FUNDAMENTALS OF OBJECT ORIENTED PROGRAMMING

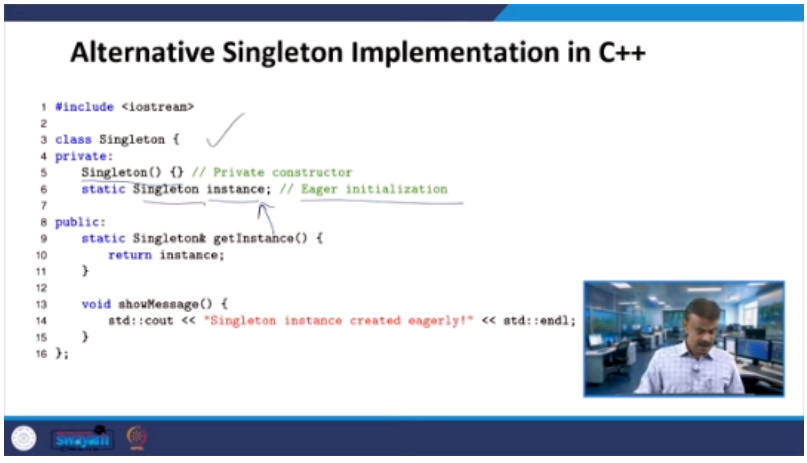
Lecture47

Lecture 47: Singleton and Factory Pattern

Thank you. Welcome to Lecture 47: Design Patterns. So, in the last lecture, I talked about Singleton, right? The Singleton implementation we had seen in C++ and Java, right. So, in this lecture, I will start with an alternative Singleton implementation in C++, right. So, let us consider the class Singleton, right? As usual, like in previous programs.

So, class Singleton. So, here we have the private constructor Singleton, right? And then you have another private instance, right. So, you call this eager initialization, right. So, this is called eager initialization. So, that means the instance will definitely occur, right.

So, When I am creating an instance under Singleton, all right, so this is called eager initialization. And then, I mean, our usual getInstance, right, which is the reference function under the Singleton class, and it will return the instance, all right. And then you have one member function called showMessage. The showMessage, here you have, see, 'Singleton instance created eagerly.'



right, the class is over here. So, now we are initializing the static member, all right. So, instance is a static member, all right, under the class singleton, right. So, now we will go to the main program. So, the main program, so we have the reference singleton, right, because when we are calling this function get instance, right, when we are calling this member function, right hand side if you look, we are calling the member function,

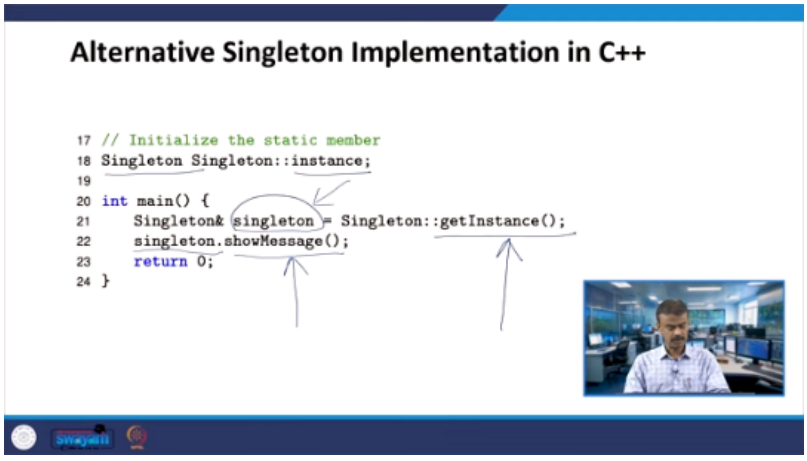
all right, which is gets instance. So, get instance if you see, so here you go, so this will return instance, right. So, like previously what we do, it is pointing to the null pointer, all right. So, we are creating instance. So, here since we have the eager initialization that is called the alternative singleton, all right.

So, the instance will definitely be there, correct. So, here you have the instance from line number 9 and 10. So, that will be returned, Correct. So, and here you have the object, the reference object singleton, you have the instance.

So, now this singleton, the object singleton, which is invoking show message. All right. So, which is invoking show message. So, when it is invoking show message, the singleton instance created eagerly will be printed. All right.

So, the main idea of this program is the eager initialization. The instance will occur. right. So, therefore, we are writing if you look at line number 9 and 10, I mean when I have this get instance member function, right. In fact, it is a reference member function with the class singleton.

So, which is returning instance, right. So, when this singleton invokes show message, correct. So, we are getting the output singleton instance created eagerly, right. So, will be the output, right. So, will be printed.

