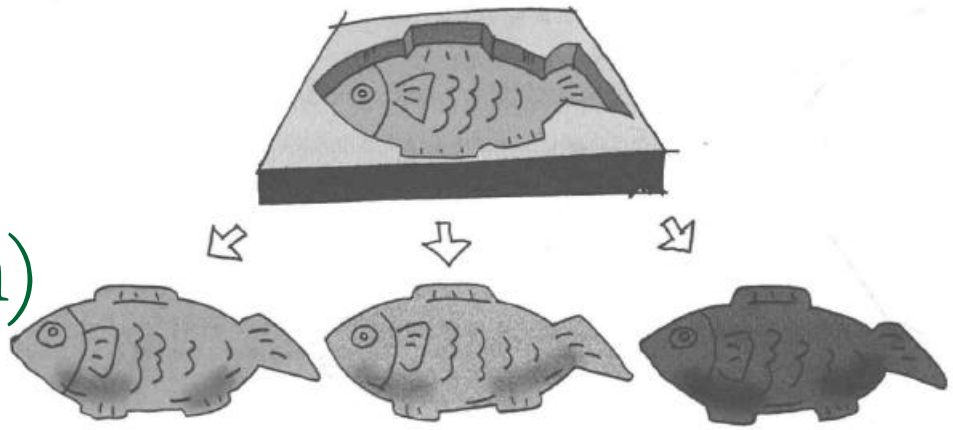
Factory Method (工厂方法, Creational Pattern)



Kai SHI

The Problem with "new" (1/2)

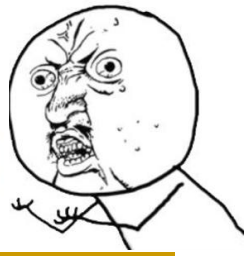
- When you see "new", think "concrete".
  - When you use **new**, you are certainly instantiating a concrete class, so that's definitely an implementation, not an interface.

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

We want to use interfaces to keep code flexible.

It should be an "interface", NOT a concrete class!

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



The Problem with "new" (2/2)

