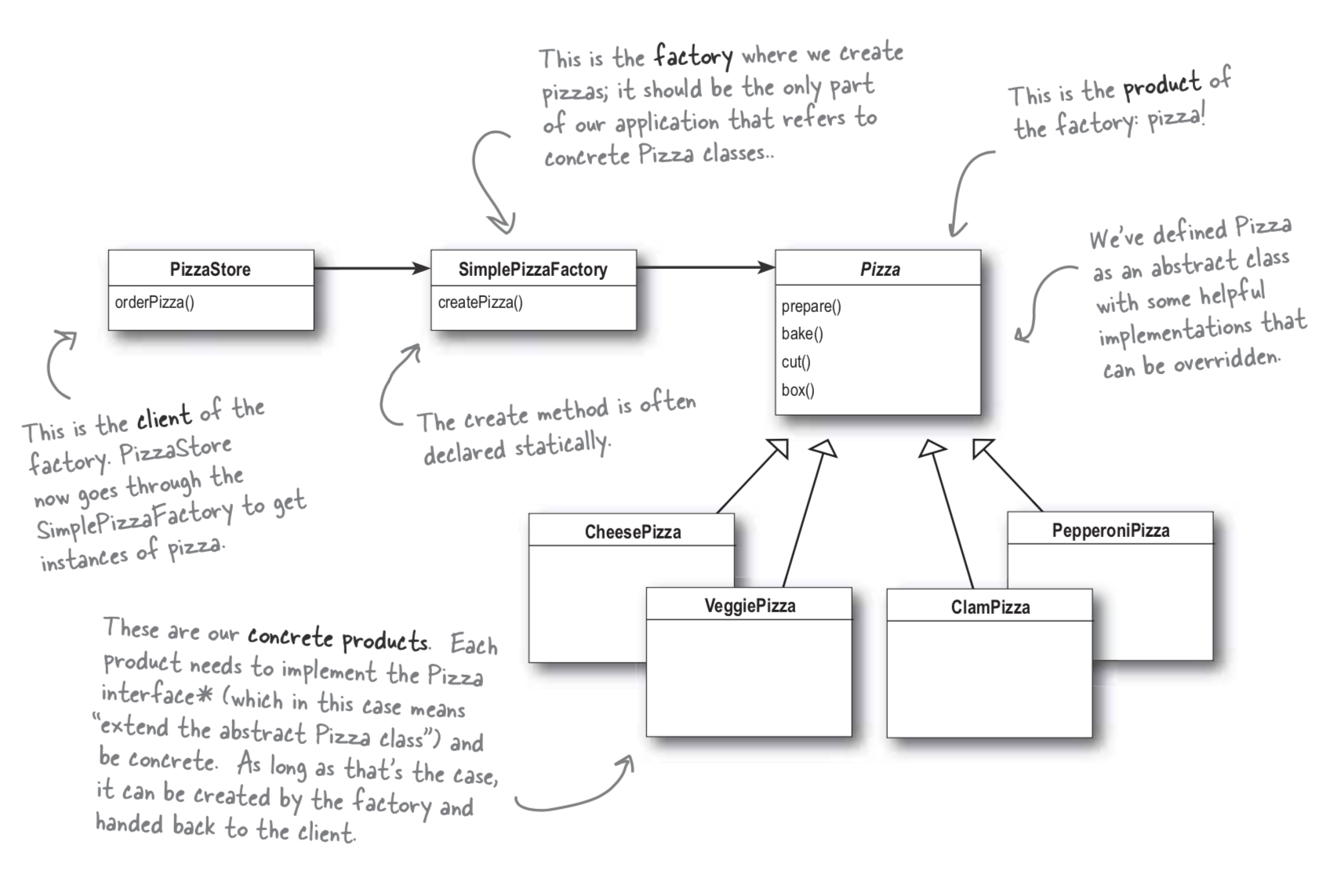
**The Factory Design: Simple Factory, Factory Method, Abstract Factory**

# **Simple Factory Pattern**

* Identify the part in our code that varies and separates them from what stays the Same
* Create new Factory Class – which will handle the details of object creation.

