

Object Oriented System Development using UML, Java and Patterns
Professor Rajib Mall
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur
Lecture 60
Iterator Pattern

(Refer Slide Time: 00:28)

Iterator Pattern: Intent

- Provide a way to access the elements of an aggregate object sequentially:
 - **Without exposing the programmer to the complexity of underlying representation.**
- Move the responsibility for access and traversal:
 - **From the aggregate object to an iterator object.**

The slide features a tree diagram with root node 'A'. Node 'A' has three children: 'C', 'K', and 'H'. Node 'C' has two children: 'J' and 'B'. Node 'B' has one child: 'E'. Node 'H' has three children: 'F', 'D', and 'G'. Node 'F' has one child: 'I'. The slide also includes a small video inset of Professor Rajib Mall in the bottom right corner. At the bottom, there are logos for IIT Kharagpur and NPTEL Online Certification Courses.

Welcome to this lecture.

In the last lecture, we had started discussing about the Iterator Pattern. The iterator pattern is a very useful pattern and every Java programmer has used iterators possibly without knowing the iterator pattern. Knowing the iterator pattern gives an insight into how the Java iterator works and also you can define your own iterator for any aggregate class that you define.

Let's just recapitulate very briefly what we are discussing in the last lecture. We said that the iterator pattern helps to provide sequential access to the elements of an aggregate object, and that the programmer does not need to know how the aggregate is stored actually internally. Just take an example we might have a tree and this is a n-ary tree (in the above slide).

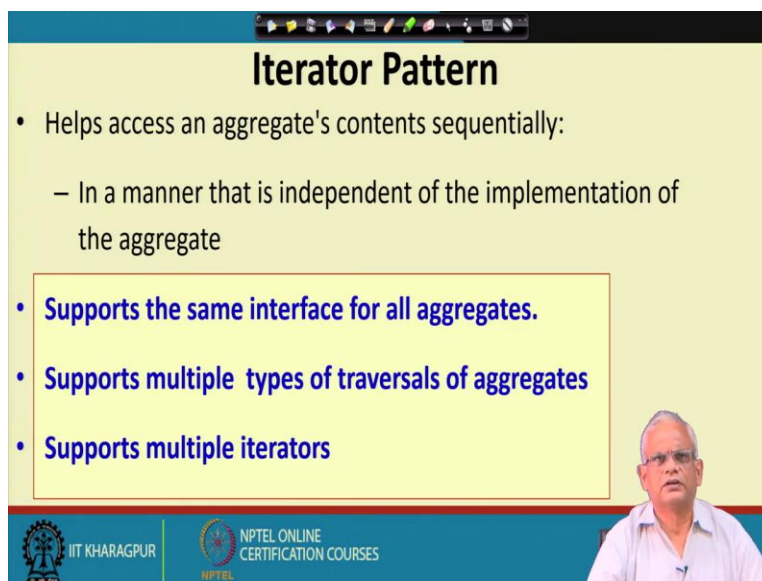
We do not know how the programmer who has implemented this aggregate would have stored it? did he use an array, array list, linked list we do not know. We do not even have to look at the code, find out how exactly it is internally presented, if you have the iterator pattern used, we can

simply call the iterator to return us one element after the other. And therefore, it makes it very easy to develop a program, to maintain a program, we can add new iterators, or change the iterator without much effort.

In a typical programming scenario, the programmers who are not aware of the iterator pattern, they implement the traversal algorithm in the aggregate itself, and also for the aggregate there may be several types of traversal and this makes the aggregate very complex and also the aggregate has too many responsibilities not only does it have to store elements but also it has to support many types of traversal algorithms.

The iterator pattern splits this responsibility from the aggregate object and gives this responsibility of traversal to an iterator object, so that's the intent to make the developer easily traverse through aggregates without knowing the internals of the aggregate and also it helps in maintenance because the iterator can be changed easily and new types of iterators can be added.

(Refer Slide Time: 03:56)



The slide is titled "Iterator Pattern" and contains the following text:

- Helps access an aggregate's contents sequentially:
 - In a manner that is independent of the implementation of the aggregate
- **Supports the same interface for all aggregates.**
- **Supports multiple types of traversals of aggregates**
- **Supports multiple iterators**

At the bottom of the slide, there are logos for IIT Kharagpur and NPTEL Online Certification Courses, along with a small video inset of a man speaking.

The iterator pattern helps to access an aggregate's contents sequentially in a manner that is independent of the implementation of the aggregate, whether it is implemented as array, array list, linked-list whatever. The programmer does who wants to traverse the iterate just does the same thing, does not have to write any different code.

And therefore, implementation can be changed without the application code which requires traversal to change. The iterator supports the same interface for all types of aggregates whether it is a tree, binary tree and n-ary tree, whether it is an array list whatever the same set of methods are used for traversal. And also, multiple types of traversals are supported because in many applications it is necessary to traverse the aggregate in different ways.

For example, a tree maybe traversed preorder, in-order, post-order, level order and so on. We can easily have different iterators which when requested the traverse the tree in different orders. And also, we can easily support multiple iterators and also multiple clients performing iterations of the same aggregate at the same time concurrency without affecting each other's traversal but if it is implemented in the aggregate it becomes very difficult to support concurrent traversal by multiple clients.

So, here the programmer irrespective of type of aggregate uses the same set of methods for traversal. Multiple types of iterators are available to implement multiple types of traversals and also concurrent traversal by different clients becomes possible.

(Refer Slide Time: 06:43)

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

The slide features two diagrams: a green tree with 10 nodes and a red tree with 10 nodes. The red tree is a binary tree with root 'A', children 'B' and 'C', and grandchildren 'D', 'E', 'F', and 'G'. The bottom of the slide includes logos for IIT KHARAGPUR and NPTEL ONLINE CERTIFICATION COURSES, along with a small video inset of a man speaking.

The main idea here is that the aggregate is traversed which is independent of the way in which it is stored. For example, we might have a tree, a binary tree or we might have a n-ary tree and we

know what is a level order traversal of this tree. And irrespective of the type of tree we can perform the level order traversal of the tree.

