Mental Model of Computing Operations

Video Transcript

Video 1 – Mental Model of a Computer

So, what do we think about when we're coding? Well, we need to have a mental model of what's going on. So, we need a mental model of the computer, and we need it at various levels of abstraction. So, at the highest level, we think of the computer as having a Central Processing Unit that can execute instructions. We think of the memory; that's where we store instructions, and we store data. And input/output devices such as the keyboard, the screen. And you can think of the disk; you might think of that as memory as well, but let's think of it as loop with an output device for now.

So, when we think about memory, it's divided into various parts. The obvious part is, we have some memory where the machine instructions are stored. As Abel talks about, we're cooking a meal, and the instructions tell us to go get the flour, or now we need to saute the potatoes. So, we have machine instructions, and we have a pointer to the next instruction that we need to execute. Now, in computer memory, there is only one memory, but it's divided by the computer into two parts. And they're called the stack and the heap.

And I'll talk about them in the moment because it's important that we understand what they're for; now the stack tends to grow upwards, so its addresses increase, whereas the heap starts at the highest address and grows down. This doesn't actually really matter, but at some stage, they are going to collide, and that'll cause problems. So, we need to avoid that. So, let's take a look quickly at what happens when we call functions. So, let's suppose we're in the main( ) function, and now we call the blue( ) function. So, the way I think about it is we're reading a book, or we're stepping through the book.

So, I'm stepping through main( ), and I reach a page that says, "call the blue( ) function". So, I stop on that page, and now I go and start reading the blue book. And I start executing the instructions in the blue book. And at some stage in the blue book, it says, "Go to the red( ) function". So now, I stop where I am in the blue book and start executing the instructions in the red book. And I go through the red book, and at some stage, we come to the end of the red book, and we pop out of that. And now, we pick up where we were in the blue book, and we go through those instructions.

And then, at some stage, we finish with the instructions in the blue book, and we return to where we left off in the main( ) function. So, we need something to manage this, popping on and off of what we call stack frames. So, we can imagine memory laid out like this, that we have the stack

to manage those stack frames that I just showed you. So, here, the blue frame or the red frame. And over on the heap, we have these references to objects, arrays, and functions, and we usually just call them all objects because, in JavaScript, everything is an object. Now we talk about memory slots, and each memory slot has an address.

So, for example, this would take up several memory slots. But in JavaScript, we talk about a reference. We don't use addresses where we can't actually get the address of what memory is being used. So, a reference is a reference to, for example, an object. And we'll see in more detail later just exactly why we're using this. So, that's the mental model that we think about. We're going to see the next level of detail in this model in the next video. So, we think about the stack and the heap, and now, we're going to put kind of flesh into those and show you exactly how they work when we have a piece of code. That's one level of mental model.

Video 2 – What Should We Be Thinking About When We are Coding?

So, what should we be thinking about when we're coding a program? Well, this tool will help you understand exactly what to think about. So here, this was written for Python, that it'll actually run JavaScript, and it's ES6 compliant. So, what we've got here is that we've got a function, and I've called it 'main'; it's an arbitrary name that takes a single argument. And then we have a variable 'y'; it's set equal to 'blue'. So, 'blue' is some function that we've got to write. And '(x)' is passed to that, and then 'y' is 'return'. And so, the value will be printed out here. Whatever is returned is going to be printed out as the result. And here, we've got a global variable.

So, let's take a look. Let's write the 'blue' function. So, here's the 'blue' function. And it takes an argument '(x)'. So, we pick it up there. And now, we let 'y = red' with '(x)' passed as an argument. So, we need a function 'red'. Let's take a look. So, here's the function 'red'. Okay, so let's take a look at this. First, it's a little confusing because we've got all these x's around. And in fact, we've got a couple of y's. So, we've got to be a little bit careful about what's happening. Let's step through this. So, here, we've got on the right-hand side, we've got the stack frames and the object heap.

Now, the stack frames here will grow down. That's just for graphical ease. Whoever wrote the app decided that they were going to put the frames going down. So, what we'll see now is that on the global frame. So, that's this kind of anonymous frame that doesn't have any name where we're typing these statements that we've got red defined, blue defined, and main defined. And they referenced functions over on the heap. So, we have these functions or objects, and so he's calling them objects here. So, we have these functions defined on the heap and these references on the global stack. Now, let's start executing, will go next, 'x = 1;'.

So, we have a variable 'x' here equal to '1'. Now, we come to the next statement. It's going to call 'main(x)'. So, we step into main. So, here we are. We go immediately to line 10. So, a stack frame called main is created. It's pushed onto the stack, so they're growing down here. And '(x)' is the

argument. And we've got 'y' as another variable that is going to be set to 'blue(x);'. So, let's execute this line, and we're going to step into the 'blue' function. So, here we are now with the blue stack frame created. So now, we're here in that stack frame. And again, we've got a variable called '(x)', that's an argument.

So, that's this one. And we've got another local variable, y, that's going to be set to 'red(x)', and we're going to step into the red stack frame. So, here we are in the 'red' function, and we're about to execute this line. And we've got another '(x)'. Now, notice these X's are all independent; they're set to the same value, but they're independent variables. So, we could change the value of one without affecting the others. They are not coupled in any way. Same with y's, we have three different y's here as a local variables.

Now, what happens is that let's step through the 'red' function. So, we have 'y' set now to '2', 'x + 1'; 'y' is set to 2. And now, we're about to return 'y'. Now, we'll see that because when we step out of this function, the x and the y's disappeared; we need something to hold that return value. So, we have something, an anonymous variable called return value that exists for a fraction. So, it exists, and it's not going to be what exists on this right-hand side here of 'y'. Okay, so the return value is going to be here. And now, when we get back to that line, we stepped over it there, and y was set to 2.

Now, we're going to 'return y + 1;', so that return value from blue is set to 3. And now, the 'y' here is going to be set to '3', here we are in main. Notice that blue and red have now disappeared. Their x's and y's are gone. So, the scope; so that's maybe a new word for you. So, the scope of a variable depends on where it's defined. But in here, the argument is defined when the function is called, and any local variables are defined there on their scope; they only exist as long as we're inside this function.

You will see that 'let' for a variable says that it's block scoped, but the block, in this case, is the same as the function scope. This argument has real function scope. So, it slightly different. We'll see when we have a 'let' in a for loop that its scope is the blocker of the for loop. But in this case, these are going to have the scope of the function. So, again, in 'blue', those local variables only existed inside that function. So now, they've gone. Now, we're left with main, and it's about to return. And so, we've got this return value, and this x and this y are about to disappear. But 3 now is going to be returned. And we're about now to execute this return here.

So, we've got Return value 3. And now, it's going to be printed, and we'll see it printed up here. Okay, so that's what you think about when you're executing code. You need to pay attention to the type of the return value. You need to pay attention to the type of argument. You'll see when we're passing in objects and arrays that they behave slightly differently to this. That arrays and objects exist over here on this heap and not in the stack frame. So, that gives them a different scope. They can exist. You'll notice these functions continued to exist no matter where we were in those functions.

So, objects on the heap have a different scope than on the stack frame. So, on the heap, their lifetime depends on whether there's a reference to them or not. As long as they're being referenced by something over on the stack frame, then they'll continue to exist. But as you saw, just like the ones, the local variables, it's possible that objects on the heap can also disappear. But this is what you think about when you're running programs or when you're programming.

Okay, let's just run through it quickly to watch the stack grow. There's main; there's blue, there's red. We're coming back from red, and it disappears. It's popped off the stack. We're coming, returning from blue. It's popped off the stack, and we're in main, and it too, disappears, and we're left with the Global frame. Okay, so this is what you should be thinking about when you're programming because it can be a little tricky sometimes. This will keep you out of danger.

