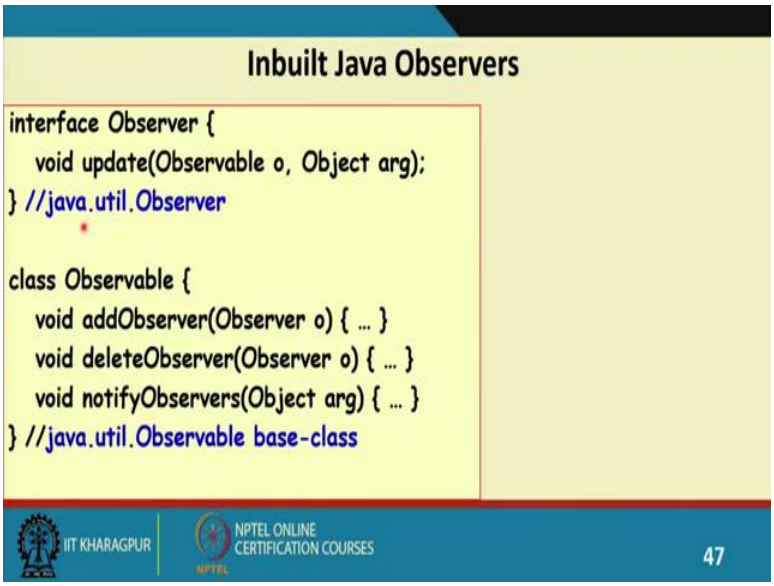
Object - Oriented System Development using UML, Java and Patterns
Professor Rajib Mall
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur
Lecture 45
Singleton Pattern-I

Welcome to this session. In the last session, we were discussing about the observer pattern, very important pattern used in many applications. I am sure you will have opportunity to use the observer pattern. The observer pattern is so important that it has been in built into the Java language. Let us see, what is the support for Java?

(Refer Slide Time: 00:41)



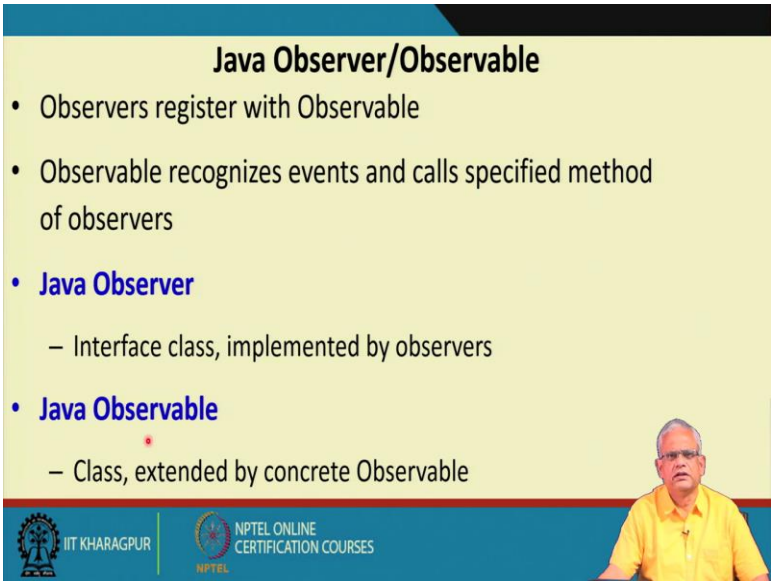
In the java.util.Observer, we have the interface observer already defined, which has the method update. The observer updates it when the observable or the model calls the update method when a significant change to the model occurs and it passes the observable o and the object argument. Please think why we need the observable o, because the observer is observing a specific observable.

Why is the observable ID is passed as parameter in the observer interface and also it passes an argument which is of type object and therefore any objective you pass is compatible to this argument, because the object is a class from which other classes are derived, all other classes are derived. So, whatever be the class you pass that will be compatible to the object class. You can

pass basically any object. Now to answer the question that why we need to pass the observer, observable or the model ID given that the observer is observing an observable or the model.

The answer is that the observer might be observing many observables. Many models and therefore the update is called in the context of which observable is identified by the observable ID here. The observable has the methods addObserver as always, the observer needs to register by addObserver calling the addObserver method of the observable and then delete observer, when the Observer wants to exit, calls the deleteObserver and for both of these the observer passes its own ID. And also, the notify observer when a significant change of the model occurs, the notify observer is called and also the context in which the model has changed is passed as argument, so that the notify observer will call the update method of all registered observers with the object arg as a parameter.

(Refer Slide Time: 03:52)

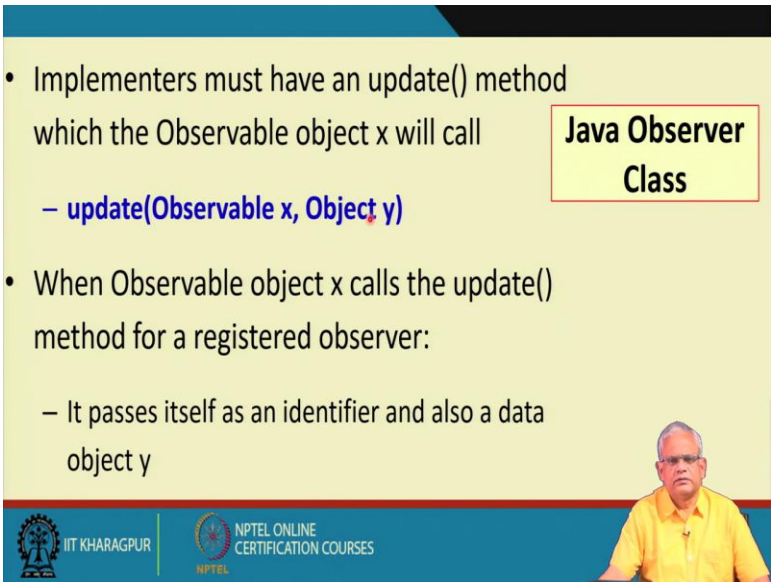


Now as usual, the observer register with observable, the observer recognizes any change to the model and then calls the notify observers. But one thing to notice is that we had the Java observer as an interface class as always. The observer is an interface class and it is implemented by observers, but then the observable if you would have noticed it is a concrete class, it is not an interface and then the concrete observables will inherit from the java observable class.

