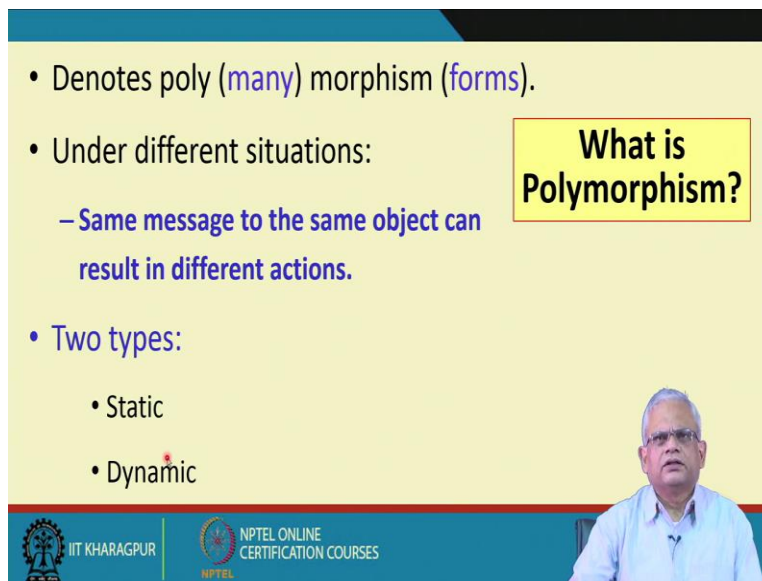


Object Oriented System Development Using UML, JAVA and Patterns
Professor Rajib Mall
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur
Lecture 18
Polymorphism

Welcome to this session.

Over the last few sessions, we had looked at classes various relations between classes and how to represent them on a UML diagram and with this opportunity we will just revisit some basic concepts which are relevant to objects and classes and one of that is polymorphism.

(Refer Slide Time: 00:44)



The slide features a yellow background with a blue header and footer. A yellow box with a red border contains the title "What is Polymorphism?". The main content consists of three bullet points: "Denotes poly (many) morphism (forms).", "Under different situations: - Same message to the same object can result in different actions.", and "Two types: • Static • Dynamic". A small video inset of Professor Rajib Mall is visible in the bottom right corner. The footer includes the logos of IIT Kharagpur and NPTEL Online Certification Courses.

- Denotes poly (**many**) morphism (**forms**).
- Under different situations:
 - Same message to the same object can result in different actions.
- Two types:
 - Static
 - Dynamic

We had started discussing in the last lecture about polymorphism.

It is an important object-oriented concept, literally poly is many and morphism is forms. In other words, an object may occur in many forms and the same object behaves differently to the same message. When the same operation is invoked on an object it behaves differently. so that's the implication of polymorphism in the context of object orientation that is the same message to the same object can result in different actions.

There are basically two types of polymorphism that are used: one is the static and the other is dynamic. In static polymorphism the behaviour is decided at the compile time. In static

polymorphism it's defined behaviour for an object at the compile time whereas the dynamic polymorphism it is define dynamically at the runtime.

(Refer Slide Time: 02:33)

```
Class Circle{  
    private float x, y, radius;  
    private int fillType;  
    public create ();  
    public create (float x,float y, float centre);  
    public create (float x, float y, float centre,  
                  int fillType);  
}
```

An Example of Static Binding

IIT KHARAGPUR NPTEL ONLINE CERTIFICATION COURSES

Let's, further explore this. The static binding is defined at the compile time. In the above slide, there is a circle class and we the create method is called on the circle class depending on the parameters of create. Different behaviour will be seen for the same operation create.

In the first case, a circle object will be created, in the second one, it is with a specific location and with specific dimension and for the third one, not only a circle is created with specific dimensions but also a specific fillType.