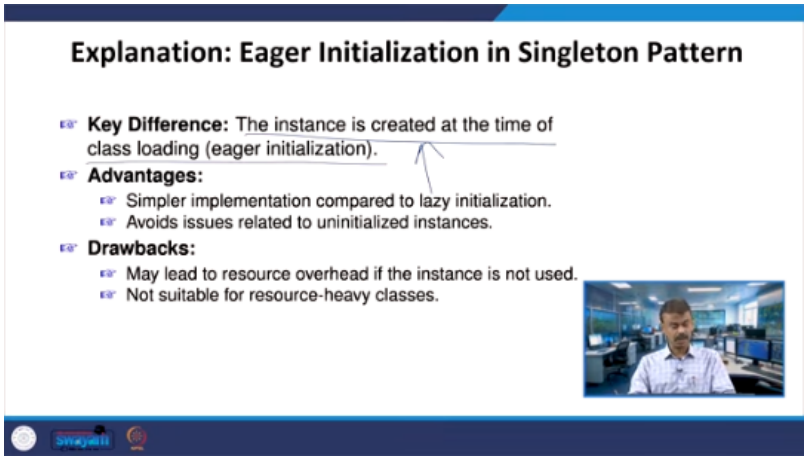


So, When I run the code, so I will get the output, singleton instance created eagerly will be printed. So, this is the alternative singleton implementation. In fact, so when we look at line number 6, right, so when you are creating the instance under singleton.

So, automatically the instance will be created and we no need to write the null pointer statement that we had written in the previous singleton implementation. So, here automatically instance will be created therefore, it will be written. So, the explanation point of view right this is for your reference. So, here the instance is created at the time of class loading. So, that is what the eager initialization the line number 6.

right the eager initialization right. So, when you are creating a instance over here. So, during this time right when you are creating at the time of class loading all right. So, the eager initial. So, instance will be created right the instance will be created at the time of class loading.

So, you call it as eager initialization that is line number 6. So, what are all the advantages of eager initialization in singleton pattern? I mean it is a simple implementation compared to The previous one we call it as a lazy initialization. So, whatever we had seen in the last lecture you call it as a lazy initialization because whenever you are creating an instance.

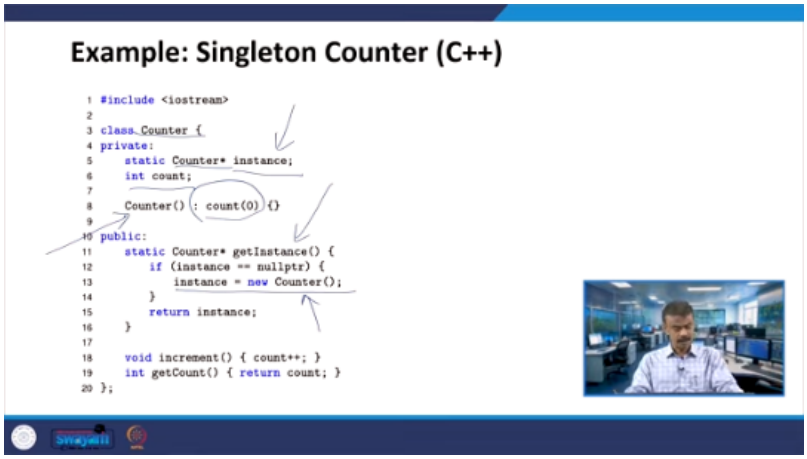


So, there is a possibility that it points to the null pointer right. So, that is what we are modifying the code. So, you call that as a lazy initialization and this one the eager initialization avoids issues related to uninitialized instances right. So, the

main drawbacks are So this may lead to resource overhead if the instance is not used alright.

So if you are not using instance then there will be a problem that means it may lead to resource overhead and another important drawback is it is not suitable for resource heavy classes. So this is all about your eager initialization in singleton pattern. So now if you look at the next program we call it as a singleton counter in C++ alright. So let us have a class counter. So the class counter as a private instance it is a pointer object under counter and you have one variable in count alright and you have a constructor counter right where count equal to 0 right it is a default constructor where count is equal to 0 and in your public member function it is a pointer member function.

get instance all right. So, again we are going for the lazy one if instance is equal to null pointer all right if instance is equal to null pointer you are creating an instance the creating a new instance correct where the constructor is counter. So, when it is calling the constructor means it is initializing to 0 all right. So, then Whatever be the case, whether the instance already exists or you are creating based on this line number 12 statement, that means if instance is equal to null pointer, you are creating an instance, it will return instance.



So, this we had seen several times, right? So, now line number 18 void increment, right? One of the member functions, count plus plus and int get count, right? This is another member function which is returning count, right? So, your counter class is over here.

So, now we are creating a object instance right pointer right pointer object initially pointing to the null pointer under counter ok. So, now go to the main program. So, you have small c counter is a object the pointer object right under counter on the right hand side it is invoking get instance. So, when it is invoking get instance the instance will be created right. So, this we had seen several times.

