

Web Scripting 1

Day 03

Agenda

- Assignment 2 walkthrough
- Conditionals
- Arrays
- Loops
- Assignment 3

Assignment 02

Walkthrough

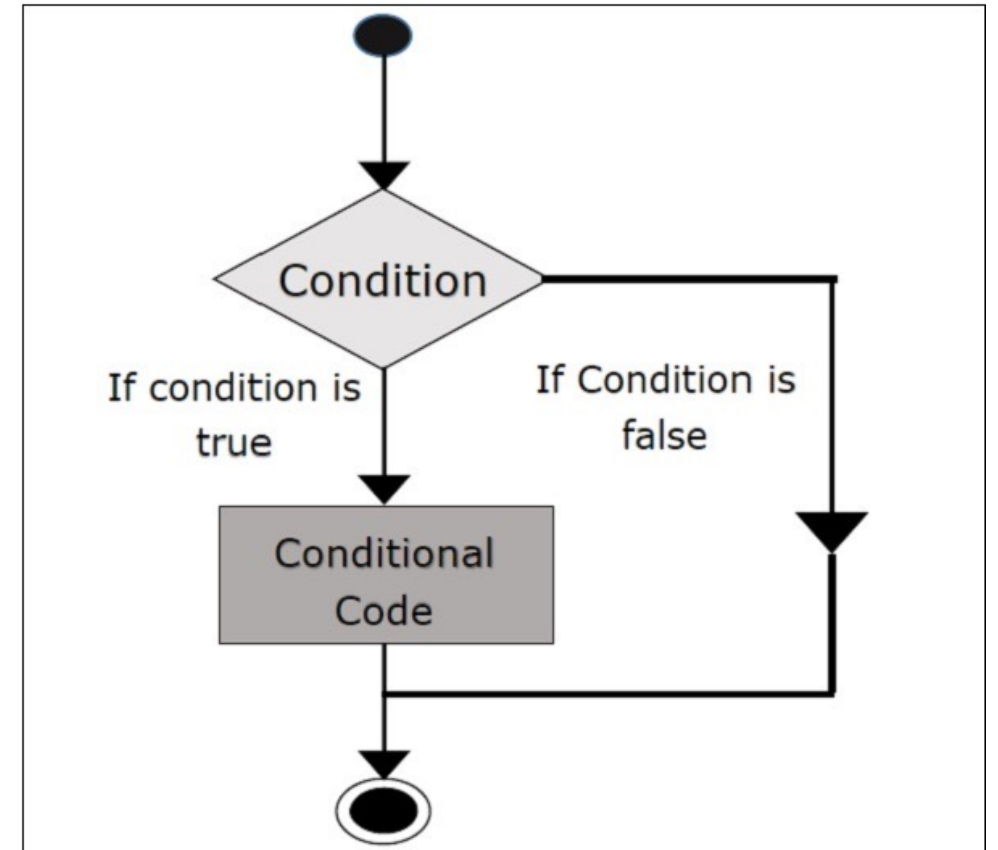
Progress Check-In

- At this time you should have a rough idea of the following concepts
 - What are the three most common JavaScript data type
 - Strings
 - Numbers
 - Booleans
 - A basic understanding of what functions do and where to use them

Conditionals

Conditionals

- Conditionals allow you to direct your script in different directions depending on a conditions
- If a statement is true do one thing, if not, do another thing (or nothing at all)



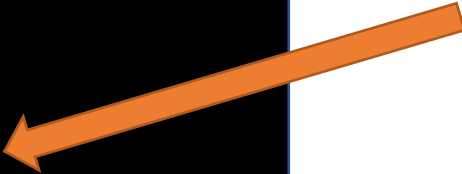
JavaScript Conditionals

- JavaScript has many types of conditional statements
 - "if", "if/else" & "if/else if/else"
 - Switch statements
 - Not covered today
 - Look up "switch" statements on MDN Docs
 - Ternary operator
 - Not covered today
 - Useful for simple conditions
 - Look up "ternary" operator on MDN Docs

JavaScript "if" Statements

- The "if" statement runs code between the "{ ... }", if a statement placed between the "(...conditional statement...)" is true

```
const fruit = 'apple';  
  
if(fruit == 'apple'){  
    console.log('I love apples!');  
}
```



This code will only run "if" the "fruit" variable is equal to "apple"

Why "==" or "===" and Not "="?

- In many computer languages if you wish to ask the question:
 - is "x" equal to "y"? ...
 - ...you would write "=="
 - Note: JavaScript also has "==="...more on that in an upcoming slide...
 - `x == y` -> means "is x equal to y?"
- The single "=" is called an assignment operator
 - `x = y` -> means the variable "x" has been assigned the value of "y"

Watch out for the single "=" in "if" Statements

```
const fruit = 'apple';
```



```
if(fruit = 'orange'){  
  console.log('I love oranges!');  
}
```

*** Note ***: This is bad code!!!

The "I love oranges!" console log will always run, no matter what you set the fruit variable to because in your "if" statement you are using a single "=" assignment operator which assigns the fruit variable the value of "orange". Doing this always results in a "true" statement

Watch out for the single "=" in "if" Statements



```
const fruit = 'apple';
```

```
if(fruit == 'orange'){  
  console.log('I love oranges!');  
}
```

Now we are using the "==" comparison operator, which tells the computer to compare two values.

