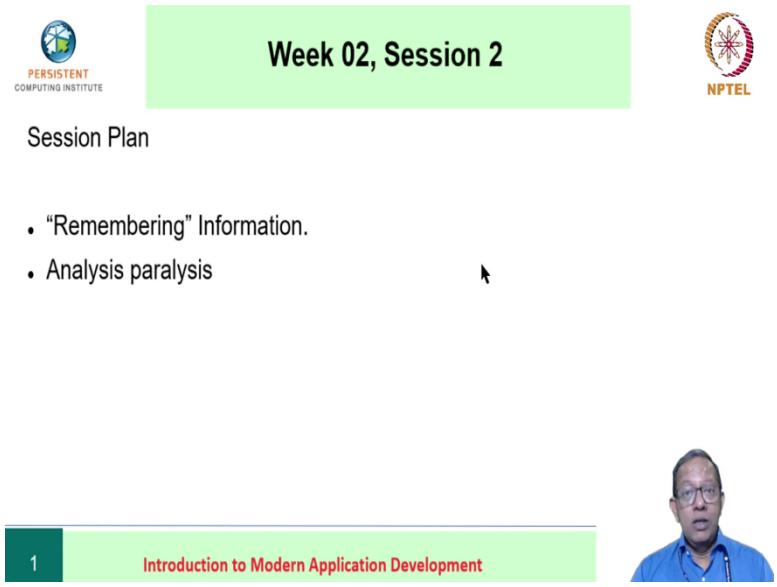
Introduction to Modern Application Development
Persistent Computer Institute

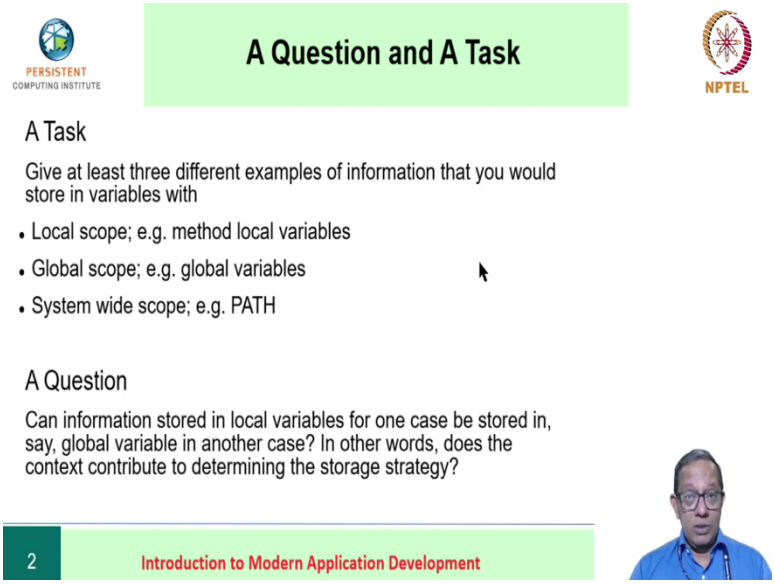
Lecture 6
Command Line - Part 2

(Refer Slide Time: 00:15)



Hello everyone, welcome to the second session of the second week of the introduction to modern applications development course. Last week we saw the basic development of a command line version of our fair share application. We saw the java code of our program and we saw how it works. It was not working perfectly in some cases, and it did crash, but nevertheless it did perform the job. We left the session by saying what is it that we need to do to remember information.
So, the plan for this session is to think about remembering information and dealing with a certain phenomenon called as **analysis paralysis**.

(Refer Slide Time: 01:06)



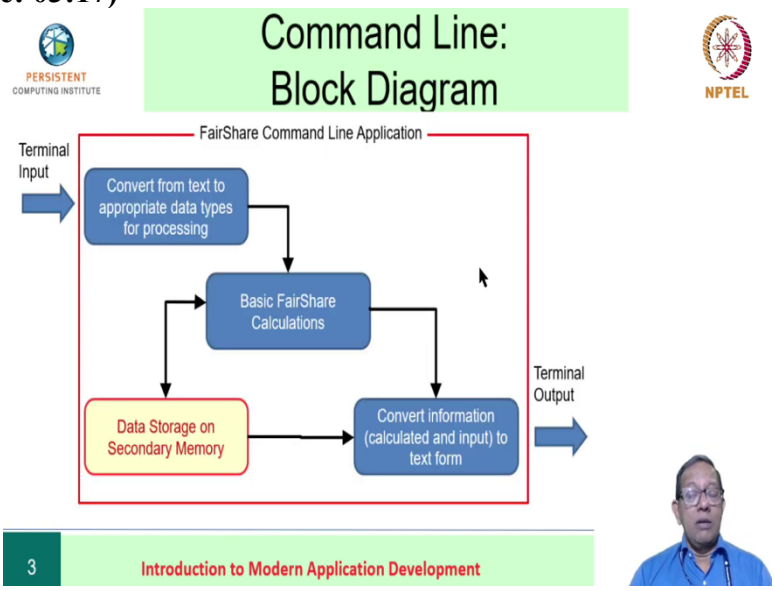
Let us begin with **a task** and **a question**. In our Java program we saw various kinds of variables:

- Global variables: e.g., the money spent array was a global variable for the program. In Java we implement this using **Class Variable**
- Local variables: e.g., the index that is used to access a particular roommate within that array can be a local variable. In fact, it often has been a local variable for the loop.

The task for you is to give at least **3 different examples** of information that you would store in variables which are *local in scope, global in scope, and system-wide in scope*. You should do this for the fair share application at hand, but you could also use other programs that you can think of.

Having done this task our next question is that is it possible that the information that you decide to store in, say, local variables for one particular program can be stored in global variables for other program? In other words, depending on the particular program, the scenario, or the problem at hand is it possible that a given piece of information may have to be stored in local variable or a global variable or maybe even a system-wide variable? Does the context contribute to determining the storage strategy? You should ponder over this question and are encouraged to discuss on the forum. It basically gives us a seed for certain ideas that we are going to explore in this session.

