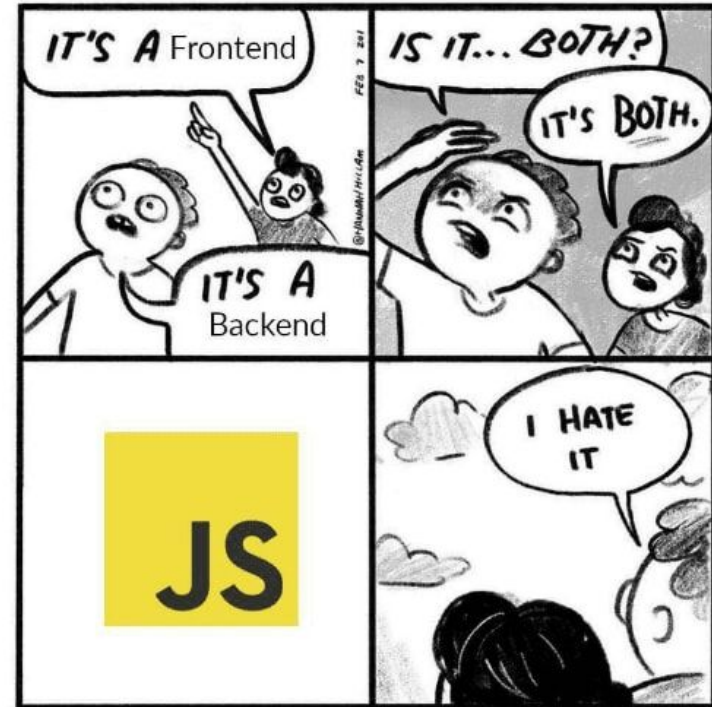




JavaScript Intermediate

Structure of the lecture

- Random number generation
- Control Statements: IF/ELSE
Conditionals & Logic
- Comparators and Equality
- Collections: JavaScript Arrays
- Control Statements: While
Loop
- Control Statements: For Loop



It's a Meteor !

Random Number Generation

`Math.random()` - is a function that generates a random number.

The random number that gets generated is a **16 decimal place number**, and it can be any number **between 0 and 0.9999999**.

So essentially it **never reaches 1**. And the number that you get out, so in this case **n**, will be different every single time you run the code, but it will always be **between 0 and 0.9** to 16 decimal places, and it never ever reaches 1.

```
var n = Math.random();
```

0.3647382746318429

16 Decimal Places

Example

Let's say if we were trying to simulate a dice roll, so we have 6 possibilities. When we roll a dice we get any number between 1 and 6, right? So how do we take this random number that we get from **Math.random()** and turn it into a number between 1 and 6? Well, we could first multiply it by **6**, so we get **2.188**.

Next we take that number and we perform **Math.floor()**.

So from our functions knowledge, we know that **n** in this case is being used as an input to this function floor, which essentially rounds it down to the nearest whole number. In this case it would be **2**.

```
var n = Math.random();
```