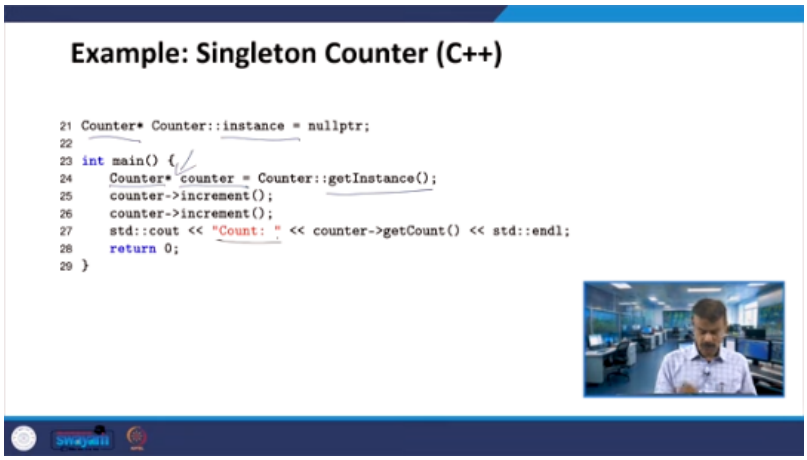
So, here you have the instance will be created and it will be returning instance right. So, now this counter is invoking increment right. The counter is invoking the increment member function. So, when it is invoking increment member function. So, here you go right count plus plus right.

So, when you are creating an object right under counter. Right. In fact, the pointer object. So it is initialized to 0 because of the line number 8. right?

It is initialized to 0. Now, the counter is invoking increment in line number 25. So, in that case, count will be incremented by 1. So, now, it is 1, count is 1, right? So, count is 1 and it is right over here, correct?

In line number 8, all right? And similarly, counter is again calling, again invoking increment, right? So, when it is again invoking the increment, line number 18, again count plus plus, all right? So, count plus plus, it is the Private member data of the class counter, right.

So, 1 plus 1, now it will be 2, right. 1 plus 1, it will be 2, right. So, now see out count. So, this will be printed. So, counter is invoking get count, right.

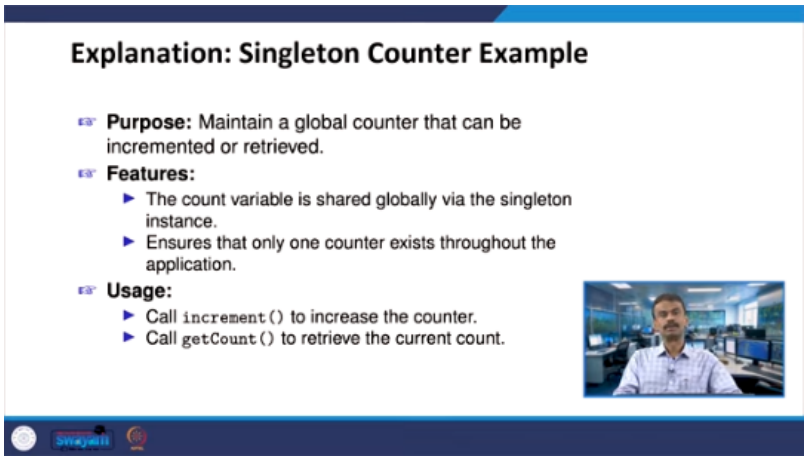


When the counter is invoking get count, what will you get? So, it is returning count. What is count? Count is 2, right. So, count

colon so we will get the output 2 right so when we run the code so we have to get this output right so this is called as a singleton counter that means i am creating an instance correct and if the instance doesn't exist That means, it is pointing to null pointer. So, we are creating a new instance fine all right or if the instance already exists whatever be the case line number 15 we are returning instance under get instance member function right it is a pointer function right. So, rest of them are clear. So, I am creating an object in line number 24 pointer object and in the right hand side get instance and we know that get instance will return instance all right.

So, therefore, when counter is invoking increment. So, count is incremented by 1 and then again it is calling count will be incremented by plus 1. So, 1 plus 1 it will become 2. So, when I run the code we will get the output count colon 2 ok.

So, this is what the output we are getting. So, this is called as the singleton counter in C++ right. So, what is the main purpose of explaining the previous code? So, when I am talking about the single counter it maintains a global counter that can be incremented or retrieved so increment case we can see so simply if you put count so it can be retrieved whatever be the count value so the count value will be printed so incremented or retrieved all right and the main features are the count variable is shared globally via the singleton instance yes i did not talk about the global variable



Right, you might be asking the question: So how, when the second time it is calling in line number 26, is it being incremented by 1 plus 1? Right? So here you go. When I am using the singleton counter, the count variable is shared globally via the singleton instance. Right? So this is one of the features. Another feature

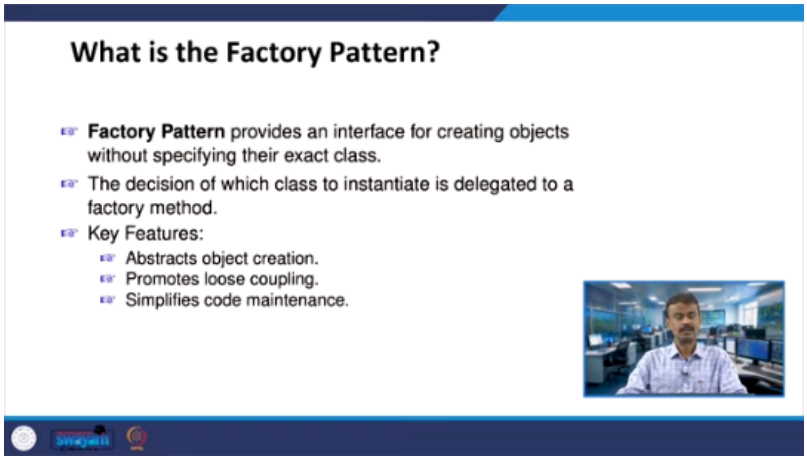
is that the singleton counter ensures that only one counter exists throughout the application. Right. So, only one counter—you will have only one counter. Right. So, these are all the main features. Whenever you have functions like member functions, such as built-in member functions like increment or get count, right? So, it is useful to increase the counter and to retrieve the current count, right? Respectively, right.

