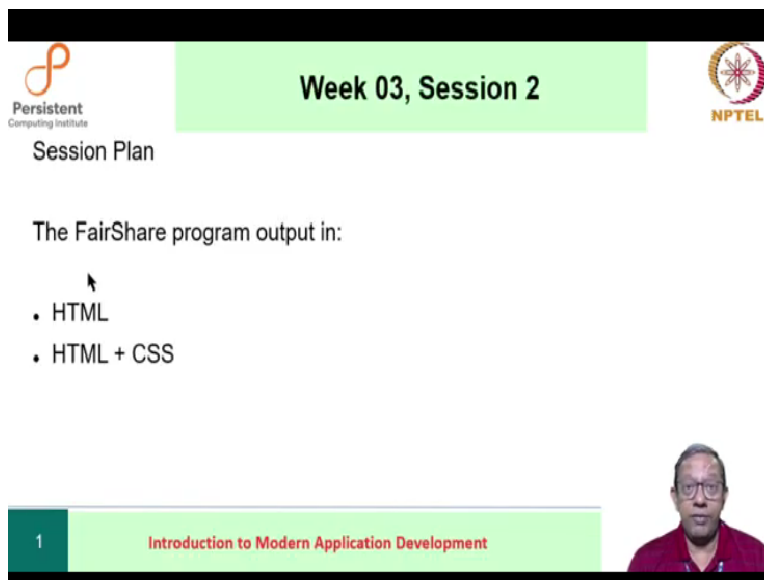


Introduction to Modern Application Development
Prof. Aamod Sane
FLAME University and Persistent Computing Institute
Abhijat Vichare
Persistent Computing Institute
Madhavan Mukund
Chennai Mathematical Institute

Lecture-12
Producing HTML+CSS output - Part 2

(Refer Slide Time: 00:12)

The image shows a screenshot of an NPTEL lecture slide. At the top, there is a black header bar. Below it, the slide has a white background. On the left, there is a logo for 'Persistent Computing Institute' and the text 'Session Plan'. In the center, there is a green rectangular box with the text 'Week 03, Session 2'. On the right, there is the NPTEL logo. Below the green box, the text 'The FairShare program output in:' is followed by a bulleted list: '• HTML' and '• HTML + CSS'. At the bottom of the slide, there is a green bar with the number '1' on the left and the text 'Introduction to Modern Application Development' on the right. A small video inset of a man in a red shirt is visible in the bottom right corner of the slide.

Hello, everyone, welcome to the second session of the third week of the course on introduction to modern applications development. In the previous session, we had seen how the web version HTML version of our FairShare application looked like and how did it behave. We saw there were essentially 2 versions:

1. one was bare simple plain HTML,
2. other was HTML which was adorned and made beautiful by using cascading style sheets or CSS as they are called in short.

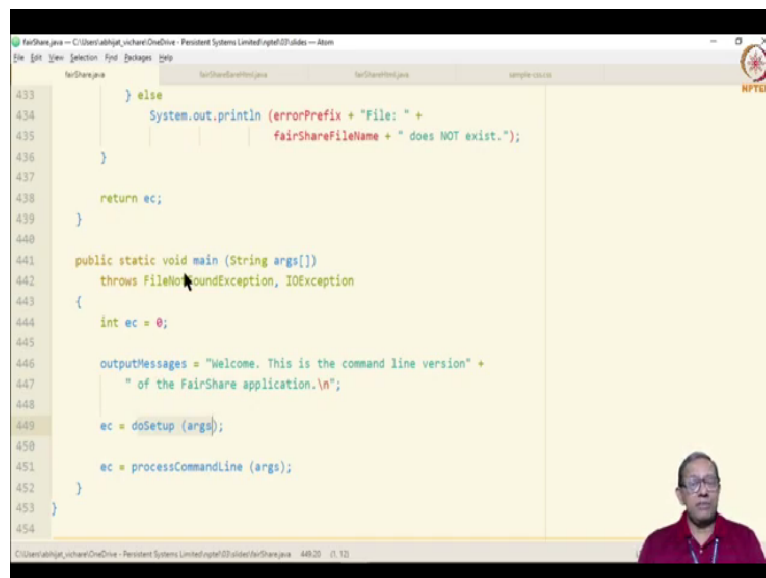
In this session we will look at the changes that we need to do in the command line version of the fair share program that will give us, first the bare HTML, and then HTML with added cascading

style sheets. Along the way we will introduce some basic HTML and style sheets concepts. We first want to get our programs running. So let us focus was on building the fair share application.

And then let us look at the basic ideas of the web, which include HTML and CSS, but also protocols that are involved in doing web transactions. The session plan for us will therefore be primarily looking at the HTML producing version and the HTML + CSS producing versions of our program.

