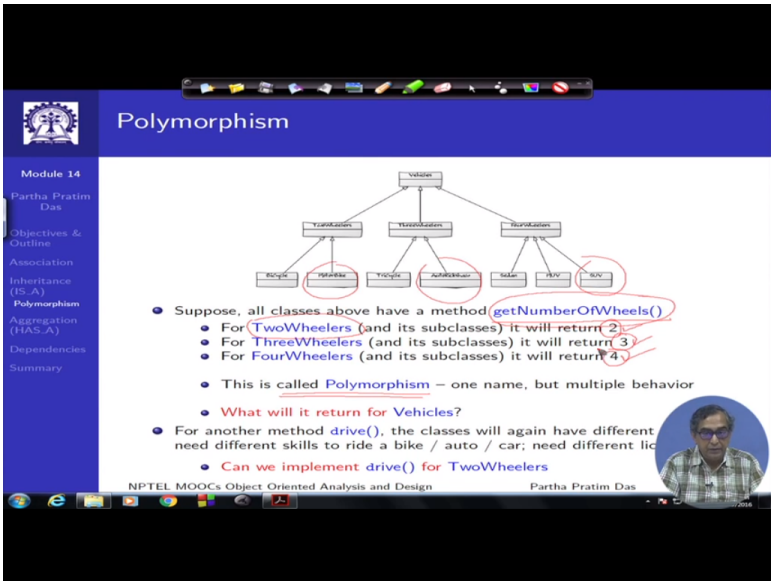
Object-Oriented Analysis and Design
Prof. Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology-Kharagpur

Lecture – 25
Relationships among objects (Contd.)

Welcome back to module 14 of object oriented analysis and design. We have been discussing about relationships among classes and we have talked about association and we have revisited the concept of inheritance relationship in connection to what was earlier done as inheritance amongst objects.

(Refer Slide Time: 00:46)



Now we continue on the most important concept on inheritance which is polymorphism so let me introduce that with a small example. So we again take the inheritance hierarchy of vehicles that that we have introduced earlier. Now let us say that suppose for since it is a hierarchy of vehicles we would like to know for any class or object of any class as to how many wheels does that vehicle have. So we will have a method called get number of wheels.

Now what is the expectation that for 2 wheeler class or for any of the subclasses of the two-wheeler class, this method should return 2, for 3wheeler it should return 3, for 4 wheeler it should return 4 and also for the respective subclasses. This property is called polymorphism. What is the property? That for all of these, the name of the method is same; the name of the method is get number of wheels. But depending on which class or object of which class whether it is two-wheeler or its motorbike.

Or its auto rickshaw or it is suv depending on which object or object of which class in invoked the

actual methods are different. The algorithm executed by the method or the implementation of the method is different so we say that polymorphism very simply is one name but multiple behavior, one name is the one name and these are the three different behaviors that we could associate with it. So this is the basic notion of polymorphism and we will see that this will come in extremely handy

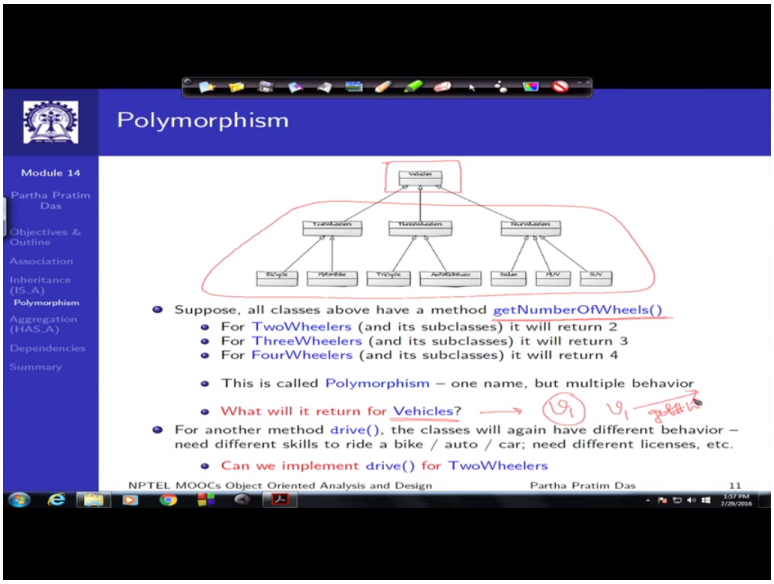
In terms of doing different object based design based on different hierarchies. Okay now let us say that suppose I take this question further and ask that for this particular method if I invoke this on waiting the root class what will it return? What will be the expected output of get number of wheels if asked for some instance of vehicles? You do not know. We cannot say because a vehicle could be a two wheeler when it should be a 2 or it could be a three-wheeler when it should be 3, it could a four-wheeler when it should be four and so on.