We can think of that the same create operation behaves differently for the same circle class. This can be decided at the compile time by looking at the parameters of the create message. The other name of this is static binding and method overloading. The create method is overloaded with 3 types of behaviour and we also called it as static binding because it is decided which method will be invoked on the compiled time.

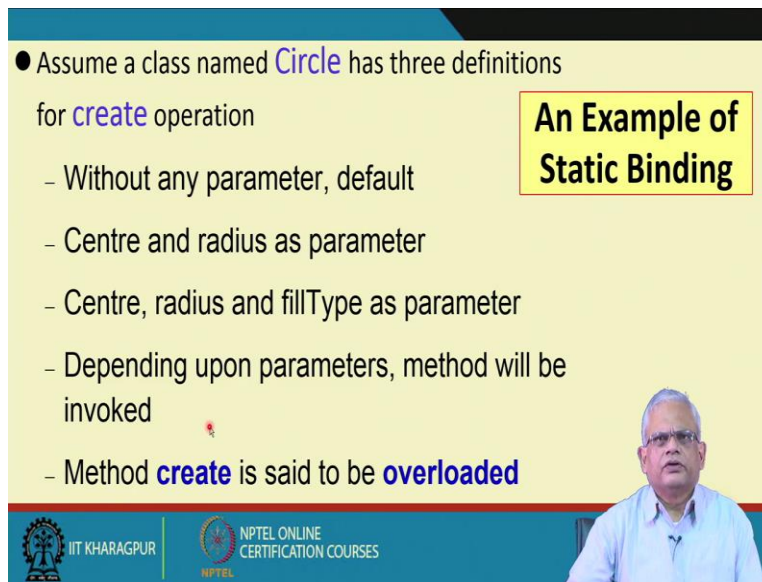
But in the dynamic binding that cannot occur we do not know which method will be invoked until at the point when it is needed at the runtime.

(Refer Slide Time: 04:48)

● Assume a class named **Circle** has three definitions for **create** operation

- Without any parameter, default
- Centre and radius as parameter
- Centre, radius and fillType as parameter
- Depending upon parameters, method will be invoked
- Method **create** is said to be **overloaded**

An Example of Static Binding



In the static binding example, we just saw that there are three create method for the same create operation and then it is decided at the compile time, which method will be invoked and the create method is overloaded.

(Refer Slide Time: 05:20)

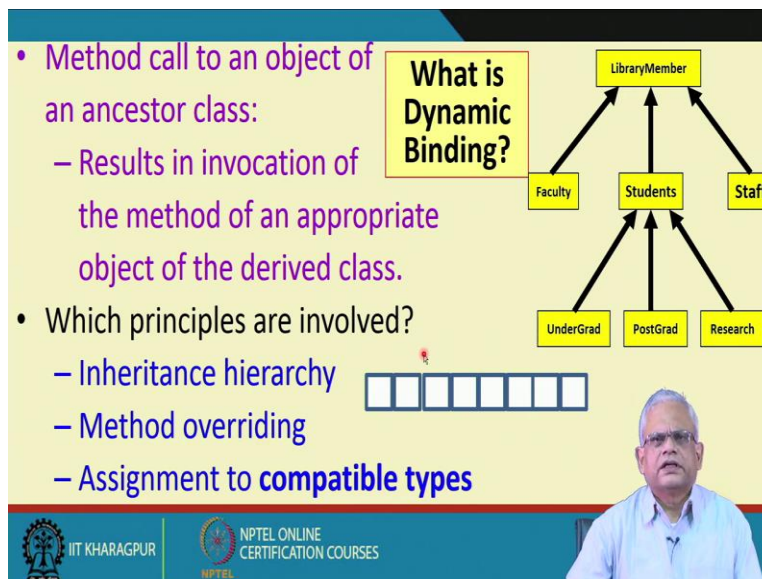
• Method call to an object of an ancestor class:

- Results in invocation of the method of an appropriate object of the derived class.

• Which principles are involved?

