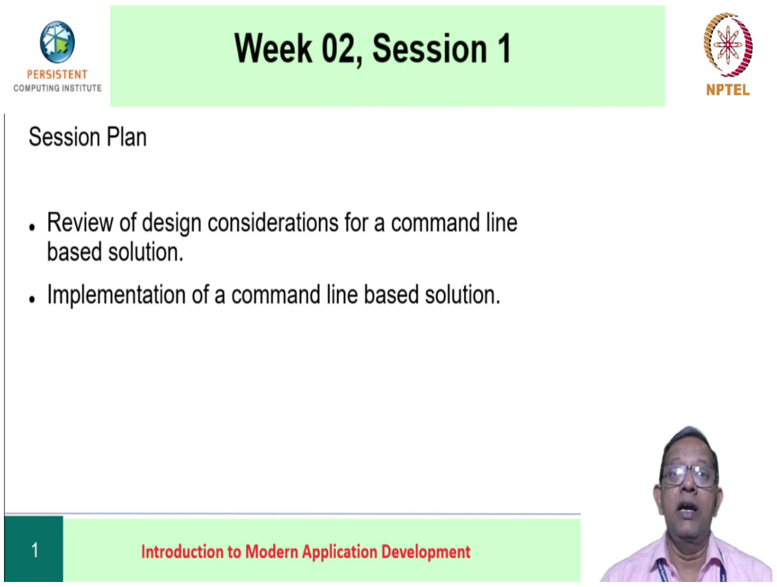
Introduction to Modern Application Development
Persistent Computer Institute

Session – 1
Command Line - Part 1

(Refer Slide Time: 00:17)



Week 02, Session 1

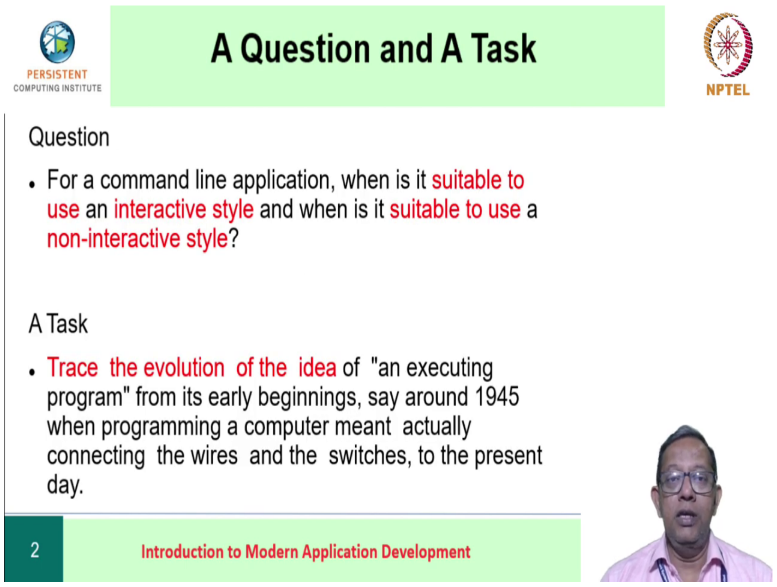
Session Plan

- Review of design considerations for a command line based solution.
- Implementation of a command line based solution.

1 Introduction to Modern Application Development

Hello everyone, welcome to the first session of the second week of the introduction to modern application development course. Let us begin by reviewing the design considerations for a command-line application that we have seen previously. This will be a more intense review, and after that we will go on to look at the implementation of the actual command-line solution. This of course will be an illustrative example; it is an implementation that is meant to illustrate how the command line should work.

(Refer Slide Time: 00:54)



A Question and A Task

Question

- For a command line application, when is it suitable to use an interactive style and when is it suitable to use a non-interactive style?

A Task

- Trace the evolution of the idea of "an executing program" from its early beginnings, say around 1945 when programming a computer meant actually connecting the wires and the switches, to the present day.

2 Introduction to Modern Application Development

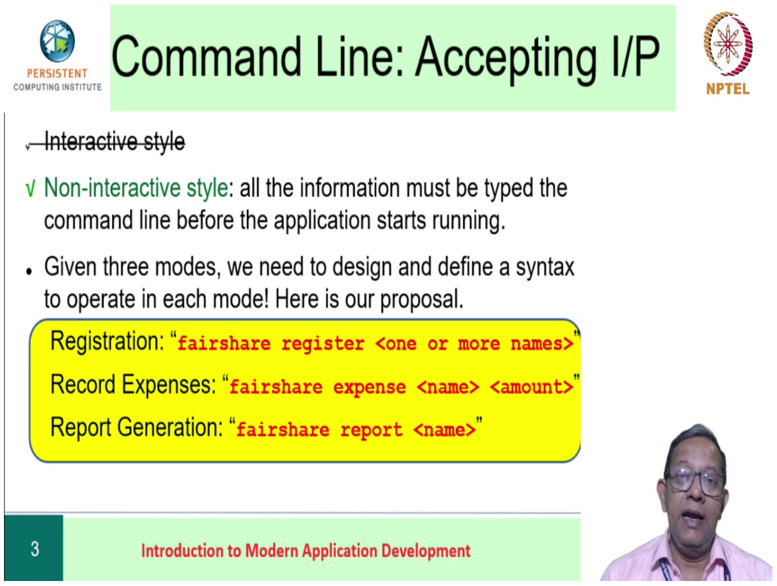
Let us start with a question, **when do you think it is suitable, for a command line application, to use an interactive style, or, when is it suitable to use a non interactive style?** We have seen what is meant by an interactive approach versus a non interactive approach for a command line. To recap, an interactive approach is the one when the information required for the program is made available after the program starts running.