Since fruit is not equal to orange the code between the "{}" will not run

"==" VS "==="

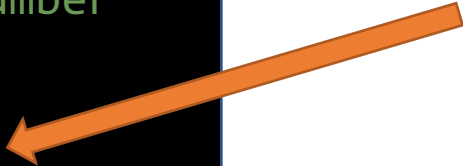
- JavaScript has two different comparison operators
 - "=="
 - Tests only if the value is the same
 - Does **NOT** test the data type
 - Testing string numbers against numbers will result in true
 - "3" == 3 -> this is true when using the "==" operator
 - "==="
 - Strict equivalence
 - Tests the value along with the data type
 - Testing string numbers against numbers will result in false
 - "3" == 3 -> this is false when using the "===" operator since the value is the same but the data type is not the same
 - 3 === 3 -> results in true, since the values (3) and the data type (number) are the same

"==" VS "==="

Testing for value only using "=="

```
const x = '3'; // Data type of string
const y = 3; // Data type of number

if(x == y){
  console.log('x equals y!');
}
```

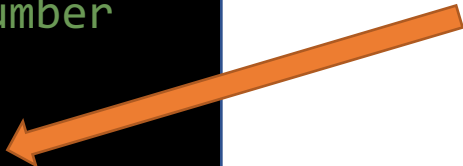


Since we are using the "==" operator the console.log() statement will run because we are just testing the values of the variables and not the data type

Testing for value and data type only using "==="

```
const x = '3'; // Data type of string
const y = 3; // Data type of number

if(x === y){
  console.log('x equals y!');
}
```




Since we are using the "===" operator the console.log() statement will **NOT** run because we are testing the values and the data type

else if

- If your first statement fails, you can test another condition using an "else if"
- You are not limited to a single "else if", you can have as many "else if" statements as you need

```
const a = 5;  
const b = 7;  
  
if(a > 7){  
    console.log('a is greater than b');  
}else if(a < 7){  
    console.log('a is less than b');  
}
```



The "else if" part of this statement will only run if the first "if" condition statement fails and the second "else if" condition statement is true

else

- You can optionally have an "else" that follows your "if" or any number of "else if" statements
- The "else" will run only if all of the "if" and "else if" statements fail
- The "else" part of an "if/else" statement is optional
 - Writing an "if" statement without an "else" is perfectly valid code

```
const animal = 'elephant';  
  
if(animal === 'giraffe'){  
  console.log('The animal is a giraffe');  
}else if(a === 'lion'){  
  console.log('The animal is a lion');  
}else {  
  console.log('The animal is neither a giraffe or a lion');  
}
```



The "else" part of this statement will only run if all of the above "if" and "else if" statements fail

Conditional Statement Operators



Equivalence (value only)



Equivalence (strict – value and data type)



Greater than



Greater than or equal to



Less than



Less than or equal to

Conditional Statement Operators



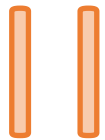
Not



Not equal to (value only)



Not equal to (strict – value and data type)



Or (this is the pipe character - "Shift+\")



And (this is the ampersand character – "Shift+7")

Where are the "|" (pipe) and "&" (and) Characters

The "&" pipe character (You need to press "Shift+7")

The "|" pipe character (You need to press "Shift+\\")



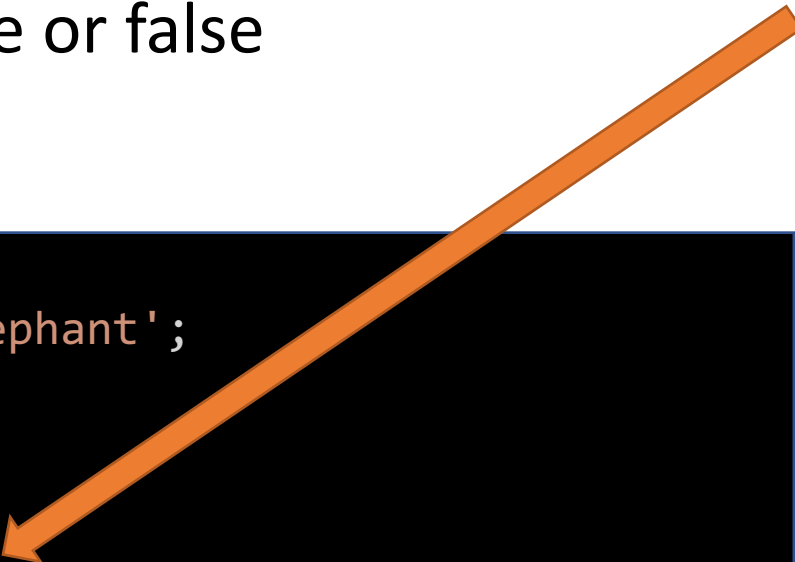
Testing a negative

- The "!" character always represents "not" or not true or false

The console.log() statement will run because we are testing if the "animal" is **NOT** equal to "giraffe" using the "!" character combined with the "==" characters

```
const animal = 'elephant';
```

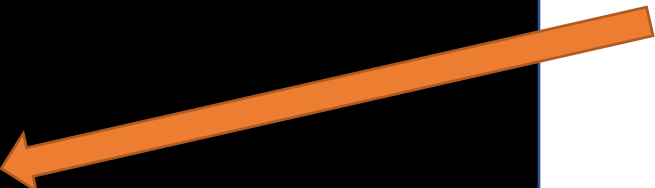
```
if(animal !== 'giraffe'){  
    console.log('The animal is not a giraffe');  
}
```



Testing Multiple Conditions

- You can test multiple conditions using the "||" (or) and/or the "&&" (and) operators

```
const animal = 'cat';  
const animalName = 'Fluffy';  
  
if(animal === 'cat' && animalName === 'Fluffy'){  
  console.log('Hello Fluffy!');  
}
```



