FUNDAMENTALS OF OBJECT ORIENTED PROGRAMMING

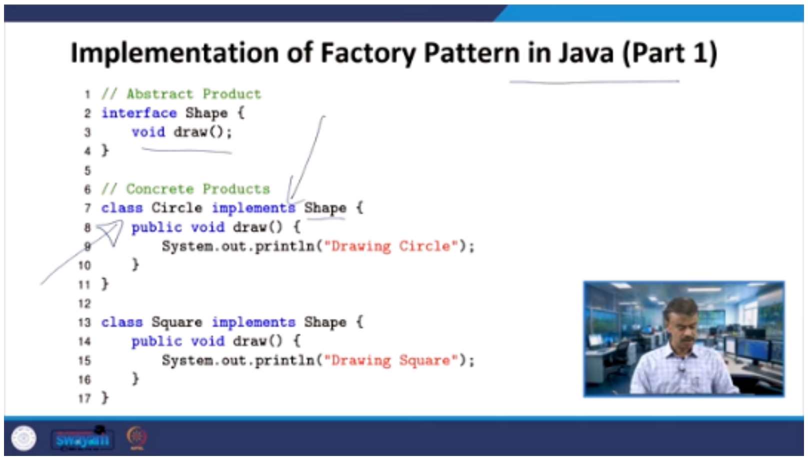
Lecture48

Lecture 48: Factory Pattern in Java

Thank you. So, welcome to Lecture 48: Design Patterns. So, in the last lecture, we talked about the factory pattern in C++, right? So, in today's lecture, what we can do. So, we will see the similar or same program in Java, right?

So, let us have an interface Shape, all right. So, I talked about interfaces when we are dealing with the object-oriented case. So, let us consider the interface Shape, all right. So, here you have void draw. It is the interface; it is abstract.

So, in fact, 100 percent abstraction we can do in an interface, all right. So, interface Shape void draw. So, now the class Circle, right? So, here you go. We have class Circle, which implements Shape, correct? Which is implementing Shape. So, I am defining void draw, that means it will override, all right? It will be overridden.



Implementation of Factory Pattern in Java (Part 1)

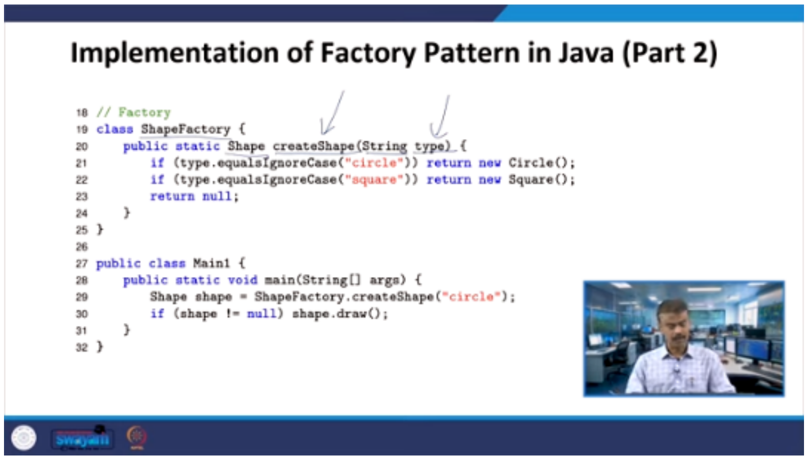
```java
// Abstract Product
interface Shape {
    void draw();
}

// Concrete Products
class Circle implements Shape {
    public void draw() {
        System.out.println("Drawing Circle");
    }
}

class Square implements Shape {
    public void draw() {
        System.out.println("Drawing Square");
    }
}
```

So, System.out.println drawing circle. Right, System.out.println drawing circle and you have class Square, right? So, the class Square implements Shape. Class Square implements Shape, and here I go: public void draw. Right. So, System.out.println drawing square, all right. So, one is drawing circle, another one is drawing square.

So, your class is Circle and class is Square, and here you go, right. So, here you go, your interface is over here, and your class Circle is over here, and here you have class Square in line number 17. Okay. So, now as usual, let us have a class ShapeFactory. Correct.

So, here you have public static Shape createShape object. All right. Under Shape. So, here we do not need to worry about the star and reference in Java. All right.

So, here you have the object type under string class. Right. Under string. Okay. Under string.



Implementation of Factory Pattern in Java (Part 2)

```java
// Factory
class ShapeFactory {
    public static Shape createShape(String type) {
        if (type.equalsIgnoreCase("circle")) return new Circle();
        if (type.equalsIgnoreCase("square")) return new Square();
        return null;
    }
}

public class Main1 {
    public static void main(String[] args) {
        Shape shape = ShapeFactory.createShape("circle");
        if (shape != null) shape.draw();
    }
}
```

So, now if the type right. So, which is invoking type is an object under string. So, if the type is invoking equals ignore case circle right. So, if it is equal to circle all right. So, this is what we had seen in the case of C++ it is equivalent to that.

Then return new circle that means it is calling the constructor circle. So, that means it will go to this particular class and suppose you are creating an object under let us say this circle right. Under this. So it will call. Right.

