Object-Oriented System Development Using UML, Java and Patterns

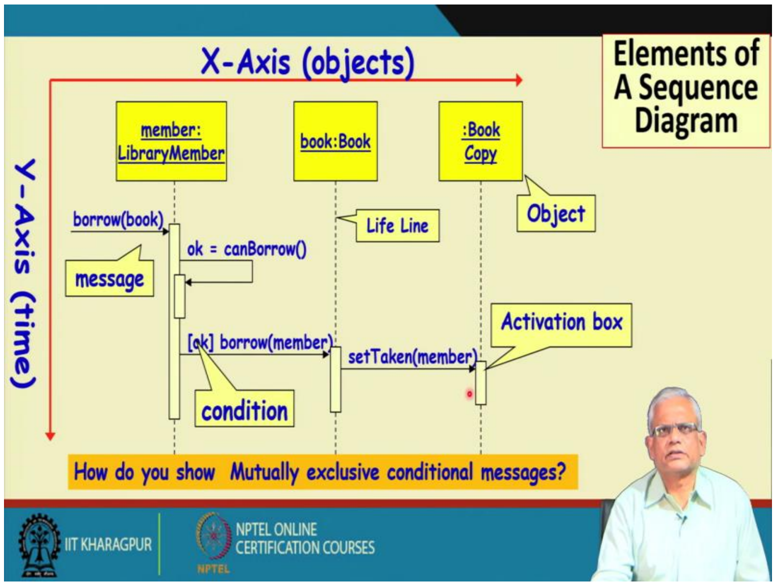
Professor Rajib Mall
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture 26
Sequence Diagram - I

Welcome to this session.

In the last session we were discussing about the interaction diagrams. Of the three interaction diagrams, we said that the sequence diagram is the one which we will be using for the simple problems that we are going to solve and the collaboration diagram or the communication diagram we can automatically get from the sequence diagram without much effort. We were looking at the different aspects of the sequence diagram and we are trying to model some simple examples.

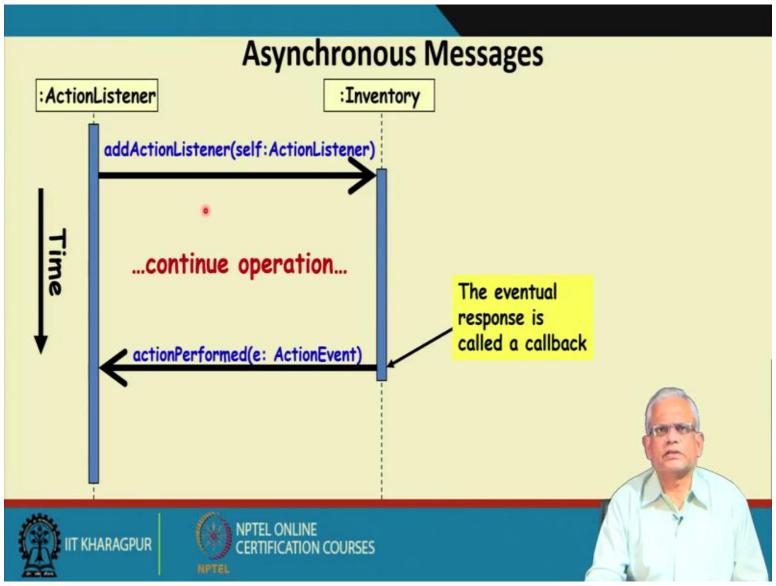
(Refer Slide Time: 1:01)



Now, let’s proceed from there. Today we will try to complete the sequence diagram and look at the activity diagram. The sequence diagram we had said is a two-dimensional chart (as shown in the above slide). Along the X-axis we write the objects at the top which exist before a use case starts. Each sequence diagram models the object interactions that occur during the execution of a use case. And here the top are the objects that forms the x axis. The y axis is the timeline and starting from time 0 as the use case starts executing the different interactions among the objects is shown here.

