

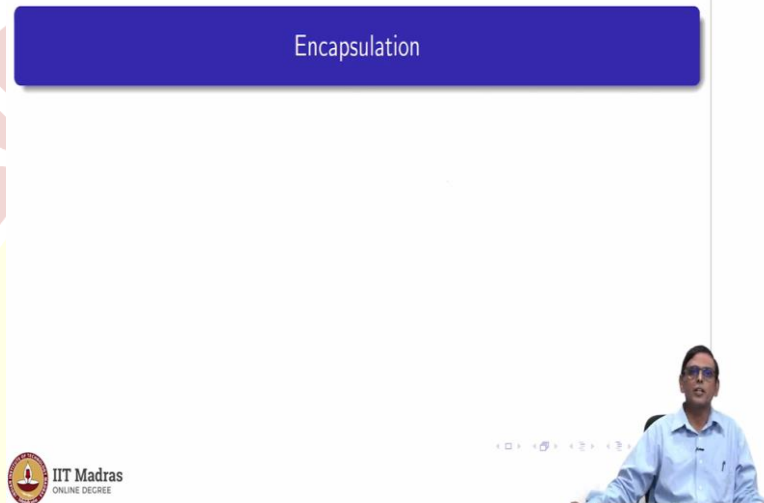


IIT Madras

ONLINE DEGREE

Computational Thinking
Professor Madhavan Mukund
Department of Computer Science
Chennai Mathematical Institute
Formalized Notations and Summary of Encapsulation

(Refer Slide Time: 00:14)



All right, so this week we saw some content on Encapsulation. The, there is a little difference in the way we have treated the material this week from what we have seen the previous weeks. So far, our focus has been on trying to understand how to write code for various kinds of problems and to organize that code using a sequence of steps and using data structures like lists, and dictionaries, and graphs, and so on.

In this week, basically, we are trying to understand how to put all this code together in a way that makes the code a little bit more easy to manage and makes it more reliable in terms of preventing mistakes in the, writing of the code, and so on. And the key concept that you have over here is this concept called encapsulation.

(Refer Slide Time: 01:01)

What is encapsulation?

- Procedures that we have seen so far have been unanchored.
 - It seems more natural to attach the procedure to the data elements on which it operates
- **Encapsulate** the data elements and the procedures into one package
 - which is called an **object**, hence the popular term **object oriented** computing/programming
- The procedures encapsulated in the object act as the interfaces through which the external world can interact with the object
- The object can hide details of the implementation from the external world
 - The changed implementation may involve additional (intermediate) data elements and additional procedures
 - Any changes made to the implementation does not impact the external world, since the procedure interface is not changed
- Allows for separation of concerns - separate the "what?" from the "how?"



So in encapsulation, basically, what we have understood is that, typically, traditionally, when we look at procedures, in real life, these procedures do not mean anything in the sense that they do not necessarily, you do not necessarily find procedures hanging around somewhere. Usually, the procedures are attached to the objects on which or the elements, the data elements on which the procedures operate.

We do not see procedures lying around here and there just like we see objects. So, for example, you might get instructions for managing a television or something like that, but that instruction will come packaged in the box in which the television comes. So that is the example we saw. So it is more natural in some sense to attach procedures to the data elements on which the procedure operates.

So we introduced this concept called encapsulation in which, what happens basically in encapsulation is that we package the data elements and the procedures which are used to operate on those data elements together into what are called objects. And which is why there is this very popular term called object oriented computing or object oriented programming that you might have seen in the literature.

So in object oriented programming, what you try to do is bring together the data element and the procedures that act on them together into one, one encapsulation, one package. So once you encapsulate data elements and the procedures that act on them, the expectation basically is that

the external world will interact with this object, this encapsulated package only via the procedures that are embedded within this object, and we should not have access directly in some sense to the data elements that are there in the object. So that is basically the expectation.

Now, so we need to basically somehow put everything together and make sure that the external world can see only certain things, they can only see the procedures as the interfaces and are not able to see the data elements. So you should have a mode of hiding is required.

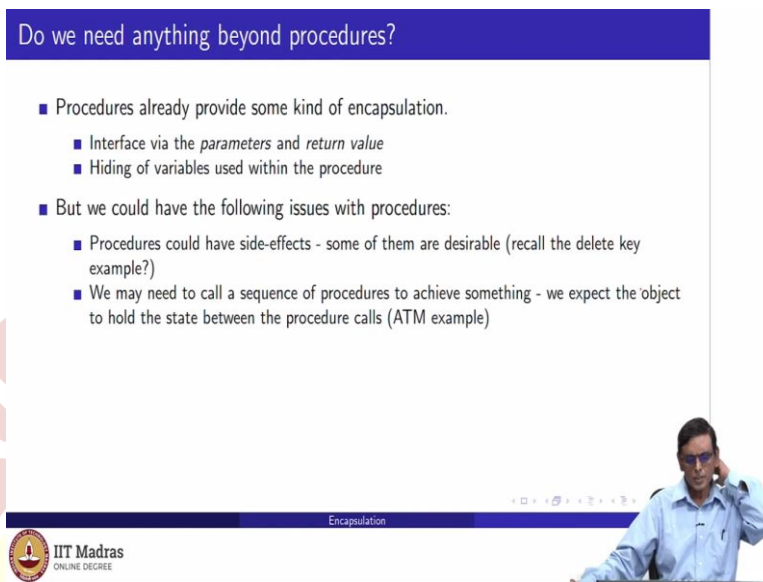
So the big advantage of hiding the implementation from the external world is that you can manage the code much better. For example, the person who is using the code now does not need to worry about how the code is implemented and the person who is implementing the code has the freedom to change the code if they want to improve its efficiency.

For example, add some intermediate data structures, add additional features, maybe add an additional parameter or do whatever they want to do and these changes that they are making internally to their code does not in any way affect the external world.

So this way, you are separating out the problem of understanding what is being implemented, what is being implemented by the code from how the code is implemented. So the person who is implementing the procedure inside the object is more interested in how the code is implemented, whereas the person who is using that procedure from outside, the procedure of the object from outside is only interested in some sense in the what; what is it doing for me.

So you have separated the what from the how and so the implementation can proceed independent of some of the use of that particular procedure. That is basically the advantage you have of encapsulation.

(Refer Slide Time: 04:24)



Do we need anything beyond procedures?

- Procedures already provide some kind of encapsulation.
 - Interface via the *parameters* and *return value*
 - Hiding of variables used within the procedure
- But we could have the following issues with procedures:
 - Procedures could have side-effects - some of them are desirable (recall the delete key example?)
 - We may need to call a sequence of procedures to achieve something - we expect the object to hold the state between the procedure calls (ATM example)

Encapsulation

IIT Madras
ONLINE DEGREE

Now, one could ask, I mean, this thought must have occurred to you, but already we are doing encapsulation through procedures, so what is new? Procedures have parameters, which are the interfaces, you talk to the procedure via parameters, you send it data, the procedure can return a value, so it gives you a result back.