- Inheritance hierarchy
- Method overriding
- Assignment to **compatible types**

What is Dynamic Binding?



But in the case of dynamic binding, we need a class hierarchy for the dynamic binding to occur and when we invoke a method on a base class then depending on the situation an appropriate method of the object of the derived class is called. Let me just repeat that again.

In dynamic binding when a base class method is called the actual method that it is bound depends on the runtime situation and a method of the derived class may get called. Here the principles that are involved are an inheritance hierarchy. We need an inheritance hierarchy and if we invoke a method of the library member depending on the runtime situation either the corresponding method on the faculty, student or undergraduate, postgraduate et cetera may get called (diagram shown in the above slide).

The other requirement for dynamic binding is that the same method must be overridden in the derived classes and that's how when we call a method on the base class the overridden method in the derived classes may get called.

The other principle involved here is assignment to compatible types. Some assignments object assignments or object substitutions are compatible that is one object can be substituted for another object but some are incompatible they cannot be substituted for another object.

So, we will see these three basic principles for the dynamic binding. Let say, we define an array of library members (in the above slide) and then we create various types of objects of type faculty members, undergraduate members, post graduate members, staff members and so on. Now, these objects are residing in that array. Basically the array or the array list is of the type library member and now if we invoke an overridden method here, let say 'issueBook' is overridden method of the library member and then the faculty can issue let say 10 books undergraduate student can issue 5 book, post graduate 7 book, research let say 8 books. Now depending on the object that is residing here in this array, the corresponding issue method will be called. If it is a faculty object, which is residing in this array, then the faculty issue book will be called even though the type of the array is library member. So, the basic principle here that even though the array type is defined to be library member the specific method that will be called depends on which object is residing in this array.

(Refer Slide Time: 09:48)

● Principle of substitutability (Liskov's substitutability principle):

Compatible Types

- An object can be either assigned to or used in place of an object of its ancestor class, but not vice versa.

A
↑
B

A a; B b;
a=b; (OK)
b=a; (not OK)

IIT KHARAGPUR NPTEL ONLINE CERTIFICATION COURSES

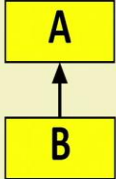
To get a better idea in this dynamic binding we need to understand, what is the principle of substitutability which is also called as Liskov's substitutability principle. Here the principle says that "an object can be either assigned to or used in place of an object of its ancestor class, but not vice versa." That means a derived class object can be assigned or used that is substituted for an object of its ancestor class, but an ancestor class object cannot be used for a derived object. We can illustrate it here, that if B is a derived class of A then a is an object of class A, b is of class B and then if we say $a = b$, then it is okay. So, we can substitute a derived class member for a base class member. We can also call a parameter where 'a' is expected and we pass 'b' that's also okay. But when 'b' is expected and we pass a, that's not okay. So that's the principle of substitutability. A derived class object is compatible with the base class object, but not vice versa.

So, the compatible type says that, a derived class object is compatible with its base class object but not vice versa.


(Refer Slide Time: 11:46)

Liskov Substitution Principle (Barbara Liskov, 1988)


- If for an object a of type A there is an object b of type B such that A is a subtype of B :
– Then it is possible to use b for a .




A a ; B b ;
 $a=b$; (OK)
 $b=a$; (not OK)