In contrast in a non interactive application all the information that is required for execution must be supplied before the program begins execution. You should write down your own lists of points about the advantages and disadvantages, or whatever you think about interactive style versus non interactive style. Further we encourage you to discuss this on the forums for this course. Here is another task for you: **trace the evolution of the idea of an executing program from its early beginnings.**

Computing machines have come up since around 1945, but in those days programming a computer meant connecting the wires and switches. So, the idea of an executing program was quite different in those days. Over the decades the idea has evolved to the present day. We encourage you to look up this idea, trace it up, maybe you could divide the entire period into decades, say, 1945- 55, 1955-65, 1965-75 and so on until the present day.

This would be an interesting idea in itself and will give us some foundation to understand the way this course has been designed.

(Refer Slide Time: 03:42)



Command Line: Accepting I/P

- ~~Interactive style~~
- √ Non-interactive style: all the information must be typed the command line before the application starts running.
- Given three modes, we need to design and define a syntax to operate in each mode! Here is our proposal.

Registration: "`fairshare register <one or more names>`"
Record Expenses: "`fairshare expense <name> <amount>`"
Report Generation: "`fairshare report <name>`"

3 Introduction to Modern Application Development

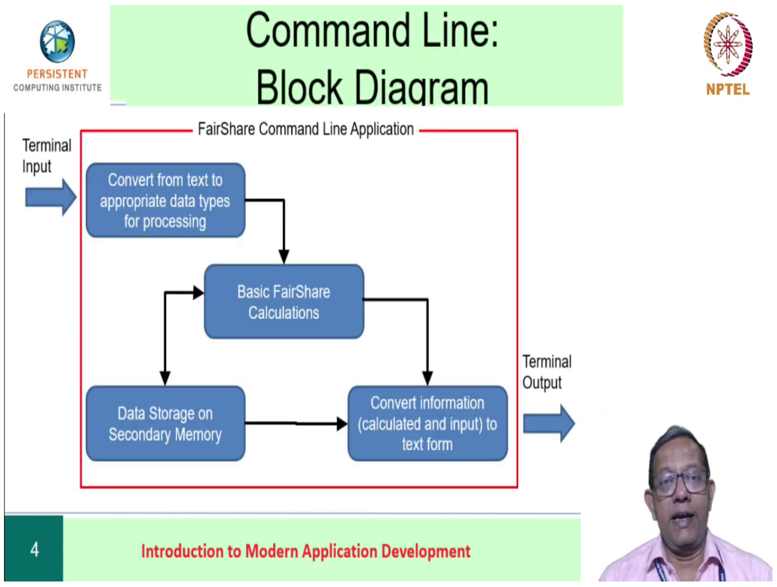
Professor Sane has used a non interactive approach when he illustrated the problem of fair share. Recall that there are 3 modes in which we expect the fair share application to run. On your screen you see the slides which we had used before.

- One of the modes is registration mode in which all the roommates register their names to the application. This allows the application to recognize the correct roommates in the future. We had seen that the registration phase would be simply a loop running over all the names given on the command line.

- The main workhorse of the application of course is the expenses recording phase in which we record the expenses for an event by roommates. The expenses phase would be simply calculating the per head share given amount of for each roommate.

- Finally, the report generation would simply involve accepting the name of the roommate whose report is to be generated. Recall that this is a design decision for us, we have decided that we would be generating report for one and only one roommate. There are other choices that we had seen before but we choose not to use any of them.

(Refer Slide Time: 05:57)


Command Line: Block Diagram

PERSISTENT COMPUTING INSTITUTE
NPTEL
FairShare Command Line Application
Terminal Input
Convert from text to appropriate data types for processing
Basic FairShare Calculations
Data Storage on Secondary Memory
Convert information (calculated and input) to text form
Terminal Output
4
Introduction to Modern Application Development

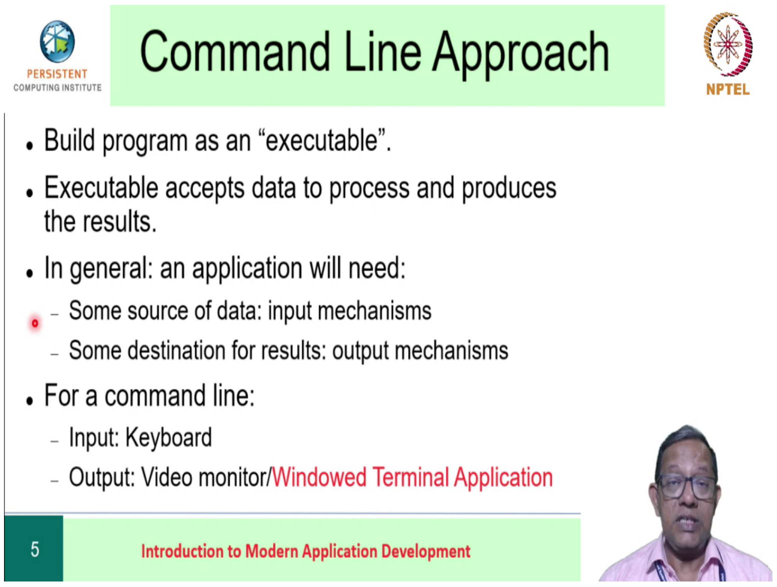
Here is the block diagram of our proposed command-line application. The red outline encloses the functionality of our application. The input comes from a terminal, for us it is the keyboard. The output is on the video monitor. The block diagrams represent the functionality that we expect that we require in the application. We had seen that one phase would be to convert all the input to appropriate data types within the programming language and store it appropriately.