In plain and simple terms, this question cannot be answered. Or in plain and simple terms, this particular method cannot be invoked on the waiting. So that gives us a very interesting scenario, I have a hierarchy where a large number of classes are involved, many of these classes are lift classes and they will have several object instances, all of them can respond to the message get number of wheels while the root class from where everybody specializes defines a basic concept of vehicles that cannot answer this question that cannot handle this message.

So this leads to the requirement that when such a situation can exist that over this generalization specialization hierarchy, it is possible that some of these polymorphic methods are not implementable in some of the classes then those classes are considered to be abstract. So that is another way of or more correct way of looking at when a class should be abstract. A class should be abstract when there exists at least one polymorphic method which can be implemented for some of the subclasses but cannot be implemented for this class.

Because if you if vehicle was not abstract then I will I can have an object v1 a concrete object v1 of the vehicle class type.

(Refer Slide Time: 05:35)

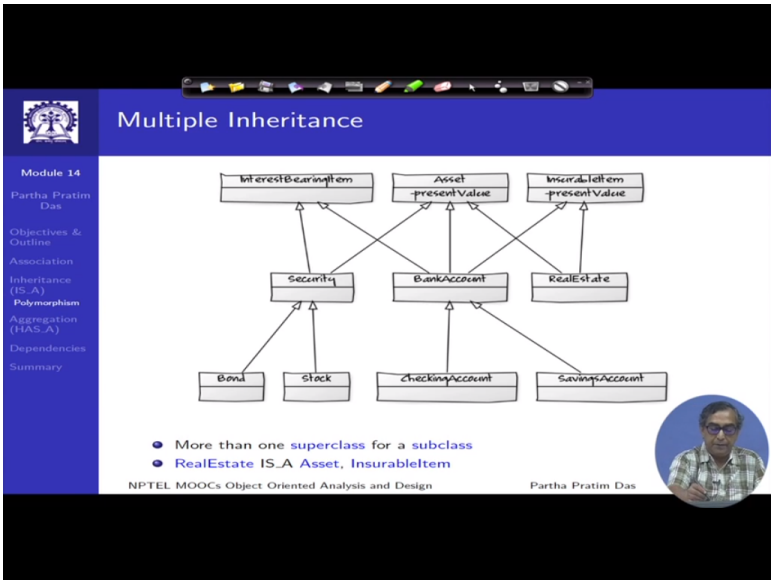


If I have v1 then I can send a message get number of wheels to this vehicle and we know that that message cannot be executed because there is no algorithm for it. So this is the basic fundamental notion of abstract classes and how polymorphism and abstract classes relate to each other you can just think further, you will find that there are you can find reasons of why two wheeler or three wheeler classes will also have to be abstract.

Because say certainly all the vehicles will have some method drive now if you are say that I the class is vehicle you just do not know how to drive because there are different bicycle needs certain kind of skills and auto rickshaw needs a different kind of skills, sedan needs yet different kind of skills, you need different types of licenses, all of different factors are different between driving a vehicle, driving a two-wheeler and driving these specific instance specific vehicles of different number of wheels.