Video 3 –Basic Types

So, let's take a quick look at the different types that exist in JavaScript. So, here we've got a whole load of variables and on the frame here, let's see where they are going to be constructed. So, here's our stack frame, and over on the right here is our heap where the objects are stored. So, the first one is a number; it's an integer number. Now, we've got a floating-point number; it looks like it's pi. Then not a number at all; NaN is when it's not a number. Infinity; 'Infinity', '-Infinity'. Now, we have a string. All of these are primitive types, boolean, 'null', 'undefined'. Now, we come to an array or a list. Notice that this can have various types inside the array.

So, here's a character or another character, and then we have numbers, and finally another character. So, arrays can store any of the types in JavaScript, store other arrays or objects, et cetera. So, here we have our first object. Notice that arrays and objects are not stored in on the stack, the heap. So, now we just have an integer 5, and it's going to be used in this array. So, let's take a look at this array. So, it's got just the number 'i', so, '5'. So, the first, let me, I'm going to scoot this up so you can see it. So, let's go through this 'obj_list'. So, we have an integer 'i'.

So, the obj_list 0 is 5. Then we have an object in the first location. And that object consists of a property '{foo:', which will have the value 6. So, here it is. And this property points to or refers to an array '[1, 2, 3]'. And then finally, we have another object 'bar'; "bar' has a value 7. Now, we have 'obj.name = 'Jane';' this is referring to this one. So, now the name, let's see. I need to stretch this down so I can do next. So, you saw the name change there and now 'obj[" name"] = "Jake"; ', and you see we're referring to it now with the bracket notation and above by the dot notation. Okay, so I suggest you play around with these, and these are the basic types in JavaScript.

Video 4 – Scope of Variables - Why Stack Frames

So, let's take a look at the scope of the variable. A variable comes into existence, and we can define it either as a const, a var, a let, or it could be an argument and come into scope when that

function is called. Now, const, var, and the arguments have function scope, so they exist as long as we're within that function. And as soon as we come out of that function and the function on the stack disappears, then they go out of existence. Now, let has even less scope. It has what we call block scope. So, it could have function scope if there are no if it's not defined in another block within the function. Now, a block is anything that has { }.

So, if we have an if statement like '(x<3)', { }, and then { }, we let some variable like y come into existence. Then it's only going to exist within those { } as soon as we're out of that if statement, that y is going to cease to exist. The for loop is a little bit more complex. It actually creates two blocks, and they're both for x, but one is with 'x = 0;' And then one is the counter that counts how many times we've been through the loop. And again, they only exist as long as we're within those { } after the for statement. And finally, we have the argument that exists within the function. Let's take a look at these in action.

So, here we've got a function 'blue' that takes an argument '(x)'. We've got a 'var y = 0;' here. Then we put an 'if' statement with a block. So, we have '{let x = y; x = x+1;' within that block. And they're only going to exist; all of these will just exist within that function. This 'var' is a global, and we start on execution on line 11. And so, that's going to exist as long as we're within this program; it's global. So, it'll exist even after we've finished the computation here on line 13; it'll still be in existence. Okay, let's step through. So, here we're going to go into 'blue', and we're going to pass in the value '1'; x is 1 there; the global. Now, in 'blue'. Here we see the stack frame for blue and x is 1, and it's going to have function scope. It's going to exist until line 9.

Why? Again, it's defined here within the function, and it's going to exist all the way through the function to line 9. Now, when we get to the if statement, will see a block comes into existence nets from the '{' on line 4 to the '}' on line 6. And we see over here on the right, (block 1); and x is undefined at the moment, but then we give it a value, and it's got a value 0. And now, we're going to increment it. But it disappears as well; we increment, and it steps out of that block. So, the x is gone there. Now, we increment y to 1, and we return 'y + 1'. So, we're going to have a return value of 2. And then the blue stack frame is going to go out of existence, and we're left with the global x. Okay, so scope is fairly simple once you realize what's a block and what's a function scope. Hopefully, this has cleared up scope for you.

Video 5 – Equals Operator

So, the author of a computing language is a little bit like Humpty Dumpty in "Alice, Through the Looking Glass." Who said, "A word means what I want it to mean. Nothing more, nothing less." So, in the computer language, we know that there are instructions, but the question is, what do they mean? So, here we have, for example, 'a = a+1'. Now, that can't be true. Usually, in mathematics, 'a' can't be equal to itself plus something else. It's equal to itself. Now, again, 'b = a'. What does that mean? Well, it depends on what 'a' is. And we need to understand what the author, in this case, for JavaScript, it was Brendan Eich, what Brendan Eich decided.

And here, we have 'c = "hello" ', which is a string, '+ 3', which is a number. So, the question is, what happens? Well, to understand it, we need to develop a mental model of why the author made the decisions that they did. So, let's take a look at the equals operator. So, here I've said 'let a = 2;'. Now, in computing, the equal sign doesn't mean that the left-hand side is equal to the right-hand side. What it means is that we assign the value on the right-hand side to the name, on the left-hand side to the variable, if you like, of a left-hand side.

So, 'a' is assigned '2' or 'a' gets '2'. So, here we see the name 'a' and its value '2'. So, 'a' gets '2'. And that makes sense because then we can say 'a = a+3;' and it makes sense that 'a' now is reassigned to the value of the right-hand side. And that is 'a', the value of 'a'. So, that's 2 + 3. So, don't think of it as equals, as whatever is on the left-hand side is being assigned whatever the value is on the right-hand side. Okay. Now, suppose we say, 'let b = a'. Now, what does this mean? Well, it means there's now a new variable 'b', and it's being assigned whatever the value of 'a' is. So, that's 5.

Now, 'b' and 'a' are totally independent. That if I now reassign 'a', saying 'a = a + 2;' , then 'a' is now 7 and 'b' is still 5. So, these are primitive types; these are numbers and floats. That's another primitive type. But the same kind of thing happens if this is, if 'a' now, let's make 'a' string. So, if 'a' is ' 'apple' '. Now, we got to be a little careful here. With strings, we can concatenate a string with another string. So, let's have 'pear'. So, now 'a' is ''apple pear'' while, and 'a' gets added '2'. Let's get rid of these a minute. So, now we 'let b = a'. So, 'b' is ''apple pear''. Now, we can reassign 'a = 'tomato' '. And you see that 'b' stays the same.

So, this is the way that the equality sign works for primitive types. Now, let's take a look how it works if we do, for example, an array. Notice that the array is an object, and it gets put on the heap. So, now 'a' is a reference to this object. So, now the equality sign is a little different because it's referencing this object over here, which is a data structure which has 'a[0]' is a number, and 'a[1]' is another number. So, let's see what happens when we say 'let b = a;'. Notice that 'b' now references the same object as 'a'.

So, now if I make a change to 'b', so 'b[0]', for example, '= 5'. Notice what is 'a[0]'? Let's just print out 'a'. So, 'a' is 5 and 2, and 'b' is 5 and 2. So, arrays behave totally differently to numbers and strings. So, we have to be very careful. We can have any number of variable names associated with the same object. So, it doesn't matter how many times we do it. There is still only one object, and we've got various references to it. Now, if we assign, for example, let's make 'd' a different array. So, here, now I created a new array and 'd' references that. Now, notice if I reassign 'd', the variable name, say to 'a', then this object will have no references to it.

