CompTIA IT Fundamentals: Software Development Concepts

Computer software is wonderful, but what makes it tick? Non-programmers might be interested to learn some of the basics of software development and how the magic is made. In this course, you will explore software development concepts, beginning with the basics of scripting, markup, programming, assembly, and query languages. Then you will delve into some of the tools used by programmers to map out program flow, including flowcharts, pseudocode, and sequence diagrams. Finally, you will discover common programming concepts, including branching and looping, variables and constants, arrays and vectors, functions, and object-oriented programming (OOP). This course helps prepare learners for the CompTIA IT Fundamentals (ITF+) certification exam, FC0-U61.

Table of Contents

1. Video: Course Overview (it_csitf23_08_enus_01)

- *discover the key concepts covered in this course*

[Video description begins] *Topic title: Course Overview. Your host for this session is Aaron Sampson.* [Video description ends]

Hi, my name is Aaron Sampson. Computer software is at the heart of the tasks we perform on our computing devices, but what makes it tick? Non-programmers might be interested to learn some of the basics of software development and how the applications we've come to depend on are made. In this course, I'll explore software development concepts, beginning with the basics of scripting, markup, programming, assembly, and query languages.

Then I'll examine some of the tools used by programmers to map out program flow, including flowcharts, pseudocode, and sequence diagrams. Finally, I'll cover common programming concepts, including branching and looping, variables and constants, arrays and vectors, functions, and object-oriented programming. Upon completion, you'll have built the foundation for the basics of IT. This course helps to prepare learners for the CompTIA IT Fundamentals or ITF+ certification exam, FC0-U61.

2. Video: Scripting Languages (it_csitf23_08_enus_02)

- *provide an overview of scripting languages*

[Video description begins] *Topic title: Scripting Languages. Your host for this session is Aaron Sampson.* [Video description ends]

In this presentation, we'll provide an overview of scripting languages, which can generally be thought of as programming languages, although scripting in itself isn't quite the same as programming due to the fact that scripts are interpreted at runtime as opposed to being compiled. Now, we'll take a closer look at what that means in just a moment, but whatever the difference is, scripting has become an integral component of software development. So, for

anyone intending to pursue a career as a developer, you'll undoubtedly need to learn some scripting languages. So, what's the difference then between something that is interpreted versus compiled? Well, for starters, when developers are creating program code for something that is not a script, while the code might seem rather cryptic to non-developers, it is still readable by humans.

So, that code is referred to as high-level code. However, while humans can understand the code, the computer itself does not. So, compiling is the process of taking that code and converting it into a machine-readable format which is also known as low-level code. In addition, all code that makes up the application is compiled into this machine executable format, so you end up with a complete application once everything has been compiled, and that program only needs to be compiled one time, at least until the code is updated or its version incremented. Scripts, however, do not require compiling. The code itself, as written by developers, is still in a fairly human-readable format, but each command of the script is processed or interpreted into machine code one by one, each time the script is run.

Ultimately, compiled code will execute more quickly because it doesn't have to convert each line of code during each execution. But scripts can usually be written much more quickly, and they're much more reusable and portable across different computers and platforms. Plus, once program code has been compiled, that's it. The software is considered to be complete and packaged, so to speak, with the exception of updates and new versions as mentioned, but a script can be modified easily at any time. Another fundamental difference is where the code can be executed. Again with a compiled program, you might be able to run it on different computers, but its function is always going to be the same no matter where it runs. But when building multi-tiered applications, developers can create different scripts that are designed to perform different tasks on different computers or what is more simply known as client-side versus server-side scripting.

Server-side scripting can be used to generate dynamic web content and interact with the database and the local file system of the server itself, whereas client-side scripts are typically designed to run in the browsers of client systems to execute instructions that are specific to that client. So, both of them can be used to create a much more flexible application. So, looking more closely at scripting languages, they're very often used to aid in automating various aspects of an application or web page or the shell of an operating system's Command Prompt. As mentioned, scripting languages are interpreted each time when they're executed, so they don't require any compilation. So, you could just type up even just one or two commands in a scripting language and save it as a file, and you have a completed script that can now be executed on any system that supports that language.

And despite their seemingly less complicated nature, scripts can be very powerful tools when used effectively. Common examples of scripting languages include JavaScript, which itself is very commonly used with browsers to enhance the functionality of web pages. Ruby which is also commonly used in web programming. Python which is more general purpose, so to speak, in that, it can be used for more than just web page programming, and PowerShell, which is native to Microsoft and commonly used for administrative task automation and the development of

management interfaces. Typical uses for scripting languages include providing automation and enhancing functionality in various types of apps, including mobile apps, text editors, operating system shells, web pages, computer games, and embedded systems. But in short, if you're able to accomplish the required task with a scripting language, then that's all that really matters. So, it will simply come down to what kind of functionality you need for your projects and whether scripting languages can provide you with a solution.

3. Video: Markup Languages (it_csitf23_08_enus_03)

- *outline the role and types of markup languages*

[Video description begins] *Topic title: Markup Languages. Your host for this session is Aaron Sampson.* [Video description ends]

In this video, we'll outline the role and types of markup languages, which are very human-readable programming languages, but instead of it being just computer code, it uses what are known as tags to define elements in the document that indicate how it should be structured and formatted. Now, a good way to think about tags is to imagine a paper-based document that has to go through several drafts before a final copy is created. In each draft, someone might proofread the current version and add in corrections, revisions, footnotes, and anything else that indicates what should be different in the final copy. That's actually where the term markup comes from because the tags are similar to the way we would mark up those paper drafts. But tags in markup languages aren't really used to define any corrections or changes.

Rather, they define the characteristics of the information being presented, such as fonts, colors, and general formatting. Now, there are several markup languages, but two of the most common are HTML and XML. HTML stands for Hypertext Markup Language, and you might recognize it as being the language used to build web pages. And again, HTML uses tags to define the layout of the web page and the objects it contains. Just as a very simple example, if you want a word on the page to appear in bold text, you place tags at both the front and the back of that word to indicate that the final copy should present that word in bold. The hyper portion of the name comes from the fact that any page can contain a link to another page known as a hyperlink, which you can click on and it immediately brings you to that other page.