So you will again similarly not be able to implement drive for a two-wheeler because somebody anybody can drive a bicycle if he or she knows balancing but those all of those may not be able to drive a motor bike unless he or she has a license to drive a motor bike. So drive in two-wheeler does not have an implementation whereas it has implementation here. That consequently makes two-wheelers as potentially abstract class.

(Refer Slide Time: 07:10)

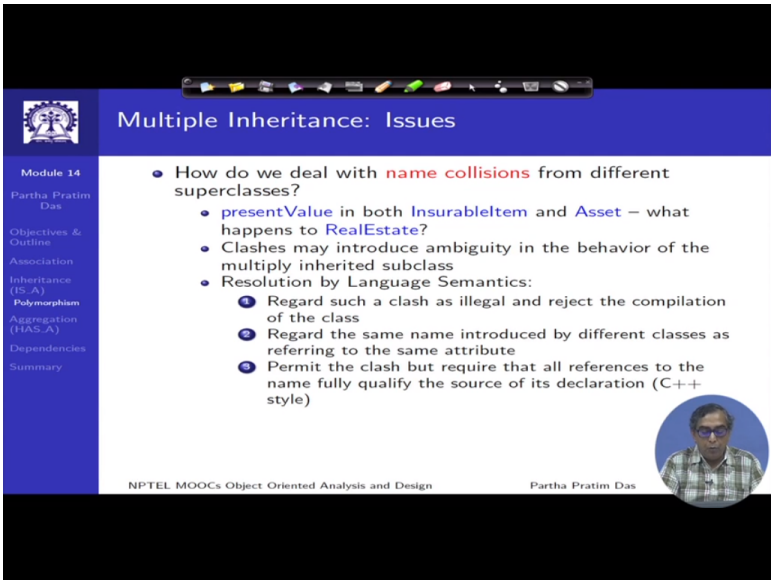


Moving on in terms of inheritance there is a special type of inheritance know as multiple inheritance where more than one super class exist for a subclass. So for example if you look into the bank account here, bank account is an insurable item, is an asset, is an interest-bearing item. Okay you can think through these kinds of examples this example is from Booch's book, so you can read up the whole details of the example from the book.

It clearly says that if I have a bank account I have money in that then naturally I can ensure that money against that. I can use that as an asset to get collaterals and certainly the bank regularly gives me interest on that. So there is a specialization of bank account which has certain properties borrowed from the interest bearing item because it can earn interest certain properties borrowed from the asset class because it can be treated as an asset and some more properties is borrowed from the insurable item.

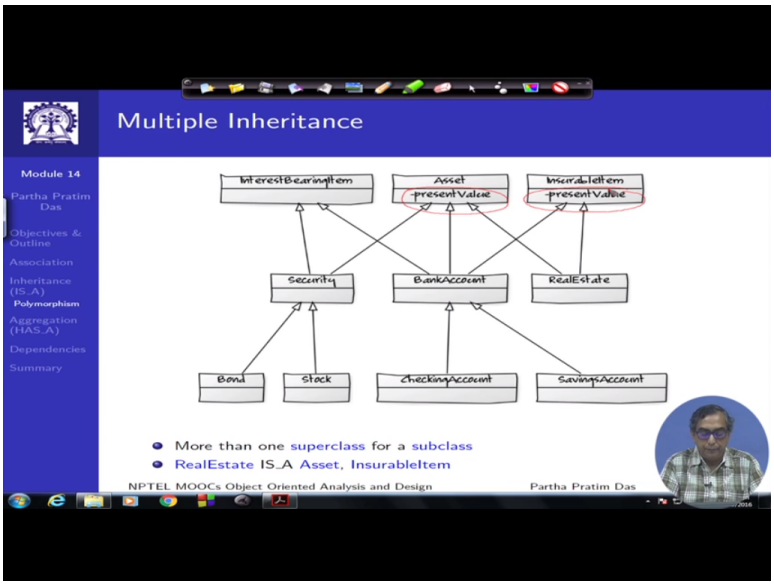
Because it satisfies all the requirements of an item which can be insured. In addition the bank account will have other further other properties if you look at real estate, real estate is an asset at the same time, it is an insurable item as well. So whenever such situations exist whenever one subclass have more than one super class we say that we have a situation for multiple inheritance. Multiple inheritance occur very frequently in real life and but it has some very fundamental difficulties to be handled