Let us begin by reviewing the command line version that we did earlier.

(Refer Slide Time: 01:59)



```
433     } else
434     {
435         System.out.println (errorPrefix + "File: " +
436                             fairShareFileName + " does NOT exist.");
437     }
438
439     return ec;
440 }
441
442 public static void main (String args[])
443     throws FileNotFoundException, IOException
444 {
445     int ec = 0;
446
447     outputMessages = "Welcome. This is the command line version" +
448                     " of the FairShare application.\n";
449
450     ec = doSetup (args);
451
452     ec = processCommandLine (args);
453 }
454
```

The command line version is a fairly simple one; there is a static main as usual. And there are two main functions or methods, as Java would say, the one that just does a set up, and the second one that processes command line arguments, and we know that this particular method is our workhorse. The `processCommandLine()` method is a workhorse in the sense that it dispatches the corresponding method to handle the request as specified via the command.

So, if we say that fair share register, then the process command line method places a call to the registration function. If the command line says expenses, then the process command line method dispatchers or calls the method that processes expenses and so on and so forth.

In the command line version, the output, or whatever was produced by each function, was really just simply associated at that point of time within that function itself.

(Video Starts: 03:28)

So, for example, the `doSetup()` function informs us about the database, the particular file that stores the database, whether it does exist, and if it exists, what is its size and so on and so forth. This information is simply printed out. We would need to change this aspect, the one that brings out information. We are now going to print it out but such that instead of producing just plain unadorned text, it actually produces HTML. In order to produce HTML, we need to know the basic structure of an HTML page.

[Video at [6:39](#) for the HTML page]

Let us have a quick look at a sample HTML page. An HTML page consists of 2 main objects, the elements of HTML which are entities which are enclosed in angle brackets like `<html>`, `<head>`, `<body>`... etc. These denote the structural markup of the document. And the other part is of course the document itself, or the contents of the document itself. Here for example, we have the line “The FairShare System” enclosed between 2 markups the `<title>` and `</title>`. This markup says that whatever is between the `<title>` and `</title>` is to be rendered, displayed, or drawn on output device in a title format. Notice that the title is a part of the head of the HTML page. The contents of the HTML page are enclosed in the body tag. And within the body, we have various elements that markup the structure of the document, for example, we have a header, we have the body text, we have a list of objects and so on so forth.

This is the output that we need to produce. Note that earlier in the command line version, for example, we just produce this part of the output, the one that is now going to be contents of an HTML page, in the previous version we only produced the content but not the markup document using HTML. Now when we want to produce HTML, we should not only produce the content but also use HTML markers, and for this we need to know the various elements of HTML. Not only should we mark them up with the markers, but also structure the document.

We now need to think about what output should be going as title, what output should become a header, what output should be emphasized, what output should not be emphasized, and so on and so forth. None of this was something that we worried about in the pure command line application.

There are various ways in which people prefer to structure their documents. Feel free to structure your version in whatever way you feel is nice. The point that we are learning here is producing the HTML output. Let us go back to the HTML version. This is a version of the command line program that produces only bare HTML without any CSS. Notice that it has the usual 2 functions, the `doSetup()` and `processCommandLine()` method.

But now there is something more. You have a bunch of methods which are called only if we are required to assemble HTML. So, if assemble HTML is true, only then our program would execute various methods like:

- `doAssembleHTMLHeader()` : assembles the HTML header,
- `doRegistrationHTML()` : does the registration part of the HTML code page,
- `doExpensesHTML()` : does the expenses
- `doReportHTML()` : part does the report part,
- `doAssembleHTMLDatabaseInfo()` : assembles the database information,
- `doShowDatabase()` : shows the database, and
- `doAssembleHTMLTail()` : assembles a tail part of HTML. All of these methods one after the other collectively produce the entire document.

In other words, while this is one single whole document, all to the end of the file, the entire file is produced in parts by each of the method. So, for example, the `doAssembleHTMLHeader()` assembles the header part, that is produces the header information. The body contains registrations data, followed by expenses information if any, followed by reports if any... etc.; each of these form various parts of the one single HTML page. They are produced by different methods in the Java program. We have already looked at the processing that will be done by the command line version and that processing is going to remain the same. So, in this part, let us focus only on the

HTML producing functionality of our code. Let us see what the `doAssembleHTMLHeader()` does.

It has a local variable string `outputMessages`. And it is in this string that we will collect parts of the HTML page. So, for example, we print out a comment in HTML command begins with a left angle bracket, and an exclamation mark followed by 2 dashes and ends with 2 dashes followed by the right-angle bracket, i.e., `<-- This is a comment -->`. The output messages string gradually goes on adding individual HTML elements. So, for example, the first it adds the document type declaration.