HTML can be created and edited using any basic text editor, including something as simple as Notepad in Windows, but there are certainly more advanced editors available with features to make coding easier. But in itself, it's not an interactive language, meaning that it doesn't accept any kind of input from us. The end result really is just a document, but it's typically viewed

through a browser as opposed to something like a word processor. HTML is very simple in its processing as well, in that it doesn't require any kind of compilation because nothing is executed in HTML. Browsers inherently process the tags, so as soon as you save your project as an HTML file, you can view it immediately using any browser. It doesn't have to be uploaded to any kind of web server either, so you can work on your HTML files locally and see how they look just by opening it with your browser.

XML stands for Extensible Markup Language, meaning that it is meant to be extended in terms of its capabilities. It still uses tags just like HTML, but the tags aren't just used to define formatting characteristics but to assign and identify certain types of information. In short, the set of tags that can be used with HTML are predefined, whereas with XML, the tags can be just about anything. Now I realize that statement might not be all that easy to visualize, so here's an example of each. At the top, we see standard HTML and its use of tags, which are the characters included within the triangular brackets. The text begins with the letter p as a tag, which indicates the beginning of a paragraph. Then text is entered, but the word italic, in this case, means that it should be presented in italicized text.

So, the letter i indicates that all letters that follow should be italicized until the tag is closed off with another letter i in the brackets, but also with a forward slash which indicates that this is the closing tag as opposed to another opening tag. As such, all characters between the two tags will be italicized. Then there is also an example of a hyperlink with the address of the link enclosed in tags that use the letter a, which by the way stands for anchor link. But with XML, the tags are describing attributes of the data. So, the company name of Easy Nomad Travel is enclosed by tags that say the word COMPANY and its web address is enclosed with tags that say WEBSITE. So, HTML would not recognize what those tags are. But what's happening here is that the data itself is being drawn from an already existing database, where COMPANY and WEBSITE are existing column names from a table.

So, this is what I was referring to when I said that the tags could be just about anything because they can reflect any attribute [Video description begins] *A slide labeled Examples appears. It contains 2 types of coding examples: HTML and XML. The codes of HTML are mentioned in the following lines. Line 1 reads: This is italic text. . Line 2 reads: Click <a. Line 3 reads: href=http://www.easynomadtravel.com>here to. Line 4 reads: visit our website. The codes of XML are mentioned in the following lines. Line 1 reads:<COMPANY>Easy Nomad Travel</COMPANY>. Line 2 reads:<WEBSITE>www.easynomadtravel.com</WEBSITE>.* [Video description ends] of any element of data from just about any other source. XML can process those tags so that the data matches the structure that has already been assigned to that data by the existing database, so it's very commonly used when the website draws its data from another source. In either case, both HTML and XML are used almost exclusively for creating web pages, so the choice of markup language simply comes down to whether you're working with already existing data or you're creating the information from scratch. But both allow you to post that information to a web server so that your data can be accessed by anyone on the public Internet, or perhaps just an internal intranet if desired. But markup languages provide us with

the ability to make information much more accessible because all that's needed to view it is a browser.

4. Video: Programming Languages (it_csitf23_08_enus_04)

- *describe the role and types of programming languages*

[Video description begins] *Topic title: Programming Languages. Your host for this session is Aaron Sampson.* [Video description ends]

In this video, we'll describe the basic role and types of programming languages used to create applications. And in this context, programming refers to languages that must be compiled, which itself means that the code must be converted from the human-readable information written by the developer into machine-readable binary code. The original code written by the developer is referred to as source code, and once compiled, it's known as object code, but it becomes an executable application that can then be installed onto your computer. However, compilers are specific to the processors and the operating systems that will host the applications. So, in many cases, source code will need to be compiled separately for different operating systems that use different types of CPUs.

Although with some modifications it is possible for applications to be compiled in a manner that supports multi-platform use. It depends on how it's being developed, but applications that are designed to run on a single platform only tend to be more stable and are easier to develop in the first place. If there are any errors in the source code itself that cannot be processed by the compiler, the process will halt and the error in the code must be addressed. So, as a developer, applications are generally not written from start to finish on the hope that it will compile correctly. Most applications can be developed in a modular fashion so that sections of the application can be built separately, and most development studios provide you with the ability to test and debug your code so that compilation errors can be avoided.

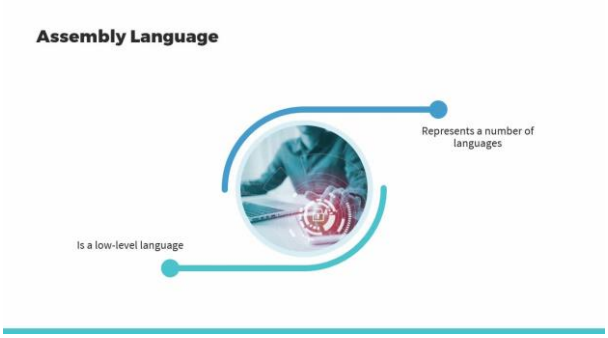
Common examples of compiled programming languages include C and C++, whereby the pluses indicate the addition of new features, Java, Visual Basic, C#, Pascal, and COBOL, which stands for Common Business Oriented Language. Now, there are certainly others, and we can't get into the specifics of each one in a short presentation, but the choice of language will depend on the situation, the type of application being developed, and the type of system on which it will run. But these days, one of the primary distinctions of any given programming language is its ability to work in what's known as an object-oriented manner, which we'll talk about in greater detail in an upcoming presentation. But in object-oriented programming, data is treated as an object,

which can also have varying values, known as variables. For example, you could create a variable called number, then assign a value of 5 to that variable, then procedures can be attached to the variable, such as multiplying it by 2, producing a result of 10.

But you could then assign a different value such as 10 to the variable object, then execute the same procedure of multiplying it by 2, and get a result of 20. So, the variability of the object allows the same procedures to produce different results. Languages that aren't object-oriented focus on functions and logic instead of objects and are sometimes referred to as procedural languages because they execute the same procedures in a top-down manner with each execution. But both types have their purposes and both are certainly required. For example, in just building the environment of an application, you would not want the code to produce any kind of variable results. You want to see the exact same environment every time. But when you need to pass in a varying value to a procedure, such as the discount percent on a purchase, you most certainly need the results to reflect the variable nature of the value being passed in.

