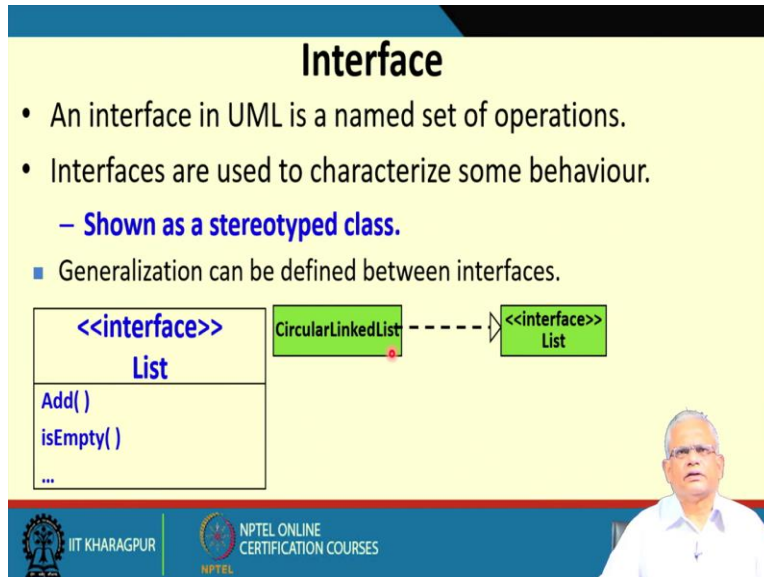


Object Oriented System Development Using UML, JAVA and Patterns
Professor Rajib Mall
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur
Lecture 17
Interfaces, Packages and Abstract Classes

Welcome to this lecture.

Over the last few sessions, we have discussed the classes, their relations and representation in the form of UML diagram. We had also tried to solve some problems based on simple text descriptions and learn how to identify the classes, their relations and represent in the form of UML diagrams. Now, let us look at a special type of class called as interface.

(Refer Slide Time: 00:54)



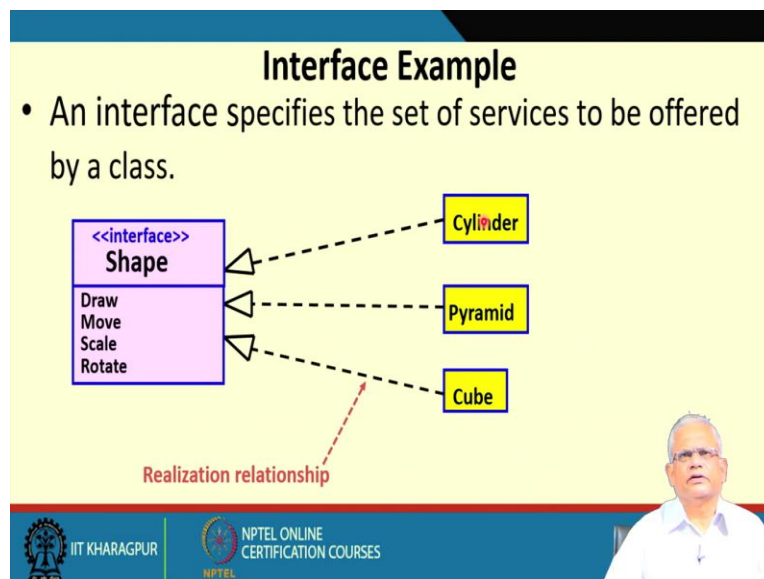
An interface is used in UML. It is very similar to Java interface, it does not have attributes, it has only a set of operations. And the interface is used to characterize some behaviour.

Some aspect of a class we can represent as an interface and we use a stereotype `<<interface>>` and the interface has only methods here just like Java interface and once we have an interface it is possible that we can have other special type of interfaces which are derived from a base interface class.

In the above slide we can see, CircularLinkedList implement List interface. Remember, this is implementation of interface it is not a special type of or it is not a derived interface class. In this example, we say that list is an interface class and it is implemented by the circular linked list.

This dotted arrow here (---->) represent implements, it is not derived it is implements relation between the interface class and circular linked list class. The implements are same as used in Java.

