CONDITIONALS, FUNCTIONS, SCOPE

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**

for, forEach, while, do while are also **loops**, which we'll expand on next week.

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, 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, try an eligibility exercise:

Using repl.it, write a program that outputs a message based on a user's age.

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.

Don't forget, once **JavaScript** evaluates one of these **expressions** as **true**, it will stop.

As usual, there are other ways of going about this. Specifically, a **switch statement**

A **switch statement** first evaluates the **expression** and then matches the **expression's** value to a **case** clause. If there's a **match**, it **executes** the **statements** for that **clause**.

We also have to use a **break** to stop it from continuing to **evaluate** statements if there's a match. There's also an option for default.

```
switch ( expression ) {
 case valueOne:
  // statements
  break;
  ...
 case valueN:
  // statements
  break;
 default:
  // statements
  break;
}
```

An actual switch statement, try this in repl.it:

```
let num = 1;

switch ( num ) {
 case "1":
        console.log("You entered the string '1'");

 case valueTwo:
        console.log("You entered the number 1");

 default:
        console.log("You did not enter 1");
}
```

What happened?

Try this one, what's the difference?

```
let num = 1;

switch ( num ) {
 case "1":
        console.log("You entered the string '1'");
        break;
 case valueTwo:
        console.log("You entered the number 1");
        break;
 default:
        console.log("You did not enter 1");
}
```

If we're comparing against specific values in an **if/else statement**, we can almost always refactor to cleaner code using a **switch statement**.

Using repl.it, refactor the following code to use a **switch statement**:

```
let grade = 'B';
if ( grade === 'A' ) {
 console.log('Awesome job');
} else if ( grade === 'B' ) {
 console.log('Good job');
} else if ( grade === 'C' ) {
 console.log('Okay job');
} else if ( grade === 'D' ) {
 console.log( 'Not so good job' );
} else if ( grade === 'F' ) {
 console.log('Poor job');
} else {
 console.log('Unexpected grade value entered');
}
```

ANSWER...

```
let grade = 'B';
switch ( grade ) {
        case 'A':
                console.log('Awesome job'); break;
        case 'B':
                console.log('Good job'); break;
        case 'C':
                console.log('Okay job'); break;
        case 'D':
                console.log('Not so good job'); break;
        case 'F':
                console.log('Poor job'); break;
        default:
                console.log('Unexpected grade value entered');
}
```

And what happens if you take the **break;** statement out?

```
// Good job
// Okay job
// Not so good job
// Poor job
// Unexpected grade value entered
```

There's a technique, similar to || in **if/else statements**. For example, what if we only cared about whether or not the student passed?

```
let grade = 'B';
switch ( grade ) {
        case 'A':
        case 'B':
        case 'C':
        case 'D':
                console.log('You passed!');
                break;
        case 'F':
                console.log('You failed!');
                break;
        default:
                console.log('Unexpected grade value entered');
}
```

FUNCTIONS AND SCOPE

What's a **function**?

We've used these before, in a limited manner, but what's actually going on?

A **function** is a reusable statement, a group of reusable statements, that can be called later or anywhere in a program. Note! As long as you have access to it...

What's the point? It helps us avoid the need to re-write the same statement over and over and over again.

Functions help us tame our code. We can divide large unwieldy pieces of code into smaller, more manageable, pieces.

This is related to the principle of **DRY** programming--
Don't Repeat Yourself

We want to write as few lines of code as possible. Work smart, not hard.

Here it gets a bit complicated, but I promise we'll go over it.

In JavaScript, every function:

- is an instance of the **object** data type
- can have **properties**
- has a link to its **constructor method**
- can be stored in a **variable**
- can be **returned** from another **function**
- can be passed into another **function** as an **argument**

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**, as noted before.

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

Function declaration:

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

Function expression:

```
let 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!' );

let 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
```

JavaScript functions are often defined as **methods** on **objects**. To call these **methods**:

```
const person = {
        speak : function() {
                console.log( "Hello World!" );
        }
}
person.speak();
```

Parameters and Arguments

If a **function** did the same thing every time it was called, that's rather limiting.

We'd have to write a **function** for every new feature or circumstance in order to add new features to our application.