- When you have a whole set of related concrete classes, often you'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();
}
```

- When it comes time for changes or extensions, you'll have to reopen this code and examine what needs to be added (or deleted).
- Bad!!!

In Strategy lesson, we learned a design principle:

Separating what changes from what stays the same.
Take what varies and "encapsulate" it.

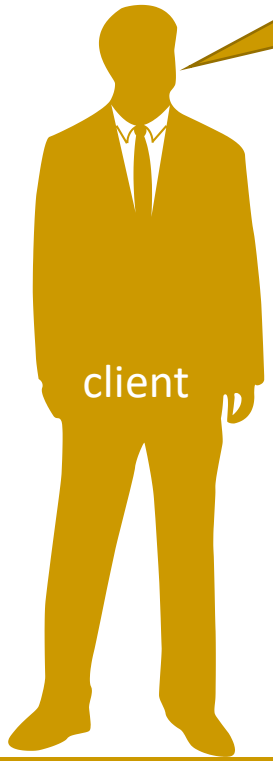
- A client come to "全聚德", want to eat a duck. He has to new a duck by himself. Are u kiding me?
- Client code:

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

Fact

I do not new ducks by myself.
I just have a cook interface (面向接口编程).
Then, a concrete cook will provide me concrete duck food.
我和鸭子解耦了。。

Encapsulate what changes (new MallardDuck()) in concrete cook.



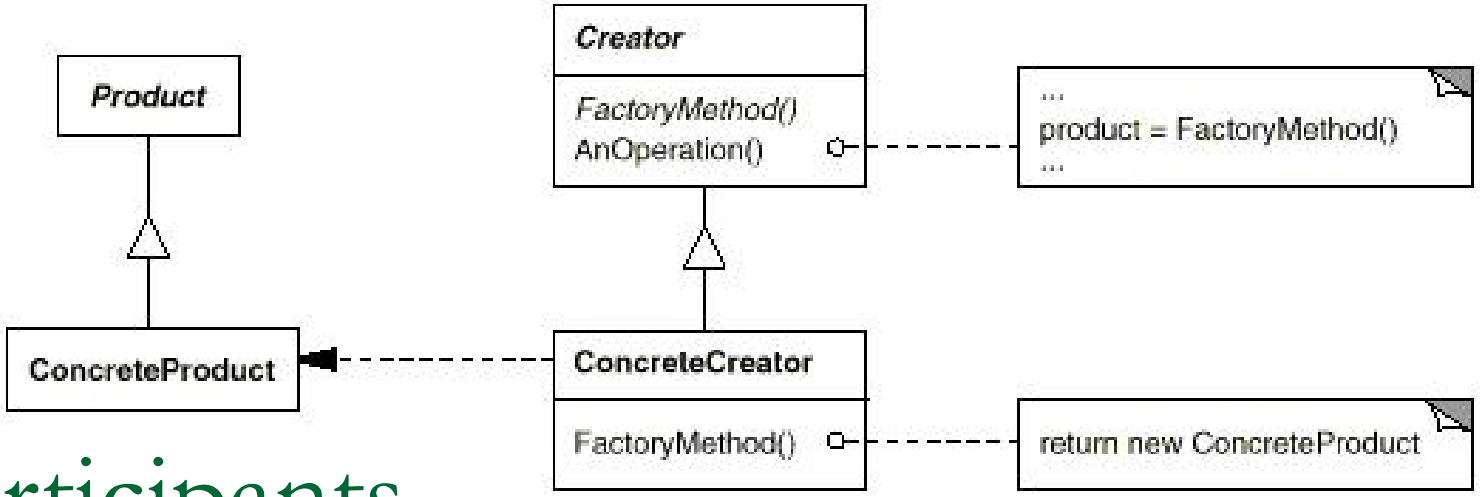
Factory Method

- Intent
  - Define an interface for creating an object, but let subclasses decide which class to instantiate.
- Also Known As
  - Virtual Constructor