(Refer Slide Time: 03:17)

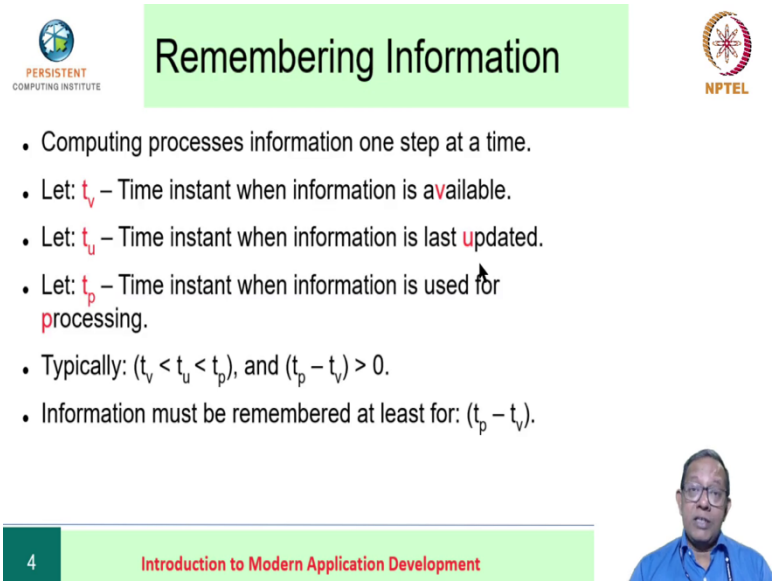


Here is the familiar block diagram of our command-line application, as indicated, in this session we are primarily going to focus on the box on your lower left that deals with data storage on secondary memory. Before we leave this recall that for us the standard input is the keyboard device and the standard output is our monitor. This standard input and standard output are typically defined by the operating system that you use.

Most operating systems will allow you to change these depending on your needs. For instance, it is possible to pick standard input from a file instead of a keyboard. Similarly, it is possible to direct the standard output to a file instead of a monitor. We suggest you learn your operating system and how your operating system allows you to change the standard output or standard input depending on your needs.

Usually there is also one more standard device called as the standard error output which is distinct from standard output which is used to flash error messages. By default, your monitor is also the standard error device.

(Refer Slide Time: 04:49)



Remembering information

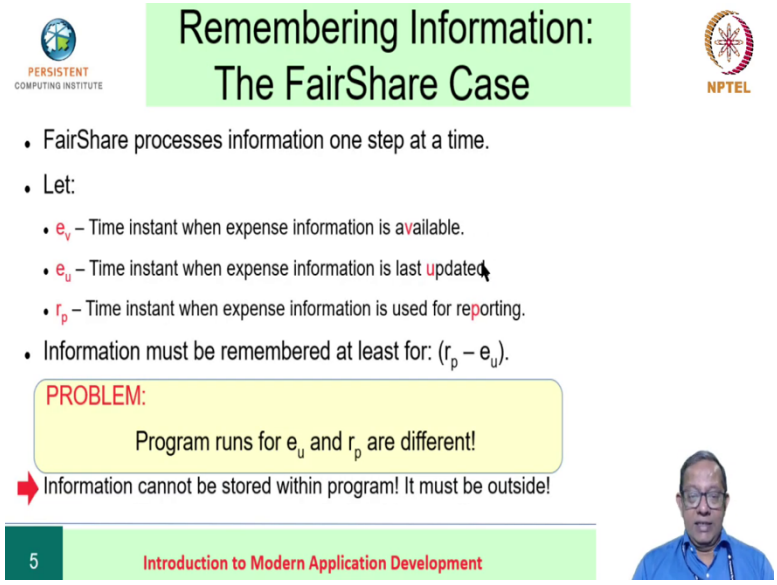
Let us continue with our fair share application. Let us try to first understand why we would need to remember information in our programs. Computing process information one step at a time. So, the entire information processing is spread across various time instants, let us label these time instants instances according to our convenience. For example, we could follow following convention:

- Let t_V denote the time instant when the time information is available.
- Similarly let t_u denote the time instant when the information is updated.
- Let t_p represent the time instant when the information is processed.

Typically, the time instant when the information is available is before the time instant when the information is updated and then which itself is before the time instant when it is used. In this case we have of course assumed that there is an update phase, but that may not be the case. You might just use the information without updating it, in that case the time instant $t_u$will not exist, but nevertheless the inequality $t_v < t_p$ will still remain true.

Also note the way we typically measure a time; we will have the difference between the time instant of processing and the time instant of availability as positive number. Simply put, there is a certain duration from the moment information becomes available and the information is used, in between this entire duration information must be remember; that is what memory in our program helped us to do.

(Refer Slide Time: 06:54)



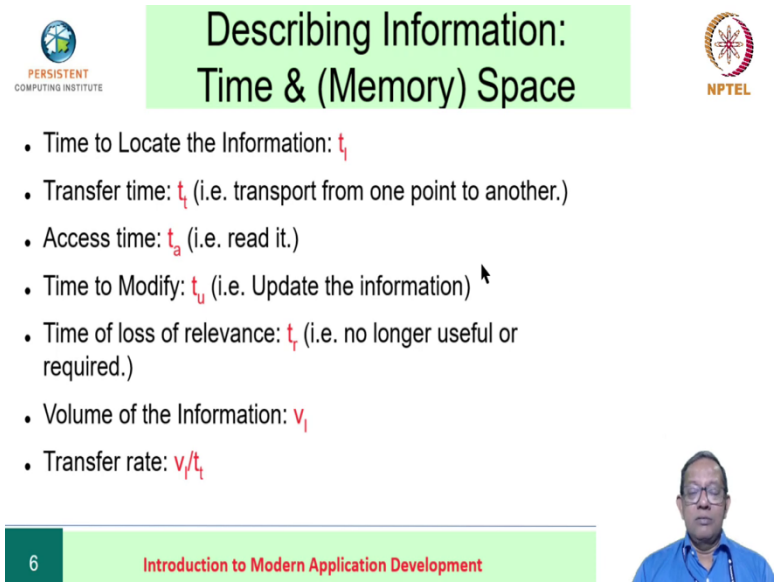
In our fair share case, here we can have following conventions:

- We will denote by e_v the instant when the expense information is available.
- Let us denote it by e_u the point of time when the expense information will be updated
- Let r_p denote the time instant when the expense information is reported.