So you are sending something into the procedure, the procedure is computing something and returning its value, and internal to the procedure, the procedure can have some variables which you cannot see; from outside you cannot see the variables that are sitting inside the procedure. So it is hiding those variables from you. And the person who is implementing the procedure has the freedom to implement procedure any way they want and they can change the implementation of the procedure and this does not affect the call of the procedure in any way.

So you can ask, if I have procedures, which are parameterized, and you have basically, you have the ability to hide things. And so on inside a procedure and you are able to pass parameters and get it on values, then what do we need anything beyond procedures at all; that can be a question that might have occurred to you.

And the answer really is that there are some issues that procedures face. One of the issues is that procedures, you have already seen have side effects which means, besides basically taking parameters, computing, and returning a value, the procedure might also make some changes to the parameters, for example, that are passed to it, or maybe even make changes to variables

which are not passed to it. So there may be some objects lying around in the environment to which the procedure is access and this procedure might make changes to those objects.

So we have seen, for example, the example of the delete key; the dictionary, you pass the dictionary and the key you want to delete to this procedure, the procedure basically deletes this key and makes a new dictionary out of it without the key and so it has it has modified the dictionary.

So there could be side effects that procedures have and very often, dealing with these side effects becomes very complicated and you need to know, person who is using the procedure needs to know what kind of side effects the procedure will have, so that it can be managed. This is one of the issues.

The second issue is that we may need to retain state between multiple procedure calls. So an example that we took in our discussion in the class was basically that of an ATM? So you have multiple interfaces. So the ATM might say insert your card, so you put your card inside, then it will ask you to authenticate, your pin or something like that and once your ATM card is authenticated, you can do a number of transactions while you are in the authenticated state.

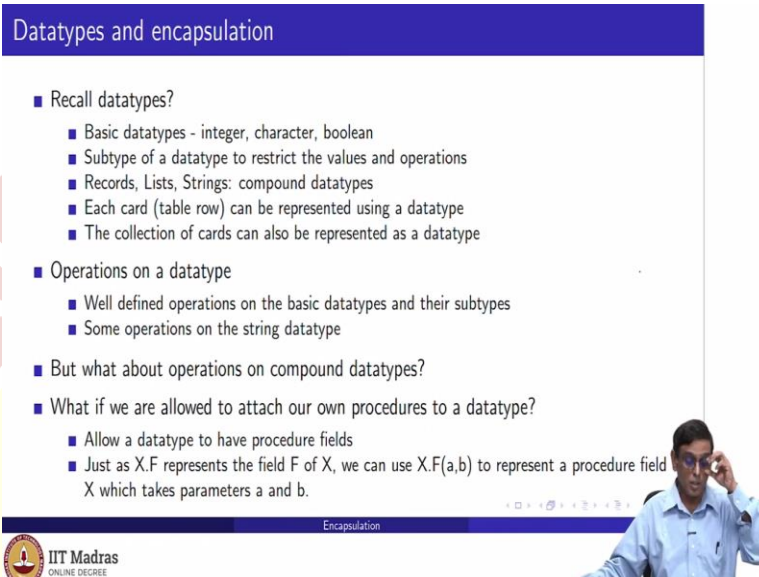
So in some sense, the interface of inserting your card and authenticating it, that interface is one procedure and once that procedure is over, the ATM is now in an authenticated state or a secure state and in that state, you can ask for your balance, withdraw some money, do those things. And then after that, you can exit that authenticated state and then you can take out your card.

So the, the between the procedure of authentication and the procedure of accessing your account, there is a state; that state, basically, is that the ATM is now in an authenticated state. So this state, sometimes many times you will find basically that you may have to make a sequence of procedure calls and the state of the system has to be retained between these sequence of calls and we will take a lot more detailed example in today's class to understand that.

So this is another reason why you might want to encapsulate because the state that is there between procedure calls, you may want to retain in a very nice way and that nice way is this object within which this procedure exists. So the object retains the state and you can call one

procedure after another to this object; the object will remember what has been called and it will keep the state.

(Refer Slide Time: 08:02)



Datatypes and encapsulation

- Recall datatypes?
 - Basic datatypes - integer, character, boolean
 - Subtype of a datatype to restrict the values and operations
 - Records, Lists, Strings: compound datatypes
 - Each card (table row) can be represented using a datatype
 - The collection of cards can also be represented as a datatype
- Operations on a datatype
 - Well defined operations on the basic datatypes and their subtypes
 - Some operations on the string datatype
- But what about operations on compound datatypes?
- What if we are allowed to attach our own procedures to a datatype?
 - Allow a datatype to have procedure fields
 - Just as $X.F$ represents the field F of X , we can use $X.F(a,b)$ to represent a procedure field X which takes parameters a and b .

Encapsulation

IIT Madras
ONLINE DEGREE

Now, let us just go back. Many classes ago, we introduced this idea called data types. After that, we have not talked much about it. Of course, you have been seeing these data types through the course. You have been seeing the use of list, strings, Boolean values, integers, and so on through the course; you have been seeing it.

But the way to organize these data types as basic data types and then you have this notion of subtypes, which basically allow you to constrain the values and operations on the data type. And then, we said, we can make more compound data types using the idea of record. So all the cards that we had in our class, the rows of a table, are all records and you can access the records field by doing $X \text{ dot } F$.

And the idea of making a list of elements, which are typically all of the same data type or you can make a record which basically, whose fields are all of different data types. So these were the basic kinds of data types that we constructed. And we saw that there were some basic operations that each of the data types allowed; like for Boolean, you can do an AND of a Boolean with another Boolean, an OR of a Boolean with another Boolean, so on.

This does not make sense to do an AND of integers but you can sum integers, you can multiply integers, like that. So every basic data type allowed you to do some operations on them. However, and even for lists and so on, we basically had some operations, like first and init and things like that, or append the two lists together, and so on based on some operations.

So what if we allow the user to write more complex operations, procedures in some sense, on compound data types, what about that? In which case what happens, basically, is that you take these data types, which are compound data types, which have maybe, let us say it is a record, it has some fields.

Now, I want to attach to this data type also a procedure which will act on the fields of this record. So, for example, I can have a new field, which is called F, which instead of being just a value of some type is actually a procedure now. So I have a procedural field.

So I add a procedural field to this record, and then, I can then call X dot F and pass it parameters; like say, a and b as parameters. And X dot F a, b will then return a value back to me. So if I extend the idea of a data type to include procedures; initially, I had fields of various types, now, I also have procedures, then, basically, what we get is this idea of an object.