The top of the sequence diagram are the objects. For example, in the above slide Library Member, Book etc. are the objects. The message or the method call also shown in the above slide. For example, borrow(book), canBorrow() methods are called by method call. In the above slide, [ok] is the condition. The lifeline which is shown as a dotted line and also the activation once an object receives a message there is a method call the object becomes active. In a condition a variable is used (‘ok’ is the variable), and depending on the value of the variable, either one of the methods is called one object or another method is called on a different object.

(Refer Slide Time: 3:05)



Asynchronous Messages

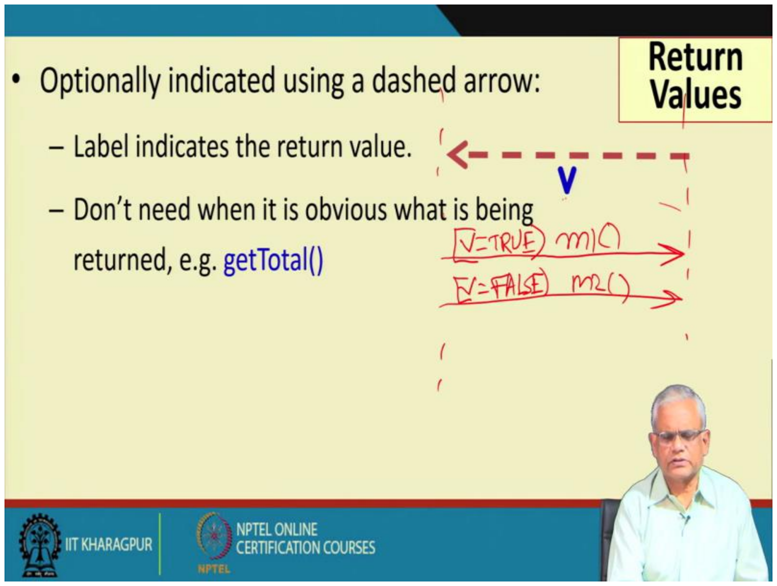
:ActionListener
:Inventory
addActionListener(self:ActionListener)
...continue operation...
actionPerformed(e: ActionEvent)
Time
The eventual response is called a callback
IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES

So far, we have been using the filled arrowhead, that is a method call but sometimes we use the open arrowhead, which represents asynchronous message. In a synchronous message once, an object calls a method of another object, it blocks, the execution of the other method starts and until the control returns back, other method just waits. And that is essentially the synchronous method call which we are all familiar with typically happens in a Java execution.

But sometimes we need asynchronous messages. Typically happens when some event is response is reported. For example, in the above slide, there is an action listener, and this calls on the inventory (AddActionListener()), so it adds itself, but then it continues executing, does not wait for the return messages from the inventory. And at some time, it may have a call back for some action, performed. The asynchronous messages typically occur in a client server situation where the client makes a method call, but locally continuous execution in the server, and later response. And also, in event-based programming the event is reported but it continues listening to further events.

We will not use the synchronous message as much here in our examples. We will be restricting ourselves to the asynchronous messages.

(Refer Slide Time: 5:17)