Please note that when you report the information you are using that information; this means that information must be remembered – the expense information must be remembered from the moment it is updated to the moment it is used for reporting.

For this entire duration the information must be remembered. However, in the case of the fair share application that we saw in the previous session there is a problem. The program that is run to update the expense information is different from the program that is run when you report that information, the same program is being run in two different times and that is why information cannot be stored within the program, it has to be outside of the program because we require same information to be persistent across distinct runs of the program.

(Refer Slide Time: 08:37)



How do we describe information and its processing?

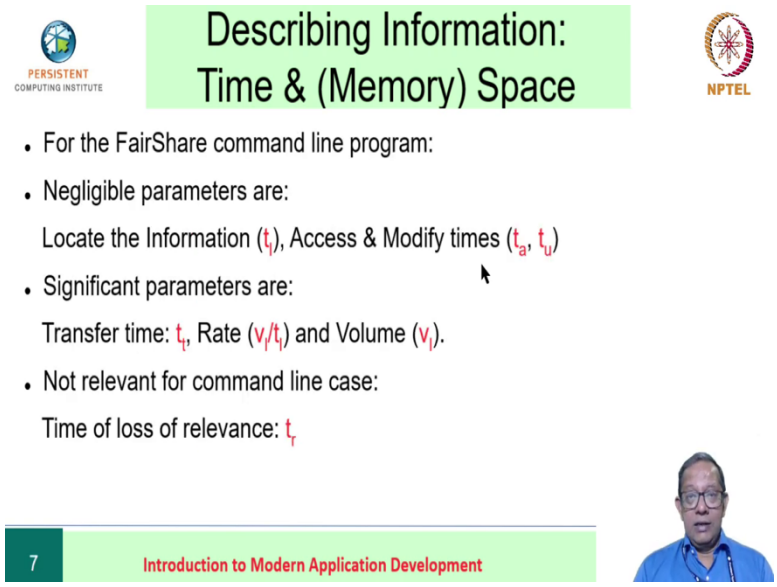
- The time it takes to locate information will be denoted by t_l.
- There is a certain time to transfer information from one point to another we will denote it by t_t.
- There is certain amount of time to access the information, which will be t_a

One might simplistically imagine that the time required to access information is the sum of time required to locate it and the time required to transport it.

There is also some time that we will denote to modify information. We will introduce one more different kind of a time: time for loss of relevance of information. The idea here is if the information is available at a certain point of time then there is a certain time, say t_r, after which the information that has been available is not useful at all. We would like to identify such a time instant also.

There is also volume of information that you might want to talk about. Finally, usually the important parameter to describe information is the transfer rate that is the volume of information the rate at which it is being transferred.

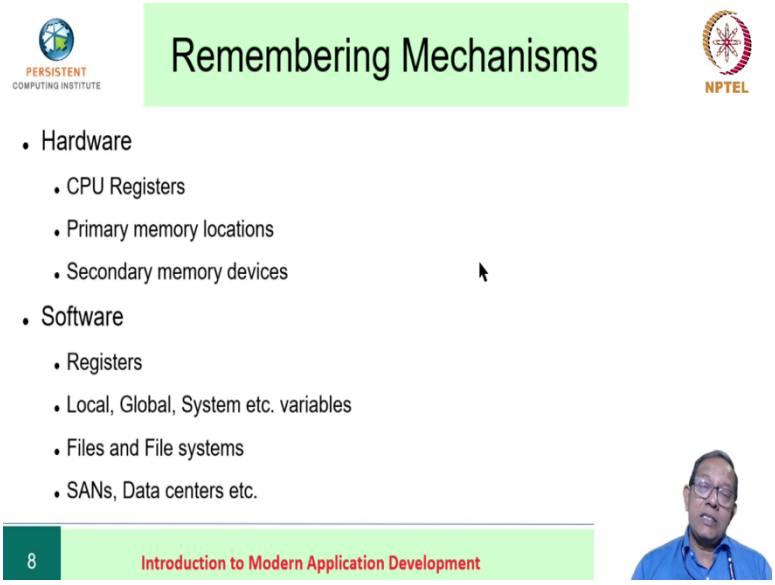
(Refer Slide Time: 10:16)



In the fair share program case, there are some parameters of describing information that are negligible. For instance, locating time or access and modify times. These times are negligible in the sense that they are very small compared to the other times of the program. The significant parameters could be the transfer time and of course the transfer rate and volume.

There could be parameters that are not relevant for this particular case. For example, the time for loss of relevance of the information is not really relevant for this case, perhaps for a later version of this fair share application particularly a web application this could be a very important consideration; if it is, we will deal with it in a future session.