So this encapsulation is nothing but taking what we understood as data types earlier, which had an object with various fields but we are now adding procedure fields to it. So that is basically what an object is.

(Refer Slide Time: 10:43)

Example: Classroom Scores dataset

- Questions most frequently asked of the Scores dataset
 - What is the average in a subject - say Maths?
 - What is the overall average?
- Suppose that we have to ask these questions many times
 - It is wasteful to carry out the same computation again and again
 - Can we not store the answer of the question, and just return the saved answer when the question is asked again ?
- This will work as long as the dataset is static.



Encapsulation



Let us take a quick example of this, which we saw in our classroom's course dataset, and ask ourselves, whether or not there is an interesting encapsulation we can do of this dataset. So the questions that are most frequently asked of the scores dataset is, for example, we could ask, what is, we have been asking these questions and then we have been answering them also.

So what is the average overall, for example, you can ask. Or you could ask, what is the average in a particular subject, what is the average in physics for the class, entire class? And then we could ask, based on that we could ask other questions, who got, which student got more than average, or if I give you six students, you can ask is this student much higher than average or much lower than average or what, and so on.

But this, this was a finding the average we saw was being used quite heavily and we know that the average can be very easily computed, just take all the marks, total marks; let us say you take all the total marks of each student and add them up, and then divided by the number of students and you get the average. So very easy to compute.

But if you are keeping on asking this question again and again, question that comes in is why do we have to keep computing it again, because each time you compute it, you have to do a sum of all elements, then you have to divide it by the count, which is the number of elements, maybe a number of elements, you can keep a record of it, so it is easier.

But, at least the sum of all the total marks you are keep doing again and again and that is costly operation because it depends on how many elements are there in the list. If there are a large number of students, this sum will basically take that much time.

So the question we can ask is cannot we just store the answer of the question, which is average in this case and next time, when we asked this question, we just returned the answer rather than computing it again. This is the method that is used very extensively in computing and it is called caching.

So when you access a webpage, for example, next time you access a webpage, it may be much faster because that page actually stored in your local computer and it is retrieved from your local computer, it is called caching. So the idea here is that we want to store or cache the result and return the result from the cache rather than having to compute it again. So that is much faster.

Now, obviously, this will work only if the dataset that we are using is static. Static means it does not keeping on changing. Now, luckily, for us, if we take a classroom's course dataset, the chances are it will be static because usually there is two classes made at the beginning of the year, and through the year, the class is not going to change; it is a static dataset.

But occasionally, you may have one student, new student may join the class sometime middle of the year and this class might change but at least for the period, when it is static for a few months, it may be static. Every time an average is called, we do not need to re compute it.

But we have to keep in mind that maybe this dataset is not static. So we have to keep that in mind. But for the moment, let us assume that the dataset is static and ask ourselves if we have to store the result of the computation, how do we do it and does encapsulation help?

(Refer Slide Time: 13:36)

Example: encapsulation

- We could create an object CT (for class teacher) of datatype ClassAve
 - ClassAve needs only the list of total marks of the students
- ClassAve can have a field marksList that holds the total marks of all the students in the classroom dataset.
- We can now add a procedure average() to ClassAve to find the average of the list
- Since we want to store the answer after the first time, we could have another field aValue that will hold the computed value of average. Initially aValue = -1.
 - CT.average() first checks if CT.aValue is -1.
 - If no, it just returns CT.aValue
 - If it is -1, this means that the average has not been computed yet. So, it computes the average by summing the marks in the list and dividing the sum by the length of the list



Encapsulation



So what we want to do really is we want to create an object. Let us say CT; CT stands for class teacher because we in our in our discussion, we basically talked about class teachers, physics teachers, and so on and these class teachers are the objects who basically do work for us. In some sense, who give us results; you ask questions to a class teacher and class teacher gives you an answer.

So we are expecting that we can ask this class teacher for the average of the class, the class teacher is able to return it. Now, class teacher, assume, is object of data type called class average. We are trying to define this data type called class average. And so, we have to define its fields, we have to define now, we have added a new thing called procedures, we have to define the procedures also of class average.

So first thing we observe is that class average really does not need all the data of the class. It does not need all the cards of the, the all the data, the name of the students and the age, date of birth, and other things it does not need. It just needs, because to compute the average, all we need is the total marks. So if you add the total marks, then of all the each student, then you can compute the average and that is all you need for this particular example.

Of course, if you have another example, you might need some other data structure but we have learned that we can collect whatever data structure we want and put it together in the form of list or dictionary or something like that.

So we can process the cards, collect all this data, and store it and then write these procedures. In this case, what we have done, basically, is processed it and kept only the total marks. We have kept a list of the total marks. So it needs only the list of total marks.

So let us say that the data type class average has a field which is called marks list and this marked list holds the total marks of all the students in the classroom dataset, which is enough for us for doing average.

Now, what we do is we add a procedure called average to this class average. So far, our records or basically our data types did not allow procedure, but we are now extending it to allow procedures. So we are adding a procedure called average to class average data type and average is supposed to find the average of the list.

Now, since we want to store the answer after we computed first time and the next time, we want to return the stored answer, let us assume that we have a field. Class average is a data type, so we have a field called aValue that class average has, the field that data type has. And this field, aValue, will hold the result of the computation.

Of course, you can ask what happens the beginning? When the object is created, there is nothing has been computed so far. So initially, let us initialize aValue to minus 1. So minus 1 is a number, just like we saw for min and max when we did, and so on. We have to choose a number which basically can never occur as a possible average value; so minus 1.

Average, we know all the marks are 0; at least 0 or higher. So the average is also going to be 0 or higher, so it can never be minus 1, negative number. So we choose aValue equal to minus 1 so that by looking at aValue, we will know whether or not the average has been computed or not. So if aValue is equal to minus 1 it means that the average has not been computed.

So how does CT dot average work, the procedure work? CT dot average will first check if CT dot aValue is minus 1 or not. So we know that minus 1 basically means that average has not been computed. So if it has, if it is minus 1, then it has to compute the average.

So it will basically go and compute the average. But if it is not minus 1, which means it is somewhat positive value let us say, then it means that the average has already been computed. So it just needs to return CT dot aValue.

So CT dot average will first check the value of CT dot aValue. If CT dot aValue is minus 1, then it computes the average by, how does it compute average? It will add up all the total marks in the list and divided by the length of the list. So that is average. But if it has already computed, then CT dot aValue is a number which is, value which is not minus 1, and so then it can basically return it.