And because it's on the heap, the heap does garbage collection. So, let's do that. So, 'd = a'. Now, 'd' is reassigned to 'a', and this object disappears. Okay, so make sure that you understand how arrays work. And we'll see that objects work very much the same way. Let's do the same thing now with an object. So, 'a' is this object. So, 'a.x' is how we refer to this value of 2. And if we say 'let b = a', we get exactly the same behaviors with arrays. The 'b' now references this object. And

if we change 'a.x', then '(b.x)' also changes. So, this is the way to think of it. Be clear about what the variable name is referencing.

Video 6 – Primitive Types Pass by Value into Function

So, let's take a look at passing a primitive type, in this case, a number, into a function. So, we've got a function 'f1', that has an argument '(b)'. And we're going to call it down here and pass in this global variable '(a)'. Now, notice because we've said 'let' here that we don't have 'a' on this stack frame yet. We've got the function because we gave it as a function definition. Notice that if we do say 'const f1 = ' then, at the beginning, notice we don't have that defined. f1 is not defined yet.

So, remember, this doesn't get hoisted. But we, okay, because we're not going to call it until line 8. Okay, let's step through this. So, now we see that 'a' is defined. And now, 'f1' references the function. And we are on line 8, and we're about to call 'f1' with the value '1'. Now, notice we get 'f1' frame pushed onto the stack, and the argument 'b' is defined as '1'. And so, notice that 'a' and 'b' are totally separated. Now, we have 'y'; it's undefined at the moment. But next step it gets set to '1' because 'b' is '1'.

Now, we're going to return, and the return value is '1'. And remember, now the stack is going to disappear. So, 'b' and 'y' are going to disappear. The return value is held so that we can pass it back to result on line 8. So, here we are on line 8, and result now is set to 1. And we can print it out. So, this is passing a primitive type, a number. This would also be the same for a string. If we had a string here, that would behave the same way. So, let's do a ' 'hello'; '. And what I mean by that is we get a copy of the primitive type in the function.

So, let's, this should be fine. Let's go back to the start. So, here again. Now, we've got 'a' set to hello. And we're about, we've got the function defined. That f1 that was a reference to the function, so we can call the function, and 'b' is set, the argument is set to a copy; it gets a copy of 'a'. And now, 'y' is set to the same thing; it's returned. And we love the result, hello. So, that's passed by value, and we'll deal with objects and arrays which are passed by reference in the next video.

Video 7 – Objects and Arrays Pass by Objects into Function

Let's take a look now at what happens with objects. So, 'a' is now an object. And let's change this. So, 'b.x = 3 return b;' Let's see what happens here. So, we get an object 'a' created here. And it has a variable 'x', that's the number, and it's '1'. So, 'a' references that object. Now, we call 'f1' with '(a)' passed in as the argument. So, let's step into here. Now notice that we have 'b' gets the same reference as 'a'; it references the same object. So, 'b' now is pointing at that object. Now we do 'b.x = 3'.

So, 'b.x' is the same as 'a.x', and they're both 3. Now, we come 'return b'. So, we're returning that. So, we get a temporary here created. Because inside here, this 'b' is going to disappear. So now,

the frame disappears; 'b' is gone, 'a' still remains, and we've got '(result) now, it has taken up that reference that return value 'b'. Now, '(result)' is referencing the same object. And so, when we print it out, we'll get '{x:3}'. So, just remember that you're operating on the same object or the same variable inside the function that we're basically passing that into the function.

It's modifying it. What do you think? Let's suppose this time we don't catch the value, but we print out '(a)'. So, do we just call the function? What do you think is going to happen? Let's take a look. So, we call the function; 'b' is updated, 'a' is updated at the same time. So, when we print out 'a', it's '3'. So, let's not even return the value. Suppose we did this. If it's an object or an array being passed in, we actually don't, and we modify it.

We don't need to return it. Let's take a quick look. Why? Because we step in, 'b' is modifying 'a' as well, at the same time. So, 'a' now has the value 3. There was no need for us to actually return it there. So, make sure you understand what's going on; when we're passing a variable into a function, we need to understand clearly whether it's a primitive type or whether it's an object or an array because they behave differently. I'm going to leave it to you to run this with an array instead of an object.

Video 8 – Why Arrays and Objects are Passed by Reference

So, have you thought about why primitives like numbers are passed by value, we make a copy of them? So, for example, here, X = 5. So, the name is X, and the value is 5. And now, if we say Y = X, then the name is Y, and we get a new copy of 5, and it's placed here. And so, we have two independent variables now, X and Y, and we can change one, and the other stays the same. But now, when we have arrays, it's a little bit different. These arrays can be huge. And if I have A equals this large array, say with a few 100 books or a 1000 or 10,000 books, or whatever objects we have. And now I say B = A.

I don't want to get a copy of this because it would take up a huge amount of memory. So, in this case, we say when B =A, we just get a pointer or a reference to this one array, this single array. If you want to make a copy of it, then you have to go to some effort. But the default is that arrays and objects are passed by reference. So, we don't get a copy; we just get a copy of that reference. We point to the same place. Why that happens? So, let's suppose here, I've got a 'var books', and it's an array. And I have a 'function' that actually produces a 'book'. If I give it a 'title', it returns an object with the 'title'.

I got a little 'factory' here that I can pass a number like '(n)', and tell it to make n books. And it creates a 'book', and it gives the title as '('a'+ i)'. Now, 'i' is going to be concatenated, going to be cast to be a string or a character in this case. And it's going to be concatenated with ' 'a' '. So, the first one will be 'a[0]', then 'a[1]' hour. We 'push' those into the array 'books'. So, 'i' can make any number. And for convenience, here, I'm going to 'return books;'. Now, I've got a clever little

function that you won't have seen before. But basically, what it does, it loops over all the books, and it creates.

Since the book is an object, it adds another property to that object, and it adds the property 'owner'. And it makes ' 'me' ' the owner. And so, here I've got a 'factory', we only produce '(3)' books because I want to show you on the graphics how it works. But now, I want to stamp all the books. Now, I don't want a copy of this book's array. It could be huge. What I need to do is to stamp the original array and say that they now belong to me. So, let's see how it works. So, here we're at 'factory', and we're going to make three books. So, here we are. We're in the factory. Here's our loop. 'n' is 3.

We just made one book. And now, we're going to put that book. Well, we're just about to return that book, and we're going to push it into books. So, that's our next step. Where's books? So, books is way up here, and it's an empty array. But now, we're going to add to it. There we are. Now, books, you'll see there's a little arrow that, and it points to an array and it's got one book, "a0". Now, let's keep going. Keep looping. Now, I made another book. And there we are. Now, it's in the array, and I'll make the third book. I think it's going to be off the screen, but that's fine. Yes, there it is. Scroll up, so you can see it.

So, now we've made all the books. Now, let's just finish that off. And we return books. Now, I want to stamp the books with my mark. I want to make ' 'me' ' the owner. Now, I'm going to pass into 'stampBooks', the 'books' array. In fact, I could get access to it anyway because it's a global variable. You'll see up here. But I decide to pass it in. And let's watch it make the stamp. Here we are in stampBooks. Now, we're going to put my mark on each book. Here's an item; there it is. And it's pointing to the first object. It adds "me" as the owner.

Next item, next book adds "me" as the owner. Final item adds "me" as the owner. Now, we're finished. We can pass out of that. And we've got the books now. Our books array has been updated, but we didn't have to make a copy of it. So, that's why we passed by reference, objects, and arrays. And why with primitive types like strings and numbers, we can afford to make a copy of them. But that's the logic behind it, of what the designers of the language, why they decided to do one, one way, and arrays, and objects another way.