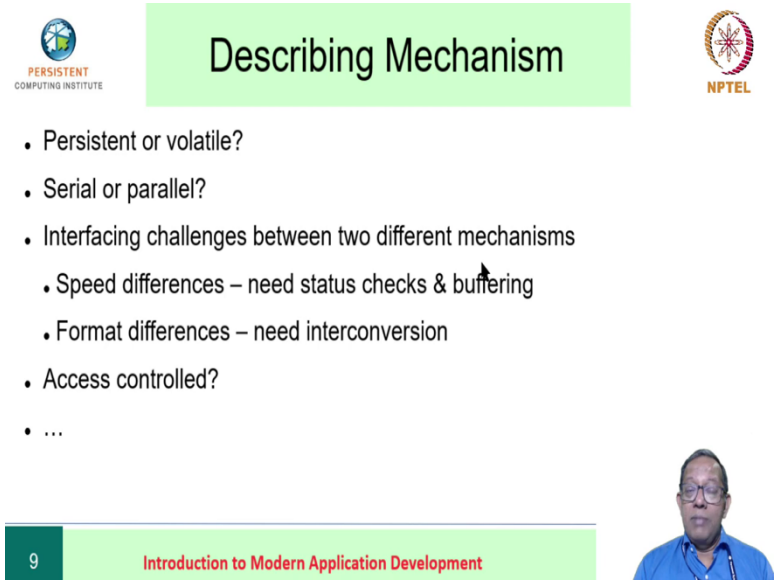
(Refer Slide Time: 11:08)



Having looked at the ideas that are required to remember information let us get a bit more concrete. Let us look at the mechanism that would be useful for remembering information. On the hardware side you have the CPU registers as one mechanism to remember information, the primary memory as well as the secondary memory are other mechanisms to remember information. From the software or a programming language side, again, we would have variables which our compiler would try to put in the CPU registers.

There are local, global, or system variables. We have files and file systems as another mechanism to remember information, and again this idea continues so that we can also have storage area networks or data centres and so on so forth.

(Refer Slide Time: 12:29)



We distinguish between the ideas required for remembering information, why we need to remember information, how do we do that, what are the real parameters that we should really be concerned with, and the mechanisms required to remember information. Given the mechanisms, the mechanisms are described by various kinds of parameters whether the mechanism is persistent or volatile. For example, the typical RAM devices that we have on our machines are volatile devices: the moment the power is switched off all the contents of the memory are erased, in contrast to the typical secondary memory devices which are persistent: in the sense that even if the power is removed the content is still preserved.

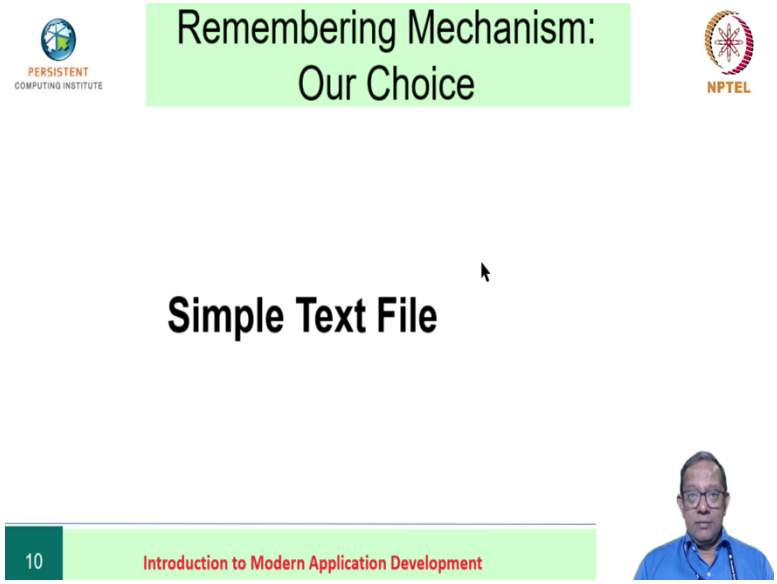
Another parameter is the transfer quantum: at any given time what is the payload of information that is being transferred. It could be one at a time in which case it is called as a **serial transfer**, or it could be a bunch of data is being transferred which is called as a **parallel transfer**.
Another parameter would be the interfacing between two devices. If we have two different mechanisms, for example a secondary memory and a primary memory, and we need to transfer exchange information between the two. Then we need to deal with the challenges of interfacing the two devices. There are two main kinds of challenge: one is the speed differences typically. For example, our primary memory is usually faster than the secondary memory. In that case whenever we exchange data, we would need to exchange data piece by piece and for every piece of exchange data we would need to check whether the data has been transferred correctly or not.

So, given that there are speed differences we would typically need status checks and buffering to perform data exchanges.

Second challenge for interfacing would be in terms of differences of format. Some kinds of mechanisms would arrange information in one format, another mechanism the same information in a different format. Recall that we talked of *file as a mechanism of remembering information*. Consider an example where the information that is to be remembered is an image in a file. Now one format of remembering that same information is JPEG format and another format of remembering the same information is a PNG format.

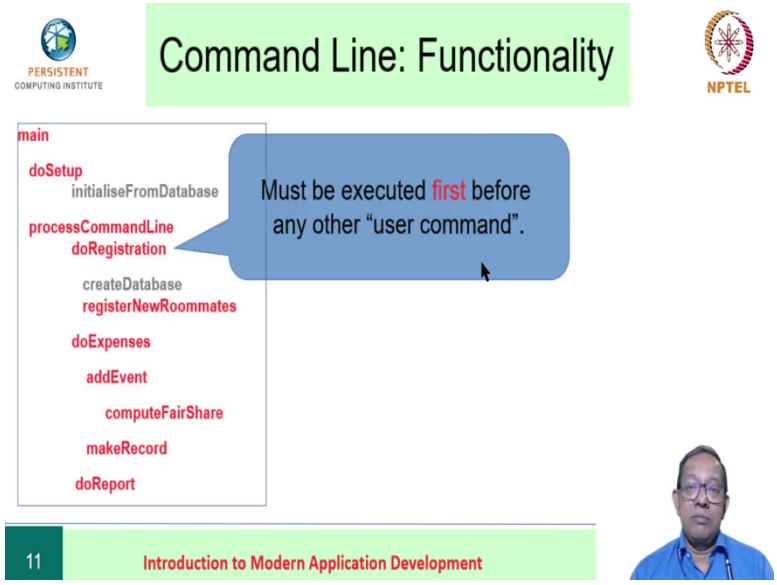
You will need inter conversions between the two. There are other parameters of describing information also, for example is it access controlled or not, you can look up all of this information from your earlier experiences or on the web.