Once these are available as suitable data types in our program, then we can perform further processing. We had also seen the processing that would be required the basic calculations the idea has been fairly simple. In fact, we have also Illustrated it using a spreadsheet. You could also try it out just using pencil and paper. The simplicity of the solution also allowed us to think about the correctness.

Our reasoning tells us that our approach will work whatever be the number of roommates, at least in principle, it is guaranteed to work. In practice of course we will have limitations because of implementation and many other restrictions, but our algorithmic approach is correct. Finally, the information that has been calculated or whatever information is required has to be displayed on the output device, that would also require conversion from the internal form within the language and what is required for the output device.

Remember that we had suggested that for input as well as the output conversion parts we should try to use the facilities that are offered to us by the programming language, in our case it is Java. So, we would be using the facilities for input and output conversions that Java provides. Finally, there is one more block over here, the data storage on secondary memory, we will come to that a little later.

(Refer Slide Time: 08:57)



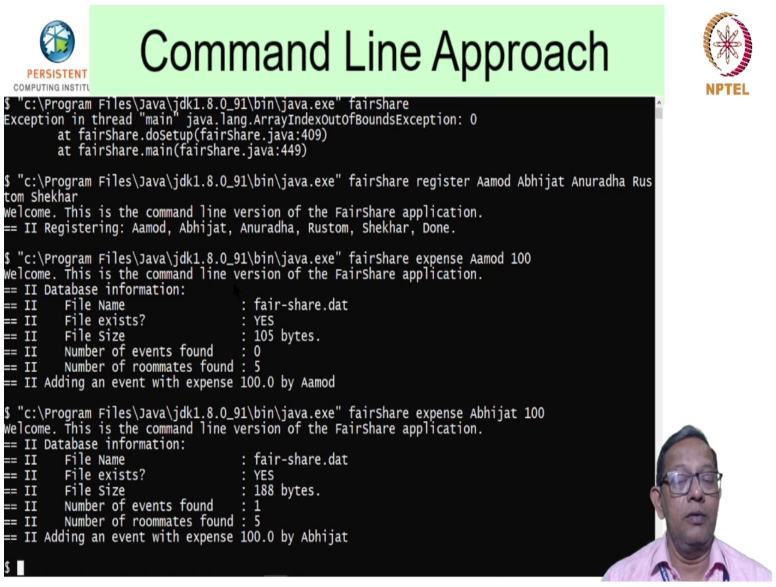
Command Line Approach

- Build program as an "executable".
- Executable accepts data to process and produces the results.
- In general: an application will need:
  - Some source of data: input mechanisms
  - Some destination for results: output mechanisms
- For a command line:
  - Input: Keyboard
  - Output: Video monitor/Windowed Terminal Application

5 Introduction to Modern Application Development

This slide helps us to recall the ideas behind the command line approach. Let us go ahead and actually look at a terminal interaction about this.

(Refer Slide Time: 09:18)



Command Line Approach

```
$ "c:\Program Files\Java\jdk1.8.0_91\bin\java.exe" fairShare
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 0
        at fairShare.doSetup(fairShare.java:409)
        at fairShare.main(fairShare.java:449)

$ "c:\Program Files\Java\jdk1.8.0_91\bin\java.exe" fairShare register Aamod Abhijat Anuradha Rus
tom Shekhar
Welcome. This is the command line version of the FairShare application.
== II Registering: Aamod, Abhijat, Anuradha, Rustom, Shekhar, Done.

$ "c:\Program Files\Java\jdk1.8.0_91\bin\java.exe" fairShare expense Aamod 100
Welcome. This is the command line version of the FairShare application.
== II Database information:
== II    File Name                : fair-share.dat
== II    File exists?             : YES
== II    File Size                : 105 bytes.
== II    Number of events found    : 0
== II    Number of roommates found : 5
== II Adding an event with expense 100.0 by Aamod

$ "c:\Program Files\Java\jdk1.8.0_91\bin\java.exe" fairShare expense Abhijat 100
Welcome. This is the command line version of the FairShare application.
== II Database information:
== II    File Name                : fair-share.dat
== II    File exists?             : YES
== II    File Size                : 188 bytes.
== II    Number of events found    : 1
== II    Number of roommates found : 5
== II Adding an event with expense 100.0 by Abhijat

$
```

Here is a standard Windows terminal whose prompt we have set as dollar and the cursor is blinking to show us that it is ready to take a command – the cursor blinks at the prompt. The way to build this program is to use the Java compiler and compile this Java program to a class file. Normally the `javac`, the Java compiler command would be available as simply as the '`javac`' command because it would be in your path (we set it using the environment variables). But we chose to illustrate the entire command line for your clarity. If the compilation is successful then all that happens is that we arrive back at the prompt.

In case there are errors then the compiler prints those errors before giving us the next prompt for the next command. Once the program successfully compiles, we can of course run it, but we need to run it on the Java Virtual Machine (JVM). This is done using the `java` command.

In the lecture we saw an error that which shows us that the command that we wrote for the fair share application was not fully complete. Also, it gives an error to the end user. In fact, a well written program should not show such an error instead it should tell the user and offer a brief help message, inform guide to the user about the correct way of using this program… etc.