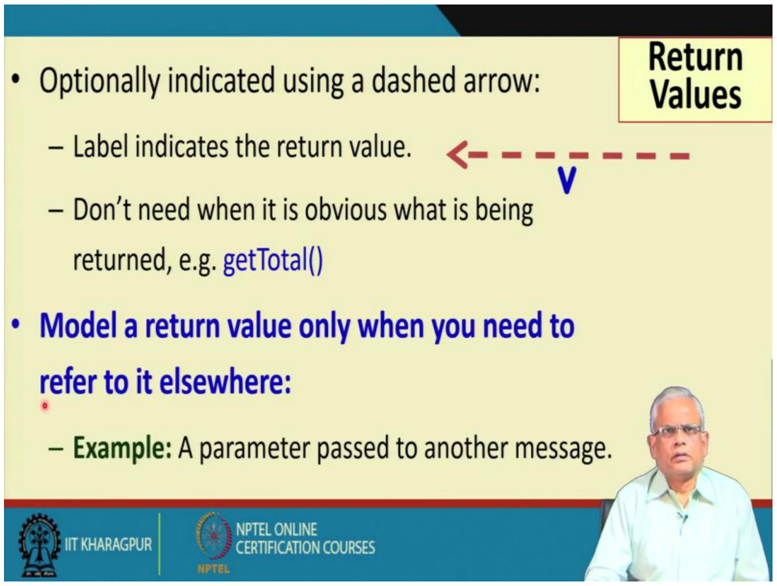


Return Values

- Optionally indicated using a dashed arrow:
  - Label indicates the return value.
  - Don't need when it is obvious what is being returned, e.g. getTotal()

Another aspect of the sequence diagram is the return value from a method call shown as a dotted arrow (in the above slide). We do not usually show the return value, it is implied, but then if some value is explicitly used later in the sequence diagram. We typically use V to represent return value. For example, if we have a sequence diagram and we have this return value V and then depending on the value V, like [V = TRUE] then call method m1 on an object else, call method m2 maybe on the same object or on a different object. And here we are using the return value V (in the above slide) and therefore, we show that from a method call the value is returned V. So, using a dotted arrow we represent return. But unless we explicitly use the variable V, we don't show it.

(Refer Slide Time: 7:14)

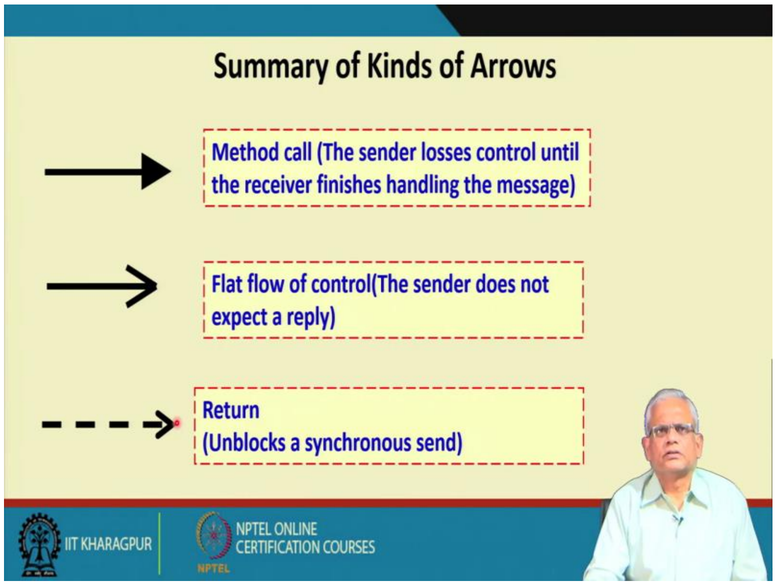

Return Values

- Optionally indicated using a dashed arrow:
  - Label indicates the return value.
  - Don't need when it is obvious what is being returned, e.g. getTotal()
- **Model a return value only when you need to refer to it elsewhere:**
  - **Example:** A parameter passed to another message.

V
IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES

We use a return value only when we need it elsewhere. V may be a parameter which is pass to another function, another method calls and so on. It can be used as a conditional also (we already saw that). We explicitly mentioned the return value using a dotted open-ended arrow, otherwise this becomes implicit.