(Refer Slide Time: 07:26)

The slide is titled "The Iterator Pattern: Context". It features a tree diagram on the right side with root node 'A' and children 'C', 'X', and 'H'. Node 'C' has children 'D' and 'E'. Node 'X' has children 'F' and 'G'. Node 'H' has children 'I' and 'J'. Below the tree, there is a video feed of a man speaking. The slide contains the following text:

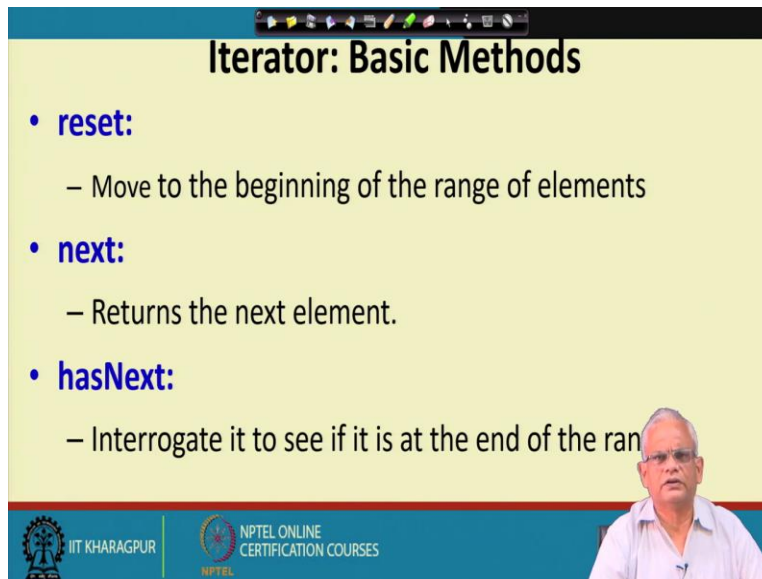
- It might be necessary to have more than one type of traversal on the same aggregate object.
 - Also, not all types of traversals can be anticipated *a priori*.
- One should not bloat the interface of the aggregate object with all possible traversals.

The slide footer includes the logos for IIT KHARAGPUR and NPTEL ONLINE CERTIFICATION COURSES.

It helps us to have different iterators which provide different types of traversals and also as the software evolves it becomes necessary to have different traversals of the same aggregate and that can be easily done by just adding a different iterator. We don't have to change the aggregate, we just add a new iterator.

In the conventional wisdom the program supports the traversal algorithm along with the aggregate which makes the aggregate very complex has many responsibilities, a bloated interface, but with iterator we split this responsibility and the software becomes more usable and also easier to develop.

(Refer Slide Time: 08:32)



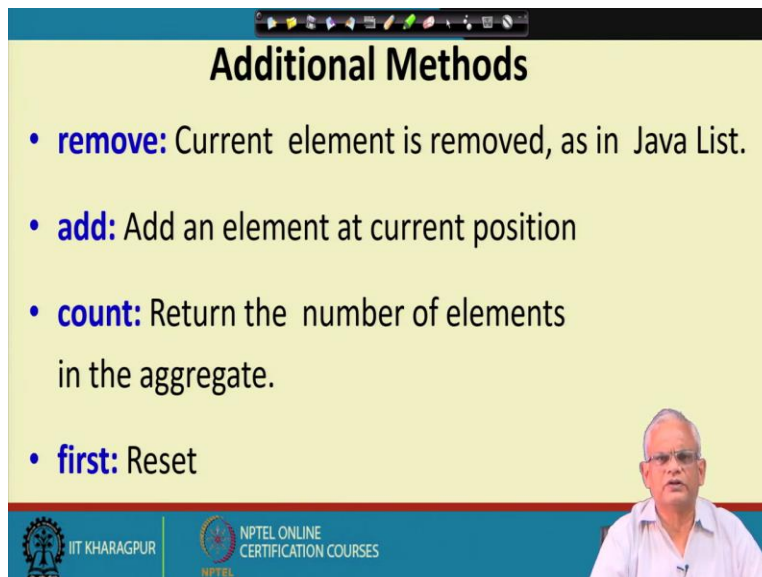
Iterator: Basic Methods

- **reset:**
 - Move to the beginning of the range of elements
- **next:**
 - Returns the next element.
- **hasNext:**
 - Interrogate it to see if it is at the end of the range

IT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

Now, let's look at what are the basic methods supported in an iterator. Basic methods are reset that is to the first element, it is reset to the beginning of the range of the elements. Next returns the next element in the aggregate. HasNext it's a method to check if there are more elements in the aggregate.

(Refer Slide Time: 09:03)



Additional Methods

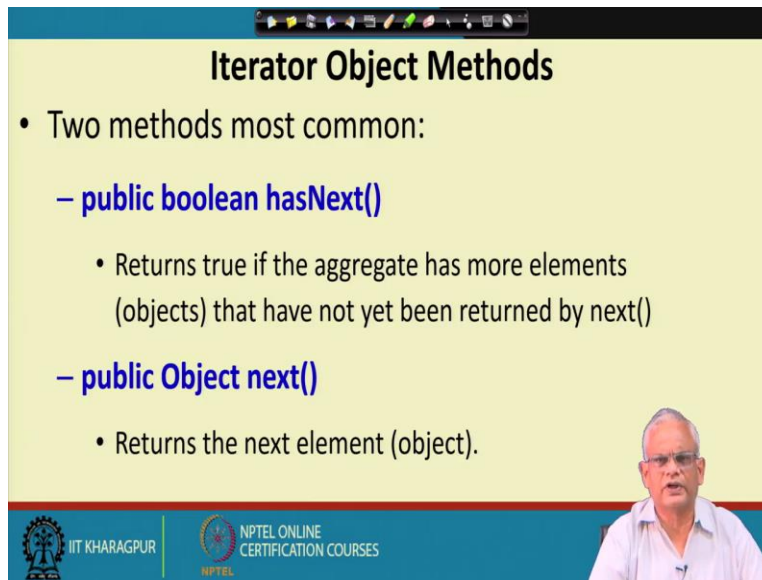
- **remove:** Current element is removed, as in Java List.
- **add:** Add an element at current position
- **count:** Return the number of elements in the aggregate.
- **first:** Reset

IT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

There also additional methods depending on the application if it is a read-only application, we don't need remove, add etc. but then sometimes we also need to change the aggregate we might

should be able to add new elements, remove elements, change elements and so on. And these are the methods supported additional methods depending on the application. Remove do current element is remove, add an element at the current position, count the number of elements in the aggregate and first is sometimes supported which is same as reset.