Let us register few roommates. I will use the names of my colleagues at persistent computing Institute. We are registering 5 friends, and here is what our program tells us, it has registered 5 roommates.

We can now add the expenses for various events that all of these people indulge in. Let us suppose that Aamod is the first one to pay say 100 rupees for some event that has occurred that all 5 of them have shared, we record this using the command:

```
$ java fairShare expense Aamod 100
```

The command tells us that there is some database in a file called `fair-share.dat`. It tells us that this file exists, how many bytes it is consuming so far, the number of events found is 0, the number of roommates found is 5.

Finally, it tells us that the event with expense 100 by Aamod has also been added. Let us add other expenses by other members of this group. Let us suppose in the second event I spend Rs. 100 and contribute Rs. 100 for the entire group of roommates. Similar to the display above our program again tells us the name of the file of the database, whether it exists or not, but notice that the number the size – the number of bytes in the file have changed.

That is because new information has been added in this run of the program. It is important to realize that the run of the program for updating Aamod's expense is different and completely disconnected from the run in which my expenses have been recorded. These are two different runs of the same program. Let us go ahead and add the expenses by others. Let us say Anuradha spends 100 rupees again and our program stores that information using the command:

```
$ java fairShare expense Anuradha 100
```

Then Rustom pays; we have a few interactions with the program that recorded expenses. If you observe the number of events found you will realize that the information that is counted actually starts from 0, and that is why when the first expense was numbered 0. A well-written program should really be informing that this that the first event would be 1 not 0. Most programmers are comfortable with starting from 0 but typical users would be very confused if your program said that the number of events is 0.

They would be very confused because they are using this application to say that the first event when our most paid 100 is to be recorded and this program is saying it is 0. Only after a couple of runs to be realized that our program has been written such that the events are numbered from 0 and not 1. Let us check this out let us have Shekhar contribute 100 rupees again, this is the fifth event because we have actually had 100-rupee contributions for 5 events by each individual member of persistent computing Institute.

The output on the terminal shows that the number of events is shown to be 4, while the number of roommates is correctly displayed as 5. We should really be writing programs such that our users understand the results and the information that is being displayed. What we have done so far is to add 5 events where 5 members of the group have equally contributed this means that after 5 events nobody really owes anything to others. Let us check if our program really reports exactly that case.

Note: Run the program (or see slides) to check that the program indeed returns the results mentioned here.

So, if for example I ask our program to tell me what is the final amount that I owe it says 0, which is correct. If I instead of me if I ask about say amount again that is 0. If I ask for Anuradha, then that is 0 too. But let us now suppose that we have the 6th event where Shekhar spends 200 rupees for the group. Let us now again ask how much does Abhijat owe – it should be 40 rupees. Awesome, the answer that is shown is exactly 40.

But again, if you notice very carefully there is a slight problem, just like the events that were numbered from 0 would be confusing to the user, a number like -40 is confusing particularly when the comment before that man number says the amount owed to others is – 40. Let me illustrate the problem by looking at Shaker's report. Shaker has contributed more so he should actually be receiving from others.

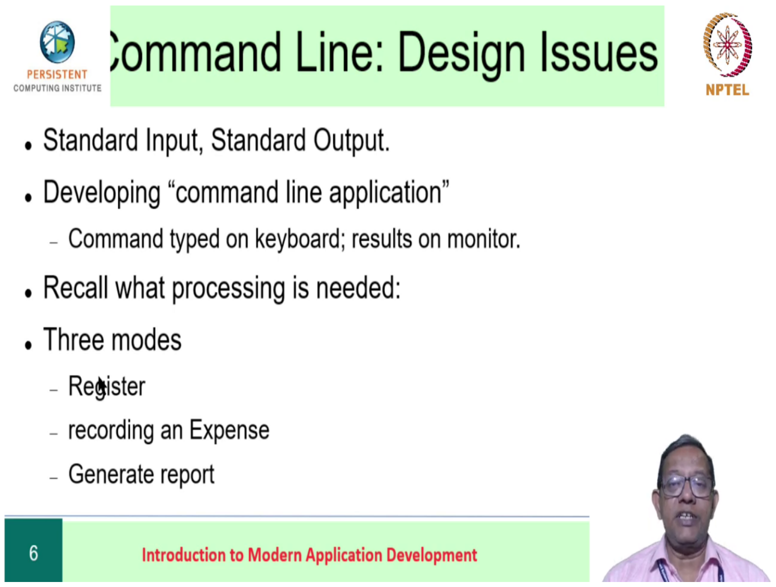
Note the way the program responds it says amount to be received from others is 160 rupees this makes sense to the end-user. However, if the end user sees that the amount owed to others is -40, then that can be confusing. What we are talking here is we need to be displaying information in a consistent form for the end-user. Recall the reason why it is minus forty is because in our program within the – within the program we chose to represent the amount that is to be paid by negative numbers and amount to be received by positive numbers.

But this is really an internal affair, something that is interval to the program. This ideally should not have anything to do with the user, either we should inform the user that the negative numbers mean that you owe money and positive numbers means that you will receive money. We should inform the user about this convention before we even display like this, or we should convert the negative numbers to positive numbers and then we can say the amount owed to the others is 40 rupees. For the convenience of the user, we should actually display the result with proper information on how the result is supposed to be interpreted. There is a certain element of design the way the program output should interact with the user even if it is a non-interactive program.

Well this illustrates the way in which our program runs. Instead of looking at the slides we chose to show you how exactly the application would be built – it is a simple command that combines a Java program to its corresponding executable format called as a **class file**. We show by explicitly running the program that the executing program accepts some data to process and produces the result.