We would have a problem like this:

```
function helloJustin() {
        console.log( "Hello Justin" );
}

function helloRonald() {
        console.log( "Hello Ronald" );
}
```

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 );
```

JavaScript functions don't perform **type checking**, like we described in previous weeks. Also, we can't specify the **type** of a **parameter** when defining the **function**.

So we have to be careful to prevent erros. We'll almost always use the same **type** for the same **parameter** every time we call the **function**.

But, the **parameters** in the **function definition** can be of different types.

Last week, we used a **return statement**.

If we want to update a **variable** using values computed in a **function** or pass it to another **function**, we use a **return statement**.

Using the **return statement** ends the **function's execution** and passes the value we're **returning**.

Some of you have noticed this, by default all functions in **JavaScript return *undefined***.

Even if we don't have the **return** keyword in our **function body**, it will return **undefined**.

We can store the **returned** value in a **variable**

```
function sum( x, y ) {
        return x + y;
}

let z = sum( 3, 4 );
console.log( z );
```

Passing a function into a function:

```
let num = sum( 3, 4 );

function double( x ) {
        return x * 2;
}
// this:
let numDouble = double( num );
// roughly same as:
let numDouble = double( sum( 3, 4) );
```

Try that in repl.it

And just a reminder, the **return** statement will stop the **function's execution**.

```
function speak( words ) {
        return;

        console.log( words );
}
// what will happen?
```

Alright, let's talk about **Scope**

Scope is a concept in programming languages that refers to the current context of **execution**, with context being which values can be referenced.

If a **variable** is *not* in **scope**, then we can't use it because we don't have access to it.

It's as if whatever piece of code we're **executing** doesn't even know it exists.

If we try to use a **variable** we don't have access to, we get an error:

```
function speak( words ) {
        console.log( words );
}
// versus this:
console.log( words );
// try that in repl.it, what happens?
```

Global scope: by default, we're in **global scope**. Anytime a **variable** is declared outside of a **function**, it is part of the **global scope**.

If that's the case, we'd call it a **global variable**.

Global variables are technically bad practice, because it's easier to overwrite the value of a **globally scoped variable**. Any **function** or **expression** on the page can reference a **global variable**.

As mentioned in the first week, when defining **let** and **const**, deal with scope.

let

const

const:

Constants are **block-scoped**, much like **variables** defined using the **let** statement. The value of a constant can't be changed through reassignment, and it can't be redeclared.

let:

let allows you to **declare variables** that are limited to a **scope** of a **block statement**, or **expression** on which it is used, unlike the **var** keyword, which defines a variable **globally**, or **locally** to an entire **function** regardless of **block scope**.

let versus var:

Variables declared by **let** have their **scope** in the **block** for which they are defined, as well as in any contained **sub-blocks**. In this way, **let** works very much like **var**.

The main difference is that the **scope** of a **var** variable is the entire **enclosing function**.

```
function varTest() {
  var x = 1;
  {
    var x = 2;  // same variable!
    console.log(x);  // 2
  }
  console.log(x);  // 2
}

function letTest() {
  let x = 1;
  {
    let x = 2;  // different variable
    console.log(x);  // 2
  }
```

The environment for **global variables** is accessible via the **global object**.

In the browser, this would be the **window object**.

All **global variables** are attached to the **global object**.

```
let message = "Hello!"
console.log("message");

// Using the window object:
console.log( window.message );
```

There's also **namespace**: a **namespace** is a container for a set of **variables** and **objects**, e.g. **functions**.

In terms of best practice, we don't want to pollute the **namespace**.

Later we'll look at how to create **namespaces** to organize our code. It's a way of preventing collision with other **objects** or **variables**.

Local scope: we can create a new **scope** whenever we declare a **function**. Inside the **function** body, we have access to **variables** declared inside that function and in the **outer scope**. Any **variables** declared inside that **function** are local to it.

A **function** inside of a **function** has access to the **outer function's variables**,

```
const globalNumber = 1;
function fn() {
        let localNumber = 2;
        console.log( globalNumber );
        console.log( localNumber );
}
fn();
// what happens if you add this, try it in repl.it:
console.log( localNumber );
```

Local scope example:

```
const a = "This a variable in the global scope.";
function myFunction() {
        let b = "This is in the scope of myFunction.";
        return b;
}
console.log( myFunction() );
console.log( b );
```

Try that in repl.it

A **function** can access **variables** of the **parent scope**. So a **function** defined in the **global scope** can access all **variables** defined in the **global scope**.

```
// Global scope
const prefix = "Hello";

// sayHello is defined in global scope
function sayHello( name ) {
        // prefix was defined in global scope
        console.log( prefix + " " + name);
}

sayHello("JavaScript");
```

If it's a **function declaration**, we can also call it anywhere that has access to **global scope**

Nested function scope example:

```
let a = 1;
function getScore() {
        let b = 2;
        let c = 3;
        function add() {
                return a + b + c;
        }
        return add();
}
getScore();
```

When a **function** is defined inside another **function**, it's possible to access **variables** defined in the **outer function** from the **inner function**.

You should definitely take time to read documentation on functions, they're hard and complicated, it'll take time to get used to.

If you're feeling confident about functions, take some time to read up on **arrow functions**, something we didn't cover in these slides but may use in the future.

Try looking at:

MDN Arrow Functions

In class we'll try and apply all of these concepts for a demo on making a dice rolling website.