What Is Programming?

by Simon Allardice

Start Course

Bookmark Add to Channel Download Course

Table of contents Description **Transcript** Exercise files Discussion Learnin

Course Overview

Course Overview

Hi, this is Simon Allardice, and welcome to Pluralsight's What is Programming? You're now watching our most proudly fundamental course. This is where to begin if you've never written a line of code in your life, or even if you have and want to review the basics. And no matter what programming language you're most interested in, even if you're not completely sure about that. This course will make learning that language easier. We'll do this by starting with the most fundamental, critical questions. How do you actually write a computer program and how do you get the computer to understand it? We'll jump into the syntax, the rules of programming languages, and see many different examples to get the big pictures of how we need to think about data and control the way that our programs flow. But this is Pluralsight, so this won't be just an overview, we'll even cover complex topics like recursion and data typing. And finish by exploring things that make real-world programming easier from libraries and frameworks to SDKs and APIs. But you won't find a lot of bullet points here, this is a highly visual course, and by the end of it you'll understand much more about the process of programming and how to move forward with writing any kind of application. But unlike most courses, this one does not require any one programming language, operating system, or application. There is nothing to download, nothing to install. So just sit back and relax as we get clear on What is Programming?

Introduction

Introduction: How to Think About an Infinite Machine

I'm not going to begin by trying to persuade that programming is important, you already know that. And unlike some, I'm won't begin by pretending that everything about programming is always simple and easy. A lot of it is, and some of it isn't and takes time, but you already know this too. So instead, let me begin by telling you why this course is different and how it'll make learning programming, if not always easy, at least easier. You see, most people would have you jump straight into one programming language, we're not going to do that here. Instead, we will explore many different languages and use them to make sense of programming itself, and what that really means, to get clear on the most important ideas, concepts, the rules that lie at the heart of all of this. So it then doesn't matter which programming language you decide to dive into, they all make sense. So we'll cover the terminology, the jargon, the words we use everyday in programming, terms like algorithms, recursion, object orientation, static typing, pseudo code, multithreading, and lot some. But it's not just to be able to talk like a programmer, more than that, you'll understand the meaning behind those words, so when you want to you can think like a programmer. And there is huge flexibility in that thinking. Programming is not one thing, it is not fixed, it is different for different people. For some, programming is corporate, it's all business. For others programming is game development, or it's art, or music, or language, or telling a robotic arm how to deal cards. We're taking this infinitely reconfigurable machine, one that can be different from moment to moment, and turning it into a machine that does what we want it to do. And, to be able to do this, to have an idea in our head and turn it into a computer program, it's not about what language you use or what operating system you have, it is about how you think.

Getting the Most from This Course

You could take this course for a lot of different reasons, let's quickly cover three of them. First, you want to become a programmer, a software developer. If you want to build programs for a desktop, or a laptop, for the web, for a phone, whether it's Mac, or Windows, or Linux, or whatever else may come in the future, what we're covering here are the core ideas and skills you will always need and always come back to. Reason number two, you may have no intention of becoming a programmer right now, but you do work with them, or manage them, or live with one, and you've been thinking it might be useful to know more about their world and what's important to them, maybe just get a better idea of what exactly does a programmer actually do all day long. You'll get that here. And reason number three, what if you're not completely sure about this? Perhaps you're worried this might not be for you. Or maybe it's too late. Maybe if you were wired up the right way, surely you would have done this already. No, it's okay, programming doesn't have to be the grand passion of your life. Now if it is, fantastic, but right now I don't even need you to be totally sure about what you plan to do with this knowledge, but is there something about this you find interesting? that's good enough, we can work with that. When you read about this or see it on films or TV, is there something about programming and coding that looks interesting or a little intriguing? Now it might also sometimes look a bit intimidating, or unfriendly, or frustrating, or boring, sure, it can be those things, for some people it's just a job, but this can also be incredibly rewarding both intellectually and in your career. Whatever background you're coming from, I guarantee you that learning more about this is always worth it.

Steps and Sequence

To learn anything well we should always be unashamed to ask even the most basic question. So, what is a computer program? Okay, I get it, that sounds too easy. I could take a room full of random people and ask, what is a computer program, and I will get answers, the things everybody kind of knows like--- A computer program is a way to tell a computer what to do. That's true, but useless, it doesn't help us here. Or I'll get--- A computer program is a list of instructions. Also true, and a bit more specific, but still, not very helpful. What does this really mean? What kind of instructions would be in a list? What do they look like? How do you tell a computer to do something, and what can you actually tell it to do? Now, sure, it would be great if we could actually just talk to it. Make an interesting navel strategy game set in the Napoleonic Wars. But as you can guess, our instructions need to be a little more specific than that. Now before I show you some I will admit that if you take them one by one, the instructions we can use in a computer program are going to seem a little simple, unremarkable, unimpressive, a bit dull, because there's simple things like add one number to another number. That's the kind of thing we can actually ask a computer to do, add one number to another number. Or, display the word Hello on the screen, or beep for 0. 5 second. These are specific, self-contained, and very small instructions. Now many people have heard this, but they don't want to admit there's something unsatisfying about the idea of computer programs just being a collections of teeny, tiny instructions; it doesn't feel like a good enough answer. Here's what I mean. If you've ever tried one of these many learn to code interactive programming tutorials on the web, they'll usually have you begin by typing one of these simple instructions, whether you're trying to learn the language JavaScript, or Swift, or Python, it doesn't matter, what they ask you to type will usually look something like this. It's one instruction to the computer to display some text on the screen. You can use this one instruction to make a program, what's often called a Hello World computer program, it's a program that just does one very basic thing to prove that it works. So you write this, you learn how to run, or execute, your new computer program, and you get your result. And you wonder if you're supposed to feel like a programmer now, but then you look a the kind of thing you actually like to do on your computer, and this does not feel like the same thing as this. So I forgive anyone for thinking, okay I can understand how this happens, but there's no way this is just a list of instructions like add one number to another number. But it is, it really is. As we'll see, all computer programs are constructed from multiple, small, specific, individual instructions. As one equivalent, think of a recipe in a cookbook, because that's also a list of instructions, it's a series of steps to follow to make one dish. And it doesn't matter if that recipe is short and simple or long and difficult, they all come down to specific, well-defined instructions like, add. 25 ts salt, or heat oven to 220 degrees. And by itself, as in programming, each individual step is kind of boring, and looking at one instruction by itself tells us nothing. This one instruction here could be used to make a cake, but it could also be part of a huge multi-day preparation of a 12-course state banquet for a 100 people involving truffled quail jelly and live lobsters. In cooking and in programming, if you want impressive, complicated results you will still use basic, simple instructions, you will just have a lot more of them. And even when you understand this fact intellectually there's often part of your brain thinking, yep, I don't see how we get from Hello World to hello world of Warcraft, and that's okay if that's not apparent yet, that's why we're doing this, but we start simple, we have to start with the programming equivalent of how to boil an egg, because even if it's unexciting it's how we get the first ideas into our bones. Idea number one, in programming, we build incredibly complex results from combining lots of simple instructions. And idea number two is that the order of those instructions is incredibly important. So imagine we're following a basic recipe with just 10 or 12 steps to it. It's simple enough, but if we take just two of these steps where we add 4 eggs and combine, and then bake at 40 minutes at 350 degrees, now you don't change anything about these instructions, you just put them the wrong way around, bake for 40 minutes then add 4 raw eggs and combine, one result is going on a plate and the other is going in the garbage. So even though each instruction, each step, might be well-defined, they're also often dependent on each other and the right sequence is the difference between a result that works and a result that doesn't. So if you understand the idea that to write a recipe, one that other people could follow reliably, you'd have to think it through. You'd have to focus on exact quantities, you'd have to break it into small steps and have those steps in the right order. And you'd probably have someone test it to make sure that you didn't forget something. And it's the same process we need in programming, to think through a larger idea, break it into small steps, those individual instructions, make sure the amounts are correct, and the sequences exact so we can tell the computer to do it, and do it again, and do it again, and do it again, a million times a second if we need to. Now, unlike a recipe where we do have some flexibility of how to phrase it for humans beings to understand, we must now write our instructions in way that the computer understand using a programming language, and we have lot of options for that, so let's see a few of them.

Source Code

Let's take that phrase we heard earlier, a computer program is a list of instructions and add, that we write using a programming language the computer understands. Right now this is a useful way to think about everything. The sentence is not entirely true. I will explain what I mean by that in about 4 minutes. But if we want to get started writing this list of instructions to the computer there are many different programming languages we could use. C, C++, C#, Objective-C, Java, JavaScript, Ruby, Python, Swift, (Inaudible). And we'll talk about why there are so many, but I say again, we're not going to focus on any one of these languages, we're going to see a lot of them. So I'll show you some older languages, like C from 1972, some new ones like Swift, released in 2014, some languages that are really popular now like Ruby and Python, and some other languages that were once popular, but aren't any longer, like ALGOL. And you'll notice a few things. Some programming languages seem to be all uppercase, others all lowercase, some use a lot of symbols, curly braces, and semicolons everywhere, others don't, but you don't have to worry about memorizing any of those details, right now that's not necessary, that's not important, because we're using these not to find out what's true about one specific language, we want to know what's true about all of this. And the first truth, to begin writing a computer program, writing instructions in any of these programming languages is simpler than many people realize. We don't need to first go and download and install a whole bunch of special new applications onto our computer before we begin because when we write these instructions we use plain text to do it, and that means you could use the simplest, most basic text editor provided on your computer, like Notepad on a Windows PC or a TextEdit on a Mac. Programmers don't need a word processor like Word or Pages because we don't want any formatting here. Nothing is in bold or italics, a programming language doesn't care about that, and it doesn't care what font or font size you use, it just wants plain text. Now regardless of programming language there is another term you'll use for programming language instructions written in plain text, we just call this source code. And when a programmer tells you they spent all day coding or all day writing code, they mean writing source code. Now, sidebar, you might wonder if this is called source code is there any other kind of code? Well, yes there is, we'll get to that shortly. End of sidebar. Let me be clear about something. I said you don't need any special software to write this source code, and that's true, but you often want one. There are applications you can install on your computer that will make your life easier. Most programmers don't just use Notepad or TextEdit, we have specialized editors designed for programmers that add things like color coding and can help correct simple mistakes in programming the same way that a word processor can help with spelling and capitalization. Now there are many of these editors available, different ones for different languages and different operating systems, we will dive deeper into this, but later on in the course.

Why Code Looks Like Code

A quick sidebar. I said the font doesn't matter when writing source code, but you might be wondering that whenever you've seen programming on TV or in the movies it always has the same kind of look to it, often something like this. You don't see source code written in a font like Times New Roman or Comic Sans, but if it doesn't matter what font we use, why does it usually look like this? Well partially it's history. Early programming environments included simple, dumb terminals that were black screens with green text, that's all they could do. And even though modern operating systems can provide more visually stimulating content, a lot of programmers are very comfortable with a very simple and straightforward layout, partly it's a little cinematic, most programmers don't code with just a green font on a black background, they use editors with color coding, some like a dark background some like a light background. But one thing is true, even though we could use any font we want because the language doesn't care, the convention is that when writing source code we use a monospaced font where each letter uses the same width as opposed to a proportional font where a letter like i or l takes up less space than a letter like m. The reason is that in a monospaced font it's easy to tell when words and sentences across different lines are the same length. But is that important, you think. Well, yes it is. You see, when we write code we care about exact indentation and how our lines line up with each other as we're looking at them. And it's not just for aesthetics, it makes the structure of our code more apparent and lets us read source code and easily tell which part of the program belong together. That's why code looks like code.

Getting the Computer to Understand You

Back again to that phrase, a computer program is a list of instructions we write in a programming language the computer understands. I said the sentence was useful, but not entirely true, and it's this bit, the programming language the computer understands. Well because the computer does not understand our source code. We will use a language like C#, or Swift, or Ruby, or Python, but the chip, the central processing unit, or CPU, in any computer, laptop, desktop, phone, server, game console, this is the piece of hardware that takes care of running or executing our program and it doesn't understand our source code at all. The only thing that this understands is what's called machine code, or machine language, which looks like this. And these are the basic instructions like add one number to another number, but now written in a form the computer understands. Now, if looking at this makes your eyes glaze over that's totally okay, that is an appropriate response, this looks like gibberish. It's intended for machines, not humans, and you do not need to be able to read this. The vast majority of professional programmers never, ever deal with machine code. Now it is possible to write a program in machine code, there are a handful of specialized positions where it's a skill, but it's not a general programming skill. Writing machine code is incredibly time consuming, it's a long, error-prone, painstaking, and tedious process. Not only that, machine code is slightly different across every version of every chip, across different hardware. The entire point of having programming languages is so we don't have to write machine code. But, if we only write source code and the chip only understand machine code then that would suggest our source code must be converted into machine code before the program can run. And yes, that must happen. There's two main ways to do it. We will either compile or we will interpret the source code. And the difference between compiling and interpreting is not about what happens, but about when does that conversion to machine code happen. Here's what I mean. Imagine I've written a simple program. So I've written some source code on my computer and I want you to run my new program over on your computer. I've written source code, but your computer needs machine code. Option number one, compilation. I will take care of this, I will use a complier. a compiler is a program that goes through my source code file, instruction by instruction and converts it into machine code. The compiler spits out a new file containing machine code, so that's something your CPU can actually now understand and run, or execute. The result of the compilation is called an executable, or executable file. After I've compiled it I just give you that new file and you can run my program. So when compiling you won't get my source code, it stays on my machine, you don't even know what programming language I used. Option number two is we use an interpreter. Now, for this, I still have my source code, but instead of me compiling it I just give you a copy of it. You get my source code and that means you need to do the conversion to machine code and you will use an interpreter to do it. And that doesn't mean you have to go out and download an interpreter program, you will often find interpreters are bundled inside web browsers or even operating systems themselves, and they will be used automatically. Probably the most common example of an interpreter is that there is a JavaScript interpreter inside very web browser. So If just sent you some JavaScript you can run it because your browser knows how to interpret it. Now one big difference is that with an interpreter this conversion to machine code doesn't happen in advance, it doesn't happen until you decide to run the program on your computer. And at that time the interpreter goes through that source code instructions line by line an converts them to machine code instructions as needed, then runs that machine code, it doesn't save it as a separate file. Now there are genuine benefits and downsides to both methods. Compiled languages can often create faster programs because you've done the conversion in advance, but one downside is that because machine code is platform-specific, if you compile your program so it runs on a PC, that means it won't work on a Mac and vice versa. Some languages support compiling multiple different versions of your program for different operating systems. Now with interpreted languages, they're often cross-platform because as long as the person you're sending your source code to has the interpreter you don't care what operating system they use. But, interpreted programs can run slower because the conversion still has to happen in addition to running the program. And on top of that, with interpreted languages you also have to give you source code to whoever wants to run your program, and that could be both a benefit or a hazard. Now it is true there is a third option somewhere in the middle of this. Modern languages often try to compromise by having programs that can be partially compiled into an intermediate state, an in between state, what's often called intermediate language, another term is bytecode. It's a way that you can do as much of the compilation as possible in advance, but not quite all the way to that platform-specific machine code, leaving the final step, what's often called just-in-time compilation, is done on the computer of whoever is running the program. Now all of these options are important to know, but isn't something you need to worry that much about, or even have to make a decision on, because most programming languages naturally fall into either compiled, or interpreted, or in between categories. There are always edge cases and exceptions, but a handful of examples here of what's most common. For compiled languages, C++, Swift, C, Objective-C, these are typically just compiled. JavaScript, Python, and PHP, these are typically interpreted languages, and the best-known language that takes that middle ground/hybrid intermediate approach is Java. But, you won't choose a language just for its compilation model, You will decide on the language for other reasons, like what platform, or what type of application are you making? And then, you'll need to understand the way that source code in that language actually makes it onto the CPU. Having covered that, it's about time we dive a little deeper into some specific source code.