(Refer Slide Time: 7:46)


Summary of Kinds of Arrows

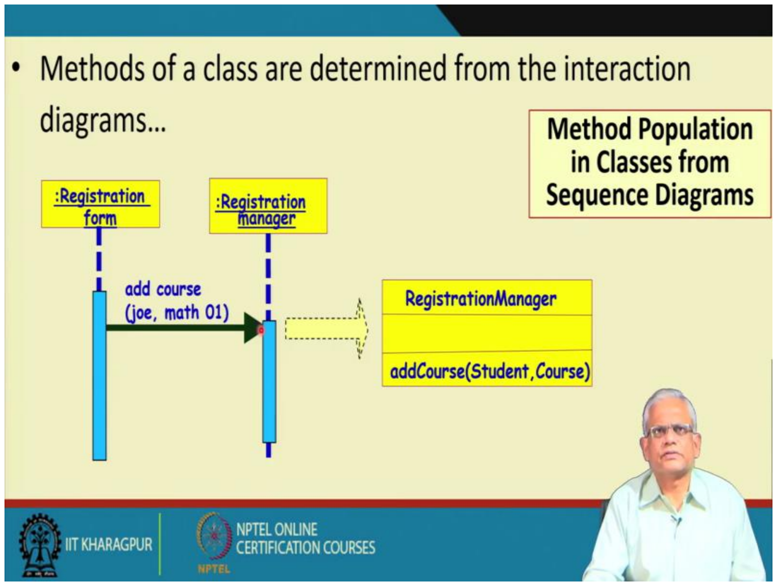
Method call (The sender losses control until the receiver finishes handling the message)

Flat flow of control(The sender does not expect a reply)

Return (Unblocks a synchronous send)

Now, we will go through different types of arrows in a sequence diagram. One is the filled arrowhead, which typically we use in making Synchronous method calls. The other one is the open arrow head, which is an asynchronous method call, here the sender does not wait for the receiver object to respond. In a synchronous message once, a sender makes a call to another object it waits until the receiver returns the result. Another arrow is the dotted arrow with open arrow head is the one used to show return values on a method call. In a synchronous method call, the sender loses control until the receiver finishes handling the messages. But in an open arrow, the sender does not wait for a reply, it continues executing. The last one (dotted open arrow head) is the return value it explicitly unblocks the object, otherwise it's implicit.

(Refer Slide Time: 9:28)



- Methods of a class are determined from the interaction diagrams...

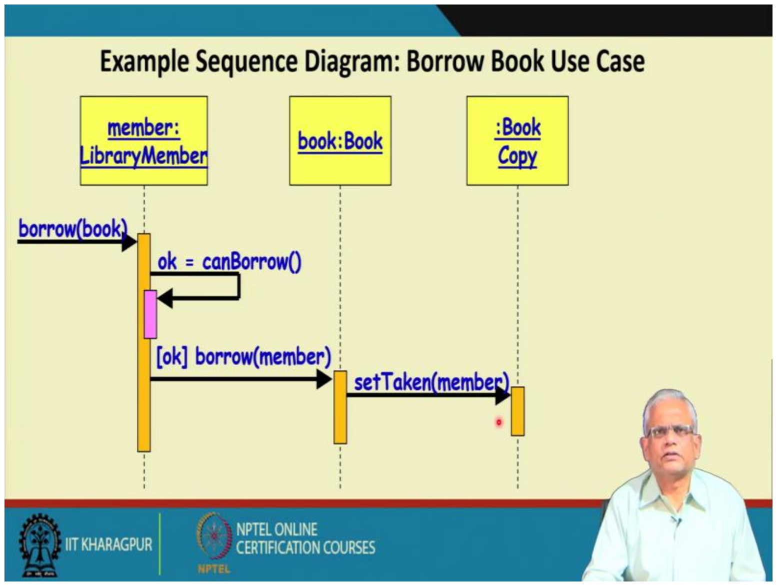
Method Population in Classes from Sequence Diagrams

:Registration form
:Registration manager
add course (joe, math 01)
RegistrationManager
addCourse(Student,Course)
IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES

Now, let see an important use of the sequence diagram. So, far we have been mentioning that sequence diagram is very important in any design solution. We had said that for every use case we develop one sequence diagram and for complex use cases we draw multiple sequence diagram which are link together by an interaction over view diagram. And once we draw the sequence diagram in a case tool, the methods in the classes are automatically populated, as we draw them on the case tool. For example (in the above slide) we draw the 'Registration form' object sends a synchronous message to the 'Registration manager', which is add course with some parameters. Now this diagram implies that to call the registration manager with the add course, the add course method must be supported by the registration manager.