The console.log() statement will only run if the "animal" variable is equal to "cat" AND (&&) the "animalName" variable is equal to "Fluffy"

Using && (and)

- Using the && (and) operator means all the conditions must be true
- The JavaScript will not test any further conditions after the first failure in an && (and) statement
 - Use this behaviour to your advantage in your script

```
const animal = 'cat';  
const animalName = 'Fluffy';  
  
if(animal === 'cat' && animalName === 'Fluffy'){  
    console.log('Hello Fluffy!');  
}
```

The console.log() state will only if the "animal" variable is equal to "cat" AND (&&) the "animalName" variable is equal to "Fluffy"


Using && (and)

JavaScript will not test any condition after the first failure in an "&&" (and) condition. The 2nd condition will never be executed if the "username" variable is equal to "null"

Since running the built-in "trim()" method on a "null" value will result in your script crashing, we can test for "null" first and then safely run the "trim" method on our second condition.



```
const username = prompt('What is your name?');
```

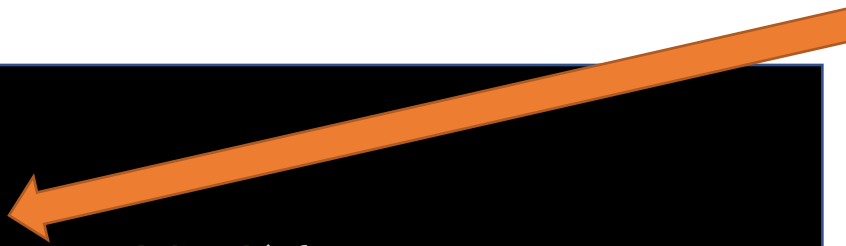


```
if(username !== null && username.trim() !== ''){  
    console.log(`Hello ${username}!`);  
}
```

Using "||" (or)

- With the "||" operator only one of the conditions has to be true
- JavaScript will not test any further conditions in an "or" statement after the first condition that results in true

```
const pet = 'cat';  
  
if(pet === 'cat' || pet === 'dog'){  
  console.log('The pet is either a cat or a dog');  
}
```



Only one condition has to be true in order for our console.log() message to run. In this example the "pet" variable could be equal to "cat" OR (||) "dog" for the console.log() message to run

Combining "||" with "&&"

- Your conditional statements can contain any combination of "||" and "&&" statements
- Just Beware of difficult to read code when making overly complex conditional statements

```
const e = 3;
const f = 5;
const g = 3;
const h = 9;
const i = 12;

if( (e < 5 && e == g) || (h < i) ){
  console.log('Ok!!! : ( ... ');
}
```


Truthy Values in JavaScript

- A truthy value is a value that is considered true by a computer language when encountered in a Boolean context¹ such as in an "if" statement
- Values that are considered truthy by JavaScript¹
 - if(true)
 - if({})
 - if([])
 - if(23) -> Any number other than zero
 - if("0") -> The string number zero is considered true
 - if("any string with at least one character") -> Any string other than an empty string ("")
 - if(new Date())
 - if(-23) -> As above any number (even negative numbers) are considered true other than zero
 - if(Infinity)
 - if(-Infinity)

1. <https://developer.mozilla.org/en-US/docs/Glossary/Truthy>

Falsy Values in JavaScript

- The following values are considered falsy¹ (false in a Boolean context)
 - if(0) -> The number zero is considered false
 - if("") -> An empty string is considered false
 - if(null)
 - if(undefined)
 - if(NaN) -> Not a Number is considered false

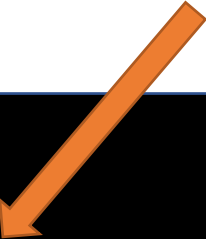
1. <https://developer.mozilla.org/en-US/docs/Glossary/Falsy>

Using Truthy and Falsy Values in JavaScript

- Since truthy values are considered true in the context of an "if" statement you can just write the variable in the condition part of an "if" statement
- We can do a similar thing with falsy values, except that we precede the variable name with a "!" (NOT) character
- Only do the above if you just want to test for truthiness or falseness
 - If you want to test for exactly true, then use "... === true"
 - If you want to test for exactly false, then use "... === false"

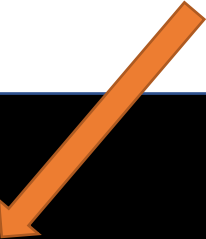
Using Truthy and Values in JavaScript

In this example we test if the "cat" variable has a truthy value. To do this we place just the variable name in the conditional statement



```
cat = 'fluffy';  
  
if(cat){  
    console.log(`Hello ${cat}!`);  
}
```

In this example we test if the "user" variable has a falsy value. To do this we place just the variable name in the conditional statement preceded by the "!" (NOT) character



```
user = null;  
  
if(!user){  
    console.log('The user is not set');  
}
```

Arrays

What is an Array

- An array is:
 - A collection or list of usually related values that are stored in a single variable
 - Example
 - An Array of fruits might store the following values in an array variable with the label of fruit
 - Apples
 - Bananas
 - Oranges

Why we need arrays?

We could store a bunch of related items in individual variables...



```
const actionFigure01 = 'Chewbacca';  
const actionFigure02 = 'Darth Vader';  
const actionFigure03 = 'Palpatine';  
const actionFigure04 = 'Luke';  
const actionFigure05 = 'Yoda';
```

...but for anything larger than a few items this becomes difficult to maintain and is memory inefficient...