So, to finish up, the advantages provided by compiled languages primarily include efficiency and performance. Once the code has been compiled, it never needs to be recompiled unless the source code itself is altered, such as when a new version is created. This is sometimes referred to as process once, run many, or also write once, read many. If you contrast that approach with something like an interpreted language, they aren't compiled so they have to be processed every time they execute, so they aren't as efficient, but they're also generally easier to create. In any case, when it comes to designing and building an application that can be installed on your computer and can then execute reliably in a self-contained manner with no other resources required, then a compiled programming language is the approach to take.

5. Video: Assembly Languages (it_csitf23_08_enus_05)



- *outline the role of assembly language*

[Video description begins] *Topic title: Assembly Languages. Your host for this session is Aaron Sampson.* [Video description ends]

In this video, we'll examine the role of assembly language, which is a low-level programming language, meaning that it isn't used for purposes such as developing software applications, rather, an assembly language is designed to interact with system hardware. Now, there is no specific language referred to as assembly. It's more of a categorization because there are several languages that operate at this level. So in terms of core functionality, an assembly language communicates with hardware directly. And this is the primary reason as to why the term

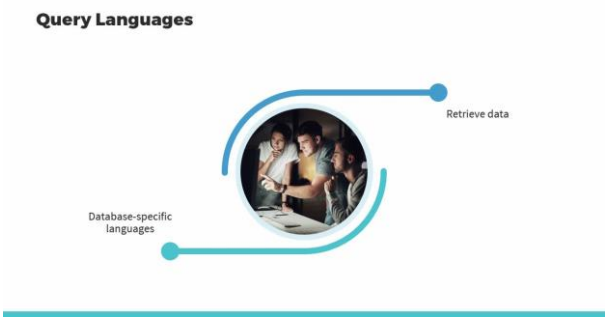
assembly is more of a categorization because processor architectures vary. Hence an assembly language has to be written for the specific CPU architecture on which it will run.

So, while I couldn't say how many assembly languages are in use, there are at least as many as there are processor architectures. Now, while the assembly language itself is designed to communicate with hardware directly, it is still a programming language that is written in a human-readable format. In other words, developers aren't writing assemblers in ones and zeros, but they have to act as a bridge between software programs and the underlying hardware, which effectively means that they're able to translate higher-level programming languages into a machine-readable language that itself is in fact made up of ones and zeros. So, since assembly language is specific to the CPU architecture, they're typically used for very specific tasks such as coding device drivers, boot codes, or low-level embedded systems.

But perhaps more to the point, they aren't used to create large applications that we as users will install, but they are required for those applications to be able to function on their respective platforms. So because of them, developers at those higher levels don't have to worry about coding their software to handle these translations themselves. They can rely on the assembly language to look after that for them. So, having an understanding of assembly language helps one to realize how applications interface with the operating system, the BIOS, and the processor, as well as how data is represented in memory, how the processor accesses and executes instructions, and how those instructions can access and process data, ultimately allowing the applications we rely on to perform the tasks we need them to do.

Now to finish up, it's worth noting that most assembly languages these days are already written, so as a developer, it's unlikely that you'll find yourself creating an assembly language. And if your focus is on web development, front-end application development, or database development, then it's quite likely that you'll never even work with an assembly language. So, like most things, their use depends on your situation. But if, for example, you're building a new type of hardware device, or coding drivers for that device, or even creating a new type of operating system, then you would almost certainly find yourself working with an assembly language.

6. Video: Query Languages (it_csitf23_08_enus_06)



- *identify the purpose of query languages*

[Video description begins] *Topic title: Query Languages. Your host for this session is Aaron Sampson.* [Video description ends]

In this presentation, we'll compare and contrast query languages, which are the primary languages used in database management systems, which themselves are the software used to store and access the data contained within any given database. And they provide the interfaces between the users and the database itself, although in this context, it would be administrative interfaces, not those used by end users. In other words, database management systems are how administrators and developers construct the databases, to begin with, then manage and control the data once it has been entered. So, query languages then are used to both create and interact with databases, but there are several different implementations that are each specific to their particular platforms.

But as the name query indicates, one of their primary purposes is to retrieve data so that it can be examined or manipulated. But to be clear, that's not their only purpose. The data is originally entered or written using a query language, and it can also be modified. Query languages are also used to define the structure of the database itself. But again, retrieving data or querying it is a very common database task. To expand upon that, common query language tasks include managing the schema or the structure of the database itself, which refers to the types of objects that can be created within a database Most notably the tables that store the data and how they relate to each other. Populating those tables with data in the form of records, updating or modifying the contents when needed, searching for records, defining the storage locations of the data, and managing what's referred to as the data integrity, which can include several methods, but a common example is by using what is known as a constraint which limits or restricts the data that can be entered.

For example, you could prevent a database from accepting a value for a person's date of birth that is later than today's date. Now, databases themselves have two primary categories known as structured and unstructured, and I'm mentioning this because it has a direct impact on the type of query language that is used in each case. A structured database refers to the fact that data is very well organized in terms of its storage. Tables are used that each have predefined columns for each specific piece of information. As an everyday example, if you've ever filled out any kind of form, then that's structured data. There are specific field that contain individual values, but the collection of all values represents a single record.

So, in a database example, if you have a customer table, that table would include individual values such as a customer ID, name, email address, phone number, and anything else you require. But all values put together form a single customer record and every customer requires that same type of information. But, of course, the values themselves will vary for each customer. But they all have a customer ID, a name, an email address, a phone number, and whatever else you require. Unstructured data is exactly as it sounds. This tabular format with predefined columns simply isn't used. An everyday example of unstructured data would be a standard document or a memo. You just start typing in whatever it is you want to say.

In terms of databases, a very commonly used example these days is gathering and storing social media posts or news articles so that the information within them can be analyzed to determine what activities people are most actively engaged in, or what is of interest or trending at the moment. This can be very useful information for targeting advertisements, releasing new

products or services, or planning events. So, the Structured Query Language or SQL, which is often pronounced SQL, is, of course, used in structured databases, which are also known as relational databases because the records of one table can be connected or related to the records of another table via a common field. For example, using our customer table, any given customer might place an order. Well, that order is a separate entity entirely and would be stored in a separate table.