(Refer Slide Time: 03:08)



Now, let's look at some other interface examples (in the above slide). An interface to think of it, it characterizes some behaviour and we can say that one class needs to provide a set of behaviour and then we can represent the behaviour interface and other classes which need to provide this behaviour they implement this interface and this way we can standardize the behaviour.

Let say shape is an interface and the behaviour that is defined for the shape is that draw a shape, move a shape, scale, and rotate. Now, there various types of concrete classes like cylinder, pyramid, cube and all these implements or realized this interface 'Shape' that means each of them will have the method draw, moved, scale, rotate. With each of these basic geometric objects you can apply all the four methods which define a standard behaviour for all classes. So that's the most important use of an interface. It helps a set of classes to provide a standard set of methods or a standard behaviour.

(Refer Slide Time: 04:52)

Realizing an Interface

- A class *realizes* an interface if it provides implementations of all the operations.
 - Similar to the *implements* keyword in Java.
- UML provides two equivalent ways of denoting this relationship:

Both represent: **“CircularLinkedList implements all the operations defined by the List interface”**.

IT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

In our above example we saw that the circular linked list class realizes or implements the behaviour of the list interface. In a Java code we will implement this kind of relation by using the ‘implements’ keyword. For example, Class circularLinkedList implements list.

Since the interfaces are used extensively in Java as we proceed in this course later will see that the interfaces are one of the extensively used UML elements while solving any problem and we will see that in the design patterns the interfaces are used extensively. We will see that there is hardly any design pattern which does not make use of the interfaces. We will use this <<interface>> stereotype for represent a class as interface and since it is so extensively used, we have an alternate symbol for that also, we can also represent an interface as a filled circle (as shown in the above slide).

Now, both these representations (dotted closed arrow and a line here between a circle), we read it as the circular linked list implements all the operations defined by the list interface.

(Refer Slide Time: 07:04)

Interface Dependency

- A class can be dependent on an interface.
 - This means that it **makes use** of the operations defined in that interface.
 - E.g., the Restaurant class makes use of the **List** interface:

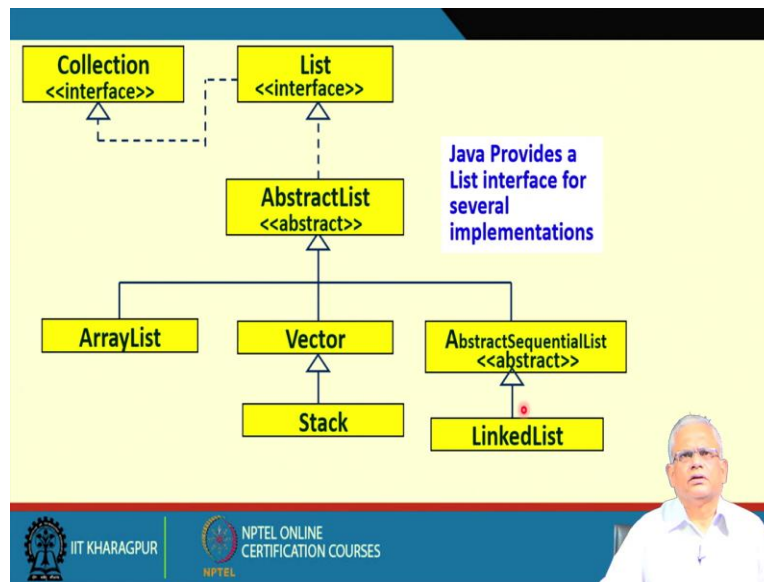
```
classDiagram
    class Restaurant
    class List
    Restaurant ..> List : <<use>>
```

The slide includes logos for IIT KHARAGPUR and NPTEL ONLINE CERTIFICATION COURSES at the bottom. A small inset image of a man is visible in the bottom right corner of the slide.

Now, once a class implements an interface there can be several classes which may be implementing the same interface and there will be few classes or many classes which will be using this interface, so they will be invoking the services of the other classes which are implementing this interface.

So the name of the method supported by the classes is same and the classes use those standardized set of names to invoke services of those classes and that is represented by a dotted arrow by an open arrowhead and this will use to mean that the restaurant class makes use of the operations defined on the list interface. The restaurant class makes use of the list interface (as shown in the above slide).