Video 9 - Pass and Fire Function

Let's take a look how we might fire a function by passing it into another function. So, here we've got two functions, 'red' and 'blue'; 'red' just returns the value of pi '3.141'. And 'blue' calls whatever's passed to it. So here, the '(red)' function is going to be passed into 'blue', and '(f)' is the argument. So, 'f' will now reference the 'red' function. And this will fire the 'red' function and capture the return value in 'y;' and return it. So, let's see how this works. So, to begin with, we've got 'red' and 'x' defined. Notice that 'blue' hasn't yet. It doesn't get hoisted.

So, it doesn't yet appear in the Global frame; 'vars' do get hoisted. So, it'll, it knows all about the 'red', doesn't yet know about the 'blue'. Let's step to next. So now, the 'red' function is defined. Now the 'blue' function is also defined, So, we have a reference to 'red' and 'blue'; 'x' is undefined because we haven't yet executed. Now, we're going to step into 'blue'. So, we're going to pass the 'red' function into 'blue'. So, 'f' now references here this argument 'f' now references the 'red' function. We have 'y' is undefined, next step.

So, we go into the 'red' function because we're firing it here. That 'f()' fires the function. The return value '3.14' is assigned to 'y'. So, here we are in 'blue' and 'y' is '3.14', and now it's going to be returned. So, the return value, the temporary variable is set, and 'x' will now notice 'x' is undefined still, x will get the value 3.14. So, this is how you can pass functions around. It's very easy in JavaScript to pass functions, and we'll find it's really useful when we come to callbacks; we'll see that we use this extensively.

Video 10 - Casting Types

So, we've seen that JavaScript test types. So, here 'a' is a primitive type, and it's a string. And here 'b' is a number '33;', it's an integer. Now, what happens if we let 'a = a +b'? Well, what happens is 'b' gets cast to be a string. So, 'a' is a string. The '+' operator for a string is concatenation. So now, it cast 'b' to be a string as well. And we get the result "2233". Now, if that's what we wanted to happen, then that's all well and good. But to make our code clear to anyone else reading it, we should explicitly cast 'b', the number to be a string.

Now, we can see that 'a' is a string and 'b' is being cast to a string, and so they get concatenated. What if we wanted it to happen the other way around? So now, let's suppose we have, let's take 'b = b + a'. What happens here? Well, 'b' is still a string; it gets cast to be a string. But if we wanted, we could make this a 'Number' and now 'b' is '33'; '(a)' is being cast to a number. So, it gets cast to "22", and the result is 55. Now be careful because not everything can be cast to a number.

Suppose we had ' 'hello' '; so now we see that 'b' is not a number. It doesn't know how to cast that string to a number, so it returns not a number. Let's take a look at a couple of other things. Let's take a look at, for example, let's suppose 'b', let's put 'b' back to being a number. So, let's get rid of that. And let say 'b = b + true;'. So, 'true' is a boolean, and any number greater than 0 or less than 0 is true. So, 'true' gets cast to be 1. So, here, now, now this is quite unusual, you shouldn't really do this at all.

But, let's see what happens if we say, 'true + true;'. Yeah, it gets 33, 34, 35; they're cast to 1s. Let's see what happens now. We know that a boolean should be equal to, the boolean 'true' should be equal to 'true', and that would therefore be a 1, and b, 33 +, So, the result of this is being cast to be a number 1. So, be careful with casting. You've seen two string and number to cast a string to a number. So, make them explicit if you can. But just be careful of some of these unusual implicit casts that you might not be expecting.

Video 11 - Model Factory

Okay, let's take a look at how we can make lots of objects. So, here I've got a variable 'locations', which is an empty array. Now, I've written a function called 'makePoint' that takes 'x' and 'y' as arguments and creates a point 'p' that has property 'x', which will be given whatever that value of the argument 'x' is, and a property 'y', that will be given that y value, and then, we 'return p;'. So, we make a point '(3, 4);', catch it with 'p', and then we 'push' that onto the 'locations' array, and print it out. So, let's take a look at this working. Let's go to the start, and we define locations as an empty array.

Notice, it's on the heap over there. And now, we're going to step into makePoint with 3 and 4. So, this x now is 3, and this one 4. At the moment, p is undefined, but the next step, we've created the object, and we've got a reference to it called p. And now, we pass that back, we return it. And a temporary Return value is created because p is going to go out of existence in a moment. So, here you can imagine that on this right-hand side. Now, there is this Return value, which is this object, and this '(p)' will be put equal to it.

So, p now, here in the main frame, in the Global frame points to that object. Now, we push it onto locations, create an entry into the array. It was empty. And now, pointed to the same object p. Okay, now how could we create lots of these? Well, let's suppose, let's create a 'factory', and we create an object '(n)' points, that's our goal. So, there's the function. Now, we can loop over 'i < n; i++)'. Okay, so now we've got a loop. And what we want to do? Well, we want to make a point.

Now, let's feed in. I'm just going to feed the 'i' value in for the x-coordinate and the 'i' value for the y-coordinate for now. But we could get those from somewhere, either random or whatever. But we made a point. Now that we've made it, we should push it into locations. Now, so we just want to call 'factory' and let's do '(3)', and we'll then print out 'locations'. So, let's step through it. So, we've got locations empty array, makePoint is the function, factory is the function.

Now, we're going to step into factory. So, we step in with n equals to 3. And this block here is, let me push this in a little bit here just so you can see. That is (block 1) there, i. And there'll be a second part to that block. This is part of this, these statements here, this loop. Now, let's keep stepping through this. So, now we call makePoint. So this point here, and x and y are both 0 because i is 0 at the moment. If you look in here, i is 0. Now, we create the point and return it.

And so, we come out of makePoint, and sorry, at the moment, we've just got p, and it's got this, this object, and now, we push it into locations. Yep, then there we push it into the 0th entry of that array, locations. 'p' has gone, and we've just got locations, and we're coming around the second time around the loop. So again, we make another point and then return it and push it into locations. We've got p. Now, we've got the second one in locations. Go around the third. And now, we've got three objects. We'll squish down, it's gone off the page, but we have three objects here in locations.

And we can step out of the factory because we finished, and now, we can print out those locations. So, this is the way we make a lot of objects. And we handle all the objects in an array. So, it's the container for our world, if you like. So, this is how factory functions work, and that can make things, can make lots of things. We could make thousands of these if we wanted to. Okay, so take a look at this code for creating objects because we'll be using it in some of our exercises.

Video 12 - Hoisting Functions & Vars

So, let's take a look at two different ways of declaring a function. If we say function red( ), that's called a function declaration. But if we do, say, var blue or const blue = function(), that's called a function expression, and they behave slightly differently. So, let's take a look and practice what it means. So here, I've defined 'var x', and I'm executing a function called 'red( );', that's defined here below it. So, I'm using the function before it's actually before I get there if you'd like. But it turns out that the interpreter reads ahead; it reads all your code.

And if it's a function, a function declaration, then you see that even though we're at the very first line, red is already referencing that function. So, we say that that function has been hoisted. It knows about it, and when it gets to this first line, it's fine. It executes red. And x will get set to the return value, which is 3.14. Now, this would not be the case if we, for example, did 'const red = function red( )'. What we'll see now is that red is not defined. It hasn't been; it hasn't got there yet. It's actually read it, but it doesn't promote it, it doesn't hoist it.

So, here we're going to get a ReferenceError that red is not defined. So, that's the big difference between a function declaration and a function expression. Now, let's take a look at how we could put this right? So, one obvious way is to read the function first and then execute, and that's fine. Now, notice that even though if I'm on the first line here, it knows about this 'var x'. Notice over here it's got x. It's undefined because it hasn't executed 'red( )' yet, but it knows about it already.

