INTRODUCTION TO JAVASCRIPT, CONDITIONALS, LOOPS, DATA TYPES

Full disclosure, we're covering a lot of ground today...

DATA TYPES

First, some review~

A **data type** is a way of classifying or identifying different types of data

With data types, we can determine:

- possible values for that type
- operations that can be performed on that type
- the meaning of the data
- the way values of that type can be stored

Most programming languages have the same **data types** (there's always a caveat)

- **strings**: word or sentence surrounded by "quotes" -- 'kittens are soft'
- **integers**: any number without a decimel point -- 7
- **floats**: any number with a decimel point -- 1.03
- **booleans**: true or false
- **arrays**: collections of data -- []
- **associative array**: collection of data with key-value pairs -- ['a': 200, 'b': 300]
- **objects**: a representation of something

Array:

```
[ 'dogs', 'cats', 'turtles', 'birds' ]
```

Object:

```
let fakeObject = {
 color: "blue",
 fabric: "cotton",
 size: "M"
};
```

In JavaScript, there are five **primitive data types**

- **string**
- **number**
- **boolean**
- **undefined**
- **null**

Everything else is an **object**

- **arrays**, [ ]
- **functions**,
- **objects**, {},

```
function nameOfFunctionWeInvented() {
 // do cool things
}
```

```
{
 name : 'Justin Elm'
}
```

There's also a core set of language elements:

- operators: +, -, ===
- control structures
- statements

```
if ( somethingIsTrue ) {
 // do something else
}
```

```
let x = 1;
```

With that brief intro/review, let's focus on **conditionals** before later moving into **arrays**.

Before we begin, let's take a minute to ensure everyone has an account on repl.it

CONDITIONALS

So, what do **conditionals** involve (at least for today)?

- **if/else statements** and **conditionals**
- **Boolean logic** to combine and manipulate conditional tests
- **for, forEach, while, do while**

Conditional statements allow us to decide which bit of code to **execute** and which to skip based on the results of whatever **condition** we stated.

A **condition** is sort of like a test

JavaScript makes use of two conditional statements: **if/else** and **switch**

if/else statements are dependent on **boolean logic**

Anyony remember what **boolean logic** is?

The block of code within the body of the **if statement**, { }, executes if the **boolean logic** evaluates to true

```
if ( boolean logic ) {
 // run this code if 'boolean logic',
 // as a parameter, evaluates to true
}
```

An actual **if statement**

```
if ( 1 > 0 ) {
 console.log( "The number 1 is greater than 0" );
}
```

Take a second, try that in repl.it

That's very useful, but also kind of limiting no?

Why?

What if the **boolean logic** evaluates to **false**?

Where does the code go? What's the next step?

Else statement!:

Here's an example:

```
if ( boolean logic ) {
 // run this code if 'boolean logic',
 // as a parameter, evaluates to true
} else {
 // evaluate this code if
 // 'boolean logic' evaluates to false
}
```

Here's another example:

```
let number = 7;

if ( number > 5 ) {
 console.log("The variable number is greater than 5");
} else {
 console.log("The variable number is less than 5");
}
```

Take another second, try that in repl.it as well, and change the value of **number** to **console.log** both strings

But that's only two options, and in the real world, we'll want more.

So we can expand to **else if statements**

else if statements can test more than one criteria

Note, **JavaScript** will stop checking **conditionals** once it hits one that evaluates to **true**

An example:

```
let name = "puppies";

if ( name === "kittens" ) {
 name += "!";
 console.log(name);
} else if ( name === "puppies" ) {
 name += "!!";
 console.log(name);
} else {
 name = "!" + name;
 console.log(name);
}
```

Take a second, do that one in repl.it and play around, see if you can get it to work

A word of caution:

Do NOT assign values within a **conditional statement**

```
if ( x = "puppies" ) {
 console.log("False!");
}
```

Try that one in repl.it too, see what it says

Something you may encounter when Googling are **ternary operators**

In short, it's a concise **if/else statement**

Ternary:

```
( expression ) ? /* true value */ : /* false value */ ;

( 1 > 0 ) ? console.log( 'true' ) : console.log( 'false' );
```

So, looks different, probably faster to type, works the same

I'm sure some would argue **ternary operators** are best practice, but I'm not too worried about it

We're essentially doing this:

```
if ( expression ) {
 /* true value */ ;
} else {
 /* false value */ ;
}
```

So an example:

```
let age = 30;
let minAgeToVote = 18;
let allowedToVote = ( age > minAgeToVote ) ? "yes" : "no";
console.log( allowedToVote );
```

Take a crack at that one in repl.it

Alright, how we feeling? Questions?