(Refer Slide Time: 08:22)





If we look at the Java documentation (in the above slide) we will find that the list interface is derived from the collection interface and the abstract list implements the list interface and there is an inheritance relation here: the **AbstractSequentialList**, **ArrayList** and **VectorList** are derived from the **AbstractList**.

The **stack** is derived from the **vector** and the **LinkedList** implements the abstract sequential list (we will have this as a dotted arrow instead of solid arrow).

(Refer Slide Time: 09:31)

UML Packages

- **A package is a general purpose grouping mechanism.**
 - Can be used to group any UML elements (e.g. use cases, actors, classes, components and other packages.)
- **Commonly used for specifying the logical grouping of classes.**
- A package does not necessarily translate into a physical sub-system.



IT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

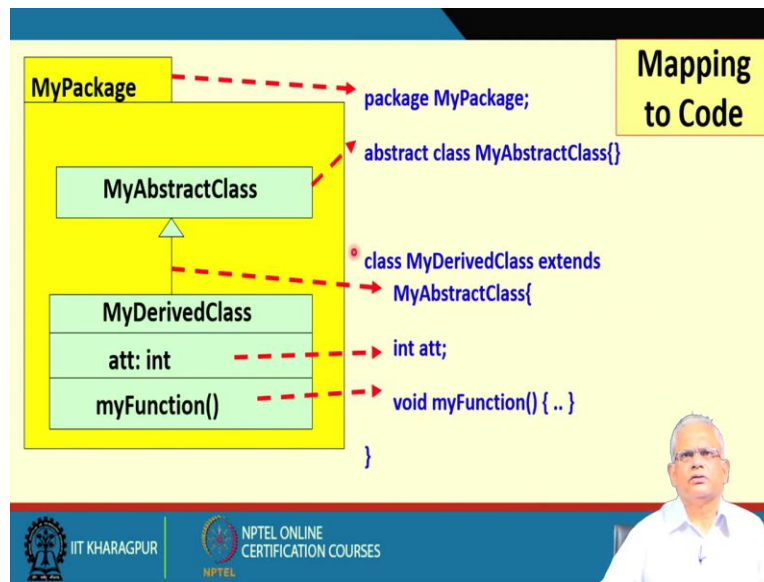
Now, let's look at one more concept of UML which is packaging. Packaging is also very frequently used in design problems, in fact the larger is the problem the more extensive is the use of the UML packages. A package is a grouping mechanism and we can use the packages with any of the UML elements for example, we can package use cases, we can package actors, we can package classes, we can package components and even we can package other packages, so a package can contain other packages.

Here the package the main use is that we group other elements and but then we will frequently use it to group classes and we know this corresponds to a Java package. So, we can have the UML package to indicate the classes which are present in a specific Java package. One thing to remember, package not necessarily translates into a physical sub-system, it can be at a more conceptual level.

The representation of a package is like a file folder and we write the folder the name of the package and inside it we can write the specific elements that are present. For example, there may be several classes inside the package and even few more packages may be inside this.

For example, Account is a package name (in the above slide). The account package contains audit package, it can have other classes like let us say payments, receivables and so on. So, a package can contain other packages and also classes.

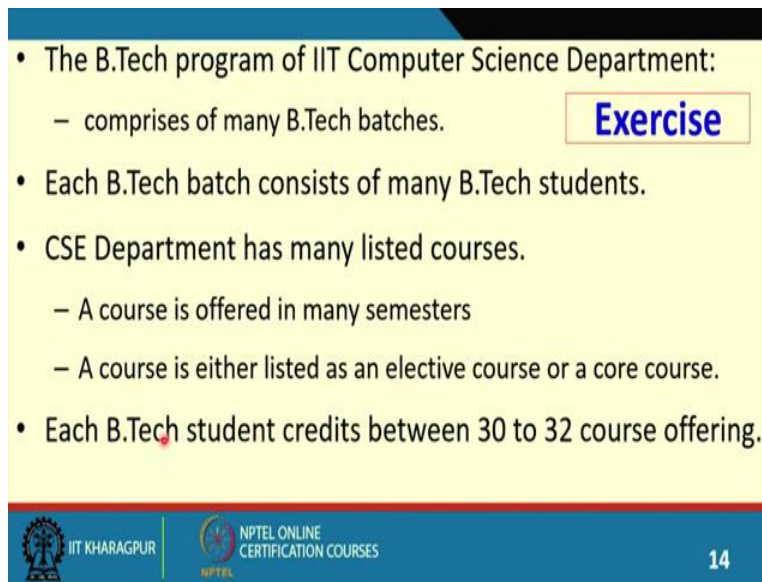
(Refer Slide Time: 12:30)