(Refer Slide Time: 09:00)



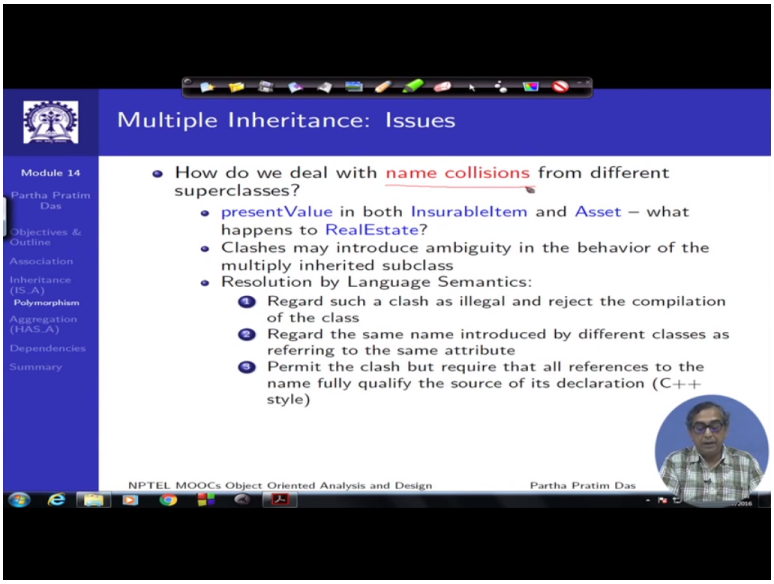
And modeled and therefore we will just mention about those. One is the issue of name collision

(Refer Slide Time: 09:12)



So if you look into this say the asset class and the insurable item class let us say these are independent classes so these are quite possible that they have a variable a data member which is common between them. So the present value so what happens when I inherit from this I inherit from this in the bank account or in the real estate I actually get 2 present value members. The question naturally is which present value member I should be used.

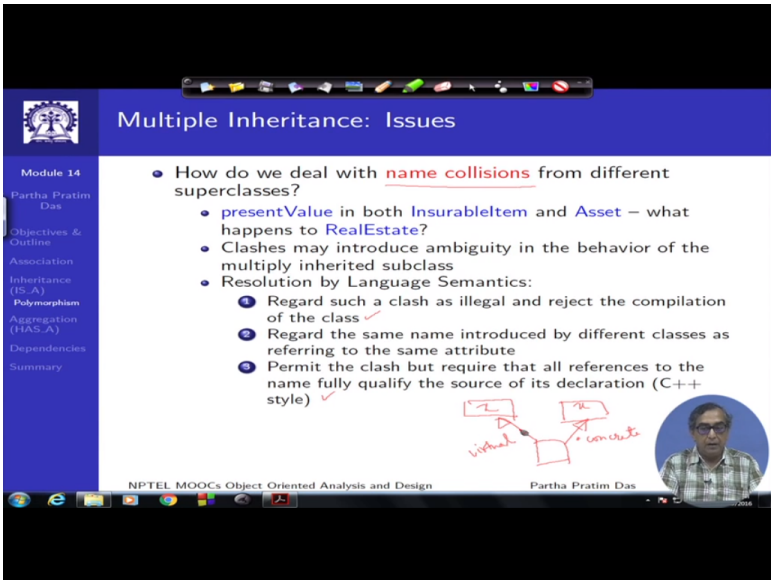
(Refer Slide Time: 09:40)



So this is what is known as a name collision. Present value in both insurable item and asset, what happens to the real estate? So this will certainly create ambiguity or clash in the behavior of the multiply inherited subclass there is no unique answer to how should this be handled, different languages depending on the language you are using, they have different semantics to handle name collisions. Some will not even not allow this to be used at all.