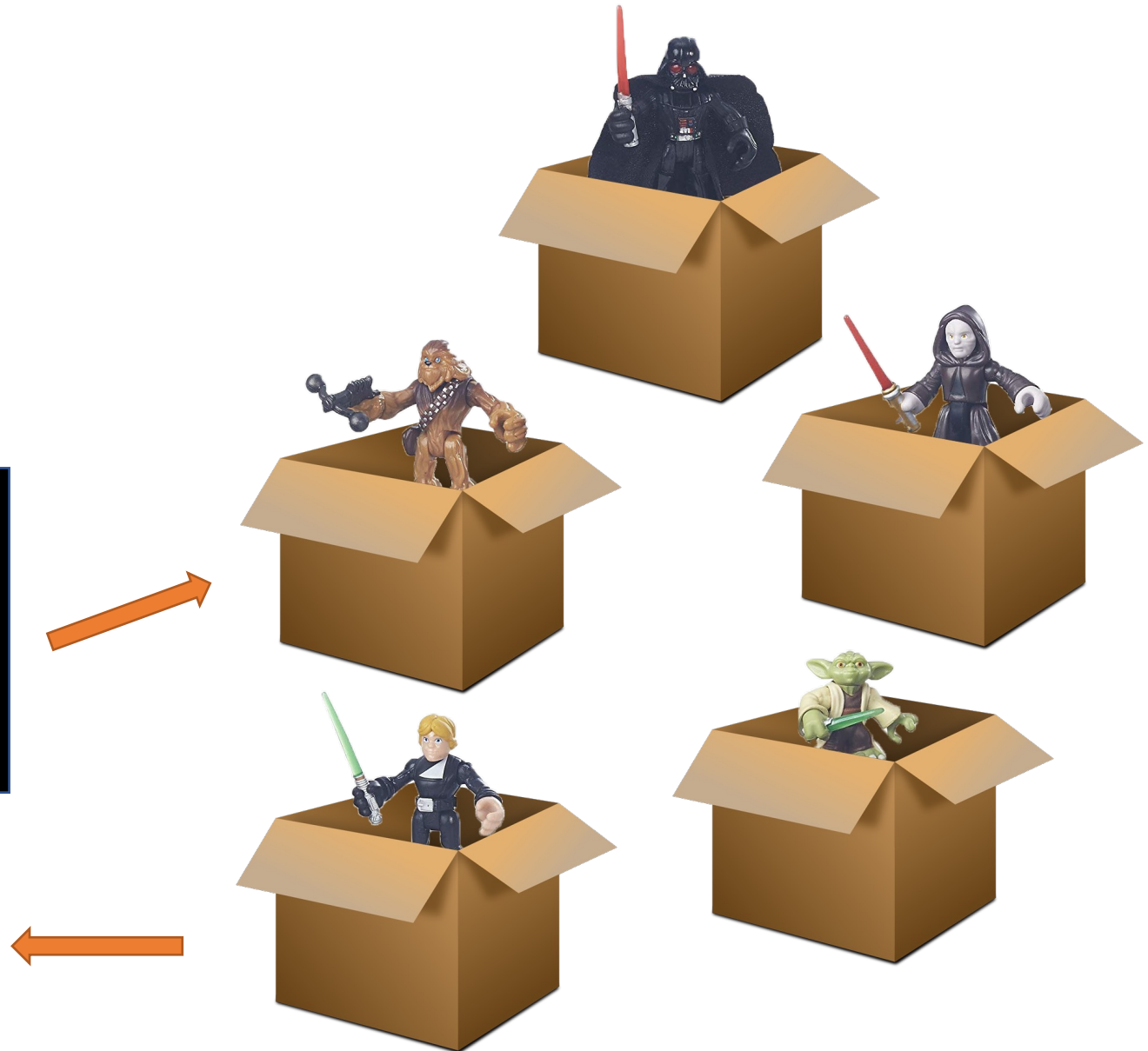


Image Credits:

Box image: <https://pixabay.com/vectors/box-open-cardboard-box-cardboard-152428/>

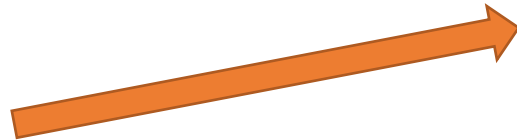
Action Figures: <https://www.amazon.com.au/Galactic-Heroes-Rivals-Action-Figure/dp/B01BMW5QCQ>

Why we need arrays?

...Instead of individual variables for each individual item, we can create a list or an "array", and store all the items in a single variables



```
const actionFigures = [  
  'Chewbacca',  
  'Darth Vader',  
  'Palpatine',  
  'Luke',  
  'Yoda'  
]
```



...having a single variable, means it is much easier to maintain, read and update

JavaScript Arrays

- You do not need to declare a length of an array or a data type when creating an array with JavaScript
 - The length of an array in JavaScript is set dynamically by the number of items in the array
 - JavaScript arrays can have any combination of data types stored in them

```
const myArray = ['foo', 'bar', 23, true];
```


*** Note ***: Although the above code is valid, usually arrays store related values, such as a list of fruits, or a list of provinces or a list of student names

The above code is valid JavaScript

Creating an Array in JavaScript

- You can create an array by calling the array constructor
 - `const myArray = new Array();`
 - Most JavaScript developers do not use this method
- The more popular way to create an array in JavaScript is to do the following:

```
const fruits = [];
```

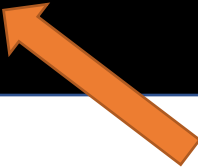


Setting the value of the fruits variable to [] (square brackets) creates an empty array

Adding Values to an Array

- You can add items to an array at the time of the variables initialization

```
const fruits = ['apples', 'bananas', 'oranges'];
```




Array values added to the fruit array at the time of the variable's initialization

Adding Values to an Array

- You can add items to an array using the `myArray[index number]` syntax

```
// Create an array called fruits
const fruits = [];
// Add items to the Fruits Array
fruits[0] = 'Apples';
fruits[1] = 'Bananas';
fruits[2] = 'Oranges';
fruits[7] = 'Blueberries';
```



*** Note ***: There is no requirement to declare array values sequentially.

In this example we create a value for "fruits[7]" without declaring any values for position 3,4,5 or 6. JavaScript will create empty array values for positions 3,4,5 and 6 with the value of "undefined" if you do this

Why Do JavaScript Arrays Start at Zero?