Now, let see the translation of a UML package into Java code (in the above slide). The package name here appears in the Java keyword package and then the MyPackage (package MyPackage;). MyPackage is the name of the package and then whatever is inside the package we write the code for that for example, abstract class 'MyAbstractClass' and then 'MyDerivedClass extends MyAbstractClass' and then some attributes and the methods 'myFunction()' mentioned here.

Most of the UML diagram that we have seen so far, there is a straightforward translation to Java and the case tools take advantage of this and generate code. Please try the Argo UML and you can draw these different elements on the case tool and see the kind of code that is generated from the diagram.

(Refer Slide Time: 13:45)



The slide contains a list of bullet points and a highlighted box. The bullet points are:

- The B.Tech program of IIT Computer Science Department:
 - comprises of many B.Tech batches.
- Each B.Tech batch consists of many B.Tech students.
- CSE Department has many listed courses.
 - A course is offered in many semesters
 - A course is either listed as an elective course or a core course.
- Each B.Tech student credits between 30 to 32 course offering.

The word "Exercise" is enclosed in a blue-bordered box with a white background.

At the bottom of the slide, there are logos for IIT KHARAGPUR and NPTEL ONLINE CERTIFICATION COURSES, along with the number 14.

Now, let's do one exercise (in the above slide). This actually will not solve it here, we will give you to work out as an assignment and please spend some time in doing this assignment.

The B.Tech program of IIT computer science department consists of many B.Tech batches. The department has a program: the association between the department and the program. The program has many batches, each batch consists of many students, the department has many courses and a course is offered in multiple semesters. So, here the department and course there is association but the semester in which it is offered can be in an association class. A course is either an elective course or a core course. A sentence like a course is either elective course or a core course says indicates that course is a general class and elective courses and core course are the derived classes.

Each B.Tech student credits between 30 to 32 course offerings-- this is an aggregation of a student and the number of courses.

Please try this exercise.

(Refer Slide Time: 15:41)

Constraints on Objects

- A constraint restricts the values that objects can take.
- **Example:** No employee's salary can not exceed the salary of his boss.

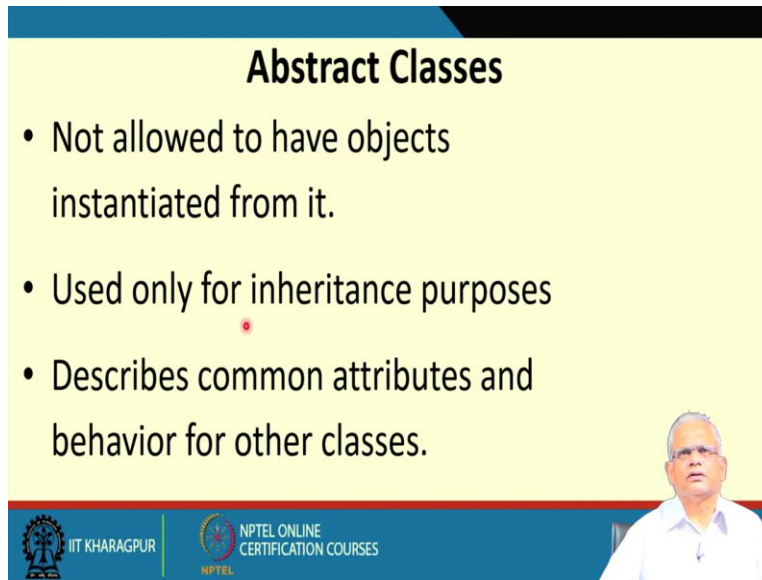
`{salary <=boss.salary}`

```
classDiagram
    class Employee {
        Salary
    }
    class Employer {
        Salary
    }
    Employee --> Employer : reports to
```

The slide features a yellow background with a blue header and footer. A small portrait of a man is visible in the bottom right corner of the slide area.