0.3647382746318429

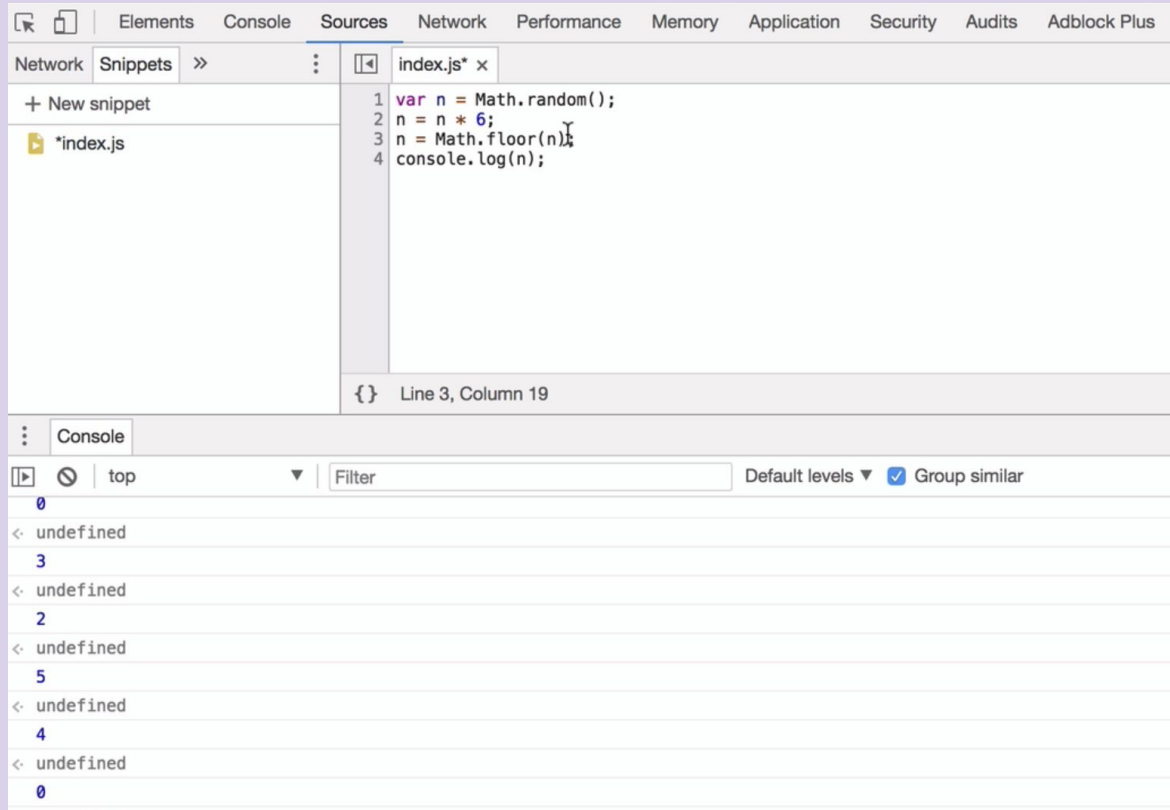
```
n = n * 6;
```

2.18842964779

```
n = Math.floor(n);
```

2

Try it by yourself



Let's create Love Calculator

```
prompt("What's your name?");
```

we are asking the name of user;

```
prompt("What's your lover name?");
```

we are asking the name of user's love;

```
var loveScore = Math.random() * 100;
```

we create a variable **loveScore** and assign a randomizer to it multiplied by 100(because we need numbers from 0 to 99)

```
loveScore = Math.floor(loveScore) + 1;
```

we assign numbers rounder through `math.floor` to our variable and add 1(because we want 100 to be included in results of love score)

```
alert("Your love score is " + loveScore);
```

we show the message and result of love score

Control Statements: IF/ELSE Conditionals & Logic

```
if (condition === right) {  
    instruction;  
} else {  
    instruction;  
}
```


Let's use if/else statement on our Love Calculator

```
prompt("What's your name?");
```

```
prompt("What's your lover name?");
```

```
var loveScore = Math.random() *100;
```

```
loveScore = Math.floor(loveScore) + 1;
```

```
if (loveScore > 75) {
```

```
  alert("Your love score is " + loveScore + ". You love each other purely.");
```

```
} else {
```

```
  alert("Your love score is " + loveScore + ". Maybe you have to find other one.");
```

```
}
```

Comparators and Equality

=== Is equal to

!== Is not equal to

> Is greater than

< Is lesser than

>= Is greater or equal to

<= Is lesser or equal to

Comparators and Equality

```
var a = 1;  
var b = "1";  
typeof(a); - number  
typeof(b); - string
```

```
if (a === b) {  
  console.log("yes");  
}else{  
  console.log("no");  
}  
Answer: no
```

```
if (a == b) {  
  console.log("yes");  
}else{  
  console.log("no");  
}  
Answer: yes
```

Instead of using a **three equal sign** sometimes you see people using **two equal signs**. There's a **big difference** however between the two of these. Let's say we have a variable **a** and it's equal to **1** and then we have a variable **b** that's equal to the **string 1**.