So, once we draw the sequence diagram on a case tool, it will automatically populate the Registration manager with add course() method with student, and course number. It is important to observe here that the add course() is not a method on the registration form, the registration form class just makes the method call on the registration manager and therefore the add course must be supported by the Registration manager. And therefore, the registration manager class must support the add course() method. This method allocation to a class is done in most case tools as we draw the sequence diagram, the methods get populated in the class diagram. So, this is an important use of the sequence diagram, that is allocation of the methods to the classes. Let me repeat that, once we have identified the classes, we proceed to draw the sequence diagram for every use case. And as we draw the sequence diagram methods are allocated to different classes. And therefore, the sequence diagram serves to populate the methods on the class diagrams.

(Refer Slide Time: 13:03)

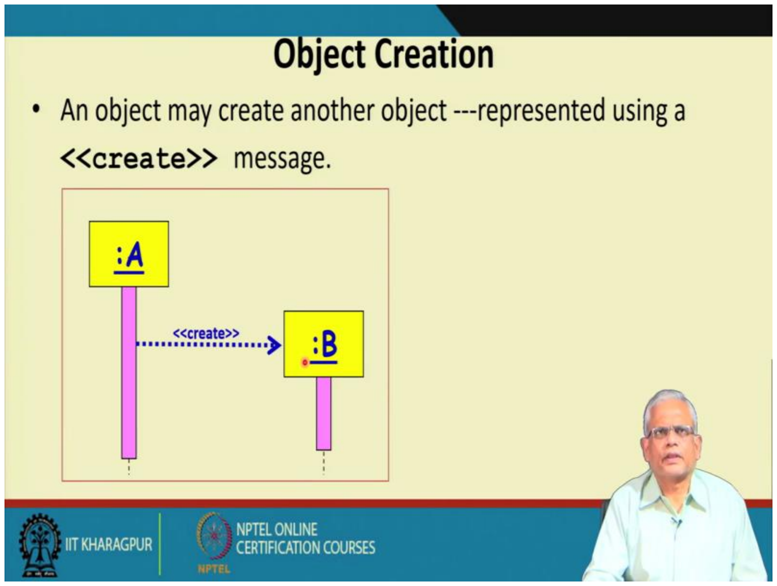


Example Sequence Diagram: Borrow Book Use Case

Now, let see another example of sequence diagram. Here this is a library member and :LibraryMember is any member of the library member class. And since it is written member here, that means this is a specific object named or whose ID is member. And :Book is a specific book object.

In the above slide, these are all synchronous messages. The borrow(book) is called on the Member. And the member checks by calling a method of itself, (canBorrow()), the canBorrow() checks whether the member's quota is not exited and also whether the membership is still valid and so on. And based on that, it returns ok, and if the ‘ok==true’, then the member object calls, the borrow(member) (here member is the own ID of member object) and calls the borrow methods of the book and then the book object issues one of the book copies. There are several copies of the book object for example, there may be 10 copies for the same book and one of the books it issues out and calls the setTaken(member) on the book copy and therefore the book copy remembers which member has issued the book. If we see here the different methods to be supported by the library member are borrow(), canBorrow() and also the canBorrow() returns a boolean value and inside the borrow calls the borrow member on the book object and therefore the library member class must support the borrow() method and also the canBorrow() method. The book class must support the borrow() method because method call ‘borrow’ is called on book object also and the book copy must support the setTaken() method.