So, the increment function is for increasing the counter, and get count retrieves the current count. So, this is all about the singleton counter example. So, now, what are all the benefits and drawbacks of the singleton pattern? So, we have talked about the singleton pattern so far, right?

So, the main advantage is it ensures a single instance of a class globally, all right. So, either if it is pointing to a null pointer, in that case, we are creating an instance; otherwise, the instance will occur. Right. And it reduces the memory overhead by reusing the same instance, right? We are having only one instance, so that will be reused. The singleton pattern provides a controlled access point to shared resources, right? So these are all the main benefits of the singleton pattern. The main drawbacks are: it is difficult to do unit tests due to global access, and the second case— I mean, this singleton pattern can lead to tightly coupled code and thread safety issues. In fact, the thread—everything we are going to study in the next chapter.

So, the thread safety issues if not implemented carefully. So, these are all the main drawbacks of the singleton pattern. So, now the next topic is what is meant by the factory pattern, all right. So, when we are talking about the objects, all right. So, how do you define this as a factory pattern?

So, this factory pattern is providing an interface, all right. So, this interface is useful for creating objects without specifying their exact class, right? So, that is called the factory pattern. And, in fact, when I am talking about the factory pattern, So, the decision of which class to instantiate, right, is delegated to a factory method. So, this is an added advantage of what we have seen so far, right. I mean, in fact, this is another important feature, right.

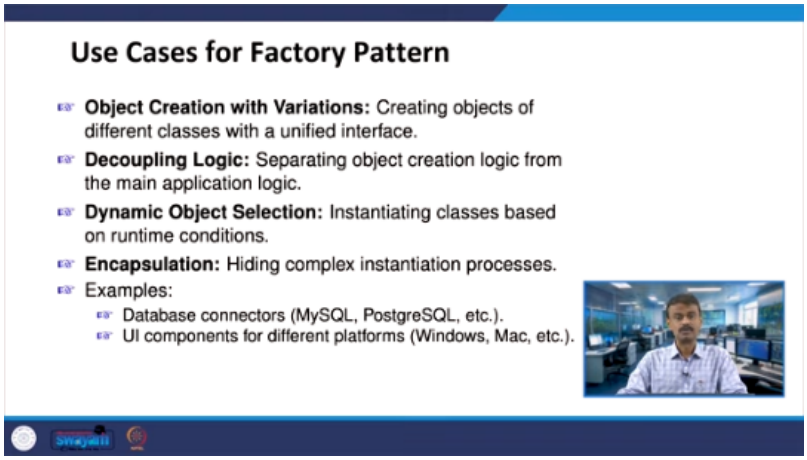


So, better than the singleton pattern. So, here the decision of which class to instantiate, right. So, I assume that you have several classes and then I mean, we have to see the factory pattern will have the decision of which class to instantiate delegated to a factory method. And key features are abstracting object creation, right?

The factory pattern abstracts object creation. This promotes loose coupling and simplifies code maintenance. So, loose coupling is nothing but the connection between systems, right? In programming. So now, what are all the use cases for the factory pattern, right?

So here, you go: object creation with variations. Creating objects of different classes with a unified interface. Right. So, that is one of the use cases. And decoupling logic.

Separating object creation logic from the main application logic. The third point is dynamic object selection. Instantiating classes based on runtime conditions. So, whenever I mean dynamic. So, you have that star.



So, you have a class and then a star object. So, when we use the factory pattern. So, this is instantiating classes based on runtime conditions. The next use case is encapsulation. Hiding complex instantiation processes.

Right, so what are all the couplings we can have? The database connectors: MySQL, PostgreSQL, right? etc. We can do that and user interface components for different platforms like Windows, Mac, etc. You have the factory pattern. So, these are all the use cases for the factory pattern. So, how do I relate this with a real-world analogy? All right, so real-world analogy: suppose you are going to a coffee shop and you want to have a coffee. Alright. So, what do you do? You order a coffee. So, then the barista decides what you ordered.

Whether you ordered a coffee or a tea. Assume that you ordered a coffee. And then you are expecting the output. Alright. Result.

Right. You have to have a very good coffee. But you are not bothering about what exactly the content. Right. Content in the sense how they mix.

