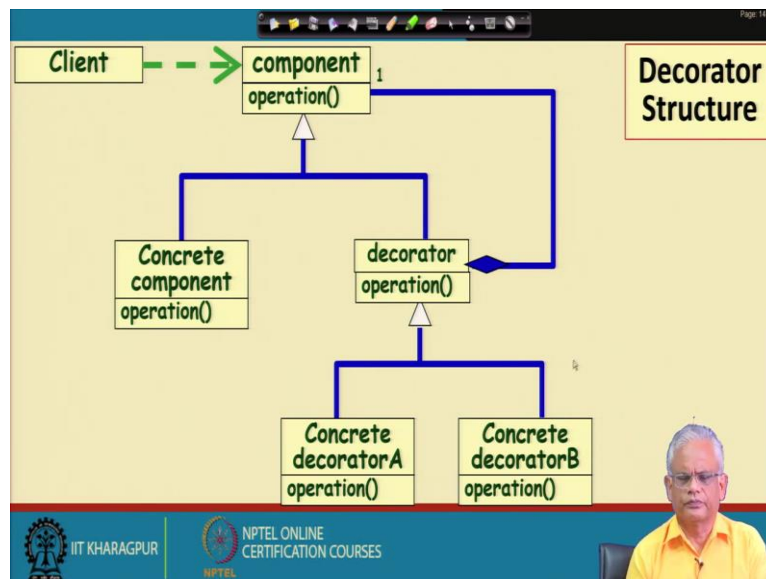
Object Oriented System Development using UML, Java and Patterns
Professor Rajib Mall
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur
Lecture 59
Decorator Pattern III

Welcome to this lecture.

In the last lecture, we were discussing about the decorator pattern. We had said that the decorator pattern is a very important design pattern, if we use this in application development it gives us a powerful mechanism to structure our application. We can add responsibilities to individual objects as compared to adding responsibility statically through inheritance. Responsibilities can be added at a time to an object removed more responsibilities added and shown.

And the client classes, the client objects they don't even distinguish a decorated object with the concrete object. The class diagram was simple will just recapitulate this class diagram. I will pose a quiz problem, please try to do the quiz problem and will display the solution so that we can compare your answer. Now let's look at the basic structure of the decorator pattern.

(Refer Slide Time: 02:08)



In the decorator pattern (in the above slide) the client interacts with the outermost decorator both the concrete component and the decorator they are derived from the component, the component can be an interface or an abstract class and therefore the concrete component in the decorator they support the same operations but a decorator can add additional operations

there can be various types of decorators. And for those operations which are defined in the concrete component, the decorator for those operations only forwards to the concrete component.

On the other hand, for the operations which are defined in the concrete decorator it executes that and does not forward any calls to the concrete component and just look at here the cardinality, the cardinality is one because at a time to a concrete component we can add one decorator at a time. We can add many decorators but one at a time that is the implication here.

(Refer Slide Time: 03:26)

Quiz

- An ice cream can be made with the following types of toppings in any combination and order:
  - Nutty
  - Honey
  - Fruity
  - Chocolate
  - Vanilla

- Apply decorator pattern
- Draw class diagram
- Write Java code

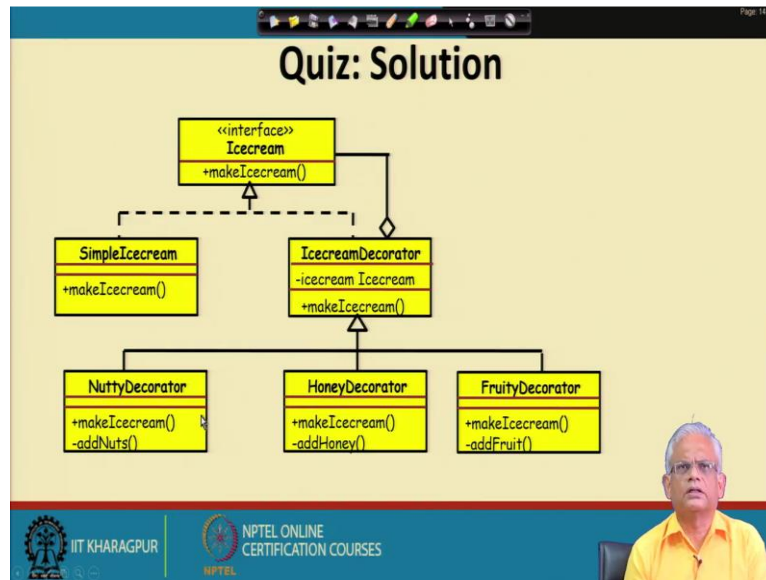
1. Draw class diagram
2. Write Java Code