But then the question is to why is the Java observable a concrete class not an interface. Do not have a very good answer to that possibly the Java observer observable implementation is very

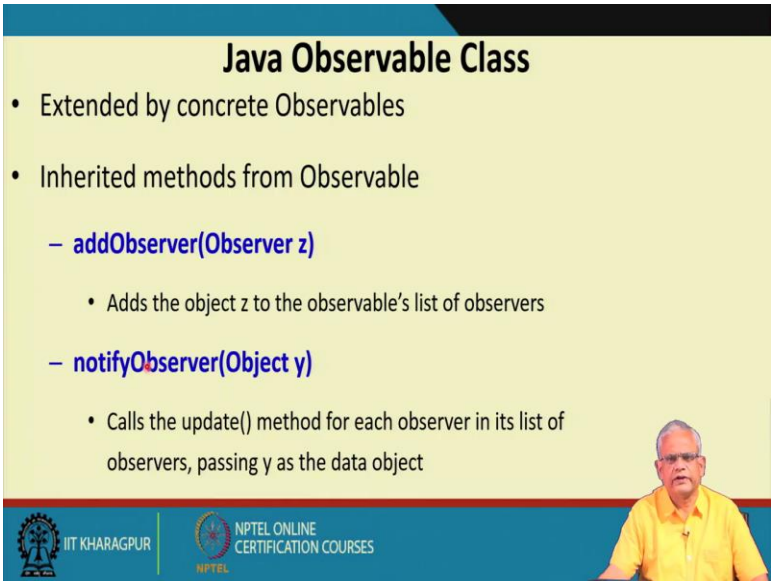
old and possibly they did not implement interface here observable is not an interface. But otherwise by sound design principle, the observable should have been an interface class. So, this I can point out or I can say that this is a shortcoming of the Java observer that has been inbuilt into the Java language.

(Refer Slide Time: 05:29)



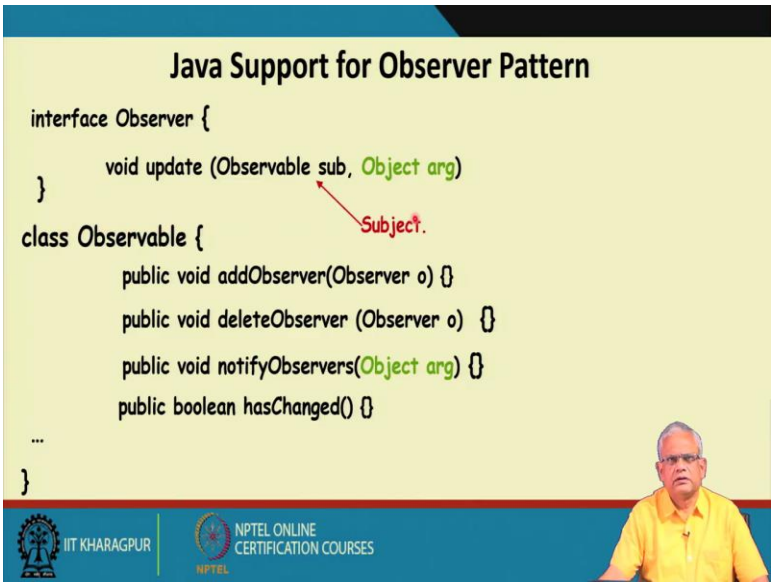
And as usual the update method is supported by the observer classes and then the observable x that is which observable has changed because an observer might be monitoring a state of many observables and based on this parameter, observable ID, the observer can know which observable or model has undergone a change and then the object Y is specific information about the change.

(Refer Slide Time: 06:12)



And as usual the Java observable class has addObserver where the observer registers and then the notifyObserver. As a change occurs the observable calls the notify and the update method of the observers are invoked inside the notify observer method.

(Refer Slide Time: 06:38)

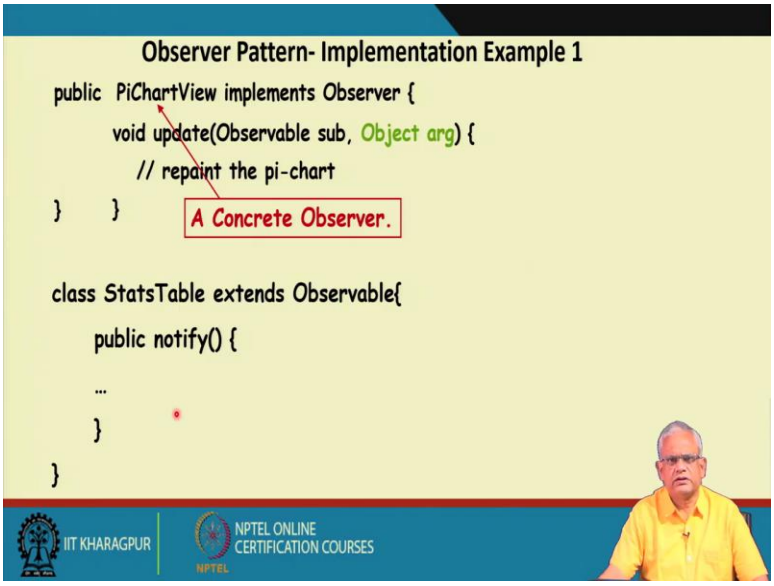


Now, let us see how we can use the Java inbuilt observer pattern. The observable as usual we have addObserver, the observer passes its ID, deleteObserver, the observer passes its own ID. notifyObservers, Objectarg this is slight deviation from the observer pattern because in the previous examples we discussed we did not have anything like observe Objectarg and as I was