IT KHARAGPUR

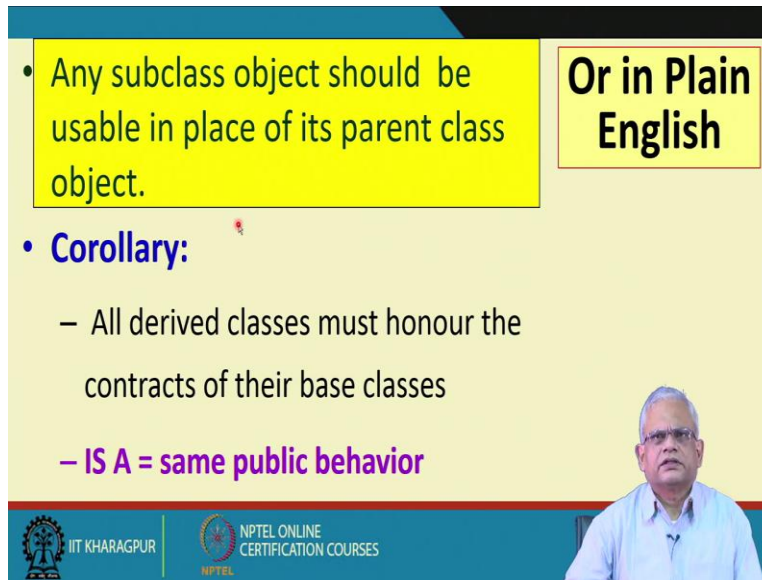


NPTEL ONLINE
CERTIFICATION COURSES



The principle was proposed by Barbara Liskov. Let me repeat the principle in another way. If we have an object of class A and there is an object of class B (a is an object of class A and b is an object of class B) and b is a subtype of A then it is possible to use b for a . So, b is a subtype of a and in that case, we can use B for A that is okay, but not a for b that is not okay. so that is the Liskov substitution principle and defines the compatibility between objects.

(Refer Slide Time: 12:36)



• Any subclass object should be usable in place of its parent class object.

Or in Plain English

• **Corollary:**

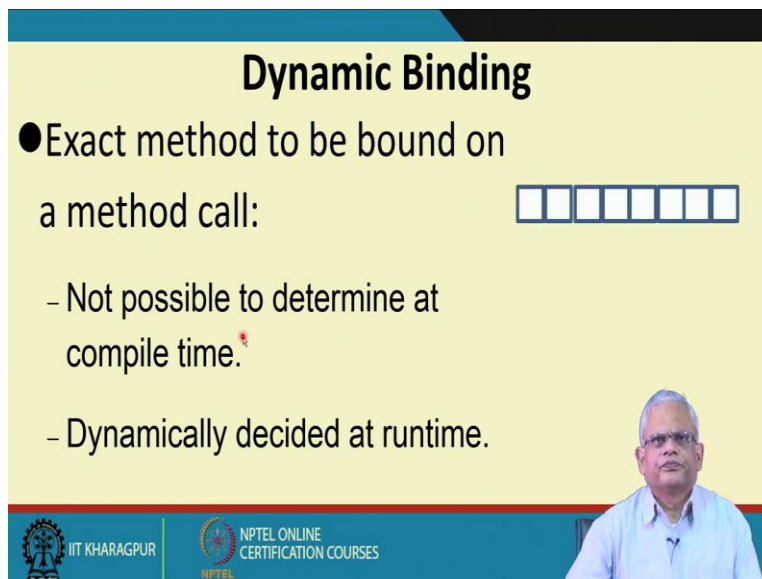
- All derived classes must honour the contracts of their base classes
- **IS A = same public behavior**

IT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

The slide features a yellow background with a blue header and footer. A speaker's video feed is visible in the bottom right corner. The text is presented in a clear, sans-serif font.

In simple English we can say any subclass object should be usable in place of its parent class object and this is intuitively clear. Inheritance is an IS A relation, a derived class object is a type of base class object, but a base class object is not IS A derived class object.

(Refer Slide Time: 13:08)



Dynamic Binding

● Exact method to be bound on a method call:

– Not possible to determine at compile time.

– Dynamically decided at runtime.

IT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

The slide features a yellow background with a blue header and footer. A speaker's video feed is visible in the bottom right corner. The text is presented in a clear, sans-serif font. A diagram of an array is shown next to the text 'a method call:'.

So, here even though we have an array of library members, we don't know if we call the issue book method on an element of the library member array, which method will get called? It will be decided dynamically at the runtime depending on the specific object that's residing in that array.