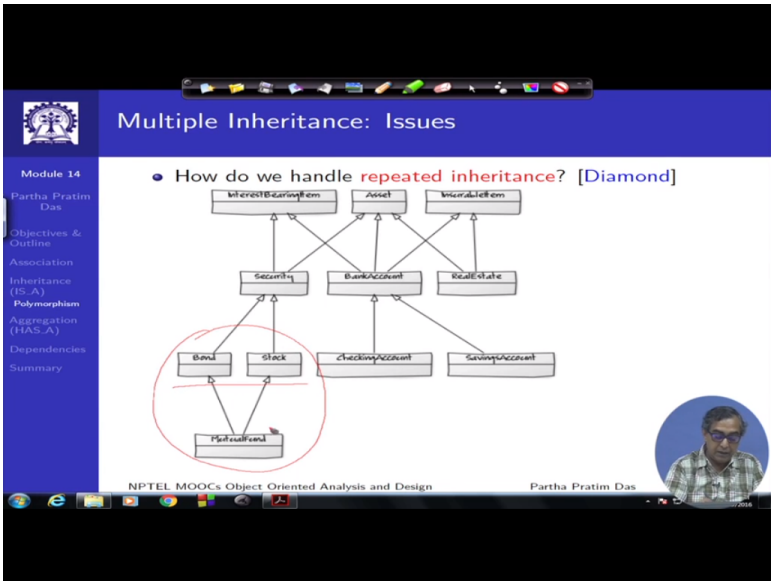
That is 2classes if they are parent of the same class, then they cannot have same data members is one possibility. One possibility is they will treat them as the same data member and then define some semantics based on that or there are some finer semantics for example c define C++ defines a semantics where it say that if I have multiple inheritance like this a situation of this kind and the same variable exists on both, then I will not inherit from both of them. One of them can be concrete.

(Refer Slide Time: 10:43)



Rest of them should be what C++ say should be virtual which means I will only use their methods but I will not use their data members if you are not familiar with C++, do not worry about these comments because the idea is not to go to a specific language, the idea is just to highlight that these are some of the very core issues that can happen if you are using multiple inheritance but at the same time multiple inheritances and extremely useful modeling feature.

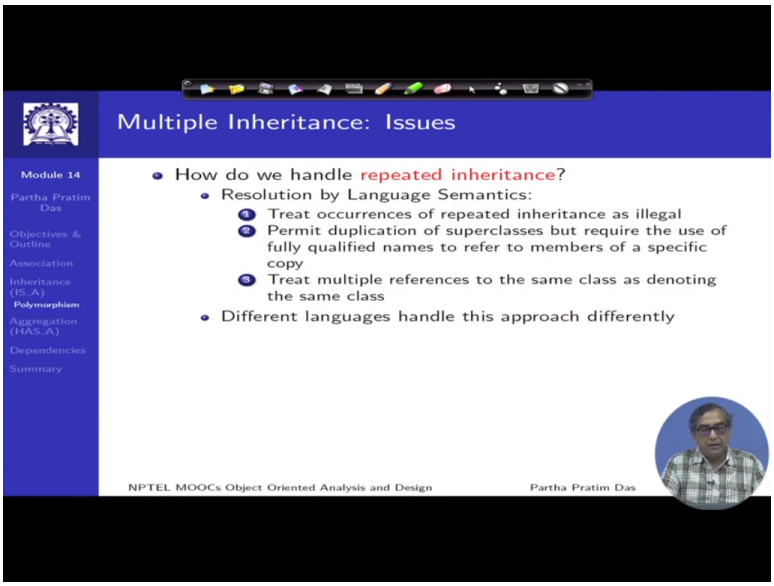
(Refer Slide Time: 11:26)



Moving on the second issue which will often be seen just look at this part this part is what I have added now. So earlier it was bonds and stocks or securities I have added mutual funds as a multiply inherited class from bond as well as stock. So what do we have? You have here, if we just look into this a diamond shaped like a diamond which means that this class this security class is actually inherited by multiple mutual fund in two different parts.