However, if we use, and that's because this has been hoisted, the 'var x' has been hoisted. It knows about that. It doesn't execute it, that doesn't get hoisted, but the 'var x' does. Now, if we use 'let' or 'const', it doesn't get noticed that we've got red undefined. Red gets defined, and now, you will see that the Return value, it still hasn't. Yes, now it's got 'x' is 3.14. Now, it knows about that. So, 'let' and 'const', don't get hoisted. But, an expression 'var' does. That's hoisting, and the main thing to remember is about the difference between a function declaration and a function expression.

Video 13 - Debugging In The Chrome IDE

So, I want to show you more about the environment that we're programming in. At the moment, we're running our programs inside the Chrome browser. So, we're on our computer, usually. And in the browser, we're bringing up, for example, the IDE. So, we'll perhaps drag and drop our HTML

program onto the browser. And then, we'll bring up the IDE. So, for example, here, I just brought up the IDE. And over here, you'll see the Stack Frame. So, the stack frame that you'll see when you are running your program in this mode already has all the variables that the Chrome browser uses.

But what it means is that you will see a lot of variables that are not in your HTML program but that related to the browser. That the browser already has a considerable number of, for example, Global Variables that you can see. One of them is called window, which we do things like window on load, and that is because we're running inside the browser. So, yeah, I've got a program, and I can find that program and drag it on to the browser window. So, let me do that. Let me just bring up again. So, here's my browser. Let me go and get the file that I need. So, the file it's called f2, and I'm going to drag it onto the browser, and I'll bring up the IDE.

So, here's the IDE, and here's the code that we just loaded, and I'm going to put a breakpoint here. So, now I'm going to reload that code, and it breaks in the IDE. Now, here, let me scoot this along a little bit. I don't want it to disappear. What we'll see is that we've got the Call Stack. So, let's take a look at here. We're going to step into this function, 'aaf2'. I numbered, I called all these things 'aa', so that they'd appear at the top of the list of variables because there's a lot of variables. So, here you see, I've got the array 'aaaA', and here it is here. And the function 'aaf1'.

So, there's that function, and then 'aaf2'. So, we're going to call 'aaf2'. Let's keep track of the Call Stack, okay? So, here let's step. So, here you'll see on the Call Stack, we're now stepped into 'aaf2'. 'aaf2' is going to call 'aaf1'. Yes, here. So, let's keep going, and it's going to call 'aaf1'. So, now we're in 'aaf1'. But look, we still have 'aaf2'. So, that's the Call Stack. Now, you can also look at the variables. So, over here, we've got the local variables. So, 'b', for example, here, that's a local variable. Notice, we've always got this Window, and this is to do with the browser.

So, to a large extent, you ignore those. Here, we've got a lot of globals that have nothing to do with us. These are all part of the fact that we're running in the browser, and these are the browser variables. So, to a large extent, we can ignore those globals. And we can usually focus just on the local ones. Okay, so that's why the stack is important. Often, if you have a problem, it will give you a stack trace. So, what I mean is this is a stack trace that, let's suppose we got an error here, divide by 0 or something like that.

Then we'd know that aaf1 was called by aaf2 and aaf2 was called by (anonymous). Line 14. So, here. This is what they're calling anonymous, okay? So, that's how you go about debugging a program and why you may be seeing a lot of variables that are not yours. So if you go to the Console, 'windows' or the 'window' is one of the variables. And you'll see that there were a lot of functions associated with that. So, that's the complication of running inside the browser. However, the upside is we have a great graphics available to us, and that we need to understand the browser because that's how web computing is done these days.

Video 14 - What We See in Chrome When Debugging

So, we've been using so that we can see the stack frames and the heap with its objects. Now, we're going to be using the browser IDE. So, let's take a quick look. So, here's the same code, and I've put it into an HTML file. So, we have '< script >' tags. So, we can put this now; we can drag this onto the browser. So, let's do that. So, here's the browser, and I've got my blue.html; I'll drag it on, and that's in the browser. Now we need to bring up the IDE. So, let's take a quick look at this. I'm going to scoot this along. So, I'm going to put a breakpoint here on line 16, and I'll recall that function. So, we'll break at line 16.

Now, we're going to watch the Call Stack, and we'll watch the Global and local variables. So, let's take a look where we've broken at line 16 here now. So, let's step into, we're going to call 'main(x)'. So, now we step into that, and we'll see that main is on the Call Stack here. So, it doesn't give us as many details. It does show us that we've got this. So, and that's pointing to Window. We've got x :1, and y : undefined. So, we're on this line; we're definitely in 'main'. Let's go to the next statement goes to 'blue'. So, 'blue' is going to go to 'red'.

So, now we're in the blue stack, and now we're in the red stack. So, you see all the stacks are there. Now, this local refers to the 'red' and '(x)' is 1 going in, and we're going to increase it by 1. So, the Return value is 2. And now, we're in. well, no, we're still there. Let's go back to the blue. Now, we're in the blue, and red's return value was 2; so, y is 2. And now, we're back to return, so we've got the return value of 2, that is going to go into y. And in main, we should see y as 2. This is main. And yeah, we've stepped out, and x now in main.

So, the debugging in the IDE is quite good. It does show us the stack frames. It doesn't show us as much detail about the stack frame and the heap where we have all our objects. But it's pretty good. The globals are a little bit of a mess because we're running inside the browser. So, we've got a lot of variables already defined, and that's especially if we type things like 'window'. We'll see that there's lots of functions associated with that. But if we stopped our program, for example, let's reload the program.

And so, we're at this breakpoint here. If we go to the Console, we will be able to take a look and see that we've got 'main', is there; 'blue'. So, we can call these, we could call 'blue', for example, with the value (6); and it'll return the value 7. So, we can execute our functions from here. So, the IDE is going to be our main place where we debug. At some stage, we will have our mental model homes. So, we won't need to go back to the Python tutor to see objects being created. But the IDE has excellent debug facilities. And so, this is what we're mainly going to be using.

Video 15 - Debugging Introduction

Debugging is hard. You will get better at it as you get more experience. But at the beginning, it can be a bit overwhelming. For this reason, start off with small examples, walk through that code. Make sure you understand what all the pieces are doing. And as you build up that confidence and you get more comfortable with more of the errors, it will become much easier to do. However, this is something that we all practice regardless of the number of years that we have been doing. And this is something that we're constantly striving to get better at. So, you start early, start small, and invest time; it's a worthwhile investment.

Video 16 - Basics Of Debugging

Debugging is a skill you will develop throughout your career. In time, it will become much easier to understand why something doesn't work and what the source of the problem is. And here at the start will give you some basic guidelines and how to walk through your code, how to understand the execution order that's taking place in your program. So, we'll start off here by creating a blank file. I'm going to save that file to a directory called sample. I'm going to call it debug.html. And within it, I'm going to write my first instruction.

And you'll see me along the way that I'm making small changes and then inspecting the functionality to see if it is what I expect. So, I'm going to say here simply 'Hello from Abel's debug file'. I'm going to go ahead and save that. And now, I'm going to load that file into the browser. I'm going to go ahead and drag over my finder, my File Explorer. And as you can see behind that, I have a browser tab, and right now, it's simply blank. It doesn't have anything. Now, I'm going to go ahead and drag and drop this file.

As you can see there, I have this text that is displayed that simply says, "Hello from Abel's debug file". Now, let's go back to the editor. And here we're going to make a new edition, and we're going to enter content that is different. We're going to enter 'script', and we're going to denote it with a '< script >' tag or passing an instruction to the browser, telling it to treat this differently. The text that you see above on line one is pure text. And now, we're going to write some script. And I'm going to pass a very simple instruction; it's going to be 'console.log( )'. And inside of it, all I'm going to do is say ' 'hello' '.