So What's the Deal with All the 1's and 0's?

Okay, let me do a quick sidebar here, because there is a question some of you may have at this point and I want to deal with it. But, Simon, you say, you showed us this machine code gobbledygook, and you said that is what the computer understands. But I've always heard that computers understand 1s and 0s, and that machine code did not look like 1s and 0s, what's the deal? Okay, computer chips, these CPUs, are, at their most microscopic fascinating, magical, mind-boggling, yet also slightly boring level made from billions with a B, of almost unimaginably tiny electrical switches, each one either on or off. If I could take this CPU and a scanning electron microscope, and there are videos on YouTube where people actually do this, and zoom in, and zoom in, and zoom in, we would eventually get to millions or billions of these tiny electrical switches or transistors on this chip, each one closed or open, on or off. To be clear, to be a successful programmer you don't have to know anything about CPUs, integrated circuits, transistors, or capacitors, or bistable flip-flops, that's hardware, we're software, different skillset. Like a surgeon needs to know how to use a scalpel or an EKG machine, but they don't need to know how to make one. What is good is to know enough about a CPU that we don't just think is a bunch of magical leprechauns making our programs work. And okay, I am simplifying, there is way more to a modern CPU than just lots of transistors, but still, it's different combinations of these switches, different groupings and arrangements of them that are used by the CPU together with other parts of our computer, like our memory modules, to represent number values, and represent text, and hold graphics, and audio, and all the programs that you're running at any moment, and exactly what they're all doing right now. And yes, that can seem puzzling, it isn't immediately obvious how we can take a bunch of tiny on/off switches and say, this bunch is now a Bach concerto and that bunch of switches is a picture of an alien spaceship. Well, okay, I can't show billions of switches, so let's begin with, for example, one. I'll use this symbol for a toggle switch, it's either on or off. And the important thing here is not whether this is connected to something or causes something to happen elsewhere, no, it's the switch itself, just by itself, because I can decide what I want this to mean. And what it represents is up to me, and it can be different in my program than in yours, as long as what it represents only has two options. In a computer program I might decide that this means a user is either logged in or not logged in. I could use another one to say we are currently recording or we are currently not recording. But what I can't use one of these for is anything more complex. I can't use one switch to represent a bank balance, or the result of a horserace, or PDF invoice, but I can add more switches, I can group them together because we have a ridiculous amount of them available. And if I have a group of two switches I go from two possible values to four possible values. Both of the switches off, one off and one on, the other way around, and both of them on. Okay, fine this is still simple, but whenever I add a switch, I'm not just adding two additional options, not just another on/off, I'm actually doubling the amount of values we can hold. Okay, when we go from just one to two switches, it's unimpressive, we need to take this further. So with two switches I have those four arrangements we just saw still very basic, if I add one more switch then I have these same four arrangements I had a second ago with this new third switch off and also those same four arrangements from a second ago with this new third switch on. So I've gone from four to eight possible arrangements of the three switches. Every time I add one switch I double the possible values again. All the arrangements from before with the new switch off, as well as all the arrangements from before with the new switch on. So it rapidly increases, always doubling. With 3 switches it's 8 possible arrangements, with 4 switches it's 16, 5 switches is 32, then at 64, then it's 128, and when we have just 8 switches there are already 256 possible arrangements of these, from all of them off, all of them on, and everything in between. I've been using these toggle switch icons, but these are getting to the end of their usefulness, I can't really fit anymore, but I still need a way to describe these arrangements. I suppose it's true, I could use words to represent them, but that's kind of clunky, or what I can do is use a 0 for an off and a 1 for an on, a very compact, concise way to do this. And it is 0s and 1s we use in computing, this is binary code, and we use it as a way to represent that most basic, most fundamental state of just part of a computer. There aren't really 0s and 1s bouncing around inside your computer, they are just ways we can represent or visualize those tiny elements being on or off. Now let me correct myself, I've been calling them a switch, but in computing we don't call each of these an elements a switch, we call it a bit for binary digit, the tiniest element of information that either 0 or 1. So, in all of these examples here I have 8 bits, 256 possible arrangements, from all 0s at the top to all 1s, and everything in between. So I could use a group of 8 bits and decide it represents some modest number I need to keep track of. Perhaps, the amount of enemies onscreen. As long as I know that number never needs to be more than 256, 8 bits is all I need to keep track of it. If I need to store bigger values I add more bits. And because computers really are built on this binary idea it is easier to deal in multiples of two. If I go from 8 bits to 16 we go from 256 to about 65, 000 possible values, double it again to 32 bits, we're already at 4. 3 billion possible values, with 64 bits it's, well, I don't know how to say this, but it's a lot. Now you may have even heard these phrases like things are 32-bit architecture or a 64-bit architecture, and that refers to the amount of bits that the CPU is most happy in dealing with and moving around. The larger the better, typically. No modern CPU just wants to deal with one measly bit as a time, it's just not that useful. But here's the thing, 64 bits can represent this huge number, but I don't have to use these just to represent a number. Because I could also take this same 64 bits, but decide will use this particular collection of information to represent an 8x8 pixel area of the computer screen and use the 0s and 1s to turn 64 specific pixels on or off. And that's how we can take a few on-off switches and use it to represent a picture of an alien spaceship. So, back to that machine code, why isn't this 1s and 0s? Well, it kind of is. Let me explain that with bananas. If I'm a grocer and at the start of each day I count the bananas I have for sale, whatever that amount is I could write it down with words. Twenty-seven. Or I could use decimal digits, a 2 and a 7. Or Roman Numerals, XXVII, or I could marks on a chalkboard. Or I could use my hands, or I could use little pictures of bananas. Whichever one I use, they all mean the same thing, none of these values are approximate, they're all exactly 27. They are different methods to convey the same information. We will use whatever method is most useful at the time, usually which one is going to be the most concise and compact way to do it. And that's what's happening here. Machine code can be shown in binary, but it's more common to use a format called hexadecimal, or hex. We can view more information in a smaller area using hex than using binary. If I have any of these groups of 8 bits and I need to describe those arrangements, with binary I need 8 1s and 0s. But using hex I just need two digits, because each digit of hex can have 16 values, from 0 through 9, but also a through f, instead of just the two different values, 0 and 1, that we get with each binary digit. So this in binary becomes this in hex. So when we see machine code it is the specific details of exactly which bits are 0 and which bits are 1, it's just using a more concise way to display it. Now right now, we do not need to get into the details of writing binary, or hex, or translating from one to other, it is not necessary when you're beginning, but do know that in the rare occasion where a programmer needs to look at information so specific, it is usually more convenient to view it in hex than view it in binary, and that is absolutely as deep as we need to get into this right now. End sidebar, time for the next module.

The Rules of Programming Languages

The Rules of Programming Languages: Introduction

Every programming language has a set of rules. How you write it, what words exist in that language, what order those words need to be written in, whether they're written in uppercase, or lowercase, or a mixture of the two, or does it even matter, whether each instruction needs to end with a line break, or a semicolon, or something else. These rules, and they are rules, not guidelines or suggestions, they are the syntax of a programming language. Now you can disagree with the rules all you like, you can privately think some of them are stupid, but if you want to write a program that works there are always rules we must follow. And many of the rules are very similar across multiple languages. So with just a few basic questions we can start to narrow down the characteristics, the things that make one programming language different from, or even similar to, another. We'll begin by covering five things we can ask about any programming language. First we'll talk about case sensitivity and take care of a few questions you may already have about what it might mean when source code that you've seen is uppercase, or lowercase, or mixed. We'll then see common ways to write our instructions, or our statements, in a programming language, and particularly how you must end each instruction. Whitespace is next, all those blank lines, indentation, and spaces you see in source code, why this is important, and who this is important to. And then comments, adding your own notes in the middle of your source code. And we'll talk about keywords, those words that have a special, fixed meaning within each programming language. It won't surprise you that different languages have different keywords, but there are also quite a few keywords you're likely to find in every single programming language. Now these five things aren't the only aspects of a language I could talk about, I could also ask whether a language is usually compiled or usually interpreted as we covered in the previous module. But whether a language is compiled or interpreted is not obvious just from looking at it, and right now I want to focus on what we can realize, what we can start to understand, just from looking at and beginning to read the source code of a few different languages.

Case Sensitivity and Capitalization

Almost all programming languages these days are case sensitive. When you're writing source code, whatever that language is, and there's a word like while or true, if your language expects that word to be all lowercase then it should always be written that way. Because when you try to run a program you must take your source code and expect it to be compiled or interpreted, and if you have that word, for example true, then according to the rules, according to the syntax of that programming language, should be written all lowercase, and then you write it with an uppercase T, either accidentally or intentionally, then the compiler won't say, oh, it's okay Simon, I know what you were going for, it just won't work. It's not like writing an email where you can make a few mistakes with your capitalization and still expect to be understood, no, this is programming, and these are rules, not guidelines. Okay, there are a few, typically older programming languages, like some version of BASIC or Pascal, which don't care, and you can use any combination of uppercase, or lowercase, or mixed-case letters in your source code. But these days, begin by assuming that every programming language is case sensitive and is very, very picky about it. It's common when you first see some source code examples, even before you've started writing any code that you might think, okay, I don't yet know exactly what this source code is doing, but this language here seems to be mostly lowercase. But then, you see a few words with an uppercase first letter, or perhaps reading more code find the occasional word written using all caps, and it can be confusing when you don't know why. Does this actually mean something, or did Fred just leave the Caps Lock key on when he was typing this? Well, it's true there is usually a meaning to those differences, just as in written English where we will capitalize different words based on the type of word they are. We take a regular sentence like, next wednesday, i will drive you to the airport to fly from phoenix to london. In correct written English I'm supposed to capitalize the opening word of the sentence. I should also capitalize days of the week and proper nouns like London and Phoenix. And in English we're supposed to capitalize the word I, but we don't capitalize the word you, or we, or they. But in a different language, like written German, that little piece is the other way around, we're supposed to capitalize the word Sie for you, but not ich for I. I point this out not to get into language semantics, but to show that yes, just as in real-world languages you are going to find differences in what programming languages choose to capitalize. As one example. In Python the word True is capitalized. In Swift true is all lowercase. Now this is not random,, it's that different languages choose to emphasize different things, just as English and German do. When you're looking at a whole bunch of languages, as we're doing here, these differences can seem daunting, but in practice they're not. Because when you start to write source code you're going to pick a language, and then it's easy to deal with this. Here's why, back to this sentence. Imagine asking a room full of people why in written English do you capitalize I but we don't capitalize the word you. A lot of people are going to shrug and say, I don't really know, I write it this way because I always read it this way. And that is a perfect way to begin writing source code. If you start writing, for example, Swift and think, you know, I'm going to write the word true with a lowercase t because whenever I see it in Swift it always seems to have a lowercase t, and I'm going to write the word string with an uppercase S because whenever I see it in this language it seems to have an uppercase S, that's a great way to begin. Are there some deep syntactical reasons and theoretical explanations we could get into to explain the differences in capitalization between programming languages? Sure, but those things just aren't that important right now. If you were learning how to write English and you wrote you first sentence like Hello, i am a programmer. What you'd want is somebody saying, hey, you should capitalize this I. What you don't need with that first sentence would be a long lecture on the history and theory of capitalization rules of the singular first-person pronoun and all its contractions, there will be plenty of time to have those discussions, and better times for them. Right now, the most important thing is consistency, to recognize that whatever language we might be using that the same word with the same meaning is always capitalized the same way. And, here's one thing that makes this even easier. In programming we don't have rules like in written English where this word next is not capitalized, oh, but if you put it at the start of a sentence then it is. No, it is simpler here. In a programming language, it doesn't matter if a word is at the beginning of a sentence, if it's lowercase it's always lowercase. And, actually, we don't talk about writing sentences in a programming language, we deal with the idea of writing statements, and it's about time we talked about those statements.

Writing - and Ending - Statements

I've talked a lot about us writing many small instructions to the computer, add one number to another number, or display the word hello on the screen. When we actually write source code we call each of these a statement, that's a few words or symbols arranged to make one self-contained specific instruction. A statement does something, it's code that has an effect in your program. And that means a statement must be complete, it must make sense in the same way that a written or spoken sentence must make sense. If I said change the color of, you'd wonder, change the color of what exactly? Or add 99 to, or print, or move. You'd understand these words fine, but you'd recognize these as being incomplete. So there's a difference between just knowing words of a programming language and writing a statement that uses those words. Now usually statements are just a few words. They'll often fit easily on a single line in your source code to say what is being done and what is it being done to? As examples, change the color of the background to light blue. Add 99 to whatever the current score is. Print the current document. To where? Well, the default printer. Or move, move what? Move the spaceship graphic. How far? Five pixels. Which direction? To the right. Again, specific, small, self-contained instructions. Now occasionally, a statement can be just one word. the same way that you can have a complete sentence in English of just one word, although that's usually in response to something. When are we meeting? Tomorrow. What's your favorite color? Red. When are you leaving? Now. For a computer program the equivalent one-word statement might be what do we do when we finish converting this file? Quit. What happens when the user presses the wrong key? Beep. But most of the time we need a few words, a few pieces to our statements. So these are the kinds of statements we can write. But as you can probably tell, I'm not writing them here using any particular programming language, just this informal description. So let's just take 1, to add 99 to the current score and see this written as a statement in a few different programming languages.

Now these all assume that whatever language I'm using, maybe we're writing a simple game, we've decided we want to keep track of and change a number that we'll call score, just as someone might scribble a score on a chalkboard and change that score while a game is being played. So let's begin with an older language. In the programming language COBOL this statement could be written as ADD 99 to score. In AppleScript, a different language, we'd say set score to score, whatever it is right now, + 99. Same intention, same result, but a different syntax to do it. And a lot of other programming languages adding 99 to score is written as score = score + 99. Sidebar. This format, this syntax, is a very, very common way of doing it, and it can confuse people new programming who think that it looks like one of these algebraic equations they have to figure out somehow. They have flashbacks to high school and think, oh is this one of those things where if x is 99 and b over c is 52, then how many apples fit on a train going to Chicago? No, it's nothing like that. This is our statement, this is our instruction. We're not asking anything, we are telling the computer to do something here. The equal sign in this statement is not a question, it is a command. You can think of this equal sign here as meaning, is now made equal to. So when we execute this one statement, score is now equal to score, whatever is was previously, plus 99. And in a lot of other languages, the syntax is almost identical, except we need to add a semicolon at the end. You see, no matter what language, it's very common to write every separate statement on a separate line in your source code file. Now, sometimes a statement can be split across multiple lines, usually that's not required, it just makes things easier to read. But, in many languages, having them on separate lines is not enough, you must also explicitly end each statement with a special character. One that says, this statement is now complete, like having a period or a full stop at the end of a sentence in English. It's what can tell the compiler, the statement is finished, go ahead and convert it into machine code. The most common character to end a statement is the semicolon. you'll find this used in C and in many languages based on or influenced by C, like Java, C#, PHP, Objective-C, C++, it is very common to see this semicolon. Whereas in COBOL, or at least some versions of COBOL, it's the same idea, but there you would end a statement with a period. In other languages, Ruby, Python, Swift for example, you don't need a special character, you just end each statement with a line break. You don't need a period, you don't need a semicolon, just by having separate lines, each one is treated as a complete statement. And again folks, right now don't worry about memorizing what language exactly does what, that's not important here, we're working on recognizing what all these languages need to do. They might do them differently, they might have a slightly different syntax, but they all do them. Because, even without memorizing any of the syntax, you start to recognize that a lot of this is shared across languages. If you learn how to add one number to another number in Ruby, it's going to be the same in Python, and the same in Swift, and almost the same in C, and Java, and PHP, and many others. So let's move on. I'm going to talk for a moment about the benefits of writing our instructions so that they're not targeted at one specific programming language. And then we're going to move onto the syntax that you don't see, the invisibles, the tabs, the blank lines, the spaces, the whitespace of a programming language.