So what is important basically is that after we compute the average, we store that average in aValue, so that the aValue is no longer minus 1. So this makes the procedure extremely fast second time. First time, when you call it, it will actually compute the rest, it will take some time, but subsequent calls, it just returns aValue. So it is extremely fast; so it is caching. So it is doing it very fast. So this is the, this is a great advantage we have.

No, but of course, the key thing you have done here is you have stored the result in a field called aValue. See, if this result were stored inside the procedure, then when the procedure returns, you do not have access to that field anymore. So it cannot be done in some sense by keeping the value inside a procedures; it has to be kept outside the procedures.

You ask if it has to be kept outside the procedure, where you keep it. Now, in the case of an object there is a natural place to keep it because you can create a field which is also inside of it. Procedures inside the object, aValue is also inside the object, so there is no problem in keeping it in a field which is outside the procedure; that is basically the advantage.

(Refer Slide Time: 18:31)

Example: encapsulation

- What about the average values of the individual subjects?
- Just like CT, we could have objects PhT, MaT and ChT (for Physics, Maths and Chemistry Teachers) that each hold (or have access to) the entire classroom dataset.
 - But again as we observed with CT, PhT needs to hold only the list of Physics marks of the students, similarly MaT and ChT need only hold the Maths and Chemistry marks lists.
- So, let us say that each of these objects are also of the same datatype ClassAve, but their marksList field holds the list of marks of the respective subject.
- Is this enough? What happens when we call PhT.average()?
 - PhT.average() checks if PhT.aValue is -1.
 - If not, it just returns aValue
 - Otherwise, it computes the average from the marks in the marksList - which is just the average of the Physics marks !
- So the same datatype ClassAve can be used for all the objects, CT, PhT, MaT and ChT



Encapsulation



Now, suppose now we can ask, suppose I want to find the average value of individual subjects, what do I do? The answer is just like CT, we could create new objects called physics teacher, math teacher and chemistry teacher. And each of these, they can hold the entire classroom dataset if they want with all the fields.

But again, we see it is not necessary. To compute the average of physics marks, all you need is the physics marks; you do not need anything else. And similarly, to find the chemistry on average, you only need the chemistry marks. So math needs only math marks.

So let us keep inside the physics teacher, PhT object, let us keep only the physics marks list and chemistry teacher object keep only the chemistry marks list, and similarly, maths, we will keep only the maths mark list.

So whether it is the class teacher or the physics teacher or it is a chemistry or the math teacher, any of these teachers, there is only one marks list they will be maintaining. So in some sense, the same data structure class average can be used for this. Because class average, we know has a field which has the marks list data, data field.

So we can use that same class average data type for this; but for storing at least for mark list. But what about the procedure? We saw that the procedure, the class average has a procedure called

average. Let us see whether we can apply this procedure called average for physics teacher and see what it returns.

Remember what average did for class teacher. It is not actually class teachers average procedures, class average is the data types average procedure and class teacher happens to be an object of that class average data type. So, therefore, class teacher also had access to average in some sense.

So since physics teacher is also an object of type class average, so physics teacher also has access to the average procedure. Now, what does physics teacher dot average procedure do? It will check whether, now, key thing here, important thing. It will not check class teacher's aValue, it will check its own aValue; physics teacher's average checks its own aValue.

So physics teacher's average checks its own aValue, which is physics teacher dot aValue and see whether it is minus 1. If it is, then basically it means that it needs to compute the values, it will compute it. Otherwise, it just returns aValue, so it works.

So we are seeing basically that not only does average, the procedure we wrote in average for actually class teacher, not only does it work for class teacher but it seems to work also for physics teacher and if it works for physics teacher, I presume it will also work for chemistry teacher, it will also work for math teacher.

So the same data type class average can now be used for all the objects; class teacher, physics teacher, math teacher, and chemistry teacher. So it makes the code really neat now. Just like a procedure, basically, we thought about a procedure as something that can be called many times with different parameters and it would do different things for us.

Now, we seem to have something called a class average data type, which we can keep reusing to make different kinds of objects. We can make class teacher object, physics teacher object, math teacher object, chemistry teacher object, and so on.

(Refer Slide Time: 21:26)

Compare parametrised procedure with encapsulation

- Procedure Avemarks takes a field as a parameter
 - We can call Avemarks with Total or the subject name
- Datatype ClassAve has a procedure average() within it
 - We call average() for each object of ClassAve datatype
 - Advantage: result is stored, next call is much faster
 - Disadvantage: objects need to be created and initialised
- Caller is unaware of what is happening inside

```
AveTotal = Avemarks(Total)
AveMaths = Avemarks(Maths)
AvePhysics = Avemarks(Physics)
AveChemistry = Avemarks(Chemistry)
```

```
AveTotal = CT.average()
AveMaths = MaT.average()
AvePhysics = PhT.average()
AveChemistry = ChT.average()
```



Encapsulation



Let us just compare these two things, the parameterized procedure with this new idea of objects and encapsulation. So the average marks procedure, which we saw earlier in one of the earlier classes when procedures and parameters were being discussed, looks like this.

So it is a procedure which takes one parameter and that parameter is a field. So you can call average marks with the total field and it will return average total, you can call it with maths field and it will return the average of maths, and so on.

So we can call average marks procedure with a field name, it could be total or it could be the subject name. So that is how we wrote the procedure call with different field values. But when we write the thing with objects, this is how we would write it.

So now, we are saying data type class average has a procedure average within it and there are four objects now of type average, of type class average, which is class teacher, math teacher, physics teacher, chemistry teacher.

So rather than writing average total equal to call the procedure average marks with the field total as an argument, we say average total equal to CT dot average. And average math is MaT dot average, average physics is PhT dot average like that.

So you can see that subtle difference between the two things. In the procedure call, you pass the field as a parameter to the procedure and get the result, whereas here, you are making an object

and calling the objects procedure, CT dot average. So there is a slight difference, whereas the, in some sense, the object was the parameter of a procedure earlier. Now, the procedure is in some sense, the parameter of the object; you can think about later. You are calling the procedure of the object, so that is how we are thinking about it now.

Now, the advantage is that as we saw earlier in the previous slide, that the result can be stored. It could in the case of the average mark procedure, result could not be stored because you do not know where to store it. Whereas here, we can store the result and so the next call can be much faster.

Of course, the disadvantage is that first you have to create these objects. So you have to make CT, you have to make MaT, you have to make PhT, you have to make ChT. And for making each of these, it means that processing the data cards in the classroom dataset and converting them into lists, and then storing it and all that.

All that work you have to do; so this work, to create all these objects. But that is one time and once you have created this object, you can keep calling average as many times as you want and it will keep doing it very fast because it returns it from the stored value.