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

Now, let’s try to solve a quiz problem. The quiz is that we want to write a software using which we can make ice creams, this is a game package and in this game package we can create ice creams and we can choose the following types of toppings in any combination in any order, so we have a basic ice cream.

And then we can add toppings to that we can first have nutty topping, honey topping, fruity topping and so on. We need to apply the decorator pattern to have this game package written, we want to draw the class diagram and we want to write the Java code. Please draw the class diagram first. The decorator has a simple structure as you have seen that the client interacts with outermost decorator and the decorator each time contains one more decorator or the concrete object.

(Refer Slide Time: 05:01)



So, this is the solution, the interface ice cream is implemented by the simple ice cream and ice cream decorator. The undecorated ice cream or the simple ice cream, we have the make ice cream method here which will draw the ice cream and then we have various types of decorator here, these are the concrete decorators and we have the nutty decorator which will add nuts, the honey decorator, the fruity decorator and so on.

And each decorator can contain another decorator: single decorator or the concrete object which is the simple ice cream. So, this is the class structure for this example (in the above slide) by applying the decorator pattern, but what about the Java code?

Please try to write the Java code for this application so that we get a feel that how do you really could such a class structure. We need to write the code for ice cream which is the interface, the simple ice cream.

(Refer Slide Time: 06:33)

Quiz: Java Code

```
public interface Icecream {
  public String makeIcecream();
}

public class SimpleIcecream implements Icecream
{
public String makeIcecream() {
    return "Base Icecream";
  }

abstract class IcecreamDecorator implements Icecream {
  protected Icecream specialIcecream;
  public IcecreamDecorator(Icecream specialIcecream) {
    this.specialIcecream = specialIcecream;}
  public String makeIcecream() {
    return specialIcecream.makeIcecream();}
}
```

IIT KHARAGPUR

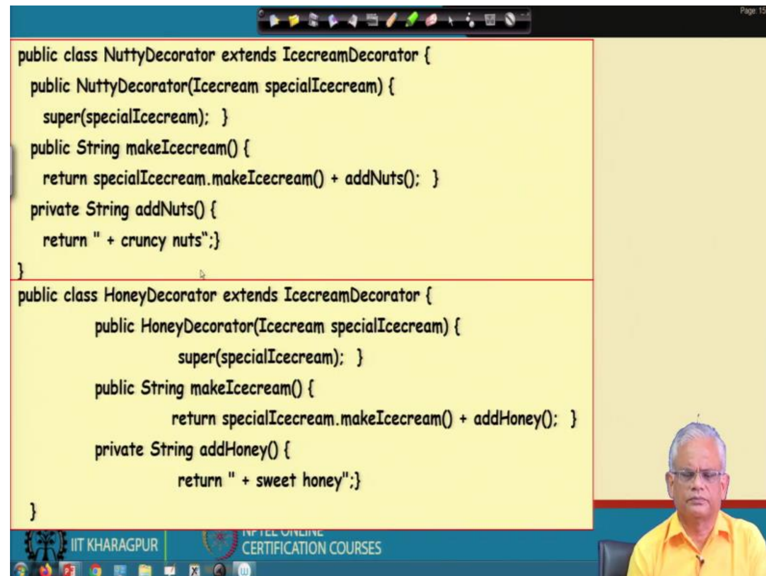
NPTEL ONLINE CERTIFICATION COURSES

Please see the above slide code. Interface ice cream only has a make ice cream method and this is written here string it will just print instead of displaying a graphic it will just write in terms of text. Simple icecream implements icecream so this is the icecream interface which is just having the method make icecream and then the simple icecream implements the icecream interface and here it just writes basic icecream we can have a graphic drawing of the simple icecream.

