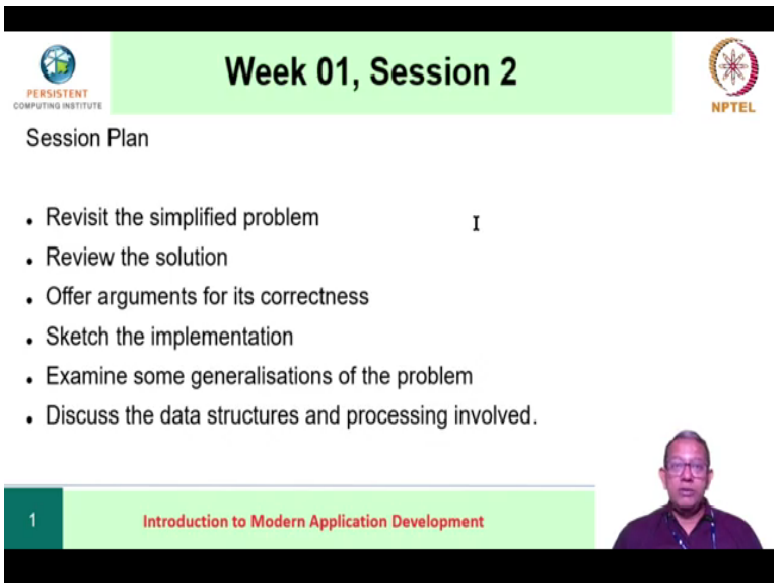
Introduction to Modern Application Development

Prof. Aamod Sane
FLAME University and Persistent Computing Institute
Abhijat Vichare
Persistent Computing Institute
Madhavan Mukund
Chennai Mathematical Institute

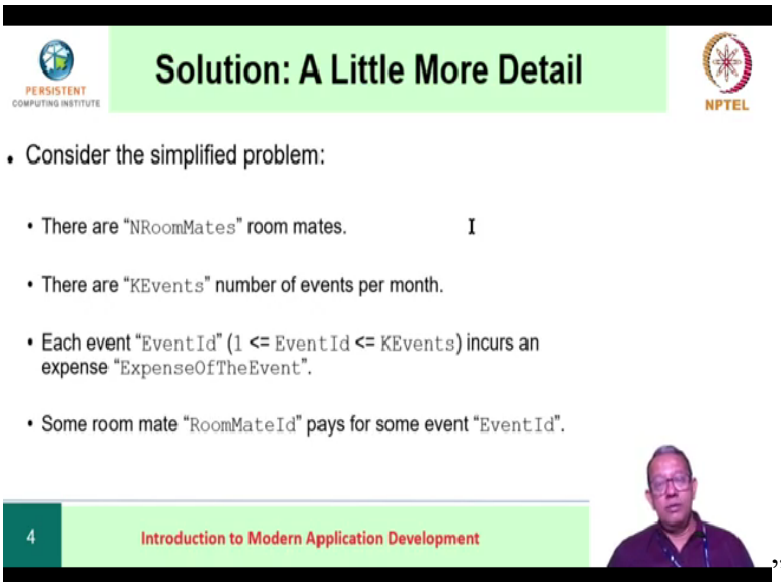
Lecture-04
Introduction To Modern Application Development Part 4

(Refer Slide Time: 00:11)



Hello there, welcome to the second session of your first week. What we will do in this session is to revisit the simplified problem, review the solution again, and mainly to offer arguments about correctness. This is what we will see mainly in this session. We will also see an implementation particularly based on spreadsheets. We will examine the generalizations of the problem and discuss the data structures and the processing involved.

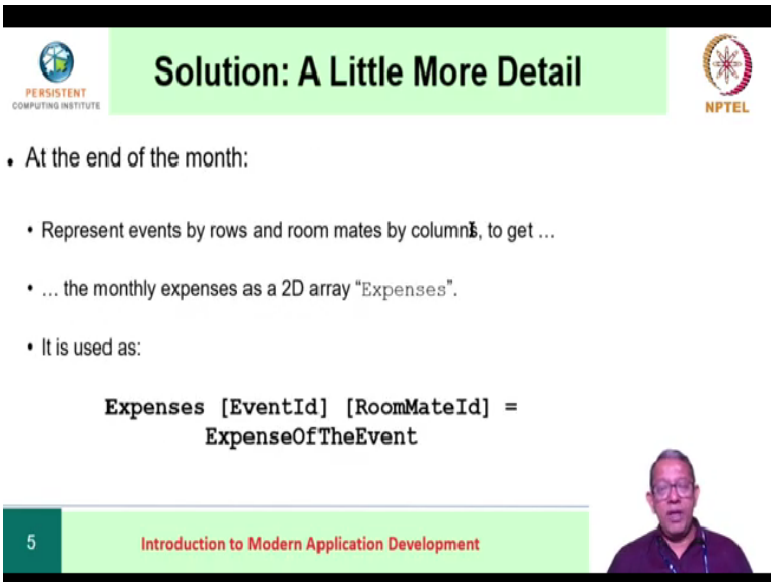
(Refer Slide Time: 00:51)



Professor Sane has discussed the simplified version of this problem where two and three roommates were considered. Now we shall generalize to an arbitrary number of roommates. We will using following notation:

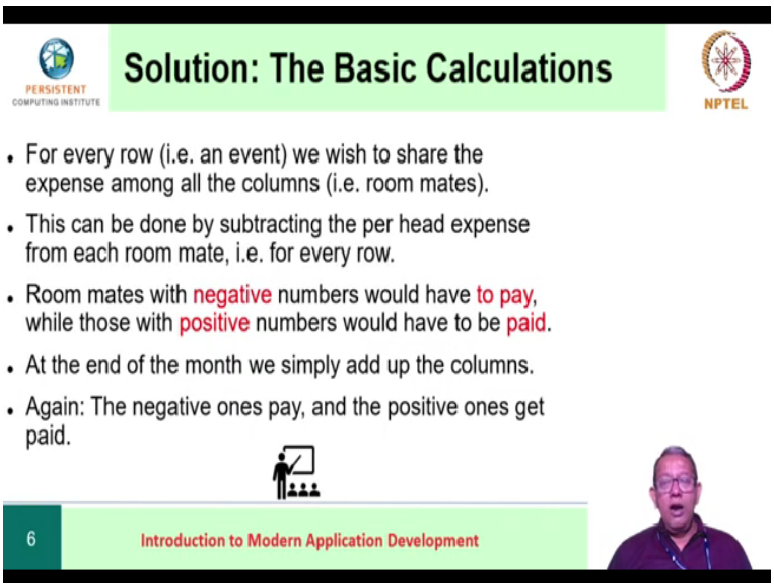
- We will denote the number of roommates by `NRoomMate`.
- `KEvents` will denote the event.
- Each event will be denoted by an `EventId`.
- Some roommate will be denoted by a `RoomMateId`.