Now, whether you are doing procedure call or we are doing this, this encapsulation type thing, the caller is unaware of what is happening inside. In the procedure also, there is something going on inside, the caller does not know what is going on inside. In the object also, there is something going on inside the object, the caller does not know what is going on. Both cases, caller is unaware of what is happening inside.

(Refer Slide Time: 24:07)

But what if the dataset is not static?

- If the dataset changes (consider for example that we add a new student to the class), then the stored average values will be wrong ! How do we deal with this?
- We need to manage the addition of a new student carefully
 - write a procedure `addStudent(newMark)` in the `ClassAve` datatype which takes as parameter the marks of the new student (total, or the respective subject marks)
 - A new student can be added only via the use of this procedure
- `addStudent` will need to append `newMark` at the end of its `marksList`
- `addStudent` will also need to reset `aValue` to `-1`, so that the average will get computed again



IIT Madras
ONLINE DEGREE

Encapsulation



Now, let us consider what will happen if the dataset changes. Recall, I said earlier that, yes, of course, the dataset is more or less static but dataset can change; a new student can come and can join the class.

Now, what will happen if a new student joins the class? Then what if you call `class teacher dot average`, it will see that `aValue` has been computed. So you will return that `aValue` and you will get an average of old average, which means the average of the class without this new student being added. And there is no way to add a new student because I have to reconstruct the object again from scratch and then call this new objects `aValue`; call this new objects `average`.

So suppose I do not want to create this object from scratch again, why should I create this object from scratch again? I just want to add a new student to the class. If we want to do that, then how do I do that? The way to do that basically is that we have to manage the addition of a student rather carefully.

So let us say we add a new procedure called `add student` to this class average data type and the object, the purpose of this `add student` procedure is to add a new student to the class. Now, obviously, when you add a new student, then you have to insert new student's marks in the marks list, which is there in class average. So you pass that new mark as a parameter to `add student`.

So we you have a procedure called add student, new mark, which is the mark of that student. If you are looking at class teacher, then we need the total marks. If you are calling the physics teacher add student, you will pass the physics marks, and so on. So you pass the correct new mark to the object and the object basically will then add this new student to its list; somehow it does it.

Now, what should the add student really do in order to do basically this? One thing it has to do is, it has to take this, it has a marks list, it has to take this new mark and append it to the end of its marked list. It can go to the beginning also because you are not depending on the order but, let us say, we want to append it to the end of the marks list, we know how to do that you take mark list and append it put it at the end of the marks list.

Now, issue is that though we may have added an element to the mark list, aValue is not minus 1 if it has been computed already. If that were the case, then average will return aValue without computing it. So you must make average to recompute the average from scratch, from first principles. So which means that add student will also need to reset aValue to minus 1.

So what this does basically is once aValue is reset to minus 1, the next caller, when they call average, it will see that the aValue is minus 1 and it will really compute the average again. So very simple; add student, all it needs to do is append the new mark to the end of the list and reset aValue to minus 1.

(Refer Slide Time: 26:57)

Complete example

- So what does ClassAve datatype look like at the end of all this?
- Has two data fields and two procedures
 - field marksList contains a list of marks (which could be total marks or marks of one subject)
 - field aValue which is either -1 (not computed) or holds the computed average value
 - procedure average() returns aValue if it is not -1, else it computes the average of marksList, stores it in aValue and returns it
 - procedure addStudent(newMark) appends newMark to the end of marksList and resets aValue to -1
- Note that the procedures average and addStudent are assumed to have access to the fields marksList and aValue, so we can reference these fields by just using their name (it is as if they are implicitly passed as parameters to the procedure)
- Note also that the procedures are allowed to (and expected to!) make changes to these fields - so the potential side-effects are made explicit
- But what if the external world access marksList or aValue directly?



IIT Madras
ONLINE DEGREE

Encapsulation



So what does the class average data type look at end of all this? It has two data fields and two procedures. What are the two data fields? It has the data field marks list which contains the list of marks, whether it is total marks or subject marks.

And it has a field which is called aValue which is either minus 1 which means not computed or holds the computer average value. And it has a procedure average that returns aValue if it has been computed already. It means if it is not minus 1 or else it computes it and then stores the result in aValue.

And procedure add student, what would it do, it will take the parameter which is the new mark and append it to the end of its marks list and it will reset aValue to minus 1 which will force the average to be computed again.

So this collection now, this collection, which is what is this collection? This encapsulation is a packet; it contains two fields and two procedures and this package is now not only going to compute the average very fast, but also is going to be accommodative of new students. This code allow you to take new students, so we are done. You have got what we need.

So there is a problem, though. What is the problem? Problem is that what prevents somebody from outside, external caller to simply take a class teacher CT and get access to its mark list. So

somebody can say CT dot marks list and get the marks list of the class and start processing the mark list do whatever they want in the mark list, they can do that.

So nothing or they can look at CT dot aValue directly without actually calling procedure average and they can set aValue to some other value, 3, let us say. Some malicious person can call CT dot aValue and take CT aValue and put 3 in it. And then the average will become 3, and then when you call average, you will get 3 and you will find out, you will not know what is going on.

So somehow this object should also, this class, this data type should also be able to say to the external world, you do not have access to the fields, aValue and marks list. You only have access to two procedures. Average, you can call average to find average or you can call add student to add a new student; you are not allowed to do anything else.

So we need to have a way to hide some things from the external world. So what I wanted to say also basically that, in a sense, what I did not mention is that average and add student can without saying, it is not necessarily that CT dot average, CT dot aValue to access aValue, it can just say aValue; that is enough because it is assumed.

You can also write CT dot aValue but if you write aValue, it is assumed basically, that aValue is a field of my object, my own object. So you do not need to you can just use their name and it is as if it has been passed as a parameter to the procedure though you are not passing it as a parameter.

So the effective thing is that every procedure inside an object has access to all the fields of the object by name, they can just use them as if they are local variables. And side effects to the fields are expected. Not only is it possible to make side effects to the field but you expect to make because as you remember, I said basically that between one call of a procedure and another call of the procedure this, but this particular object retains the state.

Now, how does it retain the state? It retains the state by changing some fields, like we saw that it retains the state whether the average has been computed or not by setting aValue to some value. So it is retaining the state. So we in fact, expect the procedures will have side effects and the side effects are to the variables, the field values of the data type.

(Refer Slide Time: 30:31)

Complete example

- ClassAve should be able to restrict access to its fields
- This is typically done by declaring the field as either a **private** or a **public** field
- ClassAve has two private fields and two procedures:
 - private field marksList contains a list of marks
 - private field aValue which is either -1 (not computed) or holds the computed average value
 - public procedure average() returns aValue if it is not -1, else it computes the average of marksList, stores it in aValue and returns it
 - public procedure addStudent(newMark) appends newMark to the end of marksList and resets aValue to -1
- Note that a procedure could also be declared private in which case it is not available to the outside world



Encapsulation



So how do you hide or restrict access to the fields of a data type? And this is done by typically marking or declaring the field as a private field or a public field. You will find various annotations that are available in different programming languages to do this. So but all of them basically, the intent is to mark the field itself to say whether it is private or it is public.