And then the icecream decorator implements the icecream interface and then it has a reference to the special icecream and then it takes the special icecream as the argument and stores the reference to that because it will pass on the calls to that and then it has a method. There is abstract class icecream decorator basically and these are the concrete classes nutty decorator, honey decorator and so on.

Now here it defines an abstract method make ice cream and it returns special icecream that make icecream, now this will be overridden by the concrete classes.

(Refer Slide Time: 08:49)



Now the nutty decorator extend the icecream decorator which is an abstract class and nutty decorator is a concrete class it takes special icecream as an argument. The constructor takes special icecream as an argument and then just call super special icecream so that the reference gets stored and, in the make, icecream it overrides the make icecream of the icecream decorator and then it just adds nuts and in the add nuts it just returns crunchy nuts.

And the honey decorator is a concrete decorator similar to the nutty decorator, again it extends the abstract class icecream decorator and then there is a constructor where it stores the reference to the argument that is passed on to it and then overrides the make icecream and then it has only the add honey here and add honey is just return sweet honey.

So, this is a simple implementation of the example that we have discussed here the decorators only write text but then the game package you might want to write in graphics those who are incline programmatically have done a lot of programming using Java swing and so on please try to write a graphics based software where using this basic skeletal code you can have the basic icecream and various types of decorators on it as the player or the user chooses different options.

(Refer Slide Time: 11:00)

Decorator: Pros

- **More flexible than static inheritance:**
  - Responsibilities can be added and removed at run-time
  - Decorators also make it easy to add a property twice. For example, to give a TextView a double border, simply attach two BorderDecorators. Inheriting from a Border class twice is error.
- **Avoids inheriting from feature-laden classes.**
  - An application needn't pay for features it doesn't use.
  - It's also easy to define new kinds of Decorators independently from the classes of objects they extend.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

Now we have some experience on applying decorator pattern, we have seen that it is more flexible than static inheritance, responsibilities can be added and removed at runtime we can add a property twice, for example we can add double border to a text view but then in inheritance you cannot do that actually because you cannot inherit from class twice that will be an error. Adding two borders will be difficult using static inheritance it will give you an error.

And also, the decorator you only add that is needed and again you can remove those responsibilities which are no more needed but using inheritance makes the classes feature laden lot of features methods get define and many of that you never used also and also you can use new kinds of decorators independent from the class of objects they extend.

(Refer Slide Time: 12:28)

But are there any disadvantages with decorators? Yes, we had said that regular use can make the code very difficult to understand and debug. For example, you might have a lot of little object 60, 70 or 100 decorators added to a basic class and these are added in runtime trying to analyse the runtime addition of the objects as decorators make it very frustrating to debug the code. As long as you add only few decorators these are well within the grasping power of a programmer but if you add too many decorators, dozen sub-decorators to a basic object these appear like little objects getting connected during runtime makes it very difficult to follow the behaviour. Ofcourse, the programmer who has written the code may be easy for him because he has added those lot of decorators. And he knows what he has been doing but somebody else trying to find out what's going on the code maybe very frustrating and hard to learn and debug by anybody other than the programmer who had written the code. So, a word of caution here if you are using decorators don't add too many decorators run time.

(Refer Slide Time: 14:13)