(Refer Slide Time: 01:30)



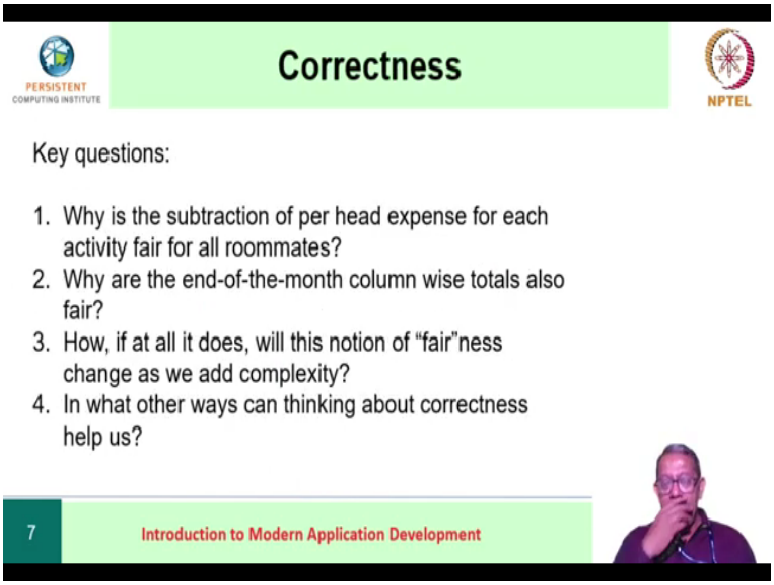
At the end of the month, we will represent an event by rows and each roommate by a column. Therefore, at the end of the month, we will have a set of rows denoting the various events that have been shared by the roommates. And along each column will be the expense or the money that has been contributed by one roommate for a given event. This 2-dimensional array, or matrix, can be represented as expenses for an `EventID` by a roommate, `J`, whatever may be the expense.

(Refer Slide Time: 02:21)



For every row we wish to share the expense amongst all the roommates, that is columns. This can be done by subtracting the per head share for each roommate from every roommate's current state of affairs. Roommates with negative numbers will be the ones who will have to pay while roommates with positive numbers against their names will be the ones who will receive money. At the end of the month, we simply add the columns. Again, the negative ones will have to pay and the positive ones will be receiving money.

(Refer Slide Time: 03:09)



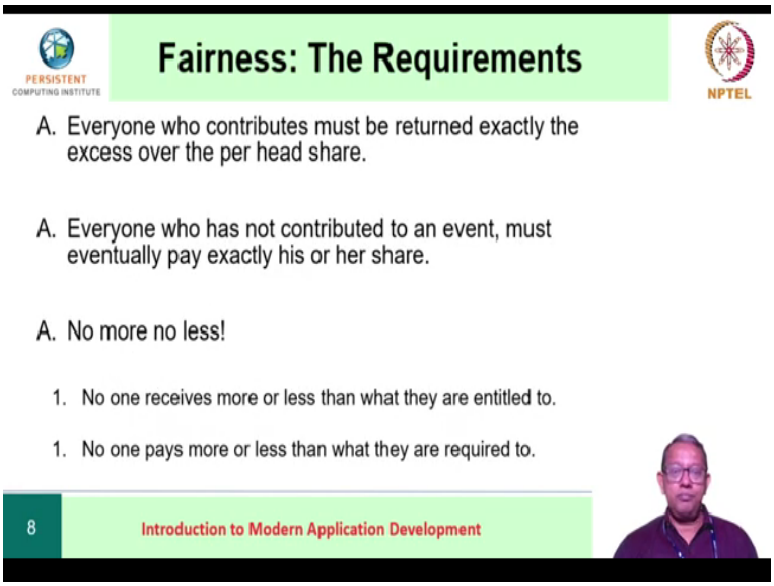
Let us look at the correctness of this approach. Following are some questions whose answers would be very important to us, after all, we are making a machine do these things, not another human being:

- What do we really mean by correctness?
- What is it that we want? Why is the subtraction of per head expenses for each activity fair for all roommates? Why are the end of the month column wise totals also fair?
- How will this notion of fairness change as we add complexity to the problem?

After all, we are considering the simplified version of the problem. But it is a good idea to have this thought in our mind as our complexity gets added, in what other ways can thinking about

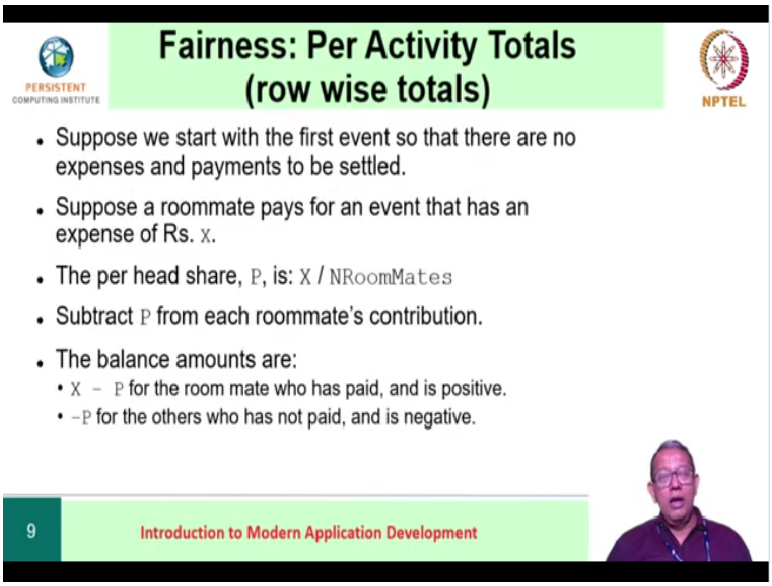
correctness be useful to us. There could be other questions too, but for our purposes these are key questions for us. Let us take them one by one.

(Refer Slide Time: 04:20)



What do we require out of the idea of correctness or the fairness approach? Well, everyone who contributes more than his or her share must be returned exactly that excess contribution. Similarly, anyone who has not contributed must eventually pay exactly that same amount, whatever they owe. In fact, more strongly, no more, no less. No one receives more than whatever is the share that he or she is entitled to. And no one pays more than whatever he or she is required to. This notion of fairness – you only pay as much as you owe and only receive as much as you have contributed.