This is 1 as a piece of text essentially, and they have different data types, because if we say type of a we get number and type of b we get string.

So at this point if we check to see if a triple equals b, then if it's true we will get console.log "yes" and if it's false then we will get console.log "no". So if I go ahead and hit enter right now then you can see that we get NO, a does not triple equal b. But, however, if I was to modify this and change it to if a is a double equal to b and I hit run, then you can see we get YES in this case.

So the **important difference** between three equal signs and two equal signs is that, even though **they both check for equality**, with **three equal signs it's also checking to see that the data types are matching**, whereas with **two equal signs it doesn't care** about the data types.

Combining Comparators

& & AND
| | OR
! NOT

AND command (&&) works whether if condition one is true and condition two is true.

You can check to see if either condition one is true or condition two is true by using the OR command, and you can do this using two pipe signs (||), and that's the straight lines that you see on your keyboard.

And finally the exclamation mark (!) means NOT, or the opposite of something.


Let's modify our Love Calculator

```
prompt("What's your name?");  
prompt("What's your lover name?");  
var loveScore = Math.random() *100;  
loveScore = Math.floor(loveScore) + 1;
```

```
if (loveScore > 75) {  
  alert("Your love score is " + loveScore + ". You love each other purely.");  
} else if (loveScore >30 && <=75) {  
  alert("Your love score is " + loveScore);  
} else {  
  alert("Your love score is " + loveScore + ". You go together like water and oil.");  
}
```

JavaScript Arrays

Arrays are generally described as "list-like objects"; they are basically single objects that contain multiple values stored in a list. Array objects can be stored in variables and dealt with in much the same way as any other type of value, the difference being that we can access each value inside the list individually, and do super useful and efficient things with the list, like loop through it and do the same thing to every value.

```
var eggs = [  ,  ,  ,  ,  ]
```

JavaScript Arrays

You can find out the length of an array (how many items are in it) in exactly the same way as you find out the length (in characters) of a string — by using the [length](#) property. Try the following:

```
const shopping = ['bread', 'milk', 'cheese', 'hummus', 'noodles'];  
console.log(shopping.length); // 5
```

Items in an array are numbered, starting from zero. This number is called the item's *index*. So the first item has index 0, the second has index 1, and so on. You can access individual items in the array using bracket notation and supplying the item's index, in the same way that you [accessed the letters in a string](#).

```
const shopping = ['bread', 'milk', 'cheese', 'hummus', 'noodles'];  
console.log(shopping[0]);  
// returns "bread"
```

JavaScript Arrays

This is fine if you know the index of an item, but what if you don't? You can find the index of a particular item using the [indexOf\(\)](#) method. This takes an item as an argument and returns the index, or `-1` if the item was not found in the array:

```
const birds = ['Parrot', 'Falcon', 'Owl'];
console.log(birds.indexOf('Owl'));    // 2
console.log(birds.indexOf('Rabbit')) // -1
```

To add one or more items to the end of an array we can use [push\(\)](#). Note that you need to include one or more items that you want to add to the end of your array.

```
const myArray = ['Manchester', 'Liverpool'];
myArray.push('Cardiff');
console.log(myArray);    // [ "Manchester", "Liverpool", "Cardiff" ]
myArray.push('Bradford', 'Brighton');
console.log(myArray);    // [ "Manchester", "Liverpool", "Cardiff", "Bradford", "Brighton" ]
```

To remove the last item from the array, use [pop\(\)](#).

```
const myArray = ['Manchester', 'Liverpool'];
myArray.pop();
console.log(myArray);    // [ "Manchester" ]
```


Control Statements: While Loop

The **while statement** creates a loop that executes a specified statement as long as the test condition evaluates to true. The condition is evaluated before executing the statement.

```
while (something is true) {  
    //Do something  
}
```

Example of using While Loop

The following `while` loop iterates as long as `n` is less than three.