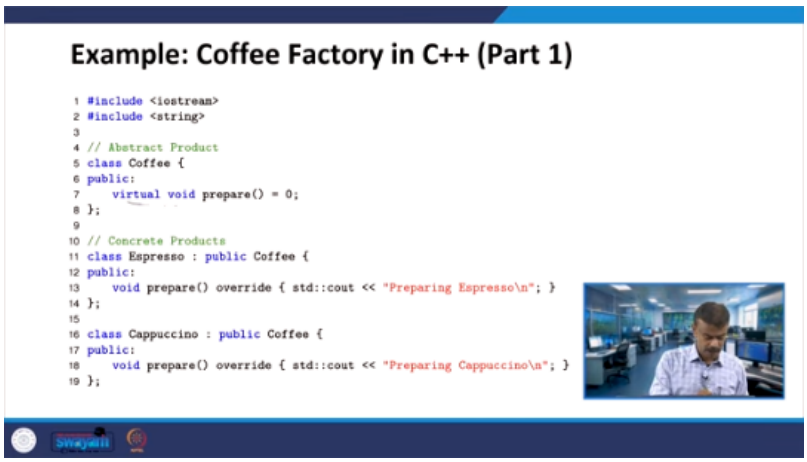
Alright. So, what is the percentage? I do not worry about all this. Right. So, exactly the factory pattern is working like that.

all right so in fact this analogy we call it as abstraction right so the factory pattern works in the same way right so the factory encapsulates the creation logic right so encapsulation means inside so how many percentage of coffee powder has been mixed how many percentage of milk has been added all right how many percentage of sugar was put it in the cup so we don't know right so only I mean I want to have the final output right the final result I want to have a very good coffee right. So, in the same analogy the factory pattern works right. So, you are

encapsulating the creation logic and then the client simply requires an object from the factory right. So, we can take the example with the same the coffee factory right I named it as a coffee factory in C++ right.

So, let us include IO stream and string. So, class coffee. All right. So, you have a pure virtual function called prepare. All right.

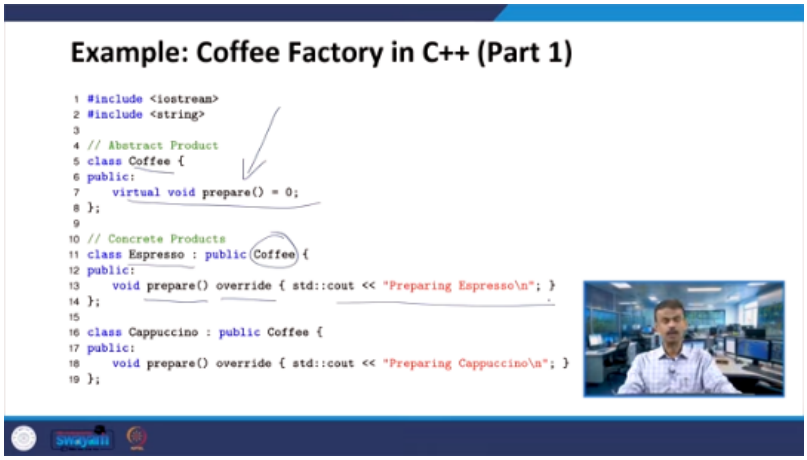
So, that is equal to zero. So, we have seen this pure virtual function. And that's called abstract. Right. The abstract product.



And here, I have class espresso. All right. So, class espresso publicly inherits coffee, the class coffee. Now, I have a void prepare member function. Right.

So already it is there. I define as abstract class. Right. When I put equal to zero, it's abstract class. So this will be obviously overridden.

And then I am printing preparing espresso. Right. Preparing espresso. And the next class. So this is the one class.



So espresso class is publicly inheriting coffee. the upset class and then i have one more class cappuccino all right so cappuccino publicly inheriting coffee again again you have the function prepare right you have the function prepare So, which will overwrite when you are calling this, when you are having the constructor, etc. We had seen many examples. So, see out preparing cappuccino, all right.

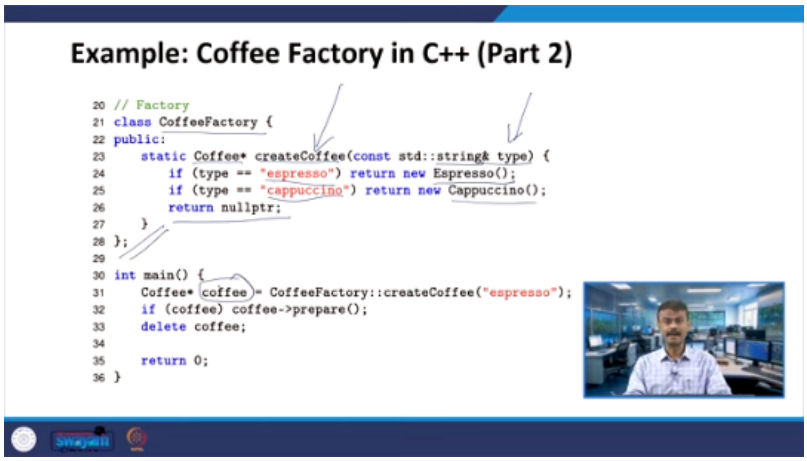
So, see out preparing cappuccino. So, now I have a class, right, the class coffee factory, all right. I have the class coffee factory. So, here you go, I have an object, create coffee, right. It is a pointer object.

under coffee, under the class coffee, all right. So, here you have coffee by pausing the type, all right. So, this is a reference type under string, right. The reference type under string. So, if type is espresso return new that means it is calling this espresso function, all right.

