GENERAL REVIEW

Week 5

DATA TYPES

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

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

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

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

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

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

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

```
var 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

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

We also investigated several methods to interact with arrays, including:

```
array.toString();
array.push( item1, item2, ..., itemN );
array.reverse();
array.shift();
array.unshift( item1, item2, ..., itemN );
```

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:

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

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

CONDITIONAL STATEMENTS

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" );
}
```

There are also **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:

```
var number = 7;

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

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:

```
var 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

FUNCTIONS

Before we **call** or **invoke** a **function**, we have to define it.

There are lots of ways to go about this, but the most common are **functions declarations** and **function expressions**.

They both, obviously, use the **function** keyword.

Function declaration:

```
function message( words ) {
        console.log( words );
}
// Note: no semicolon
```

Function expression:

```
var message = function( words ) {
        console.log( words );
}
```

Both are similar, but only **function declarations** allow us to call the **function** *before* it's defined.

In practice:

```
message( 'Hello World!' );

function message( words ) {
        console.log( words );
}
// This won't give us an error
```

Why?

A **function declaration** causes its **identifier** to be **bound** before anything in its code-block is **executed**.

The **function expression** is evaluated in a more typical top-down manner.

```
message( 'Hello World!' );

var message = function ( words ) {
        console.log( words );
}
// This will throw an error, try it in repl.it
```

Function declarations have:

- a **name** for the **function** after the **function** keyword
- statements inside the **function** body, which get **executed** every time the **function** is called, are inside curly brackets {}
- an *optional* list of parameters inside parantheses () with multiple **parameters** separated by a comma

Calling, or **invoking**, a **function executes** the code defined inside the **function**

Defining and **calling** a **function** are two different things.

A **function** *is not* called when it's defined.

We can **call** a **function** by using parantheses after its name:

```
function hello() {
        console.log( "Hello World!" );
}
hello();
// note the semicolon
```

With **parameters**, we can make our code more useful:

```
function sayHi( name ) {
        console.log( "Hello " + name );
}

sayHi("Justin");
sayHi("Ronald");
```

Parameters refer to the variables defined in the **function's declaration**. **Arguments** refer to the actual values passed into the **function** when it's called.

```
function fnName( parameter ) {

}

fnName( argument );
```

Parameters from one **function** will never affect **parameters** in another **function** so long as they're not nested. **Parameters** are **local** to each **function**

We can use a comma-separated list to write a **function** with more than one **parameter**. The **parameters** and **arguments** should be ordered the same way.

```
function sum( x, y, z ) {
        console.log( "Sum: " x + y + z );
}

sum( 1, 2, 3 );
```

SOME OTHER THINGS WE'VE MENTIONED...

Tying in an external JavaScript and CSS file:

```
<link rel="stylesheet" href="css/style.css">
<script type="text/javascript" src="js/app.js"></script>
```

Using selectors in JavaScript to target HTML Elements.

Pure Javascript:

```
var theId = document.getElementById('theNameOfTheId');
var theClass = document.getElementsByClassName("theNameOfTheCl
```

We can also change the ID or Class:

```
var theNewId = "newID";
document.getElementById('theNameOfTheId').id = theNewId;

var theNewClass = "newClass";
document.getElementsByClassName("theNameOfTheClass").className
```

The same thing, but using jQuery:

```
<script src="https://code.jquery.com/jquery-3.3.1.min.js"></sc
var theIdVariable = $("#theId");

var theClassVariable = $(".theClass");
```

And updating those values:

```
<script src="https://code.jquery.com/jquery-3.3.1.min.js"></sc
$("#targetId").attr("id", "theNewId");
$(".targetClass").attr("class", "theNewClass");

// More class methods...
$(".targetClass").addClass("theNewClass");
$(".targetClass").removeClass("theOldClass");
$(".targetClass").toggleClass("theToggleClass");
```

And finally, event listeners (vanilla JS):

```
var button = document.getElementById('myButton');
button.onclick = function() {
        // do stuff
}

button.addEventListener("click", function(){
        button.disabled = "true";
});
```

Same thing, but using jQuery:

```
var button = $("#myButton");
button.click(function(){
        // do stuff
});

button.click(function(){
        executeCustomFunction();
});
```

Alright, this should be sufficient to complete the dice game.