(Refer Slide Time: 15:26)



For our fair share case command line version, we will choose to remember information using simple text files. This is a design decision that we make at this point of time, eventually later we may have to change and go in for a different mechanism for remembering information.

(Refer Slide Time: 15:52)

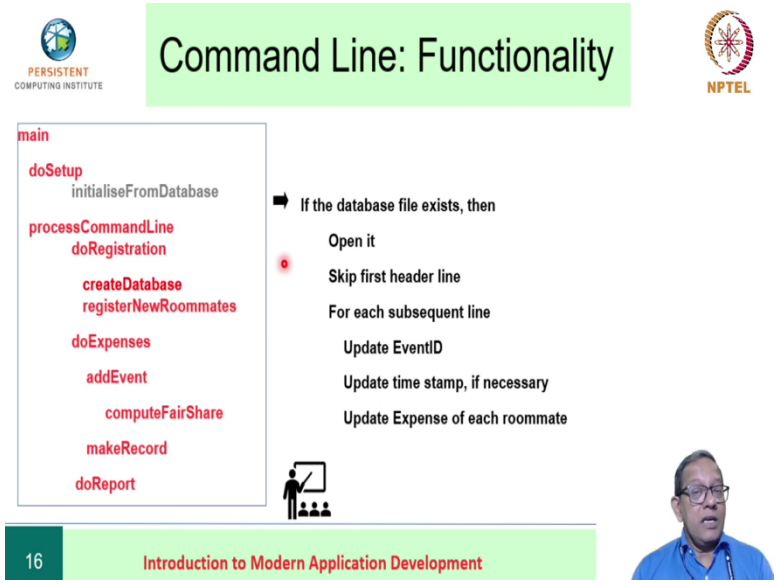


Let us now look at the command line program from a functionality point of view. In the last session we had seen this our program starts with main. The main method simply starts with the $doSetup$ and process the command line. The command line we saw was a simple dispatcher of the appropriate method to execute in response to the user command. The user commands were **register, expense,** and **report**.

Accordingly, we have 3 different methods $doRegistration$, $doExpenses$, and $doReport$. In the last session we had ignored the $initializedFromDatabase$ and $createDatabase$ methods. In this session we will now focus on these. Before we proceed to that let us realize a few aspects about our program. The registration method must be the first user command to be executed, that is how a group of roommates start recording their exchanges.

The due expense method must be executed for every event that the roommates share together, and finally the do report must be executed for each roommate. So, the registration method is a method that gets the entire interaction started, the expenses will record every event, and the reporting will be done for every roommate.

(Refer Slide Time: 17:46)



What would *createDatabase* mean in the *doRegistration* operation. The "create database" method will forget all the previous information, if it has any. This is done by simply deleting the existing file. It will then start afresh by creating and opening an empty file. The file will keep on recording the interactions in future. The file is external to the program. Recall, that we have realized that we must remember information outside of the program. A file exactly does that.

What would creating a fresh new information piece of information mean? In our case we have chosen to create a header line with following labels (as shown in slides):

- we will have an event number
- a timestamp
- for each roommate his or her name
- followed by the last column "invariant"

Once this header line is constructed within the create database function it is then written on to the file, and the file is closed.

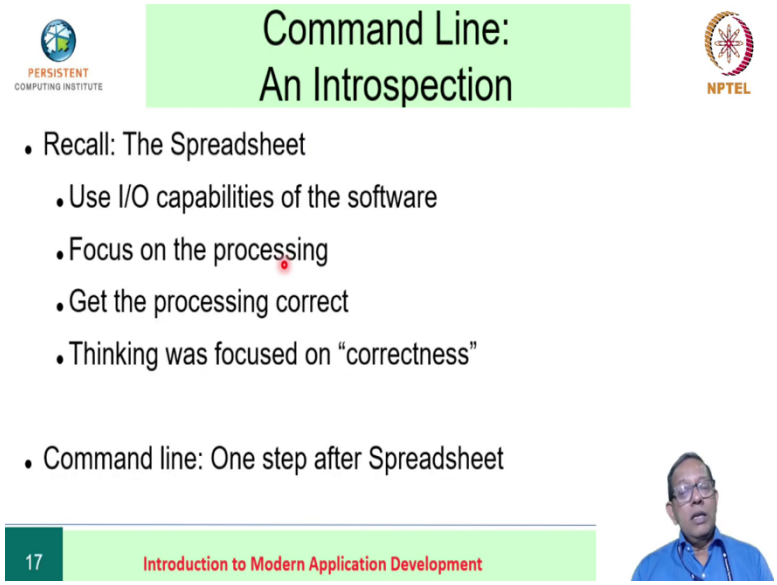
These operations of creating a file opening a file, writing to a file, and closing the file must be done using the Java I/O methods. *initializeFromDatabase*, the intent of this function is to recall previous information. So, if the database file exists then this function will open it. It will skip the first line because that is a header line, and for every subsequent line it will read the event

on that line, any timestamp on that line, and expense information of each roommate – it will read that information off and store it for use in the program runtime.

This will be done for every subsequent line and each line represents one single event. This is how our program will recall the earlier information. All it does is it reads in the file which has the information of the previous events, store information within to its own data structures. For example, it will read the information into the money spent array.

This is a point where you might want to pause and review the material we have covered so far.