(Refer Slide Time: 09:50)



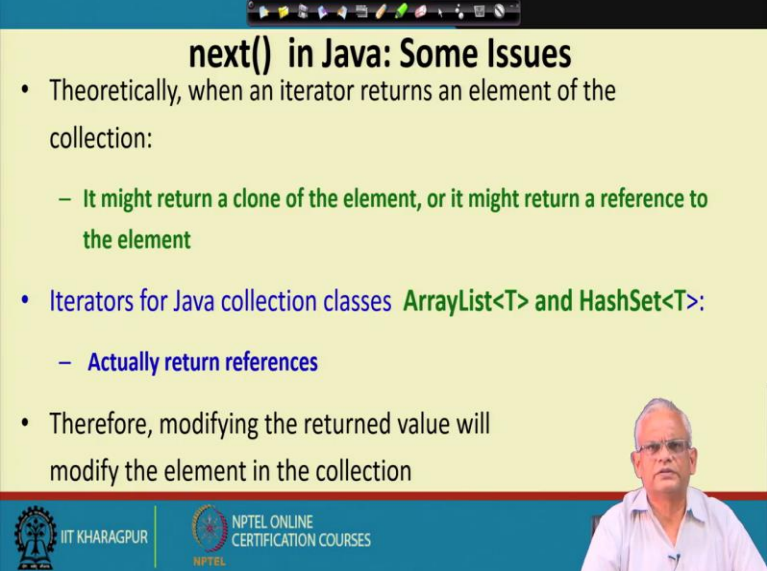
Iterator Object Methods

- Two methods most common:
 - **public boolean hasNext()**
 - Returns true if the aggregate has more elements (objects) that have not yet been returned by next()
 - **public Object next()**
 - Returns the next element (object).

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

But irrespective of the iterators used in various applications two methods are most common: one is hasNext that is whether we are at the end of the aggregate or there are more elements to traverse, it returns true if the aggregate has more elements that have not yet been returned by the next. And next method which returns the next element in the aggregate which has not so far been traversed.

(Refer Slide Time: 10:26)



next() in Java: Some Issues

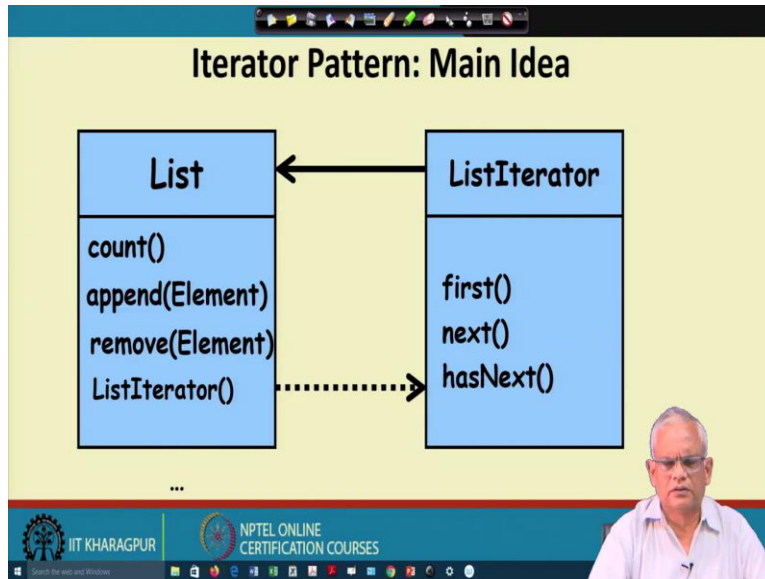
- Theoretically, when an iterator returns an element of the collection:
 - It might return a clone of the element, or it might return a reference to the element
- Iterators for Java collection classes `ArrayList<T>` and `HashSet<T>`:
 - Actually return references
- Therefore, modifying the returned value will modify the element in the collection

IT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

In Java programming we must be aware of some issues with the next method with the iterator. When an element is returned by a method there are two possibilities: one is that a copy of the element in the aggregate is made and the copy is returned to the client or reference to the element itself can be returned.

In the Java collection classes such as array list, hash set, etc. when we call the next method actually the reference is returned and therefore if we make any changes to the object it will be reflected in the aggregate itself. And therefore, it becomes very easy to change the element we do not need a separate method to change a specific element.

(Refer Slide Time: 11:48)



The main Idea here is summarized in this diagram (above slide) with an example of a list and in the list, we have appended, remove and also, we have the list iterator which creates the iterator. The list iterator method creates the list and once the list iterator is created then we can have on that object have the first, next hasNext, etc invoked.

(Refer Slide Time: 12:56)

The slide, titled "Iterator Pattern: Solution", lists the following points:

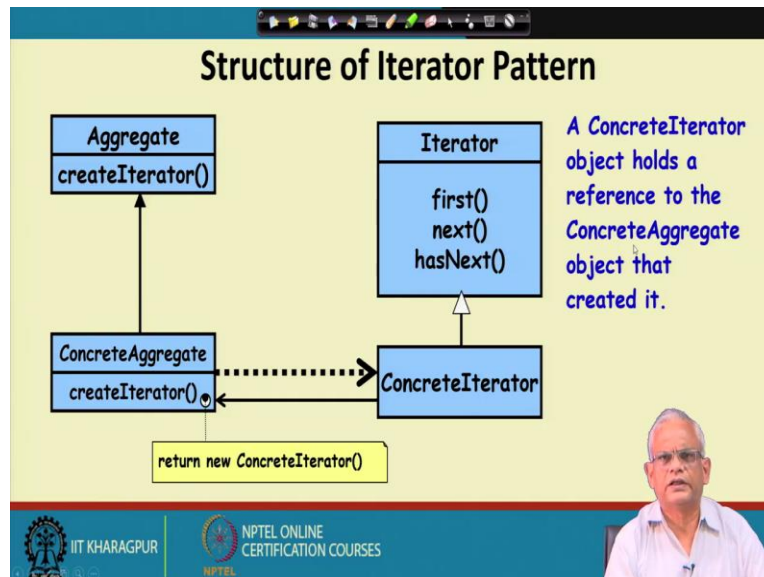
- The responsibility for accessing and traversing an aggregation object:
 - Placed in a separate iterator object and not in the aggregate object
- An iterator object should:
 - Provide the same interface for accessing and traversing elements, regardless of the class of the aggregate object and the kind of traversal performed...

The slide also features logos for IIT KHARAGPUR and NPTEL ONLINE CERTIFICATION COURSES, and a small video inset of a presenter.

So, the main classes here are the aggregate and the iterator. The responsibility for accessing and traversing an aggregate object is they are placed in the separate iterator object and not part of the

aggregate object. And the iterator object across various types of aggregates should have the same interface for accessing and traversing the elements and also for various types of traversals and various types of aggregates we have the same set of methods.

(Refer Slide Time: 13:40)



Now, this is the structure of the iterator pattern (in the above slide), very simple structure. The class structure is that the concrete aggregate is a subclass of the aggregate and the iterator, the concrete iterator is an implementation or a subclass of the iterator. The iterator can be abstract class or an interface. And in the concrete aggregate we have a create iterator method and as the create iterator method is called a new concrete iterator is created.