So, we are going to see. Let us see what we are passing. And under which. I mean, suppose I am creating an object shape. Right.

Under shape. So, then we are going to see. How exactly it is going to work. But suppose if it is square. So, the type when it is invoking equals ignore case, right?

When it is invoking equals ignore case, and then you are passing, assume that you are passing square when you are creating an object in the main, all right.

So, if this will return the pointer square, that means this will return the constructor square, right? So, square is over here, and suppose you are creating a shape object. Right, and then you are passing, let us say, square. So, then this particular function should be called, right? In fact, this will be overridden. So, let us go to the main program, right? Your shape factory class is over here, and let us go to the main program.

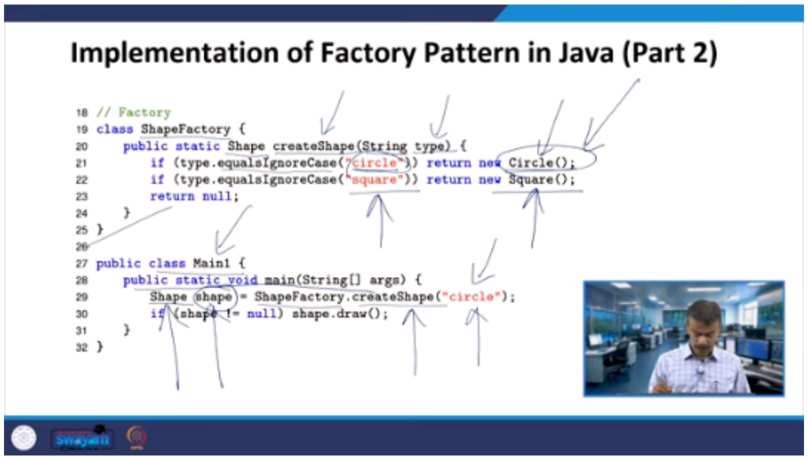
So, the main program, if you see right, the main class is correct. So, this is the main, all right. So, here you have class Shape. The class is over there. Yes.

And here you have ShapeFactory. Correct. So, here I have the main public class MainOne. So, the public class MainOne is a driver class. In fact, I am looking for the driver class.

So, MainOne is a driver class. So, under this, you have public static void main. I am creating an object Shape, small s under capital Shape. That means you have an interface Shape. Right.

And this shape factory is this particular class which is invoking create shape by passing circle right it is equivalent to the object that shape right small s which is invoking create shape that is equivalent code right so when it is invoking create shape and you are passing circle so type is circle now right so if a type is equal to this circle yes it is circle return new circle that means this particular constructor it is going right so when it is going to this particular constructor this function if suppose this object is invoking draw right. So, that will be overridden and since you have passed circle right.

So, when I am creating an object shape small s under capital shape the interface right. So, this particular function will be called the draw function will be called because the interface draw will be overridden and when it is printing system dot out dot printl and drawing circle right. So, the drawing circle correct. So, that will be printed the drawing circle will be printed right so in fact it is happening shape not equal to null yes it is not null because it is printing it is returning circle so in fact it will not go to line number 23 so this shape right now invoking draw function



Implementation of Factory Pattern in Java (Part 2)

```java
18 // Factory
19 class ShapeFactory {
20     public static Shape createShape(String type) {
21         if (type.equalsIgnoreCase("circle")) return new Circle();
22         if (type.equalsIgnoreCase("square")) return new Square();
23         return null;
24     }
25 }
26
27 public class Main1 {
28     public static void main(String[] args) {
29         Shape shape = ShapeFactory.createShape("circle");
30         if (shape != null) shape.draw();
31     }
32 }
```

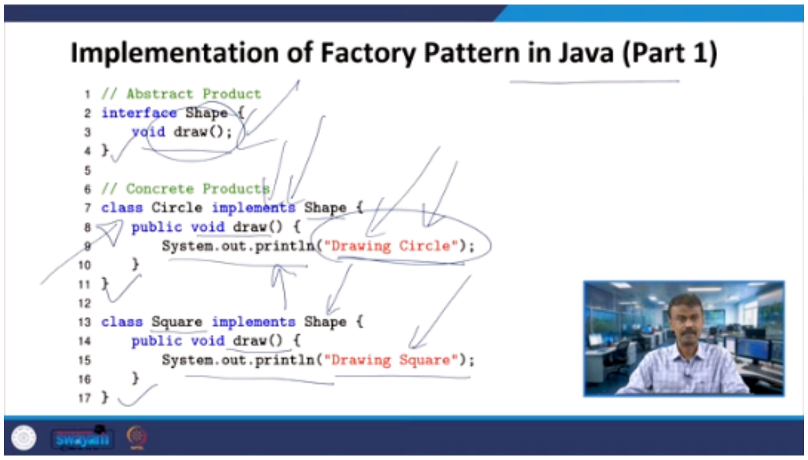
So, when it is invoking draw function since line number 29 you have passed circle right. So, line number 29 you have passed circle. So, the circle constructor. So, therefore, the function line number 8 right it is going to circle class and line number 8 you have y draw. So, this particular method will be called

And in the method, you have system.out.println drawing circle. So, therefore, you will get the output drawing circle. When you are running the code, you will get the output drawing circle, right. So, this is how the factory pattern in Java works. So, here you go line number 19.