Using Pseudocode

From the start I phrased many of our instructions in plain language. Display the word Hello on the screen, add 99 to the score, and so on. And there is value to this, when we write out individual steps, but first, just write them informally, more for humans than computers. There's a name for this, we call this pseudocode. It's sort of, kind of, like code in that we are writing small, specific instructions, but it's not actually source code, not yet. There are several benefits to this, it's a way of communicating, it's a way of thinking, it's a way to make progress when you're stuck. Pseudocode lets us work a problem without being distracted by their persnickety syntax of a programming language. And pseudocode is not just something for beginners. Many people make that mistake. They think because it looks so simple this must be the programming equivalent of training wheels on a bicycle, but in fact, this is something that's incredibly valuable even when you've been programming for decades. And a lot of people underestimate and even dismiss pseudocode when they're beginning, they want to run straight to a code editor, or start typing lines of source code as soon as possible. And that's completely understandable, but, if you remember right at the start I said, success here is all about how you think, and pseudocode is a way we can get thinking that way quicker. Because jumping straight to writing source code often just ends up with you staring at a blinking cursor and thinking, um, I don't know what to type here. And you'll find when you're programming that often the best way to make progress, the best way to get yourself unstuck is to step away from the computer, pick up a pencil and paper, or grab a whiteboard. Now when we're using pseudocode we don't try and write our entire program on it, we use it moment by moment. We go through parts of our program and choose to use it piece by piece to get better understanding. An example. Let's imagine I want to write a game, and whether it's just me or me and a few others writing it, we're thinking about what should happen when the game ends. So rather than typing code it's much better that we take a few minutes, grab a whiteboard, and write down that on the game completion we want to, well first we'll display a congratulations message. And then we should play a fanfare. After that we better reset the game, and don't forget we need to save the score. And then someone points out that if we want to save the score we better do that before we reset everything. We're starting to break things down into these smaller steps, as well as we can. It's not source code, but it's not regular written language either, it's somewhere in the middle. We don't worry about exact programming language syntax, but we also aren't trying to keep to written language syntax either. Pseudocode is not about correct punctuation or grammatical structure, it's usually very simply written. When I'm writing pseudocode I use a lot of caveman-style verb/noun phrases. Save score, open file, reset game, play fanfare. And it's true someone else might write this differently, perhaps being a bit more specific, like saying play the fanfare mp3 sound effect, instead of just play fanfare. The point is there is no one true way to write pseudocode. This is one of the few places where there aren't any rules. Pseudocode is not an actual language, there is no official syntax here. And because of that it can look very different. Pseudocode can be very generic, but experienced programmers often write pseudocode that's close to their favorite programming language. That's not really a surprise. If you've written C for 20 years and you write pseudocode, even on a whiteboard, it probably looks like C, that style of capitalization will come easily to you. You might even have handwritten curly braces and semicolons. There's no need for them, but it's force of habit. Or if you've written Visual Basic for 20 years, your pseudocode probably looks a little more like that. But this difference doesn't matter that much. One benefit to pseudocode is it's a way that programmers can get together and work through a problem, even if they don't use the same programming language, of if they're at different skill levels, or even if they haven't decided which programming language to use. It's even a way that I as a programmer could work through and make progress on a problem together with a nonprogrammer, and it's a fantastic way to get unstuck, particularly when you're not entirely sure what to do next. And that should be very important to anyone who's just beginning this. For example, let's say the game I'm writing is my first one, and though I may have been programming for years, I have no idea how to actually play a sound effect. And it is very common for programmers to have a lot of experience in a language, but still have specific tasks they've never done, just because they've never needed to. But by writing pseudocode I don't get stuck here. I know this can be done, I know sound effects are possible, just maybe not the exact syntax, but we can continue on, I can think through the rest of the steps, I can talk about other parts of the game, and perhaps tomorrow morning, my first task for myself is now, find out exactly how to play a sound effect. And that's a much more achievable task than staring at a blinking cursor and thinking, now what am I supposed to write? Now it is true that sometimes one line of pseudocode can end up expanding into multiple lines of actual source code. Now that's okay, it's not about expecting perfection. What we're trying to do here is think about a larger problem, break it into those individual steps and think about the sequence of those steps and the structure of our problem, and it allows us to do that. Even when it's not perfect, pseudocode can always save us time, and I'll be using it a lot going forward.

Dealing with Whitespace

Next, we can pick a language and ask how sensitive is this language to whitespace? Whitespace is any character that is typed, you'll hit a key on the keyboard to get it, but not visible in your source code, like a space, or a new line, or a tab. Because even in the few snippets of code I've shown so far, you'll notice, it doesn't matter what language, there always seems to be blank lines, there always seems to be indentation. So the question is, is all this whitespace significant to the syntax of a language, does it mean anything? And by that I mean, if we change something about just the invisible elements, say we move these statements over to the right, or we add several more blank lines, would this code behave any differently from a moment ago, or is this whitespace cosmetic with no effect? Well most programming languages don't care very much about whitespace, they care a little bit. If you're writing an instruction, and I'll just use pseudocode here, it shouldn't be a surprise that you can't put a line break right in the middle of a word, or take all the spaces out and join everything together. The language needs to be able to tell one piece of your statement from another piece of your statement. Beyond that, with most languages, it doesn't matter if you have more than one space between the elements in your statements, or if you add extra blank lines in between them, or even if you add extra spaces on the left and the line moves in. The language doesn't care, it just ignores it, this all works the same way. However, make no mistake, whitespace is still incredibly important, but it is most important for us, as programmers; the computer might not care about it, but we do, we use whitespace, blank lines and particularly different amounts of indentation, all this formatting, to make our programs more readable. You might think, well who cares if the source code is readable? Well, we do, because we'll spend a lot of time in it. Complex programs will take us weeks, months, sometimes years to write. We'll need to fix bugs in that code, we'll need to add new features, we might need to go back and try and make the code faster. We might have to change code somebody else worked, and someone might need to change code that we wrote. We want that source code to be readable and understandable, whether that's tomorrow, or 6 months, or 5 years from now. However, in a few languages, a few situations, whitespace can be more significant than just ease of reading. In the language Python, for example, having whitespace at the start of a line is not just cosmetic, it matters. There is a difference in behavior if I take this last statement and indent it. Because in Python we use indentation to tell the computer that certain lines of code, some statements, belong together in a group or block. So in this case, indenting this line says to Python, these three statements are to be taken as one block and these three statements will only be executed if the player score is greater than the high score. If the player did not get the high score we will just ignore all three. But if that last line was not indented then the block, the grouped statements, are now these two indented ones. So if the player score is greater than the high score we will execute these two, then keep going, and execute the last line, but the difference is, now we will always execute this last line whether the player had the high score or not. Okay, we're jumping a little bit ahead of ourselves here, but this idea is common across all languages, that although we write individual statements, individual instructions, we will choose to group them into different blocks of code. So we have more control and flexibility about how our program works and be able to change the behavior of it while it's running. So Python uses indentation to mark a block of code. Many other languages, including the C-influenced languages like C++, Java, C#, or Swift here use opening and closing curly braces to describe the same idea of a group of statements, a block of code. In these languages we will also use indentation, we'll use whitespace to make things readable, but here it is the curly braces actually doing the work of saying where this block of code begins and ends. If I change the indentation of a statement inside this block in the Swift language it has no effect on the behavior, I would need to move this outside the block if I didn't want it to be grouped with the other two statements. All right, we're going to talk more about blocks in an upcoming module on controlling program flow, but let's get the last couple of syntax topics covered first.

Commenting Your Code

It's important that source code stays readable. One way is good formatting, using indentation and whitespace. And this will become more and more useful and more familiar as we go deeper. But another direct way is to add comments to our source code. We add our own notes as to why we're writing these statements, because source is not always obvious. When you're starting out it's no surprise that you need to take your time and slowly figure out each statement line by line, and that does become easier, but let me assure you it is not unusual for experienced programmers to look at a screen full of source code and think, hang on, what is this bit doing? And a well-placed comment can make all the difference to understanding, to having that mental lightbulb go off, okay, I get it, I see what this is doing, fine. So we want source code to be well commented when we're reading it, and that means we comment it well when we're writing it, even if you think it's only ever going to be you who will read or write that source code. Because I guarantee you, that if you open up source code you wrote 6 months ago, you will look at some of it and wonder, what was I thinking here? One of the most normal ways to add comments, and I'll add some to some intentionally-blurred source code here, is just by beginning a line with two forward slashes. This is the syntax for adding a comment and programming languages like C++, Java, Swift, C#, PHP, JavaScript, Objective-C, and several others; the compiler, or interpreter, ignores the rest of these lines, you can just write whatever you want, and you can write it however you want. You could use slang or terrible spelling, it won't matter because the computer does not care about what comes after the two forward slashes. you see, a comment is not a statement because is doesn't do anything in your program, it's not actually an instruction, a comment will not be compiled or interpreted into machine code, a comment is only for our benefit. It stays in our source code to help us understand. There are other variations than the two forward slashes, Ruby and Python use the hash or pound character at the start of the comment, Visual Basic uses a single quote, Ada uses two dashes, but the idea and the purpose is the same. And we don't just write a bunch of comments in one place and a bunch of source code somewhere else, instead we salt and pepper our source code with small inline comments, a line here and there, wherever we feel they are useful, and usually you write a comment directly above the code it's referring to. But you don't have to comment everything. Some source code really is self-explanatory. For example, if you had a basic statement that will print a Thanks for playing message you don't then need to add a comment above this that says we're now displaying a message that says Thanks for playing, the code really does speak for itself. It's true that as you get more experienced in a language then more and more lines of code will seem self-explanatory, but early on, if you think a comment might be useful then just add one. They have no detrimental effect on the behavior or the performance of your program. Sometimes one line isn't enough. If you need to write long comments you can either write several individual comment lines, or most languages support writing a multiline or block comment. In C-style languages that's done by opening with a forward slash/asterisk, and then everything after that will be taken as a comment over multiple lines, whether 2, or 3, or 100, until we write the closer of this block comment, which is an asterisk/forward slash. But while we use comments to add explanation, to add clarity, they can also be used to provide reminders, marking places in source code someone intends to revise, improve, or add to. It's normal to write a few lines of code and realize it can't actually be finished yet, you're waiting for something, waiting for more information, or perhaps waiting for some kind of asset from somebody else on your team. For example, you might know that you need to show some kind of image or graphic, but you don't have it yet. So rather than come to a standstill when writing code we might write a comment like TODO: display custom graphic here. And this comment becomes easy to find later. Now you can invent your own way of marking these kinds of reminder comments, and while there's nothing official, a few habits or conventions have evolved in programming over the years, and it is quite normal to see comments that begin with a small, single word all in caps, a tag like TODO, or sometimes FIXME, or even HACK where you're commenting that a piece of code is using some quick and dirty workaround, it might work, but it isn't the best way and it really needs to be revised. Now there's nothing formal, nothing magical about these words, these tags, it's simply that they become easy to spot and easy to recognize when reading and editing source code. We can also use the comment syntax to temporarily disable some of our regular statements. Because when you're writing and testing your programs it's often convenient to briefly exclude certain statement from running. Perhaps here I just don't want to hear the fanfare playing every single time I test this code, but I don't actually want to delete that statement and then have to rewrite it again later, so I just comment it out, so the compiler now ignores this line. And when you're using a programming text editor, one that supports color coding, comments will typically display in a different color, so they're easily distinguishable from the regular statements. Different editors show different colors, but it's an important distinction, we don't apply this color or this change, it's not like a word processor or a desktop publishing software where we'd select a line or a word and choose to make it red, or blue, or green. No, in a programmer's text editor the whole point of color coding is that it happens automatically, so we can recognize different elements in our source code. And when using a programmer's text editor like this, you'll usually find different colors applied automatically, not just for comments, but also for numbers, or text, or keywords in the language. But we haven't talked about keywords yet, so let's do that now.

Keywords and Our Words

One common misconception I want to clear up, you see, programming languages, while they are very strict are nowhere near as complex and substantial as spoken or written human languages. If you are a native English speaker and you wanted to learn Japanese, or Norwegian, or Mandarin, you'd expect to learn hundreds or even thousands of words before you'd be considered fluent. Programming languages are much, much smaller than that. Take Java. Java has about 50 keywords. A keyword, and here's a few examples, is a word that when you write it in your source code, the language itself says, okay that word is mine, I know what that means, I know what it does. A language like Swift, C++, somewhere between 50 and 70 keywords. Python, a famously concise programming language only has about 30 keywords. Now the exact number is not very important, and I don't mean to suggest that if you learn 30 words you are now a Python expert, not at all. It's not just the keywords but learning what you do with them and with combinations of them. Keywords are the words that belong to the language, but they aren't the only words we'll need in our code. We'll also bring our own words to the table. Here's what I mean. When I'm writing a program, whether a business app, or a game, or anything else, I could pick a programming language almost at random and whatever it is I would expect that language to already include a bunch of keywords and also symbols that I can write in my source code to allow me to do things like add one number to another. Sometimes we'd use a word like ADD in COBOL or set in AppleScript. Sometimes it might just be symbols, or what we call operators, the plus and equal signs. I can also expect to way to ask a question, are two values the same, or is this number greater than that number, most often it's the keyword if to do this, we're asking a question, we can tell the program to do one thing if it's true and another thing if it isn't. I also expect to find a way to write a message to whoever is running the program. And of course there are other keywords, particularly for managing the flow of a program. So we can choose to execute different pieces of our code moment by moment, and we will get more into this later. So while there might be different names, different keywords, we still find all this basic generic ability, it's already there in every language, all these words were there before we came along. But we'll need more than this. Because if I'm writing, say a game, I might want a word in my source code that represents the spaceship of my lead character, so I could then write code to move it, or rotate it, or in certain circumstances make it explode. Or if I'm writing a business application for my company I might want a word that means the list of our current international office locations grouped into which ones do or do not have kitchen facilities. But you can search all you want, the keywords in Java, or C++, or Python do not include the word spaceship, and likewise there is no office locations keyword in Ruby, or JavaScript, or Swift. But the thing is I still want those words in my source code. If I'm writing a game I want to be able to write code to make the spaceship explode, even if these words don't naturally exist in the language. If I'm writing a business app spaceships aren't important, but instead I might want to write code to change the paygrade of an employee or generate a report, and I wouldn't expect these exact words to already exist in a programming language, because what this means in my program is not the same as what this would mean to somebody else in a different organization. So when the words we want aren't in the language we can still have them in our source code, but we will define them ourselves, and we will do this all the time. An example. This is a little bit of Swift that might be found somewhere inside the source code of a game. So this word here, if, is part of the Swift language, it is a keyword. it wasn't up to us, it was already there, and we cannot change what it means. And if is a keyword you will find in virtually every programming language. But words like these, currentScore and highScore, these are not keywords, they are not part of the Swift language, these are my words, my choices for two pieces of data, pieces of information I wanted to keep track of in this imaginary program here. The fact that these exist is up to me, and I could have called them something else. And this is what I mean about the difference between a keyword, which has a predefined fixed meaning in a programming language and using many of the other words that we will define ourselves within each program. But they do have to be defined somewhere else so I could use them here. And this begs the question, how do we say what these new words mean? How do we tell the computer whether these are numbers, or text, or pieces of audio, how do we name them? And that's the next module. What data, what information are we going to put into our program, and what information are we expecting to get out of it?