Now, let's move to the other UML elements. We will look at constraints that can be specified on specific objects. A constraint is used in UML to restrict the values that an object takes. For example, Employee is a class and then we want to mention that employee salary cannot be more than the boss salary. This is difficult to express in the diagram that we so far seen that the salary of one object that is employee cannot exceed the salary of the boss and in such cases we use a notation within the curly bracket and write {salary of the employee is less than the boss salary}. But it is informal. Instead of this we can write {salary<=bosssalary} which is quite formal (as shown in the above slide).


(Refer Slide Time: 17:11)



Abstract Classes

- Not allowed to have objects instantiated from it.
- Used only for inheritance purposes
- Describes common attributes and behavior for other classes.

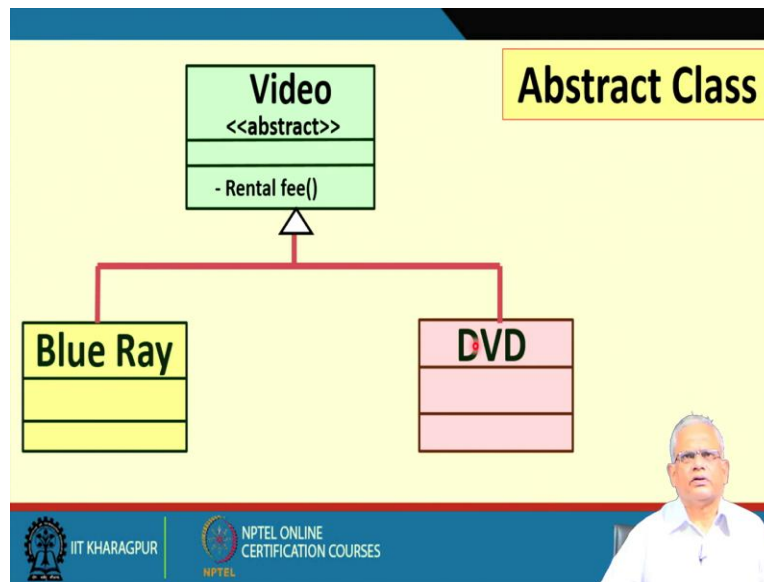
IT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES



Now, let's look at the Abstract Classes. An abstract class is not allowed to have objects instantiated from it; only concrete classes can have object instances. We can instantiate objects of a concrete class, but abstract classes we cannot create any objects using this class, then the question naturally arises is then what is the use of then abstract class? if we cannot create objects of an abstract class, why do we need to define?

The answer here is that we use it for inheritance purposes that is we can define some behaviour in the form of an abstract classes and that gets reused by various derived classes. So, some common attributes and behaviour of other classes are made into an abstract class. Even though we cannot instantiate an abstract class, but these are used by the derived classes so that we do not have to define the methods and attributes again and again in many classes. Just use one abstract class to define it and then later inherit this in the other classes.

(Refer Slide Time: 18:39)



Let see one example here (in the above slide). The abstract class is indicated by a stereotype <<abstract>>. For abstract class, we write the specific stereotype within this angled bracket that is the UML syntax. Here, video class is an abstract class and Blue Ray and DVD are two derived classes from the video class.

(Refer Slide Time: 19:14)

Abstract Class

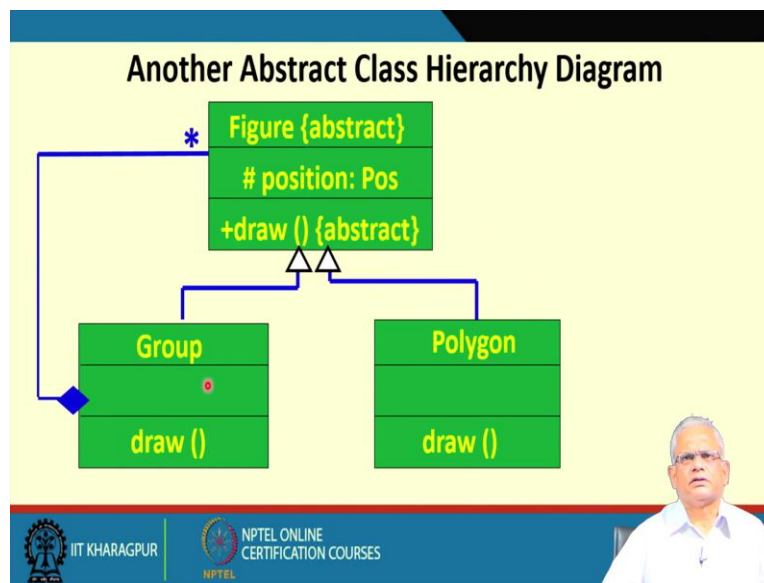
- Why use abstract class?
 - Reduces complexity of design
 - Enhances understandability
 - Increases productivity
- It has been observed that:
 - Productivity is inversely proportional to complexity.