Private means it is not available to the outside world, public means it is available to the outside world. So class average, we know now has two private fields; marks list and aValue, which is not available to the outside world. It is only available procedures inside class average.

And it has two public procedures, average and add student, which anybody can call. So there are two published or public procedures and two private fields. And only the procedures of this particular class can access its private fields. So this way you can hide the access to marks list and aValue, and so this object or this encapsulation is now secure or it is safe.

There is no way for anybody to tamper with it anymore. It has got a proper bundle of things, whatever it needs and it has got these two interfaces through which you interact with it average and add student.

So note that you can also declare a procedure to be private. Of course, you can ask if you declare a procedure to be private, what use is it because nobody can call that procedure from outside. The answer is that it may be useful as a procedure within this object itself.

So if I want to create a procedure, which will be useful for other procedures inside this object, then I can write a procedure like that, which is in some sense, a data processing procedure and that procedure can be made private, and only the other procedures inside this object have access to it; that also can be done.

(Refer Slide Time: 32:12)

Example: Shopping Bills dataset

- Questions asked of the shopping bill dataset
 - How many items of a specific category (e.g. shirts) were sold?
 - What is the minimum, maximum and average unit price for that category?
- We could define a datatype Category and create an object Shirts of this datatype
- What are its fields?
 - copy of the entire shopping bill dataset can be stored, but this is wasteful
 - We only need the list of items (i.e. rows of the bills) which are of that category
 - Category could have a private field itemList which carries all the row items from all the bills which are all of the same category
- The datatype could also encapsulate procedures that provide the desired information:
 - count() returns the number of items (i.e. size of the list)
 - min() returns the least value of unit price at which the category item was sold
 - max() returns the largest value of the unit price
 - average() returns average across all unit prices for that category

Encapsulation

IIT Madras
ONLINE DEGREE

Now, I want to take one more example before we conclude this particular lecture on Encapsulation and that is to look at the shopping bills dataset. And the reason why we are trying to take another example, which is kind of very similar, is because we want, I want to show you that these, objects that we are making also can resemble each other.

Just like code can resemble each other, we can find patterns between pieces of code, and so on, the structure of objects also may resemble each other, you might find patterns between objects. That is what we want to see.

So let us assume this shopping bill dataset is there and we want to ask ourselves what kind of questions we ask. So we, typically, somebody might ask questions like, how many shirts were sold or what is the price of the shirt, what is the average price at which a shirt was sold, or maybe even what is the maximum price at which the shirt was sold, what is the minimum price at which the shirt was sold or bought.

These are the questions that one might ask. This may be questions that may be asked very often primarily, because whenever somebody is, if you are an analyst, and you have collected all the shopping bills from the shop, these are the kinds of questions that you may want to know. You might want to know what is the range of prices at which shirts are being sold in the market, and so on.

So that is why it is important to know these kind of things. So let us say we define a data type called category and we create an object. Just like class teacher was an object of type class average, so we are saying shirts, the object shirts is of data type category and what kind of fields does it need to have?

It can, in principle, have access to all the shopping bills. We can make a copy of all the shopping bills and put it inside this object but that is wasteful. I mean, we do not need all the shopping bills; what do we need? In order to compute the average or the minimum or maximum of shirts, we need only the line items inside the bill where shirt is found.

If inside the bill there is a line item in which you see a shirt, then that line item we need to keep. All other line items something will be about food, it will be the name of something, customer and shop, and many other things, we are not interested in all that. We are only interested in the line items which have shirt in it.

So suppose I make a list of all the line items of the shopping bills which just contains this category and store that. So I process this entire shopping bill dataset, create a list of all the row items in which shirts are present, and that list is what I store. So category could have a private field; remember, private field because we do not want the rest of the world access to it, called item list which carries all the row items of all the shopping bills in which, those row items in which basically shirt is present; so shirt.

So item of type shirt, which is being sold. So we are picking up all the shirt sales from across all the bills in the shopping bill dataset and we are making a list out of it. And this list is stored in the private field called item list of category.

Now, what kind of procedures do we need? We saw that we need basically, we may want to find out how many shirts are sold. So we need a, we need, so we will see later how to find how many

shirts were sold but let us say, we have a procedure called count, which counts the number of items, that means, size of the list.

How many row items, how many row items, how many purchases of shirts were made? Now, each purchase of shirt may be 1 shirt or maybe 3 shirts or 2 shirts; we will see that later, how to handle it. But how many row items are of the bills are basically carrying shirts. So that basically is count.

And min basically is number one, row item of a shopping bill, what it looked like. It had some fields, it had basically the quantity, the number of shirts sold, and it had a unit price, which basically said, what is the price of a shirt. And then you multiply these two and it gave you the cost for that line item. So how much was spent on that line, how much money was spent on shirts.

So those were, that was what is there in one line. So if you look at the unit price, which is basically the one unit price for a shirt, we may want to find what is the minimum unit price across all the bills for shirts.

And, similarly, we might want to find the maximum unit price for across all the bills for shirt. And we may want to find the average unit price. So what is the average price at which what was sold, what is the minimum at which it was sold and a max at which, so these probably are the procedures we need.

And if we hide the list inside and give access only to these functions, count, min, max, and average, then the only thing that people can do with this now is to find these values and nothing else they can do with the list. So you can you can, therefore, control what people do with it.

(Refer Slide Time: 36:43)

Example: Shopping Bills dataset

- The Category datatype encapsulates one private field and four procedures
 - private field itemList that carries the list of items, which is hidden
 - public procedures count(), min(), max() and average() that anyone can call
- What if we want to do this for a particular shop? Or for a specific customer?
 - The itemList can store only the items sold by that shop
 - ... or bought by a specific customer
 - The procedures will all work without any change !
- Multiple objects of type Category can be created - the list being filtered by category alone, or by a combination of category, customer name, shop name, or anything else.
- What if we want to accelerate the execution of these procedures?
 - Just store their results in private fields - say cValue for count, minValue for min, maxVal for max and aValue for average.
- Can you see how the pattern for encapsulating the Category is quite similar to that for encapsulating the ClassAve?



Encapsulation

13 / 16

So the category data type encapsulates one private field, which is item list, and four procedures count, min, max, and average that anybody can call. Now, if you want to do this for a particular shop, that means you want to find all the shirts sold by big Bazaar or you want to find all the shirts sold by Sun General, then what do you do?