Working with Data

Introduction: Input, Output, and Everything In-between

As soon as we start to write code we deal with input and output, we deal with data. And I know, I know, these words input, output, and data are bland, they're so generic, so well-known. It's easy for them to sound former and lifeless, this gives me visions of 1960s computer rooms. It's an invigorating day in the data processing center. There's a bunch of new input to take care of, and boy oh boy, everyone is so excited to process that data and generate some output. But the reality of these ideas is much more interesting because at its most basic sense, when we talk about input, output, and data, we mean anything and everything we're going to put into our program, anything and everything we're going to get out of it, and whatever we need to figure out along the way. So, first, think of all the things we can physically do with a computer while a program is running. We touch the keys on the keyboard, move the mouse, we can plug in a joystick, talk into a microphone, use a web cam, plug in a music keyboard, or alternative controllers, if you have a phone or a tablet you can swipe on the screen, or tilt it, and shake it. And any of these things can be input for our program, if we decide that we care about that kind of input. And then there's also files on our hard drive, documents, images, music, or video. Those could also be input. And often, we will have multiple types of input. Think of something like iTunes, or Windows Media Player. We can use them to read music files in from the hard drive, one kind of input, or we can stream music from the internet, a different kind of input, and at the same time there's also input coming from the user. Press play, press pause, adjust volume. And with output we've already seen a simple example. Can we output a message to the screen? But being able to take that concept a step further is what lets us show lines, and then boxes, and user interfaces, or even further take over the entire screen with 3D worlds. But there are other ways to output then to the screen. We could have our program create a document and then save it out to the hard drive. We could generate an email and send it out to the internet. We could send sound to computer speakers, we can send output to a connected device, like a printer or a robotic arm. There is incredible variety to what is meant by input and output. And that's something to stay conscious of as a programmer, because here's the great thing, we don't have to think about language syntax to just ask that question, how do I need my new program to communicate with the outside world. How do I want it to interact with something outside of itself, whether that's human being, or another computer, or a device, like a printer. Because this gets us one step closer to what needs to happen in the code. If this is the input I want to provide and this is the output that I want to have, how do I turn this into that? If someone taps the left arrow on the keyboard, how do I then change the position of a character on the screen, and by how much? If someone clicks a specific button how do I send a document out to the printer? And all of this, whatever's coming in, whatever's going out, and importantly, whatever we need to figure out in between, we are going to have to hold at least temporarily inside our program so we can look at all this data, ask questions about it, change it. Okay, data is one of these words that's used a lot in computing, and often as part of another term like database, big data, data modeling, so, to be clear, right now I'm using the word data in its most generic sense, stuff. Any information that we care about, whether that's input, or output, or anything in between, but we need to say, in our source code exactly what data, what information is important to us right now, in this program, in this moment. And in the next few clips you'll see exactly how to do that.

Creating and Naming Variables

If we could look inside a program while it was running we'd see many individual pieces of data. The email address of a member, the score in a game, the URL of a website, the date of an event, whether a user is logged in or not. In your program you do this by creating variables, that's how your program can claim a piece of computer memory to use while it's running. Give it a name, and then you can then refer to that piece and use it in your source code. And there are three things to think about with each variable, its name, its value, and its type. So back to the idea of keeping track of the score in a game. If this were a variable, the name would be score, the value might begin at 0, then change to 100, or 1000, or 100000. So it's like having this little box, this container that we've given a name to. The name stays the same, but the value inside can change from moment to moment, it can vary, which is why it's called a variable. And then there's the type, what type or what kind of data do we want to put in each variable? Will this be text or is it numeric? Is it a simple yes or no, true or false? Or is it a currency amount, or a million email addresses all grouped together? Is it our company logo? Programming languages care deeply about the type of each piece of data, because it affects how much memory is needed for every variable. If you make a variable and in your code you can tell the computer the value of that variable will only ever be true or false, but nothing else, the computer only needs to set aside a tiny piece of memory to keep track of that. But if you want another variable that could contain all the text from the first 28 chapters of War and Peace, or hold all the sales numbers from the last quarter, or hold an image, or a piece of video, then the computer needs to grab a bunch more memory to store something like that. So here this new variable score will be just a simple numeric value. We might change that value, but it will always be a whole number with nothing after the decimal point, it's an integer. So if that's the idea, how do you actually do this? Well in many languages you must declare a variable before you use it. What that means is, you write a statement in your source code to say, you want to make a new variable, you need to say what its name is, and often say what type of data it is. And only after you've declared it can you then give it a value and start changing that value. The exact syntax for this of course differs between languages, but here's a few examples. Once again I'll point out I am not looking for you to memorize any of this syntax, just observe, begin to notice some of the similarities and the differences. In the Swift language, var is the keyword to create a new variable, score is just our name for this new variable, and the : Int is the Swift way to say this new variable must always be an integer. And any attempt to put in a different kind of value like a date or a piece of text would cause an error in my program. If I'm writing Visual Basic it's Dim score As Integer. Dim is an older term, it's short for dimension, a way to tell the computer, go grab an area of memory for a new variable, and As Int is the type of data we have. In C++ it's a very short succinct format, just int score. We don't need a keyword like var or Dim because in this language just using the word int this way makes a new integer variable called score. And in JavaScript it's var score. The word var, similar to Swift, to declare the variables, but unlike the other three we don't have anything in JavaScript to say this is an integer, and that's because JavaScript is a language where you don't have to provide a type for each variable. But in all of these, the word score was totally my choice. This could have been player score, or current score, or whatever else I wanted, and this name, what is called the identifier of each variable is up to us. But still, within each language there's two things to know, what is allowed when you're naming variables and what is expected then you're naming them? There are a few rules, things you just can't use as a name and would cause an error if you tried, but on top of those rules there are also guidelines for how you should name things. A simple example. And this is true in most languages, but I'll use JavaScript here, there is nothing to stop me declaring a variable called current score, but writing that name, that identifier, in a mixture of uppercase and lowercase with a 0 instead of an o, it does not violate the syntax, the actual rules of the language, meaning I won't get an error if I do this, but it's not accepted style. And if I were on a team of developers I'd expect the others to say, what are you doing here, why did you write it this way? But let's cover the few rules about naming first, and yes, there are always going to be exceptions and edge cases, so assume that right now I'm going to show you what's true for most current popular programming languages, most of the time. First, no reserved words. You can't name a variable using a word the language already owns. You can't create a variable called if, or a variable called true, or any of the other keywords that that language wants to use. Beyond that, most languages support naming your variables using any combinations of letters, numbers, and a few special characters like underscores, just written in all lowercase like score is very common to see. And bear in mind, as we've talked about, most languages are case sensitive, so there's a difference between a variable named score with a lowercase first letter and a variable named Score with an uppercase first letter. Now there's nothing that actually forces you to use real words, it's just good practice to make your variable names readable and understandable. In a few languages you see special characters at the beginning of each name, each identifier, like underscores, hash, or pound signs, and PHP for example requires the dollar sign. Beyond that, in most programming languages you can't use spaces in a variable name, the identifier must be all one piece. So if you have multiple words there are different styles to keep them together, and we're going to talk about that in just a second. Some modern programming languages go beyond just allowing the Roman alphabet and they support other alphabetic systems, even emojis. And okay, you wouldn't actually create a variable called smiley face/ frowny face, it's simply a side effect that in some programming languages you can use anything that's allowed on a multi-national keyboard, whether that's Mandarin, or Hebrew, or Arabic, or emoji. But one common syntax restriction is that while you can have numbers in your variable names you cannot begin that name with a number, because, if you could then you could also end it with a number, and you could create variables that just looked exactly like numeric values like a variable called 99, which would be confusing all around. So with these rules in mind, here's an example of what I mean by a style guideline, that across many languages, a common style, a common convention for naming a variables is to use Camel Case. So if the variable name, the identifier that you want is just a single word, like score, or name, or department, it's just written all lowercase. But often a single word isn't good enough, it's not specific enough, because you want your variable names to be obvious, for anyone to read that name and immediately understand why it exists. So we often need two or more words. Let's say score isn't good enough, we need both the player score variable and a high score variable, then we just put those words together and we would capitalize every word after the first. So it's like the hump of a camel, so Camel Case. An alternative style would be use underscores to separate multiple words. And in many languages you could technically write it either way, you could use both of them, you could write them all uppercase, all lowercase, there's nothing that would stop you, but you'd find that a typical style, like Camel Case becomes the convention, the standard way to do it in each language. And adopting that standard style makes it much easier for you to read other developer's source code and for them to read yours.

Using Variables and Operators

The entire point of declaring a variable, of giving it a name, of deciding what it means is that we then actually use it. And the first thing we typically want to do is provide an initial value. So here's the examples from that last clip, declaring that variable. In almost all languages, not quite all, but pretty close, the way we do this is using the variable name, then an equal sign, and then the value we want to assign or set that variable to. And as I mentioned in an earlier clip, when you see this single equal sign it is a command, it is not asking a question, it is telling the computer to do something. Score will now be assigned a new value, the value 0. And in programming you'll sometimes see the single equal sign referred to as the assignment operator. It sounds very formal, it really isn't. An operator in a programming language is a way to perform one very small, specific task, one single operation, like adding, or subtracting, or multiplying. We don't always use a keyword like if or while, sometimes we just use a symbol. So a plus sign is the usual symbol, the usual way to add two numbers, it's how we perform an addition operation. So this plus sign is the addition operator. A minus sign is the subtraction operator. And we have operators for multiplication and division, usually the asterisk to multiply and the forward slash to divide. And all of these operators require two items to work on, something on the left and something on the right. They don't make sense without two. And any operators that require two values are called binary operators. This doesn't mean binary as in the 1s and 0s, it's just the simpler idea of the word binary just meaning 2, they will require 2 values or 2 operands, one on each side. And what's on each side is often very important, because if we're subtracting we're taking the value on the left and subtracting from it whatever's on the right. If the same values are just the other way around, this operation has a different result. So back to the equal sign. This is the symbol, the operator to assign a new value, it is the assignment operator. When this statement executes in our code, whatever's on the left side of the equal sign will now be assigned the value of whatever's on the right side of the equal sign. And that means whatever's on the right-hand side of the equal sign doesn't have to be a simple, single value like this, it can also be an expression, something that the computer has to evaluate, to figure out, that could even use other operators. So the computer first evaluates everything on the right-hand side. It will see this plus sign, perform the addition operation of 105 + 42 and assign the result 147 to the variable score. So we declare variables, then we assign to them. And when giving a variable that first initial value you can either write them as two separate statements like we've seen here, first declaring the variable and then the next statement to assign a value, or in most languages you can combine these as one statement that both declares the variable and assigns the initial value. And after that we can write more statements to assign new values to this variable again, and again, and again, always evaluating whatever's on the right side of the equal sign first, which could be an expression, and that expression can include other operators, multiple values, other variables, and then assigning the result of that to what's on the left. So operators are everywhere in your source code. And these aren't the only operators we can use, but many of the operations and the operators that make sense all depend on how you have defined your variables, what type of data they are. so let's get a little deeper into that.

Choosing and Using Data Types

All right, we're going to talk about data types, the idea that every piece of data coming in, and out, and through our program has qualities, characteristics, not as actual value, but rather what kind of or type of data it is. A lot of people new to programming approach this reluctantly, they learn a little bit about the idea of data types, and it can sound like a deeply, technical, unfamiliar subject, but it really isn't. You see, data types have become a lot easier when you realize you already do this. What we're about to do in programming you already do in everyday life. Now if you're looking at me with a skeptical, raised eyebrow right now, let me explain. Forget about computers for a moment and consider a few pieces of information that could have been asked about someone a hundred or even two hundred years ago. What's your birthdate? What's the name of your street? How many other people live in your house, what's your age, your marital status, are you currently on active duty in the military? You might even ask them their bank balance. These were all important pieces of information long before computers, and they're still important now. And you might not consciously think about it very much, but you instinctively know there are different characteristics to each piece of data. Several things here are numbers, but even those number are not the same, and we have different expectations of them. Take Age. You could ask the age of a thousand people and you would expect and only be interested in a whole number. Unless you're talking to a child who tells you they're 3 and a half years old, you want Age to be age in years. You know this should be a positive number; -24 is not an age. And you expect this number to be in a certain pretty small range; 21 is okay, 76 is okay, 382 is not okay, something's wrong with that answer. So you already do this, you have these mental boundaries and constraints for the values you would consider acceptable for this type of information. Whereas something different, a bank balance, is also a number, but it has very different qualities, it's hopefully positive, but unlike Age, it could be a negative number. And it's not a whole number, it's dollars and cents, or pounds and pence, or rupees and piase. Like Age, we also expect a certain range of values, but that range is much wider. Could this value be in the thousands? Of course. Could it be in the millions? Well, that would be nice. Could it be in the hundreds of billions? Well, probably not. But our unspoken expectations go beyond even that. We know that unlike an age, a bank balance is a more volatile number, it could change many times a day. And we know that with different types of data we do different things with it. With a bank balance we expect that we'll add to it or subtract from it. But you would not expect to do those tasks with a piece of data like your street name. First, this is going to be text. Even if it included a number like 4th street, we don't need to treat it like a number, you're not going to subtract from Wilshire Boulevard, or multiply 5th Avenue by Kensington High Street. And we know there's a lot of variety in street names, but we still have boundaries, there's a certain expected length. You might have several words in a street name, but you wouldn't expect it to be two pages long. And for a very controlled piece of information consider a question like, are you currently on active duty in the military? You expect the answer here to be just yes or no, that's it, not a number, not the name of a vegetable, just yes or no. You see, you already understand that the information in your life has boundaries and constraints, that when dealing with names or with numbers, in the real world, you don't think about them the same way. You understand there are ranges you find acceptable in one circumstance, unacceptable in others. Some numbers are positive, some are negative, some are large, some are small, some of them are volatile, they change a lot, and other numbers don't change at all, your date of birth is whatever it is. It might be different from somebody else's, but yours doesn't change every day. The value of Pi doesn't change, so we already understand how to think this way. We now need to take these ideas and see how they work in source code.

