclasses of the class declared outside this package have access to the member.
- The fourth column indicates whether all classes have access to the member.

Access Levels

Modifier	Class	Package	Subclass	World
`public`	Y	Y	Y	Y
`protected`	Y	Y	Y	N
no modifier	Y	Y	N	N
`private`	Y	N	N	N

References

Material in this lecture is taken from Freeman, E., Robson, E., Bates, B., & Sierra, K., *Head First Design Patterns: A Brain-Friendly Guide*, O'Reilly Media, Inc., 2004.

https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html