Primitive vs. reference types in memory

Selecting transcript lines in this section will navigate to timestamp in the video

- With primitive data types, we know exactly how much space they will take up. No matter what the specific value is. Data structures are different because the amount of space they take up often depends on how they are allocated initialized and maintained. With INTs, doubles, chars, and Booleans, we've consecutive zeroes and ones in a specific amount of space. Given the number five stored an assigned INT that takes up 32 bits, the binary value would be a bunch of zeroes and then one zero one. Traditionally these ones and zeros are stored consecutively one after the other. We are able to do this because we know exactly how much space a primitive type will take up before allocating space for it. However, for data structures we do not know how much space they will take up until we know how many items they will contain. And even how many items they contain can change. We might know at a minimum how much memory they need with one item, but not the exact amount. At their core, data structures are made up of pieces of data and ultimately primitive types that take up a specific amount of bits. Let's think about what this means and how it might work if all data was stored consecutively. Say we had a collection. Collection A with two integers that were eight bits each. So each integer was represented with eight zeros or ones. Then we had another collection, Collection B that also had two eight bit integers. If we tried to add another integer to the first collection we would overwrite the first integer in the second collection. This means by storing data consecutively in data structures, we risk overwriting and losing precious data. How can we solve this? We can use something called pointers. Instead of storing the entire data structure directly in a set of consecutive bits, we'll create an address that will point to where the structure, or sometimes part of the structure is in memory. Since we use a reference to find where the data lives in memory we call strings and other data types implemented with data structures reference types. We also use the word object to describe a value in memory referenced by an identifier. Reference types reference their specific value from an address of where the item is stored rather than direct access to the data itself. This means if the address changes, the value the variable represents also changes. Again, it's important to remember that with reference types, we are adding this extra address layer whereas with primitive types, we directly access the data. This type of memory management is the foundation of data structures. Whether the tools needed to change pointers or access addresses are available to you or not depend on the programming language. Usually this is all abstracted to you and all you have to worry about is creating the string using a programming languages syntax. However, in some languages like C++ you have to manage these pointers and manage this memory in data. But in other languages like Java and Python, this memory management is handled for you. We won't get caught up in the details of this, but it's important to know that it's happening behind the scenes so that you have a better understanding of the more complicated data structures we'll explore in this course. You'll see different types of data structures in this course from lists to hashmaps to trees. But the foundation is this idea of pointers and reference data.