So, you have shape factory, right. So, again if you see line number 29, I mean you are just ordering create shape. Right. By passing circle. So then the encapsulation you have line number 19.

So and then. Right. So since you have passed the circle your type is now circle. String type is circle. So and then since it is circle it is returning the circle constructor.

So, the circle constructor means it will go to the circle class. So, the circle class, suppose you have, let us say, shape. Right. Small 's'. Right. Or it is an object under shape.



Implementation of Factory Pattern in Java (Part 1)

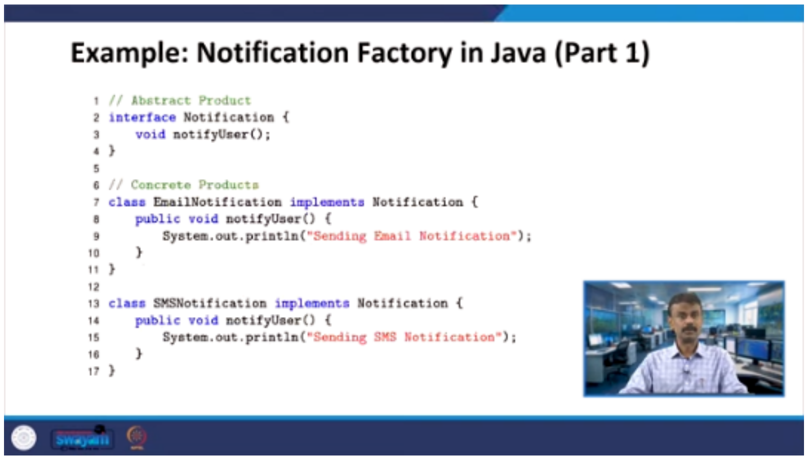
```java
1  // Abstract Product
2  interface Shape {
3      void draw();
4  }
5
6  // Concrete Products
7  class Circle implements Shape {
8      public void draw() {
9          System.out.println("Drawing Circle");
10     }
11 }
12
13 class Square implements Shape {
14     public void draw() {
15         System.out.println("Drawing Square");
16     }
17 }
```

The shape is the interface. So, the draw line number 3 will be overridden, and line number 9 will be called. Line number 8 will be called, and here you have System.out.println and drawing circle, right. So, suppose you assume that you are passing square, right. You are asking to create shape square, right. So, then the type is square, line number 22, right. It will be returned square, that means the constructor will be returned, and here you go, the square.

Correct. This particular function will be called, and the drawing square will be returned. Right. Will be printed. Right.

So this is how the implementation of factory pattern works in Java. Right. So when I run the code, since you are passing circle, drawing circle will be the output. Right. So this is all about your factory pattern.

So now I will talk about the notification factory in Java. Right. I still stick on Java. So notification factory in Java. So what do you mean by notification factor?



Example: Notification Factory in Java (Part 1)

```java
// Abstract Product
interface Notification {
    void notifyUser();
}

// Concrete Products
class EmailNotification implements Notification {
    public void notifyUser() {
        System.out.println("Sending Email Notification");
    }
}

class SMSNotification implements Notification {
    public void notifyUser() {
        System.out.println("Sending SMS Notification");
    }
}
```

Let us have the interface notification, right. So here you have one member function void notify user. It is abstract, right, 100% abstract. So that means I have to inherit this, right. I mean this function has to be inherited in the subclass.

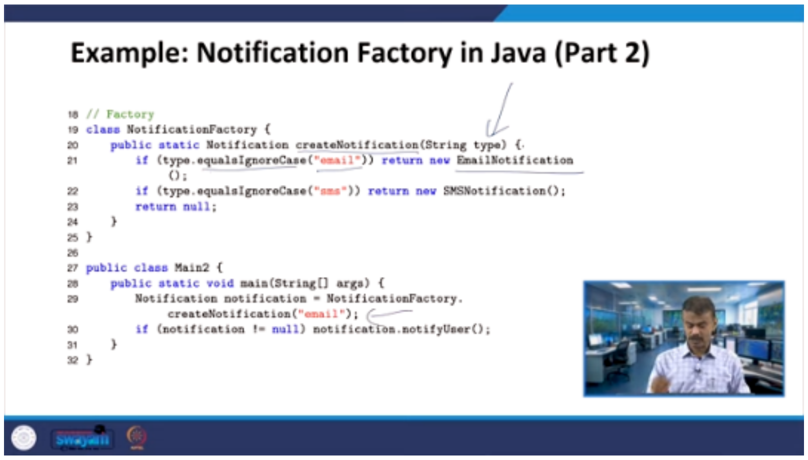
So, let us say class EmailNotification is implementing the interface Notification, EmailNotification. Class EmailNotification implements Notification, right? So, public void notifyUser—so that is what you have here in the abstract interface. In fact, that is not an abstract class; it is the interface—pure 100 percent abstraction, we know that, all right. So, line number 8, you have notifyUser, right? Which is the method, right? So, this has to be inherited, and the inherited class is EmailNotification, which is implementing Notification.