(Refer Slide Time: 13:40)


• Consider a class hierarchy of different geometric objects:

– Display method is declared in the `shape` class and overridden in each derived class

– A single call to the display method would take care of displaying the appropriate element.

An Example of Dynamic Binding

```
graph TD; Shape[Shape] --> Circle[Circle]; Shape --> Rectangle[Rectangle]; Shape --> Line[Line]; Circle --> Ellipse[Ellipse]; Rectangle --> Square[Square]; Square --> Cube[Cube];
```



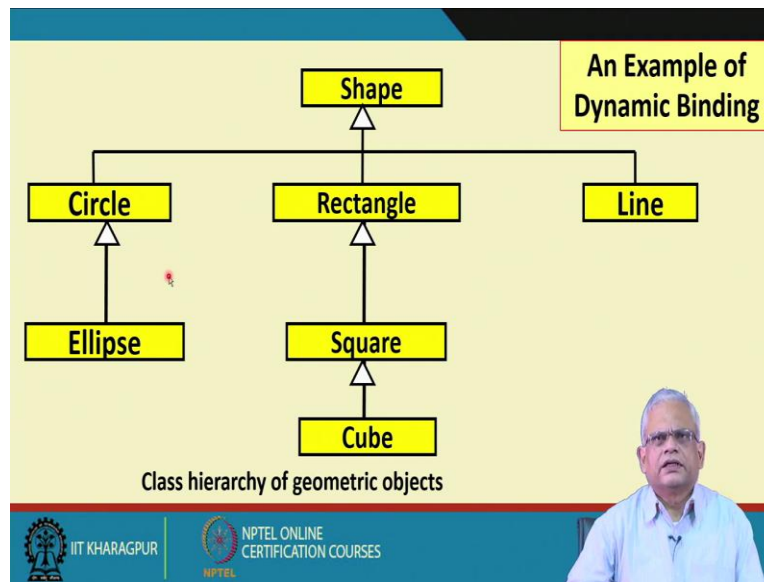
IT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

Now, let's look at an example of dynamic binding considering a shape class hierarchy (in the above slide). Now, let say we have a `display()` method of the shape class and the `display()` method is overridden in the various sub classes of shape. Shape is the base class and circle, rectangle etc. are the derived class.

Now, we have an array of shapes and in that array will create various types of objects and let say in a drawing package or something we want to keep track of various types of geometric objects that are created and we keep track using the array of shapes.

Now, we can call the display method for all the elements of the shape array and we will see that for different elements of the shape array different display methods are called depending on which type of object is residing in the array.

(Refer Slide Time: 15:07)






So, this is the shape class hierarchy (in the above slide). A various types of shape circle, ellipse, rectangle, square, cube, line etc. are there as sub class and we create different types of shapes in a drawing package and we keep track of the drawing elements in the array of shapes. While displaying, in a loop we called all the display method of all the elements of the shape array. And then even though the same method is called on the shape array, but then the exact method will be called depending on that moment which type of object is residing. So, this can only be known at runtime and that's why it is called as dynamic binding.

(Refer Slide Time: 16:06)

Example Code

Traditional code	Object-oriented code using Dynamic Binding
<pre>Shape s[1000]; for(i=0;i<1000;i++){ If (s[i] == Circle) draw_circle(); else if (s[i]== Rectangle) draw_rectangle(); - }</pre>	<pre>Shape [] s=new Shape[1000]; for(i=0;i<s.length;i++) s[i].draw(); - - -</pre>



Let's, look at the code example here (in the above slide). We have this traditional code (left side code) and here we have an array of shape (Shape s[1000]) and then we check the shape type (using 'if' condition) and then depending on the shape type we call the corresponding draw method of that shape (draw_circle(), draw_rectangle()). Not only that this code is cumbersome, but it is less maintainable because each time we add a shape we need to change the code.