mentioning that this is there for the observable to pass some information to the observer about the type of change that has happened and also the has changed, whether the model has changed and also notice that the observable inbuilt Java observable is a concrete class, it is not an interface and do not have a good explanation for this. The only explanation I can give is that the implementation of the Java inbuilt observer pattern is quite old and as we will see shortly that it had some difficulties and now the Java observer observable pattern is deprecated from the current Java distribution.

The observer is the interface and has update as usual and in the update the observable passes its ID and also passes some information regarding the type of change that has happened.

(Refer Slide Time: 08:41)

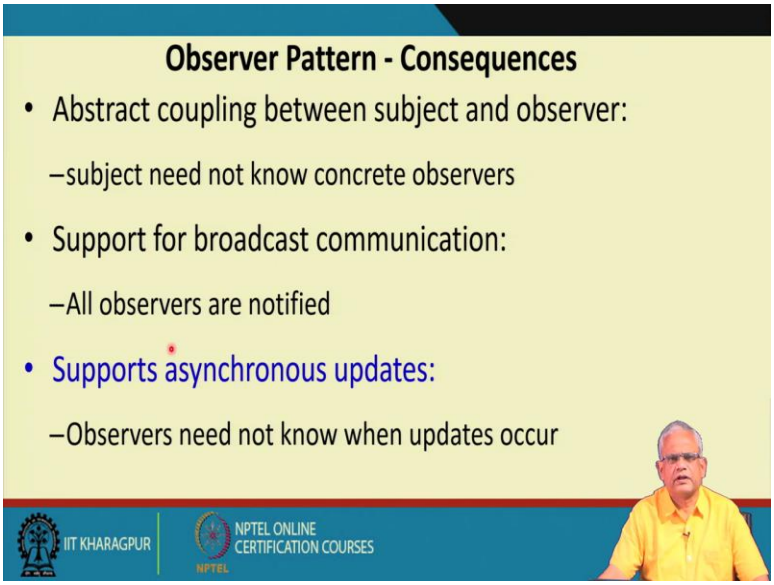


Now let us look at an example how the inability Java observer pattern can be used to write a simple application. Let us say we have a table and based on that we draw a pie chart and as the table changes the table may be changed through keyboard and we have one view which is a pie chart and another view is a, it can be a bar chart and so on.

We can have several types of observers and as the table changes, all these representations of the table should change. Now as you can see here that the update pie chart view implements observer and it has the update and has some code here how to repaint the pie chart and then the statistics table which extends the observable, statistics table is a concrete class and it extends the

observable, whereas the pie chart view implements the observer and then it has the notify method and so on.

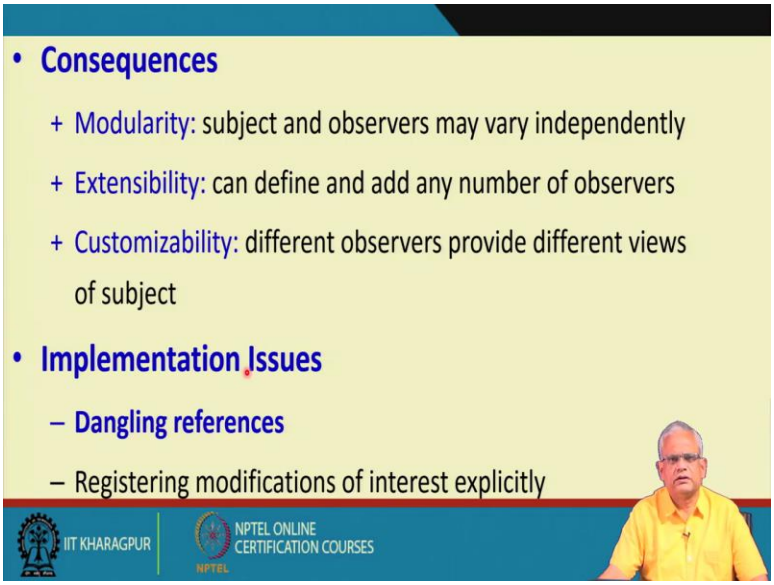
(Refer Slide Time: 10:06)



So far, we have looked at the observer pattern the context in which it is applicable the solution it offers, code examples, the inbuilt Java observer and so on. Now before we conclude the Java observer, let us just discuss a few final points. One is that the observer and observable are decoupled unlike a hard code reference if the observable maintains hard coded reference for the observer increases the coupling, requires the code to frequently change and is not a good solution.

The subject does not know the observable. They just register as the need arises and then they detach. Broadcast communication is possible because it has all the IDs of the observers and can call them as required. As the model changes asynchronously the update can be called and it meets the requirement of keeping the model and the observer synchronized when there is a change to the model.