(Refer Slide Time: 16:25)



Object Creation

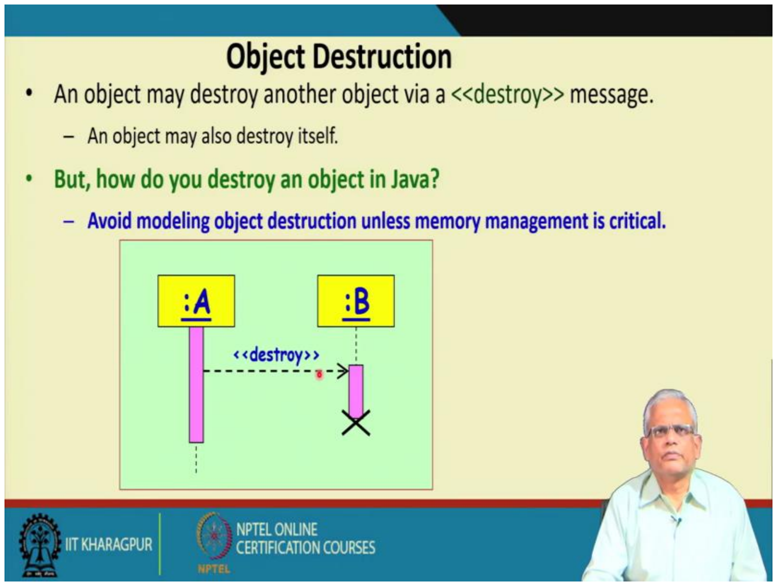
- An object may create another object ---represented using a **<<create>>** message.

:A
<<create>>
:B
IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES

During an execution of use case, some objects may be created. For example, let’s say a ‘create member’ use case in that a member after receiving the fee and validation and so on, the member object is finally created.

At the top of the sequence diagram, all the objects are mentioned which exist before the use case starts, and during the use case, some objects get created at a later time after the use case starts. To create an object, we make a method call (as shown in the above slide). Observe here in the above slide, this is the dotted open arrow and this is stereotype with a create. We use this notation: a dotted, open arrow with a create stereotype on the object for creation of the object. In the above slide, we can see that the create message is sent to the object of class B.

(Refer Slide Time: 17:52)



Object Destruction

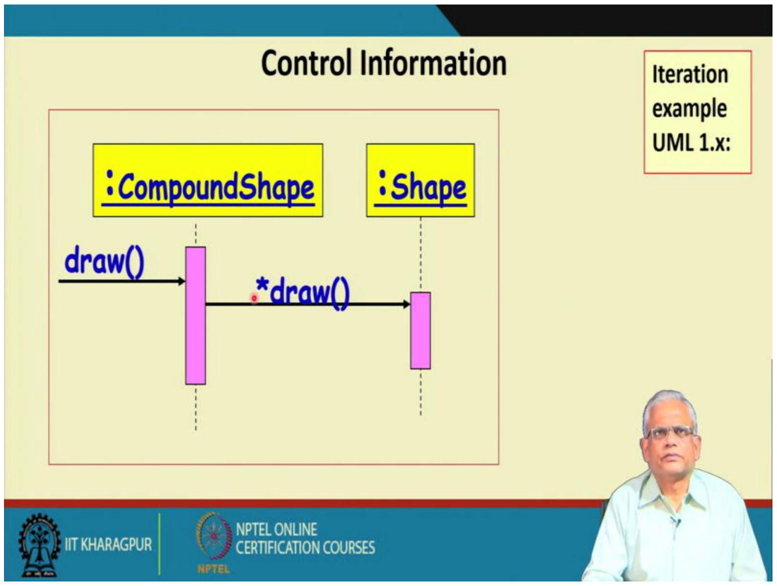
- An object may destroy another object via a <<destroy>> message.
  - An object may also destroy itself.
- But, how do you destroy an object in Java?
  - Avoid modeling object destruction unless memory management is critical.

:A
:B
<<destroy>>
IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES

We can also represent object destruction using a sequence diagram (in the above slide). An object of class A here calls the destroy method of the object of Class B, and then the object gets destroyed, which is represented by crossing its lifeline.