- Motivation
  - 前面讲了

Applicability: Use the Factory Method pattern in any of the following situations:

- The concrete products are required not to exposed to clients;
- Creator can't decide the concrete product it must create;
- Creator wants its subclasses to specify the product it creates.
- Creator delegate the responsibility of creating instance to one of several subclasses.

Structure



Participants

- Product: defines the interface of objects the factory creates.
- ConcreteProduct: implements the Product interface.
- Creator: declares the **factory method**, which returns an object of type Product.
  - Creator may also define a default implementation of the factory method that returns a default ConcreteProduct object.
- ConcreteCreator: overrides the factory method to return an instance of a ConcreteProduct, referred by Product.

Collaborations

- Creator relies on its subclasses to implement the factory method so that it returns an instance of the appropriate ConcreteProduct.

Consequences: Advantages

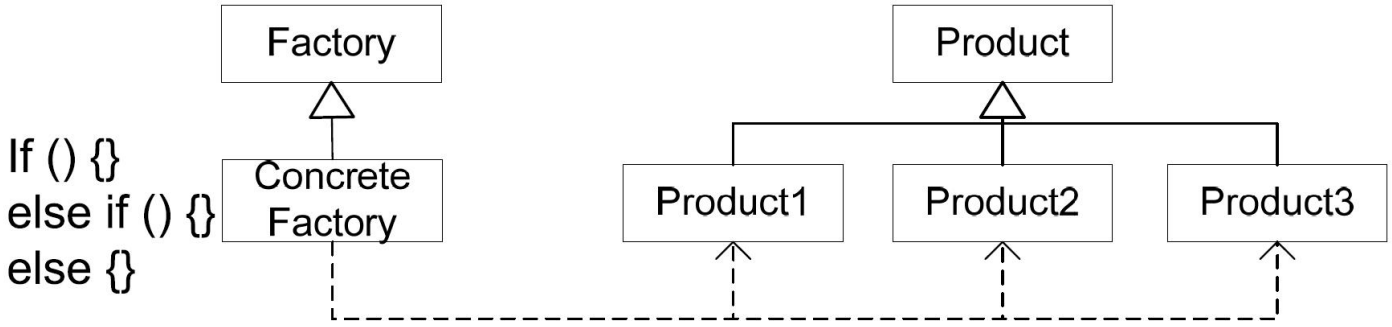
- By placing all creation code (new) in one object or method, avoid duplication in the code and provide one place to perform maintenance.
- Clients depend only upon interfaces rather than the concrete classes required to instantiate objects. -- To program to an interface, not an implementation.

Consequences: Disadvantages

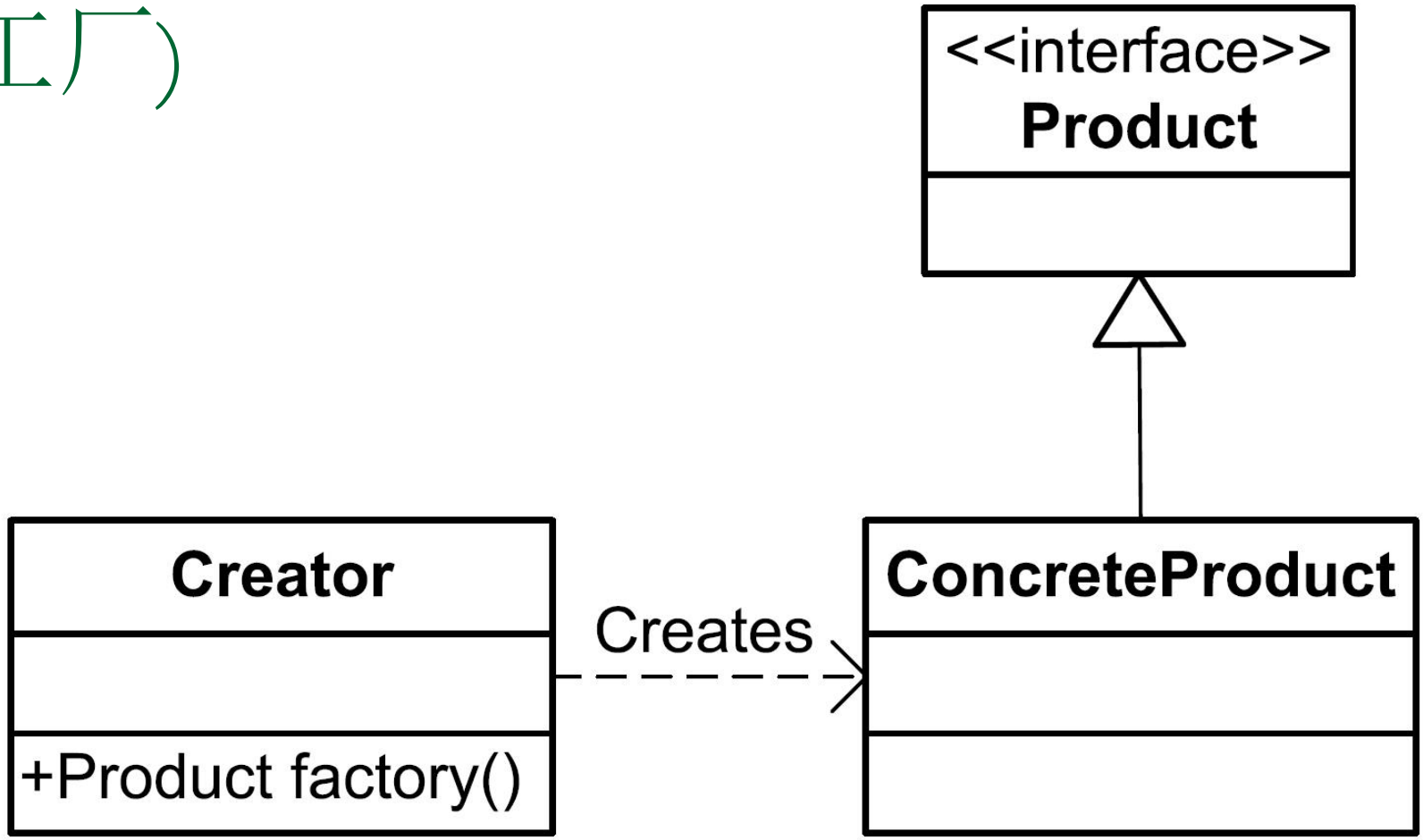
- A potential disadvantage of factory methods is that clients might have to subclass the Creator class just to create a particular ConcreteProduct object.

Implementation

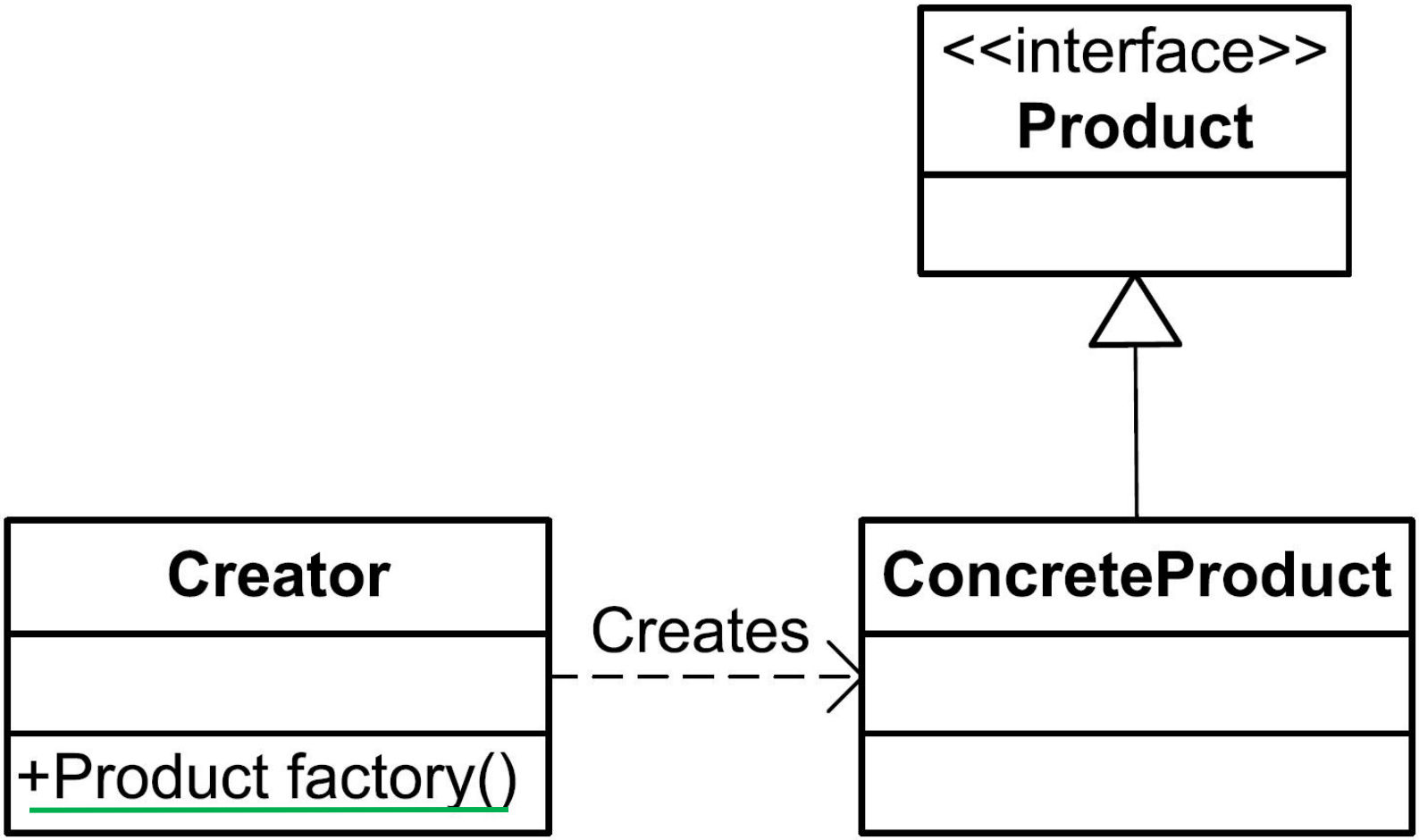
- Creator can be abstract or concrete
  - when the Creator is a concrete class and provides a default implementation for the factory method
- **Parameterized factory methods**
  - Factory method create multiple kinds of products by taking a parameter that identifies the kind of object to create.



Variation 1: Abstract Factory is omitted (**Simple Factory** Pattern, 简单工厂)



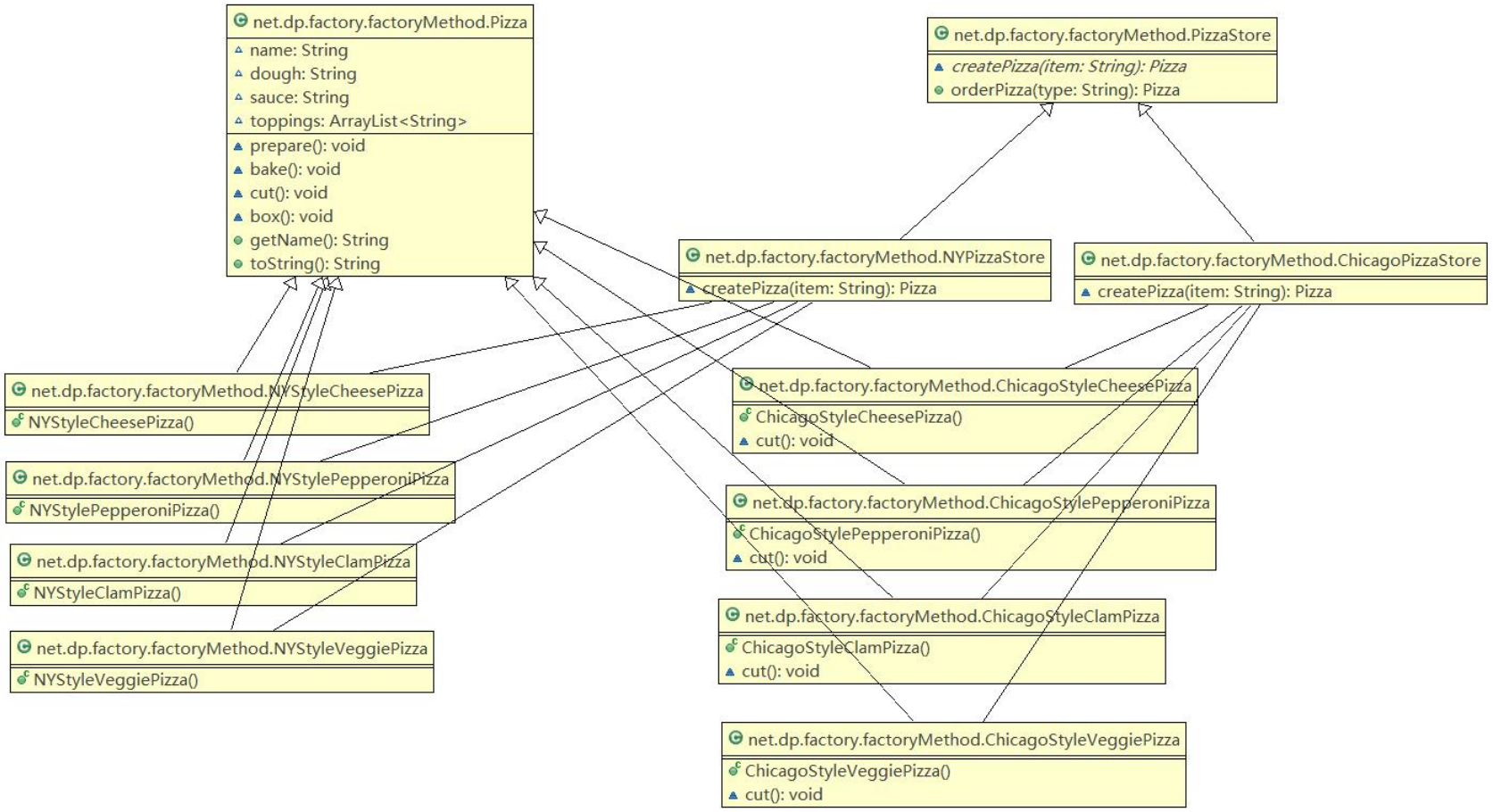
Variation 2: Static Factory Method (静态工厂)



Let's Try! Problem: A Pizza Chain

- Requirements
  - A pizza chain (连锁店) has two stores: New York store and Chicago store. Each store could order pizza and create pizza.
  - Each store sells four different type of pizza. Moreover, the taste is different in different area. e.g.,
    - ChicagoStyleCheesePizza
    - ChicagoStylePepperoniPizza
    - ChicagoStyleClamPizza
    - ChicagoStyleVeggiePizza (蔬菜)
- Use fatory method, draw class diagram NOW.

Class Diagram



Example: Spring's **BeanFactory**: Dependency Injection (依赖注入)

Without dependency injection

With dependency injection

Example: Spring's **BeanFactory**: Dependency Injection (依赖注入)

- Without dependency injection
  - the Client class contains a Service member variable that is initialized by the Client constructor. The client controls which implementation of service is used and controls its construction. In this situation, the client is said to have a hard-coded dependency on ServiceExample.