I'm going to go ahead and save that. And now, move back to the browser. And within the browser, we're going to open up our developer tools. Open up the menu in your browser and dig through and find your developer tools. When you open them, you will see the following. As you can see here, we have a Console, we have Sources, and within the Sources, we have access to the file that we just loaded. You can see here, debug.html. And we can see the code much like with it in the editor, but now the browser itself. And soon, we will walk through the execution using Sources.

But for now, all we have is that 'console.log' saying '('hello')'. And if we reload the file, we should see that at the Console, and we do. So, before we continue, let's recap what we did. We created a very simple file in our editor, and then we loaded that file onto the browser. We opened up the developer tools. We took a look at the code under Sources. And then, within the Console, we verified the functionality that we expected from that 'console.log'. Now, let's go back to the editor and write something that is a little bit more fun. To do that, we will declare a few more variables.

So, I'll start off with a 'list', and I will initialize this to be an empty string ' ' '; '. Below it, I will write a list of names. And I'm going to enter here three names, which will be '['john', 'paul', 'peter'];' Next, I'm going to write a loop that will walk through the content of names. And I'm going to start off by writing 'for', and as you can see here, the editor is making a number of suggestions for me. I could choose this, and this would create or would write a snippet with most of what I'm going to write. But since this is the first time we're doing it, I'm going to write it by hand. So, I'm going to start off with a counter.

And I'm going to call this counter 'i'. And I'm going to initialize it with the value of '0;'. Then this counter is going to run as long as the length of names is less, or as long as 'i' is less than the length of names. So, I'm going to enter here 'names.length;', and then I'm going to increase in each loop the counter, which is 'i' by one. And this is the shorthand on how you increase a counter, in this case, by one. Now, I'm going to write the body of that loop. And all I'm going to do at first, remember we're doing this partial verification of what we assume the behavior or the execution to be, is we're going to write 'console' as before, '.log'.

And inside of it, I'm simply going to enter the countered that we're using, which is '(i);'. Again, to review, the for loop has three parts. The first one is the variable that you will use as a counter. In this case, I'm starting off with 'i' and assigning it the value of '0'. And then, I then have in the middle my ending condition. This is going to execute as long as 'i' is less than the length of names. And then on each loop, run, where I'm going to increase the counter by one, and I'm going to be writing to the console those values.

So, I'm going to go ahead and go back to the browser now. And within the browser, I'm going to go ahead and reload the page. And as you can see, we got 0, then 1, then 2. Now, I'm going to go ahead and move to Sources. And I'm going to go ahead and put what is called a breakpoint. And the reason it's called that is because it stops at that point during execution. And now, I'm going to go ahead and reload the page once more. And as you can see, it has paused at that point. At this point, we can go ahead and mouseover the things that we have in the browser window.

And you can see there that the value of 'i' is '0'. And if I mouseover 'names', you can see there the values of 'names'. And if we go back to 'list', we can see that it's an empty string. Now, if I run again, if I step through till the next time it stops, our counter should have increased. There it is. You can see that the value is now 1. And then we continue this through; we'd walk through all of the values until we finish. So, I'm going to simply go ahead and run all the way till the end, and now we've completed the execution.

So, that worked; let's go back to our editor. And now, let's use the contents of names, the looping that's taking place, and the variable that we have on line 6, which has 'list', to create a long string that holds all of the names. So, we'll start off by getting a handle on each of the items as we're walking through them. And we'll start off here with a 'const'; I'll call it 'name'. And then I'm going to get access to the item on that list. And we're going to take advantage of the counter here to walk through each of those items. And the counter will not go outside of the limits because we're stopping before we surpass the length.

So, I'm going to go ahead and enter that. And then on the next line, I'm going to add that to the 'list'. So, just the same way, we are increasing our counter, here where we did 'i++'. I'm now increasing the list by adding the 'name' on each loop of the for loop. So, once we're done, now we're going to write to the 'console'. And we're going to write the value of '(list)' to the 'console'. So, let's go ahead and save this, and let's go ahead and move back. And within the browser, let's reload the page. And let's go ahead and remove the pausing that's taking place here.

And as you can see there, we have the new code. Let's take a look at the Console. And as you can see, we have what we expected. We have john, paul, and peter. Now, there's no spacing between these three. So, we'd like to have that. Let's go ahead and go back to the editor and add a space. And let's do that here. We're simply going to add a space on every loop after every name. Let's go back to the page, to the browser page, and let's go ahead and reload that code. And as you can see there, as expected, we get a nice space in between john, paul, and peter.

Let's recap what we did. We created a very small, simple file. We started off with just a string of text that you see here at the top. We then slowly added more functionality at every step of the way, verifying that what we thought was taking place was indeed happening. This is a very good practice. This incremental testing to make sure that your understanding does not get very far from the reality. So now, it's your turn. Make sure you can replicate what you saw. Go ahead and experiment; play with it. This is the best way to learn.

Video 17 - Infinite Loop

Let's write a loop that looks for a number. I'm going to start off by declaring a few variables. The first one will be 'notFound', which we will use to indicate within our logic if we have found the number or not. And so, I will start it off at 'true', meaning we have not found the number. Then I'm going to create a 'counter'. And I will initialize that counter at '0;'. Then I will write a 'while' loop. And this while loop is going to run as long as we have not found the number we're looking for. Now, in the body of the while loop, I'm going to have a check. I'm going to check that the counter if the '(counter === 5000)', and this just arbitrarily is a number I'm looking for.

And then inside of it, if I find it, I'm going to write to the 'console' that I have reached 5000, or I have '('found 5000')'. 'else', I'm going to write to the 'console' that I have not found it. The message I will write is '('still looking')'. And outside of the conditional statement, we will increase our counter

by '1' on each loop run through, okay? So, to overview, we have written, we have written a couple of variables 'notFound' and 'counter'. Then we have a loop that iterates in that on each loop increases the counter by '1'. We're then checking to see if we have matched '5000'.

If we do, we write it to the 'console'; otherwise, we say '('still looking')'. So, let's go ahead and run this code on the browser. And the browser, I am on a blank tab. And in a moment, I'm going to go ahead and drag in the file that we're working on. Before I do that, I'm going to go ahead and open up the developer tools. And I have it on the Console where we will see the messages that we were writing within our logic. So, I'm going to go ahead and bring in that file, which I've called debug.html. And you can immediately see that something's going on here.

We went right past the '5000', we found it. But the loop is still going. And in fact, my browser is becoming unresponsive. And in some cases, you might even crash your machine or your browser. So, something is wrong in what we wrote. I'm going to go ahead and close that tab because otherwise, it'll use all my resources. And now, let's take a look at our code here. We can see that we're looping. We have a condition here that we are checking for. However, we never set that condition to false. And so, our loop happily keeps on executing.

So, what we're going to do is when we match that number, the number where we equal, our counter equals 5000, we're going to go ahead and change that variable, the 'notFound' variable. At that point, we have, in fact, found it. So, this is going to be 'false;'. So, go ahead and save this file. And so, now I'm going to go ahead and reload the file. I killed the previous window because it was taking up too many resources. So, I'm going to go ahead and drag and drop this. But before I do, let me go ahead and open up the developer tools. I'm going to go ahead and drag the file in. And as you can see there, it's going forward.