(Refer Slide Time: 05:24)



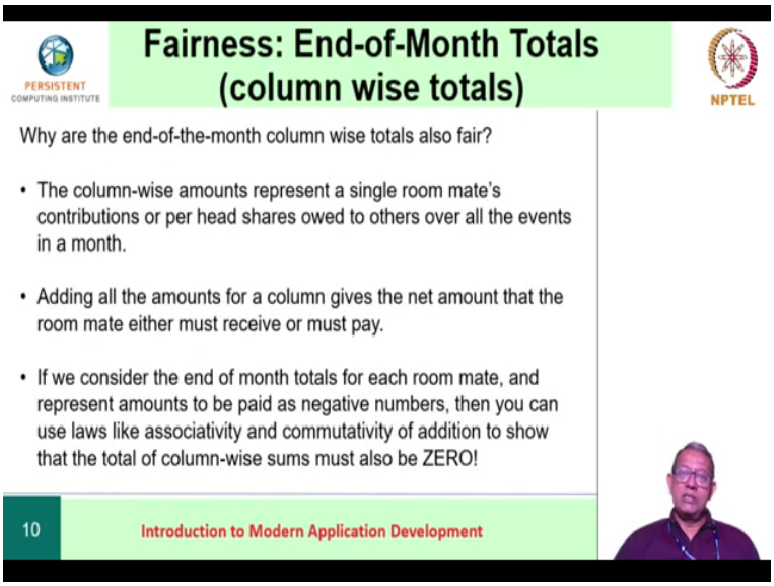
Let us look closely at how this idea of fairness actually works out, given our solution approach. Suppose, for example, we start with the first event, so that there are no expenses or payment to be settled or reconciled. Suppose a roommate pays for an event, some amount `X`. Then given that there are `NRoomMates` in total, the per head contribution is `X` divided by the number of roommates. Let us call this per head contribution as P.

$$P = \frac{X}{NRoomMates}$$

Our approach says that we subtract P from each roommate's contribution, which currently is 0, because it is the first event. As X has been contributed by that roommate, and P is his or her share. Therefore, the balance amount for this roommate who has contributed would be $X - P$. And since it is subtraction of P from 0 for the other roommates who have not contributed, we will see $-P$ for each roommate who didn't contributed.

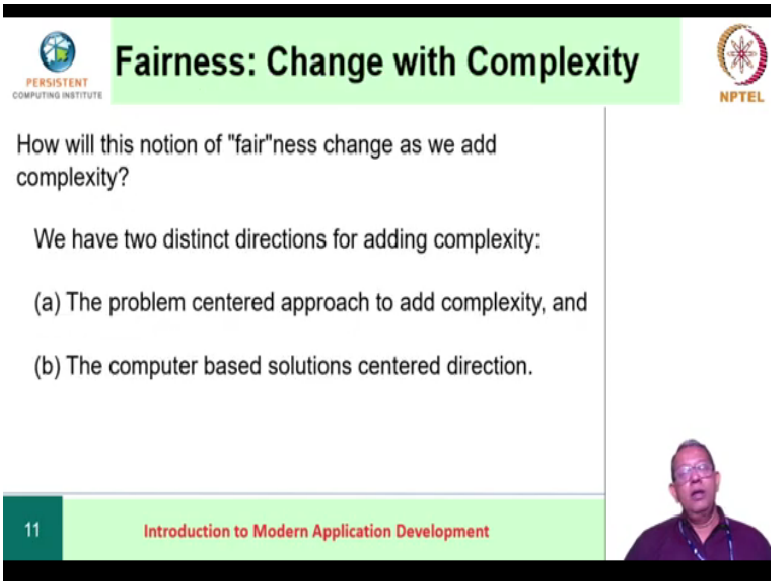
This is quite consistent with our idea that roommates with negative numbers against them would have to pay, and roommate with positive numbers against them would have to get paid. It is a simple algebra to see that the sum of $X - P$ of the contributing roommate, and the $-P$ of the rest of the roommates is 0.

(Refer Slide Time: 07:21)



Let us look at the end of the months total. Why are the end of the month totals also fair? The column wise amount represents a single roommate's contributions, or per head shares owed to others over all events in that month. Adding all amounts for a column gives the net amount that the roommate must either pay or receive. If we consider the end of the month totals for each roommate, and given that the amount to be paid are negative, and amounts to receive are positive, then you can use laws like associativity or commutativity of numbers for the addition operation, to show that the total column wise sums must always be 0.

(Refer Slide Time: 08:15)

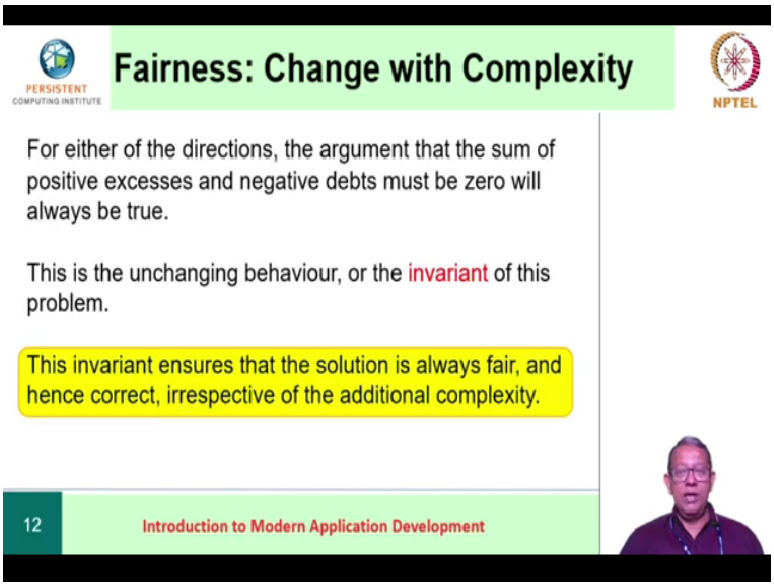


Let us go to the next step. How will this fairness change with complexity? Recall that we have 2 distinct directions in which we can add complexity in this problem:

1. One is the problems itself can be made even more general. Why should we wait at the end of the month for the totals? Why should we always have totals for everyone? Maybe I as roommate might just want my total for that point of time. There could be far more roommates than what we have just considered, and so many other ways – we have already seen all that.