So do we need to create a completely different class? The answer is no. Because whatever we had created this category data type, basically all it had was a list and it was finding the average and minimum, maximum of that list.

So what if item list, instead of storing all the shirts across all the shops, it stores only shirts that were sold by Big Bazaar. So we make a, we filter the list by Big Bazaar and we get a smaller list. So we filter the shopping bills to extract firstly, the rows which contain shirts but not only do we do that, we only take those rows from those bills, which are from Big Bazaar. So that item list basically only contains row items of shirts that were sold by Big Bazaar. So if you have a smaller list.

And similarly, if you want to do, find out what were the items, which were shirts which were bought by a customer, then instead of filtering by Big Bazaar, you can filter by the customer name. You can find out how many shirts did Srivatsan buy, for example. So you just filter by Srivatsan.

So you can filter it by shop name or by customer name or anything else, and you will get a smaller list but all the procedures will work, whether you do average, min, max, anything; all of these will work in exactly the same way. So this basically means that we can make any number of objects of the type category.

We do not need to create another data type, whereas the same data type, any number of objects we can make but while making the object, you can apply all kinds of filters to make the list, and then basically, you can keep calling the same functions, which means count, min, max, average for all these objects, and they will all work exactly in the same way. So this makes the code really cute and neat.

Now, again, you can ask, suppose I know it is a, let us say it is a static dataset, which means that I have collected these bills and I am, as an analyst, I am analyzing these bills. And while doing analysis, I have to find average many times, so this dataset is static.

So if I have to keep analyzing this, then I do not want to keep on computing average again and again; cannot I make the average faster? The answer is of course, you can. How do you do it? Same as before; store the result after computing it first time.

So let us say we have four local fields of the category data type, called count, min value, max value, and aValue. cValue, min value, max value, and aValue which hold basically the computed value of these fields and since we know basically count cannot be negative, mean value cannot be negative, all these things can be negative.

So we can basically set them to minus 1 in the beginning. So checking for minus 1 will tell you whether it is computed or not. If it is not computed, compute it and store it, and then next time, it basically becomes non minus 1. And so, you can just return its value. So it becomes very fast.

So you can see now that the pattern for class average, which means that what we are doing, we are storing a list, we are finding the list's average. When you are computing the average, you are storing the result of average in a variable, checking if that variable is minus 1. If it is not minus 1, you compute it, you return it. If it is minus 1, you compute it. This pattern is exactly the same as what we did for class teacher.

That was for classroom dataset and this is for shopping bill dataset but the pattern looks very similar. Both of them in the objects, they have a list, they have a function called average. They have a way of storing the result of the average and returning that stored result next time.

So we can see that there are some computational patterns that are coming out which are very neat, which are not contained to only procedures but procedures, storage of things on the side and using those stored things to actually access them. All of that is part of the pattern that we are seeing and this pattern is common between shopping bills and classroom datasets.

(Refer Slide Time: 40:57)

Example: Shopping Bills dataset

- The Category datatype encapsulates one private field and four procedures
 - private field itemList that carries the list of items, which is hidden
 - public procedures count(), min(), max() and average() that anyone can call
- If Shirts is an object of this datatype, we may want to find out how many shirts were sold
 - Note that this may not be the same as count(), since one row in the bill can have multiple items
 - We can write a procedure called number() which finds the sum of the quantity field of the list items
- But number() makes sense only if the category is sold in discrete units (like shirts). What if the category quantity is measured in Kg (for e.g. grapes)?
 - For such categories, we can define a procedure quantity() that finds the sum of the (possibly fractional) quantity fields
- The issue is that just like number() does not make sense for grapes, a measure of quantity in kg does not make sense for shirts !

Encapsulation

IIT Madras
ONLINE LECTURE

So effectively, the advantage of basically, advantage of using encapsulation basically is that we can look at much more interesting patterns and work with them. Now, let us consider a slightly more, somewhat different kind of example here, where I have mentioned earlier that count, the procedure count only counts the number of row items.

Now that is not very satisfactory. I want to actually find out how many shirts were sold. One row item may have four shirts, another row item may have three shirts, another may have two shirts. So actually, what I should do is I should add up all the quantities from the rows and return that as the thing.

So maybe I should have a procedure, another procedure, which is called number. So you call number you call shirts dot number and it returns how many shirts were sold; number of shirts

sold. So what will number do? Basically, it will not just simply count how many elements are there in the list, for each list item, it will look at the quantity field and add up all these quantity fields and return it, that becomes the total number.

So we could do that. The problem is we have seen basically that certain, certain kinds of only certain kinds of category items have discrete things where a number makes sense. For example, if you are buying grapes, it does not make sense to count it in terms of number of grapes or something like that, it does not make sense at all.

On the other hand, much more natural unit for grapes is kgs. Did he buy half kg of grapes, quarter kg of grapes, 300 grams of grapes or something like that. So for grapes, you would rather measure the quantity in terms of kgs, whereas for shirts, and pants, and so on, you would rather measure them in terms of discrete units; how many items were sold like 4 or 3 or something like that.

So number makes sense for category but for kgs, maybe something else like quantity makes sense; how many kilos, quantity in kilos is probably makes sense. Quantity measure makes sense.

So we are saying basically that number does not make sense for grapes and quantity in kilos does not make sense for shirts. Nobody buys shirts in kilos, they buy grapes in kilos and they buy shirts in numbers.

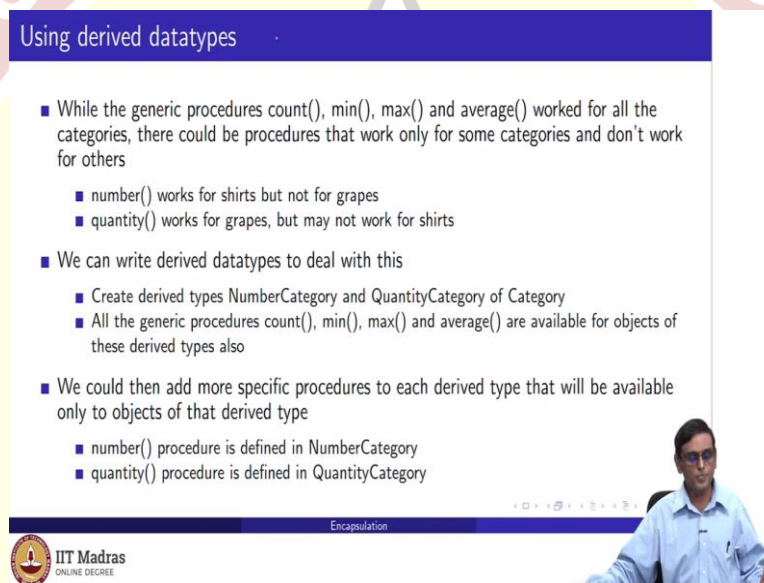
So number makes sense for shirts but not for grapes and quantity makes sense for grapes, but not so if, I one thing I can do is I can add, I can put a procedure called number which returns a integer and I can put a thing called quantity which returns a fractional value because when you add up all the quantity elements, you may get 1.35 kilos, something like that. So quantity returns a fractional value, a number returns a integer.