Alright, let's talk about **comparison operators**.

We can make comparisons using **equality comparison operators** and **relational operators**

Comparison:

```
==, !=, ===, !==
```

Relational:

```
>, <, >=, <=
```

Equality operator:

The double equals, ==

Note, JavaScript will perform something called **type conversion** in the background if the **operands** are different **types** to check if they're equal

```
"dog" == "dog";
"dog" == "cat";
"1" == "1";
1 == "2";
```

You shouldn't rely on **type conversion** though.

Try some of those in repl.it

Just to reiterate, **numbers** and **strings** that contain **numbers** of the same value will be considered equal.

Identity operator: also refered to as the **strict equality** operator. It compares *both* type and values

```
1 === "1";
```

Try that in repl.it, what's it return?

So, no **type conversion**. We should ALWAYS use this because it's the most precise.

We can also compare **objects**. So, any **object** (like an **array**) is only equal to itself.

Objects are compared by **reference**, not **value**

```
[] === [];
// => false

let a = [];
a === [];
// => false

a === a;
// => true
```

Again, to be explicit, **primitives** are compared by **value** whereas **objects** are compared via where they're stored in **memory**

Moving on to **inequality operators**, we have **!=**, and **!==** where the latter is the strict equality operator

The **inequality operator** returns true if opperands are not equal. And just like before, if the two operands are not of the same type JavaScript will try and perform **type conversion**

```
1 != "1"
// => false

1 !== "1"
// => true

1 !== 2
// => true
```

There are also **logical operators**, of which we've been using without defining.

&& - means "and"

|| - means "or"

The **&&** operator requires both values to be **true** to **return true**, otherwise it will **return false**

```
true && true
// => true

true && false
// => false

false && false
// => false
```

Okay, that's a lot of information. Let's try and practically apply this via checking a password.

Try this in repl.it:

```
let network = "SVA-Guest";
let pw = "Paintbrush";

if ( (network === "SVA-Guest") && (pw === "paintbrush") ) {
 console.log( "Wifi Access Granted" );
} else {
 console.log( "Wifi Access Denied" );
}
```

What will the console show?

The || operator only requires *either* of the values to be **true** to **return true**, other it **returns false**

```
true || false
// => true

false || true
// => true

false || false
// => false
```

The || operator with an **if statement**:

```
let day = "Monday";
if ( (day === "Monday") || (day === "Wednesday") ) {
 console.log( "We have class!" );
}
```

The || operator can often be used for **default** values, since only one value needs to be **true**

```
// our saySomething() function takes an
// argument called 'message'
function saySomething(message) {
 let loggedMessage = message || "Hello World!";
 console.log( loggedMessage );
}
// but what happens if you invoke the saySomething()
// function without passing an argument?
saySomething();
```

Try that in repl.it

With all of this in mind, let's do an eligibility exercise:

Using repl.it, write a program that outputs a message based on a user's age. Feel free to work in pairs.

The program must **console.log** *only* the most recent item a person can do. For example, if a user's age is 46, the message should **console.log** "You can run for president!"

Stipulations:

- Under 16: 'You can go to school!'
- 16 or older: 'You can drive!'
- 18 or older: 'You can vote!'
- 21 or older: 'You can (legally) drink alcohol!'
- 25 or older: 'You can rent a car!'
- 35 or older: 'You can run for president!'
- 62 or older: 'You can collect social security!'

You can hardcode the age as a variable to test your code.

Does that make sense? Questions? Do we need more repl.it time?

ARRAYS

The purpose of an **array**?

We store collections of data in an array, which is great for enumerating or recording data.

Each item in an array is called an **element**, and each element has an **index**.

The **index** always starts at 0

```
let names = ['Jerry', 'George', 'Elaine', 'Kramer'];
```

We can target each **element** in the array via its **index**:

```
let first = names[0];
names;
```

What does **names[2]** return?

We can assign **values** to an array via the **index**

```
let names = ['Jerry', 'George', 'Elaine', 'Kramer'];
names[0] = 'Justin';
```

What will **names[0]** return? And what does the array contain now?

```
names;
=> ['Justin', 'George', 'Elaine', 'Kramer']
```

We can also find the number of **elements** in an **array** using the **length** property

```
let names = ['Jerry', 'George', 'Elaine', 'Kramer'];
names.length
=> 4
```

The **length** property will always give us a value one digit greater than the last **index**

To say this explicitly, **length** returns the number of items in the **array**, not the **index**

So the **index** of the last element is **length-1**

```
let names = ['Jerry', 'George', 'Elaine', 'Kramer'];
let lastIndexedItem = names.length-1;
lastIndexedItem;
=> 3
```

There are also **types** within **arrays**.