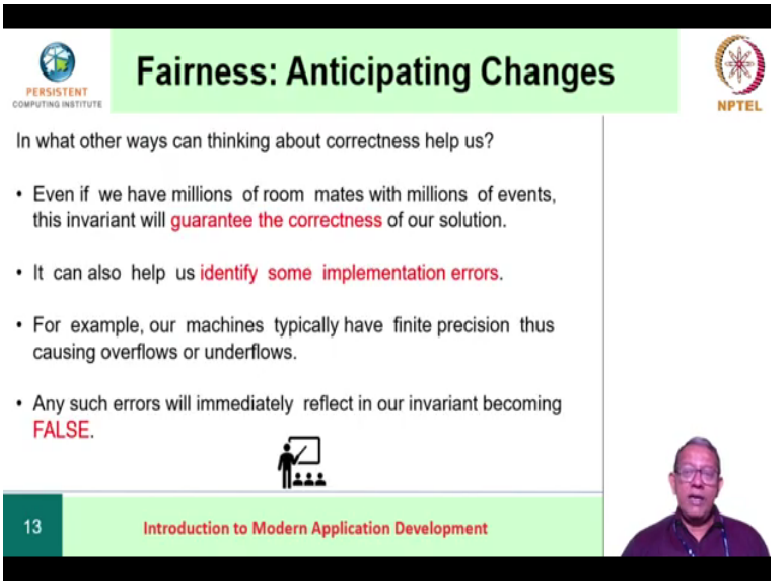
2. The second way would be the generalizing the computer-based solution. We would like to think whether our solution approach will remain fair as we add any kind of a complexity.

(Refer Slide Time: 09:23)



In fact, it will. The argument that the sum of the positive excesses and the negative debts must be 0 will always be true. This is the unchanging behavior or invariant of this problem. In fact, this invariant ensures that the solution is always fair and hence correct irrespective of the additional complexity.

(Refer Slide Time: 09:51)

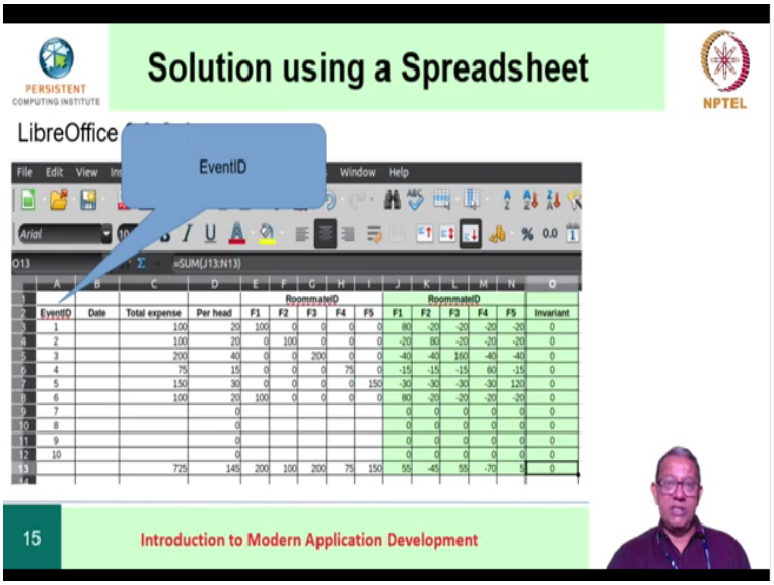


Finally, let us look at in what other ways can this thinking about correctness help us. Suppose we have millions of roommates, although it is quite unlikely, this invariant will still guarantee the correctness of our solution. In fact, it can also help us identify some implementation errors.

For example, our machines are typically finite precision machines. Therefore, overflows and under flows can occur. If we think about it a bit, then any such overflow or underflow errors will immediately reflect in our invariant becoming nonzero, which shouldn't be the case if we want to be completely fair. The moment we see that our invariant is nonzero, we know something has gone wrong!

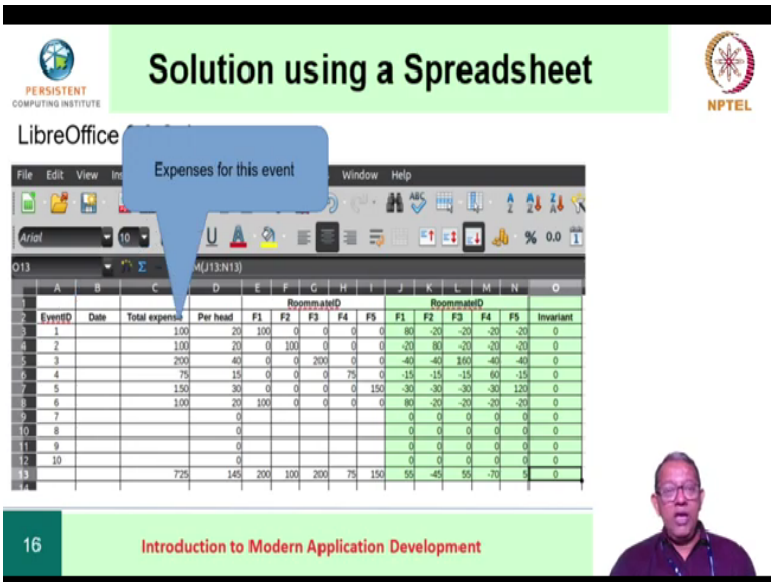
This is a good time to take a pause. And think about the issues that we have just discussed.

(Refer Slide Time: 11:04)



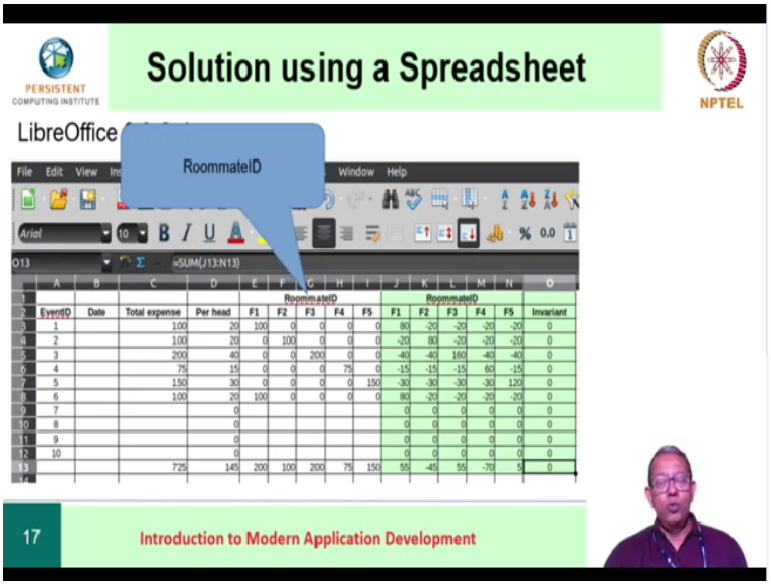
We will demonstrate the solution using a spreadsheet. On the figure given above, you will see slide which show a snapshot of Libre Office 6.0 spreadsheet. We have a column `EventID`. This basically means that each row represents an event.