The concrete iterator object is created and the reference is kept track and also the concrete iterator is passed a reference to the aggregate. And therefore, the concrete aggregate can traverse through the elements of the aggregate and that's how this association relationship: one is this dotted arrow which is the create concrete iterator, the concrete aggregate creates concrete iterator and the other arrow is association arrow. The concrete iterator is passed a reference of the aggregate and the concrete iterator stores reference to the aggregate and uses it for traversal. The concrete iterator holds a reference to the concrete aggregate object that created the iterator.

(Refer Slide Time: 15:33)

Iterator Participants

- **Iterator**
 - Defines an interface for accessing and traversing elements
- **ConcreteIterator**
 - Implements the Iterator interface.
 - Keeps track of the current position in traversal of the aggregate
- **Aggregate**
 - Defines an interface for creating an Iterator object
- **ConcreteAggregate**
 - Creates and returns an instance of ConcreteIterator

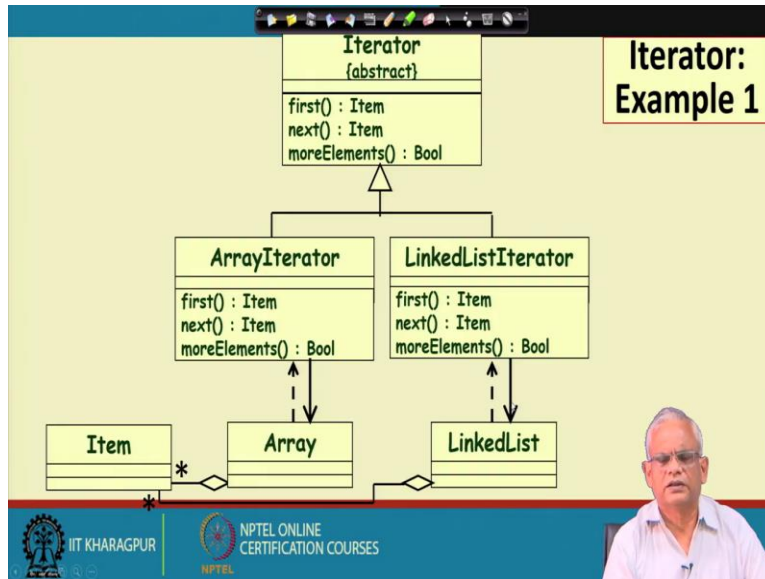
```
classDiagram
    class Aggregate {
        +CreateIterator()
    }
    class Iterator {
        +First()
        +Next()
        +IsDone()
        +CurrentItem()
    }
    class ConcreteAggregate {
        +CreateIterator()
    }
    class ConcreteIterator {
    }
    Aggregate <|-- ConcreteAggregate
    Iterator <|-- ConcreteIterator
    ConcreteAggregate ..> ConcreteIterator
```

IT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

Now, the participants in the iterator pattern or the iterator which is typically an interface or an abstract class. The concrete iterator implements the iterator interface or is a subclass of the iterator abstract class. And it keeps track of the current position of the traversal of the aggregate, so that when the next is called it returns the next element.

The aggregate just defines an element for creating the iterator object so that is create iterator and the concrete aggregate creates and returns an instance of the concrete iterator.

(Refer Slide Time: 16:25)



Now, let's look at an example of the iterator pattern use (in the above slide). There to aggregates here: one is an array which contains many items and we have a linked list which also contains many items. The array iterator, linked list iterator they are subclasses of the iterator which is an abstract class and the array create an array iterator object by passing a reference to itself, and therefore the array iterator has an association relationship with the array where it keeps track of the reference ID of the array.

Similarly, the linked list creates the linked list iterator while passing a reference of itself and the linked list iterator keeps reference to the linked list, the first item, next item and so on and that is how the association relationship is maintained.

(Refer Slide Time: 17:53)

```
public class ArrayIterator
implements Iterator {
    private String[] data;
    private int index;
    public ArrayIterator
(String[] data) {
        this.data = data;
        this.index = 0;
    }
    public String first() {
        index = 0;
        return data[0];
    }
    public String next() {
        return data[index++];
    }
    public boolean moreElements() {
        if (index >= data.length)
            return false;
        return true;
    }
}
```

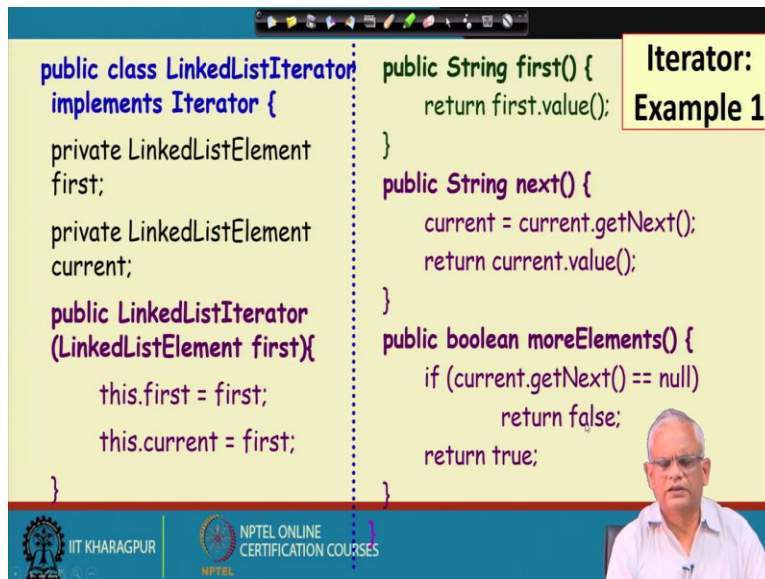
**Iterator:
Example 2**

IIT KHARAGPUR | NPTEL ONLINE
CERTIFICATION COURSES

Now, what about the code? (in the above slide) The array iterator implements the iterator interface and it has data and there is an index and the constructor for the array iterator. The reference of the aggregate is passed and it just stores here. And then the index is set to zero (`this.index=0`) that is the first element. And the method `first` also does the same thing, it sets the index to zero and returns the first element.

And the next it returns the data of the next element. More elements check whether there are more elements that is index is greater than equal to data length (`index>=data.length`), returns false otherwise. It returns true that there are more elements, so the code is really simple. This pattern is quite intuitive and simple pattern.