```
// An example without dependency injection
public class Client {
    // Internal reference to the service used by this client
    private ServiceExample service;

    // Constructor
    Client() {
        // Specify a specific implementation in the constructor instead of using dependency injection
        service = new ServiceExample();
    }

    // Method within this client that uses the services
    public String greet() {
        return "Hello " + service.getName();
    }
}
```

Three Types of Dependency Injection

- constructor injection: the dependencies are provided through a class constructor.
- setter injection: the client exposes a setter method that the injector uses to inject the dependency.
- interface injection: the dependency provides an injector method that will inject the dependency into any client passed to it. Clients must implement an interface that exposes a setter method that accepts the dependency.

Constructor injection

```
// Constructor
Client(Service service) {
    // Save the reference to the passed-in service inside this client
    this.service = service;
}
```

Setter injection

```java
// Setter method
public void setService(Service service) {
    // Save the reference to the passed-in service inside this client
    this.service = service;
}
```

Interface injection

```java
// Service setter interface.
public interface ServiceSetter {
    public void setService(Service service);
}

// Client class
public class Client implements ServiceSetter {
    // Internal reference to the service used by this client.
    private Service service;

    // Set the service that this client is to use.
    @Override
    public void setService(Service service) {
        this.service = service;
    }
}
```

Manually Assembling

- Manually assembling

```
public class Injector {
    public static void main(String[] args) {
        // Build the dependencies first
        Service service = new ServiceExample();

        // Inject the service, constructor style
        Client client = new Client(service);

        // Use the objects
        System.out.println(client.greet());
    }
}
```

BeanFactory

- Frameworks like Spring can construct these same objects and wire them together before returning a reference to client. Notice that all mention of ServiceExample has been removed from the code.

```
import org.springframework.beans.factory.BeanFactory;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Injector {
    public static void main(String[] args) {
        // -- Assembling objects -- //
        BeanFactory beanfactory = new ClassPathXmlApplicationContext("Beans.xml");
        Client client = (Client) beanfactory.getBean("client");

        // -- Using objects -- //
        System.out.println(client.greet());
    }
}
```

- This code (above) constructs objects and wires them together according to Beans.xml (next page).

Beans.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.springframework.org/schema/beans
 http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

    <bean id="service" class="ServiceExample">
    </bean>

    <bean id="client" class="Client">
        <constructor-arg value="service" />
    </bean>
</beans>
```

指定构造Client时，用service作为其构造函数参数