It also illustrates the actual ideas in the previous sessions where we talked about the fact that for any program some input would be required, and there are many mechanisms for accepting the input. Similarly, a program would generate some output and there would be a number of mechanisms for displaying the output. Here, for the command line version, we chose that the input device would be the keyboard, which is what we did by actually typing the commands on the keyboard. And the output device was a text-based terminal the video your video monitor actually in our case it was a Windows terminal application. You can look up look that application up by searching for command prompt on your Windows system.

(Refer Slide Time: 25:12)



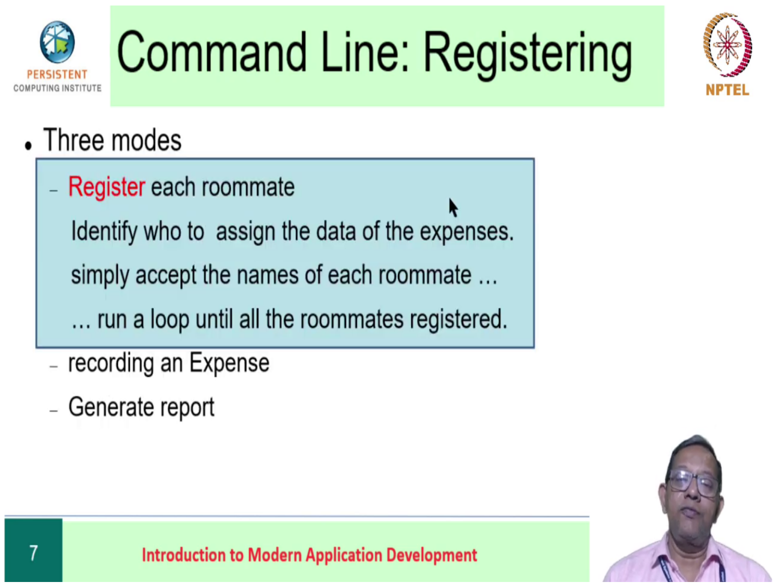
Command Line: Design Issues

- Standard Input, Standard Output.
- Developing "command line application"
  - Command typed on keyboard; results on monitor.
- Recall what processing is needed:
- Three modes
  - Register
  - recording an Expense
  - Generate report

6 Introduction to Modern Application Development

This slide summarizes what we have seen so far. We will now go ahead and look at the actual Java program and try to connect it up with the various block diagrams that we had seen before.

(Refer Slide Time: 25:28)



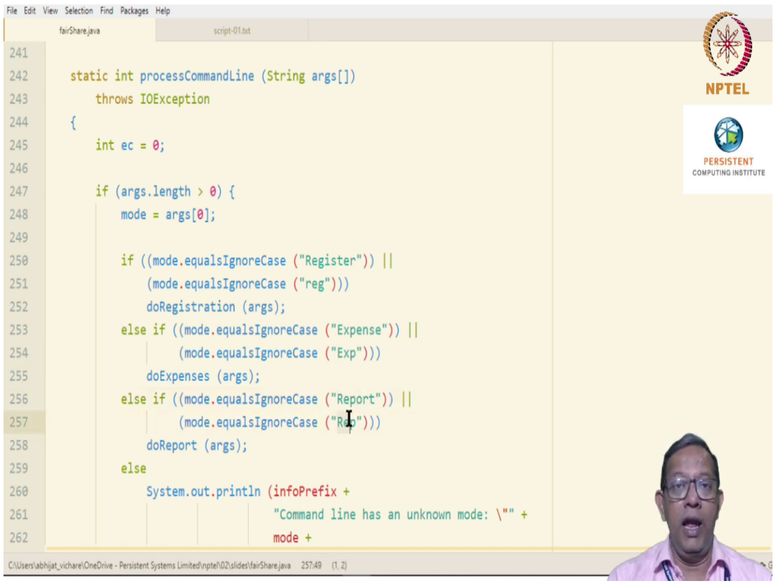
Command Line: Registering

- Three modes
  - Register each roommate
    Identify who to assign the data of the expenses.
    simply accept the names of each roommate ...
    ... run a loop until all the roommates registered.
  - recording an Expense
  - Generate report

7 Introduction to Modern Application Development

Let us try to look at the first part of the story, registering the roommates. We saw that registering the roommates is simply running a loop until all the roommates are registered, this was the idea that we had thought. Let us look at the source code of the command that we had just seen.

(Refer Slide Time: 25:54)



```
    static int processCommandLine (String args[])
        throws IOException
    {
        int ec = 0;

        if (args.length > 0) {
            mode = args[0];

            if ((mode.equalsIgnoreCase ("Register")) ||
                (mode.equalsIgnoreCase ("reg")))
                doRegistration (args);
            else if ((mode.equalsIgnoreCase ("Expense")) ||
                    (mode.equalsIgnoreCase ("Exp")))
                doExpenses (args);
            else if ((mode.equalsIgnoreCase ("Report")) ||
                    (mode.equalsIgnoreCase ("Rep")))
                doReport (args);
            else
                System.out.println (infoPrefix +
                                    "Command line has an unknown mode: \"" +
                                    mode +
```

Instead of using the terminal it might be a good idea to introduce an editor for this purpose. Here is our Java source code of the program. Like a typical Java program, it starts by doing some imports of Java functionalities that we would need, then there is a single class, class fair share and the entire program is contained within this class. As you very well know, a Java program starts with a static main – a public method, static method called main which is here in this. It is declared as: `public static void main (String[] args)`.