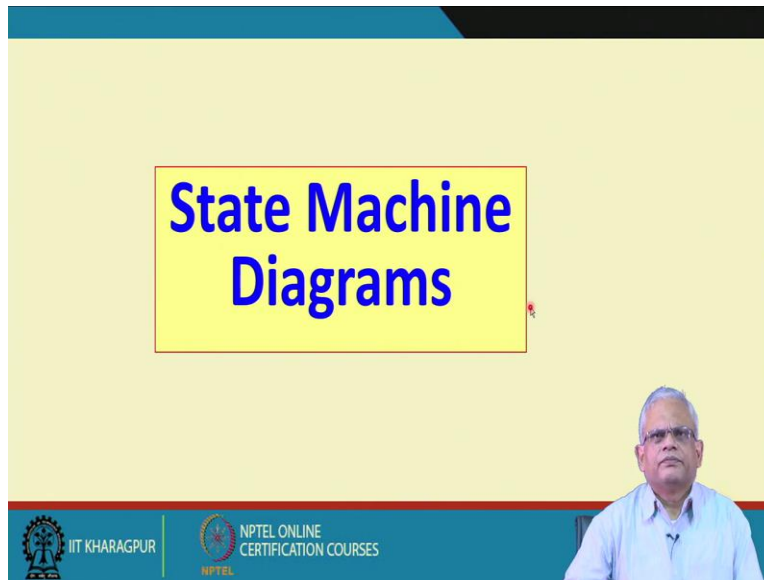
But look at this code where we have dynamic binding (right side code). Here, we have again 1000 shapes and we create the shapes and store in the array and for each element of the array we just call the draw method (s[i].draw()) and the corresponding draw will be called. So, the code is not only concise crisp but also is more maintainable. We can add more shapes but no need to change the code.

(Refer Slide Time: 17:19)

```
class Shape {  
    void draw() { System.out.println ("Shape");  
    }  
}  
class Circle extends Shape {  
    void draw() { System.out.println ("Circle"); }  
}  
class Line extends Shape {  
    void draw() { System.out.println ("Line"); }  
}  
class Rectangle extends Shape {  
    void draw() { System.out.println  
        ("Rectangle"); }  
}  
public static void main(String args[]){  
    Shape[] s = new Shape[3];  
    s[0] = new Circle();  
    s[1] = new Line();  
    s[2] = new Rectangle();  
    for (int i = 0; i < s.length; i++){  
        s[i].draw();  
        // prints Circle, Line, Rectangle  
    }  
}
```

This is a more concrete example of the same class hierarchy (in the above slide). We have the base class shape which has the draw method. We have the circle class extending shape, line extending shape, Rectangle extending shape. All these are the derived classes and each one has a different drawing implementation. So, we can say that the draw is overridden, the base class draw is overridden in the derived classes. In the main method we create three shapes: we create a circle, line, and rectangle and then we just call the draw method for each of these objects residing in the shape array and we will see that the corresponding shape method of the object gets called and this is essentially the principle of dynamic binding.

(Refer Slide Time: 18:18)



So, far we have been looking at the class diagrams and some very basic concepts like polymorphism, packages and so on. Now, let's look at a behavioural diagram which is state machine diagram.

We will see that the objects sometimes have significant number of states and we need to capture the state behaviour of the objects. We need to model that and we will see that if we are able to model the state behaviour of the objects, we can mechanically generate code and many case tools actually automatically generate code based on the state machine diagrams.

(Refer Slide Time: 19:17)

Stateless vs. Stateful Objects

- **State-independent (modeless):**
 - Type of objects that always respond the same way to an event.
- **State-dependent (modal):**
 - Type of objects that react differently to events depending on its state or mode.

Use state machine diagrams for modeling objects with complex state-dependent behavior.