(Refer Slide Time: 19:11)



```
public class LinkedListIterator
implements Iterator {
    private LinkedListElement
    first;
    private LinkedListElement
    current;
    public LinkedListIterator
    (LinkedListElement first){
        this.first = first;
        this.current = first;
    }
    public String first() {
        return first.value();
    }
    public String next() {
        current = current.getNext();
        return current.value();
    }
    public boolean moreElements() {
        if (current.getNext() == null)
            return false;
        return true;
    }
}
```

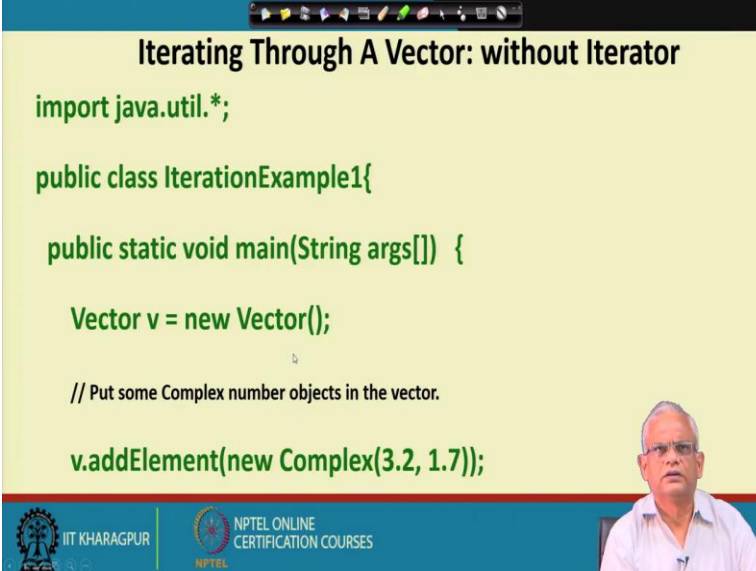
**Iterator:
Example 1**

IT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

Similarly, we can have the code for the linked list iterator (in the above slide) which implements the iterator interface and here there are two private data: one is the first element of the list and the other is the current element of the list that is the first and current. And as the constructor is called the callee that is the aggregate, it passes the first element and the first and current are both set to first (`this.first=first; this.current=first`).

And when the first is called it returns the value for the first element, the next it returns the value for the next element and more elements is just checks weather there are more elements and depending on that it returns false or true.

(Refer Slide Time: 20:08)



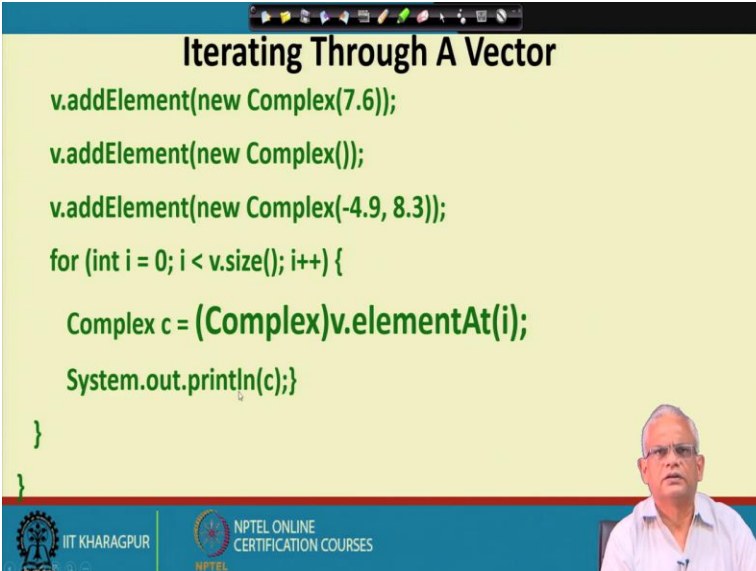
The slide displays the following Java code:

```
import java.util.*;  
  
public class IterationExample1{  
  
    public static void main(String args[]) {  
  
        Vector v = new Vector();  
  
        // Put some Complex number objects in the vector.  
  
        v.addElement(new Complex(3.2, 1.7));  
    }  
}
```

The slide also features the IIT Kharagpur and NPTEL Online Certification Courses logos at the bottom, and a small video inset of a man in the bottom right corner.

Now, let's just contrast the way we can use a vector without an iterator and how we write the code with iterator. Those who have done Java programming a bit they would already be knowing this but just for the sake of comparison we are just writing here and will spend a minute or two on that.

(Refer Slide Time: 20:45)



The slide displays the following Java code:

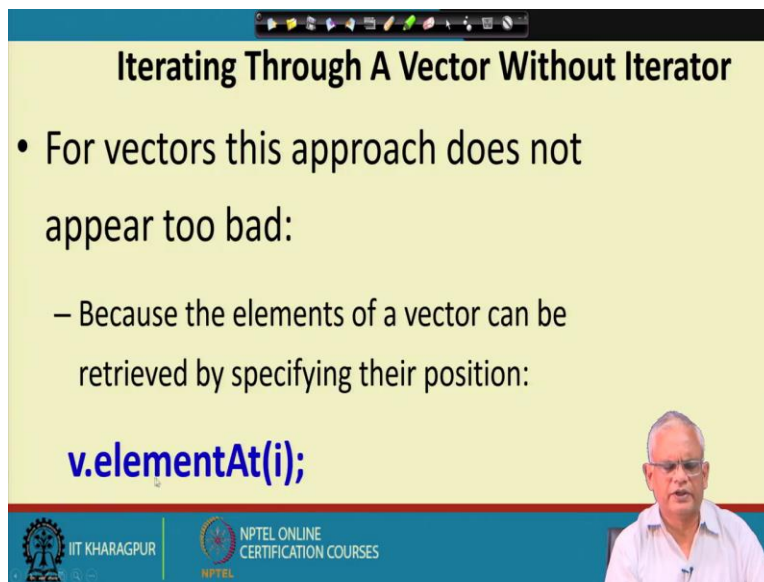
```
v.addElement(new Complex(7.6));  
v.addElement(new Complex());  
v.addElement(new Complex(-4.9, 8.3));  
  
for (int i = 0; i < v.size(); i++) {  
  
    Complex c = (Complex)v.elementAt(i);  
  
    System.out.println(c);  
  
}
```

The slide also features the IIT Kharagpur and NPTEL Online Certification Courses logos at the bottom, and a small video inset of a man in the bottom right corner.

So, here we have a vector created (in the above slide) and then we add elements to the vector: the first element, second element and third element and the main code for traversal is here in a loop.

We get the size of the vector, and then we just get the element and then display it, that is the typical way of programming.