In this has only two parts do the setup and process command-line. Let us go to processing the command line. As you can see in the slides that the process of command line function is pretty small one, in fact the only thing it does is it looks at whether the 'args' variable has the value register or reg, or has the value expense or exp, or has the value report or rep, (while ignoring cases), and depending on the value it calls the corresponding method.

If the 'args' value is register or reg (ignoring case) then it calls the method to `doRegistration`. If it is expense or exp then it calls `doExpenses`, if it is report then it calls `doReport`. The command line processing function is just basically a dispatcher depending on what is there on the command line, it dispatches the corresponding method. How does it get access to the command line? It gets access to the command line through this variable this is a parameter which represents the command line that we type on our terminal.

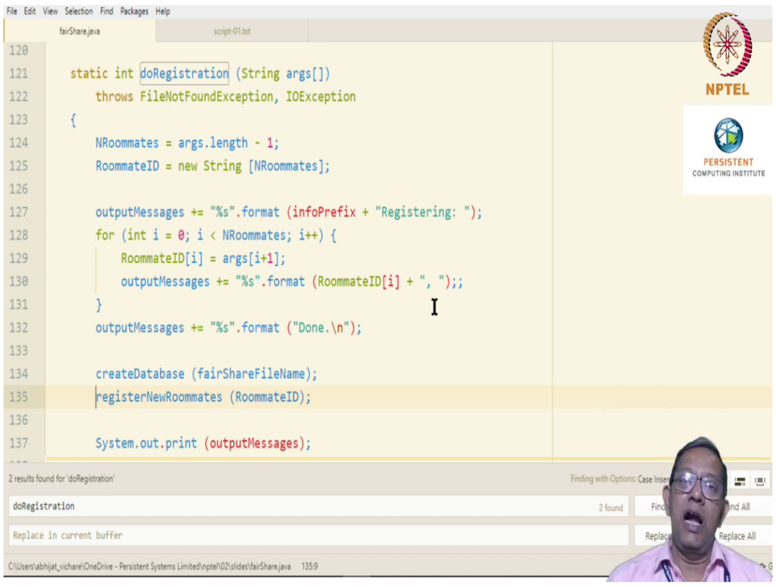
How does this command line information arrive for this process command line method well that arrives through the `main`. Notice that the `main` itself takes parameter ‘args’, this has been defined to represent the command-line arguments for a Java program; this is a Java standard. So, whatever you type on the command line within your program it is available to this function main as a parameter. And within our program we see that all that main does is passes the parameter ‘args’ to the process command line function for further processing.

For the moment we will ignore this `doSetup` function, we will eventually at some point look at what it does. Let us focus on process command line and go ahead. Having seen how the information from the command line arrives at the process command line function we see that the process command line function is just going to look at the first value of the arguments array, store it in mode, and use that. If the mode string is the same as register then dispatch this method.

Also note that the args, the command line arguments, are passed to this method. If the mode, that is the first string on the command line is expense then dispatch the `doExpenses` method and also pass the current entire command line to the do expense method also, just like main passed it to `processCommandLine()`, this function further passes the variable ‘args’ ahead to `doRegistration()`, `doExpenses()` or `doReports()` as the case may be observe.

This so process command line acts more like a dispatcher by looking at what is the operation the user command that has been requested. If the user command is registered then perform the do registration operation. If the user command is expense then perform do expense recording. If the user command is report then generate the report for the user that has been specified.

(Refer Slide Time: 32:50)


File Edit View Selection Find Packages Help
fairShare.java
script-01.txt
NPTEL
PERSISTENT COMPUTING INSTITUTE

```java
    static int doRegistration (String args[])
        throws FileNotFoundException, IOException
    {
        NRoommates = args.length - 1;
        RoommateID = new String [NRoommates];

        outputMessages += "%s".format (infoPrefix + "Registering: ");
        for (int i = 0; i < NRoommates; i++) {
            RoommateID[i] = args[i+1];
            outputMessages += "%s".format (RoommateID[i] + ", ");;
        }
        outputMessages += "%s".format ("Done.\n");

        createDatabase (fairShareFileName);
        registerNewRoommates (RoommateID);

        System.out.print (outputMessages);
```

```
2 results found for 'doRegistration'
doRegistration                                  2 found
Replace in current buffer
C:\Users\abhijat_vichare\OneDrive - Persistent Systems Limited\nptel\02\slides\fairShare.java    135:9
```

Let us look at doRegistration(). The doRegistration method first calculates the number of roommates. The number of roommates is simply the number of command-line arguments minus one. The command-line specified the users for registration as follows:

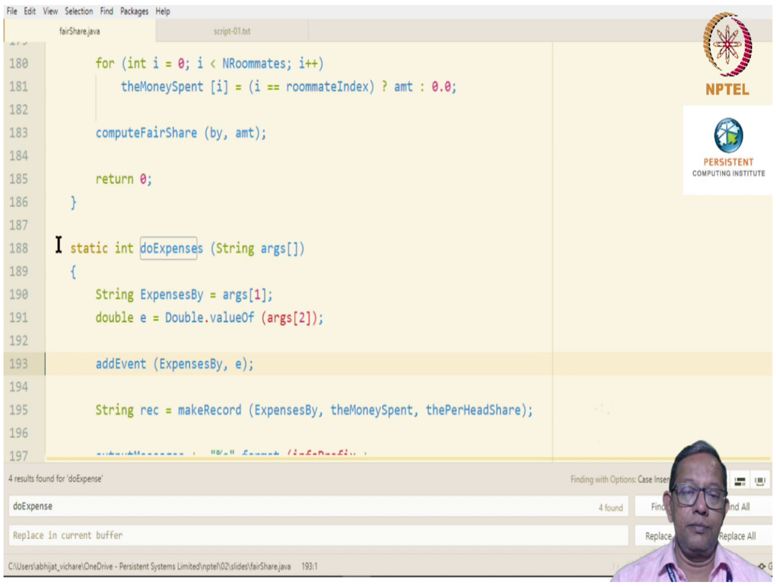
```
$ java fairShare register Aamod Abhijit Anuradha Rustom Shekher
```

So, if we skip the first key word register then the next 5 are the names of the roommates so if the number of words on the command line is X then X -1 is the number of roommates, if the first word is `register`.