But any given customer can place any number of orders, so it's referred to as a one-to-many relationship. But any given order can be traced back to the single customer who placed it by including the customer ID value as a field in the orders table. So, in terms of the fields of the orders table, you might have something like an order number to uniquely identify that particular order. But the next field might be the customer ID into which you place the unique customer ID of whoever placed that order. So, that relates the order back to the customer and creates the relationship. That customer can, of course, then make many other orders, but they would have different order numbers, but the customer ID would be the same for all of them. So, SQL has the ability to define those relationships, which is part of the schema management mentioned just a moment ago. Hence it's particularly well-suited for accessing and managing structured data.

And it also integrates very well with many other major programming languages because so many modern applications require some kind of database to act as the primary source of information. The unstructured counterpart is called XML query or XQuery for short, which as is indicated by its name, manages and interacts with data in XML or Extensible Markup Language format. Now, while XQuery is more commonly used with unstructured data, it should be mentioned that it can also be used with structured data because, in many cases, structured data is used in combination with unstructured, which is perfectly fine. It just depends on the nature of the database and there are cases when structured data needs to be presented in a more unstructured manner. Now both SQL and XQuery have varying implementations that are specific to their platforms.

For example, Microsoft implements a version of SQL known as Transact-SQL or T-SQL, and there is a Java derivative known as JavaScript Object Notation for XML or JSON, but all extensions of each are based on certain standards. But, of course, any given vendor can include any extension they want in their own products, but if those extensions stray too far from the standards, then they risk facing compatibility issues with other platforms and applications. In either case, though, if you're considering a career in database management, you'll almost certainly be dealing with both structured and unstructured data. While structured databases will most likely persist indefinitely, unstructured data is becoming more and more important to many organizations. So, it would be highly recommended to become familiar with both query languages.

7. Video: Flowcharts (it_csitf23_08_enus_07)

- *provide an overview of flowcharts and their usefulness in computer programming*

[Video description begins] *Topic title: Flowcharts. Your host for this session is Aaron Sampson.* [Video description ends]

In this presentation, we'll provide an overview of flowcharts, which, of course, in any situation can be used to help visualize the logic involved in any complex task. But they can be a very useful tool in computer programming because they can help to analyze the processes involved when determining what type of input might come into an application, what must be done with it, and what kind of output must be produced. The flowchart itself is simply comprised out of a number of visual objects that typically have meaningful shapes and colors assigned to help visualize the workflow as a process, and each shape is then connected with the others with a direction of flow to help create a step-by-step process or to represent an algorithm.

Common objects include a starting and ending point, the arrows to indicate the direction of flow, decision points, actions to take based on those decisions, input/output points, database connections that are required, wait points, and documents. But what's more important in any given environment or implementation is that they remain consistent. For example, it really wouldn't matter much if you used a red triangle to represent the decision, as opposed to a yellow diamond shape, as long as all decision points always use a red triangle. Flowcharts can be useful in just about any situation where there is some kind of process that needs to be visualized, particularly if that process is very complex, which can often be the case in developing software.

For existing processes, a flowchart can help to identify areas of improvement, or they can be used to explain the process to others for training purposes, or to plan a new project from scratch, whether it be part of developing new software for updating and refining existing applications. As a very basic example, most flow charts have some kind of starting point which usually involves a decision shortly thereafter. But, of course, any decision can have multiple possible actions that could be taken. So, in order to fully analyze the process, you have to include processing paths for all possible actions that can be taken.

So, in this example, we see a fairly simple yes or no response to the initial decision, but each response would then lead to different actions which themselves might require or create the need for additional actions. But ultimately, the goal is to arrive at a resolution point where the process is complete and the actions taken based on the decisions are appropriate and satisfactory. So, when using flowcharts, while there are diagramming applications that can assist

you with their creation, there really is no right or wrong way to use them, as long as you use them in a consistent manner and they accurately depict the process at hand.

And ultimately, as long as it makes sense, then that's what's really important. But when creating and working with them, be sure to involve those who will actually be performing the real process to ensure that you're hitting all key areas, and also include all key players such as customers, suppliers, managers, and, of course, the developers in the process so that everyone is on the same page, so to speak. This will help to ensure that developers don't get started on the wrong path or end up focusing on the wrong task.

8. Video: Pseudocode (it_csitf23_08_enus_08)

- *outline pseudocode and its usefulness in computer programming*

[Video description begins] *Topic title: Pseudocode. Your host for this session is Aaron Sampson.* [Video description ends]

In this video, we'll provide an overview of pseudocode, which is the practice of informally representing programming code in an easy-to-understand language, as opposed to the actual code of the programming language itself. It can be useful for developers to help visualize an overall process, and it can also be useful for training purposes or explaining code to others.

If, for example, you need it to justify why a particular piece of code was used. Now in terms of the format or the structure of pseudocode, it does tend to follow the same conventions as the actual programming language. So if you were to simply look at it, it wouldn't entirely just look like plain English in a document, for example.

But again, it would use more standard English words in this example to simply make it more easily consumed by humans as opposed to by a computer. A good way to think about pseudocode is as a rough draft for the eventual final revision. By using pseudocode, developers can help to improve the readability of their work. Now we'll see an example of this in just a moment, but programming code tends to use many paragraphing techniques, such as indenting to help identify certain steps in a process.

So by constructing the appropriate objects first without having to worry about the actual code, it's easier later on to simply go back and replace the pseudocode with actual code. The pseudocode itself can also be used to explain the steps of a program because many programs are very complex, so it's often difficult to visualize everything that needs to be done.

By explaining what the code needs to do, developers can simply create the code appropriate to just that step without having to remember the entire process or everything that needs to be done. It also helps to enhance the flow charting logic and can in fact act as a bridge between the flow chart and the eventual finished product. So, a logical history of the development steps can be documented and demonstrated if necessary.

In other words, if someone in a key position were to ask why was this done this way, you can lead them through the process from the flow chart through to the pseudocode through to the actual code to provide them with a clearer and more satisfactory answer. So as mentioned, here we see a basic example of pseudocode in what's known as a while loop, also sometimes referred to as a do-while loop, which itself means that an action should be taken as long as a condition evaluates to true and/or what to do if that same condition evaluates to false.