(Refer Slide Time: 11:31)



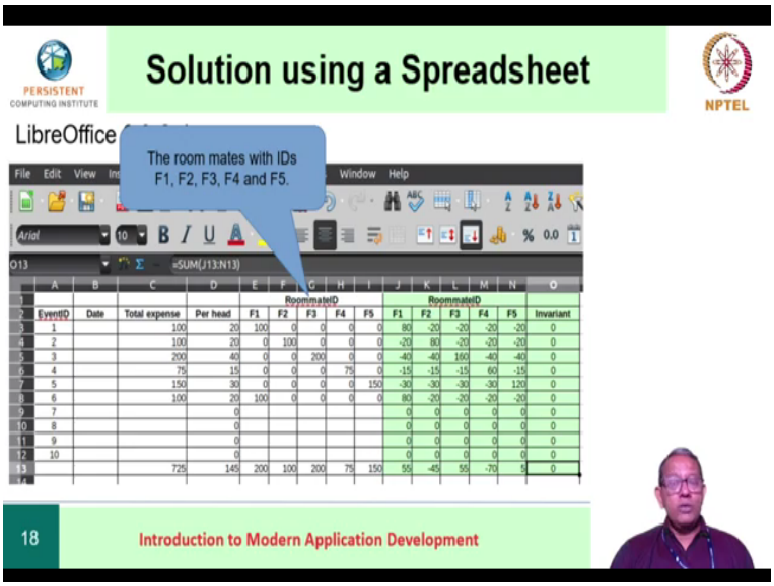
We have a column which captures the expenses for that event.

(Refer Slide Time: 11:36)



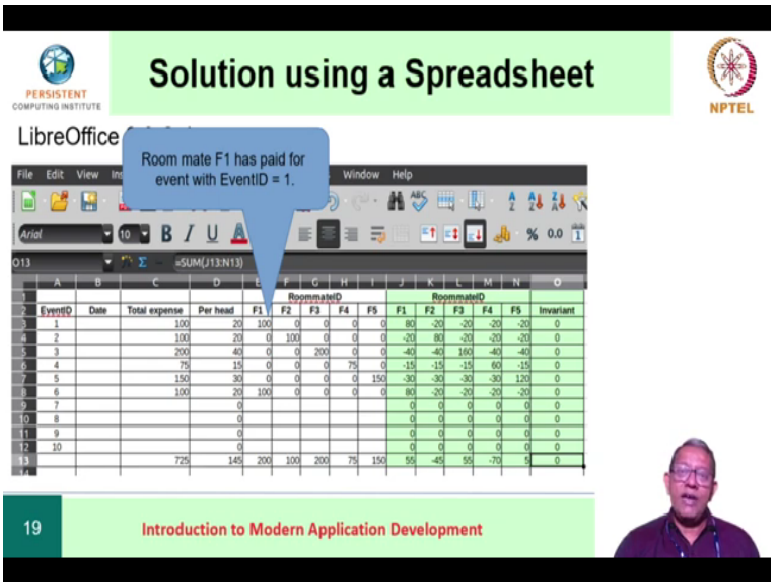
We have a bunch of columns for `RoommateIDs`.

(Refer Slide Time: 11:44)



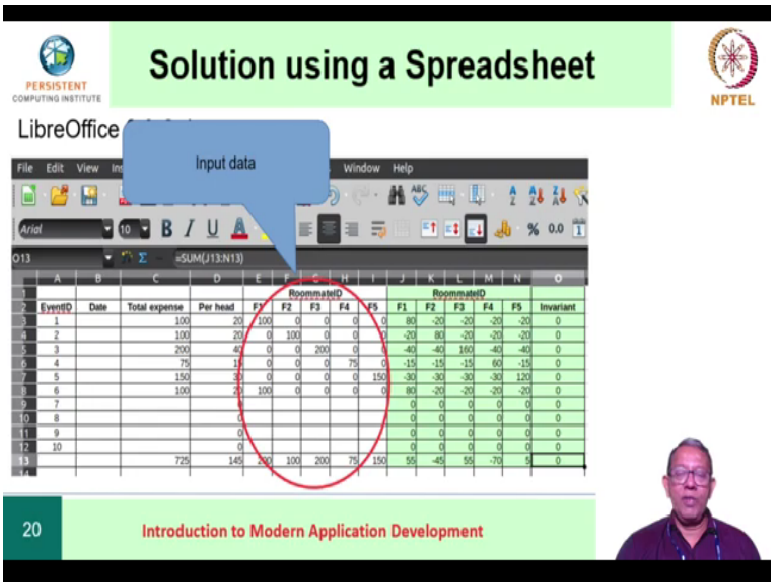
We have shown 5 roommates here, which who are `F1, F2, F3, F4`, and `F5.` In practice, these could be some real names like Ramesh, Sudha, Nikhil… etc.

(Refer Slide Time: 12:04)



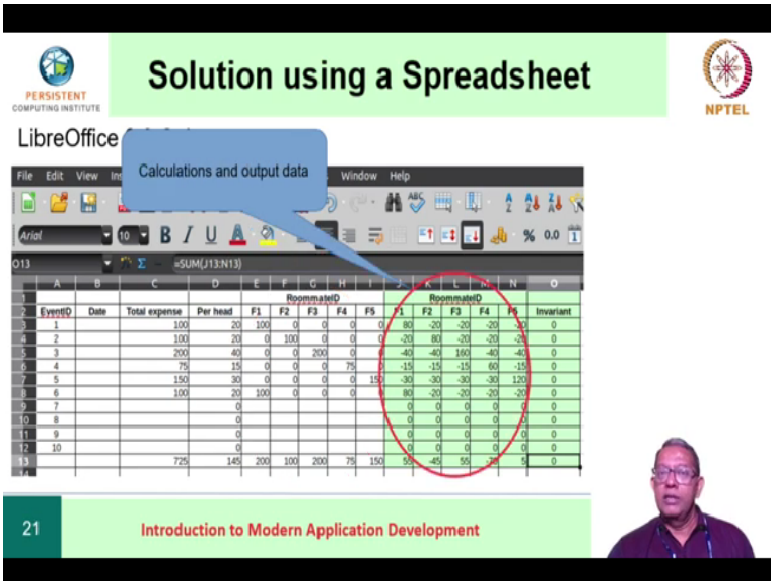
We have also illustrated that roommate `F1` has paid some amount for the event with `EventID 1.`