That is what that is how the program first calculates a number of roommates. Once the number of roommates is done, it simply creates a database and registers them. We will come to the creation of the database a little while later. In the meanwhile let us look at register new roommates, recall that when a registration occurs it is a fresh start there is no data there are no events there is no expense done so far and the only thing that is done is that the information about who are the roommates is collected.

Let us represent this information as a header line in our database so the registered roommate method just creates such a header and writes it into the file, that is all that register roommate does. We will come back to the details of this database and files and so on so forth in a little while. Here we just wanted to illustrate the operation of the doRegister method.

(Refer Slide Time: 35:41)



```
        for (int i = 0; i < NRoommates; i++)
            theMoneySpent [i] = (i == roommateIndex) ? amt : 0.0;

        computeFairShare (by, amt);

        return 0;
    }

    static int doExpenses (String args[])
    {
        String ExpensesBy = args[1];
        double e = Double.valueOf (args[2]);

        addEvent (ExpensesBy, e);

        String rec = makeRecord (ExpensesBy, theMoneySpent, thePerHeadShare);
```

```
4 results found for 'doExpense'
doExpense                                   4 found
Replace in current buffer
C:\Users\abhijat_vichare\OneDrive - Persistent Systems Limited\nptel\02\slides\fairShare.java    193:1
```

File Edit View Selection Find Packages Help
fairShare.java
script-01.txt
NPTEL
PERSISTENT COMPUTING INSTITUTE
Finding with Options: Case Inser
Find
Find All
Replace
Replace All

Let us now go ahead and look at the next part which is doExpense(). Remember that the doExpense command-line involves the expense keyword, the name of the roommate who has spent the money, and amount of money that is being spent, all of this information is available on the command line and therefore the do expense method is programmed to accept the command line. And we have already seen that the command line is passed from main to process command line, and process command line to do expense.

The first argument of the command line is the name of the person who has spent and the second argument is the amount spent. The do expense method is the first point where we now need to convert something that is in string form to something that in an internal form for our program. The amount of money that is paid by this particular roommate is a string of characters on the command line. So, when Aamod spents 100 rupees what you typed as 100 is actually just a bunch of characters `100`.

This string of characters has to be converted to the numerical value 100, this is the phase of conversion from input to a form that is used within a programming language that we had seen earlier. In this case it appears to be just one statement so all it does is the string representation is converted to the double value and stored in a variable E. Now the do expense method calls another method `addEvent()` which we will shortly see.

Just like the `doRegistration()` method created a header record, the `doExpense()` method creates a expense record and then again writes it into the database. The `doRegistration()` method deletes any earlier database and creates a fresh new database, whereas the `doExpense()` method must always add to the current database.

Let us look at what add event does, the `addEvent()` method first obtains the index of the roommate, this is another example of conversion but it is slightly subtle. The name of the roommate who paid has arrived as a parameter ‘`by`’ to the method add event. The name is a string, and we need to use the `getIndex` function to obtain the index of the given name in the array of the roommates.

The `getIndex` method takes the name of the roommate and returns the index of the roommate in the ID’s array. At this point I would like you to recall that a few sessions ago we talked of the difference between input-output and parameter passing and return values.

Here is an example of the get index method, or function if you will, which accepts the string name parameter and returns the index as a return value. Once we obtain the row index of the particular roommate then we now wish to calculate who has spent how much and what is the per head share. That is done in `addEvent` method of the program: we run a loop over all the room roommates and if the ith index is the same as the roommate who has paid then we update the amount at that particular index, or it is 0.

So, we have an array which has as many elements as the number of roommates and for all the roommates except the one who has paid the contents are 0, for the roommate who has paid the contents are the amount of the money that has been paid. The add event method then goes on to compute the fair share.

Now, what is the method compute fair share doing? Well, this is the point where we divide the amount that has been paid equally amongst all others. This method has been illustrated before, let us see how it has been implemented. The compute fair share simply calculates the per head then it updates the per head share array for every roommate. How is that update done? If the current roommate is the same as the one who has paid then the value is the amount that that roommate has paid minus his or her per head share, if it is any other roommate then it simply the per head share.

A small digression, we hope you understand this syntax.

Consider the following statement:

```
int x = (y == 0)? 1:0
```

This is called as a ternary operator and the way it operates is that if the boolean `(y == 0)` is true then it sets `x` to be equal to `1,` and if the boolean is false then sets its value to `0`. This is different from a conditional statement which is normally what we see as the if statement in our programming languages.

In the code we used the statement:

```
thePerHeadShare[i] -= (i == roommateIndex)? (amt - phs): phs;
```

It is easy to see that this is just setting the value in the array `thePerHeadShare` the required way.