[Video description begins] *A slide called Pseudocode Example (While loop) appears. It contains the following lines of code. Line 1 reads: BEGIN. Line 2 reads: WHILE. Line 3 reads: Condition. Line 4 reads: DO A THING. Line 5 reads: REPEAT. Line 6 reads: NOT condition. Line 7 reads: DO A THING. Line 8 reads: DO A THING. Line 9 reads: DO A THING. Line 10 reads: DO A THING. Line 11 reads: END.* [Video description ends]

So, this is where we see the paragraphing and indenting common to many types of code. Now I should mention that as far as the processing of this code goes, none of this formatting is actually required, but it helps the developer and really anyone else reading it, to more easily recognize where the actions of the code are, as compared to the evaluation of the condition. So by entering the pseudocode using the same format and structure, it makes it easier to identify where the real code should go.

So the BEGIN and END statements simply mark the boundaries of the piece of code itself, but they are in fact actual words used in many programming languages. But again, they also help to separate this code from other blocks of code. So, each block will usually be separated from others by several spaces as well. But the while statement initiates the evaluation of the condition, and while is also an actual word of many programming languages.

But the word Condition in this case is pseudocode and it represents where any kind of condition needs to be evaluated. Just to give you an idea of what a condition might look like, the code might need to examine a quantity field and repeat a task as long as that quantity is greater than 0. When that condition evaluates to false, the action stops. So in this case, the pseudocode of DO A THING simply represents an action that needs to be taken based on the status of the Condition, and the NOT condition is simply what to do in the opposite case, again with repetitions if necessary.

To help visualize that, if you consider a simple morning routine, the condition examines the time of day, and the specific

[Video description begins] *The slide now contains the following lines. Line 1 reads: BEGIN. Line 2 reads: WHILE. Line 3 reads: Before 7 AM. Line 4 reads: SLEEP. Line 6 reads: 7 AM or later. Line 7*

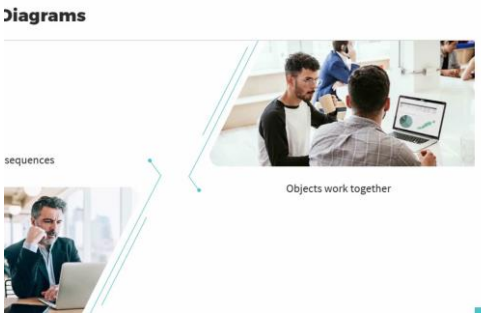
reads: GET UP. Line 8 reads: GET DRESSED. Line 9 reads: EAT BREAKFAST. Line 10 reads: GO TO WORK. Line 11 reads: END. [Video description ends]

value of concern, in this case, is whether the time is Before 7 AM or after. If it's before, then the action taken is to simply remain asleep.

If, however, it's after 7 AM, then there are several actions taken. Now obviously, that's just for visualization purposes, but with the pseudocode created, a developer can now just go back at any time, later on, and replace the pseudocode with actual code without really having to worry about remembering every aspect of the application. The condition is laid out, the necessary actions are indicated, and the purpose of the code is already established. And to finish up, I should mention that pseudocode can also be a very effective learning tool because there are a lot of keywords and syntax requirements in any programming language.

So by writing it out in plain English first, then even if you don't recall the exact keyword at the moment, at least you know what kind of code you need in that section. So, it might just be a matter of doing a simple Internet search to find the correct keyword and syntax. So, if you're planning on pursuing a career in software development, be sure to implement pseudocode to help get you started and to make coding easier as your skills improve.

9. Video: Sequence Diagrams (it_csitf23_08_enus_09)



- *describe sequence diagrams and their usefulness in computer programming*

[Video description begins] *Topic title: Sequence Diagrams. Your host for this session is Aaron Sampson.* [Video description ends]

In this presentation, we'll take a look at sequence diagrams and how they're used. And as you might guess based on the name, they're diagrams used to define the sequences required for any type of system where multiple objects have to work together as a whole to produce a certain outcome.

Now, based on that definition alone, you might imagine that a sequence diagram could be created for just about any situation where any type of system is being developed. And in fact, that is the case. But in this context, sequence diagrams do have a particular role in software development.

But before we get to that, scenarios where sequence diagrams can be especially useful include analysis of the requirements of an application with respect to both what is needed to create it

and what it needs to do, the approach to its design such as phases or stages that might need to occur in a specific order, and documenting the process during its development and after its completion. So with respect to the specific role of sequence diagrams in software development, this requires a quick introduction to the Unified Modeling Language, which is a language specifically designed for the creation of rich visual models to assist with the development of complex software applications.

And it is worth noting the term complex here because sequence diagramming just might not be necessary if the application being developed is very simple. But UML in and of itself is not a programming language. Its core use is in fact to create diagrams, but there are tools within those diagramming applications that can be used to generate code in various languages based on the diagrams that you create. In other words, once your diagram has been completed, you can choose to effectively convert that diagram into the appropriate code to implement that process.

In addition, in some cases, the opposite can be done, whereby existing source code can be converted into a sequence diagram, which can be very useful when explaining the purpose of your code, particularly for training purposes. Now, in both cases, it would be recommended that you thoroughly examine either the code or the diagram that was generated, because it certainly would not be guaranteed that either would be 100% correct. But both approaches can certainly save you time and help you to create documentation.

Use cases for sequence diagrams include analyzing the method logic of functions, procedures, or any other complex process. Usage refers to how the system being developed could potentially be used. For example, if I'm developing a new type of gadget for a particular purpose, I might discover that it could also be used for several other purposes. Sequencing, of course, is exactly that, but more specifically, it can be used not only to indicate a particular sequence that must be followed, but if any of those steps can at least occur at the same time, or if there can be any overlap.

For example, once step one is complete, steps two and three might be able to be addressed simultaneously. And service logic refers to any high-level method used by client systems used to complete a task. For example, when client systems boot up, they typically request an IP address through a service, but there are several steps in that process, so a sequence diagram can be very useful in mapping out that process.