Then it adds the HTML opening tag. Then the head tag, and then it adds the title of this document. Now, this title has been chosen by us as a matter of design depending on how we want the particular title to look. Then the head ends and the body starts. Notice that the operator used is `+=` which means that whatever is there on the right-hand side is added to the previously existing contents of this variable. In other words, it just string concatenates. Therefore, as each line is executed by our Java system, the HTML page goes on getting assembled HTML `<head>`, `<title>...</title>`, `</head>`, `<body>...</body>` and so on so forth.

At some point we finish doing the constructing the header part of the HTML, and of course, after this we will be doing the registration part. We will be assembling the registration part of the HTML. Please note that this `doRegistrationHTML` producing part of the functionality comes after all the calculations regarding fair sharing have been done. We will come to this point again.

The structure of producing HTML for registration is also very similar, we use the `outputMessages` local variable to go on collecting the contents of the registration section of the HTML page. So, it starts with a `<h2>` header roommate registrations. And then if our program is in the registration mode then it says that “it is registering”. If it is not in the registration mode, it simply says that “the registered roommates are” and then lists the roommates. So, the registration part of the HTML page is sensitive to whether our program is executing the register mode or any other more than register mode. This structure will also be seen in `do expenses` or `do reports`.

Recall that our web page, the HTML page, always has all the 3 components. Let us have a look at the web page again. We find that the web page has a title, the registration section, the expenses section, and the report section, followed by information about the database and the actual database if it exists.

If you look at the case when our program is executing the expenses mode, even then all the 3 sections exist: the registrations, the expenses section and the report section, but note that over here the expenses section adds an explicit in information about what expense for what event has been added. So, what we see here is that the structure of the functions producing HTML is fairly simple and straightforward, but you need to keep on doing it for every function and every point part of the web page. So, producing the HTML page is tedious but not hard.

Let us look at `doExpenses()`. Notice again that we are collecting the HTML output into the `outputMessages` variable. We are going on concatenating each addition to the HTML page into the same variable. Again, the structure if our program is actually doing expenses, then bring something add the event information. If it is not doing expenses, that is if it is doing any other more registration or report then it only says there are no expenses recorded.

Here is a question. How would you display the output “no expenses recorded”? What we do is that we check whether the current mode is `doExp`, and if it is we display the expenses else we add the line “No expenses recorded”.

But is that a current and user-friendly way of saying that there are “no expenses”. Currently we are not in expenses mode, for all you know your page could be talking about a report for some roommate. How else would you rephrase this particular expression? You want to tell your user that right now we are not in expenses mode but in some other mode, and therefore no expenses have been recorded. Think about that.

That is all there is to produce the expenses HTML part. The `doReport` part is also quite identical in structure, but different in content quite obviously, because its content is about a report.

It starts accumulating in the output messages, a string variable. It is outputting something if the mode is not report, and it is outputting a report, if the mode is to report something about a roommate. And once everything is assembled, it just simply prints it out on the standard output.

Here is how we collect out the database information, notice that it is now organized again using HTML tags, as they are called, we are organizing the database information as a table. On the left-hand side of the table, we are going to say what is the information that this particular row contains. On in the second column, we are actually going to give that particular information. So, for example, we are going to say, what is the file name of the particular database file. And on the right column we are going to tell display the name of the file.

This is also the information that our command line version used to produce. We are only producing that in HTML form so that it can be displayed on a browser. And we have already seen the web page produced by this program. And we are now at the tail of the HTML page. Recall again that our program first does a setup then processes command line. And during that executes all the processing required for the fair share application.

And finally assembles a complete HTML page with all the 3 commands, every time. The only thing is if it is doing registration, then the registration part of the page changes. If it is doing expenses than the expenses part changes, and if it is doing the report, the report part changes. We are having one page with all the information. But for every operation of the fair share application, it is only a part of the page that changes.

One wonders, would not it be nice if we could have HTML that is capable of only changing part of the page. That would be an interesting thing to do. Maybe we will look at it later in this course. But for now, we are going to write programs that always produce a complete HTML page.

This is a good point to take us to the pause and review the material that has been done so far.

What we have covered is producing an HTML output without any cascading style sheets. This produces bare HTML, no decorations or any visual elements. They are only markup elements. Let

us continue by looking at the fair share program but this time let us add support for cascading style sheets. We have already seen the effectiveness of cascading style sheets; the same bare HTML document actually looks drastically different when it has been when its presentation has been controlled by cascading style sheets.

How do we produce a cascading style sheets version of our code? Alright, here is the structure of the code that produces a cascading style sheet. For a start, again it starts with main, and actually has the same set of functions as the bare HTML program: doSetup() and processCommandLines(). And if we are to assemble HTML, then it calls the method which assemble the HTML header, do the registration, do the expenses, do the reporting, assemble the database information, display the database information and end with assembling the tail of HTML page.