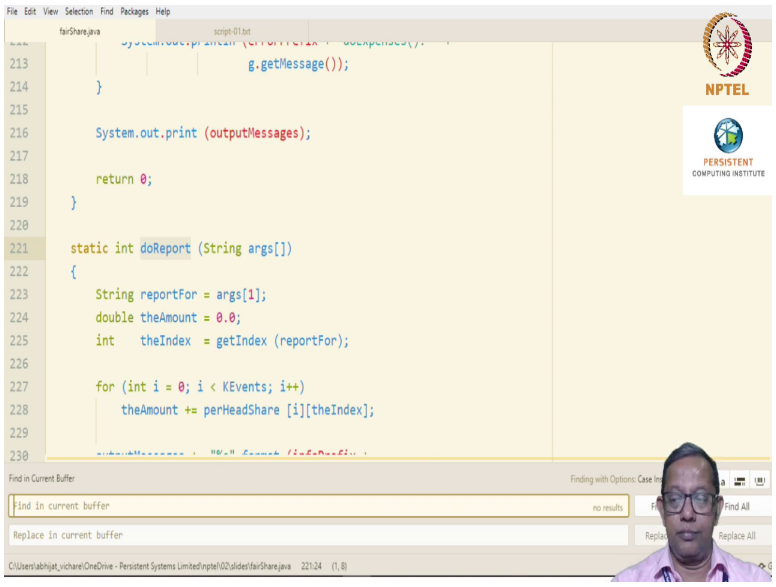
There is another way of doing the same calculations:

```
thePerHeadShare[i] = theMoneySpent[i] - phs;
```

Because the amount was already recorded in the array `theMoneySpent`; the money spent is either 0 or it is the amount that has been spent. And from everyone's individual money spent value, we can simply deduct the per head share to get the required numbers in the array. Also observe that compute fair share calculates the program invariant.

In the slides we can see what the `doExpense` method did: we saw that the `addEvent` actually just recorded the money that was spent and then computed the per head share. Having done that a record was created and written to the database; that is all that `doExpense` method does.

(Refer Slide Time: 45:46)



```
                        g.getMessage());
        }

        System.out.print (outputMessages);

        return 0;
    }

    static int doReport (String args[])
    {
        String reportFor = args[1];
        double theAmount = 0.0;
        int    theIndex  = getIndex (reportFor);

        for (int i = 0; i < KEvents; i++)
            theAmount += perHeadShare [i][theIndex];

```

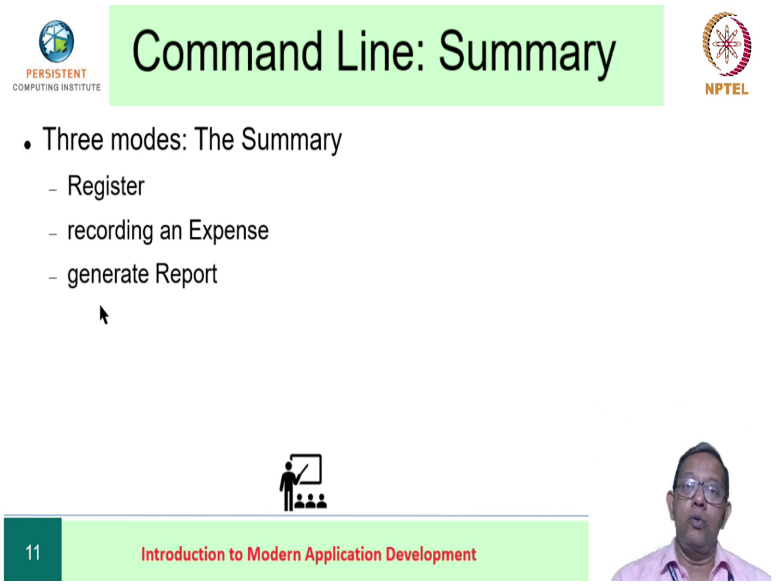
File Edit View Selection Find Packages Help
fairShare.java
script-01.txt
NPTEL
PERSISTENT COMPUTING INSTITUTE
Find in Current Buffer
Find in current buffer
Replace in current buffer
Find All
Replace All
C:\Users\abhijat_vichare\OneDrive - Persistent Systems Limited\nptel\02\slides\fairShare.java 221:24 (1, 8)

Let us look at `doReport`. Reporting would simply first take up the name of the person whose report is required. Again, this comes from the command line that we type which is automatically passed to the argument 'args' of `main` method. The main passes 'args' to process command line, the process command line dispatches the corresponding method and also passes 'args' to that method. That is how the command line arguments 'args'comes arrives at `doReport.`

As far as the reporting for one single individual is concerned conceptually, it is a very simple idea. If we use the image of the spreadsheet where events are rows and individuals or columns then the reporting for an individual the final amount is simply adding the contributions or debts for that individual across all the events. For every event whatever this individual has contributed, or not contributed, that is to be added and the final number is what is to be reported.

So, it starts with initializing `theAmount` to report by 0 and just goes on adding up the per head share across all the events of that particular roommate. Finally, we can simply return this value.

(Refer Slide Time: 48:29)



Command Line: Summary

- Three modes: The Summary
  - Register
  - recording an Expense
  - generate Report

11 Introduction to Modern Application Development

This is all there is to the program this has been a fairly intensive session and this is a good time to pause. We will continue in the next session with what would it mean to remember information. We had, in this session, skipped the ideas about creating a database, updating it, etc., we had just sort of mentioned it. In the next session we will focus on these aspects of our program.