Applying Data Types

When writing source code different programing languages require varying levels of detail when defining a data type. Some allow, and some even require you to be very specific, that when you declare a variable you can say, this variable is a number, but more than that, this variable is going to be an integer, a whole number. And not only that, but it can never be a negative. And not only that, but it has a maximum value of 65, 535. Other languages are more flexible, they do not require, and some don't even allow you to be that specific, that exact about each type of data. But there are downsides to flexibility, because if you can't enforce rules on your data up front, you may have to write more code later on to make sure that the data you have is actually correct. But even with differences like these, most languages treat data in a very similar fashion. So once again, let's cover the concepts that most language use most of the time. First, numbers. When dealing with number values it's common to make a distinction between whether you have an integer, a whole number with nothing after the decimal point, or whether you have a value that should allow a fractional part to it. If you know the piece of data you're describing is a whole number, like age, or the amount of pages in a book, or the number of full-time employees, or the speed limit on a stretch of road, or the position in the New York Times Best Seller list, or the floor in an office building, these are all authentic integers, they don't have anything after the decimal point, no book ever reached position 2. 7 in a best seller list. There is no road with a posted speed limit of 0. 00015. But if you need a number that does support a fractional part, whether that's a temperature of 72. 4 or measure a snail speed of 0. 029 miles per hour, then we need that type of data and an integer won't work. The most common keyword in programming for creating an integer data type is either integer or just int for short, and the most common keyword for the other kind of number is a float, meaning a floating point number. Now I could do an entire module, an entire course on the technical differences of how integers and floating point numbers are stored in memory, and what that might mean when your do complex calculations, I'm not going to do that here, it would make everybody's eyes glaze over, including mine. For this course, let me leave this at, yes, this subject of integers and floating point values does get deeper than just the question, do we or do we not have something after the decimal point? But for now, that is a good enough question, because you can get pretty far with just that level of understanding. But beyond whether you're dealing with integers or floating point numbers, some languages let you be really specific about whether you need each number to support both positive and negative values, we use different keywords if we do or if we don't, or if you just don't care about having negative values. And you may also find choices for different sizes of number. So if you're dealing with either very large or very small values you can define them that way. Enough about numbers, here's a far simpler data type. Most languages have a Boolean data type, sometimes just called a Bool, named after the British mathematician, George Boole. This is a value that can be either true or false, that's it, no other options. And it's very common to want values like this. A variable that can hold, is the user logged in right now, true or false? A variable that says, are we recording right now, true or false? A variable that says, is this person on active duty military service, true or false? Has our spaceship collided with an asteroid, true or false? And in most languages you will find a Boolean data type and also find that true and false are keywords. Okay, some context here. This kind of value, something that is just true of false, can sound almost simplistic, too basic, and it's kind of easy to think, oh, okay, I guess I might need that kind of value occasionally, so let me assure you that you will rarely write more than a few lines of source code without asking that really basic fundamental question of whether some situation is true or false. A

Boolean value is not a concept that we use once in a while, or just occasionally, it is something we do in programming all the time. We also expect a data type for textual content, characters and words, sentences, paragraphs. And many languages have a data type for a single character, just one letter at a time. But what's typically more desirable is a larger amount of text, what in programming we call a string. A string is a collection of characters all strung together in a sequence. Now they're very easy to create, but it's worth considering that a string is more complex for the computer than a number or a Boolean, simply because we don't really know how big it is. If you just declare a variable as a string, until you put some data in that variable the computer doesn't know if it's going to contain a word, or a paragraph, or an entire document. So until you've provided a value the computer doesn't know how much memory to set aside. And it's worth taking a look at these values. When we assign a value to a variable we just write that value by itself, and we can change the variable again and again. So these are my variables, they can change from moment to moment. And these are the values that I'm assigning to them. These are not variables they are the actual literal values themselves, and when you see these actual values written directly in your source code they are called literals. So the first one is an integer literal, the value 99, it's not a variable, we don't ask what this contains, it is literally just the value 99. Next we have a floating point literal. In some languages you would follow what's after the decimal point with an f to say this is float. And after that is a Boolean literal, just true or false. And in almost all languages, a string literal is written with double quotes to mark the beginning and the end of it. First, that's how we can see the difference between the words we want to be using as variable names in our code, and the words we want to use in our actual string data. And it's also how we can have spaces within the string, and because they're inside the double quotes we're not confusing the language. Now one thing to point out is notice that with the Boolean literal value of true here, there is no quotes around the word true, and there would be no quotes around the word false. I can certainly use the words true and false within a string, but here they're not strings, they're actual keywords in the language, and when they're used as Boolean values they don't need quotes. So each language provides certain basic, built-in data types, they're already in that language, they're ready for us to use. It is true not all languages have exactly the same options. Some languages like C++ offer a variety of types within each category, not just integers and floating point numbers, but large and small versions, versions with and without negative numbers. On the other end of the simplicity scale, JavaScript only has one data type for all numbers, they're just numbers. And they're all stored as floating point values. So even if you want a whole number, an integer, it's actually still stored as a floating point number with just 0 after the decimal point. Simpler, built-in data types in a language, whether it's integers, or Booleans, or floats, are sometimes called primitive data types because they are basic building blocks of the language, generic ways to work with simple, straightforward, single pieces of data, but we can take it deeper. What about data that has multiple pieces to it? Like a date, with a year, and a month, and a day? Or what about something like an address which could have numeric pieces and text pieces? And most languages we can do that, we can take these single pieces of data and combine them into what's called a composite or compound data type. But that is a topic for later.

Creating Constants

I've been focused on variables, but not all data is actually variable. You might want some data in your program which doesn't vary, it just doesn't change. For example, you may define a few strings to contain messages and expect to use them later in your source code, but you don't expect to change their values. Or if you create a floating point variable to hold the value of Pi, or perhaps the maximum players that your game supports, you're never going to want to change these, there's no reason to change Pi to 4, or 9, or -700. So it's very common to need a piece of data in your program which just doesn't need to change. Now it's completely true that we could just create a variable, set its value, and then just not change it, but many languages support the idea of a constant. A constant is very, very similar, almost identical to a variable. It has a name, it has a value, and it has a specific type. The one difference is, when our program runs we can only assign a value to it once and then that piece of data stays fixed, locked, and constant at that value. Any attempt to change it beyond that first assignment will cause an error. There are a couple of benefits to this, one is if you can lock down a piece of data it prevents either accidental changes or any misunderstandings about what that data means, particularly from multiple developers all working and writing code on the same project. Another benefit is that in some languages, data stored as constant is actually more efficient than the same data stored as variables, because the language can optimize the memory differently if it knows that data is never going to change. So how to do this. A quick example or two of syntax. In some languages you use an additional keyword and in some languages you use a different keyword, here's what I mean. In the language C#, to declare a string variable, I'd write it this way. String message = and then give it a string literal to set the value. To make this a constant I add an additional keyword, const, at the beginning. That's it. This is now a fixed value and this cannot be changed to a different value somewhere else in my source code. In another language, Swift, we use a different keyword. The normal way to create a variable is using var, but if I want a constant I just swap out that keyword with a new one, let, and everything else stays the same, that's it. And in some languages, like Python, you just don't, there is no formal way to make a constant, you just make a variable and try not to change it. And one final word on constants, is that in some languages it is common to see the name of constants written in all uppercase, as opposed to Camel Case, or lowercase. Now this is another naming style, it's not a rule, but it can make things easier when you're reading a lot of code to visually distinguish between what have been defined as variables and what have been defined as constants.

Exploring Language Differences

There's one difference I want to talk about in how programming languages approach the idea of data types. Now make no mistake, they all consider it important, but there's a difference in where and when they must focus on this question of what type of data do I have right now? In many languages, when you declare a variable you're both naming it and giving it a type, even before you've provided a value for this you have attached type information, in this case integer to the container. Whatever value goes in here it must be an integer, but it's one specific type. Now if a language enforces this idea, if it makes you write code that says that every variable must be of a specific type, this is often referred to as a statically typed or type safe language. But, a few languages have a different perspective from this. In Python and JavaScript, for example, when you declare a variable you're doing it without providing the type information, it's just a container, you can put anything in this. And then, at some point, you will give it a value. Now that value might be an integer, or the value might be a string, or it might be a Boolean, but here the data type is belonging to the value, it doesn't belong to the variable container. So in those languages you can do code like I've got here, you can declare a variable, then set it to an integer, then change it to a string, then change it to a Boolean, or a float. This would not be allowed in a language like Swift or C++, you wouldn't even be able to compile your code. When a language favors this approach it's usually referred to as a dynamically-typed language, meaning you don't actually know for sure what the type of a variable is until you look at the value inside it. And it could be different the next time you run the program. There are benefits and downsides to each approach, and it's something that can almost approach a religious war between programmers as to which approach is better. But it's not a war we need to have. So in the next module we're going to explore an area where languages really are significantly alike, but we must manage the flow of our program, we control how our code runs and the choices that we're going to make inside of it.

Managing Program Flow

Introduction

We've explored writing a few statements, and I've already talked about the order of those statements, of our instructions being important. But when you write more code, dozens or hundreds, or thousands of instructions, you don't want to always execute all of your statements exactly the same way, in exactly the same order, every single time, you'll want some choice in control, and you can have that, but to understand how you control your code, it's easiest to first understand what happens if you don't even try. How does code run? So let's get clear, the typical way your code executes can be best explained with a quote from Alice in Wonderland. Begin at the beginning, the Kind said, very gravely, and go on till you come to the end: then stop. This is a pretty good description of the default way you code will execute, that unless you say otherwise, when it comes time to run your program it will begin with your fist statement, execute that and finish it completely, move onto the next, finish that, then move onto the next, one after the other. Now this might be a good way to read a story, but it's not good enough for us. Only in the most trivial programs will you be able to just begin at the beginning, go right onto the end, and then stop. We must control how we move through our source code. While code will still run in general from top to bottom, we need to ask questions, because every time the program runs things can be a little different. So we might want to know at one point, is today's actual sales amount greater than today's projected sales amount? If it is, we'll make that number display in green, in if it's not, make it show in red. If the user just entered their password, does it actually match what we have on record for them? If it does, we'll execute the code to let them access the protected area, if not we'll show a message telling them to try again. And we often write a lot of statements that we actually hope will never be executed. We write code to deal with incorrect data, we write code to deal with errors, like wrong passwords, no internet connection, not enough memory, missing files. So we need to make sure that this code is only executed when something's gone wrong. And while there's often code we don't really want to execute, we can also have the opposite where we want to take a few statements we have and run those same statements a hundred times or a thousand times. If I'm writing a program to calculate payroll for a thousand people, I don't want to copy and paste my instructions a thousand times, I just want to repeat a section of code, whether that's to calculate payroll or animate a bunch of enemy spaceships across the screen. And we'll need to find ways to group our code together, to make it easier to say, do these statements, but don't do these statements, repeat those statements, but don't repeat these statements, and all of that is the focus of this module.

Making Choices and Conditions

If you could take every programming language ever created, all their ideas, their keywords, their operators, then start throwing away everything that made them different from each other, only focusing on what remained that was the same, to see if there's a word we'd use again, and again, and again in every language, it would all come down to this, if. An incredibly common keyword across programming languages, a way to ask a question within our code, and change our behavior, change what we do based on the answer to that question. A simple example, and I'm just using pseudocode here, this is no specific language, just generic C-style language using curly braces because I have to use something. Three parts to this. There's the keyword if, then there's the condition, the actual question we're asking, if what? Well in this case if balance is greater than 1000. And after that is a way to say what we do, to provide one or more statements we will run if this is true that we wouldn't run if this were false. Whatever I write as a condition must come down to that basic fundamental answer. It's either true or false. Here if I ask if balance is greater than 1000, I don't care by how much, it doesn't matter if it's 1 over or a hundred million over, it's either true of false. It's that Boolean value idea again. I'm writing my condition here and it's very common to see the condition written within parentheses. it's not strictly necessary, not all languages require it, so I'm leaving them out just for clarity. I'm just using the greater than sign here as the operator. Most languages have a variety of operators you can use in your condition, like the greater than or less than signs, a few languages use other words for this, but the idea is the same. And importantly we'll often need to check if something is equal to something else, and when we do that it's easy to make one of the most common rookie programmer mistakes. If I want to ask if an integer is equal to some specific value, like 0, or a name is equal to some specific string, well these look okay, they might look understandable and quite readable, but remember, in most programming languages the equal sign already does something, and it's not a question, it is a command. The single equal sign is the assignment operator, to set a value. If we're actually asking the question if something is equal to something else, it is typical to use the double equal sign, two equals side by side, no space between them, that's how you check for equality. And the flip side of this is to check if something is not equal to something else, where we most often use the exclamation mark equal sign. So if these are potential things we can put in a condition, after the condition we must control exactly what we're going to do. How do we say execute these statements only when this is true? We need to group our statements into a code block. A code block is not a complex idea, it simply means a way to mock a few statements as belonging together, a block of code. How do we say, do these two statements, or these three statements, or those seven statements? The most common way of showing a code block in programming, even in pseudocode is using a pair of curly braces to show where it starts and where it ends. Quick sidebar, symbols are important in programming, so just to be super clear about this, these are curly braces, these are square brackets, and these are parentheses, all of them are found in programming, they're found a lot, and used for different reasons, they're not interchangeable, but they are similar in the idea that they're used to mark where something begins and where it ends. What that means is you find them in pairs. I don't care what the language is, if you have an opening curly brace it needs a partner, a closing curly brace. You might find the different ones several lines later, and they can even be nested inside each other, but they have to pair up. So back to this, the opening curly brace marks the start of the block, the closing curly brace marks the end of the block. So if we reach this code, this statement, and this condition, balance greater than 1000, evaluates as true, then we would jump into the code block, execute any statements inside it, and then continue on. We will hit the end of the block, and after that we'll jump outside it and continue on from whatever the next statement is outside of that closing curly brace. But if the condition was not true we would bypass that code block entirely and just continue on from that next statement after the closing brace. And as I'm talking about these curly braces, let me take a quick moment and talk about two different styles of how you will see these written. We will always have an opening and a closing one, that doesn't change, but it's slightly more common to see the opening brace written on the same line as the keyword it's being used for. What I mean by that is here I have the if statement and the code block opens on the same line as the word if. But you will also see the opening brace written on the next line all by itself. And this makes no difference to the code, it is insignificant whitespace, it all works the same way, but different languages tend to standardize on doing it one particular way,. The benefit of having the braces each on their own separate line is it's easier to see where they line up, with the downside that your code then requires more lines to view it all. Of course these aren't the only ways, an alternative way you can find code block would be in the style of a language like BASIC or Visual Basic. Instead of the curly braces this uses words, typically the word end to mark the end of some code block. So we're opening with the words If, the code block begins at Then, and it closes at End If. Now both these ways tend to indent the code inside. Indentation is not required, but this does make it more readable. And as I demonstrated a little bit earlier in the course, there are a couple of languages that just use indentation to denote a block of code. Python is the most well-known. There's no curly braces here and there's no End If. This is considered the code block because it's indented, and once statements stop being indented this code block is finished. I'm going to mainly use curly braces from now on to show code blocks, even when I'm writing pseudocode, because even if you'd end up mainly programming in a language that doesn't use them, this format is so common in programming, you should be familiar reading it. Moving on. If we have this if statement, if it's true we'll do one thing. often we also want to ask if it's false, we'll do something completely different. So we have a matching keyword for the if that comes right after the closing curly brace, and that's else. Now we have two code blocks, one for the true result and the else part has its own code block for the false result. When we hit this if we will evaluate the condition, either balance is greater than 1000 or it is not, then we'll jump into one or the other blocks, never both, and then after that we will continue on outside the full if else statement. The else is optional, there doesn't have to be an else for every single if. Having an if without an else is perfectly okay, it's a way to do something extra when a situation is true. You don't always have to do something else when a situation is false, sometimes you just move on. But that is the classic if/else conditional code. And while you don't need an else for ever if, sometimes it goes the other way, and one if/else statement is not enough, you have a deeper question, you need to create more complex conditions, you need more choices, and that's next.