(Refer Slide Time: 11:37)

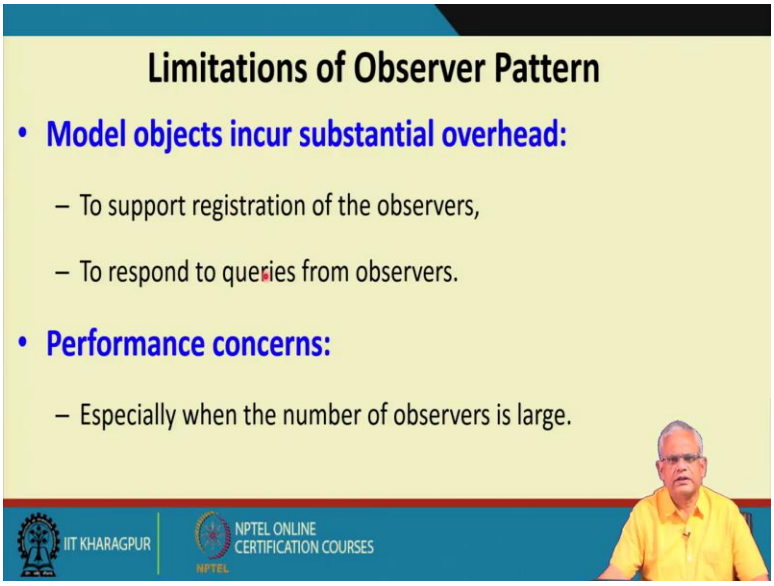


Makes the code good is modular, they vary independently, it is extendable, we can add many observers and observables, it is customizable. The different observers can have different implementations and do different activities on a model update. If it was the model which was managing the changes to the observer, for example, if the model used to call the paint method of the observer, update table of the observer and so on, it would be very difficult.

They will get very closely coupled and also the model we become extremely complicated not a good solution. Here the observer once notified about a change. Finds out, what is the change and manages its own updation. But one thing we must remember that, if the observers just do not detach themselves, they just attach and even though the observer has exited or has died without detaching then the observable has many observer IDs on which it cannot even invoke the method because the object does not exist and it will lead to errors.

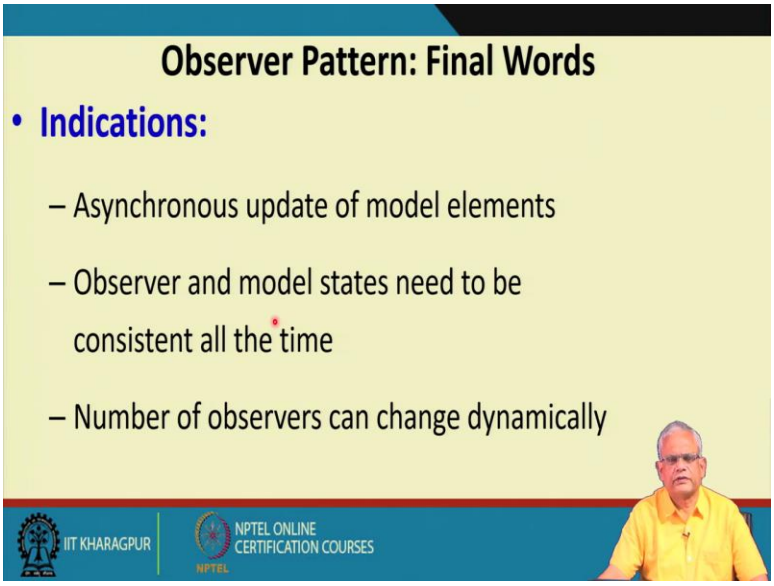
And therefore it is the programmer's responsibility to detach the observer when not required, even when the observer dies unexpectedly. The detached method the observable must be invoked.

(Refer Slide Time: 13:37)



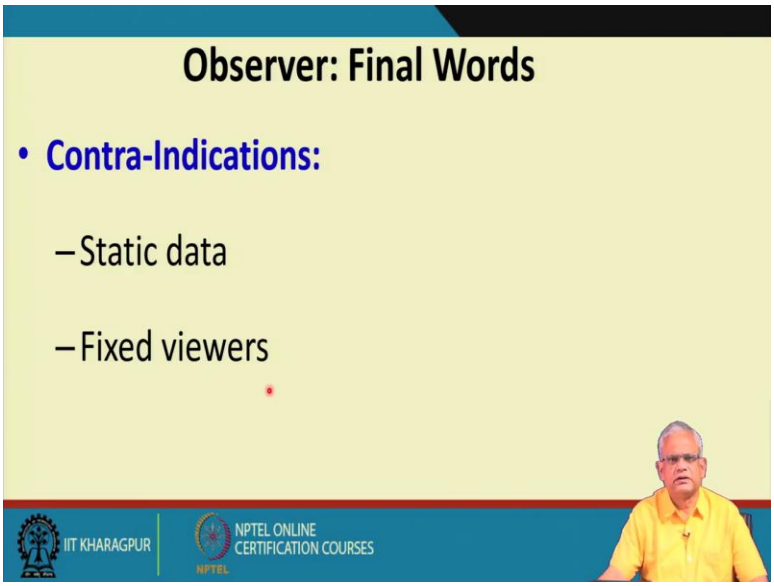
One possible short coming is that, if there are many observers then on every change invoking all the observer update methods is a substantial overhead and takes lot of time, if we have hundred observers and in a loop iterates over all the observers and calls their update method takes significant overhead and on top of that, if there are queries from the observers regarding the specific change to the model has occurred then there will be a performance penalty and the overall software will work slowly.

(Refer Slide Time: 14:24)



Just to summarize about the observer pattern. The observer pattern is useful when there is asynchronous update of the model in synchronous update we do not need the observer pattern just adds to the complexity. The observer and observable are consistent and the number of observers change dynamically. These are all possible in the observer pattern.