(Refer Slide Time: 12:12)



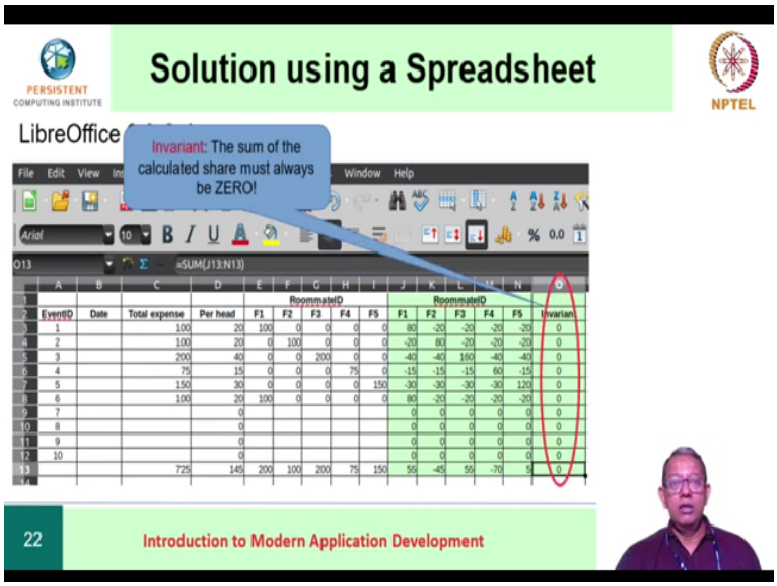
The encircled region in the figure given above denotes the input data, that is for an event, how much has been paid by whom.

(Refer Slide Time: 12:25)



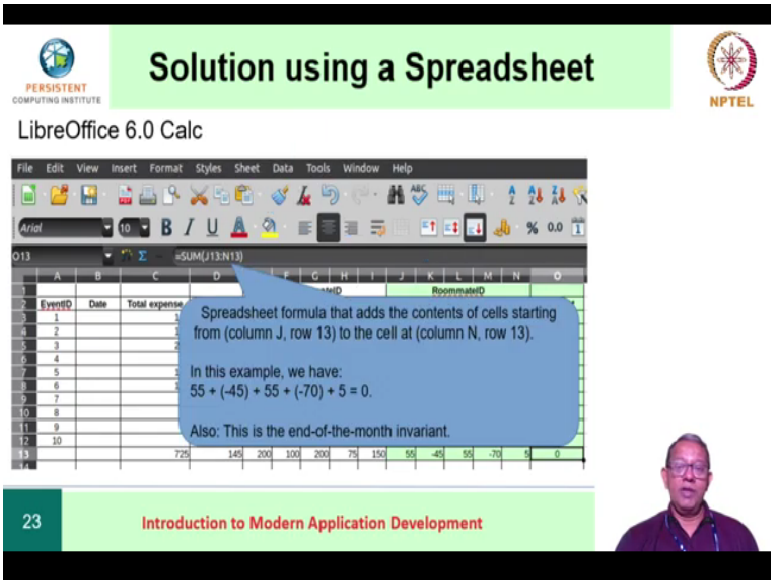
The encircled region in the figure given above denotes the calculations and the output results. This is where we calculate the fair share of each roommate for every event, and we go on accumulating the fair share for all the events.

(Refer Slide Time: 12:44)



Notice the last column which is the invariant. It is the sum of each row of the calculated part, which is the contribution minus the per head share. This column should always be zero on each row. We have already seen that this is an argument for correctness. So, in the spreadsheet, we have just summed the rows and we write that in this particular column. If that is 0 then we know that our calculations were correct.

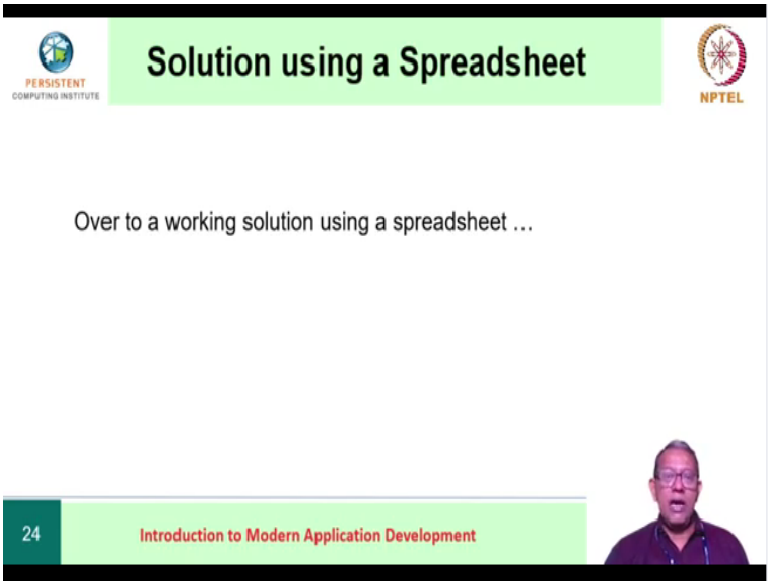
(Refer Slide Time: 13:19)



Here is also a spreadsheet formula for people who might not know about spreadsheets. This formula calculates the sum of contents of cells starting from column J and row 13 (denoted by