And it hit '5000', and we have another error. But you can see there that it did stop. Now, back in the code, if we studied this closely, you will note that 'notFound' was assigned just like we thought. However, the line that we see here is that 'notFound' is a 'const'. And so, I can't change a 'const' by definition is constant. So, I'm going to go change, go ahead and change that to a 'let'. I'm going to go ahead and go back to the browser. And we're going to go ahead and reload that page. You can see there that that increased to '5000'. It found the '5000', and the execution has now stopped.

Video 18 - Multiplication Tables

To give you an opportunity to test your debugging skills, we're going to ask you to write out the multiplication tables to the Console from 1 to 10. And the output that we're looking for is the one you see on lines 8 through 17. It's important that you match that format and structure because, as you will see, it's a little bit tricky. And that will allow you to try a few different strategies and to debug and walk through your code. So, let's go ahead and get started with some sample code. I'm going to go ahead and save the file, and I will call it tables.

And I will overwrite the one that is there. And then we'll start off by creating here a small comment. And this will be the text that will be visible in the browser window. And I'll just call it here 'Tables Exercise'. And then I will open up a '< script >' tag. And inside of it, we will start off by creating a 'const'. And we'll call that a 'multiplier'. We'll start off with '1;'. Then the next thing that we're going to write is we're going to write a loop. Inside of it, I'm going to start off my counter, which I will call 'i', add '1'. I will place us an ending condition that the loop will execute as long as 'i <= 10;'.

And I will increase the counter by '1' on every loop run. Inside of it, I am simply going to write to the 'console'. The '(i*multiplier);'. Oops, I auto-completed the wrong, the wrong thing there. And now, we can see here are a very, very first attempt at writing some of this out. To overview, I created a constant, which is a multiplier. I then created a counter called 'i'. This will run as long as that is less than 10. And on each iteration, it increases by 1. I then write out to the console the counter times the multiplier. So, let's go ahead and save this file and jump to the browser.

We're starting out with a blank page on the browser. I'm then going to go ahead and load the document that we just created. And as you can see there, we can see Tables Exercise. I'm now going to go ahead and open up the developer tools. And as you can see there, we see the output from that loop, the '1, 2, 3, 4, 5, 6, 7, 8', all the way to '10'. And if we were to change the multiplier, in this case, to '2'. And we reloaded the page. You would see there that then we get the table two. Now, a couple of things to note is we are going vertical as opposed to horizontal, and we're only doing one table at a time.

If we take a look at the output, you can see there that the table is horizontal, and each of the rows are being pulled out, not vertically but horizontally. So, the very first thing to check is, how would we do this horizontally, to print out a table horizontally? And what will we do with the next table that needs to be printed? Well, you could copy this over 10 times. However, what we want you to do is to do it all inside of one for a loop. Now, inside of the loop, you can write anything you like. However, work within that loop. And so, this should be a fun exercise for you to test some of your assumptions and to practice you're debugging skills.

Video 19 - Graphics Animation of Projectiles Exercise

So, let's practice some more JavaScript. And we're going to do some exercises involving some bouncing balls and projectiles. We're going to do some randomwalk, and then we're going to finish off with doing kind of a snaky string of characters that we drag across the screen. But all of these are meant to give you some exercise in JavaScript. And we will try and illustrate different things. And the first one we're going to deal with is this projectile.html, okay? So, download these and take a look at the README because it tells you something about each of these exercises.

Now, the first one, let's drag that onto the browser. So, here's my browser. And I need to drag on this file called projectile. So, I'll drag it on to there, now you'll see it renders a box, and we're going to kind of operate inside this box. Let's settle it. So, let's take a look at the code. We're going to

go into the development environment. So, here we are in the development environment, and we've got projectile, and here's the code and resources. We can go through the code. And the things turned out first that we link in a ' "stylesheet" ' called ' "mystyle.css" '.

And you'll see it here, mystyle.css. So, you need to load that. And it will, this will do it automatically. And here we're loading some JavaScript ' "ball.js" '. And this is the code that will make rules for us, okay? And here's the projectile code. So, you see that we've got some variables here. Let me just show you how it runs. So, we're going to go into the Console, and we're going to create a projectile, I call it 'factory', and we'll make '(1)' projectile. And you'll see that it makes it at a random position. Now, we can call 'update', and this is going to make it move.

So, here now, you'll see a ball moving. And it moves under gravity. So, we're not going to have any drag on. And so if I throw this ball, you can imagine that it's going to maintain a constant velocity in the x-direction because no other forces are acting on it. Of course, in the real world, air resistance would but won't drag, but we won't count that. But gravity acts in the y-direction. So, we have to update the velocities in the y-direction. You'll see that it's moving up in the y-direction, reaches some height, and then starts moving down. So, let's take a look at the code that does that.

You saw that we called it update. Let's go and look at update and see what's happening. So, I am going to write in here. See, let me bring up the correct. So, I'm going to break into the Source. Here's the Sources, and here's 'update'. And you see, we can break-in. Now, let's step through this code and see what's happening. So, here we are in 'update', and we've got, you can see we've got one ball. The length of 'balls'. Now, 'balls' is where we're going to store if correct more than one of these, we're going to store it in this array. Where's that defined?

Well, it's up here. Okay? And this array, we can add new balls to it, but at the moment, we've just got one in there. So, we loop over that one. Well, we're just going to keep the 'velocity' constant in the 'x', but we're going to update the position. So, this means, let's suppose we've got a time step of one that in this increment, the 'x' position will be incremented by the 'velocity'. So, the velocity times the time actually, but we'll assume that time is one. In case, so that updates the 'x' position. Now, what happens in the 'y' direction is that the velocity changes because gravity acts on it.

And the new 'velocity' is equal to the old 'velocity' plus 'gravity' times time. And again, we're going to assume time as one. And so, we update the 'velocity', and then we update the 'y' position. And this is what causes it to go up and then fall back down which, so its y velocity goes to zero and then starts increasing as it falls down, hits the floor. And now, how do we get that it hit the floor? Well, we check the walls. So, we're assuming that we have walls here at 'x', '0', that's the left wall. And '800' right-hand sidewall. And then we check the 'y'.

And in the browser, the y direction is down from the top of the screen. And I think we mentioned that before. So, if the 'y' gets '> 400', so that'll be from the top of the screen, '400' down. We assume that's the ground. And we'll reverse the velocity in the y direction. And at each of the

walls, vertical walls in the x direction, we'll reverse the x velocity. 'velocity_x = -velocity_x' Okay? So that was update. Let's look at factory. Here's factory. We made, we called it with the total of one.

Now, we're going to call it again with a different total. But what it does is it makes a loop over whatever total is and creates new balls by calling 'makeBall'. And 'makeBall' is in this file ball.js. Here it is. Now, we're hiding some of the complexity because this is all, all in here, has to do with the browser. And we'll talk more about that. But for now, we can kind of ignore this. Just note that we're creating the balls by pushing a ball into this array called 'balls'. Okay? And we keeping track as well as the x position.

We're pushing that into the array, another, a different array called 'x' and into a different array called 'y'. And similarly, with 'velocity', we're keeping track of velocity. So, that's a little bit of the complexity, but we are hiding that. And here in 'factory', we just call 'makeBall' So, that does all the work for us. And we've got something called 'getRandom'. So, we're positioning these. 'Math.random' produces a number between 0 and 1, but we're multiplying it by 'scale'. And scale here is '800' for the x direction. So, we'll get a number between 0 and 800.

And here, 'getRandom' again produces a number between 0 and 400. So, let's have fun, and let's go to the Console. Now, let's create a lot more of these balls. Let's create, for example, '100' of them. No, let's just do '(20)'. That's a nice number. Okay, and let's call 'update( )'. Now, it stopped because I got a breakpoint in here. Let's get rid of that breakpoint and continue. Here we go. So, here they are. They're all bouncing along quite happily.