Creating Complex Conditions

We can have more complex conditions, but there are a couple of different ways this can happen. First is the condition itself, the actual question we're asking more complex, so we have to ask about two or more things to get down to that single true or false result. Example one, in just pseudocode here. We're writing some code to calculate interest on a bank balance, but we only want to do this if the account balance is greater than 1000, and also the accountType must be a Savings account, and only if both of these things are true do we want to do this. Example two, we're figuring out if an order on our website qualifies for free shipping, and we want to say if either the cartTotal is greater than 100 or the shopper is a premium member they get free shipping. The first example is an and situation and the second is an or situation. In both of these the conditions are each made of two pieces, they're each made of two smaller conditions, and the computer is first going to look at each piece separately, as a single condition, figuring out each part to true or false. In the first example, both of the smaller conditions must evaluate to true, both balance greater than 1000 and accountType equal to Savings in order for the larger condition to be considered true. But in the second example either of the smaller conditions could be considered true for the larger condition to be considered true. Now in the second example it is possible they are both true, it's a premier member making a large order, but we don't care, it doesn't matter, either one by itself would have worked. So some common syntax. With complex conditions, even if you don't have to, it's often useful to surround each inner condition piece in its own set of parentheses so that it's very clear what you're looking at. And some languages may require the entire condition to be inside one set of parentheses. A few languages actually use the words and and or as the keywords between each smaller conditions, but in many languages it is two ampersands writer together to mean and, and two of the pipe, or vertical bar symbols, written to represent or. Having said that, you're not limited to two, you can take this further and you could have multiple smaller conditions separated by ands and ors. But there is another type of situation where we want more than just two possibilities, one single if block with one single else block is not good enough. Now first, it is common to add additional ifs as part of the same situation, even one inside another to go deeper, to ask multiple levels of question. To really grasp this idea, forget about syntax for a moment and think about this in terms of shopping. You see, not all situations are as black and white as, we go to the store and if they have full-fat milk we'll buy that, otherwise we'll buy skimmed milk. Life gets a little deeper. So as pseudocode for this I'm going to add a new if/else inside the original else block. So now I can asks a deeper question, if they don't have full fat then do they have 2%, and if they don't have that then we'll buy skimmed milk. And this is way I can easily deal with three options, three eventualities, rather than two. And it can get even deeper than this. These are nested if statements, they are self-contained if or if/else statements written inside the block of another if/else statement. Technically you could keep on nesting ifs inside other ifs and just keep going, 5, 10, 20 levels deep. Most programming style guides caution against going more than two, perhaps three levels, because they do become difficult to read and difficult to be confident that every eventuality will be reached correctly. There's a couple of things that plain if statements are not great at. One is when you're working with a range of values in your data. An example, a temperature that might fall into different range of acceptable and unacceptable values. You could check this with nested ifs, but it's kind of annoying to read or write. Alternatively you could just write one separate if statement after another. But one thing to be careful about with multiple disconnected if statements is that it's really easy to miss things, to make errors in your logic. In this situation, for example, if the temperature were exactly 90, none of these messages would be output because there's one option to deal with, less than 90 and one to deal with more than 90, but not 90 itself. And if the temperature were exactly 110, 2 of these messages would be output. And the issue is the language isn't going to alert about this, it'll assume that's what you want. So, because checking the same piece of data for multiple possible values gets tedious with if statements, so in many languages there's a better way, what's usually called a switch statement, though it can be called a case or select statement. It's designed for exactly this kind of situation, that we're looking at one thing, one piece of data, but describing multiple possible options for it. So here's an example, this is all one switch statement that spans multiple lines. At top we say what we're looking at, we switch on one particular thing, in this case the variable bathTemperature. And then within the block we have multiple different cases, different values to check against, with different code we can execute in each case, and usually offering a catchall option at the bottom to handle the case where none of the specific values were matched. Now switch statements are implemented a little differently across languages, some allow you to define ranges to check for, others make you list every separate value you're looking for. So I won't get deeper into them here, but it's good to know this is a much more readable format for dealing with this kind of situation. Okay, we've talked about ifs and elses, and ands, ors, nested ifs, and switches, but these conditional statements aren't only the way to affect our code. Next up, iteration, how to repeat, how to loop.

Creating Loops

A general principle in software development is if you find yourself writing exactly the same code more than one, you need to stop and figure out why. This is sometimes described with the letters D-R-Y, don't repeat yourself. And it does take some time to grasp all that this means because this does apply in several different places in programming. But generally there is no reason to have to think the same thought and then write the same code twice, or three times, or four times. You figure something out, you then write the code, and then if you need to have those instructions again you don't write them out again, you find a way to reuse what you've already written. One place where this don't repeat yourself applies is being able to repeat a section of code. In programming we also call this iteration, or more informally, just a loop, to take a few statements, put them in a code block, and then repeat that block multiple times. Multiple might mean just 5 or 10 times, it might mean 1000 or a 100000 times. It could mean, repeat as many times as you have employees in the company, or repeat for every document in that folder, or repeat for each track in that album, or every contact in your contact list. Or this might even mean, keep repeating this infinitely, or at least until something else in the code tells you to stop. Creating a loop is easy, creating a loop is easy. The thing you have to worry about is making sure that loop is ever going to stop. Here I'm going to cover two of the most common way to create a loop. First is what's called a while loop. We take the code we want to loop, put it inside a code block with the keyword while and a condition, similar to an if statement. And just like if, some languages expect that condition inside parentheses. And also, just like if statements your condition must result in a Boolean value, either true or false. So you expect to see things you'd expect to see inside if statements like, is myVariable less than 1000, is it true or is it false? Now when we run this program we will reach this while loop, we check the condition, if it's true we will jump inside the code block, into the loop, and execute all the lines inside it. And when we reach the end of the code block we won't continue on, but rather jump back to the condition, check it again. If it's still true we'll go back in and run the statements in the block again. And as long as that condition is true, we'll keep repeating. We'll repeat from now until the end of time. If we even want to stop, something needs to change, something needs to cause that condition to return false. Now if you accidentally write code that keeps looping and every programmer has done that, it's called an infinite loop, you may need to go into your task manager or activity monitor and force that program to quit because it will just keep going otherwise. So what we expect is something in this code will eventually change this condition when we check it to false. And when that happens we will jump right out of this loop, we'll jump to the end of the code block and just continue on executing any statements after this. So when you're looping it is very common to use a new variable to keep count of how many times you've been around the loop. Here's an example. Before the loop I'm going to create an integer variable counter and give it the value 0. Inside the loop I will print out the current value of that variable, and inside the code block at the end I'm going to add one to it. So that'll give us this output. We'll start at 0, print that out, add 1 to it, check it again, and print out the new value. And at some point we're going to increase the counter variable from 9 to 10. Now at the point when we jump back to the condition and check it, is counter less than 10, well no, it isn't, it's not less than 10, it is 10, so this is now false, we will jump out of the loop and continue on to print out that last line. So no matter what code you want to repeat, when you're iterating around a while loop it's very common to see three elements that are just there to keep track of the loop itself. As something to set up a correct starting point, so you're initializing some counter or index. the second piece is the condition, what do we keep checking every time to see if we keep going. But the third piece you expect is inside the loop, they'll be something to increment or add 1 to the counter so that at some point this will change enough to make the condition false. And in fact, there is another kind of loop you can write that will bring all these three pieces together, and this is called a for loop. So I'm going to write a for loop as written in many C-style languages. The general structure follows the same as if and while statements, it's a keyword for the parentheses for your condition and braces for the code block. But I need a lot more space inside these parentheses, because with a C-style for loop there are three pieces separated by semicolons, something to initialize, something that'll happen right at the start. Then there's the condition. And finally, what's sometimes called the afterthought, or at the end, something to run at the end of every time around the loop. This is the kind of thing you'd expect to see. The initializer, the first section is run once, the first time we ever hit this for loop. Then the condition is checked, we'll jump into the code block, we'll run all the code, and when we hit the end of the code block, that closing curly brace, we will run the third piece, typically to just increment our counter. Here I'm using the ++ symbol, which is called the increment operator, it's an easy way to add one to a numeric variable. So the output of this is identical to the while loop version, it's that a for loop has the format where you can see all the pieces managing your loop in the one place. Many languages also have what's called a for-in or a for-each loop. This is for when you don't want to keep track of a specific number of iterations like 5 or 5000, you want to repeat a section of code based on how many things you have. The typical format is you're going to through an item in a particular collection. So for each person in a contact list, for each file in a document's folder, for each track in an album. And one thing this of course assumes is that you do actually have these collections on hand in your program, pieces of data that contain other pieces of data, like an album that contains tracks or a folder that contains files. Now we haven't dived into these collections yet, but the idea here should still make sense, that with this kind of loop we don't have to manually track how many times we've been around it, we just trust the language to iterate around this as many times as we need. Iteration is one way we can reuse code that we've written, but it's not the only way, and in the next module I'll cover how we start to divide our programs up into smaller, manageable, and reusable pieces.

Making Things Modular

Introduction

This module is about breaking things apart, and as much as that, it's also about putting things together. Okay, this sounds almost meaningless, but it isn't. This is another thing we do all the item, it's a way we naturally think. When we group things together we're also separating them from something else. Think of a book, that by grouping pages into chapters you're also making those chapters separate from each other. If you organize your possessions, your clothes, or your books, you're both grouping some things together, and by nature separating them from something else. Here's my group of fiction books, separate from my group of non-fiction books. Not only that, but groups can contain other groups. Within non-fiction there's travel, here's languages, here's technical. So we group and we separate, and it makes it easier to understand what we have, to manage what we have, and to find what we want. So in code, what does this mean? If we have a lot of code and we just wrote it as one continuous list of statements, even if we used if statements and loops, reading this would be like reading a very technical book written as one massive bunch of sentences with no paragraphs or chapters, you could read it, but there'd be no way to understand the structure of it. So we don't do this. First we can separate our code into multiple files, that's one way to make it easier to manage. But we don't stop there. Within those files the code can be grouped and divided into different meaningful sections. Here's the code to connect to a database, here's the code that calculates interest, here's the code that handles the user interface, or the code that takes care of error messages. Now not only will we group and separate our code, but we will want ways to group and separate our data. To go from having those individual variables, like score, and name, or balance, to being able to say, here's the group data for all our employees, here's the data for sales for last year, which contains the grouped information for sales for each month, which itself contains the grouped information about sales for each week. So in this module we'll begin with functions, ways to take some code that belongs together and give that section of code a name so we can reuse it and reuse it. And I'll cover the benefits of functions including powerful techniques like recursion. I'll talk about grouping data, see what we can do with composite data types and collections, and finish with object orientation, another way to group and separate our code and our data using a set of ideas common in most popular modern programming languages.

Creating Functions

We will break our code into modular-named pieces because it makes it easier to understand what we have, to manage it and find what we're looking for. It makes it easier to maintain, to change it, more flexible if we have different ideas and new plans. And, the ability to do this is the same thing that's going to save us from having reinvent the wheel every day. This will be what lets us tap into and build on top of all the things that other programmers have written, whether that was yesterday or 20 years ago. So when we're writing code we'll often write a section of code that's several lines, maybe even just four or five statements. And we figure out that these lines kind of belong together. Perhaps it's a few lines of code that plays a sound effect. Or a few lines of code that checks the internet connection, or shows the scoreboard in a game. So we decide to take this code and give it a name, we're going to make it a function, to create this self-contained chunk of code, like having a mini program within out program. We'll then be able to use it and more importantly reuse it from other places in our code. A quick sidebar. There are some other terms you'll hear for this same idea, sometimes it's called a function and sometimes it's called a module, or a subroutine, or a subprogram, or a process, or a method, but, function is the most generic and the most widely-understood term for this idea. End sidebar. To create one we just need to begin with three things, what's it called, where it starts, and where it ends. In many languages we'll have a keyword like function, or func, and then our name for this new function, what do we want to call this new chunk of code? Naming a function I like naming a variable. It's up to us, but we want something meaningful. And just like naming a variable, it's very common to use the style Camel Case when naming a function. But even though we're using Camel Case a function name is going to feel very different from a variable name. Because when you name a variable you're naming a piece of data, score, or file name, or phone number, but with a function you're naming a task. This is a block of code that does something. So it's very common to name a function with a verb/noun format like validatePassword, or playSoundEffect, showScoreboard, explodeSpaceship, createEmail, encryptFile, connectToDatabase, or printMessage. There is no rule for this naming style, it's just hope we can read a function name, even if we didn't write it, and just from that name get a pretty good idea of what this function does. And then it's just a code block to say where this starts or ends. As with other code blocks we've seen sometimes it might use words, and in other languages it uses the curly braces, but it's just a code block, the same as code blocks in conditionals or loops, to simply mark where does this function begin and where does it end? And once again the code inside is commonly indented. But we're missing a vital step, because the code now placed inside this function will never be executed unless some other part of my program says, go run that function. So when we talk about functions we have two tasks, how do you write one and how do you then run one? Any programmer will know what you mean if you said you wanted to run a function, but the term we usually use is that we call a function. We write a function and then we can call it. So here's how we might call it. Now this is still pseudocode, it's not real syntax, different languages do call functions in different ways, but it is very common to see a function call written as just the name of the function with a pair of parentheses after it. I'll talk more about this in a moment. And only when this line is executed will we then jump into the function itself, run through all the lines inside it, and then when we get to the end, jump back out of it to where we were after the call to the function and move on. Some context here, as a working programmer sometimes you'll write a function and somebody else on your team will be the one who calls it. Sometimes as a programmer someone else will write that function and you'll be the person who needs to know the name of it so you can call it. And very often, you'll find useful functions that were written years and years ago, and you find them and think, great somebody else already did this task so I don't have to, I can just call that function they wrote, and that's a good thing. We're back this idea of code reuse, don't repeat yourself, don't reinvent the wheel, unless you have an idea for a better wheel. But sometimes you need your functions to be more than just a few lines of code with a name on it. You need to be able to pass information into that function and get information out of it.

Returning Values and Using Parameters

The next step is that often we want a function to do something and then return a value, return some kind of result. So here's an example, still just pseudocode. The idea here is that we want to get information like the name of the current user and today's date, and then create a new string variable by combining some of those values. Now the entire point of this code is to create a message. So if I wanted to turn this into a function, I need a way for this function to return this string, the result of doing all this. And the way to do it is that inside the function as the last line, you'll add a return statement to send back this message to whoever calls this function. When I call this function somewhere later in my code I could just use the name of it, but I do expect a result back. So instead, I'd write a line like this, string myResult = createMessage. So as ever we're using the assignment operator, the equal sign. So we'll first evaluate what's on the right-hand side, the function createMessage, we'll execute that, it'll return a result, and whatever is returned will be assigned to this value, myResult. So the other side of this coin of calling a function and then expecting a value back from it is instead to be able to pass new information into the function when you're calling it. here's an example. This function, displayAreaOfRectangle, is defined to accept two parameters, in this case, two integers, one called width, one called height. And it now expects these two pieces of information to be passed in the function every time it's called, and these can be different every time the function is called. And now, when I call this function I must provide that information. And I'm providing it inside the parentheses after the function name. I could provide it as integer literals, the direct values that I'm passing in here, or I could do integer variables. So inside this function those will be treated as the two variables that can be used, width and height, multiplied together and we get a result printed out. If I call the function with different integers I'd get a different result. So that's the reason why the common format to call a function does use those parentheses, it's how we say what we're passing into the function when we call it. And it's why when we're passing nothing into a function you typically just see a pair of empty parentheses, that's the syntax to be explicit that we're not passing any information into the function. However, the problem is, is if that function expects you to pass it two pieces of information and then you don't, expect an error. So these are the basics of functions that return values and functions that accept parameters. And you can have functions that do both and functions that do neither. I'm not going to get deeper into syntax here because there are so many slight differences between languages it would make for a long and tedious comparison. So if you have the basic idea here, that's good enough. What I want to do more is focus on the benefits of doing this, and next explore the idea of recursion.