They can contain any type of **element** or **data** in JavaScript, and they can shrink or grow.

```
let sillyArray = [ 'Hello World!', true, undefined,
null, 42, ['Look!', 'a', 'nested Array!'], false ];
```

Side note, this code is *very* poor; you're just making your life difficult.

Keep different **data types** in separate **arrays**

Strings are similar to **arrays** in that we can find the **length** of them the same way we operate on **arrays**

```
let string = "Hello World!";
string.length;
=> 12

string[0];
=> H
```

We can also create arrays

```
let a = new Array();
a[0] = "dog";

a;
=> ["dog"]

let pets = new Array("dog", "cat", "unicorn");
pets;
=> ["dog", "cat", "unicorn"]
```

There are also a bunch of helper **methods**

The toString() method returns a string with each element separated by a comma:

```
array.toString();
```

The join() method returns a string with each element separated by a **parameter**:

```
array.join( param );
```

The pop() method returns the last item from the **array**:

```
array.pop();
```

The push() method adds one or more items to the end and returns the new **length**:

```
array.push( item1, item2, ..., itemN );
```

We can reverse the **array**:

```
array.reverse();
```

We can remove and return the first item:

```
array.shift();
```

We can add one or more **elements** to the front and return the new length:

```
array.unshift( item1, item2, ..., itemN );
```

An example, create an **array** and add **elements** to it using the **push method** in repl.it

```
let message = [];
message.push(1);

message.push( 'e', 'g', 'a', 's', 's' );
=> 6

message.push( 'e', 'm', 'T', 'E', 'R', 'C', 'E', 'S', 'X' );
=> 15
```

pop(), shift(), unshift()

```
message.pop();
=> 'X'

message.shift();
=> 1

message.unshift( 'duh' );
=> 14
```

Array reversal using reverse()

```
message.reverse();
[ 'S', 'E', 'C', 'R', 'E', 'T', 'm', 'e', 's', 's', 'a', 'g',
```

Turn that **array** into a string

```
message.join(' ');
'S E C R E T m e s s a g e duh'
```

LOOPS AND ITERATING

Okay, let's **loop** back to **loops**, and specifically the **while loop**.

We can use the **while statement** to run a block of code as long as the conditions are **true**. The condition is evaluated *before* executing the code.

```
while ( condition ) {
 // statement
}
```

In a basic sense, loops execute blocks of code a set number of times.

An **infinite loop** is when we don't give the code a stopping point.

That will break your code, and I'm sure you'll all accidentally do it soon enough.

But, to reiterate (pun?), the **loop's** power is in the ability to run the same code over and over and over again.

These are tricky because you can easily get stuck in an infinite loop again:

```
while ( true ) {
 // infinite loop
}

while ( false ) {
 // the loop will never run
}
```

Another example, adding **numbers** to an **array** using a **while loop**.

```
let num = 1;
let numArray = [];
while ( num < 11 ) {
 numArray.push( num );
 num++
}
console.log( numArray );
```

Try that in repl.it

There's a **do-while loop**, which runs a block of code *until* the **condition** is **false**. The **condition** is **evaluated** after executing the **statement** once.

```
let num = 10;
let numArray = [];
do {
 numArray.push( num );
 num -= 1;
} while ( num > 0 );
console.log( numArray );
```

Try that in repl.it too

Let's look at a for loop:

```
let a = [ 1, 2, 3, 4, 5 ];
for ( let i = 0; i < a.length; i++ ) {
        console.log( a[i] );
}
```

We can also cache the array's length, to save some time:

```
let a = [ 1, 2, 3, 4, 5 ];
let arrayLength = a.length;
for ( let i = 0; i < arrayLength; i++ ) {
        console.log( a[i] );
}
```

And one more:

```
let pets = [ "dog", "cat", "turle", "bunny" ];

pets.forEach(
 function( currentValue, index) {
         console.log( "I want a ", currentValue );
         console.log( index );
 }
);
```

```
let departments = ['Fine Art', 'Illustration', 'Cartooning'];

for ( let i = 0; i < departments.length; i++ ) {
 let department = departments[i];
 console.log( department );
}
```

JavaScript **arrays** have several **iterator methods**. Many of the methods require a **function** to be passed in as an **argument**

Each **element** in the **array** has the **statement** in the **function body** applied to it individually.