So, the great thing about the computer is once you've got one to work, it's easy to make the others. They just, they do exactly the same. We'll just loop over them and do the same update to every ball. Now, here we're seeing that the balls don't hit in the middle of the air. But this is fairly accurate. And if you remember in school, you know, if you want to throw a ball the furthest, you need to launch it at about 45 degrees.

Air resistance makes it a little different. And if you watch baseball, they use the bounce as well to get it to the catcher. That they don't throw it high up in the air, but they let it hit, often, let it hit the ground in between because that's the fastest way to get the ball over a distance. But this is simple physics and demonstrates a number of things about how to program. Okay, so that's the projectile. Take a look at it and see if you can make some changes. For example, see if you can change the color that if we make 20 balls like this, maybe we have some random colors that we assigned.

Video 20 - Random Walk Exercise

Okay, so let's take a look at the next exercise. We're going to do a randomwalk here. Let me show you what it looks like. So, this code actually looks very similar to the last code. Let me show you what happens first. So, let's bring up the DevTools, and here we got randomwalker. And it has a lot of the same code that we saw before. It has a factory that makes these balls and an update,

but this time our update is slightly different. Let's see how it runs. Let's code it in the Console. So, this time, this particle undergoes what we call a randomwalk. So, it just makes steps in the x and the y direction of a random length, and you see it wanders away from its starting point.

That's one of the properties of a randomwalker. The more steps that it takes, on average, the more, the more the distance that it covers that is a little bit further away from its starting position. So, we get a kind of diffusion. And we'll take a look at how this might apply, say, to a virus like COVID, that if I sneeze, how do these particles move? And it turns out that randomwalk is quite a good physical description of that. So, let's take a look at maybe on code. We got our update. Let's see it's how different to the update that we did with the projectile. So, here we 'update' the 'x' position.

So, here's our 'function update()'. We update the 'x' position by getting a random number, and we pass in the '(stepSize)'. So, we choose the stepSize, and the stepSize here you see is '5;'. And here, we get a random number, and we multiply it by twice the step size. So, this is because I want a number that goes from minus to plus, and 'Math.random' produces a number between 0 and 1. So if I multiply it by here, '2' times the step size, that's 10. Now, I've got a random number between 0 and 10. Now, if I subtract off 'step;', so that's 5. I'll get a random number between -5 and +5. So, that's why I do this.

Okay? It's a trick, if you like, to get a random number between the numbers that I want, and I want between -5 and +5. So, now the 'x' takes a random '(stepSize)', we get speed for a moment. That's going to be an extra, like a sneeze if you like. The wind speed that's going to drag the particles. And the 'y' does the same, just too random. And then we update the positions. And that's it. And here's our timer that holds update again every '100' milliseconds. So, that's a 10th of a second. So, we could go faster. You can make this so much faster, but here we've got a factory that makes particles.

So, we can call that. Here, we called it just to show you one particle, to give you a starter. But we can use factory now to create more. So, let's get to the Console, and let me update this again. And in the Console, let me call 'factory'. And let's, for example, have '(20)' particles, and let's call 'update( )', and that'll stop them all doing randomization. There they go. Most of them are blue particles. Now, let's suppose you sneeze, and we give a drift speed of your sneeze. Let's make it, let's make it '2'. So, now you're sneezing, and these particles are moving over here. So, the question is, how far would they move before gravity takes them to the ground?

I want you to put in gravity to drag them down. It doesn't exist at the moment. Let's start this over again and do our 'factory'. Well, let's set the 'speed' for the rest of the sneeze if you like. So, let's set the speed to '5'. That would be from left to right. And now, let's do 'factory', and we'll create '(20)' of them. There they are. They're all on top of each other. But now, we'll do 'update'. So, here's your sneeze. And it's going along fusing. Now, gravity should take these particles to the ground. I want you to add in gravity.

Okay, so from the last projectile project, you can see how we dealt with gravity. And here, we can add '(100)' particles. There, they are. Now, we'll change 'speed = -10'. You get that fixed. And we'll retrieve our other group. Remember, the other group? Where did they get to? Let's see if they come back. They're probably all spread out by now. Yeah, I think they're all spread out. They're not coming back. Okay, so that's randomwalk. Now, it would be nice if we could track these particles. So, let me start this again. And let's do '(10)' of them.

It would be nice if we could track how they defused. Yeah, so these don't track. They just, these are just 10 particles actually 11 because of the red one. And I didn't have them hit the walls. You can do that if you like. But it'll be nice to see that how what tracks they make. Let's see if we can do that. It turns out that I've given you another one called randomWalk2. So, take a look at randomWalk2, and let's see how that works. I need to, oh, yeah. We can create particles just by clicking. That's quite nice. Let's see if we can make them move as well. So, this is in, it's in the DevTools.

Let's go to the Console. And let's do 'update( )'. Oh yeah! Nice. Now, these are tracking. They're keeping a track of where they were, where they started, and where they're going. So, this green one quite well. Oh! the red one is slow. Oh, let's create some more. Oh! This is good fun. How am I doing this? How do I create particles like this? Let's take a look. Let's see if we can figure out what am I doing there. So, let's look at our Source code. So, this Source code for the randomWalk2. I'm going to cover it up. So, it doesn't distract. Actually, let me, I'm going to stop it because it'll cause my fan to go on if I use too much CPU.

So, here's randomWalk2. So, we see we've got an update. Well, it looks very much the same as before, but it's not quite the same. We'll come back. Here's our 'factory' that's pretty much the same. But here's something new. We've got a function called 'mouse'. And here, we've added something called an 'EventListener'. And we've said if ' "mousedown" ' call this function ' 'mouse' ', and what ' 'mouse' ' does is it gives you the position of where you clicked. So, we're catching the position of that mouse click.

Now, we're making a ball at that position with a 'randomColor'. Let's see that how I'm doing 'randomColor' is up here. Maybe you can change this so you can make more. colors. I'm only making four different colors. Okay? So, this gives, allows us to click and make new particles. That's pretty neat. So, we can let these go. Let's go into the editor, actually. And I'm going to change in randomWalk2. Let me change where this 'setTimeout', let's call 'update' again. So, this is how it's stepping in time. Let's call it faster.

So, let's go and do that. I'm going to redo this and let's put a few in here. Let's get a nice painting if we can. Let's see how that goes. Okay. And so, I need to bring our. Now, call 'update', yeah. Yep. So, we do 'update( )'. And that'll get us time-stepping. There we go. Beautiful. Beautiful. So, see if you can add some more colors to this. Let me fill in a few pieces here that's not getting much action. Here, I am going to get it all covered in. Let's see. Yeah, all pretty good. Let's see what kind of painting you can do with randomwalk.

Video 21 - Big Bang Exercise

So, this is an example called Big Bang. So, I'm going to drag the file from my file system, bigBang.html, and let's bring up the development environment. Okay, so here's that. And in this case, we're going to make, in our factory, we're going to make two balls, but going in equal and opposite directions. So, we're setting one with velocity, 'velx, vely,' and the other one with '-velx, -vely,' So, let's start this. 'gravity = 0'. 'factory' will make to '(100)', so we're going to make them in pairs. So, there they are, all in the same position, and now let's do 'update( )'. So, there they go. So, you might ask yourself, why are we getting these patterns when I'm giving these random velocities? The clue is that the velocities are integer values, not floating-point. But it seems like, with Big Bang, we're having the Big Bang; the universe explodes. But then, at some stage, it all comes back together again.