It is inherited as mutual fund is a bond, bond is a security in that part or mutual fund is a stock and stock is a security in that part naturally if you do it both on both these parts, then the kind of properties that get associated with them differ and the behavior will differ and you may have often ambiguous contradictory situations happening.

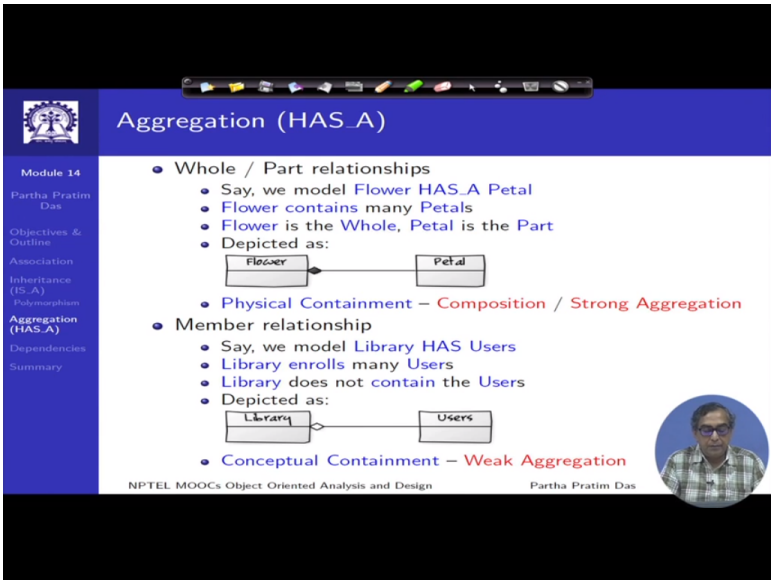
(Refer Slide Time: 12:31)



Again there is no unique solution to this repeated inheritance problem again the different languages when you go to implement such situation will have different restrictions of doing that, some will simply not allow that, some will only allow one parent but you will have to decide on which part the parent will come and so on. So please remain informed that multiple inheritance as such is a very strong modeling relationship amongst classes.

But you will have to be careful about the name class and the repeated inheritance when you are doing your design and as far as possible you should try to avoid both of these in your design.

(Refer Slide Time: 13:21)



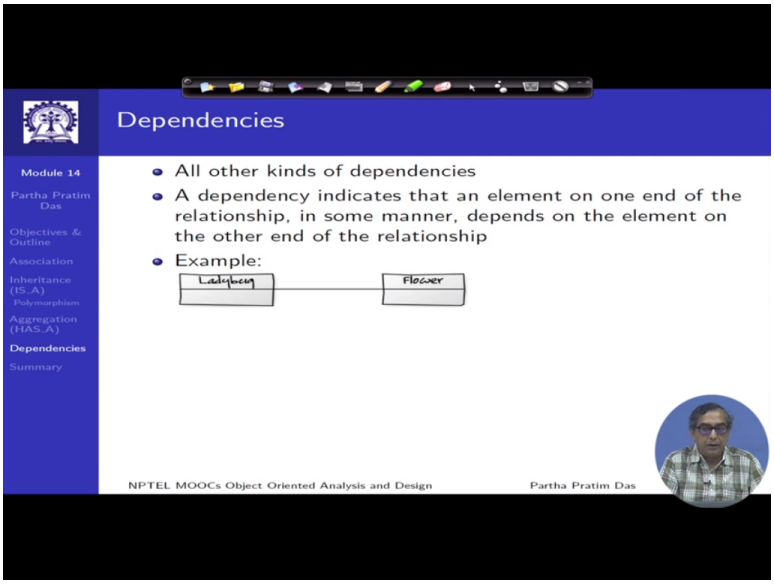
Let us move on to the relationship which is aggregation which we know is has a kind of relationship and I will be quick on this because we have explained this in depth in terms of relationships amongst objects so it is primarily a whole part relationship. So we say flower has a petal so that means that flower contains many petals and we say the flower is the whole and certainly petal is a part.