So, here we are defining All right. So, System.out.println and sending email notification. Simple. And here, one more class also implements Notification.

Right. So, Notification is an interface. That means SMSNotification class is also implementing Notification. And here, you have void notifyUser. Right.

Void notify user. So, printing sending SMS notification. So, notify user which is printing sending SMS notification. So, here you have the subclass SMS notification and one more subclass email notification and here you have interface notification right. So, now let us say you have notification factory right.

So, here you have notification factory and I have create notification object. right so that you are pausing the type right so that you are pausing the type if type dot equals ignore case that means type is invoking equals ignore case like exactly the previous program and type is email assume that type is which is nothing but the string and the string is email then it is returning by calling the constructor email notification so suppose i am pausing email right assume that i am pausing here email So, then your type is email. So, that means it is calling the constructor email notification. And since I have created an object notification under the interface notification.



Example: Notification Factory in Java (Part 2)

```java
// Factory
class NotificationFactory {
    public static Notification createNotification(String type) {
        if (type.equalsIgnoreCase("email")) return new EmailNotification
            ();
        if (type.equalsIgnoreCase("sms")) return new SMSNotification();
        return null;
    }
}

public class Main2 {
    public static void main(String[] args) {
        Notification notification = NotificationFactory.
            createNotification("email");
        if (notification != null) notification.notifyUser();
    }
}
```

So, this particular function will be called because this will be overridden. All right. So, this will be called. That means this notify user will be called. Sending email notification will be printed.

Right. So, this is exactly what is happening. So, suppose my text, I mean to say my string, is SMS, alright. So, if it is equal to SMS, then it will call the SMS notification constructor. So, that means this is going over here, and then 'sending SMS notification' will be printed, alright.

So, if you go to the main program, this is what your notification factory case is all about, alright. So, now suppose I have class Main2, which is a driver class, alright. So, here I have the main program. The main function, your notification is an object under the notification interface, right. So, here this notification is invoking create notification by passing email, right.

So, you are passing email. So, when I pass email, here you have written email if it is equal, right. The type is equal to this email, then it is invoking the email notification constructor. So, when it is invoking this, since I have created an object notification under capital notification, which is the interface, this will be overridden and this will be called. This notify user will be called.

So when it is calling this notify user, you can see system.out.println, it is printing, sending email notification, right? So that will be printed, correct? So if notification not equal to null, here it is invoking. So, here the way of invoking so that means the notification object which is invoking the method notify user. So, when it is invoking notify user what is happening?

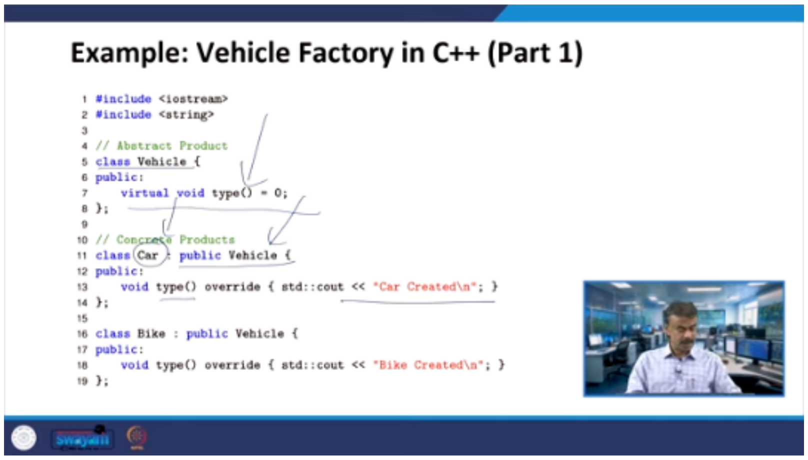
So, here you go sending email notification will be printed. So, when I run the code I have to get the output sending email notification. So, this is what exactly happening. So, this is a notification factory in Java and here you go this we call it as factory. So, similarly let us have one more example.

Now, I am going back to C++ right. So, we call it as vehicle factory in C++ right. So, here you have let us say abstract class vehicle and I have a pure virtual function called type right. So, type when I put equal to 0 it is a pure virtual function. And now I have car which is publicly inheriting the abstract class vehicle, right?

Which is publicly inheriting the abstract class vehicle. So, here when I put virtual void type is equal to 0, it is a abstract class. In fact, this is a pure virtual function. So, that means you have to inherit, right? It has to be defined in the subclass, right?

Or in the derived class. So, now I have a class car which is publicly inheriting vehicle. So, now here I have the type function. So, naturally it will overwrite. So, cout car created.

And I have another class called bike. I have another class called bike, which is publicly inheriting vehicle. And then I have a public member function, the same type function, which of course will overwrite this one. And then see out bike created. Alright.