So to finish up with a more specific example, in this diagram, we see the process of a user logging into a database environment and each step of the process is laid out in the

[Video description begins] *A slide appears with the heading Example. It contains the 3 stages: Login, Validation, and Database. The initial stage is User Login stage contains 3 steps: Click login button, Login successful message, and Display profile page. It leads to the next stage: Validation. It contains the following steps: Validate user ID and password, Login successful, and Profile information released. The last stage leads to Database. It contains the following steps: User ID and password match and Profile information.* [Video description ends]

sequence by which each step must occur. And the steps are also categorized in terms of their function or the objects involved with the processing of that request. And as mentioned just a moment ago, once the user passes in their user ID and password and that information is validated, we see steps in the validation and login stages that can be processed simultaneously, whereas other steps must occur in the correct order.

For example, the user ID and password cannot be validated until the user has supplied them. Now that's certainly somewhat obvious to state, but it's not always so cut and dry, so to speak. So by correctly diagramming the sequence of any part of any application, the developer has a much better handle on the logic of the application, which can reduce development time and ensure that the application functions more reliably.

10. Video: Branching and Looping (it_csitf23_08_enus_10)

- *provide an overview of branching and looping and how they're used in computer programming*

[Video description begins] *Topic title: Branching and Looping. Your host for this session is Aaron Sampson.* [Video description ends]

In this presentation, we'll provide an overview of branching and looping in computer programming, which are logic components of the application that are designed to address different possibilities that might arise from its use. Now before getting into the specifics, a good way to imagine branching and looping is simply through the activities of our daily lives.

Virtually every time you're faced with a decision, the options being considered represent branching, in that you ultimately choose one option over the others. For example, if it's a nice day, you might choose to walk to your destination, but if it's raining, you might choose to drive, but you can't do both.

But in other situations, you find yourself needing to repeat the same operation multiple times to complete any given task. For example, the task at hand might be to chop the wood for your fireplace, but that overall task requires the same action over and over again for each piece of wood until the entire pile has been chopped.

Now, branching is most commonly implemented through the use of if...else statements or switch or case statements depending on the language being used. But if else quite literally defines the actions that should be taken if any given condition evaluates to true and the actions

that should be taken if that condition evaluates to false. So if we go back to my example of daily life, the situation might be that if the weather is nice, then I'm going to walk to work, otherwise, I'm going to drive.

So in terms of the logic, nice weather is the condition being evaluated. Walking is the action taken if nice weather evaluates to true and driving is the action taken if nice weather evaluates to false. However, not every situation only has two possible paths. Perhaps if the weather is foggy but not entirely raining, there might be a third middle ground option to walk to the bus stop, then take the bus. Or in some cases, there are many options, so switch and/or case can be used in those situations.

A common example is when a range of values is evaluated and there are different actions or results for each range. For example, in a grading system, scores between 90 and 100% might be rated as excellent, scores from 80 to 90% would be very good, scores from 70 to 80% would be good, and scores from 60 to 70% would be fair. Now, it's also worth mentioning that if statements can also be nested, in which case, there is a primary decision which could then lead to another decision. For example, if it's foggy but not raining, I'll take the bus, but if it starts raining while walking to the bus station, then I should bring an umbrella.

So, any given if statement can be nested within another if statement. But this is where switch and case might be more effectively used because there really is no limit to the number of possible paths that can be defined with switch and case.

So, here's a simple if else statement in pseudocode whereby if a condition being examined evaluates to true, then take an action, if it evaluates to false, then take a different action.

[Video description begins] *A slide appears with the heading If...Else Statement in Pseudocode. It contains the following lines of commands. Line 1 reads: BEGIN. Line 2 reads: If (condition = true). Line 3 reads: Do Something. Line 4 reads: Else // condition <> true (condition is false). Line 5 reads: Do something else. Line 6 reads: END. The next set of lines reads as follows. Line 1 reads: BEGIN. Line 2 reads: If (7 AM yet?). Line 3 reads: Get up. Line 4 reads: Else. Line 5 reads: Keep sleeping. Line 6 reads: END.* [Video description ends]

So in this case, the logical test is whether the time is later than 7 AM. If it is, then the action taken is to Get up. If it isn't, then the action taken is to continue sleeping.

Looping, however, requires multiple operations to be performed over and over again until a condition evaluates to either true or false, and they typically implement while, for, and do...while statements to accomplish their task. Now, in terms of logical processing, a while loop checks the condition first, then executes the statements based on the evaluation of the logical test. A for loop specifies a particular starting value, then execute statements until a specified termination value is reached.

And a do while loop is similar to a while loop except that it will execute the statement first at least once, then examine the condition to determine if the task has been completed. So in my earlier example of chopping wood, a while loop would be to examine the number of pieces of

wood that remain unchopped and if that value is greater than 0, I chop another piece. Then I would look at the pile again to see if the total number of pieces remaining unchopped is still greater than 0, and if so, then I chop another piece and I keep doing so until the number of remaining pieces is no longer greater than 0.

In a for loop, I might realize that the pile of wood is too large for me to chop in one session, so I would specify to start with just one piece, then continue chopping until 100 pieces have been chopped. And in a do while loop, I would simply go out and chop a piece of wood first without examining the number remaining in the pile. Then afterward, I would examine the pile to see if that single operation completed the whole task, and if not, then I continue with the standard while loop.

If so, then I'm already done. So, whenever you encounter a situation in your application code where there is more than one possibility to address for any given situation, the use of logic statements can almost certainly address those varying possibilities.

11. Video: Variables and Constants (it_csitf23_08_enus_11)

- *outline variables and constants and how they are used in computer programming*

[Video description begins] *Topic title: Variables and Constants. Your host for this session is Aaron Sampson.* [Video description ends]

In this presentation, we'll take a look at variables and constants, which are types of identifiers in programming languages, which themselves are user-defined names of program elements. So, what does that mean? Well, if we begin with variables, they're named containers for storing some kind of data, and as the name variable indicates, the value of that data can be changed for each execution of the code.

As a very basic example, to perform a mathematical calculation, you could create 2 variables, 1 called Val 1 and one called Val 2. As a developer, you can use whatever names you want, hence the named container aspect. But by defining them as variables, your computer reserves a little bit of memory to hold values that can then be assigned to them.

So you could then set Val 1 equal to 3 and Val 2 equal to 4. Then in the code, execute Val 1 times Val 2 and you would get an answer of 12. But you could then change the value of Val 2 to 5 and re-execute that exact same code, but you'd get an answer of 15.