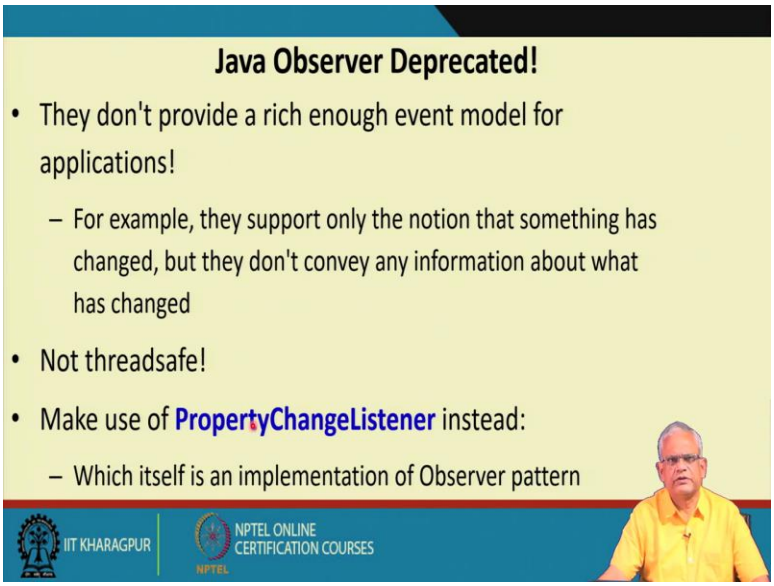
(Refer Slide Time: 14:57)



We should not use the observer pattern, when the model does not change the data is static and therefore if we implement the observer pattern in a situation where the model does not change, is unnecessary, makes the implementation unnecessarily complicated should be avoided and also

when the number of observers is fixed the specific observer and number of observers is fixed, then we do not need the observer pattern, the observable can directly hardcode the IDs of the observers because they are fixed and they are not going to change.

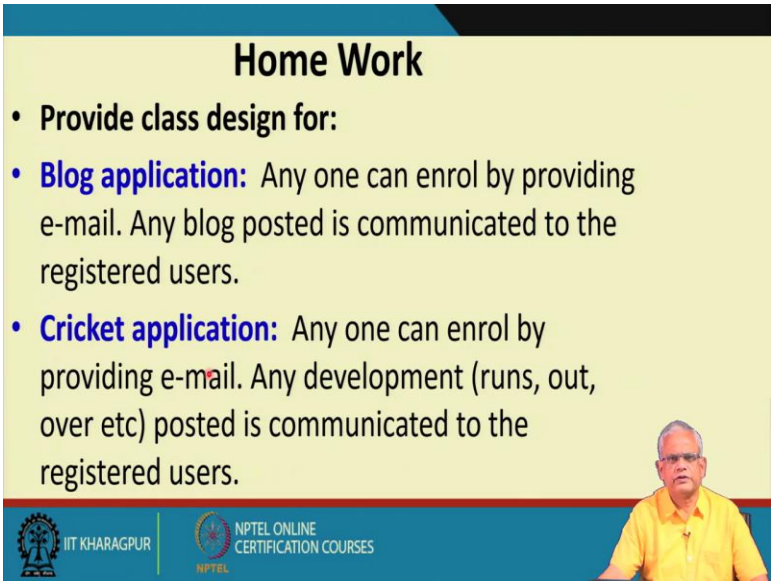
(Refer Slide Time: 15:50)



But as I was mentioning the inbuilt Java observer in recent distributions has been deprecated. Because of some of the shortcomings of the inbuilt Java observer, one was possibly that the observable is a concrete class since it was developed quite some time back when concurrent programming was not prevalent. It is not thread safe, it does not provide information about the specific changes the observer cannot express interest in an event or interest in a particular type of change and they get called only when that change occurs.

Here the observer is called irrespective of the type of the change and these are some of the shortcomings of the observer pattern and now the property change listener, we can consider this to be an implementation of the observer pattern and present Java programming we can use the property change listener whenever there is a need for something like an observer pattern.

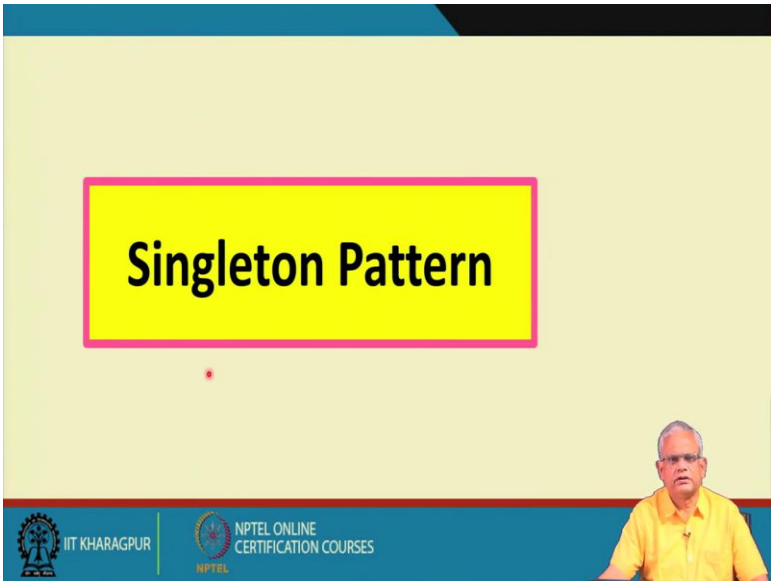
(Refer Slide Time: 17:14)