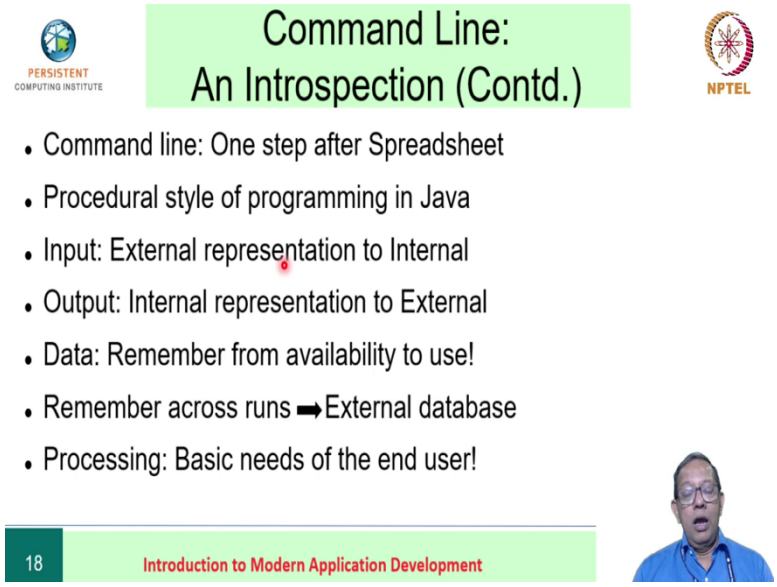
(Refer Slide Time: 20:44)



Let us now think about what has been done in the command line. Let us do this by comparing with the spreadsheet solution. The spreadsheet is an application, a software, or a program that is written by someone else and we simply use it. That program that has been written by someone else has its own input and output capabilities. In other words, it has its own ways of picking input and displaying output. Using the spreadsheet relieves us of dealing with input/output because that has been taken care of by the program, instead we can focus completely on the processing.

We use this to get our ideas about processing correct, in fact, we spend some time thinking about the correctness of our solution. The command line is one step after the spreadsheet, but that step has quite a number of points to consider.

(Refer Slide Time: 22:08)



In the first place, recall that our program was completely within a single file.

(Refer Slide Time: 22:21)

Let us have a look at that; here you see the entire program in a single class file, the class $fairShare$. It has a set of global variables, a set of methods that help us to do the various operations that we need to do in the program. And finally, at the end of the file the class code finishes. Actually, this is a very procedural style of programming, although it has been done in

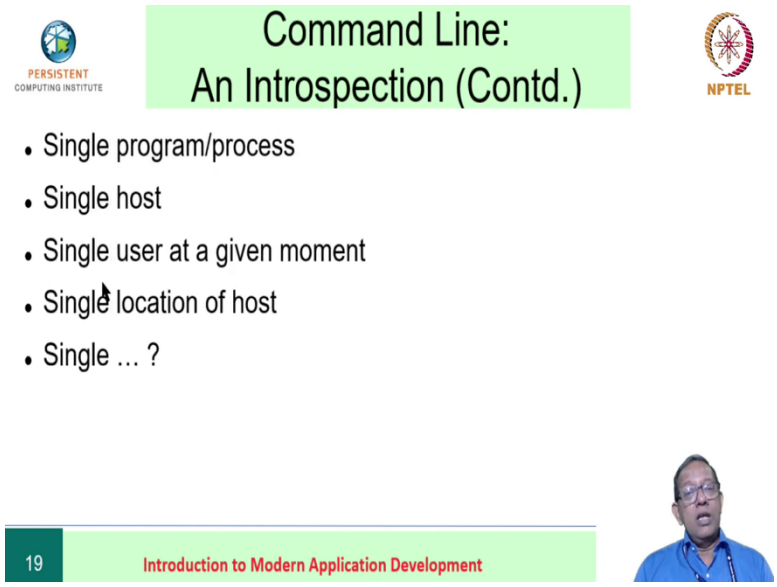
Java. Java supports object-oriented programming but we are just illustrating procedural programming in Java. In a short while we will also look at an object-oriented version of the same program.

We have also used the command line program to take care of the input and output. In particular, we saw that during input there is an external representation of the information which has to be converted into an internal representation for use, and similarly during output whatever is the internal representation of the data we have to convert it to an external form for display on the output devices.

Our command line version was our first exposure to this kind of a conversion between external and internal representations for input or output as the case may be. We also saw that the command line version required us to remember data information. And in particular this information had to be remembered outside the program, that is because we needed that information across different runs of the program.

We also saw the basic processing that is there which is of course the most fundamental part of the entire story, that fortunately we have already done in the spreadsheet and we just use the same ideas in our programs, and hence we know that our processing is correct. So, we are dealing with input and output, but in a spreadsheet, we had used a program written by somebody else and now we are using our own program.

(Refer Slide Time: 25:26)

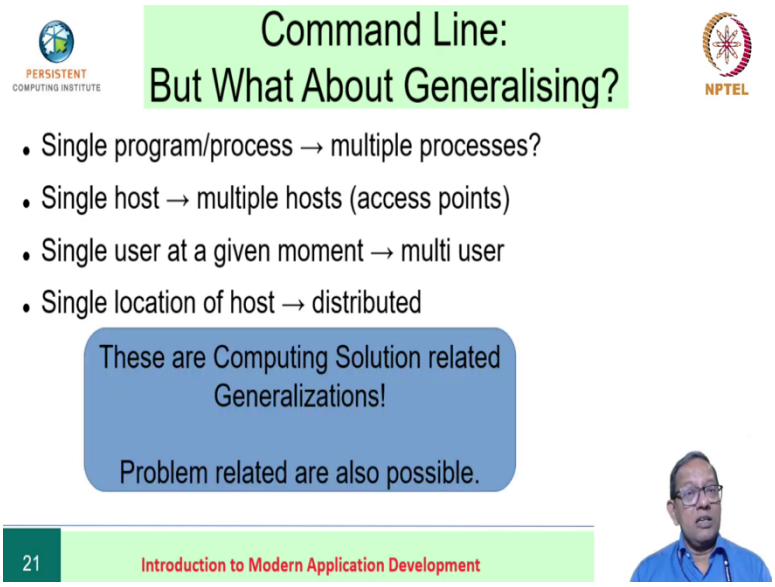


There are a few more characteristics to this program, one of them is that it is a single program that has a single process. It is on a one single computer that we call as the **host machine**. One and only one user at any given point of time actually uses this program.