So, the variables are the named program elements identifying the areas where values can be stored for later execution. Now, when variables are defined, programming languages will require you to specify what kind of information will be stored in the variable. In my previous example, of course, integers were used, but you certainly aren't limited to just integers. Floating point numbers can also be used which refer to numbers that use decimal points.

Char refers to characters and would store either a single character or perhaps a small number of them for values such as a product code and string is effectively a group of characters for when phrases or sentences need to be stored, or for simply larger groups of text-based characters such as a person's mailing address. There are many other types of variables that can be defined, but the idea is to ensure that the correct type is used so that the appropriate operations or actions can be performed against them.

For example, to perform a mathematical calculation, values do need to be numeric. Variables also have a property known as scope, which refers to how available they are to other areas of the application, particularly when it comes to using functions in your code, and we'll talk about functions later on as well. A local variable is only available to the function in which it has been declared. So in my previous example of using Val 1 and Val 2, if I had declared those variables within the code of a function that I'm creating called multiplier, then I can only reference those variables within the code of the multiplier function.

If I had another function called divider that was defined entirely separately from multiplier, then I would have to declare separate variables again in the divider function. If, however, I were to declare those variables as being global, then both the multiplier and the divider functions could reference the same variables, so it's up to you as the developer.

But in short, if the same variables need to be called from other functions or procedures, then they should be declared as global. But if they're only needed for that particular function, they should be declared as local. Now, you might think that it would just make more sense to declare every variable as being global, because then you could just reference it anyway regardless of the situation. But in fact, that's not the case, because global variables are exposed to all functions and procedures, so it can represent a security risk to make the values so widely available.

Now, as you might imagine, a constant is effectively the opposite of a variable. It's still a named container for holding data, but its value is predetermined and cannot be changed during normal execution. Constants are sometimes also referred to as literals in some languages. Now, different programming languages will implement their own constants, so they won't always be consistent across every language with respect to how they're used.

But one example that tends to be fairly universal is what's known as a boolean data type, which is commonly used for true and false evaluations. So, when any given condition is evaluated, the result is either going to be true or false. For example, we might compare the date of an order to a specified date, such as the 1st of the month, and evaluate if that order was later than the first.

Well, that's going to be either true or false in every case, and we don't set those values ourselves, nor does it change during execution. So for that particular test, the value remains

constant. So, while their implementation may vary to a degree for certain languages, the use of variables and constants are one of the most commonly used aspects of all software development, so be sure to familiarize yourself with their use if you're considering a career as a programmer.

12. Video: Arrays and Vectors (it_csitf23_08_enus_12)

- *describe arrays and vectors and how they are used in computer programming*

[Video description begins] *Topic title: Arrays and Vectors. Your host for this session is Aaron Sampson.* [Video description ends]

In this video, we'll examine arrays and vectors in computer programming, both of which are types of containers, which themselves are simply collections of other objects. Just like a box can hold many other objects, but containers in programming store objects in an organized manner with specific and defined access rules.

A container, then can contain an array or a vector, both of which serve the same primary purpose of storing data and objects for viewing, loading, and unloading, but each has their own method of providing that functionality. So, beginning with arrays. They're data structures that can be used to hold a group of items that typically share the same data type, such as a collection of numbers or characters.

But what distinguishes an array is that it has a fixed size in terms of the data it contains, as defined by the developer. As such, arrays should only be used when the size of that collection is known in advance and is unlikely to be changed, because if it does, then the developer will effectively have to allocate a new array and copy the existing data into the new one.

In other words, arrays are not dynamic. Vectors then are also collections of data that typically share the same data type as well. But unlike arrays, vectors are dynamic, meaning they can be resized on the fly, so to speak, so new objects can be inserted into the collections whenever it's necessary.

So, vectors tend to be more commonly used in today's programming environments, not only because they can be resized, but they also offer several other advantages, including automatic memory management. They support built-in sorting functions, whereas arrays require manual sorting. And vectors are a standardized feature of most modern programming languages, which leads to wider support across multiple languages.

Now, these advantages might lead you to ask why arrays are used at all, and in short, you might not need to use them. But one consideration, albeit rather minor, is that anything that is dynamic tends to have greater performance demands than anything that is static, because there's simply more processing involved with managing any dynamic characteristic, whereas that management simply isn't required for something that is statically set.

But as mentioned, it's a rather minor consideration, and the other advantages provided by vectors usually outweigh any performance considerations. So, I would say that it's likely that if you're considering a career in software development, you'll likely find yourself using vectors much more frequently than arrays.

13. Video: Functions (it_csitf23_08_enus_13)

- *outline functions and how they are used in computer programming*

[Video description begins] *Topic title: Functions. Your host for this session is Aaron Sampson.* [Video description ends]

In this video, we'll provide an overview of functions in computer programming, which are essentially just routines that are designed to perform a specific purpose in a consistent manner. Now, almost every programming language comes with its own predefined functions that are available to be used, and in fact they aren't limited to just programming languages.

One of the most common examples of using functions is in spreadsheet applications, whereby you can simply select a group of numbers and click on the SUM button to add them up. That's an example of a function. It takes in values, performs an action, and provides a result. And in fact, that's one of the defining characteristics of a function. They always produce some kind of a value as a result.

Now, while there are many predefined functions available in just about any language, sometimes the function you need just doesn't exist, so they can also be custom written as well. But regardless of where it comes from, a function has to be called or invoked within the program code for it to be able to perform its task. But you can call that function as many times as is needed within your code until your overall operation is complete.

In addition to predefined and custom written functions, there are also function libraries which are still predefined, but they exist outside of the program so to speak, and can in fact be called and used by different programs. Now some libraries are included within a programming language, but they can also be acquired or licensed as an add-on if they aren't.

But they provide the benefit of someone else having already written the code for you, which of course can save you time as a developer because you don't have to code it yourself. If you do have to create your own custom function, then regardless of its purpose, another great feature of functions is their reusability. Any function need only be defined or written once. From that point it can be used or referenced any number of times for as long as you need it, which is primarily why so many functions already exist or are available through libraries because of the fact that they persist indefinitely.

So it's worth investigating to see if the function you need does in fact exist already so that you don't have to create it yourself. But even if you do, you can make that function available to anyone else as well.