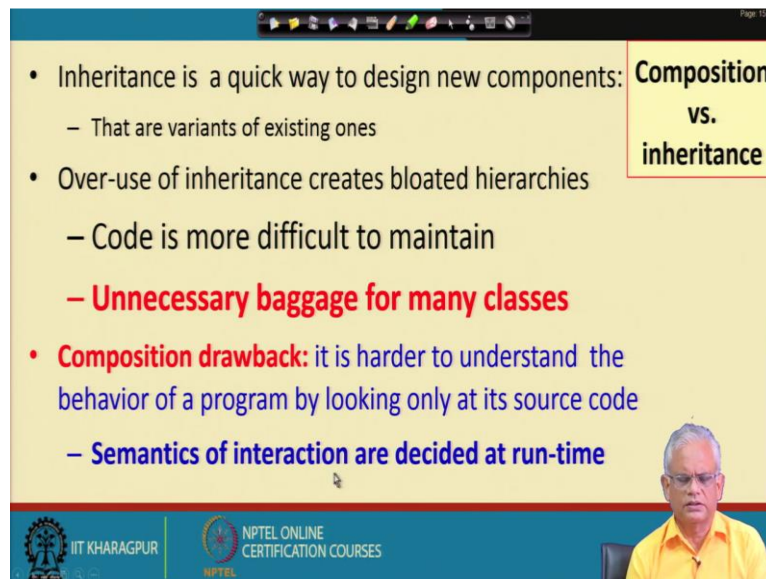
Composition vs. Inheritance

- Both are ways to re-use functionality
- Inheritance:
  - Re-use functionality of parent class
  - Statically decided
  - Weakens encapsulation
- Composition:
  - Re-use functionality of objects at run-time
  - Invoked through the interface
  - Dynamic: multiple types with same interface
  - Black-box re-use

IIT KHARAGPUR NPTEL ONLINE CERTIFICATION COURSES

We have been discussing the advantage of composition over inheritance in many applications both are mechanisms to reuse functionality. In inheritance we use the functionality of the parent class and these are statically bound and weakens encapsulation. Whereas composition, we again reuse functionality of objects at runtime this is basically the reuse occurs through delegation invoked through an interface these are dynamic responsibilities get added dynamically. And this is an example of a black box reuse (in the above slide) just need to call the method of the internal object whereas if we use inheritance maybe you can modify the behaviour of a method.

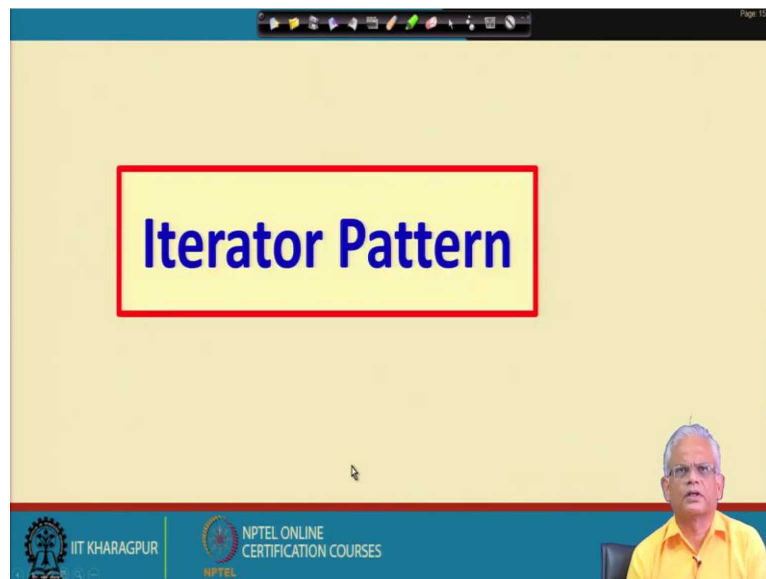
(Refer Slide Time: 15:22)



So that's a white box reuse using inheritance. Inheritance is a quick way to design new components just use extends and then it is done but many programmers make the mistake up using too many inheritance and result in bloated hierarchy code becomes difficult to maintain and unnecessary baggage many of the it has large number of methods and as far as specific application the programmer does not even need half of the methods just uses only few of them, the rest are only unnecessary baggage just cluttering.

On the other hand, composition, the drawback is that if we have too many decorators it becomes difficult to understand the behaviour because the semantics of interaction are decided at runtime.