So, in fact the espresso constructor, all right. Otherwise if it is cappuccino return calling this constructor cappuccino right. If both are not there it is returning null pointer all right. So, otherwise it will return null pointer.

So, if the type is let us say espresso it will return this one all right calling this constructor return new this constructor or if it is cappuccino return new cappuccino. If both are not there, it will return null pointer, right? So, this is what exactly happening in this member function which you call it as create coffee, right? It is a pointer object under coffee, right? So, now the class is over here and you are having the main program.

So, in this main program, you have object coffee, right? In fact, the pointer object, right? So, the pointer object under the class coffee, right? So, the right hand side, it is invoking create coffee all right it is invoking create coffee so this is the one right by passing espresso you are passing espresso so if type is espresso it will call this espresso constructor correct so here you go so that means automatically here it will override all right so it will go in fact assume that this object is calling prepare right so now the object that you are creating

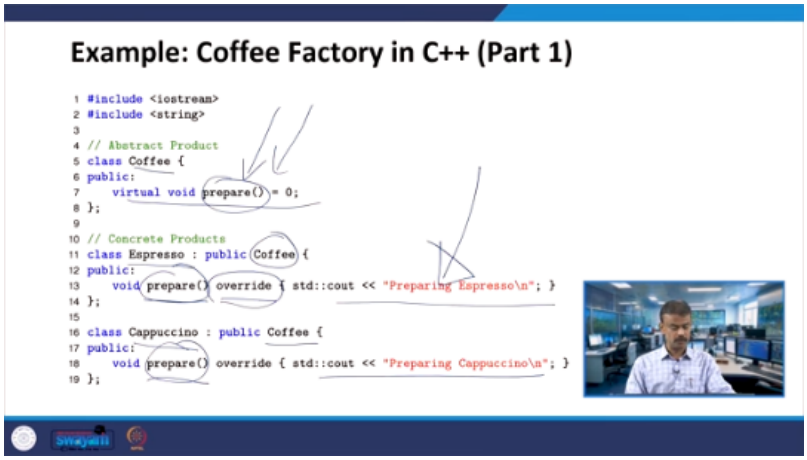


Right. Under coffee. Right. Since you are passing this and it is calling the espresso constructor. So the meaning is, suppose the object is invoking, let us say, prepare automatically.

This will be right overridden, and this will be called. Right. So, that is what you are saying. All right. All right.

And then here you are passing espresso, and then here it is returning new espresso. So, we know now what will happen. Right. So, when— Right, if coffee —so coffee is not pointing to the null pointer because it is returning new espresso, right?

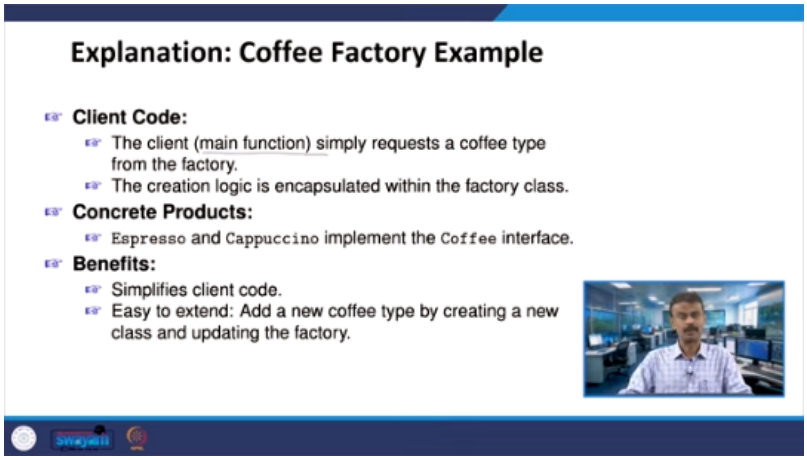
So, it is not null; coffee is not equal to a null pointer. So, therefore, it is invoking the object coffee—the pointer object coffee—is invoking prepare using the arrow operator, correct? So, here you go. So, when it is invoking— So, this particular function will be called, right? The member function prepare will be called because the abstract class coffee has prepare, but it is an abstract class, right? It is a virtual function.



So, therefore, preparing espresso will be printed, right. So, when I run the code, I have to get the output preparing espresso, right, and then I am deallocating the memory. for the object coffee. So, the delete coffee is indicating that I am deallocating the memory. So, then you are having return 0 right.

So, that means I will get the output preparing espresso right. So, I hope it is clear for everyone. So, coffee factory example itself we had seen and this example right useful for learning factory pattern right. So, you have learned factory pattern. So, in the previous example

Right. Client code. What do you mean by client code? So the main function, the client is nothing but a main function, simply requires a coffee type from the factory. Correct.