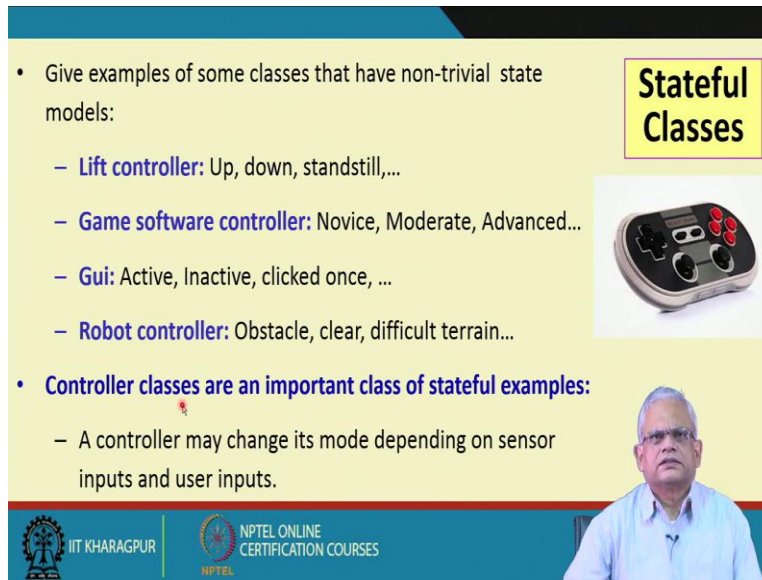
Let's, first start with some very basic concepts. If we look at object-oriented implementation of a object-oriented program, we see that during runtime there are various types of objects that are created. We can roughly classify these objects into two types: one is the stateless objects and the other is stateful objects. The stateless objects are also called as state independent objects or modeless objects. On the other hand, the stateful objects are called a state dependent objects or modal objects. The state independent or modeless object do not have significant state behaviour, there is only a single state in which the object exists. And it keeps on remaining in that state. Whereas the stateful objects there are several states, maybe 3, 5, 10 depending on the complexity of the state behaviour. There will be many states and at any time during the execution the object will be in different states and the behaviour of the object will depend on which state it is.

In a state independent object to all the messages that are invoked on object, it always responds the same way on the other hand in a state dependent or stateful object, once the methods are called, then the response of the object is different depending on the state in which the object is present.

We use a state machine diagram to model the different states of an object and how it transits among the states. Here is a model of a keyboard (in the above slide), if we press the caps lock key then all the letters that we type are transmitted to the computer in capital letter.

But if we have caps unlock, then whatever we type the small case letters are transmitted to the computer. So, we can say that the keyboard exists in two states, one is in caps lock state and the other is the default state or the caps unlocked state. By default, it is in caps unlock and if we press the caps lock key then it goes to the caps lock and whatever key we transmit in the caps lock state these are transmitted to the computer in capital.

(Refer Slide Time: 23:13)



Stateful Classes

- Give examples of some classes that have non-trivial state models:
 - **Lift controller:** Up, down, standstill,...
 - **Game software controller:** Novice, Moderate, Advanced...
 - **Gui:** Active, Inactive, clicked once, ...
 - **Robot controller:** Obstacle, clear, difficult terrain...
- **Controller classes are an important class of stateful examples:**
 - A controller may change its mode depending on sensor inputs and user inputs.

The slide includes a video inset of a man speaking in the bottom right corner. The footer contains the logos for IIT KHARAGPUR and NPTEL ONLINE CERTIFICATION COURSES.

Now, we will get some exposure in object-oriented programming, I am sure that before doing this course you have written several programs in Java or C++. Can you recollect any object which has stateful that is they have different states and behave differently depending on the state you need there? It will be really nice if you can recollect any object which you think is stateful, then we will compare with the answer that I gave you.

There are many objects which are stateful, one is a lift controller. A lift when it is going up it is in the upstate and when it is in going up any requests in the up direction is accepted, from a higher floor who wants to go further up that request will be accepted but if somebody trying to go down will not be accepted it will be ignored by the lift. Only when it is going down the down request will be accepted.