(Refer Slide Time: 16:27)

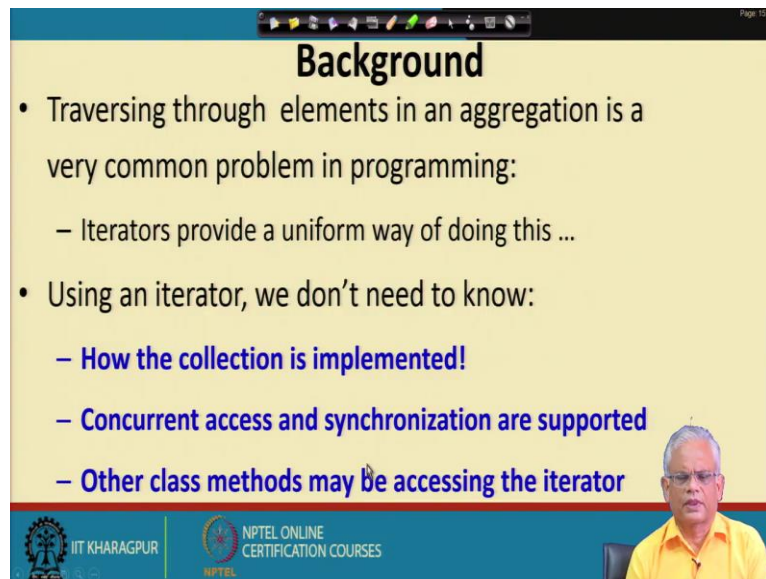


We have discussed already an important pattern: the decorator pattern, we will look at another pattern which is also a very important pattern, the iterator pattern. The iterator pattern any Java programmer would have already been using the Java iterators but ofcourse, many Java programmers they don't know why they have been doing it that way just because it is there in the book they have been doing, they have not asked the question that why iterators, how does it work? and so on.

If we know the iterator design pattern will know why you have been using the Java iterators for the collection classes? What is the advantage for another class which is not part of the collection class or you have extended from the collection class how do you define an iterator for that? And so on.

If you know the iterator pattern not only that you will understand how the Java iterators work why you have been doing something in some particular way what is the advantage? What is the exact mechanism that was used? But also you can use the iterators for a new application. So, it's an important pattern.

(Refer Slide Time: 18:02)

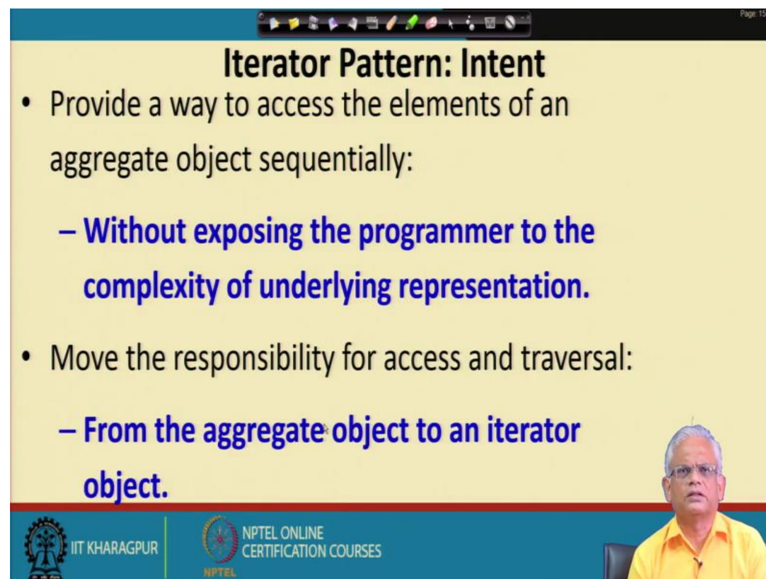


Traversing of the elements in an aggregation is a very common problem in programming. We do traverse through an aggregation for various purposes, we might like to search something, sort, display all the contents and so on. Every Java programmer knows that the iterators provide a uniform way of doing this.

When using an iterator, we don't need to know how is the collection implemented? How is a tree implemented? Is the tree is implemented an array list? Is it implemented on linked list? And so, we do not bother we just use the iterator.

Also, for the same aggregate different client classes may be traversing it concurrently and synchronisation is supported by the iterator class and even the other methods may be accessing the iterator still will be independent even though different iterations on the same aggregate be occurring but then these don't impact each other.