For example, the forEach() method is a cleaner approach to the previous code:

```
let departments = ['Fine Art', 'Illustration', 'Cartooning'];

departments.forEach( function( department ) {
 console.log( department );
});
```

In the previous example, **'department'** was just an **element**; it was arbitrary

And the **function** is called a callback

In brief, a **callback** is a **function** to execute for each **element**

The **callback** also takes three **arguments**, the element value, the element index, the array being traversed

So, this:

```
departments.forEach( function(department) {
  console.log(department);
});
```

And this:

```
function useThisLater(element, index, array) {
  console.log("element: " + element);
  console.log("index: " + index);
  console.log(" ");
}

departments.forEach( useThisLater );
```

Function the same way.

Similar to how we discussed the difference in **variables** in **ES5** versus **ES6** (var versus let & const), there are are options to how we might write a **function** or **callback** in **ES6**.

We're specifically covering **functions** in all their complexity and nuance next week, but I'll give a quick rundown so you're not confused by documenation.

There are **function declartions**:

```
function myFuncName(param) {
        return param;
}
```

Function expressions:

```
let myFuncName = function(param) {
        return param;
}
```

And now, with ES6, arrow functions:

```
(param1, param2, …, paramN) => { statements }
(param1, param2, …, paramN) => expression
// equivalent to: => { return expression; }

// Parentheses are optional when there's only one parameter na
(singleParam) => { statements }
singleParam => { statements }

// The parameter list for a function with no parameters
// should be written with a pair of parentheses.
() => { statements }
```

In a practical context, this would manifest like this:

```
let materials = ['Hydrogen', 'Helium', 'Lithium', 'Beryllium']

console.log(materials.map(material => material.length));
// expected output: Array [8, 6, 7, 9]
```

Taken from MDN Arrow Functions using the Map method.

A direct comparison:

```
let materials = ['Hydrogen', 'Helium', 'Lithium', 'Beryllium']

materials.map(material => material.length);
```

```
let materials = ['Hydrogen', 'Helium', 'Lithium', 'Beryllium']

materials.map(function(material){
        return material.length;
})
```

Again, we'll go over this more next week, but don't be confused when you read **documentation** and they're using **arrow function callbacks**.

Well, you can be confused but they're doing it on purpose because **arrow functions** are best practice in that context.

So, we just covered a lot of ground and remembering all of the particular **syntax** and names of these **methods** is exceedingly difficult to memorize--which is totally fine and normal.

Because of this, we constantly need to reference **documentation**.

If you recall, many documentation websites are on the syllabus.

In any case, let's take roughly 10 or so minutes to skim over some documention on the Mozilla developer site.

Go here: https://developer.mozilla.org/en-US/docs/Web/JavaScript and track down the documention for:

- .every()
- .some()
- .filter()
- .map()

After you've looked over the **documentation**, open repl.it and create these **arrays**:

```
let evens = [];
evens.push( 2, 4, 6, 8, 10 );

let odds = [];
odds.push( 1, 3, 5, 7, 9 );
```

The **every() method** tests whether **ALL** elements in an array pass the test implemented by the provided **function**

```
let evenResult = evens.every(num => num % 2 === 0);
let allDivisibleByFour = evens.every(num => num % 4 === 0);

console.log("evenResult", evenResult);
console.log("allDivisibleByFour", allDivisibleByFour);
```

The **some() method** tests whether **AN** element in the array passes the test implemented by the provided **function**

```
let someDivisibleByFour = evens.some(num => num % 4 === 0);

console.log("someDivisibleByFour", someDivisibleByFour);
```

The **filter() method** creates **a new array** with all elements that pass the test implented by the provided **function**

Note, this **method** does not mutate the original **array**

```
let bigNums = evens.filter(num => num > 5);
let smallNums = odds.filter(num => num < 5);

console.log("bigNums", bigNums);
console.log("smallNums", smallNums);
```

The **map() method** creates **a new array** with the results of calling a provided **function** on every element in the original **array**

```
let timesFive = evens.map(num => num * 5);
let timesTen = odds.map(num => num * 10);

console.log("timesFive", timesFive);
console.log("timesTen", timesTen);
```

How are we doing so far? Questions? Anyone confused or lost?

Okay, assuming we made it through and have time, let's create a piece of software that will generate quality passwords for us to demonstrate all of these concepts in one demo.