But then, how does one destroy an object in Java? In C++ we know that we can call the destructor of an object and object gets reallocate. But how does one destroy an object in Java? Okay, in Java, we do not have a destructor, but then the objects fall out of scope. As soon as they fall out of scope, they are garbage collected. But even though we are representing this destruction and so on. We don't really show destruction in the models unless the memory management is critical. In typical examples that we will discuss and even in slightly moderate examples that you might be solving in industry, the object destruction is typically not used unless it is an embedded application. In embedded application, the memory management is very crucial.

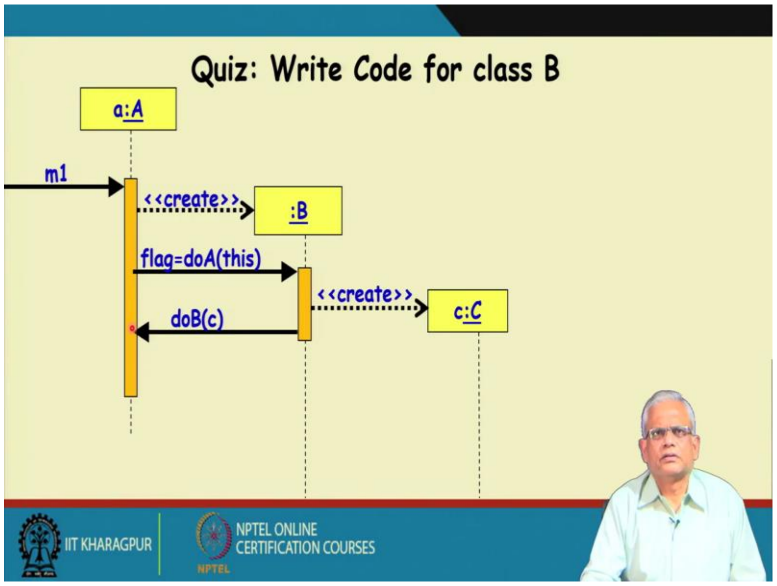
(Refer Slide Time: 19:36)


Control Information

Iteration example UML 1.x:

We had discussed about the control information before. Here in the above slide, a compound shape may consist of many shapes like, some rectangles, triangles, lines and so on. Now, let say we want to refresh that redraw then here should call the draw method all the component sets and that we show using a * in the UML 1.x notation.

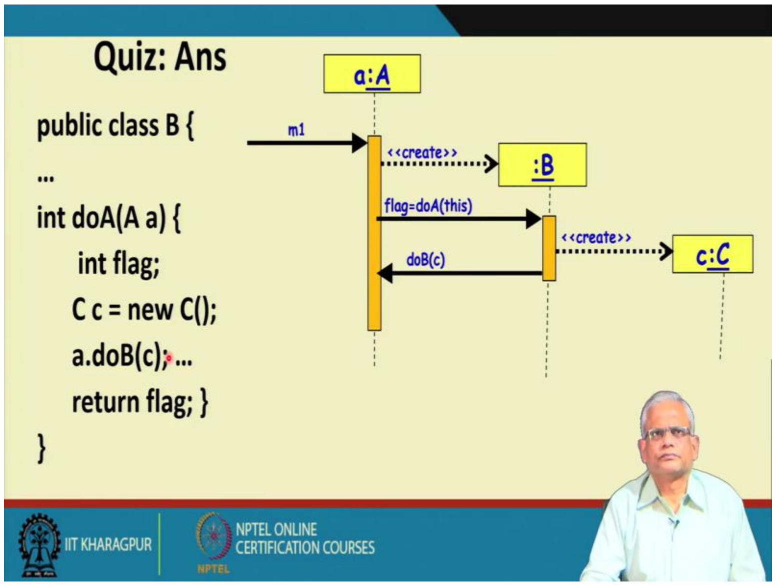
(Refer Slide Time: 20:15)


Quiz: Write Code for class B

Now, let's write the code for Class B (shown in the above slide). In the sequence diagram m1 is the method call on the class A object, and after it receives the m1 method call, it creates a B object. And then it calls the doA(this) on the B object and the B object in turn creates a C object and then returns the idea of the C object. So, can we write the method population and the code that can be generated for the Class B?