(Refer Slide Time: 19:49)

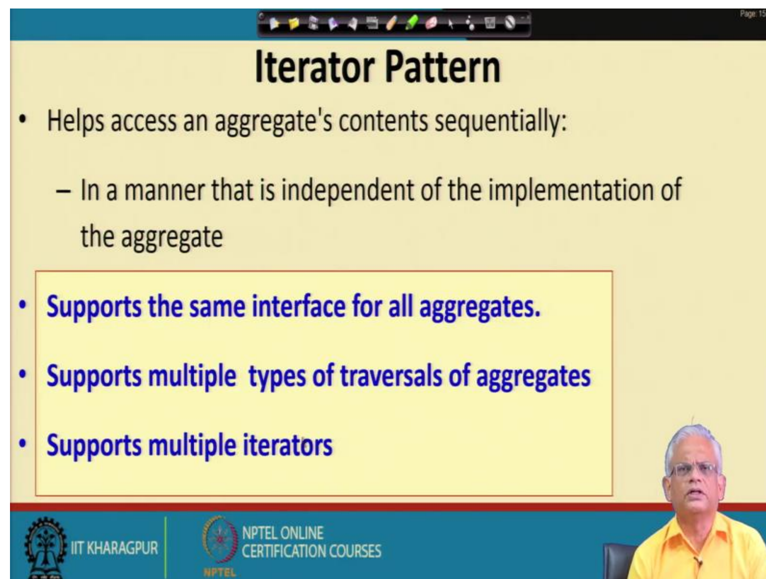


The intent of the iterator pattern is to provide a way to access the elements of an aggregate object sequentially and at the same time, the programmer need not understand the complexity of the underlying representation. The programmer just calls the methods or the iterator next and so on. The programmer did not look at how is the aggregate represented internally and then find out how to identify the next element? How to identify end of the tree? or whatever and the leaf nodes and so on.

And therefore, it tremendously reduces the programming effort and makes the code writing easy and understanding the code easy. Instead of having the aggregate object define the way it will be traversed it is moved to the responsibility of traversing to another class which is the iterator class.

Otherwise, if different types of iterator are supported in the same aggregate class, for example we want to traverse pre-order, post-order, in-order, level order and so on the aggregate object may become too complex. So here the responsibility of traversal is shifted to the iterator rather than to the aggregate object.

(Refer Slide Time: 21:42)



Here, we access the aggregates content sequentially that is independent of the implementation of the aggregate, whether it is implemented in array list, linked list, hash table we don't bother and also whatever be the aggregate whether it is array list, linked list, hash table whatever all the iterators have the same interface and therefore it becomes easy for the programmer.

And also, we can use different types of iterators to support different types of traversals on the same aggregate and also, we can have multiple iterators and each of these multiple iterators they don't interfere with the functioning of other. Maybe one iterator is traversing from the front to the end, first element to the last element and another iterator may be travelling from the last element to the first element. And then we have created these two iterators they won't even interfere with each other.

(Refer Slide Time: 23:17)

Iterator: Essential Idea

- The elements of a collection:
  - Accessed in some sequential order that may be independent of the specific collection.
- For example:
  - Level order: Labelling each object of a tree from left to right
  - Inorder, post order, preorder, etc

IIT KHARAGPUR NPTEL ONLINE CERTIFICATION COURSES

The main idea access elements of an aggregation in some sequential order that is independent of the specific collection. We might define what is level order and given any tree we can just have a iterator which does the level order traversal, in-order, post-order, pre-order traversal and so on.

(Refer Slide Time: 23:55)