And in terms of the depiction in diagrammatic depiction we will show it like this and in this case whole relationship, in this case there is a physical containment, the petal is physically contained in the flower. So often it is referred to actually as composition and some authors call it as a strong aggregation also. In addition we have seen that there is a different kind of aggregation which is can be referred to as member relationship where library has its users. Now naturally a library will have many of its users.

And without that users, the library has no purpose but at the same time, the library enrolls the users but library do not contain them, the users are not physically contained within the library, we talked about this earlier in the context of objects also naturally in terms of classes as well. This will have to be kept in mind and this is how we diagrammatically depict that and since this confinement is kind of conceptual there is no physical containment. We call this as a week aggregation.

In terms of the diagrammatic depiction of weak aggregation and strong aggregation I would like to just draw your attention to the fact that in case of weak aggregation the diamond that we use at this end is empty one in case of strong aggregation the diamond you use is a filled up one. So that is how you will be able to depict different aggregations quite easily.

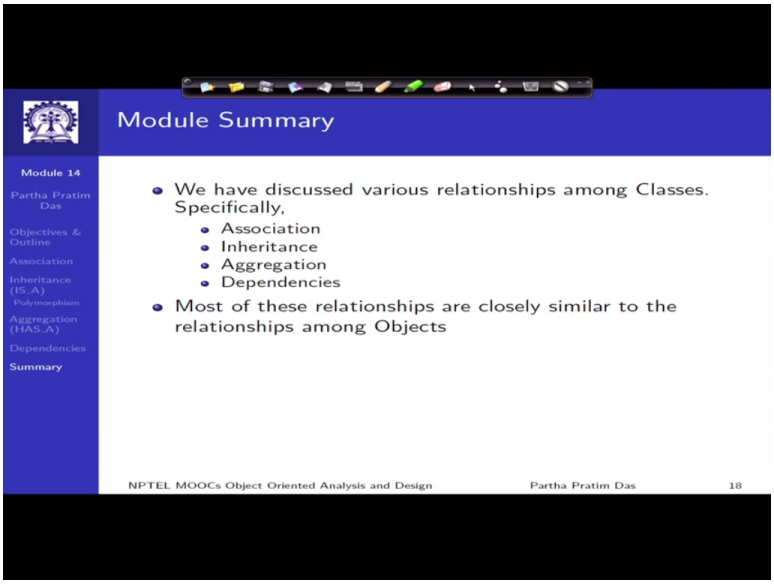
(Refer Slide Time: 15:32)



Finally so beyond association, inheritance and aggregation which of the three main forms of relationship the classes can have. Then there could be all other kinds of dependencies anywhere when you find that some element of a class is influenced by is related to or depends in some way with another class. Then you should mark that there is a dependency so when we talk about ladybug and flower, there is a dependency relationship.

There is something semantically dependent in terms of protecting ladybugs, protecting the flower but these could be there could be lot of variations on these dependencies and we will see as we move further into particularly the UML diagrams and modeling of variety of situations, we will see that a large number of finer relationships can be clubbed are clubbed together in terms of a general dependency relationship but till you go to that depth you should always keep in mind that anything that comes across where you say that things can be somewhat related you put in dependency and put it in the diet.

(Refer Slide Time: 16:49)



So in this module we have discussed various relationships amongst classes, specifically we have talked about association, we have talked about the inheritance which is the most important relationship amongst classes the hierarchy, particularly hierarchical as well as multiple inheritance and in the multiple inheritance, we have highlighted the issues that can commonly happen in terms of the name clashes and in terms of repeated inheritance, we have then discussed about the aggregation composition and weak aggregation relationships amongst classes.

So most of these relationships I will again repeat, reiterate are pretty much like the relationships, we saw amongst the objects only difference being that when we talk about objects, we are talking about concrete entities that exist and when you are talking about classes we are basically talking about abstract plans based on which objects will be created at the runtime so like objects have a lifetime which is very dynamic.

Classes will also have lifetime but they are more static they will exist from one version of the design till the next version of the design and within the lifetime of one class, several objects will be created, operated and destroyed.