- Good question
- JavaScript borrows some syntax from the C family of languages
- In C an array points to the location in the memory, so in expression *array[n]*, ***n*** should not be treated as an index, but as an **offset** from the array's head¹
- Confused? Or want to know more? Give this short Medium article a quick read
 - [Why Do Arrays Start With Index 0?](https://medium.com/@albertkoz/why-does-array-start-with-index-0-65ffc07cbce8)

1. <https://medium.com/@albertkoz/why-does-array-start-with-index-0-65ffc07cbce8>

Getting the Length of an Array

- Finding out the length of an array is a common task
- The length property will return the length of the array

```
const animals = [];  
animals[0] = 'hedgehog';  
animals[1] = 'tiger';  
animals[2] = 'moose';  
  
// Get the length of the animals array  
console.log( animals.length ) // Outputs -> 3
```



The "animals" array contains 3 items




The "length" property returns the length of an array

Adding an Item to the End of An Array


- Adding an item to the end of an array is a common task
- Below are a couple of ways to do it with JavaScript

```
const carsCompanies = [];  
carsCompanies[0] = 'Buick';  
carsCompanies[1] = 'Toyota';  
carsCompanies[2] = 'BMW';  
// Add item to the end of the  
// carsCompanies array using the  
// push() method  
carsCompanies.push('Kia');
```



Adding an item to the end of an array
using the push() method

```
const carsCompanies = [];  
carsCompanies[0] = 'Buick';  
carsCompanies[1] = 'Toyota';  
carsCompanies[2] = 'BMW';  
// Add item to the end of the  
// carsCompanies array using the  
// length property  
carsCompanies[carsCompanies.length] = 'Kia';
```



Adding an item to the end of an array
using the the "length" property

Array Methods

- Manipulating arrays is a common requirement for many scripts
- JavaScript provides a myriad of built-in Array methods for doing things such as:
 - Iterating over an array – `forEach()`
 - Running a function on each item in the array – `map()`
 - Sorting an array – `sort()`
 - Searching an array – `every()`, `find()`
 - Filtering an array – `filter()`
 - Plus many more...
- If you need to do something to an array try Googling it and see if JavaScript provides a built-in method
 - Example: "find an element in an array JavaScript"

Array Methods

- No need to memorize all the JavaScript array methods
- Just know that they exist and get the syntax when needed
- JavaScript Array Methods Resources:
 - MDN Docs Page on JavaScript Arrays
 - Contains information about arrays plus links to further documentation on all the array methods
 - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array

Loops

What is a Loop

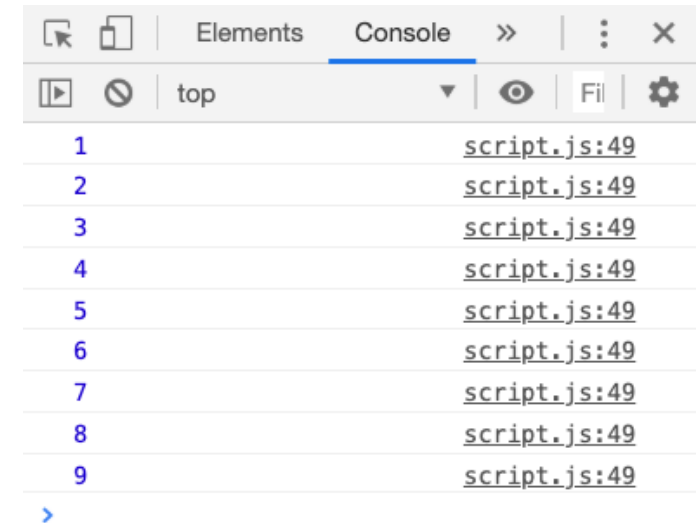
- A loop is a set of instructions that your program will execute repeatedly until a stop condition is met

This is the stop condition. As long as this condition results in "true" the loop will run

```
let counter = 1;  
while(counter < 10){  
  console.log(counter);  
  counter++; // add 1 to the counter  
}
```

This is an example of a "while" loop. This code will run over and over again until the counter is no longer less than 10