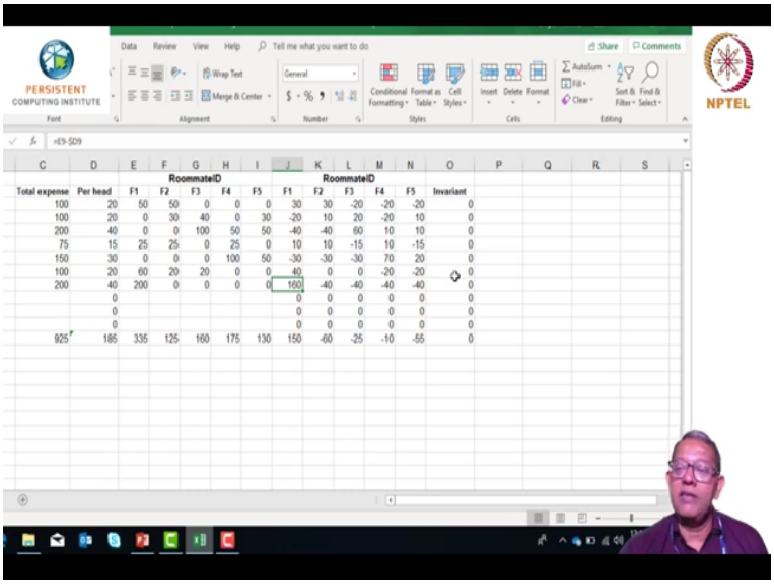
$J:13$) to column N and row 13 (denoted by $N:13$). In this particular example, we have $55 - 45 + 55 - 70 + 5 = 0$. Therefore, our spreadsheets solution is correct. It exactly balances the amount that is to be given and the amount that is to be received.

(Refer Slide Time: 14:15)



Let us have a look at actual working solution.

(Refer Slide Time: 14:21)

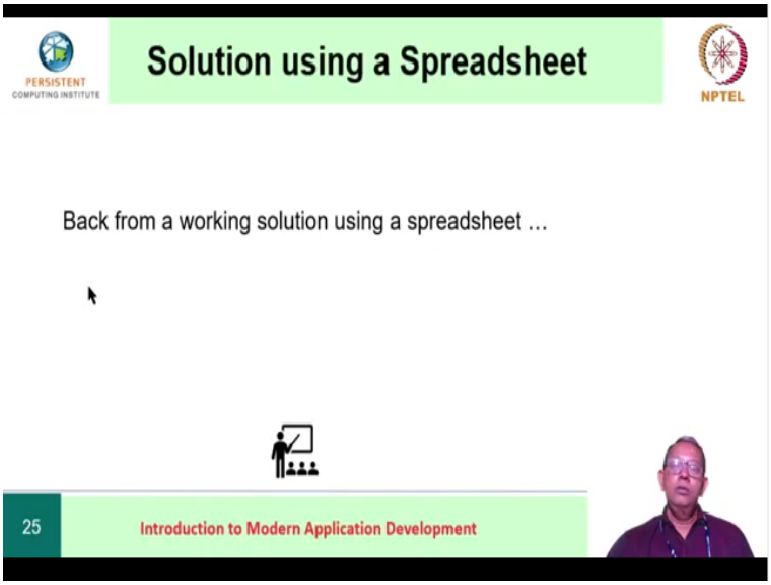


Here is an Excel spreadsheet of the same solution that we have seen. Let us add a 7th event. Let us say that it is rupees 200. This means that amongst 5 friends, the per head contribution is rupees 40. Let us assume that friend number 1 pays 20 rupees, pays all amount 200 while others do not

pay at all. Note that for friend 1, the excess amount is rupees 160. His or her share of rupees 40 has been subtracted from 200 and therefore 160 is the excess amount.

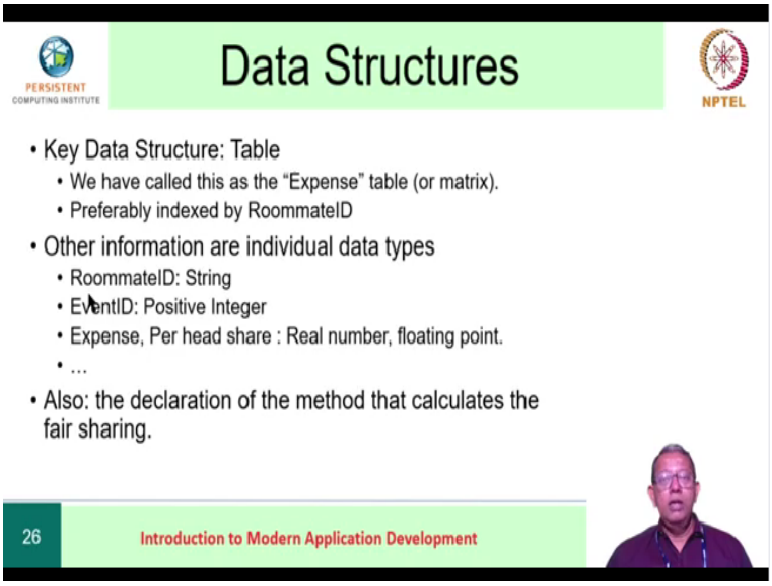
Quite obviously, the other have to pay rupees 40 each, there are 4 of them. Therefore, 160 rupees totally have to be paid by the other 4 and our friend who has contributed must receive 160 rupees. The invariant is therefore 0.

(Refer Slide Time: 15:45)



This is again a good point to pause and review what has been done.

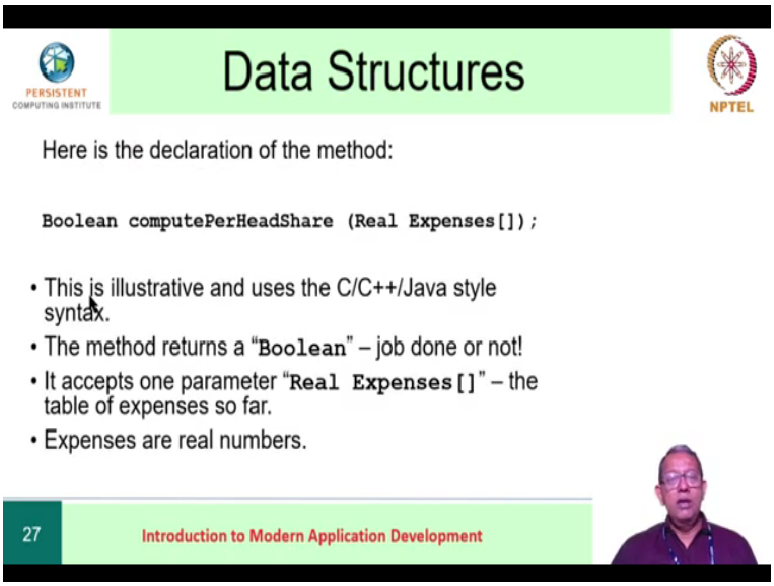
(Refer Slide Time: 15:52)



Let us now look at a few data structures of this problem. The key data structure is of course, a table. In our discussion, we have called this as the expenses table, or it may be also referred to as a matrix. It would be good if we could index this table using the `RoommateID`. This could be done, for example, by using dictionaries in Python. We leave this discussion here as the program can be implemented in any language.