Each iteration, the loop increments `n` and adds it to `x`. Therefore, `x` and `n` take on the following values:

- After the first pass: `n` = 1 and `x` = 1
- After the second pass: `n` = 2 and `x` = 3
- After the third pass: `n` = 3 and `x` = 6

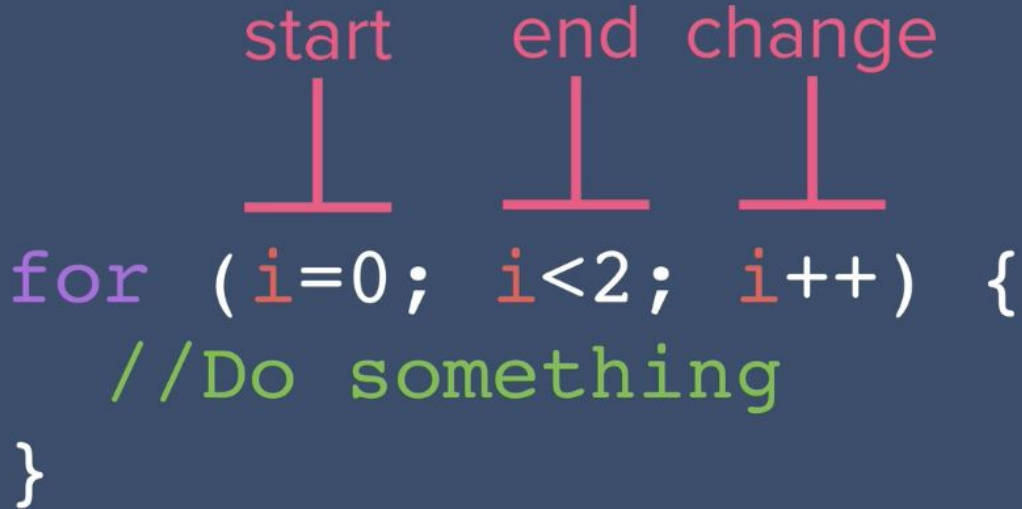
After completing the third pass, the condition `n < 3` is no longer true, so the loop terminates.

```
var n = 0;
var x = 0;

while (n < 3) {
    n++;
    x += n;
}
```

Control Statements: For Loop

The **for statement** creates a loop that consists of three optional expressions, enclosed in parentheses and separated by semicolons, followed by a statement (usually a [block statement](#)) to be executed in the loop.



The diagram illustrates the structure of a for loop. Above the loop's parentheses, three labels are positioned: 'start' above 'i=0', 'end' above 'i<2', and 'change' above 'i++'. Each label is connected to its corresponding expression by a vertical line. The loop itself is written as 'for (i=0; i<2; i++) { //Do something }'. The labels 'start', 'end', and 'change' are in pink, while the code 'for (i=0; i<2; i++) { //Do something }' is in a monospace font with 'for' in purple, the loop body in green, and the closing brace in white.

```
      start      end      change  
      |          |          |  
for (i=0; i<2; i++) {  
    //Do something  
}
```

Example of using While Loop

The following `for` statement starts by declaring the variable `i` and initializing it to `0`. It checks that `i` is less than nine, performs the two succeeding statements, and increments `i` by 1 after each pass through the loop.

```
for (let i = 0; i < 9; i++) {  
    console.log(i);  
    // more statements  
}
```

Useful links

- [Randomness](#) -video by Khan Academy
- [Why can't programmers program?](#) A blog post from Coding Horror
- [Now that's what I call a Hacker.](#) The legendary story retold in English.
- [The original story](#) from above link in Russian.
- [JavaScript Arrays](#)
- [While Loops](#)
- [For Loops](#)