I could have both but then what happens is somebody from outside and you have to make it public because you do not know who is going to call. So you make both number and quantity public, then some guy might just go and call quantity for shirts. And somebody else may go and call number for grapes and you will get some nonsense value.

So to prevent that, we must make number available only for shirts and quantity available only for grapes. How do I do that? So I do not want to recreate completely the object from scratch. This class I have already done a lot of work on category by creating these procedures, min, max, and all that which seems to work for everything.

So I want to continue to use min, max, count, and average but I now want to add two more procedures number and quantity but number is only for shirts and quantity only is for grapes. So how do I do that?

(Refer Slide Time: 44:17)



Using derived datatypes

- While the generic procedures `count()`, `min()`, `max()` and `average()` worked for all the categories, there could be procedures that work only for some categories and don't work for others
 - `number()` works for shirts but not for grapes
 - `quantity()` works for grapes, but may not work for shirts
- We can write derived datatypes to deal with this
 - Create derived types `NumberCategory` and `QuantityCategory` of `Category`
 - All the generic procedures `count()`, `min()`, `max()` and `average()` are available for objects of these derived types also
- We could then add more specific procedures to each derived type that will be available only to objects of that derived type
 - `number()` procedure is defined in `NumberCategory`
 - `quantity()` procedure is defined in `QuantityCategory`

Encapsulation

IIT Madras
ONLINE DEGREE

Now, the way to do that basically is to use what are called derived data types. So we create two derived data types, one is called number category, another is called quantity category. And these two derived categories are derived from category. So every procedure of categories available.

So generic procedures like count, min, max, and average are all, remember this notion called subtype, which we talked about when we did data types. The subtype basically has access to all the parents procedures but may override some of them or maybe narrowing them down or redefining them.

So in the same way here, the derive type has access to the parent type. In this case, quantity category has access to all of categories procedures but in addition, the quantity category has its

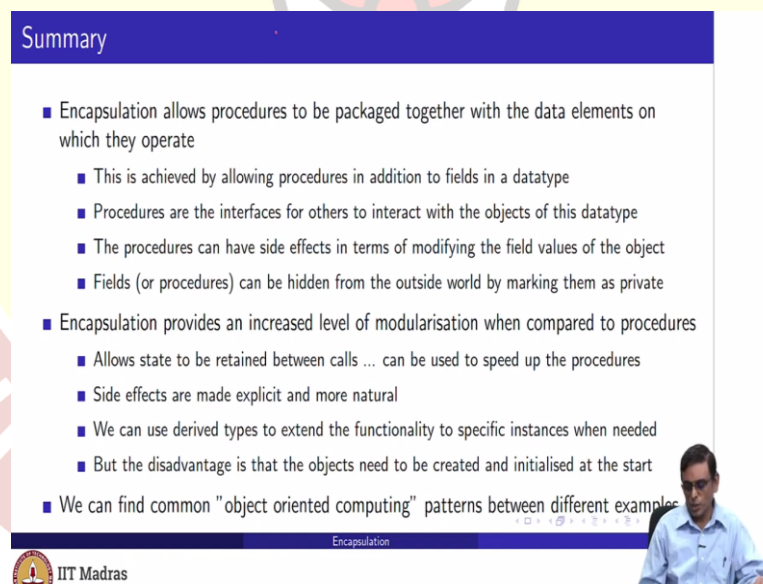
own procedure, which is quantity and the number category has its own procedure, which is called number.

So now, you cannot call quantity of number category item like shirts. So now, we define shirts not to be an object of type category but you call shirts an object of type number category. So that it not only has access to count, min, max, and average, but it also has access to another procedure called number.

Similarly, you can define grapes as an object of type quantity category, so it will have access to count, min, max, average and also has access to quantity. But grapes will not have access to number anymore because grapes is of type quantity category and quantity category does not have number defined on it.

So this is the way you deal with these kind of specific variations or branches that you are getting in things when you start from the top.

(Refer Slide Time: 45:51)



Summary

- Encapsulation allows procedures to be packaged together with the data elements on which they operate
 - This is achieved by allowing procedures in addition to fields in a datatype
 - Procedures are the interfaces for others to interact with the objects of this datatype
 - The procedures can have side effects in terms of modifying the field values of the object
 - Fields (or procedures) can be hidden from the outside world by marking them as private
- Encapsulation provides an increased level of modularisation when compared to procedures
 - Allows state to be retained between calls ... can be used to speed up the procedures
 - Side effects are made explicit and more natural
 - We can use derived types to extend the functionality to specific instances when needed
 - But the disadvantage is that the objects need to be created and initialised at the start
- We can find common "object oriented computing" patterns between different examples

Encapsulation

IIT Madras

So to summarize, the idea of encapsulation basically is to package procedures along with data elements on which they operate, so that the data elements in the procedures together, neatly can be put in a box. And we saw that we can use the notion of a data type, we can extend the notion of a data type to handle this by just adding a procedure field to the data type.

And the procedure field basically can make naturally have access to all other fields and also can make side effects because we now expect it to make side effects to the other fields of the data type. And this basically has a very interesting use case, which is basically that you can accelerate procedures, for example, and you can store the result, use that state between two calls and so on.

So encapsulation is therefore, in some sense, a little better than procedures in terms of how it modularizes. Of course, procedures is also modularize, it also has parameters, it has some result value, it has local variables, it hides the local variables, it separates the implementation from the caller, and so on. But you have some additional modularization possibilities in encapsulation.

First one is what we discussed, it retains the state between calls, and so we used it to speed up processes, you can use it for other things. It makes side effects much more explicit and natural as compared to procedures and by using derived types, you can keep on extending the functionality and do what you need.

The disadvantage, of course, of this object oriented method is that you have to, for every task, you have to first create the object and initialize it; set up it set up all the things inside it. So all this work you have to do. So there is work to do before you start, so that is the disadvantage.

So finally, we said basically, that by looking at patterns between different problems, when they are represented as objects, you can, you can get much better insight into what is going on. Not only are you seeing patterns in the procedure, you are also seeing patterns in the data elements, the procedures, the way the data elements and procedures are packaged together to work with each other like we saw in the case of storing and retrieving the aValue for the classroom dataset and we saw that it is very similar to how it works in the shopping bill dataset. So with that, we come to the end of encapsulation. So we then, see you next week.