The slide features logos for IIT KHARAGPUR and NPTEL ONLINE CERTIFICATION COURSES, and a small portrait of a man in the bottom right corner.

We have so far looked at the abstract class, we already said abstract class just like any other class but stereotype abstract is present and we have also answered so far that why we need to use an abstract class. If you look back and you will see that it reduces complexity of the design and

helps reuse of code, we define it once and reuse it. It also enhances understandability and increases productivity because we need to write less code. And if our complexity reduces then we can write larger code in a certain time and able to easily understand the programs and able to easily debug the program.

(Refer Slide Time: 20:17)



This is another example of an abstract class (in the above slide). The Figure here is an abstract class. In the figure, abstract notation written wrongly. It should be <<abstract>> instead of {abstract}. Here, we derive the group and the polygon classes and the group here consists of many figures. The polygon is a special type of figure and therefore, each group can contain many polygons and also it can contain many groups. This is a very popular class diagram in many implementations. Here is a recursive inclusion of a figure is shown. A group has many polygons and other groups and inside the group has other polygons and groups and so on.


And this tree hierarchy is very useful later as we proceed, we will see that there are specific patterns that proposed such a tree hierarchy class relation.

The figure is an abstract class and then group and polygons are concrete classes that are derived from the abstract class and the group contains many figures and a figure can be a polygon or a group. And therefore, a group can contain many polygons and many other groups recursively.

(Refer Slide Time: 22:47)

```
Abstract public class Figure {
    abstract public void draw();
    protected Pos position;
}
public class Group extends Figure {
    private Vector <Figure> figures = new Vector <Figure> ();
    public void draw () {
    }
}
public class Polygon extends Figure {
    public void draw () { // draw polygon code
    }
}
```

Java Implementation



IT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NPTEL

Now, what will be the code for this?

If we look at the Java implementation (in the above slide), Figure is the abstract class, write: 'Abstract public class figure' and we have abstract method here 'draw' and an attribute 'position'. Group extends figure say group is the derived class and the group contains many figures for that we use a vector or an ArrayList. Group class also provides an implementation of the 'draw' because the base class method 'draw' is abstract method and we need to provide the the implementation of the method draw in the concrete class. And then the class 'polygon' also extends figure and it also provides an implementation of draw.

So, this is the Java code for the class diagram that we just saw. And as you can see it is a very simple class diagram, we had seen such diagrams earlier and here just the abstract class representation is shown.

(Refer Slide Time: 24:19)

What is Polymorphism?

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

Logo of IIT KHARAGPUR and NPTEL ONLINE CERTIFICATION COURSES. Page number 21.


Now, let's look at polymorphism. Polymorphism literally means poly and morphism. Poly means many and morphism is forms. Something that appears in many forms that is the meaning here polymorphism.

For example, if an object is polymorphic then the same message can result in different actions when it is sent to polymorphic objects. So, it represents different behaviour by the polymorphic elements to the same message. We can classify polymorphism into two types: one is called a static polymorphism and the other is dynamic polymorphism.

(Refer Slide Time: 25:37)

```
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



Logo of IIT KHARAGPUR and NPTEL ONLINE CERTIFICATION COURSES are visible at the bottom of the slide.

Static polymorphism is also called a static binding and here the concept is same as method overloading. For example (in the above slide), the create method either appears without any parameter (first declaration of create method) or takes three parameters (second declaration of create method) or it takes four parameters (third declaration of create method).

Depending on the parameter the create method will be called. The parameters the specific method body will be bound and this is bound at the compile time. And that is why it is called a static binding because based on the parameters it is determined which method we are calling and it will be bound to that method.

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

Thank you.