So, it looks pretty ordinary. In fact, it is quite same as producing bare HTML. And that is how it is. Because as far as style sheets are concerned, the only thing style sheets do is to specify how individual elements of HTML are to be displayed on the target browser. There are 2 separate pieces of information the content that is the HTML page and the styling information in the style sheets. If you do not have style sheets, we have bare HTML, it is as simple as that.

Let us have a look at how the HTML header is assembled when we want style sheets. Is it different from the barebones version? Yes, it is. You will see that we now have one more piece of information in the header that is a link to a file called as a style sheet. What does the style sheet do? We just mentioned, a style sheets specifies how every element of HTML, every tag of HTML, every class in the document... etc. are to be displayed on the output device.

What are the things that we control in display? we could control the font of display, we could control the color of the text of the display, we could control the color of the background and so on so forth. Lot many aspects of presentation can be specified through style sheets. In the HTML page, the first thing we do is insert a link to the styling information in a style sheet. So that is why our header has that link tag, which actually contains the particular name of the file which contains the style sheet.

This is the name of the file “sample-css.css”. The rest of the structure is the same. Additionally, apart from marking the HTML tags, we would also like to mark the logical regions of the document. Recall that we displayed individual user command output in a different way compared to the others. So, we would like to mark parts of the HTML page with respect to their logical content.

So, we would say that a certain part is a *welcome part* and mark it up using class tag in HTML element `<div>`; we give it a class called as `welcomeBlock` and there is a corresponding closing block. So, whatever is between these `<div>` and `</div>` is going to be treated as a welcome block with class `welcomeBlock`. Why do we call it as a class `welcomeBlock`? well, CSS allows us to treat a bunch of HTML tags as a class and decide or describe the presentation details.

So, from a user presentation perspective we decided to markup block of HTML as logically relating to welcoming the user to this application. This idea continues again and we have a registration block with the corresponding end marker slash div, forward slash div. Whatever is between this `<div>` and `</div>` is going to be treated as a registration block. And we will specify all the common properties of the registration block in this part of the CSS. In a short while we will have a look at the CSS code.

Again, as usual the structure continues. In the string variable `outputMessages` we start collecting output of the code for assembling an HTML page. Here we define another class called as `expBlock` and have a corresponding end marker forward slash div that will come sometime later below. We output the usual expenses HTML as was done in the HTML version, and this function ends.

A similar functionality, but this time we will name the block as a report block. A similar functionality for the block reporting part of the HTML page will be used here. And this idea continues.

[Video at [31:15](#) for the CSS file]

Let us have a look at the sample CSS code as a final part of the entire story. Here is a sample piece of cascading style sheet which describes the style for the h2 element of HTML.

What does style say? the style sets the color of the font as the color described by RGB values in HEX; this number is in the format of RRGGBB. So red, green blue in hexadecimal digits. So, we do not want any red or green, but all blue, then we will use **#0000FF**. This means that headers of level 2 are to be displayed in blue text and the font is bold, header 3 is all red, no green, no blue, and is bold. Recall that we labeled a certain region of HTML page as a block, and we named it as a welcome block.

We describe using CSS that a welcome block part (elements with corresponding class) is supposed to have a color of all red, some green, some blue, and its font is one and half times what is the standard font chosen by the user (`font-size: 150%;`). If, however, the welcome block is in the header h1 (we capture this using `h1.welcomeBlock`) then it has to be displayed slightly: it has to be aligned in the center and has to have a different color. In contrast, an h1 tag that is a header of type one without any class is going to be displayed with the color 996633 which represents 99 red, 66 green and 33 blue in RGB values.

Any part of the HTML which is demarcated by registration block will be having this color, we could add additional specifications. For example, we could say that the font could be bold, or font could be italics. The way to specify is very simple. In the HTML page we name, and in the CSS, we describe how that name is to be presented. As far as describing is concerned, we take the attribute color and the way it is to be described; the value of the description. The attributes a font size, and the value, the attribute font type and the value. The general format is: “*property: value;*”.

So, a cascading style sheet is actually a very simple way of simply specifying the attribute of presentation and the particular value that it should take for a given presentation type or region of presentation.

The commenting style in CSS is very C like, it starts with `/*` and ends with `*/`.

E.g., `/* this is a css comment */`

Notice that we have marked each part of the web page by a separate class. In fact, that is because we want to display individual parts of the web page in different styles according to the need. This is how we generate pages which are given the cascading style sheets, such pages are presented in a much more better form than bare HTML.

With that we come to the end of this session where we have just been introduced to basic HTML and cascading style sheets, and how we write Java programs that generate HTML and CSS for the FairShare application.

(Video Ends: 35:30)

See you in the next session.