Using Recursion

We can make a function and we can call a function. We can have code inside one function that calls a different function. So we can have functionA call functionB, and we can have functionB call functionC, and functionC call functions D, E, F, and G. But, we can also have a function that calls itself. That might not seem to make sense, it might even seem like a recipe for an infinite loop, but it's not that. What functions allow us to do is a very powerful programming technique called recursion. It's not something you need all the time, but there are certain problems in programming where recursion is the best and the easiest way to get your result. Let's go through an example, and I'm going to do this all in pseudocode, we're focusing here on the idea, we do not need specific syntax, this can be done in any languages. And we'll begin without recursion. Imagine I've been asked to write a function that takes one parameter, the name of a folder or a directory on a hard drive. And I need to look inside that folder and go through all the items, one by one, and write out the names of any mp3 files. So even in this Pseudocode I'm using a for/each loop, found in most languages, this is a loop that will iterate around however many items are in the folder. For each item I then ask, is it an mp3, and if it is we print out the name of that item. So that's it, and this is absolutely fine. If we call this function passing in the name of a folder, it'll take that parameter and then start to go through every item in that folder, whether that's 50 items or 5000 items, and write out the name of each mp3. Here's the problem, I am then asked, oh, but this folder might have a subfolder, it might, it might not. But if it does we want to go inside that subfolder too and list all the mp3's inside it. Now okay, I think that doesn't sound too bad, and here's one way I might be tempted to deal with this. We're already iterating around every item in the folder, and a subfolder is a kind of item, so instead of just asking, is every item an mp3 I'll add code to ask if every item is a folder. If it is I'm going to repeat the code I'm already using that will iterate around inside that subfolder. So what we're doing is taking the code that already worked and nesting it down inside another level. And this would actually work, it works as long as there are only two possible levels in the folder. But here's the problem with folders on a hard drive. Reality gets in the way, this is completely unpredictable, we don't know how far the nesting goes. Maybe it's 2 levels deep, maybe it's 10 or 20 levels deep, and I would have to write 20 levels of nested code one inside the other, hoping that I've added enough to deal with all the possible options, that doesn't work for me. So back to the version that works on one level. hers's how I change it to use recursion. When we go through each item in the folder I'll add code to check if any item is itself another folder. And if it is, this function will call itself. So we begin with a call to that normal top-level folder, we'd go into that folder, and if we had a more complex setup, a more complex bunch of directories, we'll hit the for/each loop, we'll go through each item, and whenever we find a subfolder we'll make that recursive function call, we will call ourselves, but this time passing in the name of the subfolder, so this function just begins again working its way through that subfolder, and this is recursion. Now this code will work with 1 level, it'll work with 2 levels, it'll work with 50 levels of incredibly convoluted folder structure, it will just work. It'll find every mp3 in every possible combination. A folder structure is a classic example of where recursion is useful, but it's useful for anything where you have unpredictable amounts of items and unpredictable depth, like navigating a company organization chart, or exploring the tags in a web page, finding all the moves for a chess piece of a chess board, there are many problems where recursion is useful. As much as anything, I wanted to cover it here to expose you to this idea that in programming you will find these techniques that might sound intimidating, that take a few minutes to wrap your head around, but they lead to a solution that's easier to write and requires much less code than any of the alternative ways of fixing this problem.

Creating and Using Composite Data Types

You can't describe everything you ever need in your program using single pieces of data like one integer, one floating point number, one Boolean, or even one string. So, most languages support and encourage ways to group multiple pieces of data together. A general term for this is a composite, also known as a compound data type, they mean the same thing, data that's made from smaller pieces of data. An example. Imagine someone starts to write a program to keep track of their vintage video game collection. So it begins with one game using four independent variables. A couple of strings for title and publisher, an integer for year of release, and a Boolean for completed. Now if I want another game, I guess I could repeat all those variables, but it's going to get annoying to keep copying and pasting the same lines, it's easy to make a mistake when doing it. And right now there is no actual structure here, there's no way to keep these together, they're just a bunch of independent variables that happen to be written on lines next to each other. So what most languages support is a way to define your own data types to impose a structure on these that'll keep the smaller pieces of data together for each one. For this the first thing is to define this new data type. There are a few different keywords, but a very common one is struct, We're making a structure. Inside this, inside this code block, I'm defining four variables. Now writing this code did not create any of this, I'm defining a new type. My new type is called Game, and it's a type like at string, or an integer, or a Boolean as a type. But once it's defined, what I can then do is make a new variable of this type. So create a variable called firstGame and of type Game, it isn't a string, or an integer, or a Boolean, it's a game. This game is a composite data type, a piece of data that contains other pieces of data, and we need to drill down into this new variable to get to the inner data, what are often called the member variables, and the typical way is using something called dot syntax. We use the name of the variable, firstGame, dot, and then the name of title, and publisher, and yearReleased, and completed, that have been defined as being part of this data type. And then we have this self-contained firstGame variable that contains four pieces of data inside it. I can easily make a new one just creating another variable of the type Game. This may seem like more work, but there are actually a lot of benefits to imposing a structure on and around you data. One is that as we start writing more complex code we will be passing information from part of our program to another, passing things from function to function. And it's lot easier to pass a single Game variable than to separately have to pass all the individual elements inside of it. Beyond that, in most languages there are ways not to just define a type that holds a few pieces of data, but also to add extra behavior to it, to add custom code to it. I'm getting ahead of myself, that's coming up shortly. First, there is another way to group data that we need to cover. So we don't always want to group data together like this with different pieces, with different data types. Sometimes, all we want to do is take a single piece of data and repeat it.

Arrays and Collections

Computer programs have always been great at dealing with multiple items of the same kind of thing. We wouldn't write a program to generate payroll for one employee, we want to do it for a thousand. We don't have just one song in our media library, we have hundreds of them. We don't have 1 enemy spaceship on the screen, we have 20 of them. Repetition is everywhere. Now, we've already talked about writing code that can repeat, writing loops, but what we're focused on here is defining data that repeats. Okay, we are quickly covering a lot of ways to group data, so just in case you're a little hazy on the difference between this idea and the composite data type from the previous clip, a composite data type is how I would say something like, in my program I want an album to be made of a title, and an artist, and a year, and a genre. but this is how I would then say, now I want a thousand albums, or a thousand strings, or a thousand integers, or 12 months, or 31 days. Now sure, there's nothing that would actually stop you from manually typing out a thousand separately-named integer variables in your source code, except perhaps pain in your hands, but importantly, when you're writing source code you often won't know exactly how many of something you're going to have when your program runs. Now a media player program doesn't know in advance how many albums will be on someone's device, a payroll program still needs to work if the number of employees changes from week to week. So we need a way in our code to say, I have many of something, even if I'm not completely sure what many will always mean. The most basic, most fundamental, and still incredibly useful way to do this is called the array. Instead of creating a single integer variable, we can create an array of integers, a collection of them, several independent values inside one container. Now, as ever, syntax is not the focus of this course, but I wanted to demonstrate that while the syntax to create an array is different across languages, one thing that's very common is to see square brackets being used somewhere in the syntax. Here these are all doing the same thing. They create an array of five integers, and provides the values 1, 3, 5, 7, 9 for those 5 numbers. Now of course, this could be an array of ints, it could be an array of strings, or an array of Booleans, or whatever you want an array of. And you could have 5 of them, or 10 of them, or 10000. Each array you make has a name, but each individual element inside that array does not have a name. Let me say that again. The array, the container that holds the values has a name, but the individual elements inside it do not, because we don't want to have name everything. But, we still need to be able to get to each item, whether to look at it or change it. And we can do that because an array is always an ordered collection of items, and what that means is, each item, or what we usually call an element inside the array, it won't have a name, but it will have an index, it will have a number. And that number isn't up to us. The index of each array is always in order, it's a strict sequence, there's nothing random about this. If I have an array of five integers the index of this will be 0, 1, 2, 3, 4. And we can get, or set, any particular element in an array by knowing the name of the array and providing a number, the index. Once again, it's most common to see the square brackets being used to get to the element at a specific index in the array. As in this, I'm retrieving that, just printing out the value 5 that's in that middle element, but I could also use that to change it, reach into position 4, which is the last element. Let's cover a few things that are usually true about a classic standard array. First, a 0-based index. Typically an array index starts at 0, not at 1, and it goes up 1 by 1. So with an array of five elements the last element is index 4. There are a few languages like COBOL and Smalltalk where arrays start with index 1, but its far more common to assume the index always starts at 0. Next, it's common that any standard array you make is limited to holding one specific data type, that you make an array of integers, or an array of strings, or an array of Booleans, you can make as many as you like, but you can't mix the data inside them. A few languages are more flexible and allow different types of data at different positions. And if you define your own composite data type, like the game data type from the previous clip, you could then make an array of games with each element in the array itself containing multiple pieces of data. And also, in many languages when a basic array is created it's created at a specific size. Now, it can be any size, 5, or 50, 5000, but once it's created it can't have elements added ore removed from it. The term you sometimes hear for this is that that array is immutable, which means unchangeable. But bear in mind, you can create an array using a variable as the size of it, and the variable can change based on previous logic in your program, so you don't have to know the exact size of every array when you're actually writing your source code, you just have to know that each time the program runs and the array gets created it will be created at a specific size. Arrays are the classic way to hold and organize multiple data items in your program, but they aren't the only way, and as programming languages evolved over the years, this idea has been extended, and many languages now offer additional methods to do this, including dynamic arrays that can have new elements added and removed as necessary. You can have arrays that can hold any type of data and aren't limited to just integers or just strings. And this is actually the default in several languages including Python, Ruby, and JavaScript. We have associative arrays, an array that uses a different kind of index, not 0-based, but where you have your own choice of what you want to use as the index for every single element. As they get more advanced we stop calling them arrays and we have other specialized names like dictionaries, and queues, and lists, and stacks, and hash tables. While they're each good at different things, they are similar in that they're all designed to hold multiple items. And because of that, the general terms for these in programming is that we call them all collections. But getting deeper into collections is a topic for another course, particularly because different programming languages have very different options for this. So one last thing to cover in this module, next, we are going to explore the ideas that are shared between many of the most popular programming languages, hence the ideas of object orientation.

Introducing Object-oriented Programming

It's not unusual that when you first approach programming you will encounter this phrase object-oriented programming, or perhaps just object orientation. And you try and figure out what this means, but just reading about it just makes things worse because then you're surrounded by a bunch of jargon, terms like polymorphism and encapsulation, phrases like composition over inheritance or dependency inversion, and it is challenging to understand at first. And that's because object orientation is an idea, or rather, a set of ideas, it's a bunch of different principles that have been adopted and implemented in slightly different ways by many, many programming languages. Out of all these languages the only one that doesn't support at least some object-oriented features is C, and a lot of these languages are highly object oriented. So it's important, because most popular languages have some object-oriented features and some of the have a lot. And that means the language expects you to understand those ideas and follow those principles. And if you don't, writing code is going to be like pushing against the tide. Now I could do an entire course on object orientation, in fact, I have done, and I will do again, but that's not my focus here. So in the next few minutes I want to cover a few of the most important ideas, and a few of the words you'll encounter again and again. But before we get into any of the jargon, here's the larger, more fundamental idea behind all of this. If we just sit down and write code, if we go by the seat of our pants, we'll write variables, we'll write functions, we'll use loops, we can use all of the things we've covered so far. So imagine I'm writing the ecommerce section of a website, dealing with members and their purchases, and I'll have a bunch of different variables, information about orders, data about customers, data about discounts, and a bunch of different functions.

Everything from signing up for a newsletter, to processing a payment, to changing the email address of a member. And even if I were trying to organize a code as I wrote it, I might do something like just put all my variables, all my data in one place, and all my functions in another. I might even put them in separate files. But with object orientation what would be important is the way that we separate and group things, and just putting your data and your logic in two massive areas like this is the wrong approach. Instead, we should think about the objects in our program, so what are objects? Well first, think of the things I'd naturally say as a human being when talking about what this program needs to do. What are the nouns I would use? If I'm talking about ecommerce I might say a customer can begin a new order by adding an item to the shopping cart. First I'll look at the nouns in this, customer, order, item, and shopping cart, and those might be good potential objects I might use in an object-oriented version of this program. In a media player program the objects might be things like albums, tracks, and playlists. In a game the objects could be spaceships, enemies, asteroids, and missiles. So it's these objects that should drive the way that we group and separate our program. And the important thing is, objects are not just data, they're not just a few variables gathered together, objects are both data and functionality. They have a behavior, they have code that actually performed tasks on that data. So in a game I might want a spaceship object that will hold data, like its position, and its shield level, and its name, and its color, but it would also hold its own functionality, like firing a missile, flying, exploding, and everything that the spaceship needs to do. So each object becomes almost a self-contained miniature program, able to manage both its own data and its own behavior. Now there are a few words used a lot across object-oriented programming languages, and here's the first two, class and object. I'm explaining both at the time because these words, these ideas, are two sides of the same coin, you can't have one without the other. In object-oriented programming, a class is how we define what an object should be, how we describe it, whereas an object isn't a description of anything, it is the object itself. To make two analogies, a class is like a blueprint for a house and the object is the house actually made from that blueprint. Or alternatively, a class is like a recipe, it is the detailed description of the steps and ingredients. But you can't eat a recipe, so the object is the actual dish created when that recipe is followed. There are two things to take from these analogies. One is that each class can be used to make multiple objects, like one blueprint could be used to make multiple houses, or a recipe could be followed again and again. And the other idea is that the class comes first. You define a class and then you can make objects from that class. So when writing code, the class is the code we'll actually write to define what it means to be a customer, or a spaceship, or a person object. You define a customer class and then you can make a thousand customer objects from that one customer class, or a spaceship class can make a dozen spaceships from that class. Now you don't have to make multiple objects every time, but you can. And in most object-oriented programming languages there is a keyword, class, that you use when defining one. Here I'm creating a class, I'm calling it spaceship, the name is up to me, and inside the class, inside the code block I'll define both the data and the behavior that I want for objects of this class. This could be something as simple as just writing a few variables and a function or two. We've talked a lot about variables and functions in earlier clips, but in object orientation we often use a different name for variables and functions defined inside a class to differentiate them from conventional variables and functions. So we say that the pieces of data are the properties of this class, and any behaviors we have are the methods of the class. But like a blueprint or a recipe there is no point defining a class unless you're then going to create an object from that class. So there's one more word. When we actually make an object from a particular class there's a term we use for this, instantiation. We are creating a new instance of this class. There's often a specific keyword used for instantiation, the most common one is new, like this. So after this spaceship class was defined, I can create a new object of that spaceship class type. I call this new object whatever I want, it's the usual naming rules, I can then access what's inside it, usually with dot syntax to get to its data or its properties, and I can also call its methods, its behavior. And if I wanted to, I could then create another new instance of this class, another object, and each object is completely independent from the other objects. They all contain their own copies of their own data, their own behavior. Okay, and if the last few minutes was your very first introduction to these idea I would not expect things to seem clear yet. Bear in mind, these are principles and concepts that have been evolving for 35, 40+ years from some of the nerdiest people on the planet. It is okay if it takes some time to grasp this. But if you can get comfortable with one of the basic ideas that once a class has been defined, and you can create or instantiate multiple objects based on that class, and I can tell you something that will make this easier, you see, most of the objects you will instantiate, you won't have to make the class for them because it'll already exist, it's already done. In most real-world programming environments there are a huge amount of classes already defined and just available for you to use. Often hundreds or even thousands of them. Everything from basic things like dates and times, to images or sound players, classes for graphics, classes for audio, classes for encryption, classes for making user interface like buttons, and sliders, and checkboxes, it's all there. And becoming productive as a programmer is often less about writing everything yourself and more about finding what's available and tapping into it. But we had to talk about this for that subject to make sense, and that's what we're going to explore in the final module.