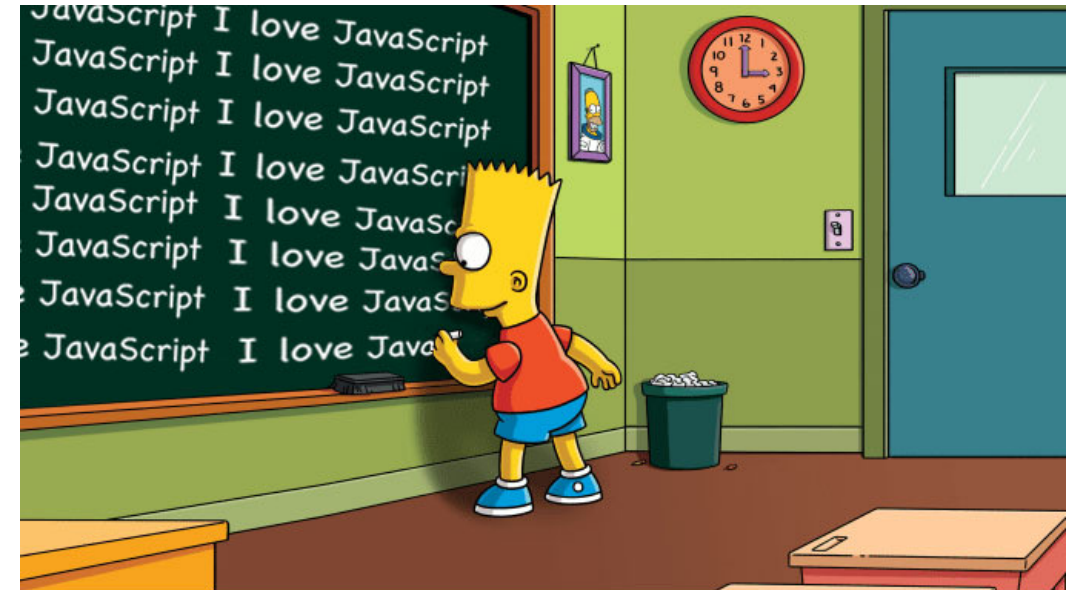
Console Output



1	script.js:49
2	script.js:49
3	script.js:49
4	script.js:49
5	script.js:49
6	script.js:49
7	script.js:49
8	script.js:49
9	script.js:49

Why use Loops?

- Humans don't enjoy doing repetitive tasks repeatedly
- If I asked you to hand in a handwritten piece of paper that said, "I love JavaScript!" 100 times, you would not be happy... : (
- A script does not think or care either way if it executes a line of code once or a 1000 times
- If your code requires repeated code, consider using a loop




Why Use Loops

- Loops are useful for iterating over arrays

Manual Method


```
const students = [  
  'Sally',  
  'Marie',  
  'Joe',  
  'Kate',  
  'Frank'  
];  
  
console.log(students[0]);  
console.log(students[1]);  
console.log(students[2]);  
console.log(students[3]);  
console.log(students[4]);
```

This works, but is tedious to write and error prone. Imagine if we had a 100 students!



Using a forEach() Loop

```
students.forEach(function(student){  
  console.log(student);  
});
```



Since the code we need to run is repetitive we can use a loop.

In this example we use a special type of loop available on arrays called a `forEach()` loop to "iterate" over each item in the array and output each item's value to the console

Beware of Endless Loops

- Be careful when writing your loop code
- Always make sure your loop will eventually stop
- If the browser crashes when running your script, it is often caused by your script running an endless loop



Beware of Endless Loops

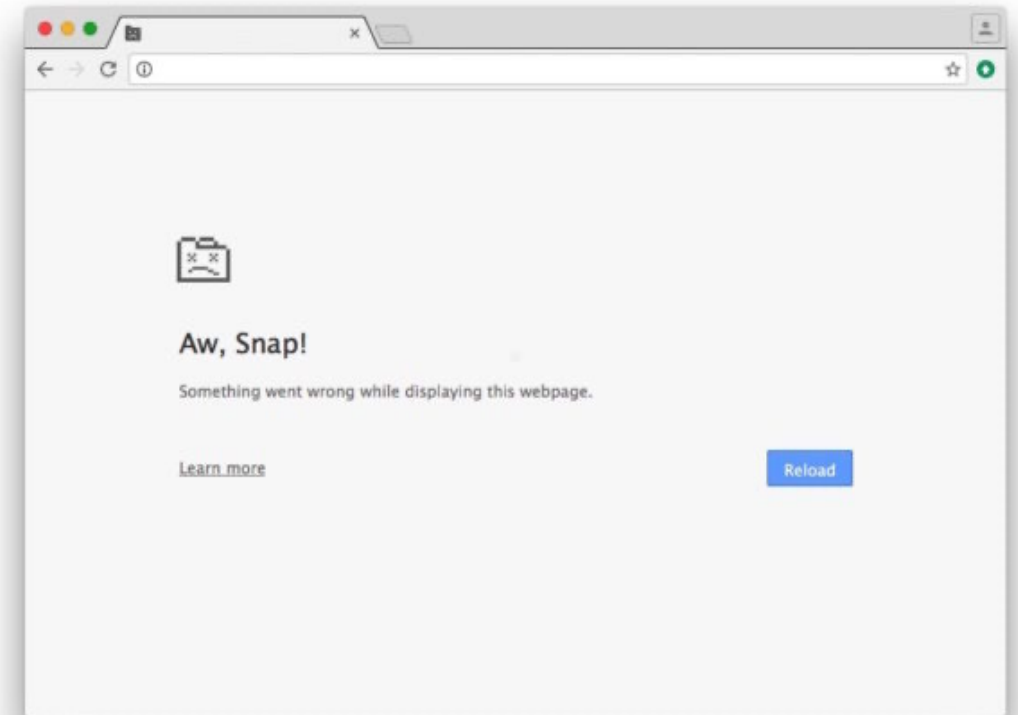
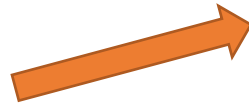
Since we never update the value of the "counter" variable, the script will enter an endless loop because the counter variable will always remain less than 10. This will eventually cause your browser to crash



```
let counter = 1;  
while(counter < 10){  
  console.log(counter);  
}
```