Class Diagram



Code in Action

|  |
| --- |
| **pizzastore.ts** |
| **class** PizzaStore {    factory: SimplePizzaFactory;    **constructor**(factory: SimplePizzaFactory) {  **this**.factory = factory;  }    **public** orderPizza (type: **string**) {  **let** pizza: Pizza;    // This is what varies. At the pizza selection changes over time, we need to modify this over and over again.  pizza = **this**.factory.createPizza(type); // replacing concrete instantiation of Pizza class using 'new' --> with a method of factory object - No more concrete instantiation here.    // This is what we expect to stay the same  pizza.prepare();  pizza.bake();  pizza.cut();  pizza.box();  **return** pizza;  } } |
| **simplePizzaFactory.ts** |
| **class** SimplePizzaFactory {    pizza: Pizza = **null**;    **public** createPizza(type: **string**): Pizza {  **if** (type === 'cheese') {  **this**.pizza = **new** CheesePizza()  } **else if** (type === 'pepperoni') {  **this**.pizza = **new** PepproniPizza();  } **else if** ( type === 'clam') {  **this**.pizza = **new** ClamPizza()  } **else if** ( type === 'veggie') {  **this**.pizza = **new** VeggiePizza();  }  **return this**.pizza;  } } |
| **Pizza.ts** |
| **abstract class** Pizza {  prepare(){}  bake(){}  cut(){}  box(){} }  **class** CheesePizza **extends** Pizza { }  **class** PepproniPizza **extends** Pizza { }  **class** ClamPizza **extends** Pizza { }  **class** VeggiePizza **extends** Pizza { } |

# **Factory Method Pattern\*\***

* When you are trying to instantiate, let’s encapsulate that instantiation, so we can make a uniform across all the place.

🡪 So, we can use this encapsulated class (factory), whenever we want to instantiate an object, and the factory is responsible for appropriate instantiation of the object

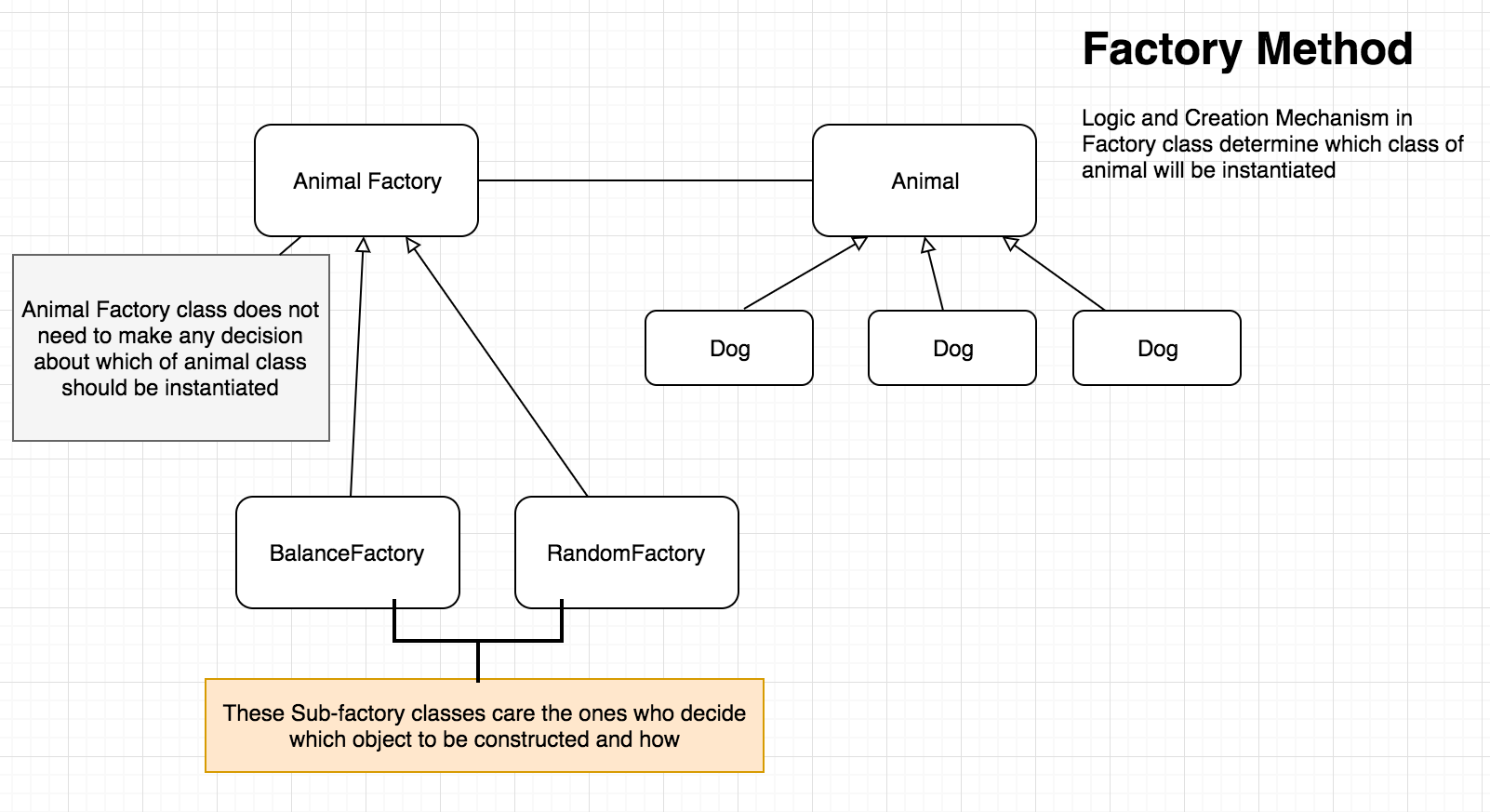
🡪 In other words, **factory is responsible for particular business logic and creation mechanism (there in logic encapsulated in the factory to determine what can be instantiated!)**