For our course we have used, we will be using Java. How exactly we have to realize `RoommateID` based indexing is something which we leave it on you! The other pieces of information are really individual data types. For example, the `RoommateID` is a string. The `EventID` is a positive integer. The expense or per head share, or such objects, are real numbers or floating-point numbers.

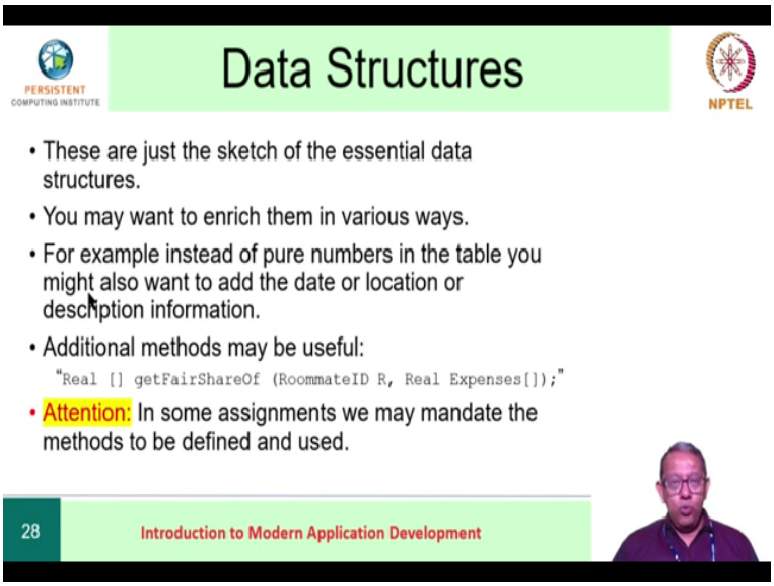
(Refer Slide Time: 17:24)



We will also need the declaration of the method that calculates fair sharing. That is one important function processing to be done in this solution. Here is a declaration sketch of the method, `Boolean computePerHeadShare (Real Expenses[]);`. This declaration is illustrative and uses a C, C++ and Java style syntax. The method returns a Boolean: which tell us whether the job is done or not. Maybe as we progress and add complexity, we might have to change the return type to a more suitable one. The method accepts one parameter, an array of real numbers named `Expenses`, which represents the table of expenses so far.

Quick Check 1: In the lecture we used `Boolean` as the return type for `computePerHeadShare`, but actually this doesn't serve the purpose well. What could be a better return type for this function, and why?

(Refer Slide Time: 18:15)



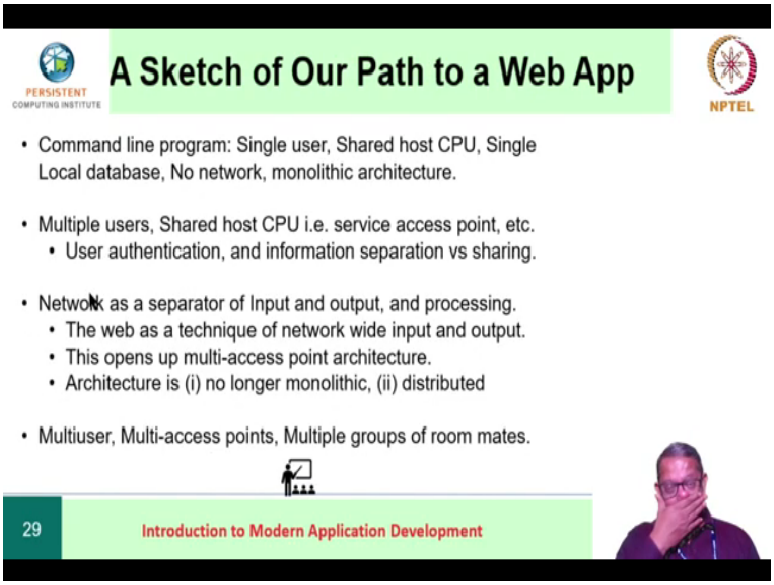
We repeat that this is just a sketch of the essential data structures. Depending on the language that you use, you might have to use the correct ways of defining these data structures. Further, you may want to enrich this data structures. These are just bare bones sketch. For example, instead of just numbers in the expenses table, you might want also want to add date or location or description information.

In our illustrations in the spreadsheet, we had shown one column date, that information was also captured there. You might also need additional methods. For example, it is possible that you might want to compute the fair share of a given roommate instead of all of them. So, you might want to return an array of real numbers indicating the share of a given roommate given by `RoommateID R`, and the `Expenses[]` array.

Quick Check 2: In the lecture we used `Real[]` as the return type for `getFairShareOf`, but do we really need an array of real number? What could be a better return type for this function, and why?

This is a point where we will want to draw your attention. In some assignments, we may mandate the methods to be defined and used, we might state explicitly that which are the methods that you need to define and use. Whenever such a mandate is given, then your solution must be constructed using the stated methods and other additional methods that you may deem to be needed. But stated methods are must required. We may use automated testing, and hence if they do not exist in your solution, we will not be able to evaluate it and, therefore, you will not earn credits for that effort.

(Refer Slide Time: 20:39)



As we come to the end of this session, let me sketch our path to the web app. We will shortly look at the command line version of the program will have following main properties:

- The command line version is going to be single user app.
- It is going to run on a single CPU that is shared amongst individual roommates.
- There will be single localized information base. There is no network.

Essentially, this is a very monolithic architecture. We would generalize by incorporating multiple users, but still with a shared CPU. That shared CPU is what we would call as a service access point. This would involve including extra piece of functionality, like user authentication. And we might have to think about information separation versus sharing. We will look into these as the course progresses.

Next, we will introduce the network. But the main idea of the network is that a separator of input and output and the processing. We will look at the web as a technique of network wide input and output. It opens up the possibilities of multi-access point architectures. Therefore, our architecture will no longer be monolithic and will be distributed. Our next generalizations would be along making our app multi user with multiple access point with multiple groups of roommates and so on. This you can use this preview to start thinking about how to generalize your app in various ways.

Towards the end of this course, you will have a 2-week project where you can generalize in as many different ways as you can think of. Of course, you will be required to commit your generalization first, you have to state what is the generalization that you are going to do and only submit that part.

This is a good point again to pause.

With that we come to the end of the second session of week 1. Thank you. See you in the next session.