Example: Vehicle Factory in C++ (Part 1)

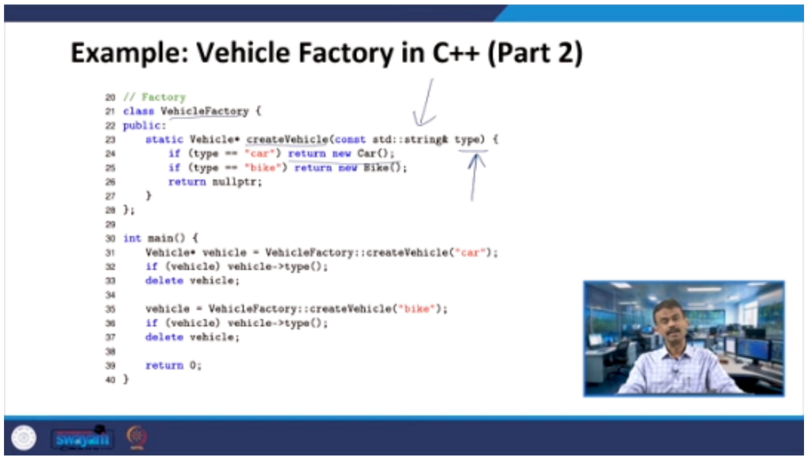
```cpp
#include <iostream>
#include <string>

// Abstract Product
class Vehicle {
public:
    virtual void type() = 0;
};

// Concrete Products
class Car : public Vehicle {
public:
    void type() override { std::cout << "Car Created\n"; }
};

class Bike : public Vehicle {
public:
    void type() override { std::cout << "Bike Created\n"; }
};
```

So your bike class is over here. And your car class is over here. And here your upside-class vehicle is over here. Alright. So now we go to the factory.

Vehicle factory. Alright. So I created an object called createVehicle. Correct. I have created an object called createVehicle.

And here you have the type. Alright. The reference type. Under string. If type is equal to car, return new car.

If type is equal to car, which is returning your constructor car. So, suppose I have created, let us say, pointer object under vehicle. Let us assume I am passing car. So, in that case, what will happen? Suppose my type is car.



Example: Vehicle Factory in C++ (Part 2)

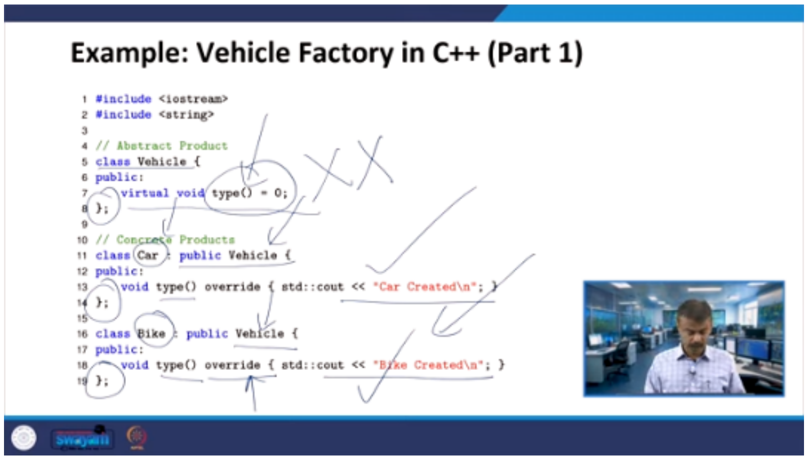
```cpp
// Factory
class VehicleFactory {
public:
    static Vehicle* createVehicle(const std::string& type) {
        if (type == "car") return new Car();
        if (type == "bike") return new Bike();
        return nullptr;
    }
};

int main() {
    Vehicle* vehicle = VehicleFactory::createVehicle("car");
    if (vehicle) vehicle->type();
    delete vehicle;

    vehicle = VehicleFactory::createVehicle("bike");
    if (vehicle) vehicle->type();
    delete vehicle;

    return 0;
}
```

So, your car constructor will be called. Car constructor is under the class car. So that means suppose you are invoking type the object is invoking type right. So in that case this will be overridden and this will be printed right. So whereas suppose you are creating an object right under vehicle pointer object under vehicle but you are passing let us say bike right.

So in that case It is calling the constructor bike. All right. It is calling the constructor bike. Correct.

So that means it will go to this particular class by overriding this. And this will be called over. I mean, the type will be called. Right. Bike created will be printed.

Right. So this is what exactly happening. Bike created will be printed. All right. So let us see this.



Example: Vehicle Factory in C++ (Part 1)

```cpp
#include <iostream>
#include <string>

// Abstract Product
class Vehicle {
public:
    virtual void type() = 0;
};

// Concrete Products
class Car : public Vehicle {
public:
    void type() override { std::cout << "Car Created\n"; }
};

class Bike : public Vehicle {
public:
    void type() override { std::cout << "Bike Created\n"; }
};
```