(Refer Slide Time: 21:19)



The slide features a title bar with navigation icons. The main content is on a light yellow background. It includes a title, a bullet point, a sub-point, and a code snippet. A small video inset of a speaker is in the bottom right corner. The footer contains logos for IIT Kharagpur and NPTEL.

Iterating Through A Vector Without Iterator

- For vectors this approach does not appear too bad:
 - Because the elements of a vector can be retrieved by specifying their position:

```
v.elementAt(i);
```

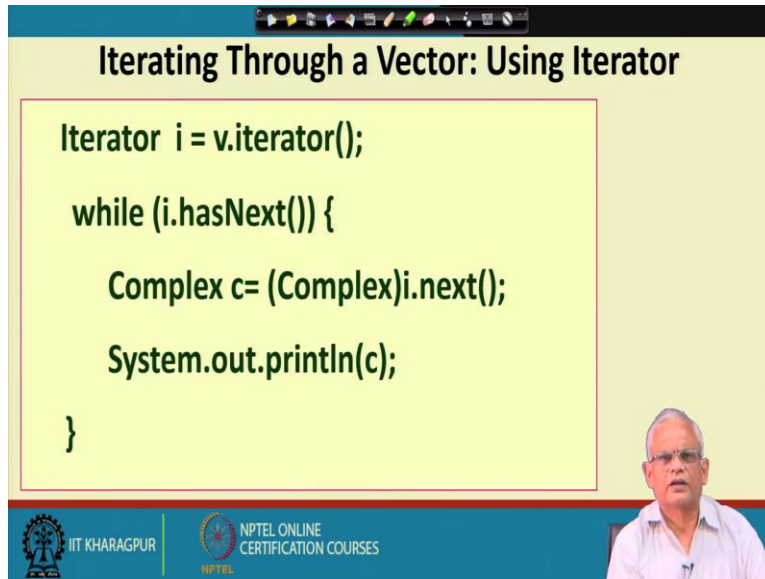
IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

But not too bad because we had this method `v.elementAt(i)`; but for many types of aggregates we may not have such a method. For example, a tree which may be implemented in different ways we may not have a way to increment `i` and get the element. So the vector without the iterator is not too painful, is not too bad we just increment `i` and get the element `v.elementAt(i)`. But for tree and so that might be difficult.

(Refer Slide Time: 22:03)

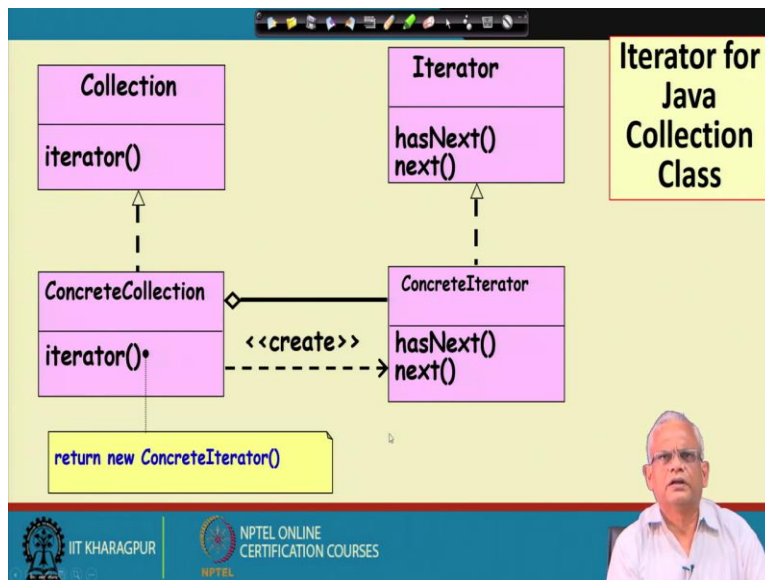
Iterating Through a Vector: Using Iterator

```
Iterator i = v.iterator();  
  
while (i.hasNext()) {  
    Complex c= (Complex)i.next();  
    System.out.println(c);  
}
```



Now, let say how we traverse the same vector using an iterator (in the above slide). The first is we create the iterator object by calling `v.iterator()` and `i` is the iterator object and then we call the method `i.hasNext()` with more elements and get the next element `i.next()` and then display that and you can see that the code is much more elegant than the code for the one without the iterator.

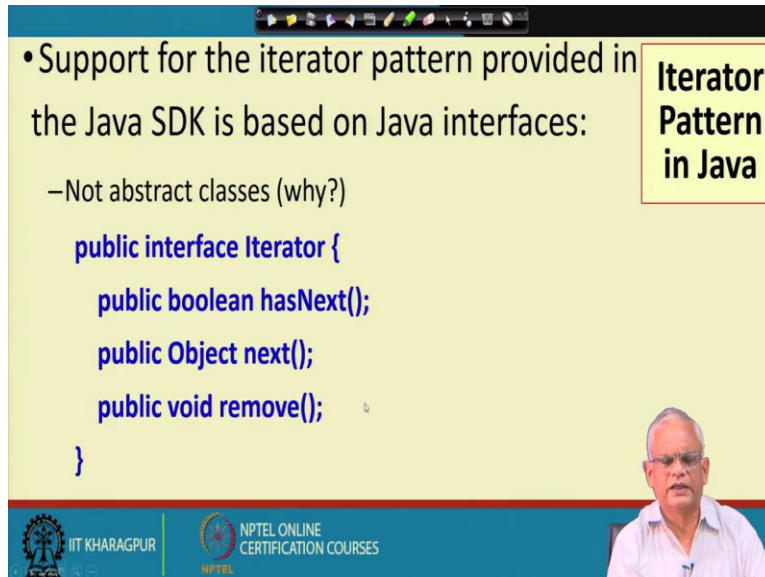
(Refer Slide Time: 22:41)



Now, let's look at the iterator for the Java collection classes. In the Java collection classes we have the iterator interface which has the `hasNext` and `next` methods just like the iterator pattern

we are discussing. The concrete iterator implements the iterator and we call the iterator method which returns the concrete iterator and then we keep on traversing the collection using the next and hasNext.

(Refer Slide Time: 23:25)



• Support for the iterator pattern provided in the Java SDK is based on Java interfaces:

- Not abstract classes (why?)

```
public interface Iterator {  
    public boolean hasNext();  
    public Object next();  
    public void remove();  
}
```

Iterator Pattern in Java

NPTEL ONLINE CERTIFICATION COURSES


IIT KHARAGPUR

In the Java SDK the iterator is an interface, so this is the public interface iterator hasNext, next and remove and so on. But then just asking that why it is not an abstract class and why is it an interface? The answer to this is that we have various types of collections starting from array list, sets and so on, hash set and so on. So, the traversal for them is very different and therefore we cannot define the methods in the abstract class we will have to anyway override and therefore just provide the template of the methods in the interface and these are implemented by the different iterators in the different aggregates.