Programming in the Real World

Introduction: Frameworks and Libraries

At the very beginning of the course I told you that programs were made of small, individual instructions, and that's totally true, but it's possible to get the wrong idea about what this means. Because everything changes as you begin to understand the small idea that makes a huge difference, that because we have this ability to take some code, wrap it up, and give it a name so we can reuse it, whether that's where the basic idea of a function or the most substantial idea of a class. Because we can do that what looks like one single instruction could actually be 1000. And that's what gets us from hello world to hello world of Warcraft. And what gets us from I can instantiate one object to I can make an iPhone app. Here's one way to think about this. Imagine you had to some work in programming languages that was so limited the only kind of output the language actually provided was the ability to draw a single dot on the screen. That you could write a statement that would turn on one pixel at one specific x and y position. So many pixels in and so many pixels down, but that was it. And this is certainly limiting, but if you think about it for a moment you might realize, well, if I can draw 1 dot I could draw 20 dots 1 after the other. So why don't I do that inside a loop and draw a line. Wrap that up and now call it my drawLine function and I can all that when I want to. And then you think about it a little more and you decide to make this a function that takes some parameters to you can provide a different start and end position, and then figure out how many dots to draw. Now this would take a little more thought, a little more calculation, but with a few hours at it you could end up with a drawLine function that could draw horizontal, vertical, and diagonal lines. So far so good, but the great thing is when your colleague says to you, oh hey, you figured that out, well great, I'm not even going to look at your drawLine function, but I'm going to use it, add a little bit of my own logic, and call it four times, draw four lines, and make a new function called drawSquare. And then you other colleague says, well I'm going to take both of your functions, add a little code to draw two squares and a few extra lines, and I'll make a function called drawCube. And someone else takes that drawCube function, adds a little bit more to it and puts it in a loop that draws it at a slightly different position every time. And what you end up with is that a new developer could walk in and find they could write one line of code that says drawSpinningCube, and get exactly that, just from this idea of developers incrementally building a little bit at a time on top of what's already been done. And this is just a tiny example of the power of this, and it's not theory, this is exactly what has happened. From the earliest days of programming some code was always worth sharing. that if a developer had needed to close themselves off in their office for a month thinking about nothing but date, and that they thought about daylight savings time, and thought about leap years, and thought about time zones, and they've written a whole bunch of code to validate dates and calculate how far one date is from another date. Well if all that work was good, there's no reason for anyone else to have to do this, ever. Now somebody else might have obsessed about the fastest way to take an array of integers and sort all the values in it. They've worked through these algorithms, they figured all the different steps, all the potential ways this could be done. Or perhaps, the best way to search through an array for a specific number and return where it is. And someone else might have built functions that could take a string and convert it to uppercase, or convert it to lowercase, or tell you if were a valid URL or a valid email address. And these kinds of generic functionality, and many more, all became very desirable. So this kind of code was all collected together, but what do we call it? Because this might be code, but it's not a program. We don't want to run all these things just by themselves, we want to be able to use them from our code. the most common term for this is a library, it's a collection of prewritten functionality, intended to be shared and used by other programs. A few languages use a term like module, but library is generally understood. And now, when we write out own programs we can use this library, we can have it all available. One distinction. Using a library does not just mean, hey, here's a file full of source code for you to copy and paste bits that you like. The point of a library is it's complete. Somebody else has though about this, they figured out the algorithms, the steps, the ways to accomplish all these tasks, they wrote the code, and that code's been battle tested and optimized often over many years. And if you're using a compiled programming language, the library will have already been compiled into machine code. You may not have access to the source code, even if you did want to look at it, which most of the time, you don't. So, when using a library, you often need to read some documentation to find out what's available in it, and what the functions are called, and how you would call them, but it's not about copying and pasting somebody else's source code. Instead, you just link to, or import that library from your program. Often it's a single line in your source code like import or include. And by adding that one line, everything in that library is made available to you in your source code. The first libraries were general ones, generic tasks you might want in any kind of program. They make it easier to work with dates and times, to work with text, they may contain useful mathematical functions and ways to make it easier to read and write files from the hard drive. They might include ways to help deal with errors, and they might include things like multithreading, the ability to do multiple things at the same time. Or include new collections, other ways to group data together. But some libraries are more specialized, they're libraries not everybody needs. Perhaps a library full of code to help you work with audio, or just 2D graphics, or just 3D graphics. A library to help connect to databases, or one that's full of ways to work with web pages in HTML, or a library to help you encrypt or decrypt things. The first kind, the general, generic library, is often called the standard library of a language. The term standard library is used for C++, and Ruby, and Python, and many others. And if there is a standard library it's just taken as a given that whatever program you ever write in that language you're using that library, you can basically consider it part of the language. But after that, any other library is optional, it's up to you. In some of these discussions you will also hear the word framework as a similar kind of idea, and some people use these words framework and library casually and interchangeably. Two differences. First, you usually see that word framework applied to a well-defined specific situation, like a framework for building web sites, like Ruby on Rails, or Angular JS, or a framework for making desktop applications, or a framework for creating a phone application. And what that means, and as the name might even suggest, is when we choose to use a framework we are choosing it because it provides a lot of predefined structure for making that kind of application. It's as if you wanted to build a house, and you could click one button and instantly have a foundation, and the plumbing, and the scaffolding all in place, it's great, it's really helpful, it can save us incredible amounts of time, and that's the point of using a framework. But, we must also understand that we are now volunteering to fit our code into that framework, we are handing over some control and we must now play by their rules for this to work. So that's the key difference, both frameworks and libraries are prewritten code, but when using a library we're in charge and we decide when to call it. When we're using a framework it's in charge and it decides when to call us. Okay, there are two more terms related to this that I want to cover, SDK and API.

What Is an SDK?

Once you start looking at developing on a particular platform, say for Android, or Windows 10, or iOS, you're likely to come across this word SDK. And you may already even know this stands for software development kit, but what does that mean and how does that differ from things like libraries and frameworks that we just covered. And why the word kit? Usually when you hear the word kit it's in a phrase like kit car, or electronics kit, or model airplane kit, kit meaning a bunch of stuff in a box. Well, actually that is what SDK means, or at least that's what it used to mean. Let me show you one. This is what an SDK meant in 1992. This is the Microsoft Windows C and C++ software development kit. And when you got an SDK in those days what you got was a big old box weighing about 30 pounds. Inside that box you would find things like a reference book for the C language, another reference book for the C++ language, yet another reference book for the class library, the standard library, you'd get a bunch of floppy disks including separate programs for writing code, compiling it, debugging it, you'd get some tutorials and some help books, and even a getting started manual to help you actually understand what was in the box. And that was a software development kit, a big box full of everything you needed, and hopefully everything you might ever need to develop software on that platform. And these were expensive things to make and ship. So they first moved to being mainly on CD Rom, with electronic versions of everything, and as the web became more of a thing, SDKs all became downloads. But these days the term SDK has become a little vague, and different platforms, some will tell you to download the main development application, and that will include the SDK, And others will tell you to download the SDK and that will include the main development application, it really doesn't matter, it's still kind of a catchall term for a bundle of different programs and tools that a developer might need during the entire process of writing, debugging, testing, developing, and releasing code. These days an SDK might include things like emulators to test a phone application using your desktop or laptop. And the SDK will include relevant frameworks and libraries that are available on that platform, they will be part of it.

What Is a API?

So what is an API? And by that I don't just mean what does it stand for? A lot of people can tell me that. API stands for application programming interface, yeah, okay, but what does that mean, really? Well, imagine for a moment you're a plumber, and it's your first day on staff at a new location, an office building, an oil refinery, an apartment complex, miles of pipeline, a massive system already in place and running and working. You didn't make this, somebody else did. Do you think your first task would be to take a big wrench to some random part of the plumbing and start unscrewing it? No, that's a good way to get fired or covered in consequences. But neither is your first task to have chart, and diagram, and learn every single bend and curve, start pulling out the drywall so you can memorize every joint and piece of solder behind that walls, that would be a pointless waste of your time. The system is running, it works, and there are parts of it that would work for decades without anybody's attention. What you do need to do is figure how you're supposed to interact with it, at what point is there some intentional place that you affect this? With plumbing it might be a shutoff valve of a pressure gauge. With electrical work a fuse box or a control panel, what we might call some kind of interface, a specific way to interact with the system, and it's the same in programming. Now true, the term interface happens a few times in software development. If a program has a user interface and not all of them do, that's what intentionally provided so that I as a human being can interact with the program. If a program has an API, an application programming interface, and again, not all of them do, that is what is provided so I can write code to interact with this program. But the thing is, I don't want to have to learn exactly how that other program works, that other system. I don't want to read its source code that would be the equivalent of pulling out all the drywall and having to learn every bend and curve of the plumbing, I just want to know the small, specific way that I am supposed to interact with it when I write my code. An example. Say I'm writing a program and I need to be able to compress an image, but I don't know how, so I do some searching and I find that there's a compression library I can import and use. That sounds good. Well compression is difficult, so any compression library I find is probably actually made of thousands and thousands of lines of code, internally it could be hundreds of methods and functions, dozens of different classes, but I don't care, I only want to know how I'm supposed to write code to interact with this, what is its API. So I'd look for some documentation. Perhaps I'd find I'm supposed to first instantiate a new object, I then tell it the name of my file, oh, and then I read that I need to set the output file. And then I read I can change the compression type, but if I don't that the default algorithm is fine for most things. And finally I read I'm supposed to call the zip method to go ahead and compress it. Then I read a little further and find that this returns a status, so I suppose I'd better check to see if this worked. And my hope is that I don't need my TODO anymore. I get the functionality that I need without having to get any deeper into the details. Now, as ever, this is of course pseudocode. there are compression libraries that are kind of like this, but it's the idea of the API and how involved I need to get with this code. It's that process of finding out how you're supposed to write code to interact with another system, whether that's a library, or a framework, or something like a website. Companies like Facebook and Google make multiple APIs available, so you can write programs that interact with them, whether that's posting to a Facebook timeline or using Google Maps. What you hope for is that the API well documented, it's up to date, it has some sample code. Sometimes you'll find ones that are a bit more primitive and you have to use a bit of trial and error. But the hope is that by doing this we once again save ourselves from having to reinvent the wheel. As ever, you can expect different options depending on which programming language you've chosen to use. So it's about time we finally talked about that choice.

Choosing a Programming Language

As you might imagine, I have often been asked the question, okay, Simon, so what programming languages should I learn? Sometimes this is phrased as what is the best programming language to learn? And for this let's talk about trains. Asking what's the best programming language is like asking what's the best train? It is a perfectly valid question, but imagine asking that question in a room full of train enthusiasts. You will get a lot of heartfelt, enthusiastic, and completely different answers. This train is because it's the most friendly and accessible. This train is the fastest. This train, it's the most popular, This train is proven, it's been around the longest. This train, it's the newest one, it has all the modern features. This train is the safest, that's the most reliable, that one can climb steep hills, that one goes underground, that one's free. All these reasons are good, all these reasons are true, and they are understandable, but they're missing a vital question. Where do you want to go? Because more than anything else that question will tell you which train you need to get on. You could be admiring the most magical and technologically-advanced train on the planet, but it's not going in your intended direction it ain't the train for you. Now it's true your decision might even be straightforward, if you're learning this content because you want to expand your role in your current company, and they mainly use C# and. NET, or mainly use Java, or mainly use Ruby on Rails, then whatever you're surrounded by and have a chance to use is probably the first technology to dive into. I'm more talking to you if this next step is your decision. Perhaps you've even been reading about different languages, but hesitant to commit because you're worried about what happens if you pick the wrong one. Now in practice, this is really not an issue, because as we've seen throughout this course so many of these skills apply across languages. So the first thing to ask is not what language to choose, but rather what kind of development you'd like to explore, what type of things you're interested in building. And that shifts us from what train do I pick to where am I going? You don't need some massive 5 or 10-year master plan, but if you can decide on a short to medium term goal this will immediately narrow your choices. Are you most interested in making games, you'd then consider technologies like Unity and Unreal, and you'd find certain languages more popular here, like C++. But C++ would not be a language I'd suggest if you're focus was web development. Then it might be a framework like Ruby on Rails or ASP. NET, with their associated languages, and very often JavaScript. So do you can make phone applications? And if so do you lean to Android or Apple devices. Or maybe none of these, perhaps you're more interested in writing programs for enterprise-level businesses where your goal is to work for a Fortune 500 Company. Or perhaps you'd like to work with visualizing massive amounts of data to make sense of it all. Now there are of course many other specialized areas, computer vision, audio, robotics, a host of others. And of course, I'm just scratching the surface here, being intentionally brief. There are many other options within each of these categories, but the point is, you will find some languages and technologies are simply more associated with particular types of development. And if you see this and think, but I want to try it all, well you can, you just can't do it all at the same time. So figure out what you want to do first. And if you're still not sure then let me narrow it down to my personal and unashamedly biased opinion, my top three suggestions for a first programming language, Python, Swift, or JavaScript. Why? Well first, they're all languages that would be considered easier and friendlier than some of the alternatives, but they're not introductory or beginner languages, they're all modern, practical, and pragmatic real-world programming languages. If you're interested in making websites look closer at JavaScript. If you enjoy and mainly work on Apple hardware look closer at Swift. If you can rule both of those out then look at Python. Pick a language, maybe watch an hour or two of a fundamentals course or a getting started course here at Pluralsight, and if looking at that languages doesn't then make you want to run screaming from the building, then jump in. But go into it with the idea that you might spend 3 to 6 months regularly spending time learning it, playing with it, breaking code, fixing code, before you even worry about or make a decision about what to do next. And picking a technology, whether it's one of my suggestions or something entirely different will also make your choice of programming tool easier, what application will you actually be working in to write you source code and develop your program? If you chose C# you'd use Visual Studio, if you picked Swift you'd use Xcode, if you want to make Android apps you would use Android Studio. And give yourself permission to be frustrated occasionally. This is programming, some of this is easy and some of this takes time. So expect to get stuck from time to time, and expect to work your way through it. The first languages is always the hardest to learn. Good luck and enjoy the journey. Thank you for joining me for this course, I will see you in the next one.

Course author

Simon Allardice

Simon is a staff author at Pluralsight. With over three decades of software development experience, he's programmed in every discipline: from finance to transportation, nuclear reactors to game...

Course info

Level	Beginner
Rating	★★★★★ (1100)
My rating	★★★★★

Duration	2h 50m
Released	17 Jun 2016

Share course