Of course, it must support the doA() method and that must take an object of class A is parameter and return a boolean and inside the method it creates an object of class C and the idea of that class it returns to the class A. It calls the doB method on the class A. So, inside class B must support d A method which returns a boolean it takes an argument which is an object of class A and then it creates an object of class C and returns the id by calling the doB() method. It calls the doB() method of Class A with C as the parameter.

(Refer Slide Time: 22:32)

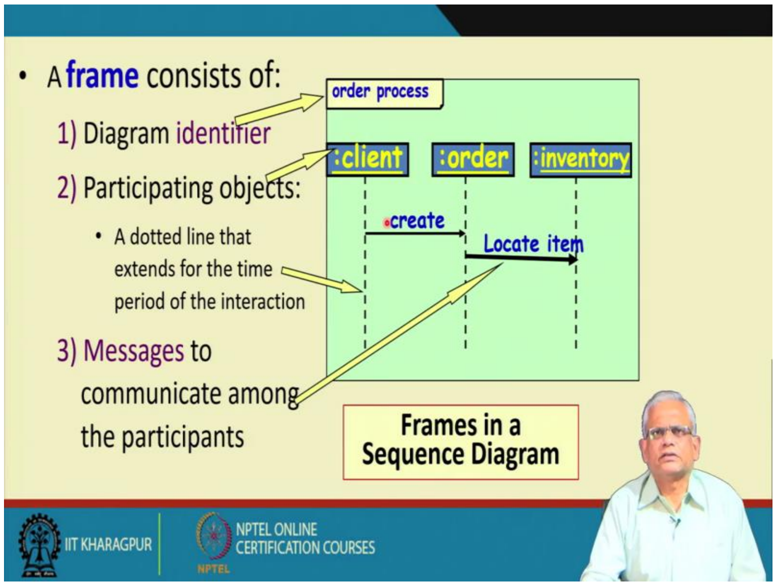


Quiz: Ans

```
public class B {
...
int doA(A a) {
    int flag;
    C c = new C();
    a.doB(c);...
    return flag; }
}
```

Now, let's write the code for that (as shown in the above slide). So, Class B might have several other methods based on the other interaction diagrams, but based on this sequence diagram, it will have the doA() method and it takes an object A as the parameter. And it creates an object of Class C calling its new method. It calls the new operator on class C, and then it had the ID of the object A and it calls a.doB with C is the parameter (a.doB(c)) and it returns the flag. Of course, the flag must be set somewhere, this is just the skeletal code needs a little bit of refinement, but this code can be automatically generated for the class B once we draw this diagram, only small parts need to be filled. Most of the code that it obvious from this diagram can be represented in a method.

(Refer Slide Time: 24:10)



- A **frame** consists of:
  1) Diagram identifier
  2) Participating objects:
     - A dotted line that extends for the time period of the interaction
  3) Messages to communicate among the participants

From UML 2.0 onwards, the frames were introduced in sequence diagram. The frames are an elegant concept. It helps in many ways, let’s look at a frame (in the above slide). In the above slide, frames are shown and, in the frame, we write the name of the use case to which it represents earlier remember in the use cases we did not have any annotation of the use case. Here a sequence diagram is enclosed within a frame and it can contain several other frames. And at the top level of the sequence diagram, we write the name of the use case. Name of the use case is the diagram identifier; this the order process use case for which this diagram is being drawn.

And then we already know that these top once (:client, :order etc.) are the participating objects in this use case execution, and this is the lifeline (the dotted line). And this is the communication among different objects (arrows). But then a frame can contain other frames and which can represent different operations like Loop, if else, concurrency and so on,

We have just looked at the top-level frame and this top-level frame may contain other frames inside it, and that greatly makes the diagram easier to understand. And we can represent certain aspects which were not possible to represent using UML 1.x by using the frame notation. We will point that out in the next session that how we will use the different operations and their arguments in a frame.

We are almost at the end of this session, we will stop here and continue in the next session.

Thank you.