(Refer Slide Time: 24:31)

Java Iterator

- An Iterator object is an instance of a class that implements the Iterator interface:
 - Java Iterator interface defines the `hasNext()`, `next()`, and `remove()` methods

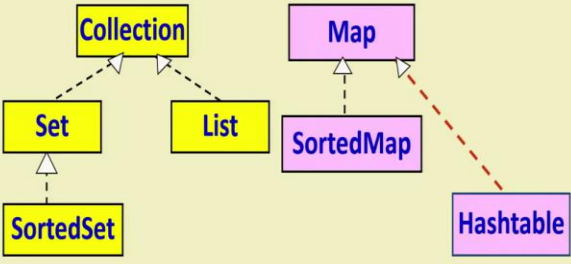


IT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

An iterator object is an instance of class that implements the iterator interface and the iterator-interface we have already seen has the method `hasNext`, `next` and `remove`.

(Refer Slide Time: 24:46)

Java Collections



```
graph TD; Collection[Collection] -.-> Set[Set]; Collection -.-> List[List]; Set -.-> SortedSet[SortedSet]; Map[Map] -.-> SortedMap[SortedMap]; Map -.-> Hashtable[Hashtable];
```

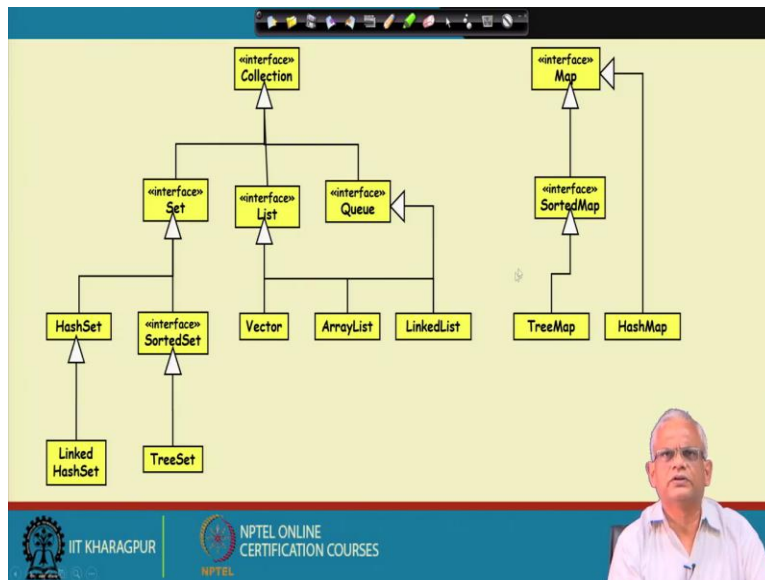
- Hashtable is an old (pre-Collections) class
- Hashtable has been retrofitted to implement the Map interface

IT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

The Java collections if we look, we have the set, list and the sorted set and so on and on the other hand we have map, sorted map and so on and the hash table is an old pre-collection class it existed before the Java collections and once the Java collections were defined these were fitted

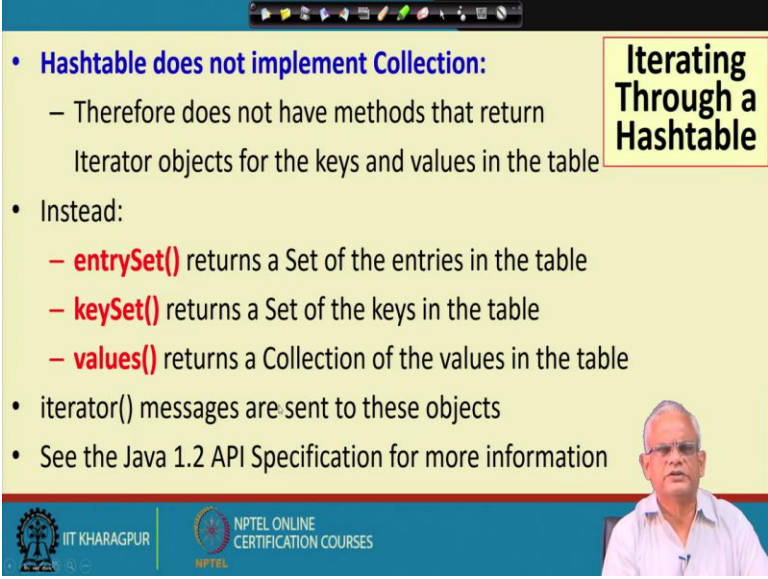
here on the map but the map does not implement the collection interface. So, these don't have iterators and let us see how iteration in the map can be performed.

(Refer Slide Time: 25:30)



So, this is the more complete diagram of the collection set (in the above slide): list, queue, hash set, sorted set, linked hash set, tree set, vector, array list, linked list and so on and on the map we have sorted map, tree map and hash map. So, these are the collection classes the left side and all of these provide iterators to create an iterator and traverse whereas on the map we do not have an iterator this do not implement the iterator interface.

(Refer Slide Time: 26:06)



Iterating Through a Hashtable

- **Hashtable does not implement Collection:**
 - Therefore does not have methods that return Iterator objects for the keys and values in the table
- Instead:
 - **entrySet()** returns a Set of the entries in the table
 - **keySet()** returns a Set of the keys in the table
 - **values()** returns a Collection of the values in the table
- iterator() messages are sent to these objects
- See the Java 1.2 API Specification for more information

IT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

But then how does one traversed element in the hash table? Here we cannot create iterator object because it is, does not implement the iterator interface. Instead, it has entry set, key set and values defined which create set of entries, set of keys and collection of values and the set of entries, the set of keys and the collection of values for this we can have iterators.

The basic hash table we don't have iterators but then once we call the method entry set, key set and values we get collections and for this we can get the iterator object and traverse this.