*** Note ***: This is bad code!!!



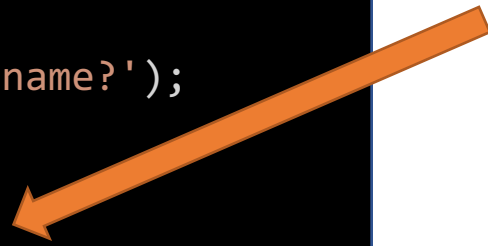
Types of JavaScript Loops

- While
 - Useful for when you are not sure how many times you need to run the loop
- Do While
 - Similar to a "while" loop, except the condition is always run at least once
- For
 - Useful for when you know exactly how many times you need to run the loop
- For Each
 - Only available on arrays and on some DOM collections
- For...in (not covered today, but look it up on MDN Docs)
 - Useful for iterating over an object (more on objects in a future class)
- For...of (not covered today, but look it up on MDN Docs)
 - Useful for iterating over arrays and other iterable objects

While Loops

- Useful when you are not sure how many times you want to run the loop

```
let user = prompt('What is your name?');  
  
while(user == null || user.trim() == ''){  
    prompt('You must enter a name to proceed.  
    Please enter a name.');
```



```
}  
  
out.innerHTML = `Hello ${user}.`;
```

Since we are not sure how many times our user will not enter a name we can use a while loop. The computer will keep repeating the loop code until the user enters a name into the prompt box

Do While Loops

- Similar to "while" loops, except that the conditional loop code will always execute at least once

```
let user;

do {
  // This code will run at least once,
  // even if the while condition is never
  // false
  user = prompt('What is your name?');
}while(user == null || user.trim() == '');

out.innerHTML = `Hello ${user}.`;
```

For Loop

- Useful when you know the number of times that you need to run the loop

```
const colours = ['red', 'orange', 'purple', 'green', 'blue'];  
  
for(i = 0, l = colours.length; i < l; i++){  
    out.innerHTML += `${colours[i]} <br>`;  
}
```

For Loop Syntax

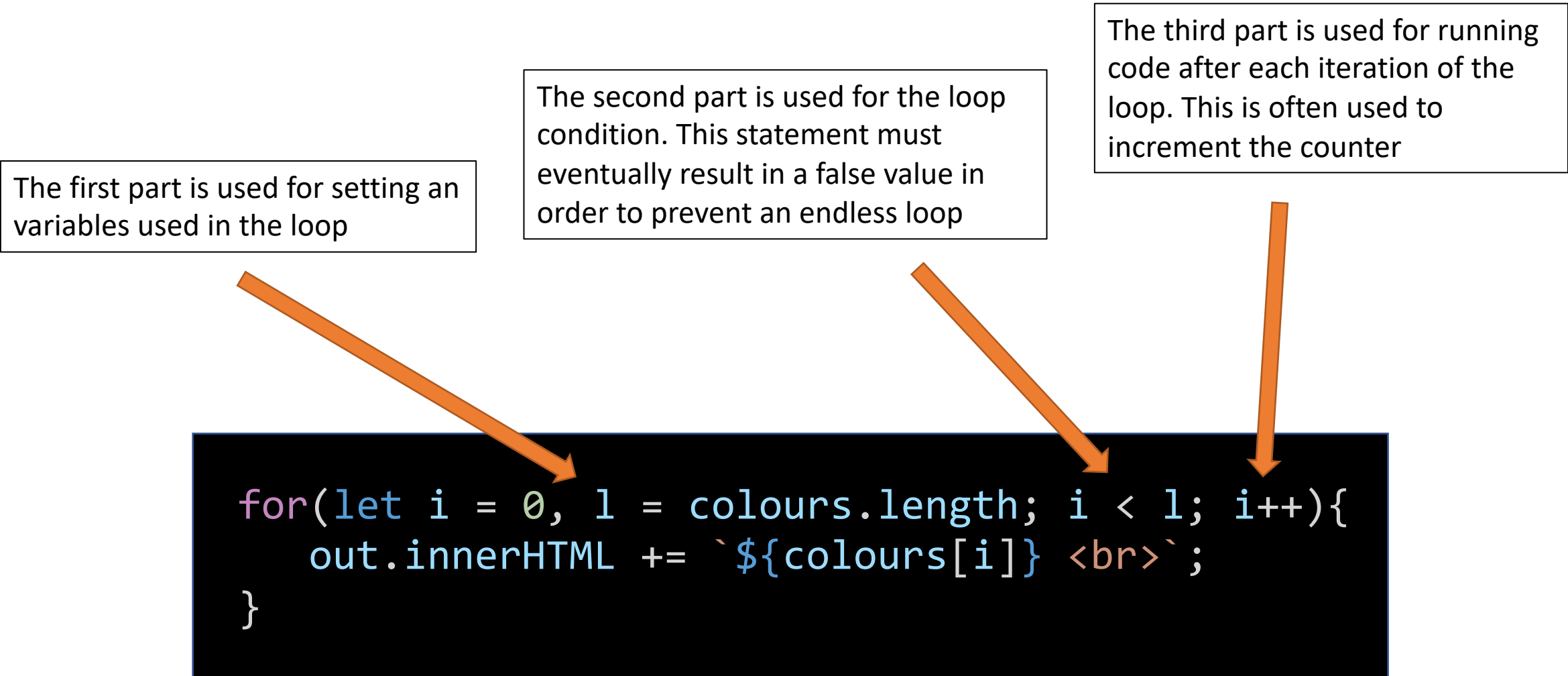
- The syntax between the "(...)" in a for loop can be a bit confusing
- The code between the "(...)" contains three parts separated by a ";"
 - The first part is used to setup any starting variables used in the loop
 - We often set our counter here (often set as "i" for index)
 - The second part is used as the condition for the loop
 - This condition must eventually result in a false value to prevent an endless loop
 - The third part is used for code that runs after each iteration of the loop
 - This is often used to increment the counter set up in the first part