* Factory Method Pattern defines an interface (= common contract) for creating objects, but lets’ subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

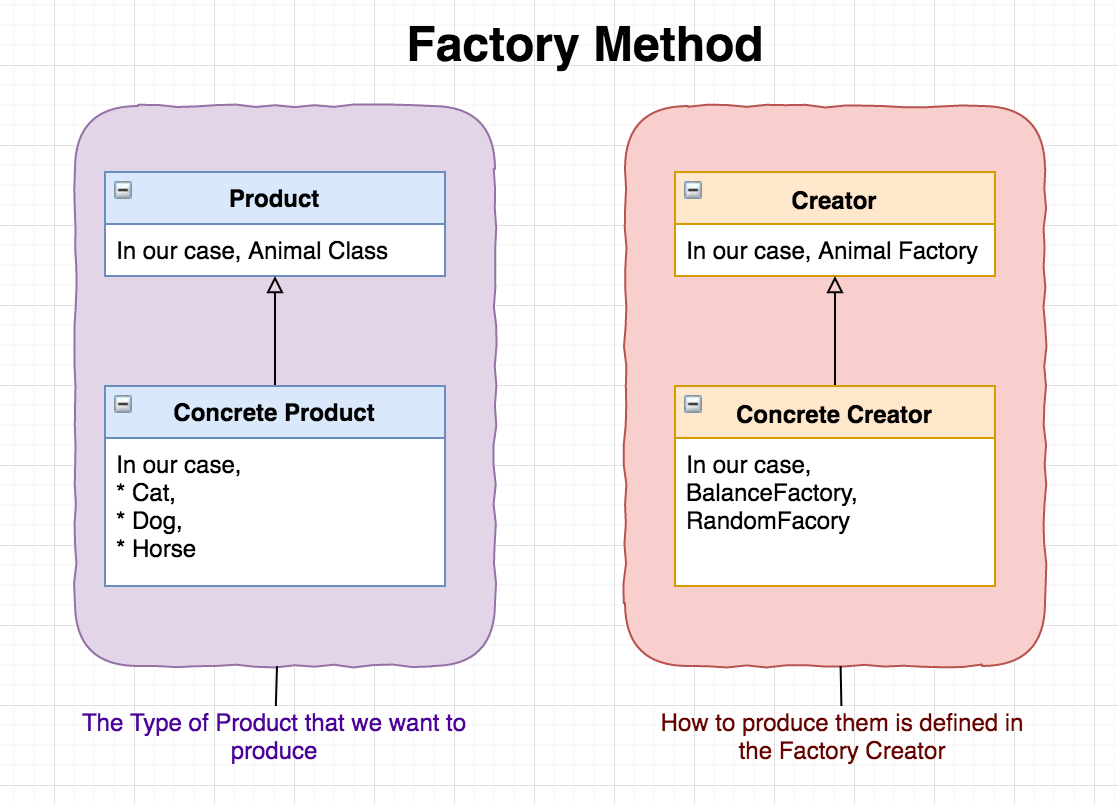
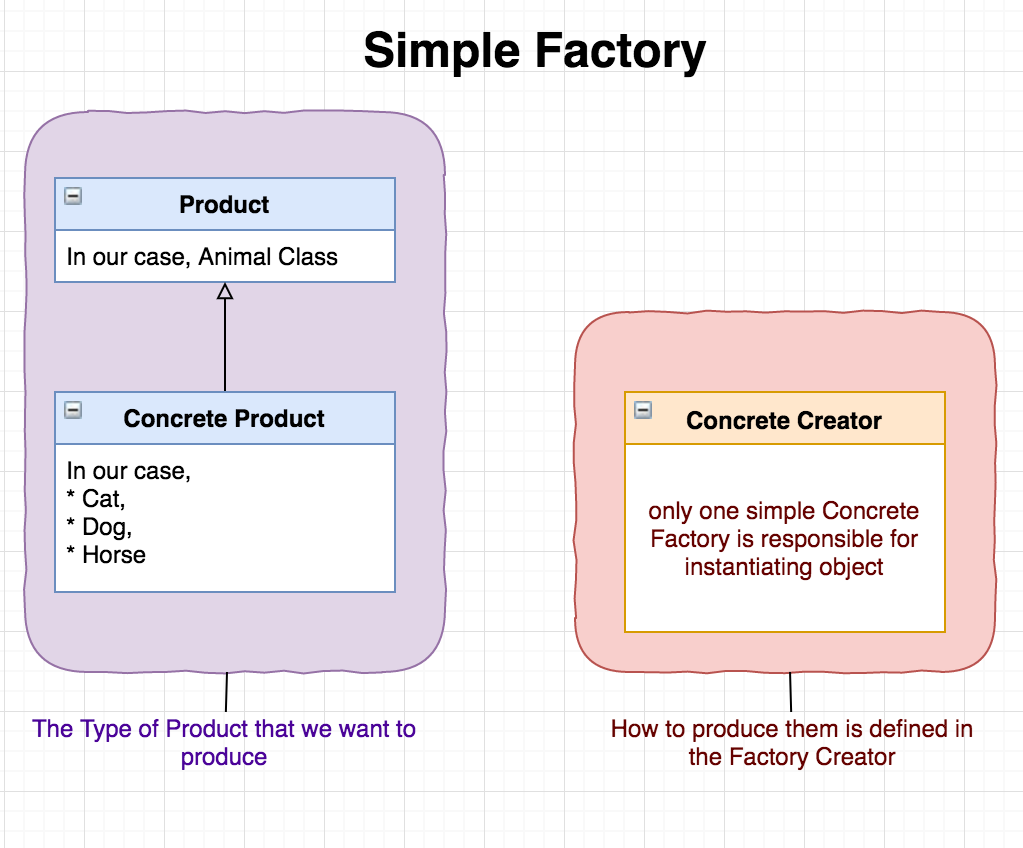
🡪 The key point of the Factory Method pattern is in the end, we want an object.

🡪 We don’t want to know how or why we want to construct that object, or what parameter that we want to pass when constructing that object. This part we want somebody else to take it (defer instantiation to subclasses)

## Simple Class Example of Factory Method



## Factory Method vs. Simple Factory

* Concrete Creator returns Product\*\*\*

# **Abstract Factory Pattern\*\***

* **The Abstract Factory pattern is a set of factory methods**

🡪 Which means that the abstract factory pattern makes use of multiple factory methods.

* The abstract Factory Pattern provides an interface for **creating families of related or dependent objects** **without specifying their concrete classes.**

🡪 A single difference between abstract factory pattern and factory method is

* **Abstract factory pattern** constructs **multiple objects** while
* **Factory method pattern** constructs **a single objet**.