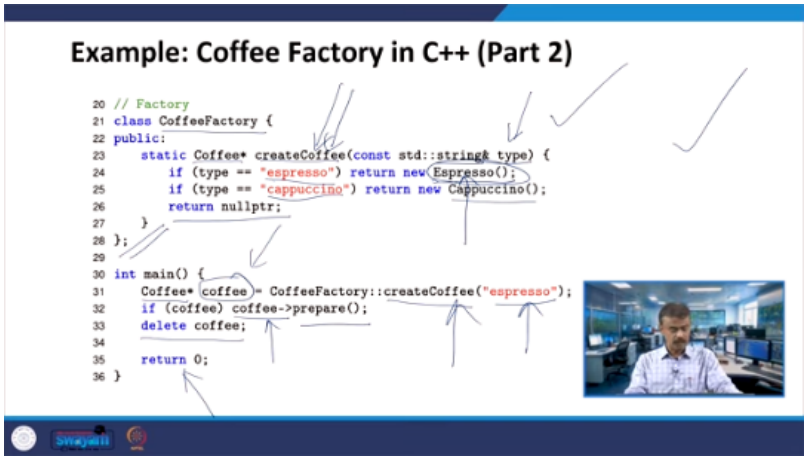
So that is exactly what the main function is doing. Here you go. The main function is simply requesting the coffee. All right. So you have coffee that is a pointer object under coffee, and you are ordering create coffee espresso.

That's it. Exactly. Exactly the same analogy. Right. So you are creating espresso.

So once you do this, what is going on inside? Right. So you are passing espresso. So if the type is equal to espresso, the new espresso will be called. Right.

The constructor will be called. So when you are overriding, Correct. When you are creating an object like this, Yes, I am creating an object by calling create coffee.

All right. The object is a pointer object in line number 31 under the coffee class. Correct. So, when in line number 32, if coffee, Right.



So, coffee is not equal to a null pointer. Right. Because it is returning espresso. New espresso. Line number 24.

Right. So, when it is invoking. So, we know that espresso is calling that particular constructor on line number 13 to prepare. Because coffee is invoking that, preparing espresso will be printed. Right.

So this is what exactly is called the client code. So the main function is simply requesting a coffee type from the factory. So then you have encapsulated all the logic in the factory class. So that is called the factory class. So if you look right over here, yes, coffee factory.

Right. In the factory class. So you have encapsulated everything. Correct. So.

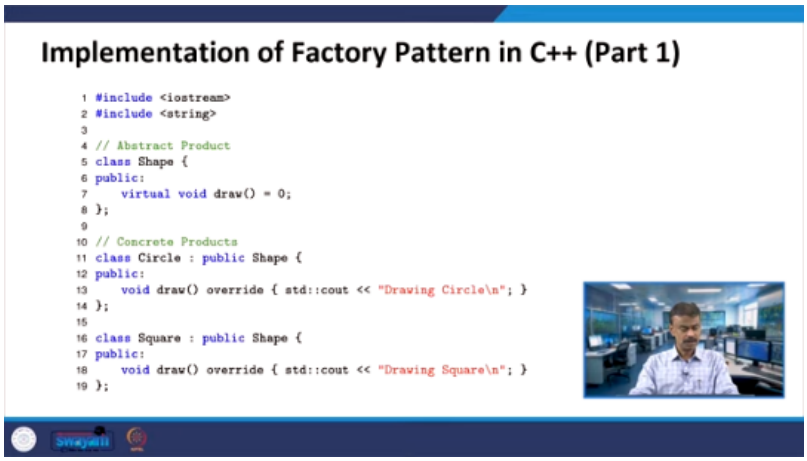
Your line number 21 is a factory class up to line number 28, where you are encapsulating the complete logic, all right. So, this is exactly what we have done,

and the concrete products are espresso and cappuccino, right? So, they implement the coffee interface, right? So, here you have the interface, correct? The class Coffee, which is abstract, correct? So, here, espresso and cappuccino, if you look.

So, they are inheriting from the Coffee. So, this is what I am talking about in theory, right? Espresso and cappuccino implement the Coffee interface or inherit from the abstract class, all right. So, what are all the benefits? So, it simplifies the client code, right? The main function. So, in the main function, you are just ordering, all right.

So, what exactly are you doing, all right? Create coffee, you are asking to make a coffee, espresso, all right. So, that is what simplifies the client code, and it is easy to extend. So, initially, we have a Coffee class, and then we are extending it with new types: cappuccino and espresso. So, we have added a new coffee type by creating a new class, and we have also updated the factory class. So, I hope it is clear for everyone.

So, now let us consider one more program for the factory pattern. So, in the factory pattern, So here we have the class Shape. All right. So here we have class Shape.

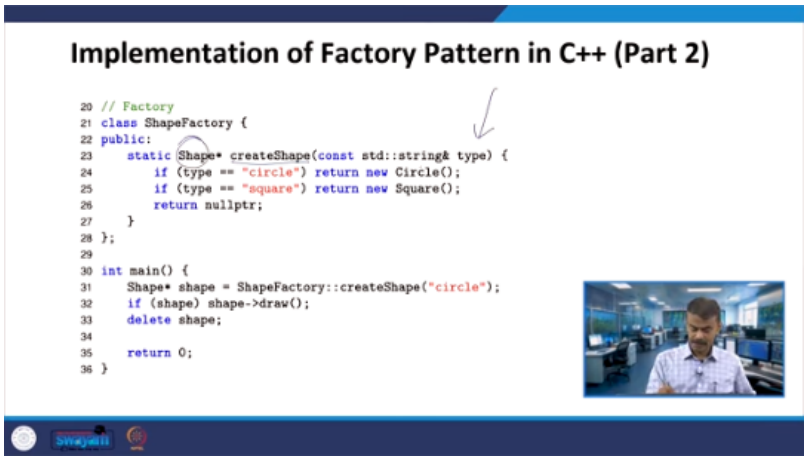


So here is the pure virtual function draw. All right. So that means I have to inherit this in the subclass. So here you have the class Circle. So, publicly inheriting Shape.