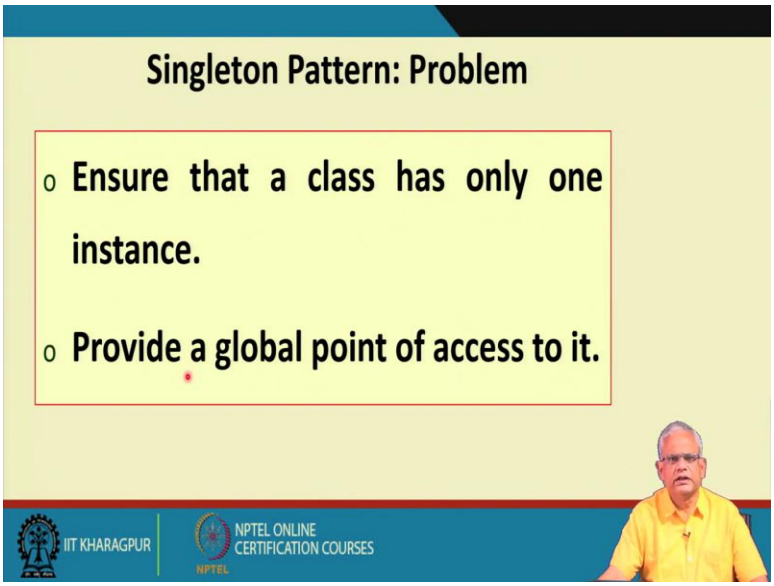


Now there is some homework here, please try to do based on the knowledge that we acquired about the observer pattern the context in which it is applied, the specific solution and the code, please try to do a blog application, where anyone can enrol in a blog by sending an email by providing the email ID, ofcourse have to call the method here.

And whenever is blog is posted, those who have registered with the email ID will be communicated about a new blog appearing. Again, this is asynchronous change the observers register by sending the email ID and the email is sent whenever there is a new blog post. Another homework is a cricket application, again here in the cricket application any user can enrol by providing email and whenever there is a change that run scored, out etc. It will be notified to the observers, please use the observer pattern to solve these two small problems.

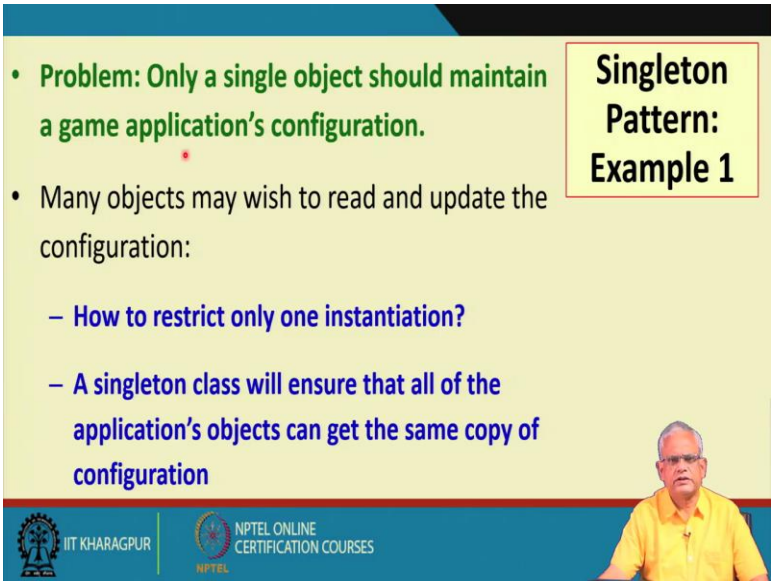
(Refer Slide Time: 18:38)





Now, let us look at the Singleton pattern. This is also a frequently used popular pattern. This is the creational pattern deals with how to create an object. But here the problem is that the class should have only one instance and we must ensure that only one instance of the class is created, we cannot create multiple instances and also we need to provide a global point of access to this single object, so that all other classes can access this object.

(Refer Slide Time: 19:22)

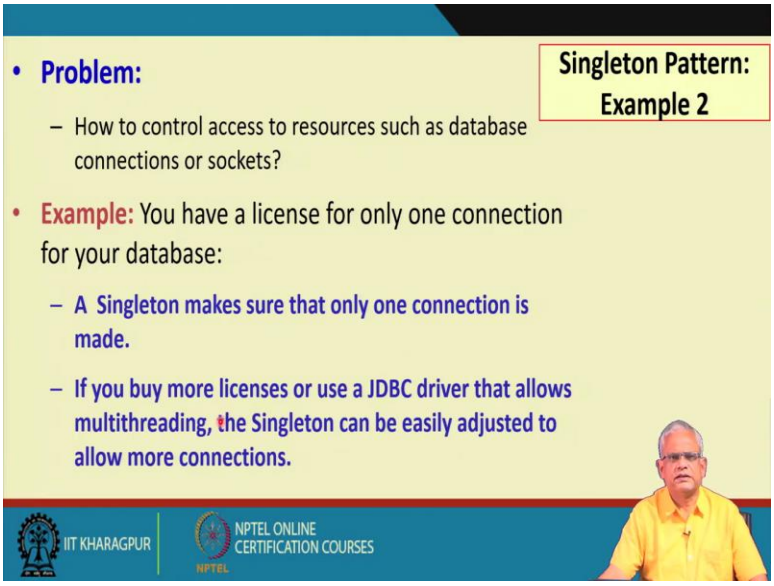


Now, let us see some example problems where a singleton pattern is useful, where there is a necessity for creating exactly one object. One is that we have a game application and the game applications configuration present, configuration is maintained in a single object. The configuration maybe the level at which the game will be played, is it beginner, intermediate, expert.