(Refer Slide Time: 27:08)

Java Iterator

```
public interface java.util.Collection
... // List, Set extend Collection
    public Iterator iterator();
}

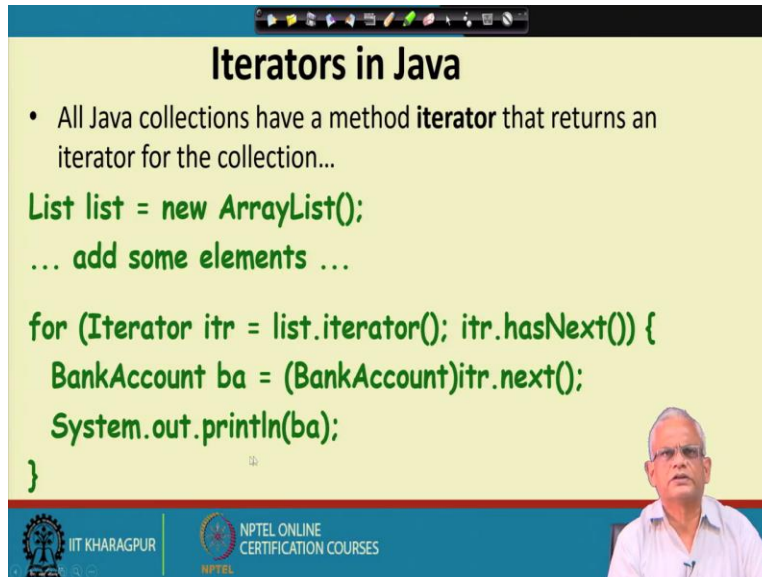
public interface java.util.Map {
...
    public Set keySet();
    public Collection values();
}

public interface java.util.Iterator {
    public boolean hasNext();
    public Object next();
    public void remove();
}
```

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

So, this is the Java iterator (in the above slide), the iterator is defined in Java.util, it is a interface and then hasNext, next and remove are the methods and the collection implements the iterator interface, whereas map does not implement the iterator interface on the other hand you can get key set values and so on which are collections and you can define iterators for these key set values and so on.

(Refer Slide Time: 27:46)



Iterators in Java

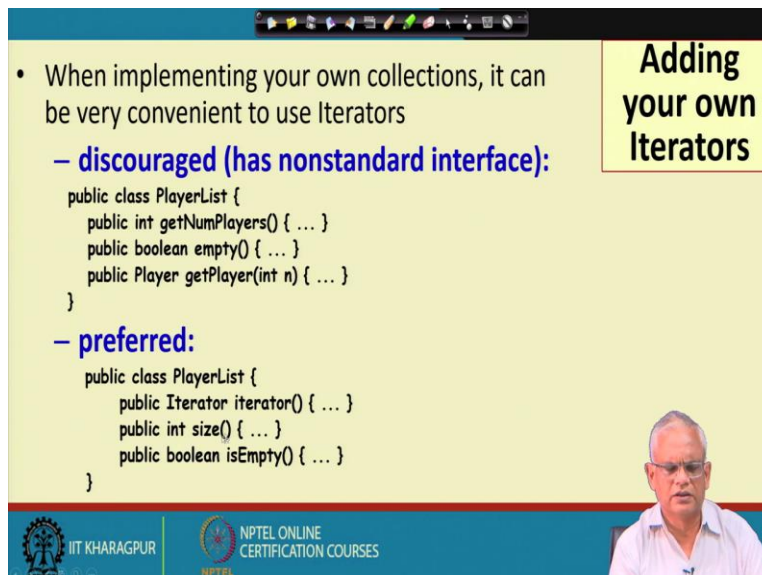
- All Java collections have a method `iterator` that returns an iterator for the collection...

```
List list = new ArrayList();  
... add some elements ...  
  
for (Iterator itr = list.iterator(); itr.hasNext()) {  
    BankAccount ba = (BankAccount)itr.next();  
    System.out.println(ba);  
}
```

IT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

The uses of iterators in Java might be familiar that we first create a list and do some operations on that, define the elements and so on and for traversal it's very simple using the iterator we just create an iterator object 'itr' by calling `list.iterator` and then while iterator has next we get the next element `itr.next` and then we just process that.

(Refer Slide Time: 28:24)



Adding your own Iterators

- When implementing your own collections, it can be very convenient to use Iterators

- **discouraged (has nonstandard interface):**

```
public class PlayerList {  
    public int getNumPlayers() { ... }  
    public boolean empty() { ... }  
    public Player getPlayer(int n) { ... }  
}
```
- **preferred:**

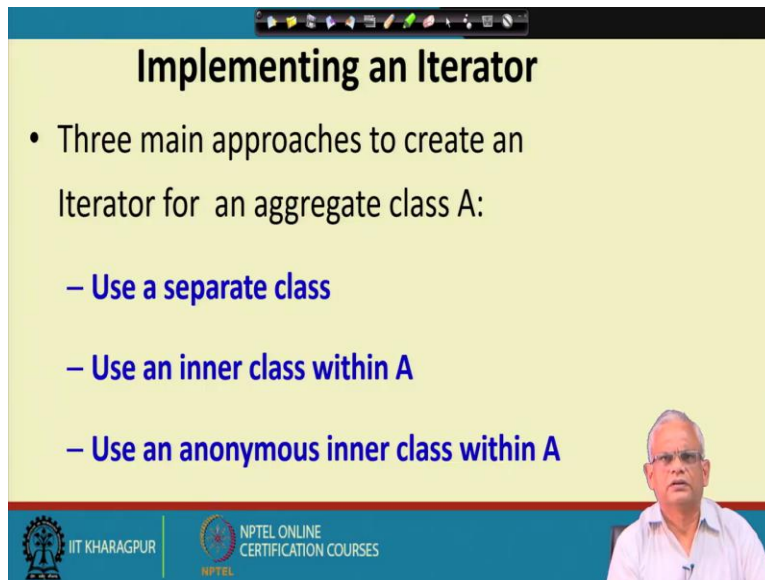
```
public class PlayerList {  
    public Iterator iterator() { ... }  
    public int size() { ... }  
    public boolean isEmpty() { ... }  
}
```

IT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

We can define our own iterators in a collection that we define. We might define an aggregate and we can have iterator implemented for that, we can implement the iterator interface or we can

write our own interface here. But then please do not use non-standard interface like get number of players, get player etc, we should use the standard terms because that makes it easier to program consistent iterators across all types of aggregates so we have the iterator object created here, size is empty and so on.

(Refer Slide Time: 29:15)



Implementing an Iterator

- Three main approaches to create an iterator for an aggregate class A:
 - Use a separate class
 - Use an inner class within A
 - Use an anonymous inner class within A

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

Implementing an iterator in our own application we have many different possibilities, one is using a separate iterator class just like we had seen or we can make it an inner class within the aggregate class or we can use an anonymous inner class within A. We have already seen an example of a separate class.

We can use the inner class because the iterator is used only in the context of that aggregate and therefore it can be inner class of the class and the advantage of this is that we don't have to separately pass the reference to the element to the aggregate to the iterator because it being an inner class already has access to the private data of the aggregate. And we can also use an anonymous inner class within A.

We are almost at the end of this lecture. We will just end here.

Thank you.