And the lift maybe in stand idle at that time any request either up or down will be accepted. A game software controller can be set to novice mode, moderate mode or advanced mode. Let say chess game and we can set it to be novice, moderate, advanced and so on and the way the responses the game to a move by the player will be different depending on the state of the game.

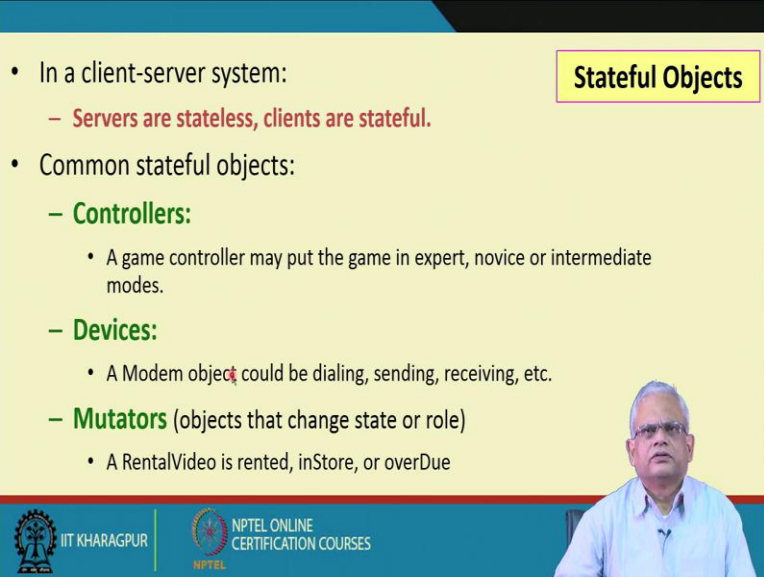
In a graphical user interface, a menu maybe active and we can invoke that menu, if the state is inactive, we cannot invoke that. A 'robo' controller control a robot. The movement command given to the robo controller let say move straight, but then in front there is an obstacle then the robot will not move. But for the same movement if the path is clear it will make move and if it is

a difficult terrain, then the movement may be different. So, we are trying to press the same move command in robo controller and depending on the state the robo behaves differently.

We can see here that the most of the examples that we gave are with respect to controllers and as we proceed with this course, we will see that the controllers are present almost in every program that we write.

The programs that we will write as part of this course, we will see that there will be a plenty of controller classes and all the controller classes happened to be stateful objects. So, the controller classes is one of the important example of stateful classes.

(Refer Slide Time: 27:08)



The slide, titled "Stateful Objects", is presented on a yellow background. It contains a bulleted list of examples of stateful objects. The first bullet point is "In a client-server system:", followed by a sub-bullet "Servers are stateless, clients are stateful." The second bullet point is "Common stateful objects:", followed by three sub-bullets: "Controllers:" (with a note that a game controller may have expert, novice, or intermediate modes), "Devices:" (with a note that a Modem object could be dialing, sending, or receiving), and "Mutators (objects that change state or role)" (with a note that a RentalVideo is rented, inStore, or overDue). A small video inset of a man in a light blue shirt is visible in the bottom right corner of the slide. The slide footer includes the logos for IIT KHARAGPUR and NPTEL ONLINE CERTIFICATION COURSES.

- In a client-server system:
 - Servers are stateless, clients are stateful.
- Common stateful objects:
 - **Controllers:**
 - A game controller may put the game in expert, novice or intermediate modes.
 - **Devices:**
 - A Modem object could be dialing, sending, receiving, etc.
 - **Mutators** (objects that change state or role)
 - A RentalVideo is rented, inStore, or overDue

But there are some other examples, like in a client-server system, servers are stateless but clients are stateful. Devices is another example of stateful and mutators also example of stateful objects.

We are almost at the end of this lecture; we will stop here and we will continue from this point.

Thank you.