Another configuration parameter, maybe the timeout for the player, another configuration parameter maybe the specific display that is used and so on. But the problem here is that we should have exactly one configuration object. Whenever there is a chance per multiple configuration objects will be created then the game we get totally messed up and it will not work properly and we need to use observer pattern here.

Here there are many classes in the game application who will access the configuration, they should have a global point of access and there must be a single object created. Here we will use a singleton pattern will see the details of the singleton pattern that how will create exactly one object of the configuration.

(Refer Slide Time: 21:12)

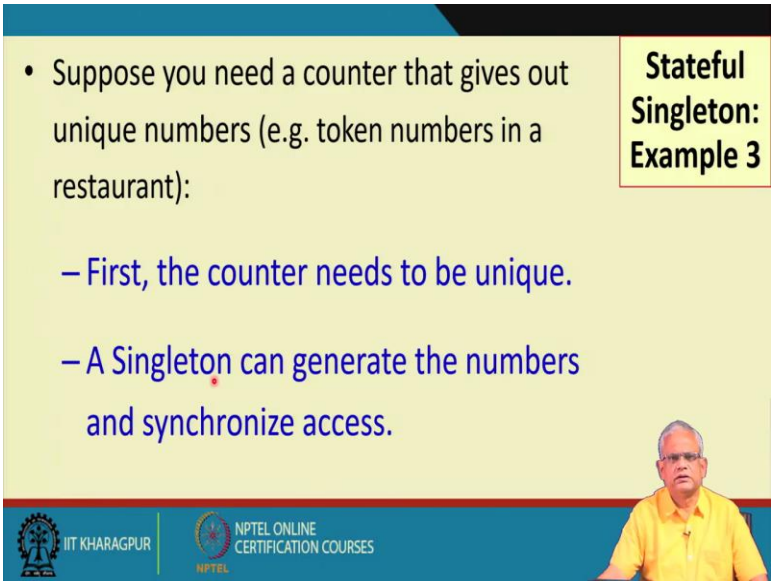


Another example is that we have a commercial database and then we have many parts of the application trying to access the database. Different parts can open connection to the database but then the license requires that we must have it anytime only one data base you should have a single connection to the database and here we create a singleton object which manages the connection.

All applications they get the singleton and request it to get a connection and if the connection is not taken, the singleton will provide connection and if the connection is already taken by another part, then it will make it wait. The singleton will make sure that a single connection is made to the database.

But again, we can slightly generalize the solution that the singleton will allow more connections. Maybe we purchase license for two, so at any time we will allow only two connections and this please observe that this is the internal of the singleton connection implementation whether it allows one or two, but then the connection management is done by a single object, if multiple objects can get created there is a chance that under some circumstances multiple connection management objects get created, then it will violate the license agreement and the application will not work.

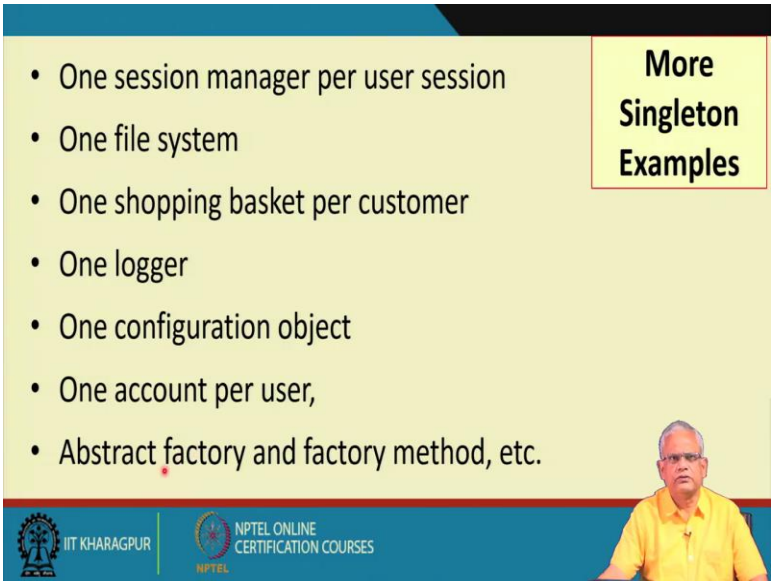
(Refer Slide Time: 23:20)



Third example is let us say we have a restaurant and then there are several parts of the restaurant, maybe on the snack counter, maybe on the dinner counter and so on and when the customers place order, they are issued a token. But the problem here is that the token should be unique. Based on the token, we should be able to identify the specific customer.

In this situation will use a singleton pattern, we will have a single object which is responsible for generating the token numbers and we need to have synchronized access of the different parts of the restaurant to this singleton object providing us tokens.

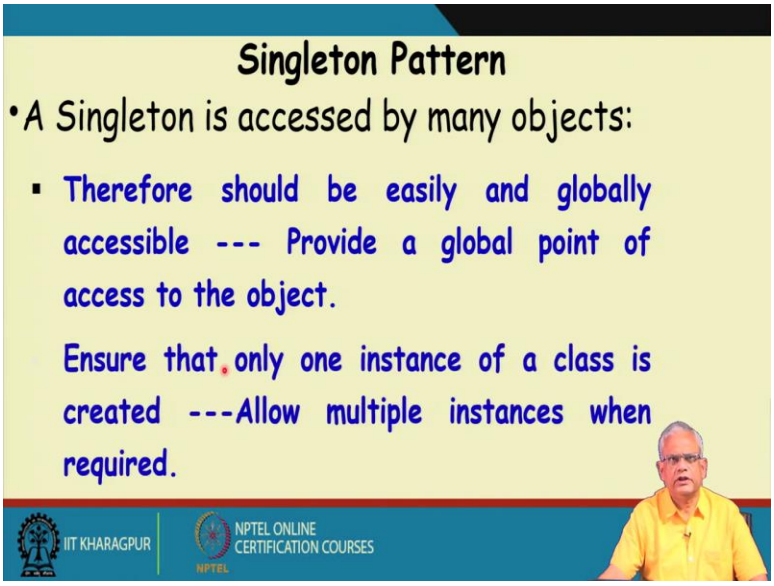
(Refer Slide Time: 24:22)



There are many other examples where singleton pattern is useful where we need to create exactly one object. Let us look at the other applications quickly. In a user session we need exactly one session manager. In a computer we need exactly one file system. In e-commerce application we should have exactly one shopping basket per customer. We cannot create multiple shopping baskets.