For Loop Syntax

The first part is used for setting an variables used in the loop

The second part is used for the loop condition. This statement must eventually result in a false value in order to prevent an endless loop

The third part is used for running code after each iteration of the loop. This is often used to increment the counter



```
for(let i = 0, l = colours.length; i < l; i++){  
    out.innerHTML += `${colours[i]} <br>`;  
}
```

What Does "++" Mean?

- Programmers like to save keystrokes
- Adding "1" to an existing number is a common task that many scripts require
- Since adding "1" to a number is a common task many languages provide a shorthand syntax for doing this
- Writing "x++" is the same as writing "x = x + 1"

```
let x = 1;  
// Add 1 to x  
x++; // x -> 2
```

These two
code snippets
do essentially
the same thing

```
let x = 1;  
// Add 1 to x  
x = x + 1; // x -> 2
```

forEach() Loop

- The forEach() method provides an easy way to iterate over an array
- The forEach() method is only available on arrays and on the elements returned by the document.querySelectorAll() method
- forEach() works on
 - Arrays
 - DOM collections returned by:
 - document.querySelectorAll();

forEach() Loop with Arrays

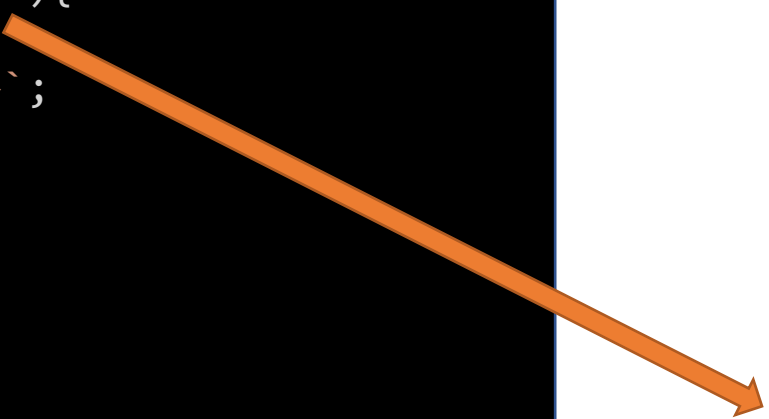
```
let html = '<ul>';

const colours = ['red', 'green', 'blue'];
// forEach() will run over each item
// in an array
colours.forEach(function(item){

    html += `<li>${item}</li>`;

});

html += '</ul>';
// html ->
// <ul>
//   <li>red</li>
//   <li>green</li>
//   <li>blue</li>
// </ul>
```



The "item" parameter will be set to the value of each item in the array as the loop iterates over each item in the array

What is the "+=" Operator

- Another common programming task is to add to an existing value of a variable
- The long hand way of doing this would be to write the following:

```
let sayHello = 'Hello';  
sayHello = sayHello + ' World'; // sayHello -> Hello World
```

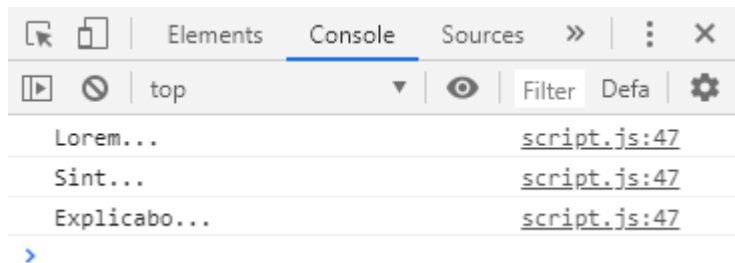
- The shorthand way uses the "+=" operator
 - `x += 'bar'` -> means adds the string "bar" to whatever value "x" is
 - See the syntax on the next page

forEach() Loops with querySelectorAll()

The HTML

```
<p class="special">Lorem...</p>
<p class="special">Sint...</p>
<p class="special">Explicabo...</p>
```

The Console



```
// This creates an HTML collection of all the HTML
// elements that would be selected by the CSS
// selector ".special"
const elSpecial =
document.querySelectorAll('.special');

// Use a forEach() loop to iterate over each HTML
// element
// in the the elSpecial HTML collection and output
// their HTML text
elSpecial.forEach(function(el){
  console.log( el.innerHTML );
});

// The above code will output the following to the
// console:
// Lorem...
// Sint...
// Explicabo...
```

What is the "+=" Operator

- Another common programming task is to add to an existing value of a variable
- The long hand way of doing this would be to write the following:

```
let sayHello = 'Hello';  
sayHello = sayHello + ' World'; // sayHello -> Hello World
```

- The shorthand way uses the "+=" operator
 - `x += 'bar'` -> means adds the string "bar" to whatever value "x" is
 - See the syntax on the next page

What is the "+=" Operator

Adding to an existing value using longhand syntax



```
let sayHello = 'Hello';  
sayHello = sayHello + ' World'; // sayHello -> Hello World
```

Adding to an existing value using the "+=" syntax



```
let sayHello = 'Hello';  
sayHello += ' World'; // sayHello -> Hello World
```



The "+=" operator adds a value to an existing value