The Iterator Pattern: Context

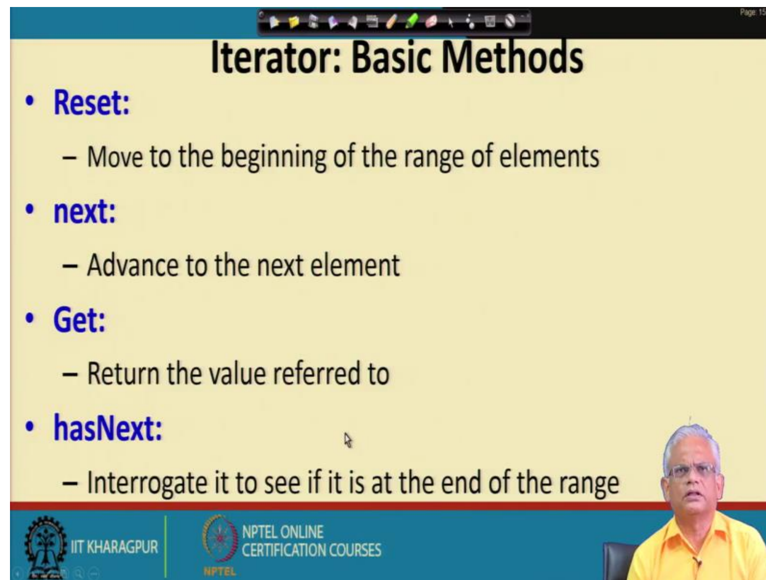
- It might be necessary to have more than one type of traversal on the same aggregate object.
  - Also, not all types of traversals can be anticipated a priori.
- One should not bloat the interface of the aggregate object with all possible traversals.

IIT KHARAGPUR NPTEL ONLINE CERTIFICATION COURSES

Now the context of this pattern is that often we need different types of traversal on the same aggregate. We might need multiple iterators by the same client or maybe by different clients and also all types of traversal cannot be anticipated priori. We can define the iterators for any specific type of traversal that we need and we don't want to change the aggregate class

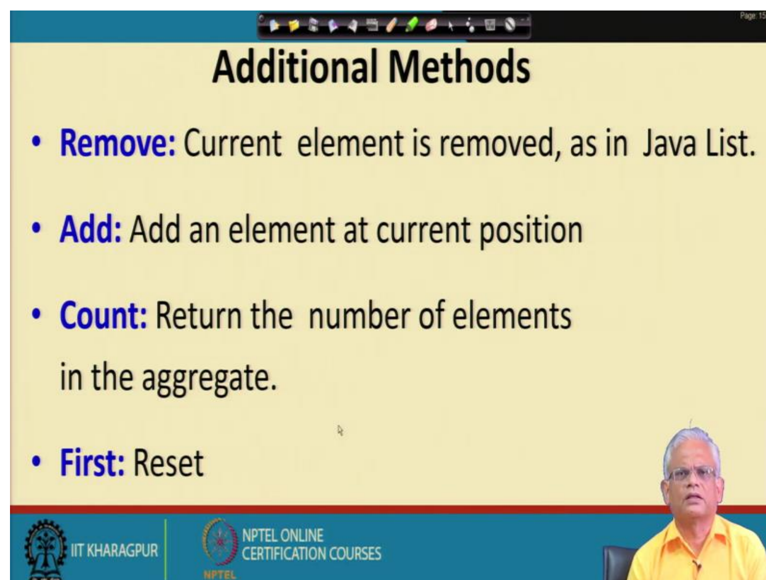
neither we want the aggregate to have a very bloated interface supporting all possible types of traversals that somebody may need.

(Refer Slide Time: 24:51)



In the iterator pattern the basic methods that are supported by the iterator are reset, next, get and hasNext. The reset intuitively clear, move to the beginning of the range of elements, next advance to the next element, get return the current value that is referred to, hasNext interrogate to check if there are more elements in the range or we have reached the end of the range.

(Refer Slide Time: 25:37)



We can have some additional methods for the iterator popular methods are removed, the current element may be removed as in the Java list but it may not be necessary for all types of

aggregates. Add an element at the current position, count, return the number of elements in the aggregate and first which is basically reset.

We will discuss about the class structure of the iterator, we will see how the iterator object is created and how the reference of the aggregate remembered by the iterator, how does the iterator perform the traversal and how does it implement all these different methods. And that will help us to understand the way the java iterators for the collection classes work.

And also, will try to write iterator for a given class which we have derived from the collection classes for which iterators have not been defined.

We are almost at the end of this lecture and those activities that is the class structure of the iterator, the code for the iterator and writing the iterator for a given application that will take up in the next class.

Thank you.