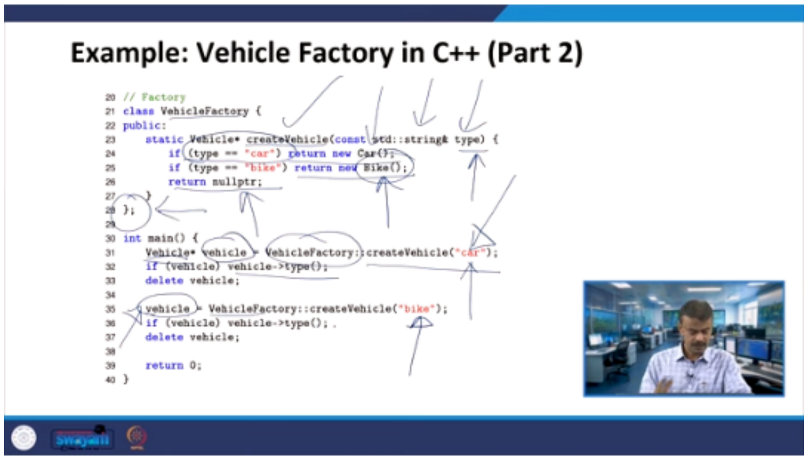
So now if there is no type. Right. If it is other than car and bike. return null pointer right so let us say you put bus assume that it is some other text so some other string so it will go to return here because if type is equal to car will be failed if type double equal to bike will be failed so return null pointer if it is any other string right so your factory class is over here and let us go to the main program main program i have created a pointer object right i have created a pointer object under vehicle right and this pointer object is invoking let us say create vehicle by pausing car right so you are invoking create vehicle right so the create vehicle which is available in the vehicle factory class right so that is being invoked by the pointer object vehicle

right so in that case since you are passing car right if type is equal to car true right so when it is true return the new car that means returning the constructor car so since i have created an object vehicle under vehicle pointer object vehicle under vehicle correct so since it is returning car in line number 24 car constructor so this particular function will be called by overriding this type right so it is what exactly happening right so this if vehicle it is not a null pointer if vehicle not a null pointer so vehicle is invoking type so when it is invoking type so here you go right so this will be overridden again I am putting the third cross it will be overridden and then here you see over right correct so car created will be printed first the first output over here is car created so now i am deallocating the memory for the pointer object vehicle all right line number 33 it is already printed what is the first print out car created is the first print out right so the first print right so now line number 33 i am deallocating a memory for vehicle right so now again right anyway line number 31 will work right so you have vehicle star vehicle that is the object under vehicle so now you are pointing your memory So, vehicle is in the left side and now here you have that create vehicle will be called by pausing bike correct. So, you are allocating a memory for vehicle by pausing right it is invoking in fact create vehicle by pausing bike right.

So, that means your type is bike. So, if the type is bike it is returning new bike right which is a constructor. So, that means vehicle star vehicle right. So, here you have vehicle. So this object when you are passing by correct.

So the meaning is this will call this particular constructor. So this constructor is available over here right. So this constructor which is available over here correct. So that means this will be overridden. Fourth time I am putting cross.

This will be overridden and this will be called. If at all, it is invoking. Yes, it is invoking here. If vehicle, yes, it is not a null pointer. Again, I am telling it is not a null pointer.



Example: Vehicle Factory in C++ (Part 2)

```cpp
// Factory
class VehicleFactory {
public:
    static Vehicle* createVehicle(const std::string& type) {
        if (type == "car") return new Car();
        if (type == "bike") return new Bike();
        return nullptr;
    }
};

int main() {
    Vehicle* vehicle = VehicleFactory::createVehicle("car");
    if (vehicle) vehicle->type();
    delete vehicle;

    vehicle = VehicleFactory::createVehicle("bike");
    if (vehicle) vehicle->type();
    delete vehicle;

    return 0;
}
```

So, vehicle is invoking type. So, that means this will be called and the second output will be byte created, will be printed, right. Then again, what you are doing? You are deallocating the memory, right. You are deallocating the memory for the object, return 0.

So, first output you have got car created and second output you got bike created. So, when I run the code I have to get this output right. So, what is the output car created and bike created right. So, this is what we are expecting I put 1 and 2. So, they are getting printed and you can see how the vehicle factory in C++ is working.

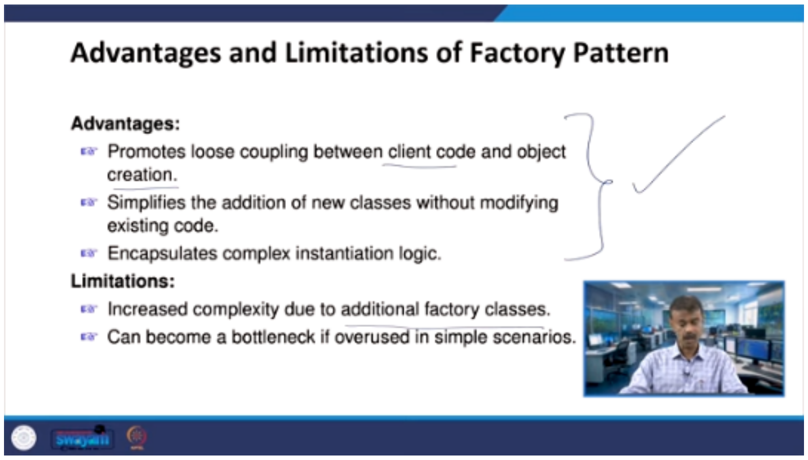
So, we have taken the vehicle example. So, you have seen this is called the factory in C++. So, what are all the advantages and what are all the limitations of factory pattern? So, the main advantage. So, it promotes the loose coupling between client code and object creation.