Some of you may wonder our operating systems are multi-user, hence all we require is that a different user logs in. But note that the same machine is used by different people but at different times: it is in that sense that our program is being used by a single user at any given point of time.

It is also located on the single host, which means that if the roommates have to use this program, they have to come near the machine on which the program exists and use it. So, as users of these programs, in order to access the service provided by this program, the users must come towards the service. We can imagine that the service is stationary on the machine and users must come to it in order to access the service.

(Refer Slide Time: 27:10)

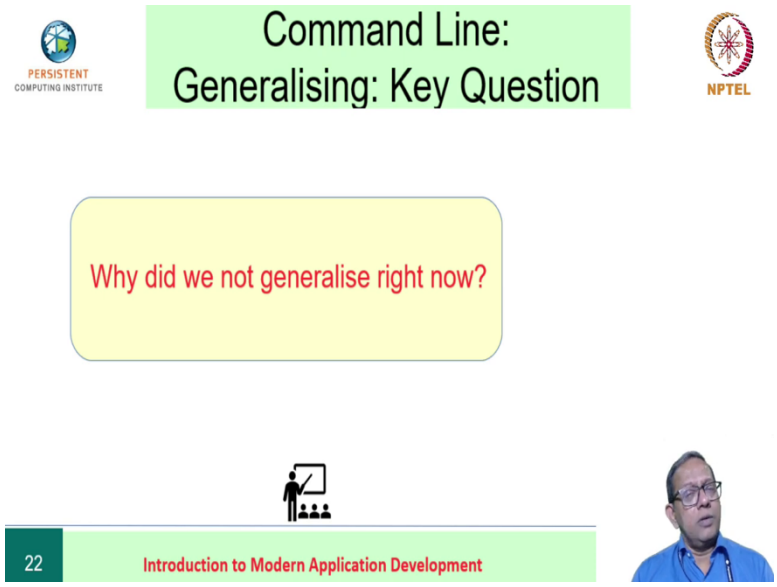


Quite naturally there are some obvious generalizations, we had talked about this in the beginning of this course. In fact, this incremental approach is a characteristic of this course, we will shortly see why is it such a valuable and useful approach. But for now, we could take a single host, a single program system, and single process system and try to develop a multiple process system.

Suppose there are multiple processes of this program that are running simultaneously, instead of a single host we would like to have multiple hosts. What do multiple hosts allow us? If there are multiple machines on which this program is available then it means that the roommates can access that service from different, different points and not a single point. At any given point of time our program in its current form is a single user system our generalization could be making it a multi user program.

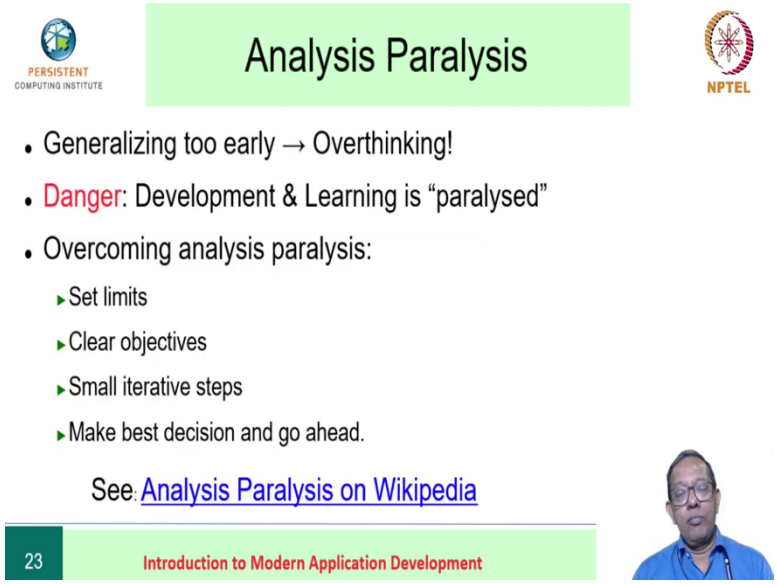
Instead of a host located a single place, if we could have multiple hosts which were distributed across the globe anywhere then we could have a distributed version of this program. There are many such generalizations possible, however please note that these are computing solution related generalizations. There are of course problem related generalizations also, but for the moment we will not worry about that second dimension of generalizing.

(Refer Slide Time: 29:02)



The question that we want to ask now is why did we not generalize right away from the spreadsheet once we realize to what is the nature of our solution? Before proceeding, we suggest you pause the video and think for yourself about why this could be – *Why is it not useful to generalize now?*

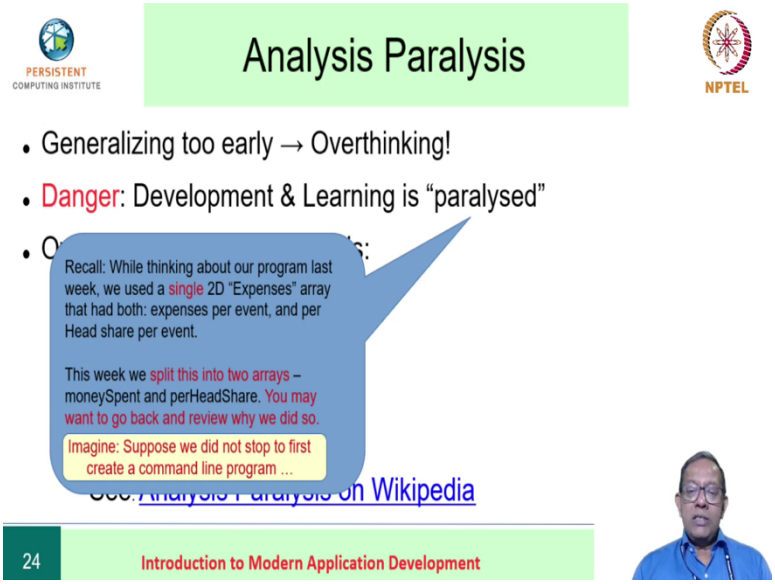
(Refer Slide Time: 29:37)



The key phenomenon that is behind the reason why we did not generalize right away is called as **analysis paralysis**. It basically is means that when we generalize too early, we over think a lot.