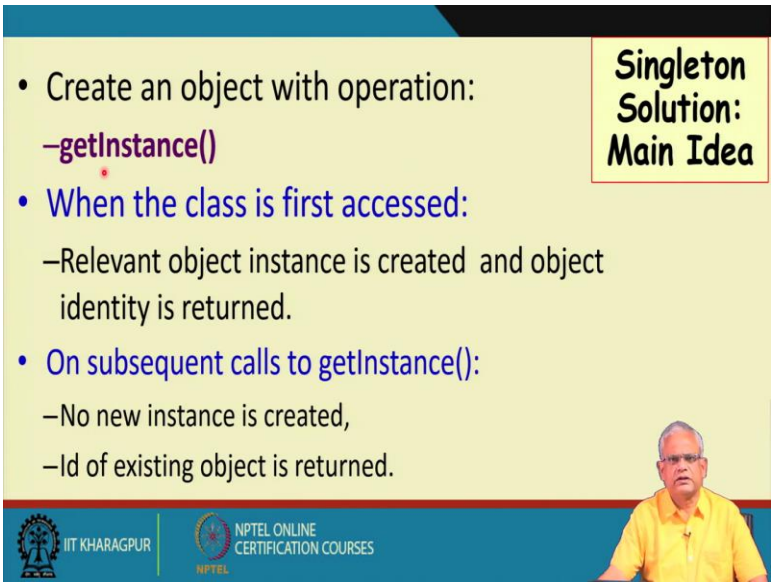
There is a one logger in application, it maintains exactly the log in a single file, if we have multiple logs coming up, the application will not work properly, one configuration per application one configuration object. We had seen that the example of a game configuration. One account per user and also many other single, many other patterns use the singleton pattern. For example, we will discuss about the abstract factory and the factory method they make use of the singleton pattern.

(Refer Slide Time: 25:39)



The context of the singleton is that the singleton is exactly one object which is easily accessible globally provides a single global point of access to different objects in an application and also, we need to ensure that exactly one object is created and ofcourse you can generalize little bit later, where if we, you might need exactly two objects and so on.

(Refer Slide Time: 26:28)

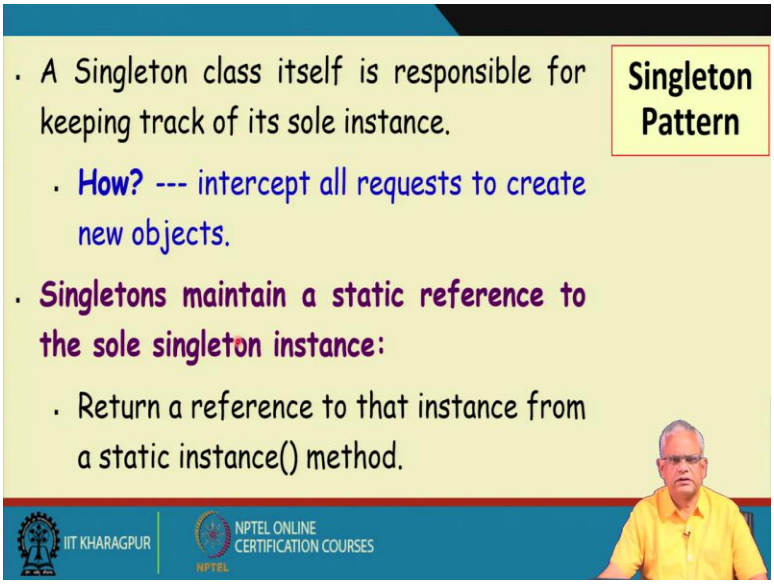


The main idea of the singleton pattern is that all applications call the singleton, the getInstance. The getInstance method of the singleton is called by all other objects. When the first time request

is made using the getInstance, if the singleton has not been created, then it will create the response, the necessary object and return the ID.

But once the object has been created subsequent calls to get instance will only send the ID of the object that has been created and therefore we can say that the getInstance is the only way to create an object and here the getInstance checks whether an object has been created and if it is not created, it creates it and if it is already created, returns the ID only.

(Refer Slide Time: 27:28)



That is the main idea here, that the getInstance method intercepts all requests to create new objects that means, the constructor for the singleton object must be private and the only way somebody can create is by using the getInstance method which intercepts all requests to create the object and it checks if the object has been created, if not it creates, otherwise it just sends the ID of the object that has been created.

There is a static reference to the soul instance and as a request to the get instance method comes then it returns this reference. But then the question that would like to ask you is that why a static reference to the sole singleton instance. We will look at the code, understand the working of the singleton application use it for couple of applications.

And but then please try to answer this as you look at the code or otherwise that why is needed for the singleton object, that is created a static reference to the singleton object. We are almost at the

of this session we stop here and continue our discussion on the singleton pattern in the next session. Thank you.