All right. So the draw will be overridden with 'cout' drawing circle. All right. And I have another shape called Square. So, publicly inheriting Shape, right.

So, here you have void draw. So, which will also be overridden and then here you have a print drawing square, right. The class is over here for square, the class is over here for circle and the class is over here for shape, the upset class, right. So, now shape factory, right. So, I create the shape pointer object under shape, right by pausing certain type.

right. The type is nothing but the reference type which is a string, right. So, if type is equal to circle exactly like a previous program type is equal to circle return new circle that means it is calling this constructor, right. So, circle is the class here, right. If you are passing square return new square otherwise it will return null pointer, right.



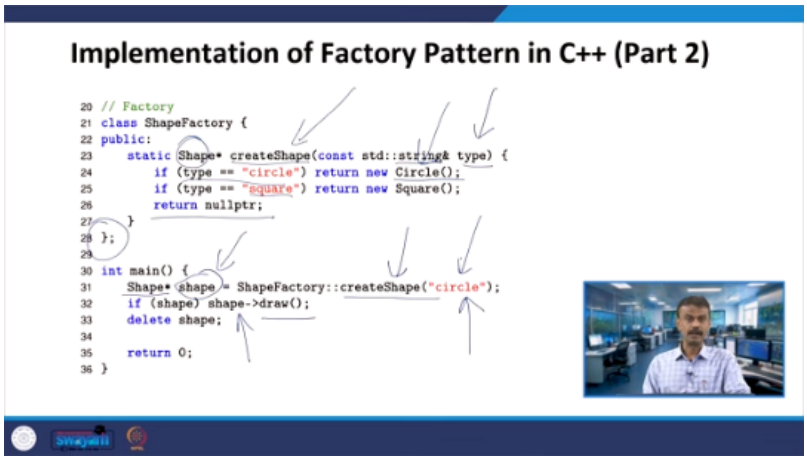
So, your factory class is over here line number 28 shape factory. So, now int main I have a pointer object shape right. I have a pointer object shape. So, under in fact this is a pointer object under shape right. So, right hand side create shape will be invoked by passing circle.

Now you are passing a circle, right? So, when I am passing a circle, this particular function will be invoked, and you are passing a circle. Therefore, the type will be a circle, right? So, if the type is a circle, return a new circle, right? That means this particular constructor will be called, meaning it will be in the class Circle. So now, when you are creating an object—yes, I have created an object, Shape, which is a pointer object, right? And if it is invoking, obviously, if Shape...

So, Shape is not equal to a null pointer because it is returning The new circle, line number 24. So now, when it is invoking, Shape is invoking draw. So, as I

said, this will be going to this particular In fact, this draw will be overridden, and it will go over here.

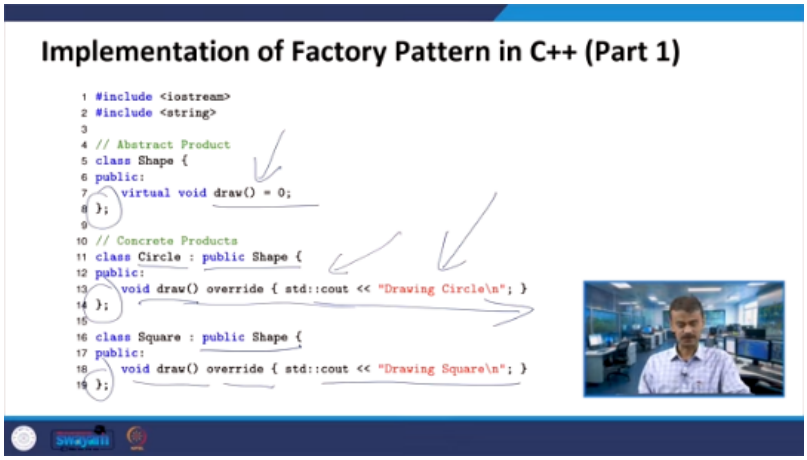
And "drawing circle" will be printed. Right. So, when I run the code, I have to get the output "drawing circle." Right. So now, you delete Shape.



Right, that means deallocating the memory for the object shape. Return 0. So, when I run the code, I will get the output: drawing circle. So, this is another example of a factory pattern, right? So, in the next lecture, what can we do? So, we will talk about the same factory pattern program with the help of Java, okay?

So, we have seen in the case of C++. So, we will see in the case of Java. So, the factory pattern—if you look—we have seen two different examples, right? One is the same coffee factory example itself, all right. Another one, mathematically, if you take some shape, all right.

So, let us say circle or square. So, now you can understand how this factory pattern in C++ is working. So, we have done two different implementations. So, with this, I will conclude this lecture. And we will see the same factory pattern example in Java in the next lecture.



Thank you very much.