So, this is the meaning of why I talked about the loose coupling even previously. So, the client code that means the main function and the object creation. So, it is promoting the loose coupling between the client code and object creation and it is also simplifying the addition of new classes without modifying existing code. So, this is the second advantage. and the encapsulation right.

So, we had the instantiation logic the complex instantiation logic. So, here we have right. So, when you look at the vehicle factory which is the encapsulation line number 20 to 28 right. So, this factory pattern which is encapsulates complex instantiation logic right. So, what are all the limitations?

So, these are all the advantages. So, these are all the good, I mean, advantages. What are all the limitations? So, the factory pattern increases complexity due to the additional factory classes, all right. So, the complexity will be more because of the additional factory classes, and the factory pattern can become a bottleneck if overused in simple scenarios, right.

So, these are all the limitations, and these three are all advantages, right. So, these are all the advantages and limitations of the factory pattern. So, what is the observer pattern, right? So, next, we will talk about the observer pattern; the factory pattern we have seen.



Advantages and Limitations of Factory Pattern

Advantages:
- Promotes loose coupling between client code and object creation.
- Simplifies the addition of new classes without modifying existing code.
- Encapsulates complex instantiation logic.

Limitations:
- Increased complexity due to additional factory classes.
- Can become a bottleneck if overused in simple scenarios.

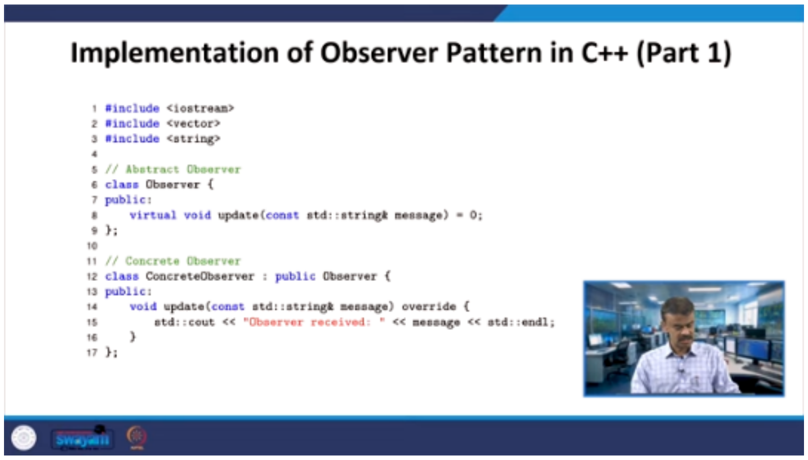
So, now we will talk about the observer pattern. So, the observer pattern establishes a one-to-many relationship between objects. Right, a one-to-many relationship between the objects. So, when the subject changes, all observers—that means the many observers—are notified automatically. So, this we will see with an example, all right.

So, the main features are decouples subject and observers and it is providing a publish subscribe mechanisms all right. So, the commonly used for event driven programming and reactive systems all right. So, these observer pattern They are commonly used for even driven programming and reactive systems, right. So, these are all the use cases.

I talked about the even handling, right. So, even handling. So, these are all the use cases of observer pattern. So, even handling is nothing but notifying user interface components when data changes, correct. and the use case in the real-time systems so for example suppose you need the broadcasting updates in the problems like a financial or a stock trading applications so this observer pattern are very much useful that means the real-time system these observer patterns are having very good applications next one is logging and monitoring all right when you alter the multiple loggers

of a specified event right the observer pattern come into the picture right so this is called as a logging and monitoring reactive programming right so when using the observer pattern when you want to handle streams of data with dynamic updates right so when you have a streams of data with the dynamic updates all right we call it as a reactive programming observer patterns are very much useful and the game development right, synchronizing game states between the objects, observer pattern comes into play, right. So, that means even handling real-time systems, logging and monitoring, reactive programming and game development, your observer pattern plays a vital role. So, let us consider the program, right. So, with the help of a program, we will see, right, how this observer pattern is working, right, so in C++.

So, let us have vector and string you have class observer. So, which is abstract class here you have a virtual function update right the string reference you have message which is equal to 0 ok. So, you have observer abstract class and here you have class concrete observer which is overriding right in fact this is inheriting observer which is the abstract class. And again you have the update function, update member function overridden, right. In fact, we have to define, right.



Implementation of Observer Pattern in C++ (Part 1)

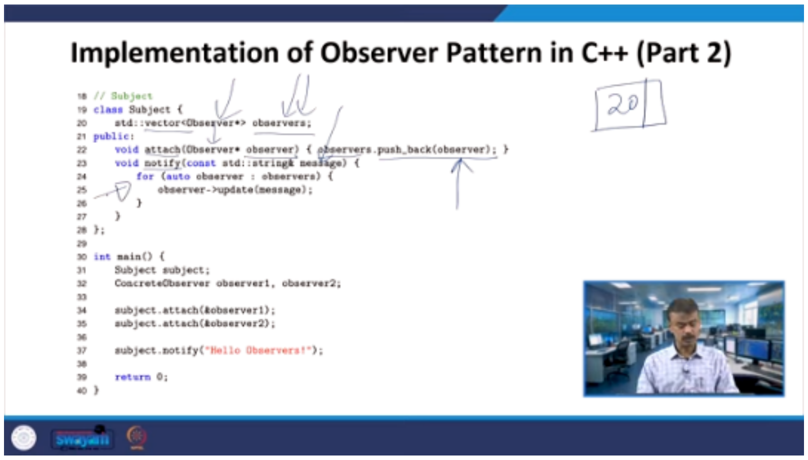
```cpp
#include <iostream>
#include <vector>
#include <string>

// Abstract Observer
class Observer {
public:
    virtual void update(const std::string& message) = 0;
};

// Concrete Observer
class ConcreteObserver : public Observer {
public:
    void update(const std::string& message) override {
        std::cout << "Observer received: " << message << std::endl;
    }
};
```