So to finish up, here is a simple pseudocode example of a function called GET UP and my name in parentheses after the

[Video description begins] *A slide appears with the heading Function Example in Pseudocode. It contains the following lines of commands. Line 1 reads: BEGIN. Line 2 reads: If (7 AM yet?). Line 3 reads: Execute function GET UP (Aaron). Line 4 reads: Else. Line 5 reads: Keep sleeping. Line 7 reads: FUNCTION GET UP (Aaron). Line 8 reads: Shower. Line 9 reads: Put clothes on. Line 10 reads: Make breakfast. Line 11 reads: Go to work. Line 12 reads: END.* [Video description ends]

function name is known as a parameter which is a specific value passed into the function so that it would only apply to me and not someone else. Now again, this is only for visualization purposes, but a conditional test is performed first which evaluates whether the time is 7 AM. If it is, then the function is called. If it isn't, then I can stay in bed.

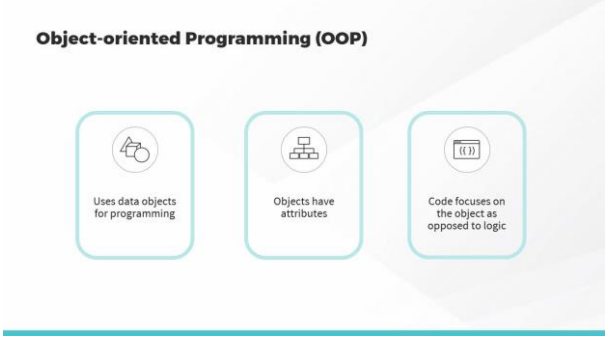
So on the assumption that it is 7 AM, then the function specifies several tasks that Aaron must complete, and it should be noted that those individual tasks themselves could also be functions. In other words, any function can call another function. For example, Make breakfast might require another function to brew coffee. But the idea is that these tasks are always executed consistently based on the evaluation of the condition and applied to the parameter specified, which again in this case is me.

So if for example, I had a roommate who didn't get up until 7:30, the function knows not to operate on the roommate because I passed in the parameter value of Aaron. Now, just as a simpler yet more realistic example, if we go back to the SUM function of a spreadsheet, the function would look like the word SUM followed by at least two values in parentheses separated by a comma. But the values would typically reference cells of the spreadsheet as opposed to specifically entered values, although that would also work.

So it would look something like the word SUM, then an open parenthesis, then maybe A1, A2, where A1 and A2 are the addresses of the cells. But again, that function has already been written and we can just call it as many times as we need. So, we can use that same function for different cells and it will still perform the same core operation. So, by using functions you can

significantly reduce the development time and increase the usability and capabilities of your code.

14. Video: Object-oriented Programming (it_csitf23_08_enus_14)



- *provide an overview of object-oriented programming*

[Video description begins] *Topic title: Object-oriented Programming. Your host for this session is Aaron Sampson.* [Video description ends]

For our final presentation of this course, we'll provide an overview of object-oriented programming, which is an approach whereby data is regarded as objects which can then have attributes that further refine the definition of the object. But the code of the application is focused on the object itself, as opposed to solely performing logic operations or functions.

For example, procedures can be assigned to the object that each perform different tasks, but they all apply to the same object, such as updating the level of inventory of a particular product each time it's ordered, or changing its price when market conditions change.

Object-oriented programming also relies on the concept of classes, which act as a blueprint or a template for the object, whereby each object is a specific instance of that class. They also define the set of attributes that describe the object and the methods or procedures that can be assigned to them. To help visualize that, imagine an everyday object such as a car.

The specific instance of my car is considered to be an object, but the scope of all cars would be the class. Then the attributes would be the things that describe that specific instance, like the make and model, the color and the body style.

And the methods would be the operations I can perform against it, such as changing the oil or filling it with fuel. Now, objects can be defined by a developer as an abstract data type or as a reflection of a real-world entity. For example, something like a text item could be defined as an object into which just about any text-based information could be entered.

So it doesn't refer to any real world object, but the text would still have attributes such as the font, size, and color. However, something like a customer can also be defined as an object and that would clearly represent a real-world entity. But a customer would still require attributes such as name, phone number, and e-mail address.

The types of objects used in any given application are simply dependent on the nature of the application itself. Now, looking a little more closely at the nature of an object, as mentioned, an

object uses attributes to describe the object itself. So again, coming back to something like a car, examples of its attributes might include its shape, color, size, or brand, whereby each value of any given attribute further refines the description of the object, and the collection of all attributes describe the object fully, at least with respect to what is required by the application.

But objects can also have properties. Now, that might sound similar, but there are special types of attributes that tend to be more abstract than attributes. Or to put that another way, attributes tend to have more specific values. For example, if you were asked to describe a person, a very common attribute you might mention would be the color of their hair. For instance, you might say that George has brown hair, but having hair in the first place is not common to all people, so you might describe George as being bald, in which case there is no hair color.

So, you can think of this scenario as hair itself being a property of all people and the value might just be something like yes or no or true or false. Whereas hair color as an attribute would have a specific value such as brown, black, or blonde, but specific to a particular instance of a person. So in terms of programming code or any given person, you might enter something like has hair equals true, hair color equals brown, or if has hair equals false, then hair color equals null.

And to finish up, the methods refer to the procedures that can be assigned to an object which then determine the actions that can be taken against that object. In my everyday example of a car, I mentioned that we can change the oil or fill it with fuel. But in an application, if I need to change the inventory level of a product, I would define a method that would increase the inventory level whenever new products are received from our supplier and decrease the level when units are ordered by our customers.

I might also need to update the price of a product or set its status if for example, it's on back order or if it has been discontinued. Ultimately, object-oriented programming is very common in the software development world of today, and many different languages are built around this approach. So once you become familiar with working with objects, it makes it easier to learn additional programming languages.

15. Video: Course Summary (it_csitf23_08_enus_15)

- *summarize the key concepts covered in this course*

[Video description begins] *Topic title: Course Summary* [Video description ends]

So in this course, we've examined basic concepts surrounding computer software development. We did this by exploring scripting, markup, programming, assembly, and query languages. Flowcharts, pseudocode, and sequence diagrams in computer programming. Branching and looping, variables and constants, arrays and vectors, functions, and Object Oriented Programming. In our next course, we'll move on to explore basic concepts surrounding databases.