Transcript: Software Concepts

>> Let's look at software concepts in a little more detail. We'll take a look at algorithms and data. We'll dissect a Java program. Then we'll consider the software development lifecycle, translation and execution. We'll look at syntax, semantics, and errors, and overview of the programming languages. And then we'll finish up looking at object-oriented programming.

So, algorithms and data. We said that algorithms and data essentially made up software. Well, algorithms themselves consist of sequence, selection and iteration of instructions. By sequence, we mean the instructions are executed one after another. Selection means we may choose among alternatives, these will be if statements. And then iteration becomes our loops.

Iteration means we're gonna go through a set of instructions over and over. And we'll see how all this unfolds as we go along. Pseudo-code, this might be the initial program design. It's not actual software code, of course. Pseudo-code means false code. But, it's what you might write out in English about what the program is supposed to do in each step, or each method, and so on.

And these can become the formal comments in the language itself. We end up with lots of code, many pieces of code for algorithms and data. And, we're gonna organize these into classes, which define objects. And, and essentially, this is the idea of object-oriented programming. So let's dissect a, a Java program.

This is one we've looked at, the one that prints War Eagle three times. To begin with, we have comments. And recall, we're gonna have two Java.comments, one for the class and one for each method, and you see those there. And then the class begins here with public class FirstProgram and it goes, from the open brace to a close brace.

And we got a main method. It starts inside the class. We've got public static void main, and then an open brace, close brace. And inside it, of course, we have our programming statements. This is the algorithm itself that says, we're gonna print War Eagle, then War Eagle and War Eagle.

Identifiers are essentially words in the language, if you will. And, we have a number of them. Reserved words are the words you see here in purple or blue. Public class. Here's static void. So, so all of these are reserved words. And a reserved word means it can't be used for anything else in Java.

It's sort of off limits to you for, to use for, for anything but its reserved purpose, so to speak. And then we've got other identifiers, the method, the class name, method main, that's, that's a, an identifier. Notice main is not a reserved word, just an identifier. And then we'll discover variable names and things like that as we go forward.

We have the Java application programming interface. The System.out.println, this is the System class. Out is one of the fields in the System class. It's an object. And then we're calling the println method on it. Out itself is an output string, and this is how we're printing. So, we're already using parts of the Java API.

Also, one thing to note, and we'll probably note this numerous times, all class names should begin with a capital letter. And then other variables begin with a lowercase letter, like main and, and so on, and all of your variables, and we'll see that. We do have some literals in this program.

War Eagle is a string literal. And when we go forth, we'll have numeric literals. If we had the number 1, 2, 3, which represent a 123, that would be a literal, rather than say a variable. Then we've got white space. This is the indentation and skipping lines and all that sort of thing.

The white space is essentially, mostly, for human consumption. We do need the white space between reserved words and variables and things like that. For instance, static and void, this white space is, is critical, but we could have a lot more white space in there if we wanted to.

The compiler ignores all white space, except for one space, if you will, between identifiers. So, legally identifier can be any combination of letters and digits, and then two special characters, dollar sign and underscore. And, it can't begin with a digit. And of course, Java is case sensitive. So essentially, this, this is the way you form any identifier.

And this includes variable names, and so on, which we'll take up a bit later. But, just keep in mind, any combination of letters and digits together with the two special characters, dollar sign and underscore, and it is case sensitive. Case sensitive means that a capital letter versus a lowercase letter is two different things.

So capital X is a different identifier than lowercase x. Now, in the notes here, you'll see Q1, Q2 and Q3. These are pointers, if you will, to questions that are in the question folder. So I invite you to look at those and review those as you go through the notes here.

So, let's take a look at the program development lifecycle, we call it. There's a lot more to developing software than coding. And in coding, we think of construction or implementation. But, in general, this is, more or less, the traditional lifecycle. It's called the waterfall lifecycle, as you see.

And it's got a system engineering phase, requirements analysis, design, construction, testing and maintenance. And, we'll talk about all of these in more detail. But essentially, system engineering is dealing with hardware and software. Requirements analysis, mostly about the software. And out of it comes a specification, which we design from.

And then design produces a document that we can actually build the software from. The design says how we're gonna do it, and requirement says what we're gonna do. And then the construction phase is where, usually, the coding and unit testing takes place. So this where we build the software, and so on.

Now, there's a feedback loop in all of these. So, during design, if you discover issues, it goes back to requirements analysis. Certainly, during construction, it may go back to design, and so on. Now, this is the area of our focus. We're mostly about construction, but we're looking at a little design and, and a little bit of testing.

There's many variants of this process model, but this is sort of a typical one. And, it gives you a sort of a sense of where we fit into the big picture. So, as we said, construction includes code and unit testing. And code, by code we mean writing the source code that's gonna be compiled into an executable.

There's usually coding standards, that is, rules how the source code should be formatted, documented, and so on. And, we're gonna use Checkstyle to enforce our coding standard. And then also construction usually includes unit test. And by unit test, we mean a particular unit, usually, this is a class that, that's our unit level here.

And, it means once you write the program you need to make sure that the actual output of your program matches the expected output. In other words, what was said in the problem statement or a requirements document, and so on. And you can do that informally, as in you run the program and you observe that it ran correctly.

Or we could do it more formally with a testing framework like JUnit, where you actually write software that runs the program and compares an actual output with expected output. So, we need program tools. They're, they're valuable to us. A good integrated development environment is essential. That would include a program editor, debugger interactions, and so on.

And this should become one of your sort of best sources of productivity, if you will. jGRASP with Java and Checkstyle and JUnit and, and Web-CAT, that's an example of an IDE and, and that's the one we'll be using. And again, we use Checkstyle to sort of enforce our coding standard.

So looking at translation, this is sort of the big picture, if you will. We're gonna start out with Java source code. This is what you write, a .java file. And, then that's compiled. We saw where you click the little green plus to call the Java compiler. And then from that, we get Bytecode, a .class file.

You can verify that over in your browse tab when you run your, the compiler. If it compiles, you'll end up with a .class file with the same name. And then this runs on the Java interpreter called the Java Virtual Machine or Java Runtime Environment. We'll use those terms interchangeably.

This is fast. There's one other step, a Bytecode compiler that's not used often, but here, the .class file is sent to a compiler and generates machine code that'll run on a specific computer. The .class files themselves will run on any computer where there's a, an appropriate Java Virtual Machine.

So this, that's a rarely used thing. We won't be using it here, we'll just be using the Java interpreter. And so let's finish up with sort of a review of what we do in jGRASP. In the Edit window, you create your source code and save it. And then you compile it using the green plus.

And, if there are errors, you go back in Edit and correct the errors, compile it again. Eventually, you get clean compile. And then you run it, you inspect the output. If there's an issue, you go back and make changes, compile it again and run. If it's a hard problem to figure out, you may need to set a breakpoint and run in debug mode, and step through your program and see if you can resolve the issue that way.

Eventually, though, you get a clean compile and run, and, and you're, you're good to go. So this cycle, implies incremental program construction. You wanna create the class with nothing in it, and then create a method with not much in it. If you got more than one method, you'd create just the skeleton code of each one, essentially enough to make it compile.

And then you go back and, and fill out the details of it, compiling each time so that if there is an error, you can catch the error right on the spot. You need to plan to repeat this cycle early and often.