So, it is equal to 0 when I put it is a pure virtual function, right. So, now here you have message, the reference message under string overwrite, right. So, see out statement you have observer received, right. So, whatever message that we are going to pass that will be printed, right. So, this class is concrete observer which is inheriting observer.

And here you have class subject. right this is not inheriting anything class subject. So, under class subject I have vector pointer observer right. So, here you have an object observers pointer object observers. So, now I have a member function attached observer is the pointer object under the class observer right.

Here you have observers right I mean you have already defined line number 20 the private member data. or private object right the pointer object observers so this observers is invoking pushback observer so that means this is a inbuilt function right the inbuilt member function which is available in vector all right so this is a inbuilt member function which is available in vector so that means so when i am passing something so that will be pushed back right it is like we have seen the linked list right so it will be pushed back for example i have some data that will be putting over here all right so that is what exactly it is doing push back push back is the inbuilt function in vector So it is putting in the stack, putting in the stack or the list, right? So now next one is notify member function, right?

So I have notify member function. So message is the string, right? Message is the string. In fact, it is a reference string and I have range for loop, right? So observer.



Implementation of Observer Pattern in C++ (Part 2)

```
18 // Subject
19 class Subject {
20     std::vector<Observer*> observers;
21 public:
22     void attach(Observer* observer) { observers.push_back(observer); }
23     void notify(const std::string& message) {
24         for (auto observer : observers) {
25             observer->update(message);
26         }
27     }
28 };
29
30 int main() {
31     Subject subject;
32     ConcreteObserver observer1, observer2;
33
34     subject.attach(&observer1);
35     subject.attach(&observer2);
36
37     subject.notify("Hello Observers!");
38
39     return 0;
40 }
```

$\boxed{20}$

So the observers are copied into observer. Both are pointing to the same location and observer is invoking update message, right? So this is what exactly happening in class subject, right? The class subject is over here. Now in the main program, I have an object subject under class subject, fine?

So I have two objects, observer 1 and observer 2 under concrete observer. Where is concrete observer? Over here. And your pure virtual function is over here. Right.

Virtual class or abstract class. Correct. So now I have created two objects under here. This particular class line number 12. Okay.

The name of the class is observer 1 and observer 2. So now subject under class subject. which is invoking attach, right, which is invoking attach and you are pausing observer 1, the reference object observer 1, right. So, you are pausing observer 1. So, what is happening?

Attach, subject is invoking attach. Yes, it will be called. So, observer is nothing but now observer 1, right. Just consider this particular line. So, observer 1.

So, both are pointing to the same location, observer and observer 1 right now. Anyway, when you are creating an object, instantiating an object, so the observers can push back observer. So, that means observer 1 is in the list.

Observer 1 is in the list, push back, all right. So, it is pushing observer 1, let us say in the list, right.

Next, again subject which is invoking attach by pausing the reference of observer 2. So, again now observer and observer 2 will point to the same location correct and observers is pushing observer 2 right. So, the next one is let us say observer 2. So, in the list you have observer 1 and observer 2 alright. So, without loss of generality let us consider the stack observer 1 in the bottom observer 2 is like this or if you consider link list push back.

I am putting in the back it is understood right. So, maybe observer 1 and push back you have observer 2 if you consider like this fine either you consider this or this. So, it is like this now. The object subject which is invoking notify. Alright.

Which is invoking notify. Here you go. Alright. You are passing the message. The message is nothing but allow observers.

So now your observers. So which are nothing but observer 1 and observer 2. Alright. This is what the range for loop. So observer will be observer 1 and observer 2.

Correct. So now the observer updates the message. Invoking the update message. So observer 1 will invoke the update message. That is the meaning.

So when it is invoking the update message. Observer received will be printed with the message. What is the message? You passed all observers. So the first time will be printed.

Right. Again, the pointer now has observer 2. Because it is a for loop. Range for loop. So observer 2 will invoke update by passing the message.

What you are passing? Allow observers, right? So, again observer received. The message is allow observers will be printed, right? So, when I run the code, 2 times it has to be printed, right?

So, you make sure you draw the diagram. So, either you take it as a list, right? Or the link list like this. So, now when I run the code, I have to get the output like this, right? This is what we are expecting, 2 times I said.

So, 2 times it will be printed. So, this is what exactly happened, right. So, this is called as the observer pattern in C++, right. So, I hope you understand how the observer pattern is working in C++. So, what we can do?

The similar or same program that we can do, we will see with the help of Java in the next class. Thank you very much.