What is a problem with over thinking? It actually decelerates development as well as learning, in fact, decelerates is a very soft way, *it actually paralyzes*.

(Refer Slide Time: 30:15)

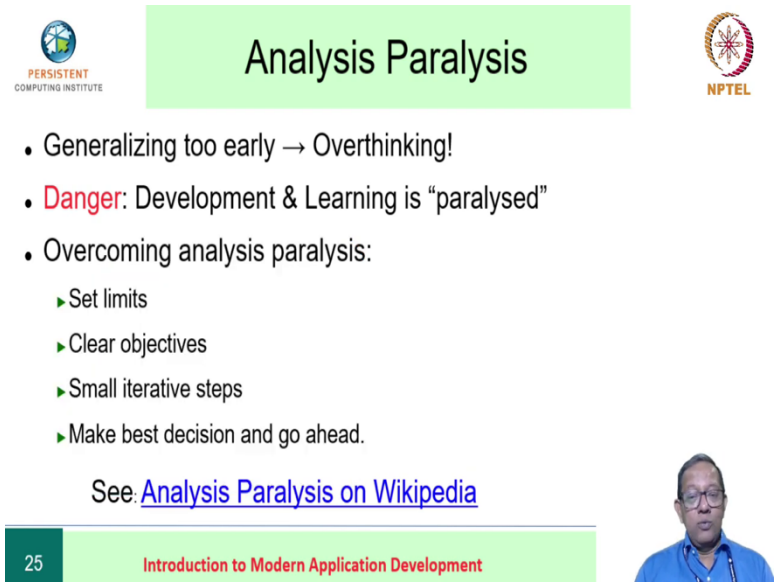


Here is an example that had occurred earlier when we actually thought about our program, we imagined a two-dimensional array "expenses". It was a single array that had both pieces of information, the expenses as well as the per head share. We had created this array given our spreadsheet-based solution. But as we developed the command line version of our program, we split this into two arrays money spent and per head share.

Perhaps it would be good idea to go back and visit that lecture where we discussed why we did that split. But now imagine, suppose we did not stop to first create a command line program, when we created the expenses array while thinking about the program of our fair share application, if we did not stop to do a command line program then we would have simply kept on generalizing to whatever way is possible.

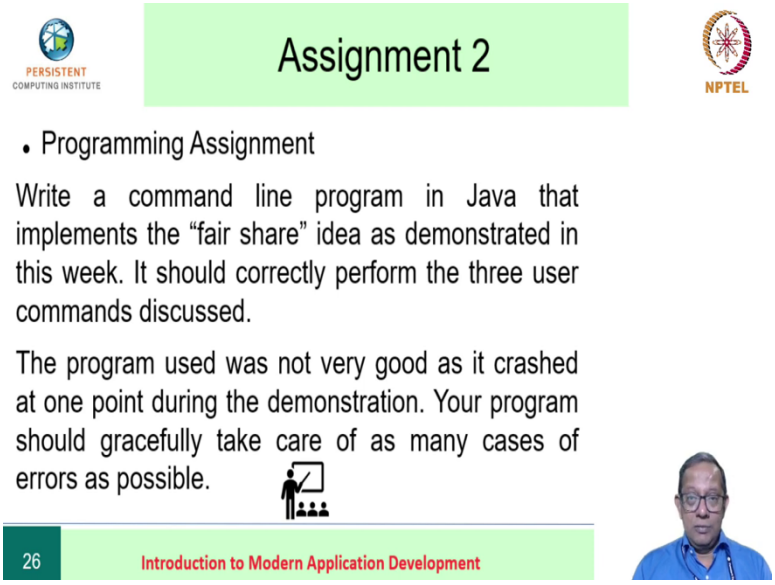
Doing the command line version actually allowed us to stop, focus, and understand what is really required, implement it, and realize that during implementation the things that can/should be changed. We learn some aspects during the actual development, and this helps us to incrementally develop our solution well.

(Refer Slide Time: 32:13)



In fact, some of the suggested ways of overcoming analysis paralysis is to set limits, have clear objectives, take small iterative steps, make the best decision right now and go ahead. If you notice, we have actually followed exactly these suggestions not only in actually developing the program but in designing this entire course. *The Wikipedia article on analysis paralysis is quite an interesting one and we suggest you have a look at it.*

(Refer Slide Time: 32:57)



At the end of this session now let us look at the programming assignment that we need to do in the coming week. In this week we are illustrated the command line version of the solution, you are to implement that same command line version in Java. It should correctly perform the three user commands that we have discussed. The 3 user commands are **register, expense,** and **report.**

The command line version has had some assumptions when it was done. For example, at any event one and only one roommate pays; you need not generalize beyond this. We also suggest that you write your program such that it does not crash in ways that happened with our program during the demonstration in lecture. A program with such a behaviour (i.e., doesn't crashes usually) is said to be stable. Having a stable program is a required step towards creating industry strength software.

Your program should gracefully take care of as many cases of errors as possible. In our demonstration program we had not taken care of a one case which is why it very explicitly crashed, it complained; this shows that the program we used was not stable.
Your assignment program must be stable; it should take care of as many errors as possible.

Thank you very much for your attention see you in the next session.